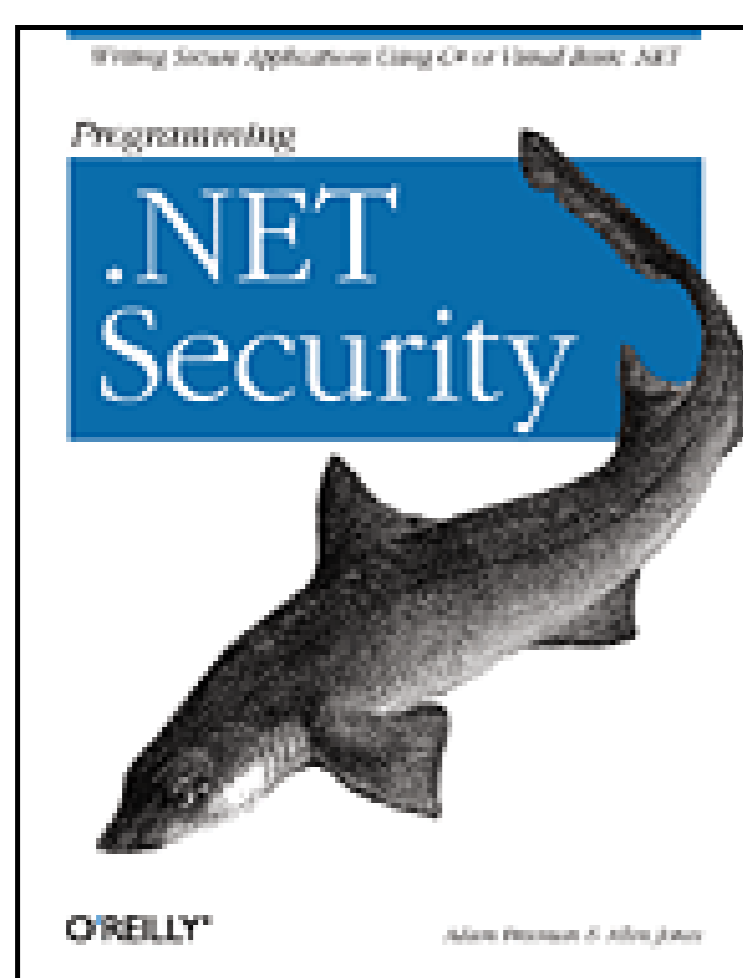


[\[Team LiB \]](#)



- [Table of Contents](#)
- [Index](#)
- [Reviews](#)
- [Reader Reviews](#)
- [Errata](#)

Programming .NET Security

By [Adam Freeman](#), [Allen Jones](#)

Publisher: O'Reilly

Pub Date: June 2003

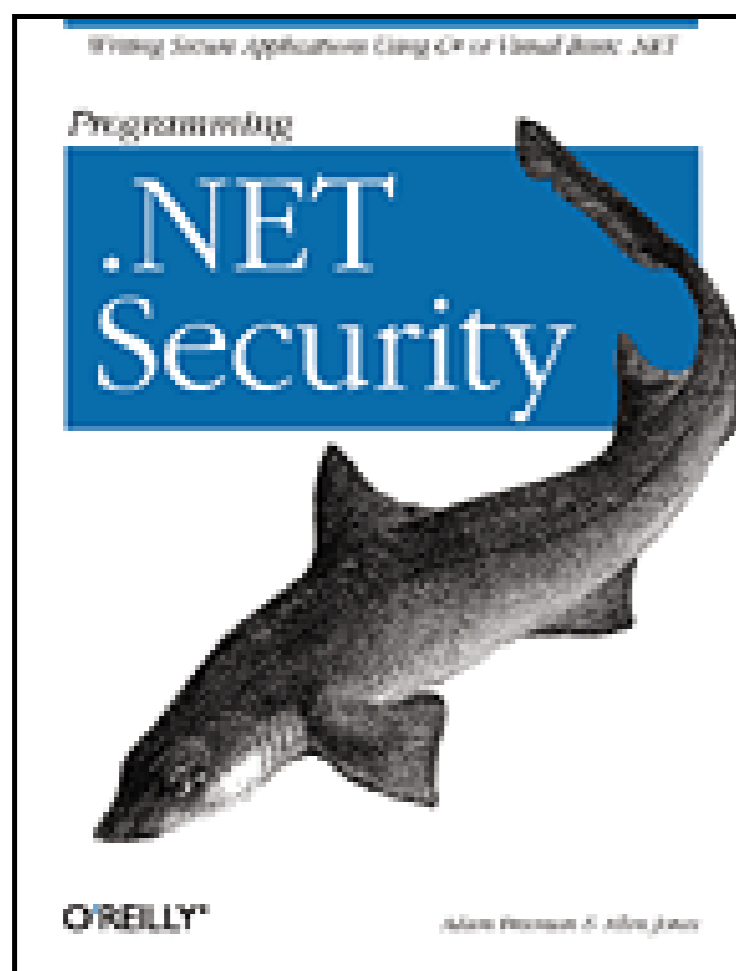
ISBN: 0-596-00442-7

Pages: 714

With the spread of web-enabled desktop clients and web-server based applications, developers can no longer afford to treat security as an afterthought. It's one topic, in fact, that .NET forces you to address, since Microsoft has placed security-related features at the core of the .NET Framework. Yet, because a developer's carelessness or lack of experience can still allow a program to be used in an unintended way, *Programming .NET Security* shows you how the various tools will help you write secure applications.

[\[Team LiB \]](#)

[[Team LiB](#)]



- [Table of Contents](#)
- [Index](#)
- [Reviews](#)
- [Reader Reviews](#)
- [Errata](#)

Programming .NET Security

By [Adam Freeman](#), [Allen Jones](#)

Publisher: O'Reilly

Pub Date: June 2003

ISBN: 0-596-00442-7

Pages: 714

[Copyright](#)

[Dedication](#)

[Preface](#)

[How This Book Is Organized](#)

[Who Should Read This Book](#)

[Assumptions This Book Makes](#)

[Conventions Used in This Book](#)

[How to Contact Us](#)

[Part I: Fundamentals](#)

[Chapter 1. Security Fundamentals](#)

[Section 1.1. The Need for Security](#)

[Section 1.2. Roles in Security](#)

[Section 1.3. Understanding Software Security](#)

[Section 1.4. End-to-End Security](#)

[Chapter 2. Assemblies](#)

[Section 2.1. Assemblies Explained](#)

[Section 2.2. Creating Assemblies](#)

[Section 2.3. Shared Assemblies](#)

[Section 2.4. Strong Names](#)

[Section 2.5. Publisher Certificates](#)

[Section 2.6. Decompiling Explained](#)

[Chapter 3. Application Domains](#)

[Section 3.1. Application Domains Explained](#)

[Chapter 4. The Lifetime of a Secure Application](#)

[Section 4.1. Designing a Secure .NET Application](#)

[Section 4.2. Developing a Secure .NET Application](#)

[Section 4.3. Security Testing a .NET Application](#)

[Section 4.4. Deploying a .NET Application](#)

[Section 4.5. Executing a .NET Application](#)

[Section 4.6. Monitoring a .NET Application](#)

[Part II: .NET Security](#)

[Chapter 5. Introduction to Runtime Security](#)

[Section 5.1. Runtime Security Explained](#)

[Section 5.2. Introducing Role-Based Security](#)

[Section 5.3. Introducing Code-Access Security](#)

[Section 5.4. Introducing Isolated Storage](#)

[Chapter 6. Evidence and Code Identity](#)

[Section 6.1. Evidence Explained](#)

[Section 6.2. Programming Evidence](#)

[Section 6.3. Extending the .NET Framework](#)

[Chapter 7. Permissions](#)

[Section 7.1. Permissions Explained](#)

[Section 7.2. Programming Code-Access Security](#)

[Section 7.3. Extending the .NET Framework](#)

[Chapter 8. Security Policy](#)

[Section 8.1. Security Policy Explained](#)

[Section 8.2. Programming Security Policy](#)

[Section 8.3. Extending the .NET Framework](#)

[Chapter 9. Administering Code-Access Security](#)

[Section 9.1. Default Security Policy](#)

[Section 9.2. Inspecting Declarative Security Statements](#)

[Section 9.3. Using the .NET Framework Configuration Tool](#)

[Section 9.4. Using the Code-Access Security Policy Tool](#)

[Chapter 10. Role-Based Security](#)

[Section 10.1. Role-Based Security Explained](#)

[Section 10.2. Programming Role-Based Security](#)

[Chapter 11. Isolated Storage](#)

[Section 11.1. Isolated Storage Explained](#)

[Section 11.2. Programming Isolated Storage](#)

[Section 11.3. Administering Isolated Storage](#)

[Part III: .NET Cryptography](#)

[Chapter 12. Introduction to Cryptography](#)

[Section 12.1. Cryptography Explained](#)

[Section 12.2. Cryptography Is Key Management](#)

[Section 12.3. Cryptographic Attacks](#)

[Chapter 13. Hashing Algorithms](#)

[Section 13.1. Hashing Algorithms Explained](#)

[Section 13.2. Programming Hashing Algorithms](#)

[Section 13.3. Keyed Hashing Algorithms Explained](#)

[Section 13.4. Programming Keyed Hashing Algorithms](#)

[Section 13.5. Extending the .NET Framework](#)

[Chapter 14. Symmetric Encryption](#)

[Section 14.1. Encryption Revisited](#)

[Section 14.2. Symmetric Encryption Explained](#)

[Section 14.3. Programming Symmetrical Encryption](#)

[Section 14.4. Extending the .NET Framework](#)

[Chapter 15. Asymmetric Encryption](#)

[Section 15.1. Asymmetric Encryption Explained](#)

[Section 15.2. Programming Asymmetrical Encryption](#)

[Section 15.3. Extending the .NET Framework](#)

[Chapter 16. Digital Signatures](#)

[Section 16.1. Digital Signatures Explained](#)

[Section 16.2. Programming Digital Signatures](#)

[Section 16.3. Programming XML Signatures](#)

[Section 16.4. Extending the .NET Framework](#)

[Chapter 17. Cryptographic Keys](#)

[Section 17.1. Cryptographic Keys Explained](#)

[Section 17.2. Programming Cryptographic Keys](#)

[Section 17.3. Extending the .NET Framework](#)

[Part IV: .NET Application Frameworks](#)

[Chapter 18. ASP.NET Application Security](#)

[Section 18.1. ASP.NET Security Explained](#)

[Section 18.2. Configuring the ASP.NET Worker Process Identity](#)

[Section 18.3. Authentication](#)

[Section 18.4. Authorization](#)

[Section 18.5. Impersonation](#)

[Section 18.6. ASP.NET and Code-Access Security](#)

[Chapter 19. COM+ Security](#)

[Section 19.1. COM+ Security Explained](#)

[Section 19.2. Programming COM+ Security](#)

[Section 19.3. Administering COM+ Security](#)

[Chapter 20. The Event Log Service](#)

[Section 20.1. The Event Log Service Explained](#)

[Section 20.2. Programming the Event Log Service](#)

[Part V: API Quick Reference](#)

[Chapter 21. How to Use This Quick Reference](#)

[Section 21.1. Finding a Quick-Reference Entry](#)

[Section 21.2. Reading a Quick-Reference Entry](#)

[Chapter 22. Converting from C# to VB Syntax](#)

[Section 22.1. General Considerations](#)

[Section 22.2. Classes](#)

[Section 22.3. Structures](#)

[Section 22.4. Interfaces](#)

[Section 22.5. Class, Structure, and Interface Members](#)

[Section 22.6. Delegates](#)

[Section 22.7. Enumerations](#)

[Chapter 23. The System.Security Namespace](#)

[AllowPartiallyTrustedCallersAttribute](#)

[CodeAccessPermission](#)

[IEvidenceFactory](#)

[IPermission](#)

[ISecurityEncodable](#)

[ISecurityPolicyEncodable](#)

[IStackWalk](#)

[NamedPermissionSet](#)

[PermissionSet](#)

[PolicyLevelType](#)

[SecurityElement](#)

[SecurityException](#)

[SecurityManager](#)

[SecurityZone](#)

[SuppressUnmanagedCodeSecurityAttribute](#)

[UnverifiableCodeAttribute](#)

[VerificationException](#)

[XmlSyntaxException](#)

[Chapter 24. The System.Security.Cryptography Namespace](#)

[AsymmetricAlgorithm](#)

[AsymmetricKeyExchangeDeformatter](#)

[AsymmetricKeyExchangeFormatter](#)

[AsymmetricSignatureDeformatter](#)

[AsymmetricSignatureFormatter](#)

[CipherMode](#)

[CryptoAPITransform](#)

[CryptoConfig](#)

[CryptographicException](#)

[CryptographicUnexpectedOperationException](#)

[CryptoStream](#)

[CryptoStreamMode](#)

[CspParameters](#)

[CspProviderFlags](#)

[DeriveBytes](#)

[DES](#)

[DESCryptoServiceProvider](#)

[DSA](#)

[DSACryptoServiceProvider](#)

[DSAParameters](#)

[DSASignatureDeformatter](#)

[DSASignatureFormatter](#)

[FromBase64Transform](#)

[FromBase64TransformMode](#)

[HashAlgorithm](#)

[HMACSHA1](#)

[ICryptoTransform](#)

[KeyedHashAlgorithm](#)

[KeySizes](#)

[MACTripleDES](#)

[MaskGenerationMethod](#)

[MD5](#)

[MD5CryptoServiceProvider](#)

[PaddingMode](#)

[PasswordDeriveBytes](#)

[PKCS1MaskGenerationMethod](#)

[RandomNumberGenerator](#)

[RC2](#)

[RC2CryptoServiceProvider](#)

[Rijndael](#)

[RijndaelManaged](#)

[RNGCryptoServiceProvider](#)

[RSA](#)

[RSACryptoServiceProvider](#)

[RSAOAEPKeyExchangeDeformatter](#)

[RSAOAEPKeyExchangeFormatter](#)

[RSAParameters](#)

[RSAPKCS1KeyExchangeDeformatter](#)

[RSAPKCS1KeyExchangeFormatter](#)

[RSAPKCS1SignatureDeformatter](#)

[RSAPKCS1SignatureFormatter](#)

[SHA1](#)

[SHA1CryptoServiceProvider](#)

[SHA1Managed](#)

[SHA256](#)

[SHA256Managed](#)

[SHA384](#)

[SHA384Managed](#)

[SHA512](#)

[SHA512Managed](#)

[SignatureDescription](#)

[SymmetricAlgorithm](#)

[ToBase64Transform](#)

[TripleDES](#)

[TripleDESCryptoServiceProvider](#)

[Chapter 25. The System.Security.Cryptography.X509Certificates Namespace](#)

[X509Certificate](#)

[X509CertificateCollection](#)

[X509CertificateCollection.X509CertificateEnumerator](#)

[Chapter 26. The System.Security.Cryptography.Xml Namespace](#)

[DataObject](#)

[DSAKeyValue](#)

[KeyInfo](#)

[KeyInfoClause](#)

[KeyInfoName](#)

[KeyInfoNode](#)

[KeyInfoRetrievalMethod](#)

[KeyInfoX509Data](#)

[Reference](#)

[RSAKeyValue](#)

[Signature](#)

[SignedInfo](#)

[SignedXml](#)

[Transform](#)

[TransformChain](#)

[XmlDsigBase64Transform](#)
[XmlDsigC14NTransform](#)
[XmlDsigC14NWithCommentsTransform](#)
[XmlDsigEnvelopedSignatureTransform](#)
[XmlDsigXPathTransform](#)
[XmlDsigXsltTransform](#)

Chapter 27. The System.Security.Permissions Namespace

[CodeAccessSecurityAttribute](#)
[EnvironmentPermission](#)
[EnvironmentPermissionAccess](#)
[EnvironmentPermissionAttribute](#)
[FileDialogPermission](#)
[FileDialogPermissionAccess](#)
[FileDialogPermissionAttribute](#)
[FileIOPermission](#)
[FileIOPermissionAccess](#)
[FileIOPermissionAttribute](#)
[IsolatedStorageContainment](#)
[IsolatedStorageFilePermission](#)
[IsolatedStorageFilePermissionAttribute](#)
[IsolatedStoragePermission](#)
[IsolatedStoragePermissionAttribute](#)
[IUnrestrictedPermission](#)
[PermissionSetAttribute](#)
[PermissionState](#)
[PrincipalPermission](#)
[PrincipalPermissionAttribute](#)
[PublisherIdentityPermission](#)
[PublisherIdentityPermissionAttribute](#)
[ReflectionPermission](#)
[ReflectionPermissionAttribute](#)
[ReflectionPermissionFlag](#)
[RegistryPermission](#)
[RegistryPermissionAccess](#)
[RegistryPermissionAttribute](#)
[ResourcePermissionBase](#)
[ResourcePermissionBaseEntry](#)
[SecurityAction](#)
[SecurityAttribute](#)
[SecurityPermission](#)
[SecurityPermissionAttribute](#)
[SecurityPermissionFlag](#)
[SiteIdentityPermission](#)
[SiteIdentityPermissionAttribute](#)
[StrongNameIdentityPermission](#)

[StrongNameIdentityPermissionAttribute](#)

[StrongNamePublicKeyBlob](#)

[UIPermission](#)

[UIPermissionAttribute](#)

[UIPermissionClipboard](#)

[UIPermissionWindow](#)

[UriIdentityPermission](#)

[UriIdentityPermissionAttribute](#)

[ZoneIdentityPermission](#)

[ZoneIdentityPermissionAttribute](#)

[Chapter 28. The System.Security.Policy Namespace](#)

[AllMembershipCondition](#)

[ApplicationDirectory](#)

[ApplicationDirectoryMembershipCondition](#)

[CodeGroup](#)

[Evidence](#)

[FileCodeGroup](#)

[FirstMatchCodeGroup](#)

[Hash](#)

[HashMembershipCondition](#)

[IIdentityPermissionFactory](#)

[IMembershipCondition](#)

[NetCodeGroup](#)

[PermissionRequestEvidence](#)

[PolicyException](#)

[PolicyLevel](#)

[PolicyStatement](#)

[PolicyStatementAttribute](#)

[Publisher](#)

[PublisherMembershipCondition](#)

[Site](#)

[SiteMembershipCondition](#)

[StrongName](#)

[StrongNameMembershipCondition](#)

[UnionCodeGroup](#)

[Uri](#)

[UriMembershipCondition](#)

[Zone](#)

[ZoneMembershipCondition](#)

[Chapter 29. The System.Security.Principal Namespace](#)

[GenericIdentity](#)

[GenericPrincipal](#)

[IIdentity](#)

[IPrincipal](#)

[PrincipalPolicy](#)

[WindowsAccountType](#)

[WindowsBuiltInRole](#)

[WindowsIdentity](#)

[WindowsImpersonationContext](#)

[WindowsPrincipal](#)

[Colophon](#)

[Index](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Copyright

Copyright © 2003 O'Reilly & Associates, Inc.

Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly & Associates books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safari.oreilly.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly & Associates, Inc. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly & Associates, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps. Microsoft, the .NET logo, Visual Basic .NET, Visual Studio .NET, ADO.NET, Windows, and Windows 2000 are registered trademarks of the Microsoft Corporation. The association between the Smoothhound shark and the topic of .NET security is a trademark of O'Reilly & Associates, Inc.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Dedication

Dedicated with love to Jacqui. She deserves a dedication to herself for a change.

-Adam Freeman

For my wife Elena and my daughter Anya. I love you both.

-Allen Jones

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Preface

The .NET Framework is a flexible general-purpose computing platform designed to address the needs of commercial organizations and individuals alike, and to support a variety of application models. .NET supports recent trends that favor highly distributed systems, component-based applications, and web-based server solutions (including XML web services). Although these trends have led to improvements in application functionality and programmer productivity, they also require that software consumers, producers, and service providers pay closer attention to software and system security.

Traditionally, programmers have treated security as an afterthought; however, there is a growing appreciation that security is a requirement, not an option, for contemporary applications and that security should be integrated into their development. The simple fact is that you cannot ignore security when developing a .NET application, because security is at the core of the .NET Framework and implicit in applications that you write. You must understand how to program .NET security, if for no other reason than to respond to the default actions of the .NET Framework, but more importantly, to create effective and functional .NET applications that will be welcomed by their users.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

How This Book Is Organized

This book is divided into five parts. [Part I](#) discusses fundamental software security concepts. [Part II](#) focuses on programming the security features of the .NET runtime. [Part III](#) covers the programmatic use of the cryptography classes contained in the .NET class library. [Part IV](#) discusses the use of security features more related to the platform on which your applications run. Finally, [Part V](#) contains an API reference covering the security classes we discuss in this book.

Part I: Fundamentals

[Chapter 1](#)

Introduces some fundamental software security concepts that you should understand before continuing to later chapters. Explains why there is a need for security, discusses the goal of software security, and introduces some important topics that you should be aware of as you begin to develop your own security programming skills.

[Chapter 2](#)

Provides an overview of the .NET assembly, which is a key component in .NET security. Describes the structure and contents of an assembly, demonstrates how to create the different types of assembly, and discusses protecting your assemblies from tampering and reverse engineering.

[Chapter 3](#)

Explains the role of application domains and discusses the effect they have on application isolation, security, and configuration.

[Chapter 4](#)

Discusses the way in which software security is integrated into the application lifetime, and provides practical advice that will help you understand the content contained in later chapters.

Part II: .NET Security

[Chapter 5](#)

Introduces the key security features provided by the .NET runtime. Explains the purpose and function of these security features, how they interact with each other, and their relationship with the security provided by the underlying operating system.

[Chapter 6](#)

Explains what evidence is and where it comes from, and describes the purpose and use of the different types of evidence, including the standard evidence classes provided with the .NET Framework. Demonstrates how to use evidence programmatically and how to extend the security capabilities of the .NET Framework by developing custom evidence classes.

[Chapter 7](#)

Explains what permissions are and discusses their role in the implementation of code-access security. Describes the mechanisms that the runtime uses to enforce code-level security, and explains how you can use permissions to manipulate these mechanisms. Finally, this chapter shows you how to extend code-access security by implementing your own custom permissions.

[Chapter 8](#)

Explains how the .NET runtime uses security policy to determine which permissions to grant to an assembly or application domain. Describes the structure of security policy and explains how the component elements interact at runtime. Explains how to manipulate security policy programmatically, and demonstrates the use of application domain policy.

[Chapter 9](#)

Outlines the default security policy implemented by the .NET Framework, and discusses the use of the .NET security tools to manage security policy.

[Chapter 10](#)

Explains what role-based security is and discusses the .NET Framework's implementation. Describes the classes used to access role-based security, and demonstrates how to use them in your programs.

[Chapter 11](#)

Describes what isolated storage is and explains where it provides benefits over existing data storage options. Demonstrates how to use isolated storage in your own programs, as well as how to administer and control access to it.

Part III: .NET Cryptography

[Chapter 12](#)

Provides an overview of the different aspects of cryptography and discusses some of the dangers and limitations of which you should be aware.

[Chapter 13](#)

Examines hash codes in more depth and shows you how to create and validate hash codes using the .NET Framework classes. Extends the .NET Framework by adding a new hashing algorithm.

[Chapter 14](#)

Discusses how to achieve confidentiality with symmetric data encryption, and how to encrypt and decrypt data using the .NET Framework. Shows you how to extend the .NET Framework by adding a new symmetric encryption algorithm.

[Chapter 15](#)

Discusses asymmetric encryption, explaining what it is, how it works, and how it overcomes the problem of exchanging secret keys. Demonstrates how to extend the .NET Framework by adding a new asymmetric encryption algorithm.

[Chapter 16](#)

Explains what digital signatures are, how they work, and how to use them within your .NET

applications. Demonstrates how to extend the .NET Framework by adding support for a custom digital signatures algorithm.

[Chapter 17](#)

Discusses the way in which the .NET Framework supports cryptographic keys. Explains the importance of keys, and the decisions you must make in deciding how you will create them.

Part IV: .NET Applications Frameworks

[Chapter 18](#)

Introduces the features you can use to increase the security of your ASP.NET applications. Discusses the overall issue of ASP.NET application security, and the mechanisms the .NET Framework implements to provide authentication, authorization, and impersonation services to ASP.NET applications.

[Chapter 19](#)

Discusses the COM+ security services, and explains how you can apply them to your COM+ components.

[Chapter 20](#)

Describes how to use the Windows Event Logging Service from your .NET applications in order to audit Windows security events.

Part V: API Quick Reference

Part V contains a comprehensive API reference covering the following security-related namespaces of the .NET Framework base class library:

```
System.Security  
System.Security.Cryptography  
System.Security.Cryptography.X509Certificates  
System.Security.Cryptography.Xml  
System.Security.Permissions  
System.Security.Policy  
System.Security.Principal
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Who Should Read This Book

Programming .NET Security was written for two groups of people. First, we wrote for the architects and designers of .NET applications who must understand the capabilities and limitations of .NET security to factor them into their designs and plans. Each chapter in Parts II, III, and IV begins with a detailed discussion that describes the technology but does not go to the level of discussing individual classes and methods.

Second, we wrote for all C# and Visual Basic .NET programmers who want to know how to use the features of the .NET Framework to write more secure applications. Following a technical introduction, each chapter in Parts II, III, and IV contains a detailed explanation of how to program the features of the .NET Framework being discussed; numerous code samples accompany these sections to clarify the points being made. Specifically for advanced programmers, our discussions of code-access security and cryptography contain fully worked examples on how to extend the .NET Framework security mechanisms.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Assumptions This Book Makes

This book is focused on .NET Framework security programming; we assume no prior exposure to the .NET security classes, but expect the reader to have basic experience as a competent C# or Visual Basic .NET programmer.

In Part IV of this book, we discuss the interaction of .NET application security with features and security capabilities of the Windows platform and other external services. In these chapters, we expect the reader to be familiar with the external technologies, and focus only on the aspect of security programming.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Conventions Used in This Book

The following font conventions are used throughout this book:

Italic is used for:

- Pathnames, filenames, and program names
- Internet addresses, such as domain names and URLs
- New terms where they are defined

Constant Width is used for:

- Command lines and options that should be typed verbatim
- Names and keywords in program examples, including method names, variable names, and class names

Constant Width Bold is used for:

- Emphasis within program code

 This icon designates a note, which is an important aside to the nearby text.

 This icon designates a warning relating to the nearby text.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

How to Contact Us

We have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions for future editions, by writing to:

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the U.S. or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

There is a web page for the book that lists errata, examples, or any additional information. You can access this page at:

<http://www.oreilly.com/catalog/prognetsec>

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about books, conferences, Resource Centers, and the O'Reilly Network, see the O'Reilly web site:

<http://www.oreilly.com>

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Part I: Fundamentals

Discusses the need for security and the approaches to adopt when developing secure software. These chapters also discuss assemblies and application domains-two fundamental building blocks of .NET applications that play a crucial role in the creation of secure software:

[Chapter 1](#)

[Chapter 2](#)

[Chapter 3](#)

[Chapter 4](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 1. Security Fundamentals

This chapter introduces the fundamental software security concepts that you need to understand before continuing to later chapters. We explain why there is a need for security, and the roles that are important to the development and operation of software security policies. We also discuss the goals of software security and introduce some important concepts that you should understand as you develop your own security programming skills.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

1.1 The Need for Security

Only a few years ago, software applications tended to be isolated. Users of these applications were required to present themselves in a known location (for example, a bank branch or office block) that was protected by physical barriers to access such as locks, surveillance cameras, and security guards. Attacks against such software systems were fewer than are experienced today, in part, because gaining access to such a location presented a barrier that many found insurmountable.

The increased connectivity and prevalence of networked applications has removed the insurmountable barrier presented by physical security, and it is not only the networked applications themselves at risk. Increasingly, software systems control access to valuable physical resources (for example, banking software can be used to credit or debit a customer account). Subverting or compromising the software system may be the simplest way to gain access to the physical resource; for example, it may be easier to break into the banking application and create fictitious transactions than it is to crack open the bank vault.

Today, a talented 15-year-old Italian schoolboy, who would be unable to get past a company security guard, might, for personal amusement, be able to convince a networked application that he's a 37-year-old trusted employee from Alabama. More serious, however, is the increase in software hacking for criminal reasons—either to steal intellectual property or, more commonly, to steal information that can be sold to other criminals, such as lists of credit card numbers.

In short, the world has become more hostile towards software. In light of recent changes to social and political attitudes to security, it should be no surprise that the public has an increased expectation that software will be secure. The kinds of security that we discuss in this book can provide some protection against the increased frequency and sophistication of attempts to subvert applications. However, security has also become a tool to promote the sale of software, and claims of "unbreakable" security are now commonplace. The effective use of software security has fallen behind the ideal that is portrayed by marketing departments. Another purpose of this book is to close the gap between the perception and the reality, and to demonstrate how you can increase the security of your applications through the careful application of tried-and-tested technologies.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

1.2 Roles in Security

In a normal software development project, there are many people who influence software security. People often look at software security from different perspectives and hope to gain different results from its implementation; these are often at odds with the goals of others. In this section, we describe the most common roles, and explain the motivations and goals of those who hold them. The content of this book is aimed at the technical reader, but it is important that you appreciate the complete set of influences that shape the need for and implementation of software security. See [Chapter 4](#) for a more detailed examination of some of these roles and the way they influence the life cycle of an application.

1.2.1 The Business Sponsor

The *business sponsor* is responsible for commissioning a software development project, and usually owns the problem that the application is intended to solve. The role of the business sponsor, and his expectations of software security, varies depending on the nature of the business and the purpose of the software.

The business sponsor typically lacks technical expertise, but controls the development budget and may support the implementation of software security for the following reasons:

- Security is a known requirement of the systems users.
- Legislation dictates that the software must implement certain security measures.
- Security features are necessary to compete with other products and look good on marketing material.

Lacking formal requirements, the business sponsor will often have opinions to offer on the importance and implementation of software security. These opinions may or may not be in line with the real requirements of the project.

Business sponsors are often the biggest source of tension on a project when it comes to the correct application of security. As you will see throughout this book, software security can be applied only after a careful assessment of the application requirements; however, the business sponsor often wants to bring the application into production as quickly as possible, and this creates a tension between the careful application of a planned security policy and the business requirement that the application ship quickly.

1.2.2 The Architect

The project *architect* is responsible for the overall design of the application, ensuring that the planned development will meet the business and technical goals that have been specified by the business

sponsor. The architect is ideally placed to assess the security needs of the application and to formulate the security policy that will be implemented by the programmers.

1.2.3 The Programmer

The *programmer* is responsible for implementing the application design produced by the architect and for meeting the software security goals specified in the design. The programmer must have a firm understanding of the security features provided by the development platform in use, and must be trusted to implement the security policy completely and without modification.

1.2.4 The Security Tester

The *security tester* does not perform the same role as an ordinary application tester. A normal tester creates test scenarios that ensure that the planned functionality works as expected, by simulating the actions of a user. By contrast, the security tester simulates the actions of a hacker, in order to uncover behaviors that would circumvent the software security measures. Security testing is an underrated and underemployed activity, but is vital in validating the security measures designed by the architect and implemented by the programmer.

1.2.5 The System Administrator

The *system administrator* is responsible for installing, configuring, and managing the application; these tasks require a good understanding of general security issues, and an appreciation of the security features provided by the development platform and the application itself.

One of the most important aspects of system administration is application monitoring. Well-designed applications provide system administrators with information about potential security breaches, and it is the responsibility of the system administrator to monitor for such information and to formulate a response plan in the event that the security of an application is subverted.

1.2.6 The User

The *user* is the final consumer of the functionality provided by the application, and is often required to interact with its software security measures—for example, by entering a username and password to gain access to its functionality.

The users of an application create their own tensions against the security policy; their expectations that the software system will protect them are high, but their willingness to be constrained by intrusive security measures is limited. For example, retail customers expect that a software system will conceal their credit card numbers from unauthorized third parties and protect their accounts from unauthorized changes. However, the same users will resist taking any responsibility for their own security—for example, by remembering and specifying a PIN code when they purchase goods.

Successful security policies take into account the users' attitudes, and do not force them to accept security demands that they cannot or will not adhere to. Unsuccessful security policies do not take

into account the needs of the user—for example, requiring users to remember long and difficult passwords that are frequently changed. In such circumstances, users will simply write the password down on a piece of paper and thereby negate all of the effort made during the development process.

1.2.7 The Hacker/Cracker

The final role is the *cracker*, more popularly known as a *hacker*. The hacker attempts to circumvent or subvert software security for financial gain or perhaps to rise to a perceived intellectual challenge. The hacker is the person whom security measures are meant to foil, but the label does not accurately describe the range of people who will attack software security systems. Throughout this book, we detail a number of specific security systems and explain the type of attack against which each is intended to protect.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

1.3 Understanding Software Security

There are two kinds of assets that software security sets out to protect; restricted resources and secrets. In this section, we provide an overview of these important categories, which we build on throughout the rest of the book.

1.3.1 Restricted Resources

A *restricted resource* is any object, feature, or function of your application (or of any software or hardware that your application depends on) that you do not wish to be used or accessed by unauthorized people. This is a very broad definition, but casting our net this wide allows us to demonstrate the common solution to a wide range of closely related issues; the following list describes some restricted resources that you may encounter:

Disk files

The most commonly encountered restricted resource is the disk file. For example, by default, the Windows operating system allows users to access their own files, but not the files of other users or disk files used by the operating system itself. Users are *restricted* from accessing files that they have not created.

Software functions

One of the most familiar restricted resources for you is the software function that should not be available to all users of the application or service. For example, the accounts clerk in a bank should not be able to authorize mortgage loans; such activities are *restricted* to qualified loan officers.

Hardware resources

Software security is often used to restrict access to important hardware resources, such as a high-quality color printer. Ordinary users are *restricted* to printing their documents in monochrome, while the sales staff prints customer presentations in color.

External services

Increasingly, software security restrictions are applied to external services that have no tangible physical attributes, but affect a company either by incurring a direct financial cost or by distracting staff from their duties. Examples of such services are Internet access, personal emails, and international telephone calls. Access to these services is *restricted* to those who need them to do their jobs

1.3.2 Trust

In terms of software security, when we *trust* someone, we grant that person access to one or more restricted resources; for example, we trust our bank loan officer to approve mortgage loans in a responsible way and grant the officer access to the software features for loan approval.

The first step in managing trust is to establish *identity*, which is the means that a system uses to uniquely differentiate between users. The complexity of the identities used by an application is influenced by the number of users that need identification; if there are small numbers of users, then identities as simple as "Alice" and "Bob" may be sufficient to uniquely identify each person. By contrast, providing unique identification for thousands or millions of people may require more complex identities—for example, social security numbers.

You use *authentication* to ensure that you have correctly established the identity of a user. If you wish to restrict access to a valuable resource, you cannot take a statement of identity by a user at face value. Consider the ramifications if your bank believed a claim by you that you were "Bill Gates" and granted you access to his deposits. The most common form of authentication requires a user to provide a username and a password to access an application; the username represents the stated user identity and the password is a secret known only to the user (see the next section for information about secrets). We expect that others who might wish to assume this identity will not know the secret password and therefore won't pass the authentication process.

Once you have authenticated and identified a user, you *authorize* the user to access one or more restricted resources. The resources to which a user is granted access depend on the level of trust granted by the application; the level of trust is typically determined by the nature of the tasks that a user will undertake; for example, you might grant a user access to a disk file containing details of a new product because that user is employed as a product development engineer, and the contents of that file are required by the user to discharge her duties.

The process of establishing and authenticating an identity, and authorizing access to resources based on that identity is illustrated in [Figure 1-1](#). This process is at the heart of software security, and many applications and systems that may appear to work in radically different ways implement this common approach.

Figure 1-1. The process of granting access to restricted resources

See [Chapter 5](#) for an introduction to the way in which .NET defines and supports restricted resources and [Chapter 6](#) through [Chapter 10](#) for detailed coverage on how to apply .NET security to your application projects.

1.3.3 Secrets

A *secret* is any data created or processed by your application that you do not wish to be publicly known. Examples of secrets include the credit card numbers of your clients, and passwords your users enter to authenticate their identities for your application. Secrets are the counterpart to restricted resources. While a restricted resource often represents the ability to perform an action (such as approving a mortgage or printing a color document), a secret typically embodies the data the resource will process (such as the financial details of a customer or the secret marketing plan).

1.3.3.1 The lifetime of secrets

It is important to consider each type of secret that you work with and assess how long it needs to remain a secret; for example, a customer credit card number needs to be protected only until the card expires, after which, the information you hold on file cannot be used to purchase goods. By contrast, some secrets must be protected forever, such as medical histories.

One mistake that is frequently made is considering the lifetime of a secret in isolation from the real world. For example, you may choose to protect a user's secret password only until it is changed, perhaps as part of a process where a password is valid for a fixed period. The problem that this presents is that users will often change a password back to its original value as soon as they can, which means that a hacker could access your application if your list of expired passwords were allowed to become public. When you assess the lifetime of a secret, consider that the data itself may have value if it persists outside the application.

1.3.3.2 Protecting secrets

Secrets are typically protected with *cryptology*, which is the subject of [Part III](#). Cryptology uses complex mathematical algorithms to encode secrets, and the type of algorithm used depends on the length of time the data needs protection, which is, in turn, influenced by the lifetime of the secret. See [Chapter 12](#) for an introduction to cryptology, and [Chapter 13](#) through [Chapter 17](#) for in-depth coverage of how the .NET Framework supports cryptographic techniques.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

1.4 End-to-End Security

The final concept we introduce in this chapter is *end-to-end security*, which is the result of considering the wider aspects of security, beyond the resources and secrets protected by your application. Although this book is focused on implementing software security, it is important that you take a higher-level view, taking into account the real world and its complexity. The following sections highlight other issues that you should consider.

1.4.1 Real-World Trust Relationships

One of the most important things to remember about security is that not everyone shares your motivations and aspirations, and not everyone thinks the way that you think. The most carefully defined software trust system may not reflect the actual trust that has been granted to users of an application.

As a simple example, when we outlined the important roles that play in security, we differentiated between the legitimate users of an application and the hackers who want to subvert it. The reality is less clear-cut; most fraud is perpetrated by employees otherwise trusted by their company; when you grant trust to a user, you may be providing that individual tools that will be used to defraud, and otherwise rob, your company.

Differences in motivation are especially relevant when considering coercion; although you may feel that the security and profitability of your application is paramount, others may not. In some countries, it is common to rob a bank by kidnapping the branch manager's children, coercing the manager to unlock the bank and provide access to the vaults in order to secure the freedom of his offspring. The robbers have secured access to the protected resources with the explicit cooperation of a trusted employee. When thinking about security, you must give consideration to motivation, both of the potential users and the potential attackers.

1.4.2 Side Channels

A *side-channel attack* employs methods that have little to do with the software security measures that protect an application. For example, looking over a person's shoulder while he types his password circumvents any policy that may be put in place to control identity authentication; the attacker has side-stepped the security measures and can now access the system.

Identifying potential side-channel attacks takes a lot of lateral thinking, and requires careful evaluation of the information exposed by your security systems that attackers might be able to exploit. Various smart-card technologies, for example, have been compromised by analyzing variations in power consumption, electromagnetic radiation, and the amount of time taken to process an authentication request associated with their use. Such an attack takes time and determination, but can be profitable if the smart card is used to manage cash transactions or provides access to expensive resources.

1.4.3 Physical Security

Physical security is often ignored when application software is secured. Many years of effort may go into designing an application that carefully controls access to the contents of a warehouse, but if the security of the warehouse itself is weak, then attackers will simply steal your goods directly rather than attempting to subvert your application to ship goods free-of-charge.

A good example of physical security is represented by automated teller machines (ATMs). An ATM employs physical measures to protect against tampering and outright theft, as well as the software measures that authenticate your card and PIN number.

1.4.4 Third-Party Software

Finally, we draw your attention to third-party software, which you may rely on for the development of your application. Examples of this kind of software include development environments (such as Microsoft Visual Studio .NET), class libraries, and language compilers.

It is important for you to understand that by using third-party software, you are trusting the software publisher to produce software that does not present a security risk, either in the functionality that is provided or by the introduction of malicious features. You should also realize that when you distribute software that depends on third-party software, you are asking your customers to grant the same level of trust to the third parties as they do to you and your organization.

For example, you may trust Microsoft to produce safe and secure software, and you may feel confident that Internet Explorer provides a secure environment in which to browse the World Wide Web. However, if you look at the information provided by the "About Internet Explorer" window, you will see that Internet Explorer 6 is built, in part, with software licenses from the following organizations:

- National Center for Supercomputing Applications at the University of Illinois at Champaign
- Spyglass, Inc.
- RSA Data Security, Inc.
- The Independent JPEG Group
- Intel Corp.
- Mainsoft Corp.

When you trust Microsoft to deliver Internet Explorer without the inclusion of security-related defects or malicious code, you also implicitly trust all of the companies listed above, some that you may not even have heard of. Each of these companies may in turn license content or functionality from other publishers, and the chain continues; the trust that you confer on an individual software publisher goes far beyond what you may expect, and this is equally true for development software and class libraries as it is for Windows applications.

When selecting third-party tools and libraries for your development projects, you must consider the

level of trust that you assign to the software publisher *and* any other companies or individuals that may have contributed to these products.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 2. Assemblies

This chapter provides an overview of the .NET *assembly*, which is a key component in .NET security. We describe the structure and contents of an assembly, demonstrate how to create the different types of assembly, and discuss how you can protect your assemblies from tampering and reverse engineering. The information in this chapter provides a foundation essential for understanding the more advanced topics presented in later chapters, especially those related to Code Access Security.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

2.1 Assemblies Explained

An assembly contains one or more .NET data types compiled into the Microsoft Intermediate Language (MSIL); in other words, an assembly contains the code that the common language runtime executes. The .NET Framework uses the assembly as the basic unit of deployment and versioning and, most importantly for our purposes, as the basic security boundary. In the following chapters, we will demonstrate how the .NET Framework uses assemblies to enforce security policy; in this section, however, we provide a brief overview of assemblies as a foundation for more advanced topics.

Microsoft Intermediate Language

When you compile a .NET source file, the compiler converts contained data types to the MSIL, which consists of a series of CPU-independent instructions. The .NET Framework compiles the MSIL describing your data type into instructions for a specific CPU at execution; this approach allows .NET applications to be deployed to different CPU architectures without having to recompile the source code. Consult the .NET documentation for further information about MSIL.

Each assembly consists of the following:

Assembly manifest

The assembly manifest contains metadata elements that describe the assembly, summarized as follows:

Assembly name

A text string specifying the assembly's name

Assembly version

A string in the form *a.b.c.d*, where *a* and *b* are major and minor version numbers, and *c* and *d* are revision and build numbers

Assembly culture

Information on the culture or language that the assembly supports, for example, "en-US" for U.S. English or "fr-CA" for Canadian French

Assembly strong name

Details of the public key used to create the assembly strong name; see [Section 2.4](#) for further information

A list of all of the files in the assembly

Lists all of the files that comprise the assembly; as we explain later, assemblies can be made up of more than one file

Type reference information

Provides information about the data types that are exported from this assembly for use in other assemblies

Referenced assemblies information

Lists the details of any other assemblies that this assembly depends on

Informational attributes

A set of human-readable strings that provide additional information about an assembly, including trademark and copyright restrictions, product and company names, and a description of the assembly contents

Type metadata

This part of the assembly contains information about the data types contained in the MSIL code portion of the assembly, described later.

MSIL code

The MSIL code section of the assembly contains the MSIL statements that will be executed by the .NET Framework runtime, describing one or more .NET types.

Application resources

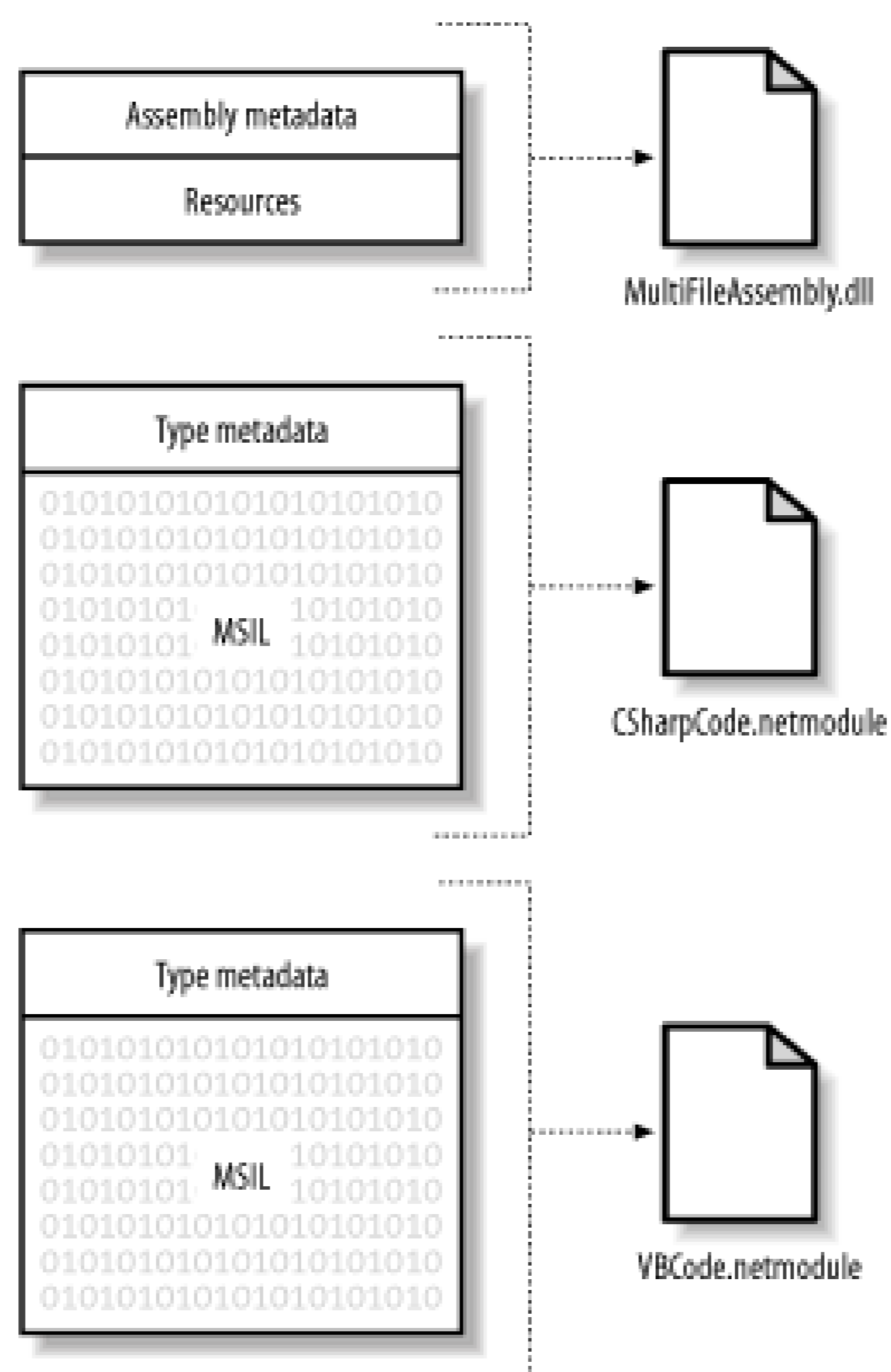
There are a set of resources, such as icons and sounds, that the application uses.

There are two types of assembly; each type differs in where the assembly contents are stored. The simplest and most common type is a *single-file assembly*, where all of the assembly contents are stored in a single file. As shown by [Figure 2-1](#), the four assembly components are contained in the file named *SingleFileAssembly.dll*.

Figure 2-1. A single-file assembly contains all of the assembly contents in a single disk file

The second assembly type is a *multifile assembly*, where the contents of the assembly are stored in more than one file, as illustrated by [Figure 2-2](#).

Figure 2-2. A multifile assembly stores the assembly contents in more than one file



In a multifile assembly, the assembly metadata and any resources are stored in one file (*MultiFileAssembly.dll* in [Figure 2-1](#)), while the MSIL code is contained in one or more *modules*, each of which contains one or more data types and any associated type metadata. The assembly is made up of *all three files*, which must be deployed together; there is no association between the files enforced by the filesystem—the relationship between the files is described by the file list element of the assembly metadata.

The advantage of a multifile assembly is that each module can contain types written in a different .NET language; in [Figure 2-2](#), we illustrate this by including the *CSharpCode.netmodule* and *VBCode.netmodule* files in the assembly, representing data types written in C# and Visual Basic .NET, respectively. In the next section, we demonstrate how to create an assembly using modules written in different languages.

Visual Studio .NET does not include support for creating modules or multifile assemblies; you must use the command-line tools included with the .NET Framework SDK to create multifile assemblies.

Each assembly can have one of three "flavors," depending on the intended function of the assembly, summarized as follows:

Library assembly

A library assembly contains .NET types that are consumed by other assemblies. A library assembly cannot be executed.

Executable assembly

An executable assembly can be executed as an application, but the types defined in the assembly cannot be used by other assemblies.

Windows executable assembly

A Windows executable assembly is a variation of an executable assembly that is specifically for GUI applications.

This brief overview of assemblies should provide you with enough information to understand the rest of this chapter and the content presented in the rest of this book. [Chapter 4](#) provides more information about the role of assemblies in the life cycle of a .NET application.

[\[Team LiB \]](#)

[Team LiB]

2.2 Creating Assemblies

To transform source code files into assemblies, you must use a language-specific .NET compiler. For C# source code files, we use the C# compiler (*csc.exe*) and we use the Visual Basic .NET compiler (*vbc.exe*) for Visual Basic .NET source files. In this section, we will demonstrate how to create single- and multifile assemblies; we do not go into detail about how to use the .NET compilers. We begin by defining two simple types, the first of which is as follows:

C#

```
public class SumNumbers {
    private int o_total;

    /// <summary>
    /// Default constructor - initializes the total to zero
    /// </summary>
    public SumNumbers( ) {
        // initialize the total
        o_total = 0;
    }

    /// <summary>
    /// Add a number to the total
    /// </summary>
    /// <param name="p_number">The number to add</param>
    public void AddNumber(int p_number) {
        o_total += p_number;
    }

    /// <summary>
    /// Get the total
    /// </summary>
    /// <returns>The total of all values presented to AddNumber</returns>
    public int GetTotal( ) {
        return o_total;
    }
}
```

Visual Basic .NET

```
Public Class SumNumbers
    Private o_total As Integer

    Public Sub New( )
        ' initialize the total
        o_total = 0
    End Sub
```

```

Public Sub AddNumber(ByVal p_number As Integer)
    o_total += p_number
End Sub

```

```

Public Function GetTotal( ) As Integer
    Return o_total
End Function

```

```
End Class
```

The `SumNumbers` class maintains a running total of integer values using the `AddNumber` method; the total value is obtained using the `GetTotal` method. The second type, `SumArray`, defines the static member `SumArrayOfIntegers`, which accepts an array of integers to be added together; this class is a consumer of `SumNumbers`.

```
# C#
```

```

public class SumArray {

    /// <summary>
    /// Static method that sums together the values in
    /// an array of integers
    /// </summary>
    /// <param name="p_arr"></param>
    /// <returns></returns>
    public static int SumArrayOfIntegers(int[] p_arr) {
        // create a new instance of SumNumbers
        SumNumbers x_sum = new SumNumbers( );
        // add each value in the array to the sum
        foreach (int x_int in p_arr) {
            x_sum.AddNumber(x_int);
        }
        // return the total from the sum
        return x_sum.GetTotal( );
    }
}

```

```
# Visual Basic .NET
```

```
Public Class SumArray
```

```

Public Shared Function SumArrayOfIntegers(ByVal p_arr( ) As Integer) _
As Integer

```

```
    ' create a new instance of the SumNumbers class
```

```
    Dim x_sum As SumNumbers = New SumNumbers
```

```
    ' add each value in the array to the sum
```

```
    Dim x_int As Integer
```

```
    For Each x_int In p_arr
```

```
        x_sum.AddNumber(x_int)
```

```
    Next
```

```
    ' return the total from the sum
```

```
    Return x_sum.GetTotal( )
```

```
End Function
```

```
End Class
```

We will save the `SumNumbers` type in a file named *SumNumbers.cs* (C#) / *SumNumbers.vb* (Visual Basic .NET) and the `SumArray` type in a file named *SumArray.cs* (C#) / *SumArray.vb* (Visual Basic .NET).

2.2.1 Creating a Single-File Assembly

This is the default assembly type created by the C# and Visual Basic .NET compilers. The assembly metadata and the MSIL statements are included in a single file. The following statements demonstrate how we create a single-file library assembly from our source files:

```
# C#

csc /out:SingleFileAssembly.dll /target:library SumNumbers.cs SumArray.cs

# Visual Basic .NET

vbc /out:SingleFileAssembly.dll /target:library SumNumbers.vb SumArray.vb
```

These statements create a single-file assembly named *SingleFileAssembly.dll*, which contains the compiled `SumNumbers` and `SumArray` types; the `/out` argument allows you to specify the name of the created file, while the `/target` option allows you to choose between application and library assemblies; consult the .NET documentation for full details of the compiler options.

2.2.2 Creating a Multifile Assembly

Creating a multifile assembly is more complicated than creating a single-file assembly but does provide additional flexibility; in this section, we demonstrate how to create an assembly that contains modules written in different .NET languages. We start by compiling our C# implementation of the `SumNumber` type into its own module:

```
csc /out:SumNumbers.netmodule /target:module SumNumbers.cs
```

This command creates a new module called *SumNumbers.netmodule*. The "netmodule" suffix is the standard for module files, in the same way that "dll" is for library files. We will now compile our Visual Basic .NET implementation of the `SumArray` type into a module. The `SumArray` class depends on the functionality of the `SumNumbers` class, and we reference the *SumNumbers.netmodule* file with the `/addmodule` argument:

```
vbc /out:SumArray.netmodule /target:module /addmodule:SumNumbers.netmodule \
    SumArray.vb
```

We have now created two modules- *SumNumbers.netmodule* is created from a C# class, while *SumArray.netmodule* is created from a Visual Basic .NET class. .NET modules can contain more than one type, although each type must be defined in the same language-we have created modules containing only one type in order to present a simple and clear example.

We can now create a multifile assembly that contains our modules by using the Assembly Linker tool (*al.exe*). The following command creates a library assembly called *MultiFileAssembly.dll*:

```
al /out:MultiFileAssembly.dll /target:library SumNumbers.netmodule SumArray.netmodule
```


The Assembly Linker creates the DLL file, which contains only the assembly metadata and references to the module files, each of which includes one of our example classes.



A multifile assembly consists of the metadata file (the *.dll* or *.exe*) *and* all of the modules you specified to the Assembly Linker. When distributing a multifile assembly, you must include all of the files with a *netmodule* suffix as well as the metadata file.

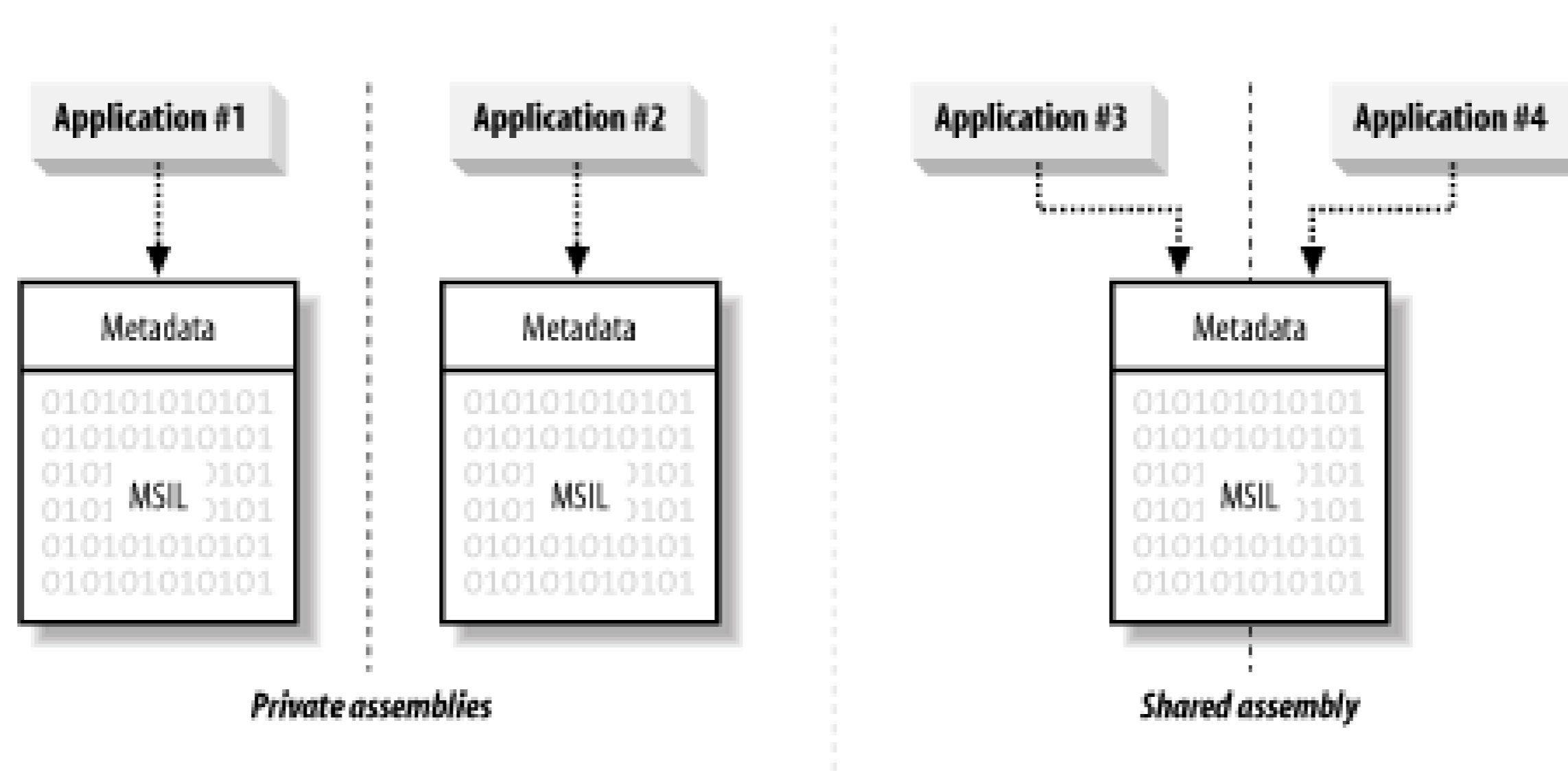
[Team LiB]

[[Team LiB](#)]

2.3 Shared Assemblies

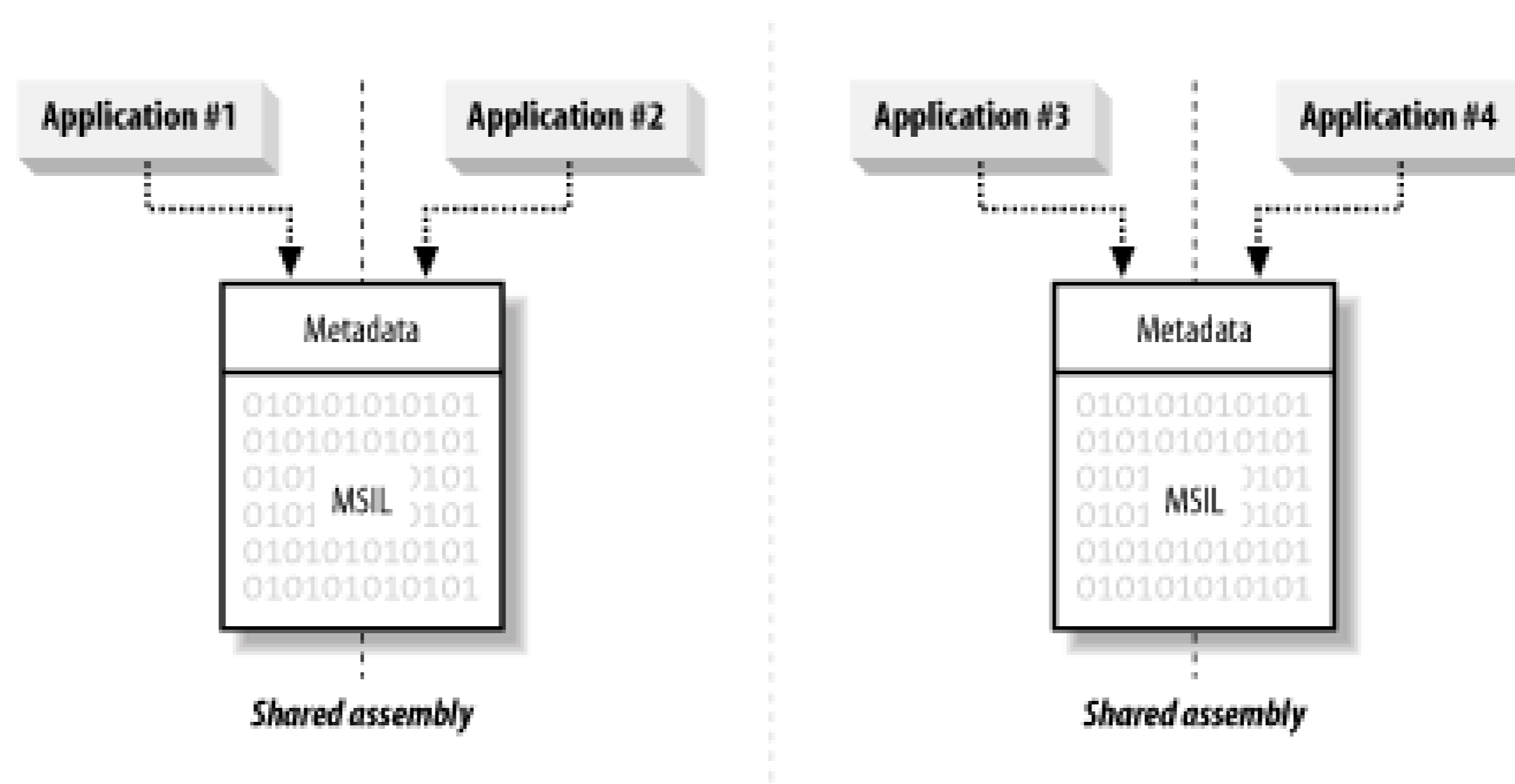
Assemblies can be private or shared, as illustrated by [Figure 2-3](#). Each application that uses a *private assembly* has its own copy, which is stored alongside the other application components. If there are two applications installed on the same computer, and each application relies on the same private assembly, then there will be two copies of the assembly files installed. Each time you install a new application that relies on the private assembly, you create new instances of the assembly files on disk. Each assembly is independent of the others, and each application is completely self-contained.

Figure 2-3. Private and shared assemblies



By contrast, several applications can use a single instance of a *shared assembly*. The assembly is placed in a common location (a shared directory or network server), and every application that requires the assembly uses the same assembly disk files. More than one copy of a shared assembly can be installed, and groups of applications can rely on different instances, as illustrated by [Figure 2-4](#).

Figure 2-4. Groups of applications relying on multiple identical private assemblies



The .NET Framework also provides the *Global Assembly Cache* (GAC), which is a central repository for shared assemblies. All applications that rely on a shared assembly stored in the GAC use the same disk files, as illustrated by [Figure 2-5](#); such applications do not need to be configured to find shared assemblies in specific locations; the GAC is always available to all applications. See [Section 2.5.1](#) for further details.

Figure 2-5. The GAC hosts shared assemblies

When we say that applications "share" an assembly, we mean that they use the same disk file. Each application is given its own copy of the data types contained in the assembly, and no data is shared between the applications. See [Chapter 4](#) for more information on the role of an assembly in an application.

The benefits of sharing an assembly are administrative—the assembly security policies and configuration that we discuss in the following chapters are applied to a shared assembly once, affecting all of the applications that rely on the assembly, whereas each instance of a private assembly must be configured individually.

[\[Team LiB \]](#)

[Team LiB]

2.4 Strong Names

Shared assemblies must have a *strong name*. A strong name consists of the assembly's name, version and culture metadata, plus a cryptographic public key and a digital signature. Because it is sufficiently unlikely that two different assemblies will have the same strong name, Microsoft considers a strong name to be a *unique* identifier of an assembly. Two properties of digital signatures support this position:

1. Digital signatures are associated with a hash code generated from the contents of the assembly; this hash code acts as a "fingerprint" for the assembly contents, and it is very difficult to find two assemblies with the same fingerprint.
2. Cryptographic key pairs, used to create the digital signature, are created randomly.

The randomness of key pairs, and the uniqueness of the fingerprint provided by the digital hash code and signature, means that a strong name, for all practical purposes, provides a unique identification for an assembly. See Chapter 13 for details of digital hash codes and Chapter 16 for details of digital signatures.

The chapters in the Cryptography section of this book provide information about assessing the cryptographic meaning of words and phrases such as "unique" and "very difficult."

The uniqueness of a strong name allows the .NET Framework to verify the contents of an assembly to protect against tampering; the hash code generated from the assembly contents means that changing even a single MSIL statement will invalidate the digital signature, leading the .NET Framework to reject the assembly. This protection against tampering is a fundamental security feature of strongly named assemblies and is a key function in the life cycle of a .NET application, which we discuss in Chapter 4. Strong names are a key feature of code access security, which we discuss in Chapter 6 through Chapter 9.

Library assemblies with strong names can be used only by *fully trusted* code, unless the `AllowPartiallyTrustedCallers` attribute has been applied to the assembly. See Chapter 7 for information about trusted code and the implications for your development projects.

2.4.1 Creating a Key Pair

The first step towards creating a strong name is to generate a public and private key. The private key is used to create the digital signature and must be kept secret; the public key is included in the assembly as part of the strong name and is used by the .NET Framework to validate the digital signature and to determine if the contents of an assembly have been tampered with. See Chapter 15 for information about public/private key pairs and Chapter 16 for details of digital signatures.

Create a new pair of keys using the Strong Name tool (*sn.exe*), supplied as part of the .NET Framework SDK. The following command illustrates how to create a new pair of keys, stored in a file named *MyKeyPair.snk*:

```
sn -k MyKeyPair.snk
```

The contents of the *MyKeyPair.snk* file are not human-readable, but you can view the public key with the following command:

```
sn -tp MyKeyPair.snk
```

The output is below; the public key is a sequence of hexadecimal characters. This command also displays the *public key token* (marked in bold), which is a hash code of the public key, used by other assemblies as a compact way of referencing your public key.

```
Microsoft (R) .NET Framework Strong Name Utility Version 1.1.4322.510
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.
```

Public key is

```
00240000048000009400000006020000002400005253413100040000010001004b251886d12dfd0501c03
2d9b1d36888f3ee68ecc047a56d457ac21d64a54467d7d223c5be2e99bf2ed499c16db4cee567c16d94d2
94aece0348a00025eb7d415005d887c4cee3ee26b83d9cb4b29ed7b21b0c5097f4277d260938f5e00a4cf
f90cf20befd491782b1f3a34675c3ce6c2224bdcc2e329057031937f7d4e032b7
```

Public key token is 060fc0167cf4a028

You can install the newly created key pair into the Windows Cryptographic Service Provider (CSP) container, which provides a centralized store of key pairs. Installing key pairs in the CSP offers easier administration than storing the keys in a disk file; see the Windows documentation for full details. The following command demonstrates how to install keys into a CSP container called *MyCSPContainer*; you will see how to use this container in the following sections.

```
sn -i MyKeyPair.snk MyCSPContainer
```

It is very important that your private key remains a secret-if not, then assemblies can be assigned strong names, which will appear to have originated from you. And why be granted an unwarranted level of trust by system administrators?

2.4.2 Creating an Assembly Strong Name

Before you can generate a strong name, you need to specify the assembly's version and culture metadata; remember that a strong name consists of the assembly name, the version and culture information, plus a public key and a digital signature.

2.4.2.1 Specifying the version and culture metadata

The simplest way to specify the version and culture metadata is to use the *System.Reflection.AssemblyVersion* and *System.Reflection.AssemblyCulture* attributes, applied at the assembly level. Visual Studio .NET creates a file called *AssemblyInfo.cs* (C#) or *AssemblyInfo.vb* (Visual Basic .NET) as part of a project; this file contains attributes for all of the

configurable assembly metadata. We shall follow this convention to specify the metadata for our example assembly (from Section 2.2) as follows; we have specified that this is Version 2.1.0 of our assembly and that the assembly is suitable for any culture setting (see the .NET documentation for details of version and culture configuration):

```
# C#

using System.Reflection;
using System.Runtime.CompilerServices;

// Specify the culture and version metadata
[assembly: AssemblyCulture("")]
[assembly: AssemblyVersion("2.1.0")]

# Visual Basic .NET

Imports System
Imports System.Reflection
Imports System.Runtime.InteropServices

' Specify the culture and version metadata
<Assembly: AssemblyCulture("")>
<Assembly: AssemblyVersion("2.1.0")>
```

2.4.2.2 Specifying the key pair

We specify the key pair to use when creating the strong name by adding an attribute that refers to either the file or CSP container that contains our key pair; we do this with the `AssemblyKeyFile` and `AssemblyKeyName` attributes. You should specify a file *or* a CSP container-do not specify both. To specify a key pair file, add the following attribute to the *AssemblyInfo.cs* or *AssemblyInfo.vb* file:

```
# C#

[assembly: AssemblyKeyFile("MyKeyPair.snk")]

# Visual Basic .NET

<Assembly: AssemblyKeyFile("MyKeyPair.snk")>
```

To specify that the key pair should be obtained from the `MyCSPContainer` CSP container, add the following attribute to the *AssemblyInfo.cs* or *AssemblyInfo.vb* file:

```
# C#

[assembly: AssemblyKeyName("MyCSPContainer")]

# Visual Basic .NET

<Assembly: AssemblyKeyName("MyCSPContainer")>
```

2.4.2.3 Creating the strong name

The .NET language compilers will automatically create the strong name if you specify *AssemblyInfo.cs* or *AssemblyInfo.vb* in the list of files to compile. The following command demonstrates how to apply the strong name to the example single-file assembly that you created previously:

```
# C#
```

```
csc /out:SingleFileAssembly.dll /target:library SumNumbers.cs SumArray.cs
AssemblyInfo.cs
```

```
# Visual Basic .NET
```

```
vbc /out:SingleFileAssembly.dll /target:library SumNumbers.vb SumArray.vb
AssemblyInfo.vb
```

2.4.2.4 Creating a strong name for a multiframe assembly

For a single-file assembly, the compiler uses the assembly attributes to locate the key pair and generates the strong name. When working with a multiframe assembly, you must specify the information required for the strong name as arguments to the Assembly Linker tool (*al.exe*); the following command demonstrates how to create a strong-named assembly using the modules you created previously:

```
al /out:MultiFileAssembly.dll /target:library /culture:"" /version:2.1.0 /keyfile:
MyKeyPair.snk SumNumbers.netmodule SumArray.netmodule
```

We specify the culture and version metadata with the `/culture` and `/version` arguments, and the file containing the key pair with the `/keyfile` argument. The Assembly Linker supports the `/keyname` argument to specify a CSP container for the keys.

2.4.3 Delayed Signing

The examples in the previous sections assume that you are able to either create or access the key pair used to create strong names. In large organizations, a *signing authority* may manage the key pair, and you are required to submit your assemblies for signing; this is especially likely if a single key pair is used to create strong names for the assemblies produced by more than one project.

The .NET Framework supports *delayed signing*, where the public key is included in the assembly, but the signature is not created; this allows you to compile and test your assemblies without requiring access to the secret private key. The first step is to obtain the public key from the key pair, a task usually performed by the signing authority; the following command demonstrates how to extract the public key from the key file and place it in a file called *MyPublicKey.snk*:

```
sn -p MyKeyPair.snk MyPublicKey.snk
```

You can also extract the public key from a key pair stored in a CSP container:

```
sn -pc MyCSPContainer MyPublicKey.snk
```

For a single-file assembly, enable delayed signing with the `AssemblyDelaySign` attribute, which you add to the *AssemblyInfo.cs* or *AssemblyInfo.vb* file as follows, specifying the name of the file containing the public key with the `AssemblyKeyFile` attribute:


```
# C#

using System.Reflection;
using System.Runtime.CompilerServices;

// Specify the culture and version metadata
[assembly: AssemblyCulture("")]
[assembly: AssemblyVersion("2.1.0")]

// Enable delayed signing
[assembly: AssemblyDelaySign(true)]
// specify the file containing the public key
[assembly: AssemblyKeyFile("MyPublicKey.snk")]
```

```
# Visual Basic .NET
```

```
Imports System
Imports System.Reflection
Imports System.Runtime.InteropServices

' Specify the culture and version metadata
<Assembly: AssemblyCulture("")>
<Assembly: AssemblyVersion("2.1.0")>

' Enable delayed signing
<Assembly: AssemblyDelaySign(true)>
' specify the file containing the public key
<Assembly: AssemblyKeyFile("MyPublicKey.snk")>
```

Compile the assembly as before:

```
# C#
```

```
csc /out:SingleFileAssembly.dll /target:library SumNumbers.cs SumArray.cs
AssemblyInfo.cs
```

```
# Visual Basic .NET
```

```
vbc /out:SingleFileAssembly.dll /target:library SumNumbers.vb SumArray.vb
AssemblyInfo.vb
```

For a multifile assembly, use the `/delaysign+` argument to enable delayed signing and specify the key file with the `/keyfile` argument, as shown below:

```
al /out:MultiFileAssembly.dll /target:library /culture:"" /version:2.1.0 /delaysign+
/keyfile:MyPublicKey.snk SumNumbers.netmodule SumArray.netmodule
```

The assemblies that these commands create do not have strong names, because they are not digitally signed. The signing authority can subsequently complete the strong name by using the `R` argument of the Strong Name tool, as follows:

```
sn -R MultiFileAssembly.dll MyKeyPair.snk
```

This technique applies to single- and multifile assemblies. As well as specifying the name of the assembly to sign, you must provide the name of the key file that contains the public *and* private keys-the public key file cannot be used to create a digital signature. You can specify a CSP container from which to obtain the key pair with the `-Rc` argument, as follows:

```
sn -Rc SingleFileAssembly.dll MyCSPContainer
```

2.4.4 Verifying a Strong Name

You can verify an assembly strong name with the Strong Name tool (`sn.exe`); verification checks that the contents of the assembly have not been changed since the digital signature was created. The following command verifies our example single-file assembly:

```
sn -v SingleFileAssembly.dll
```

For a valid assembly, the output from this command is as follows:

```
Microsoft (R) .NET Framework Strong Name Utility Version 1.1.4322.510  
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.
```

```
Assembly 'SingleFileAssembly.dll' is valid
```

You can also determine the public key from the key pair used to create the digital signature using the following command:

```
sn -Tp SingleFileAssembly.dll
```

The result of this command is shown below, and you can see that this matches the public key we created in Section 2.4.1 :

```
Microsoft (R) .NET Framework Strong Name Utility Version 1.1.4322.510  
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.
```

```
Public key is
```

```
00240000048000009400000006020000002400005253413100040000010001004b251886d12dfd0501c03  
2d9b1d36888f3ee68ecc047a56d457ac21d64a54467d7d223c5be2e99bf2ed499c16db4cee567c16d94d2  
94aece0348a00025eb7d415005d887c4cee3ee26b83d9cb4b29ed7b21b0c5097f4277d260938f5e00a4cf  
f90cf20befd491782b1f3a34675c3ce6c2224bdcc2e329057031937f7d4e032b7
```

```
Public key token is 060fc0167cf4a028
```

```
[ Team LiB ]
```

[[Team LiB](#)]

2.5 Publisher Certificates

Strong names provide unique identities for assemblies and protect against tampering, but they do not contain any information about the identity of the assembly publisher. The .NET Framework supports the *Signcode* scheme, which requires a publisher to prove its identity to a trusted third-party authority and obtain a *software publisher's certificate* (SPC). Strong names and Signcode are complimentary technologies, and both can be applied to the same assembly. When you use Signcode a digital signature is created using the private key component of the certificate and is embedded into the assembly along with the public components of the SPC. The digital signature provides the link between your SPC and the assembly you signed; other parties cannot sign assemblies with your SPC unless they know the value of the secret key, which is not included in the assembly. See [Chapter 15](#) for more information about public and private keys.

You can use the Certificate Creation tool (*makecert.exe*) to create an X509 certificate for testing purposes and then convert it to an SPC; genuine certificates are obtained from companies such as Verisign. The following command creates a new X509 certificate; the Certificate Creation tool supports a number of options to configure the test certificate, but we will accept the defaults. The test certificate is stored in a file named *TestCert.cer*:

```
makecert -sv MyPrivateKey.pvk TestCert.cer
```

The *-sv* option stores the private key component of the certificate in a file named *MyPrivateKey.pvk*; you will need to use this key later. You will be prompted to enter a password to protect the private key file, as shown in [Figure 2-6](#).

Figure 2-6. Protecting the private key component of the X509 certificate with a password

Convert the X509 certificate into an SPC with the Software Publisher Certificate Test tool (*Cert2spc.exe*), as follows:

```
cert2spc TestCert.cer TestSPC.spc
```

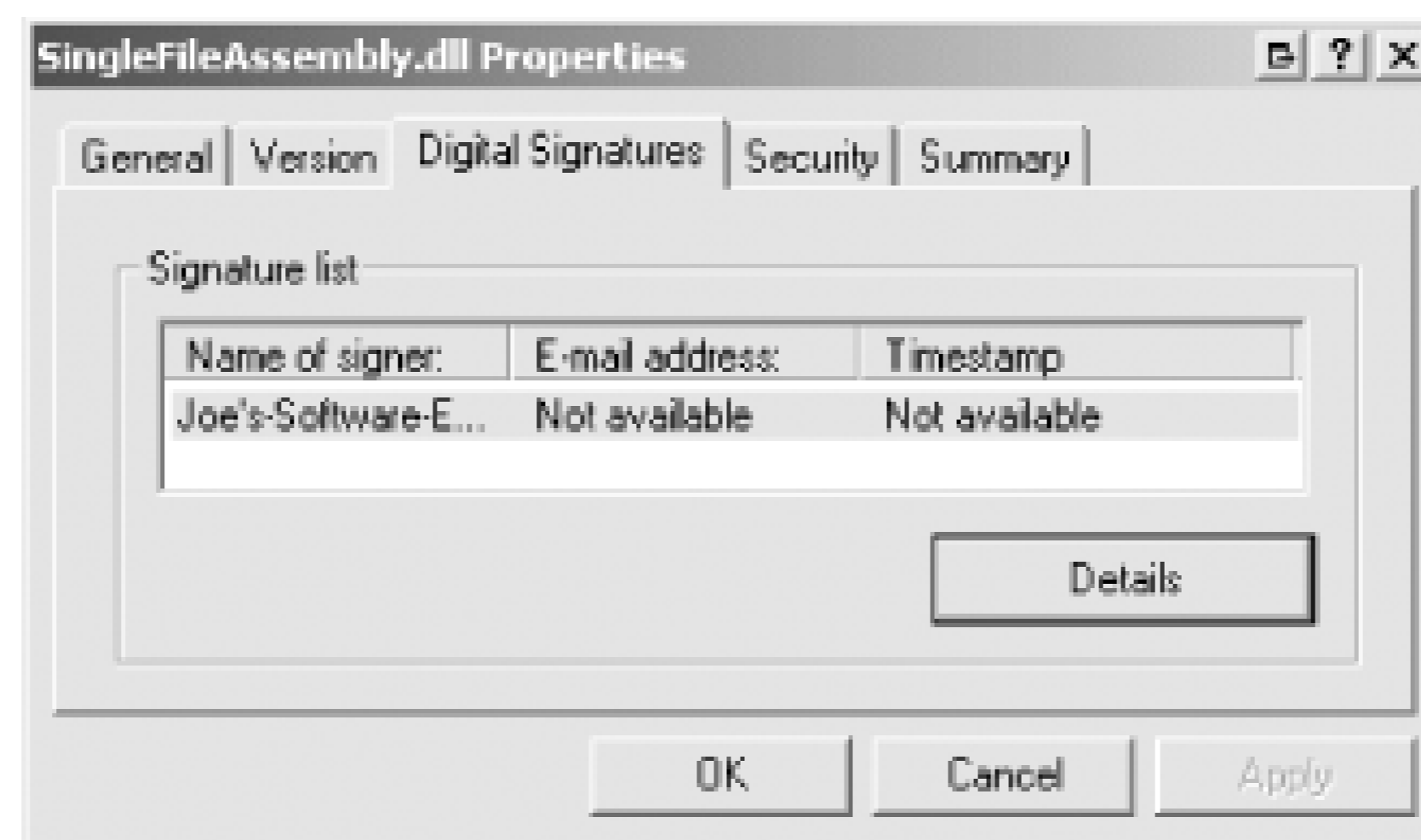
Now that you have an SPC, you can sign assemblies to identify ourselves as the publishers. The following command uses *signcode.exe* to embed our test SPC into our example assembly:

```
signcode -spc TestSPC.spc -v MyPrivateKey.pvk SingleFileAssembly.dll
```

We need to specify the SPC and the file containing the private key in order to sign the assembly. You will be prompted to enter the password you created to protect the private key file. If you right-click

the assembly in the Windows Explorer and select the Properties menu item, you will see that a Digital Signatures tab is available, as shown in [Figure 2-7](#); this tab describes our publisher's certificate, which is created in the name of Joe's-Software-Emporium. A real certificate would contain detailed information about you or your company.

Figure 2-7. The Digital Signatures tab of an assembly signed with a publisher's certificate



2.5.1 Using the Global Assembly Cache

If an assembly is shared by several applications, you can install it into the Global Assembly Cache (GAC), which is a machine-wide store of shared assemblies. There are several reasons why you may wish to install an assembly into the GAC:

Application configuration

The .NET Framework automatically looks for assemblies in the GAC, and an application does not need to be configured to look for assemblies in a known location, as is the case when an ordinary directory is used to store shared assemblies.

File security

The physical location used by the GAC to store assembly files is contained with the WINNT or Windows directory, which system administrators typically protect with a higher level of Windows security than that used for other directories. Installing an assembly into the GAC requires administrator privileges, which reduces the chances of tampering.

Administration

Installing a shared assembly into the GAC can simplify the administration process, allowing configuration policies to be applied to a single assembly instance, but affecting all of the applications that depend on the assembly.

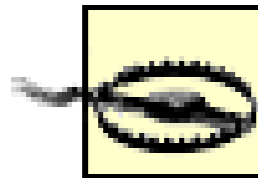
Side-by-side versioning

Multiple copies of assemblies with the same name but different version information can be maintained in the global assembly cache; see the .NET documentation for details.



Microsoft recommends that you install an assembly into the GAC only when necessary. As a rule, you should locate assemblies alongside other application components unless you seek one of the benefits listed earlier.

Use the Global Assembly Cache tool (*gacutil.exe*) to manage the GAC. We will demonstrate how to install an assembly into the GAC, inspect the contents of the GAC, and uninstall an assembly, all of which are simple tasks.



Assemblies must have a strong name before they can be installed in the GAC. See [Section 2.4](#) of this chapter for details.

The following command installs our example single-file assembly into the GAC:

```
gacutil /i SingleFilessembly.dll
```

The following command lists the contents of the GAC, displaying information about every installed assembly:

```
gacutil /l
```

The list that this command produces is very long, but we can filter the output by specifying the name of the assembly we are interested in (note that we do not specify the file suffix once the assembly has been installed into the GAC):

```
gacutil /l SingleFileAssembly
```

The output is below and provides a textual summary of the assembly strong name. You can see the name of the assembly, the version number, the culture setting, and the public key token we assigned to the assembly in [Section 2.4](#) of this chapter:

```
Microsoft (R) .NET Global Assembly Cache Utility. Version 1.1.4322.510
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.
```

```
The Global Assembly Cache contains the following assemblies:
    SingleFileAssembly, Version=2.1.0.0, Culture=neutral,
    PublicKeyToken=060fc0167cf4a028, Custom=null
```

```
The cache of ngen files contains the following entries:
```

```
Number of items = 1
```

Finally, uninstall our assembly from the GAC with the following command:

```
gacutil /u SingleFileAssembly
```

The GAC is a runtime service, meaning that the .NET Framework consults the GAC only when executing an application. You must have a local copy of the shared assemblies that your application depends on at compile time, because the .NET language compilers will not locate assemblies in the GAC.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

2.6 Decompiling Explained

When you compile source code into an assembly, the compiler interprets your C# or Visual Basic .NET statements, and creates a series of MSIL statements that will be executed by the .NET Framework. A *decompiler* is an application that analyses the MSIL statements in order to recreate the original Visual Basic .NET or C# statements written by the programmer.

Unfortunately, the .NET compilers contain human-readable information from our source code in the MSIL, including the names of types, methods, and fields. A decompiler can use this information to create source code that is very similar to the original. Some information, such as comments and blocks of code excluded by conditional compiler statements, are not included in assemblies by the compiler and cannot be restored by a decompiler.

The nature of MSIL makes it easier to decompile .NET assemblies than native Windows applications, which are compiled into instructions that are targeted at a specific CPU, such as an Intel Pentium. Lower-level instructions are more difficult to reconstruct into code statements than the relatively abstract MSIL statements. The proliferation and use of decompilers is more widespread than you might think. There are three main reasons why an assembly is decompiled:

Interest

The most benign reason for decompiling an assembly is simply to gain an understanding of how an application or library is written; the person who decompiles the assembly has no malicious intent, and simply seeks to improve her knowledge of .NET programming.

Intellectual property theft

Your assemblies can be decompiled to reveal your business secrets, which can be used commercially by your competitors. In this context, business secrets may be proprietary algorithms, or the inner workings of your complete application.

Application subversion

Decompiling an assembly can provide details of how to subvert an application, exposing licensing details and allowing license codes to be generated that allow illegal copies of your application to be activated.

The scope for intellectual property theft through decompilation has been lessened by the increased use of thinner clients to connect to network services (this includes the move towards XML web services); there is less complexity to the client application, and the sophisticated logic is deployed within a remote network. In contrast, the prevalence of network services increases the scope for application subversion. Any network service that grants trust to clients based on data that is included in an assembly is subject to subversion through decompilation. Analysis of a client application can provide a wealth of information on network protocols and security configuration, which can be used to manipulate the network components of an application against the wishes and expectations of the developer.

Decompiling into Another Language

MSIL is common between all .NET languages, meaning that an assembly written in Visual Basic .NET can be decompiled to produce C# statements, or statements in any other .NET-compliant language. There is no additional risk posed by this feature, but it does lower the barrier to understand your code; not only can the source code statements be reconstructed, but they can be reconstructed in a language which a potential attacker is familiar with.

2.6.1 Decompiling Assemblies

In this section, we demonstrate how much detail a decompiler exposes from an assembly. You will use the open source Anakrino/Exemplar decompiler to decompile the single-file assembly you created in the previous section; at the time of writing, the decompiler is available at <http://www.saurik.com/net/exemplar/>.

The decompiled versions of the `SumNumbers` and `SumArray` classes are below-the decompiler we have selected generates only C# source code. We do not explain how to install or use the decompiler in this book-we present the decompiled output so that you can understand what kind of information can be obtained from an assembly:

```
# C#  
  
using System;  
  
public class SumNumbers {  
    private int o_total;  
  
    public SumNumbers( ) {  
        o_total = 0;  
    }  
  
    public void AddNumber(int p_number) {  
        o_total += p_number;  
    }  
  
    public int GetTotal( ) {  
        return o_total;  
    }  
}
```

Be careful when you decompile an assembly. You may be in violation of your country's law if you decompile an assembly to which you do not own the copyright or intellectual property rights. We do not condone the decompilation of such assemblies. We accept no responsibility if you decompile assemblies which contain the intellectual or commercial property of others.

The efficacy of a decompiler is measured by the accuracy of the source code that it generates-the

better a decompiler is, the more the decompiled source code resembles the original statements. Our decompilation has produced a rendition of the `SumNumbers` class that is very close to the original; the names of the fields are preserved, and the structure and function of the class is clear:

```
# C#

public class SumArray {

    public static int SumArrayOfIntegers(int[] p_arr) {
        SumNumbers sumNumbers = new SumNumbers( );
        int[] nums = p_arr;
        for (int k = 0; k < (int)nums.Length; k++) {
            int j = nums[k];
            sumNumbers.AddNumber(j);
        }
        return sumNumbers.GetTotal( );
    }
}
```

The decompiled version of the `SumArray` class is less like the original but still clearly demonstrates the implementation. Our simple assembly is easily decompiled, and the workings of our data types are clearly exposed; logic that is more complex can cause difficulties for decompilers, but in general, an unprotected assembly will yield its secrets easily.

2.6.2 Protecting Against Decompilation

If your assemblies contain no proprietary data, and no information that can be used to subvert your application, then you are in a position to distribute the assemblies freely; otherwise, you should consider protecting against decompilation with one of the techniques discussed below.

You should not protect your assemblies against decompilation unless you have a good reason—all of the techniques that make decompilation more difficult rely on changing the contents of your assemblies; this makes it very hard to debug problems and may even introduce new defects into your applications.

2.6.2.1 Obfuscation

Obfuscation is the technique of altering the MSIL statements so that the application executes in the same way, but the output of a decompiler is unreadable. Obfuscation is such an important technique that Microsoft has included a copy of a limited functionality obfuscator in Visual Studio .NET 2003. Different obfuscators use different approaches to obscure decompiler output, but we summarize the more common types of obfuscation below:

Renaming methods and fields

Obfuscators rename the nonpublic methods and fields defined by your data types in a way that makes it difficult to read the decompiled output. A common technique is to use very long strings that differ by a single character or to use non-printing characters accepted by the .NET runtime, but which text editors do not display correctly.

Adding control flow obfuscation

Obfuscators make application logic more difficult to follow by creating complex sequences of method calls that do not do anything. This is more effective than it sounds, because it is hard to establish if these methods are related to the logic of the application. This technique is especially effective when combined with method and field renaming, creating especially complex decompiled output.

Encrypting literal strings

Encryption is applied to the literal strings defined within your assemblies; the purpose of this is to slow down searches that may reveal the purpose of sections of code; for example, searching for the word "license" may reveal which parts of your code deal with application license control

Effective obfuscators combine these approaches and often apply proprietary techniques. There is a kind of "arms race" between the developers of obfuscators and the developers of decompilers, where each new feature added by an obfuscator is eventually compromised by a decompiler.

The biggest problem with obfuscation is that it alters the MSIL within your assembly; when problems arise, you will find that the obfuscation process can seriously hamper the debugging process. As a general guideline, do not obfuscate your assemblies unless you have to, and always select an obfuscator from a reputable company that will be able to support you if you encounter problems.

You must remember that obfuscation does not protect your assemblies against determined decompilation. Although an obfuscator will create MSIL that produces difficult-to-read decompiler output, your application logic is still contained in the MSIL and can eventually be reverse-engineered; at best, obfuscation slows down the process.

2.6.2.2 Native compilation

As you will see in [Chapter 4](#), the .NET Framework runtime compiles your MSIL statements into native commands for the CPU before the code is executed. An alternative to obfuscation is to perform this compilation yourself and to create native instructions that cannot be processed by an MSIL decompiler.

Native compilation is a relatively new technique as applied to .NET assemblies, and the tools available at the time of writing are immature; the principal risk with native compilation is that the output can differ from that produced by the normal .NET compilation process, which can hamper the debugging process.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 3. Application Domains

Application domains play a critical role in the implementation of the Microsoft .NET Framework's security model. Although you will rarely need to work directly with application domains when writing simple applications, they become an essential program element once you start writing more complex systems-in particular, systems that utilize and manipulate the code-access security (CAS) mechanism of the Microsoft .NET Common Language Runtime (CLR).



Only highly trusted code can manipulate application domains-we discuss the specific permissions required to carry out various operations in [Chapter 5](#), [Chapter 6](#), and [Chapter 7](#). For now, it is enough to know that applications running from the hard drive of your machine can utilize all of the techniques we describe here as long as you have not modified .NET's default security policy.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

3.1 Application Domains Explained

To ensure application and operating system security and stability, it is necessary to isolate concurrently executing applications from each other. Application isolation ensures that one application cannot purposefully or inadvertently modify the memory, or access the resources owned by another. Traditionally, operating systems isolate applications by running each application in its own process, and providing each process with its own virtual memory space and resource handles. Memory references are associated with a single process, and therefore code cannot obtain a reference that affects the memory of another process. Code cannot access objects in other processes directly and must use some intermediate communications mechanism, such as a remote procedure call or a proxy in order to invoke methods.

3.1.1 Application Isolation in the .NET Framework

Application isolation in the .NET Framework is very different from traditional application environments. Before the CLR loads an assembly, it verifies that the assembly contains only *type-safe* code. Type safety guarantees that the code does not perform illegal operations, access memory directly, or try to access type members incorrectly. If an assembly fails the verification process, the CLR will refuse to load it and will throw an exception. We discuss the verification of assemblies further in [Chapter 4](#).

Loading Non-Type-Safe Code

It is possible to load non-type-safe code into the CLR; however, that undermines the application isolation and security mechanisms provided by the CLR. Because every assembly is running in a single process, code that directly manipulates the memory of the containing process does not only modify the memory used by other application assemblies, but it can also modify the memory used by the CLR and the .NET class library. A piece of poorly written software could accidentally overwrite blocks of memory used by other applications. Even worse, a well-written piece of malicious code could use pointers to change data critical to the operation of the CLR.

Only highly trusted code from a verifiable source should be granted the permission to load non-type-safe code and only when specific needs demand. We will discuss this more in [Chapter 7](#), when we talk about code-access security permissions.

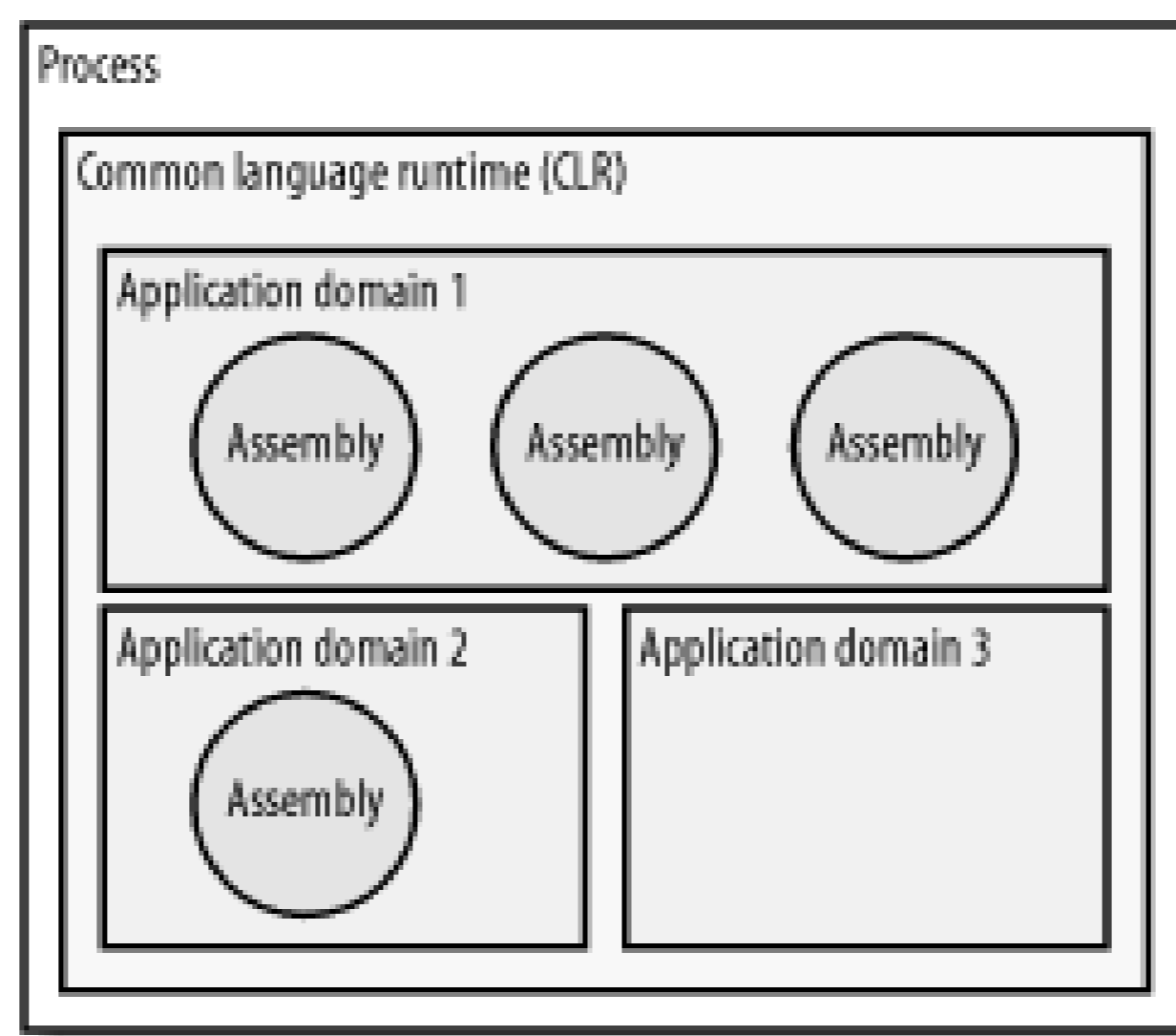
The fact that code is type-safe allows the CLR to load many assemblies into the same operating system process without the risk of those assemblies crashing or corrupting each other. This solves many code stability issues and enables the CLR to use application domains—a lightweight alternative to processes—to ensure application isolation. Application domains are logical containers within the CLR that provide an isolation boundary between type-safe assemblies equivalent to that provided by processes for native applications.



Switching between processes is a relatively expensive operation on most software platforms. Enabling multiple applications to run within a single process (without sacrificing application isolation) significantly reduces the runtime overheads. This is particularly useful in server solutions (such as ASP.NET) that must scale to support large numbers of concurrent users across many different applications.

The relationship between a process, the CLR, application domains, and assemblies is illustrated in [Figure 3-1](#). Three assemblies have been loaded into the first application domain and one assembly into the second. As yet, no assemblies have been loaded into the third application domain.

Figure 3-1. The relationship between a process, the CLR, application domains, and assemblies



By isolating at the assembly level, .NET provides a fine-grained level of control over the isolation of your applications. Not only can you isolate entire applications from each other, but you can also isolate individual parts (assemblies) from the rest of your application. This is particularly useful if your application provides for extensibility using plug-ins or dynamically loadable modules that are of questionable quality, origin, or intent (for example, Internet Explorer, which allows .NET controls to be downloaded and run from web sites).

3.1.2 Application Domain and Assembly Management

The CLR provides an execution environment for managed code, but a runtime host—at least written partially in native code—is required to load the CLR before it can execute your managed code. When the CLR is loaded, it automatically creates a default application domain. However, the runtime host usually determines when and if additional application domains are created. Runtime hosts are free to use different application domain strategies to suit their application execution model and security requirements. Although it is possible to create custom runtime hosts, the three you will encounter most frequently are those supplied with the .NET Framework:

- The ASP.NET runtime uses a single CLR that runs all ASP.NET web applications; each web application is loaded into a separate application domain.

- Internet Explorer uses a single CLR that executes all downloaded controls. An application domain is created for each web site from which controls are downloaded; all controls from the same site run in a single application domain.
- The CLR used to execute commands run from the operating system shell creates a single application domain that the executable assembly and all referenced assemblies are loaded to.

When the CLR loads an assembly, it usually loads it to the same application domain as the code that referenced it. As mentioned, the runtime host may create a separate application domain in which to load the assembly. It is also possible for application code with the appropriate permissions to create application domains and to load assemblies into those application domains. This allows application code to take direct control of isolating specific assemblies, and provides a mechanism through which code can unload assemblies. It is not possible to unload an individual assembly once it is loaded; however, it is possible to unload an entire application domain, which unloads all the assemblies loaded in to it.

3.1.3 Assembly Isolation with Application Domains

Code loaded into one application domain has no direct access to code loaded into other application domains. While it is possible for code to enumerate the assemblies and types loaded into its own application domain, instantiate those types, and invoke object methods, it is not possible for code to perform these actions in a remote application domain without first obtaining a reference to the remote domain. Only the creator of an application domain or code running in an application domain can obtain a reference to a domain; it is not possible for managed code to enumerate the application domains that exist in the CLR. Because all code is type-safe and guaranteed not to perform direct memory access, it cannot break through this isolation to access directly the code or data of an assembly in another application domain, even though they are loaded in the same process.

With an object reference to a remote application domain, code has more ability to interact with that domain; however, there are still access controls.

Application domains are .NET Remoting boundaries, which place limitations on the communications between application domains. We refer you to the .NET Framework SDK for a discussion of .NET Remoting.

3.1.4 Application Domains and Runtime Security

Application domains play a key role in the operation and enforcement of runtime security. Runtime security is a large and complex topic, which we explore fully in Part II of this book. In the following sections, we provide a summary of the relationship between application domains and each aspect of runtime security.

3.1.4.1 Assembly evidence and identity

When you load an assembly into an application domain, you can specify additional evidence to apply to the assembly. The CLR uses evidence to determine an identity for an assembly, from which it determines the permissions to grant to the assembly. The permissions granted to an assembly control the actions and resources that are accessible to the code contained in the assembly. By

changing an assembly's evidence, you change its identity, which can radically change the permissions granted to it by the CLR. We demonstrate how to modify the evidence of an assembly in [Chapter 6](#).

3.1.4.2 Application domain evidence and identity

When you create an application domain, you can assign evidence to it. As with assemblies, evidence gives application domains an identity from which the CLR derives a set of permissions. When code tries to perform a protected operation, the CLR checks not only the permissions granted to the assembly in which the code is contained, but also the application domain in which the code is running. There is more to this process, but the key point to remember is that the permission of an application domain restricts the capabilities of the code running in it. This is a powerful feature, especially when loading and executing code from untrusted sources. We discuss how to assign evidence to application domains in [Chapter 6](#), and describe the effect an application domain's permissions have on the code in [Chapter 8](#).

3.1.4.3 Application domains and security policy

The CLR uses a set of configurable rules (security policy) to provide a mapping between an assembly's evidence and the set of permissions granted to it when it is loaded. Security policy is divided into four layers, or policy levels: enterprise, machine, user, and application domain. Each layer is evaluated in order from highest (enterprise) to lowest (application domain), with each layer further restricting the permissions that are granted by the previous layer; lower layers are never able to grant more permissions than those granted by higher layers. The highest three layers (enterprise, machine, and user) are statically configured using administrative tools and stored in configuration files. Application domain policy is defined programmatically, allowing you to set security policy at runtime instead of relying on statically defined security policy only. We discuss application domain security policy in [Chapter 8](#).

3.1.4.4 Role-based security

Although code-access security solves many of the long-standing security issues surrounding the use of mobile code, it does not remove the need for the more familiar role-based security mechanisms. The .NET framework also contains support for role-based security. When code attempts to perform actions, it does so in the context of a principal. A principal represents the identity of the active user and any roles (user groups) to which the user belongs. By default, the application domain through which a thread runs defines the principal on whose behalf the thread acts. We discuss role-based security in [Chapter 10](#).

3.1.4.5 Application domains and isolated storage

.NET provides a facility named isolated storage, which provides a managed storage area for .NET applications to write user and application state information. Isolated storage provides a relatively safe area where applications can write data without being granted access to important resources such as the hard drive and the Windows registry. Isolated storage also ensures that one application cannot access another application's data. The mechanism used to protect one application's data from another is based on the identity of the assembly and the application domain in which the assembly is loaded.

We discuss isolated storage in [Chapter 11](#).

3.1.5 Application Domains and Application Configuration

The configuration of an application domain determines how the CLR loads assemblies into the application domain and provides important configuration information that controls the runtime operation of the loaded assemblies. This means that an assembly loaded into one application domain can operate very differently than the same assembly loaded into another application domain. An application domain's configuration controls the following behavior, which can have major consequences for application security if used inappropriately.

- Depending on the mechanism used to load an assembly, the CLR will search for it in different places. These usually include the directory from which the executing assembly was loaded and the global assembly cache, which we discussed in [Chapter 2](#). Using application domain configuration, it is possible to specify additional locations where the CLR will look for assemblies to load. If configured incorrectly, code could have the ability to load assemblies to which it should not have access or could even attempt to load the wrong assembly inadvertently.
- Application domain configuration defines the name and location of the configuration file used by assemblies that are loaded into it. Depending on the use an application makes of configuration files, specifying a different one can fundamentally change the application's behavior. This setting is particularly important if the application uses a configuration file as a source of security-related information, such as the authentication or authorization mechanisms to use, or which cryptographic algorithms to use to encrypt data.

Consult the .NET Framework SDK documentation for specifics on how to configure these application domain settings.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 4. The Lifetime of a Secure Application

This chapter looks at the way in which software security is integrated into the application lifetime and provides practical advice that will help you understand the content contained in later chapters.

We define the term *secure application* to mean an application designed with security in mind. We firmly believe that truly effective software security is achieved only when it is completely integrated into the application development process and is understood to be every bit as important as stability, performance, and feature completeness.

We recommend that you read this chapter twice. Read it now to help build a context for the technical content of the following chapters. When you have finished reading this book and have a better grasp of .NET security, read this chapter again, and consider how you can improve your development process to best implement the advice and recommendations we make.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

4.1 Designing a Secure .NET Application

With few exceptions, a design is produced for an application before development begins; for smaller projects, the programmers may produce the design, which may be closely related to the implementation and contain low-level technical details. Larger projects usually engage an application architect to produce a more abstract design, leaving development of its components to individual development teams.

Security is an important part of the design process and cannot be left until the implementation phase. A fully integrated security policy will provide the greatest protection against your application being subverted and simplify the process of integrating security functionality into your code. You cannot retrofit a comprehensive security model into a design.

As the application designer, you need to have an understanding of the security capabilities provided by the platform that the application uses, in the same way that you must understand the features and functions of other components, such as databases and operating systems. This knowledge is important even if you will not be involved in the implementation of the application. Where possible we have written each chapter so that an explanation of the security features offered by .NET is separate from the details of how to apply the functionality during coding; we recommend that architects working at even the most abstract levels should read the latter material.

The first step towards applying security to an application design is to identify the *restricted resources* and *secrets*, two concepts that we introduced in [Chapter 1](#). Recall that a restricted resource is functionality to which you wish to control access to, and a secret is some piece of data that you wish to hide from third parties.

4.1.1 Identifying Restricted Resources

Restricted resources tend to fall into three categories:

Functional resources

Functional resources are the features that your application provides, for example, the ability to approve a loan within a banking application. These resources are easy to identify and are defined with the functional specification for the application.

External resources

External resources are those that underpin your application—for example, a database. Access to these resources should be coordinated with access to your functional resources, so that, for example, users who are unable to approve loans through a functional resource are not able to edit the database directly to achieve the same effect. This coordination proves the need for a wider security view that we introduced in [Chapter 1](#).

Subversion resources

Subversion resources do not appear to be significant at first glance but can be used in conjunction with a functional or external resource in order to subvert your application or the

platform on which your application executed. For example, one resource is the ability to write data to a file that is used by the operating system to enforce security policy.

Creating the list of restricted resources associated with your application is the foundation for understanding the trust relationships that you need to define, which we discuss in the next section. We make the following suggestions to assist in developing your skills in identifying restricted resources:

- Consider the way your application interacts with other systems. Think carefully about the way in which your application depends on other services. Access to some resources may need to be restricted in order to protect other systems, even though they cannot be used to subvert your application.
- Apply common sense. Do not follow the business specification slavishly—as an architect, you are responsible for designing an application that satisfies *all* of the business and technical objectives of the project, even those that are not stated explicitly. By applying some common sense, you can often identify resources that must be restricted in order to achieve the wider objectives of your organization.
- Define and follow design standards. By applying a common design methodology to all of your projects, you can create patterns of functionality that are recognized easily as restricted resources.
- Open your design to review. Do not work in isolation—ask for, and act on, the comments of your colleagues. Different people think in different ways, and we have found that reviewing application designs in groups is especially effective for identifying subversion resources.

4.1.2 Identifying Trust

Once you have identified the protected resources your application uses, you can define the trust levels your application will require of its users or clients to grant access to them. Trust can be granted to a wide range of entities, including users, code, external libraries, and different computers.

When you trust a user, you are granting the individual an ability represented by a restricted function resource. For example, if you were developing a banking application, you might allow loan officers to approve loan requests. .NET supports this approach with *role-based security*, where you define the types of user that will use the system, and consider which resources each requires to perform his work. We discuss .NET role-based security in [Chapter 10](#).

If you grant trust to code, or to a class library, then you grant trust to the assembly that contains the classes; we introduced assemblies in [Chapter 2](#). .NET uses attributes of the assembly (the strong name, the publisher, etc.), known as *evidence*, to grant trust to assemblies. We discuss evidence in [Chapter 6](#), and its uses in [Chapter 7](#), [Chapter 8](#), and [Chapter 9](#).

When you *require* trust, you are in effect asking to consume the restricted resources of another application. For example, if your application depends on access to a database, then you must ensure that the database server will accept connections from your computer, and that your application is able to read and write the data it needs. Trust is a chain of relationships that extends through the services on which your application depends, your application itself, and the services or users that make use of your application.

Identifying trust is the process of examining your restricted resources and establishing how access to them should be allocated. You should also examine how the services you depend on assign trust and ensure that your application design provides for complying with their trust demands. We make the following suggestions to assist in developing your skills in identifying trust:

- Assign the smallest amount of trust required to perform a task. Define levels of trust that are closely associated with the tasks that will be performed by your users, and ensure that you are granting the smallest amount of access to restricted resources that the user requires to perform her tasks. Do not group trust into large, poorly defined levels in the name of "simplicity"; such an approach will lead to a poor application design.
- Consider the effect of the trust chain and the possible subversion it creates. Ensure that your application is not a tool for gaining illegal access to a more important resource. For example, if a database trusts your application to read and write data, does your application provide an easier hacking target than the database itself? Your application should not grant trust to external resources in a way that bypasses security measures implemented by those resources.
- Consider the real-world organization of trust. Think about the way in which trust is assigned to the users of your application in the real world, and consider following that model as the basis for specifying roles in your application.
- Consider what overlapping trust allows. Examine the trust that your application requires, and think about the effect that granting multiple trust levels will have. For example, you may have defined one trust level that grants access to read the list of customer accounts and another trust level that grants the ability to send emails. If you assign both of these trust levels to a single user, you have granted the ability to email details of the customer account outside of the company. Try to think how a user might tie together apparently unrelated resources and activities to subvert your security policies.
- Document your decisions clearly. When you document the design of your application, explain why you have defined your trust levels, and clearly illustrate the way in which these trust levels should be assigned to user roles or services. Clearly documenting this information will simplify the development process and serve as an authoritative reference during application testing.

4.1.3 Identifying Secrets

Identifying secrets is usually a simpler process than identifying restricted resources; you must examine each type of data that your application creates or processes, and decide if that data requires protection. In Part III, we discuss the measures that you can take to protect data, but we offer the following advice to assist you in ensuring that you correctly classify your application's data:

- Consider who owns the data. If you process data that is created or supplied by another application, you should protect the data to at least the same level as that application does. Your application should not be an easy means of accessing data that is better protected elsewhere.
- Consider your legal obligations. You may have legal obligations to ensure that certain information remains private, especially the personal details of your customers. Seek legal advice

to establish your responsibilities.

- Consider the effect of disclosure. Understand the impact of disclosing the data that your application works with, and use this information to assess what data should be a secret. Remember, sometimes data is protected for reasons of public image rather than practical considerations. For example, the damage to your company's reputation exceeds the damage to a credit card holder if you expose his card number to the world. The credit card provider limits the cardholder's liability, but there is no limit to the amount of damage bad publicity can do to your business.

4.1.4 Failing Gracefully

You must accept that you cannot design or implement an application that is invulnerable to attack—even the best security can be broken. As part of an application design, you should always specify what actions will be taken in the event of a security breach, and define a plan of action that ensures that your application fails gracefully, and does not expose other applications and services to subversion.

It is not acceptable to simply write security events to a log file and hope that someone acts on them. As the designer of the application, you have a responsibility to minimize the risk to which you expose your clients and customers. The details of the failure plan will differ based on the purpose and complexity of the application, but should include, at a minimum, details of how security breaches are to be dealt with and what immediate actions should be taken to protect your data and protected resources from further compromises.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

4.2 Developing a Secure .NET Application

The developer takes the application design from the architect and develops the classes that form the application implementation. The developer must have a good working knowledge of software security, and especially the security features provided by the development platform. This knowledge is required so that the developer can correctly program the security policy as part of the implementation process.

The developer takes a much more concrete approach than the abstract approach taken during the design, and takes the broad policy laid out in the application design and transforms it into a robust and accurate implementation.

We do not suggest that the developer should follow the application design to the exclusion of everything else. As the developer, you have an obligation to assess the practicality and suitability of the application design, and the security policy the design defines. Question the design appropriately and bear in mind that your in-depth knowledge of programming security should be used to collaborate on improving a faulty design, rather than as a weapon in a political or cultural war. Nonetheless, respect the purpose of the application design, and do not deviate from it unless you have the architect's permission—needlessly deviating from the design will lead to implementation defects, which can present unforeseen opportunities to attack and subvert your application.

The developer is often responsible for making implementation decisions, such as the strength of cryptography used to protect secrets or the way security roles are employed. There is often a temptation to adopt new and exciting technologies, which is a dangerous approach when applied to application security. Security is best established by using tried-and-tested techniques, and by using algorithms and implementations that have been subjected to extensive testing.

As you will see in later chapters, .NET security is implemented by the developer but is configured by the system administrator. You should implement your security policy to simplify the configuration wherever possible, and to use default settings that offer a reasonable level of security without any configuration at all. You cannot expect a system administrator to have the in-depth knowledge required to develop the application or the time to invest in learning the intricacies of your application. Document the default settings you have used, and explain their significance. We offer the following advice to assist you in developing applications securely:

- Ensure that someone knows when you make a change. Implementing changes in isolation is likely to open security holes in your application. Components of a software system are often highly dependent on each other. Unless told of a change, other people working from the original design will assume that your components function as specified and will make implementation decisions for their own components based on those assumptions.
- Do not be afraid to ask questions. You should always seek clarification when you do not understand part of the application design; many developers feel that this is a sign of "weakness," but our experience is that confusion is often caused by lack of clarity in the design or an error made by the architect. Always make sure you understand *all* of the design before implementing the application.

- Take the time to understand the business. You will find it easier to understand the decisions made in the application design if you take the time to understand the business problem the application is intended to solve. Remember that the architect is the "bridge" between the business problem and the technical solution, and decisions that may appear to have no technical justification are often influenced by business factors.
- Do not rely on untested security libraries. Developers are usually responsible for selecting third-party tools and libraries for the application implementation. We recommend that you select security libraries from reputable companies and submit their products to your own security-testing procedure (see [Section 4.3](#)).
- Apply rigorous unit testing. You should test all of the classes that you develop as part of the application implementation. This testing should not only test the expected behavior, but also make sure that unexpected inputs or actions do not expose security weaknesses. In this regard your unit testing is a simplified form of the security testing that we proscribe below.
- Remove any default accounts before deployment. It is usual to create default user accounts or trust levels that simplify unit testing; you must ensure that these are disabled or removed before the application is tested and deployed.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

4.3 Security Testing a .NET Application

As we stated in [Chapter 1](#), security testing is unlike ordinary application testing-the security tester looks for ways to subvert the security of an application prior to its deployment. Effective security testing can significantly reduce the number of security defects in an application and can highlight flaws in the application design. We offer the following advice to assist you in security testing applications:

- Play the part of the employee. Do not limit your simulated attacks to those you expect a hacker to make-be sure to determine if it is possible for a disgruntled employee to subvert the application security. Employees are usually assigned more trust in an application security mode which can sometimes provide easier routes of attack.
- Test beyond the application itself. Your testing should include attacks on the system on which the application depends, including database, directory, and email servers. In the case of .NET, you should also see if you can subvert your application via an attack on the runtime components. Poor configuration or a poor understanding of security functionality can often provide an avenue for an attack that can subvert the application indirectly.
- Test beyond the application design. Do not fall into the trap of simply testing to ensure that the application design has been correctly implemented; this is functional testing, and it does not offer many insights into security failures.
- Monitor trends in general attack strategies. Expand your range of simulated attacks by monitoring the way real attacks are performed. Your customers may furnish you with descriptions of attacks they have seen, and you can learn from the way other applications and services are attacked.

There is a growing awareness of the value in security testing, and tools have started to emerge to assist in the testing process. The first generation of tools are focused on testing the configuration of an application and the .NET runtime, but work is in progressing on more complex software that will automate applying common types of attack. See the Microsoft .NET home page for information about .NET testing tools in general and some links to security-testing tools.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

4.4 Deploying a .NET Application

The system administrator is responsible for installing and configuring the application. This task includes assigning user accounts to the roles defined by .NET role-based security (see [Chapter 10](#)) and assigning appropriate levels of trust to the assemblies that make up the application. Equally important is the configuration of the services required by the application, such as database and directory servers.

If you are the system administrator, you have an obligation to gain an understanding of how the application should be configured, and to spend the time to determine how the security configuration is best tailored to your enterprise. You have a reasonable expectation that the software publisher will provide you with a robust and functional application, and the software publisher has a reasonable expectation that you will install and configure its application by following its instructions and by applying your knowledge of the company.

Nonetheless, you should consider carefully the levels of trust that you assign to a publisher's assemblies, and ensure that you are not granting an application more permissions than it requires to perform correctly. You should also ensure that you are not compromising the security of your corporate network by configuring the application and the services that it depends on.

Your final obligation is to monitor the application in order to watch for security defects or breaches and to report these problems to the software publisher. See [Section 4.6](#) for an explanation of how the management of security continues for as long as the application is in use.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

4.5 Executing a .NET Application

Once a system administrator has installed and configured the application, it can be executed using the Common Language Runtime (CLR). The CLR is a complex piece of software, and a detailed discussion of how the CLR works is outside the scope of this book. In this section, we discuss the two facets of CLR operation that relate specifically to security.

Verification is the first aspect of the CLR we are interested in. Before executing a managed application, the CLR completes a verification process that is the first step toward enforcing .NET security. If the application is made up of strong-named assemblies (see [Chapter 2](#) for details), then the contents of the assemblies are checked and compared with the value of the digital signature; the CLR will not execute code from any assembly that has been tampered with. If the contents of the assembly are unchanged, the CLR verifies that the code contained within the assemblies is *type-safe*, meaning that the code does not perform illegal operations, access memory directly, or try to access type members incorrectly.

If the code contained in the assemblies is type-safe, and the contents of the assembly have not been tampered with, then the CLR examines the *evidence* of each assembly, and grants *permissions* to the code within that assembly based on the .NET *security policy* configuration. We discuss evidence in [Chapter 6](#), permissions in [Chapter 7](#), and security policies in [Chapter 8](#).

The important fact, and one you must understand, is that the security implementation (performed by the developer) and the configuration (performed by the system administrator) are combined by the CLR when the application is started to determine how the application is allowed to execute. .NET security is not simply proscribed by the software publisher; it is something that requires the cooperation and understanding of the customer.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

4.6 Monitoring a .NET Application

It is extremely difficult to develop and deploy software that is reasonably secure, and impossible to develop and deploy software that is invulnerable to attack. Reconcile yourself to the fact that your application may be subverted, and plan accordingly.

Do not stop thinking about the security of your application when you have deployed the final product or even when the system administrator has completed the configurations and users are making use of the functionality. The impact of security lasts as long as the life of the application itself; you should be prepared to monitor for security breaches, and have a plan in place to deal with them.

An effective weapon against hacking is education; you should ensure that your customers understand how you have applied security within your application, assist them in understanding how to recognize an in-progress attack, and how to tell when a hacker has bypassed or subverted your application security. You should aim to build a relationship with your clients that makes it possible for them to report security problems to you, and endeavor to respond to such reports in a responsible and sensible manner. We believe that as a publisher of software, your responsibilities include:

Analyzing security reports

If you are successful in establishing a way in which customers can report security attacks and defects, you have a responsibility to use this information to assess the impact of potential problems and act to reduce the risk to your customers and users.

Assisting your customers in developing a failure strategy

You should portray the security software techniques you apply to your applications in an accurate and honest manner, and not make unreasonable or unlikely claims. You should educate your customer, ensuring that they understand that no security precautions are impervious to concerted attack, and help them to develop a plan of action to execute in the event of a successful subversion of your security measures. Such a plan should contain a reasonable approach to assessing the impact of the attack and steps to take to restore the application to service.

Informing your customers of attack trends and patterns

Your customers can benefit from knowing how other customers are being attacked and whether those attacks are successful. This information will allow for an informed assessment of the security risk as it affects their wider enterprise. We recommend that you should be open and transparent in your handling of security matters, and make as much information available to your customers as possible.

Publishing security fixes quickly and publicly

You should use the customer reports you receive to identify security defects in your application and issue updates or workarounds quickly, effectively and, ideally, without charge. Security defects are unlike other software defects and can expose your customer to a wide range of risks, beyond the compromise of your application; a clear and concise security fix policy can only enhance your reputation as a software publisher, and will help you to build a more stable and secure application.

Companies that understand security to be a necessity publish the software products that we consider trustworthy. These companies accept the inevitability of security defects being discovered in their products, and they act quickly and decisively to provide solutions to their customers. We strongly recommend that you do the same for your customers.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Part II: .NET Security

Contains information about the security-related features provided by the .NET runtime. These chapters describe how the runtime enforces application security and how you can manipulate, customize, and extend runtime security to meet your own security requirements:

[Chapter 5](#)

[Chapter 6](#)

[Chapter 7](#)

[Chapter 8](#)

[Chapter 9](#)

[Chapter 10](#)

[Chapter 11](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 5. Introduction to Runtime Security

In this chapter, we introduce the security features provided by the .NET runtime that most directly affect the code you develop and run. At a high level, we explain the purpose and function of these security features, how they interact with each other, and their relationship with the security provided by the underlying operating system. In the next six chapters, we explore these features in detail, demonstrate how to use them in your own programs, and show you how to extend and customize them.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

5.1 Runtime Security Explained

The .NET Framework is a flexible general-purpose computing platform designed to address the needs of commercial organizations and individuals alike, and supports many different application models. However, .NET places an emphasis on supporting the trend towards highly distributed systems, component-based applications, and web-based server solutions, including XML web services.

The .NET Framework is designed to run on multiple operating system platforms. Although Windows is currently the platform of choice for .NET deployment, there are implementations available for FreeBSD, Linux, and Mac OS X that are compliant with ECMA Standard 335 (which defines the core functionality of the .NET Framework). It is likely that in the future, .NET applications compliant with ECMA Standard 335 will run and integrate seamlessly across a variety of other operating systems.

The .NET Framework's strong support for distributed application models and its cross-platform capabilities means that software is more mobile, accessible, and integrated than before; although this provides functionality and productivity benefits, it also means that software consumers, producers, and service providers need to place a greater emphasis on software and system security. The increase in accessible web-based server solutions and mobile components also produces an increase in the number of people who try to hack those systems or who distribute malicious code in the form of viruses, worms, and Trojan horses for the purpose of gathering information or causing damage.

Cross-platform solutions cannot always rely on the underlying operating system to provide the same security services on all platforms. The .NET runtime implements the following three features that provide platform-independent security services to improve the security of both client- and server-side software solutions:

- Role-based security
- Code-access security
- Isolated storage

These features surround and permeate the .NET runtime to ensure that managed code does not perform actions and access resources it should not. They provide a baseline set of security, enforced by the .NET runtime, that managed code must comply with and can depend on regardless of the underlying operating system. These features do not replace existing operating system security measures; they either augment or compliment existing platform security, and mean that the .NET Framework provides one of the most secure and flexible environments in which to run distributed software solutions.

5.1.1 Running Unverifiable and Native Code

Runtime security works because the CLR is a managed execution environment that goes to great lengths to ensure that the code it loads is type-safe, meaning that it does not perform illegal operations, access memory directly, or try to access type members incorrectly. We described the

validation and verification process used to determine type safety in [Chapter 4](#). Code that is not type-safe is referred to as unverifiable code.

During execution, the .NET runtime, class libraries, and all required application assemblies are loaded to the same operating system process, albeit across different application domains. If type safety is not enforced, malicious code could manipulate the state of the runtime's security system, enabling it to access resources and carry out operations to which it would not normally have permission.

Another common risk to runtime security is the use of native code. Clearly, for the purpose of backward compatibility and to enable access to functionality not yet provided by the .NET class library, managed code must be able to call native code, such as Win32 DLLs and COM objects. However, once native code is running, the .NET runtime has no control over the actions the code performs; malicious native code can delete important files or even change the .NET runtime's security configuration files, which are stored as XML. Only the underlying operating system security can control the actions of unmanaged code, and this is not always sufficient because it bases security decisions on the identity of the current user.

Despite the risks, the .NET runtime designers understood that there are situations when it is necessary to use both native and unverifiable code. However, the runtime restricts access to these capabilities using its code-access security mechanism. Permission to load unverifiable code or call native code are two of the highest trust permissions you can grant to code. Once you grant these permissions to code, you are placing the security of your system at risk. You should have the utmost trust in the source and integrity of code before granting it these capabilities and must be confident that your operating system security is sufficient to protect your machine against attack. We discuss the permission to load unverifiable code and to run native code in [Chapter 7](#).

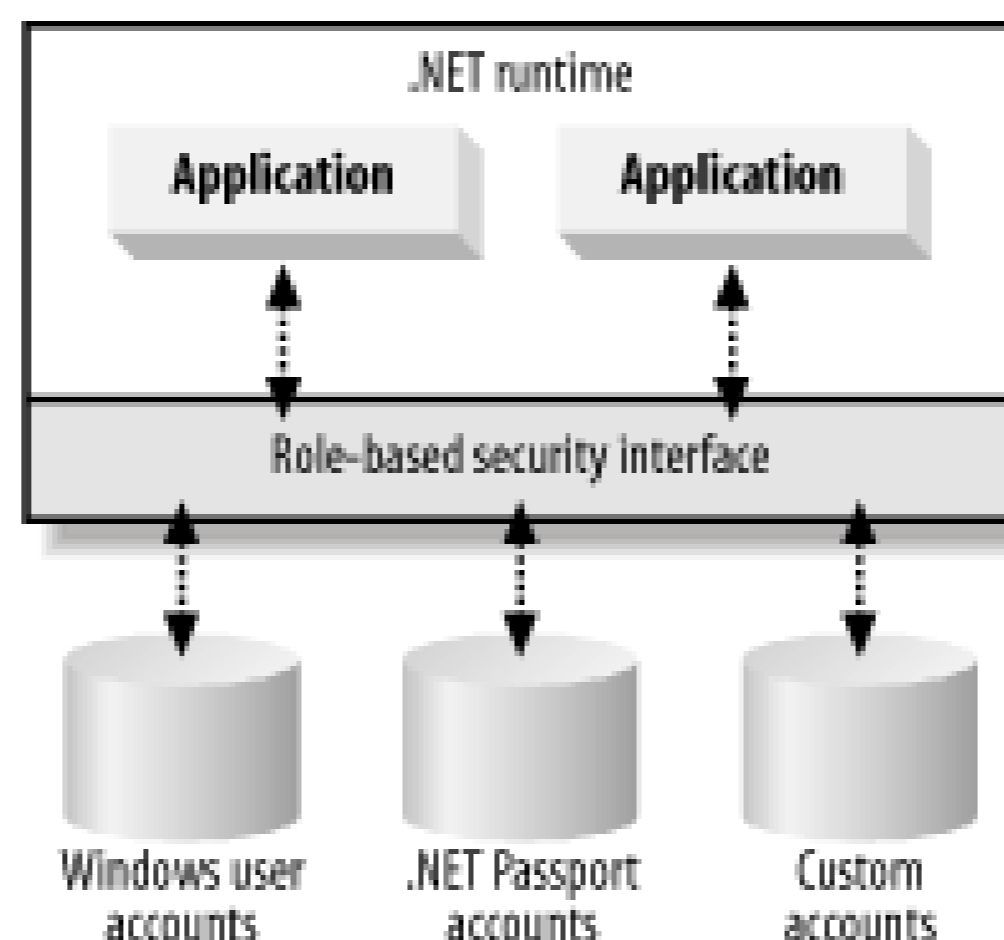
[\[Team LiB \]](#)

[\[Team LiB \]](#)

5.2 Introducing Role-Based Security

The .NET Framework provides a generic role-based security mechanism to represent the identity and roles of the user on whose behalf code is running. As illustrated in [Figure 5-1](#), you can use .NET's role-based security mechanism to integrate with an existing user account system, such as that provided by Windows or Microsoft's Passport .NET authentication mechanism. However, it is just as easy to integrate with other custom user account mechanisms.

Figure 5-1. Role-based security



Regardless of the underlying source of user information, .NET's role-based security interfaces provide a standard mechanism through which you can make runtime security decisions based on the identity and roles of a user. For example, you can make decisions, such as:

- Ensure that only users who are members of the "Administrators" or "Managers" roles can execute a protected class member
- Ensure that only the user "Peter" can load a class that inherits from a protected class

.NET's abstraction of the role-based security interfaces from the underlying authentication and authorization mechanisms make it relatively easy to change from one mechanism to another should the need arise. We provide a detailed coverage of .NET's role-based security mechanisms in [Chapter 10](#).

[\[Team LiB \]](#)

[\[Team LiB \]](#)

5.3 Introducing Code-Access Security

An important goal of the .NET Framework is to facilitate the development of highly distributed, component-based systems. With the .NET Framework, you can easily create applications that utilize code from different publishers, dynamically loading assemblies from different locations as required. Internet Explorer is an early example of this flexible architecture: downloading and executing .NET controls on demand, and giving the user a rich interface to the Web. Over the coming years, you can expect to see many more .NET applications componentized and delivered "on demand."

Traditionally, code executes using the identity, roles, and permissions of the user that runs it. As we discussed in the previous section, role-based security is important for ensuring system security. However, in a connected world where distributed systems and highly mobile code are the order of the day, it is insufficient to base system security decisions solely on the permission granted to the user running an application. Faulty code can damage files to which the user has access, or, worse, malicious code can take advantage of the user's authority to perform all kinds of mischief. A highly trusted user (with extensive system permissions) cannot freely download and run an application from an untrusted source for fear of what the code may do to his system and other systems on his network.

Many companies tackle the problem of code security by "locking down" their users' desktops, restricting the applications that people can install and run. This heavy-handed approach requires extensive engineering and support resources to maintain and reduces the utility and flexibility of each user's computer.

The .NET Framework includes an important new feature known as *code-access security (CAS)*, which provides fine-grained control over the operations and resources to which managed code has access. With CAS, access is based on the identity of the code, not the user running it. Some examples of operations protected by CAS include:

- Creating, deleting, and modifying files and directories
- Reading from and writing to the Windows registry
- Using sockets to accept or initiate a network connection
- Creating new application domains
- Running native (unmanaged) code

Other pre-.NET technologies have offered code-level security by differentiating between the identity of code from the user running it. These technologies include Java with its "sandbox" approach and Windows Internet Zones, which restrict the functionality of downloaded controls. However, few of these technologies provide the level of control, flexibility, and extensibility of CAS.

5.3.1 Evidence, Security Policy, and Permissions

The three key elements of code-access security are *evidence*, *security policy*, and *permissions*. When the runtime loads an assembly, it inspects various characteristics of the assembly (evidence) to determine an identity for the code. Based on a configurable set of rules (security policy), the runtime uses the assembly's evidence as input to a process named *policy resolution*. The result of policy resolution is a set of protected operations and resources to which the code within the assembly has access (permissions).

In the course of execution, code attempts to perform a variety of operations. If the code attempts to perform a protected operation, then the runtime will look at the permissions granted to the assembly containing the code and determine if they include the permission necessary to allow the action to go ahead. [Figure 5-2](#) illustrates the relationship between evidence, security policy, and permissions during policy resolution. Here is a detailed account of each element:

Evidence

Evidence includes the characteristics of an assembly used by the runtime to establish an identity for the assembly. Establishing an identity for each assembly allows the runtime to apply security policy in order to determine the set of permissions to grant to the assembly. Evidence includes characteristics, such as the strong name of an assembly, the web site from where an assembly is loaded, or the publisher certificate used to sign an assembly. We discuss evidence and code identity in [Chapter 6](#).

Security policy

Security policy is a configurable set of rules that the runtime uses during policy resolution to determine which permissions to grant to an assembly by evaluating the assembly's evidence. For example, security policy could dictate that all code downloaded from the web site <http://www.oreilly.com> has permission to write to the *C:\Test.txt* file, or that only code signed with Microsoft's publisher certificate can run unmanaged code. We discuss security policy in [Chapter 8](#).

Permissions

Permissions represent the authority of an assembly's code to access protected operations and resources. Permissions can be very specific, such as permission to open a network connection to the web site <http://www.oreilly.com>, or very broad, such as permission to read and write anywhere on the hard disk. We discuss permissions in [Chapter 7](#).

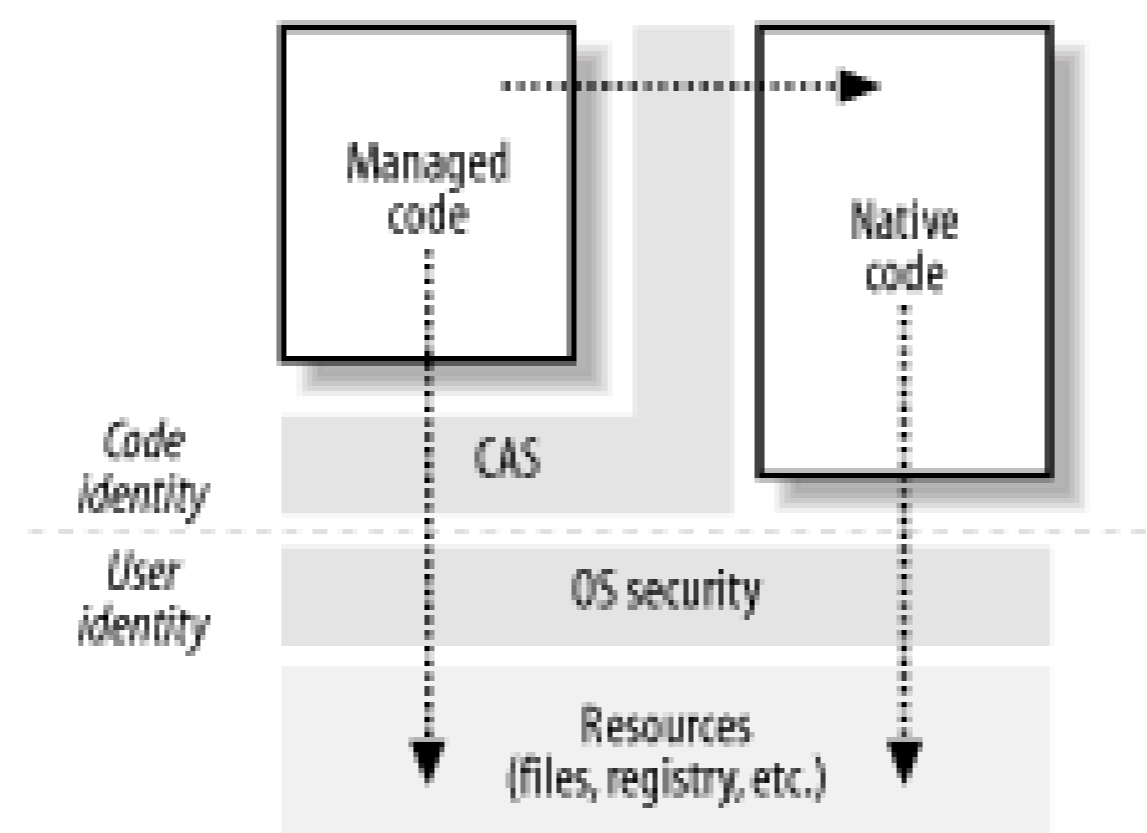
Figure 5-2. Evidence, security policy, and permissions

5.3.2 Windows Security and Code-Access Security

CAS does not replace or circumvent the security provided by the Windows operating system. As shown in [Figure 5-3](#), CAS is effectively another layer of security that managed code must pass

through before it can interact with protected system resources, such as the hard disk and the Windows registry. The important difference between CAS and Windows security is that CAS bases security decisions on the identity of the code performing an action, whereas Windows bases security decisions on the identity of the user on whose behalf the code is running.

Figure 5-3. Relationship of CAS and OS security



For example, if a managed application tries to write a file to the hard disk, CAS first determines if the code has the necessary authority to write to the file by evaluating the permissions granted to the assembly when it was loaded. If the code does not have the necessary authority, then the write operation fails and the runtime throws a security exception. On the other hand, if the code does have permission, the runtime interacts with the operating system on behalf of the current user to access the file. If Windows file permissions do not allow the user to write to the specified file, then access is denied and the runtime throws an exception. If the user has write permission, then the file is written.

As [Figure 5-3](#) also shows, CAS controls the ability of managed code to call unmanaged (native) code, such as the Win32 APIs. This is important, because when a managed application is allowed to call into unmanaged code, there is no CAS layer to restrict what the unmanaged code can do on behalf of the managed code. Instead of having permissions based on code identity, you are back to the situation where only the permissions of the current user restrict the code's actions. As we discussed at the beginning of this section, this is not sufficient when dealing with mobile code from potentially unknown and untrusted sources.

Finally, CAS also controls access to functionality that is internal to the CLR and does not affect operating system resources, such as the ability to create an application domain. We discuss CAS in greater detail in [Chapter 6](#) through [Chapter 9](#).

[\[Team LiB \]](#)

[\[Team LiB \]](#)

5.4 Introducing Isolated Storage

Many applications need to store state or user preferences so that they are available the next time the application runs. Windows applications usually store this type of data in configuration files, the Windows registry, or external databases.

For regularly used applications that are trusted, you will configure CAS security policy to give the application access to the required resource so that it can store this data successfully. However, because your hard disk, registry, and databases contain important information, it is unlikely that you will want to configure CAS to allow unknown and untrusted code to access them. Nor do you want to have to reconfigure security policy every time you install a new application or download mobile code from the Internet.

Isolated storage provides a safe storage area on the hard disk where managed code can store noncritical data. The .NET runtime manages isolated storage to ensure that code cannot access other code's data and to limit the space used by each application. Most importantly, because the interfaces and CAS permissions that control access to isolated storage are different from those of normal file I/O, the risk of malicious code using isolated storage as a gateway through which to access other data on the hard drive is minimized. [Figure 5-4](#) illustrates two managed applications that store data to the hard drive.

Figure 5-4. Isolated storage

The developer of an application must decide to use isolated storage; it is not something you can force any application to use.

In [Figure 5-4](#), Application A accesses its data through the .NET Framework's isolated storage mechanism, while Application B accesses the hard disk directly. We Provide a detailed discussion of isolated storage in [Chapter 11](#).

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 6. Evidence and Code Identity

Computerized security systems often use biometric characteristics, such as fingerprints, retina patterns, and keyboard dynamics to establish the identity of a person. In an analogous manner, .NET establishes the identity of an assembly based on characteristics derived from the assembly's content, structure, and source location. .NET refers to these identifying characteristics as *evidence*, and uses them to determine the actions and resources that code in the assembly has permission to access.

Our discussion of evidence begins with an explanation of what it is and where it comes from. We describe the purpose and use of the different types of evidence, including the standard evidence classes provided with the .NET Framework. Then we demonstrate how to use evidence programmatically in order to control the access permissions of your code. Finally, we show you how to extend CAS by developing custom evidence classes.

Evidence is most commonly used to determine the permissions to grant to an assembly but also plays an important role in securing application domains. To simplify our discussion, we focus on using evidence with assemblies. In almost all instances, the techniques we discuss are applicable without change to application domains, and we highlight where this is not the case.

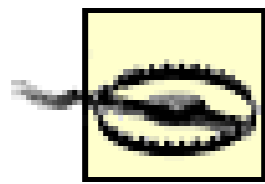
[\[Team LiB \]](#)

[[Team LiB](#)]

6.1 Evidence Explained

Some types of evidence are inherent in the structure and content of an assembly, such as the assembly strong name, the hash value of the assembly's content, or the publisher certificate used to sign the assembly. Other types of evidence are determined at runtime, based on characteristics such as the web site or URL from where the assembly is loaded.

The combined set of evidence establishes an *identity* for the assembly. When the runtime loads an assembly, it compares the assembly's identity with the rules defined in the security policy to determine the permissions to grant to the assembly, called the *grant set*. The grant set defines the operations that the code contained in the assembly can perform. We discuss permissions in [Chapter 7](#) and security policy in [Chapter 8](#).



It is important to understand that the same assembly can have very different identities (and hence permissions) based on runtime evidence. If you run an assembly from an Internet site, its evidence will be different than if you first download the assembly to your hard disk and run it locally. We demonstrate this in "Viewing Evidence" later in this chapter.

Application domains also have an identity based on evidence. As with assemblies, the runtime uses evidence and security policy to determine the application domain's grant set. The grant set of an application domain has an important but indirect effect on the operations and resources accessible to all of the assemblies running in it, which we discuss in [Chapter 7](#).

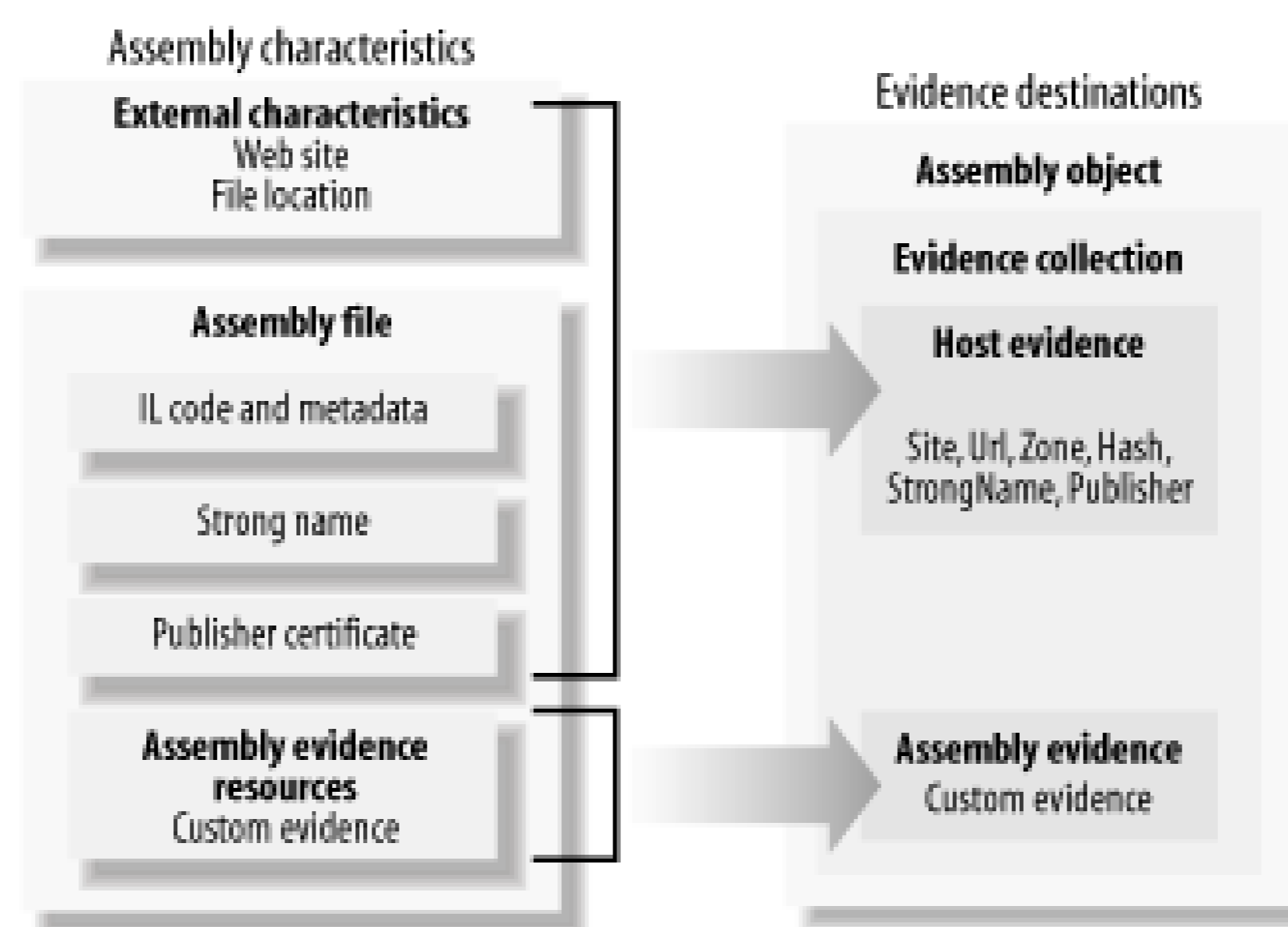
6.1.1 Sources and Types of Evidence

There are two types of evidence: *host evidence* and *assembly evidence*. When the runtime loads an assembly, it instantiates objects to represent each piece of evidence and associates the set of objects with the assembly as host evidence. An assembly can contain additional evidence, embedded at build time. As the runtime loads the assembly, it extracts this additional evidence and associates it with the assembly as assembly evidence.

The purpose of both types of evidence is to give the assembly an identity, which the runtime uses to determine the permissions to grant to it. However, the default behavior of the .NET runtime is to ignore assembly evidence; only through the implementation of CAS extensions can you enable assembly evidence to drive the policy resolution process. We demonstrate how to enable assembly evidence in [Section 6.3](#).

[Figure 6-1](#) shows the key characteristics found in an assembly characteristics and how the runtime uses each to create either host and assembly evidence. The following two sections discuss the nature and purpose of host and assembly evidence in more detail.

Figure 6-1. Sources and destinations of evidence



6.1.1.1 Host evidence

The host, in conjunction with the assembly loader component of the CLR, provides the host evidence for an assembly; the host could be one of the standard .NET runtime hosts (command shell, Internet Explorer, or ASP.NET), a custom runtime host, or a trusted piece of managed code that explicitly loads an assembly at runtime. The assembly loader provides a default set of evidence, but evidence provided by the host takes precedence if it conflicts with the default evidence.

A host's ability to provide and override evidence is extremely powerful; essentially the host can give any identity to an assembly. With this ability, malicious code could manipulate the evidence of the assemblies it loads in order to make the runtime assign them artificially high levels of trust. Therefore, .NET protects the ability to provide host evidence with the `ControlEvidence` permission—an element of the `System.Security.Permissions.SecurityPermission` permission, which we discuss in [Chapter 7](#).

A class in the `System.Security.Policy` namespace represents each of the seven standard types of host evidence that the .NET Framework supports. Not every loaded assembly has all types of evidence; an assembly's evidence depends on both its characteristics and the characteristics of the host that loaded it. [Table 6-1](#) lists the standard evidence types. We take a detailed look at each evidence class later in [Section 6.2.3](#).

Table 6-1. Standard .NET evidence types

Evidence class	Description
<code>ApplicationDirectory</code>	Represents the root directory of an application. <code>ApplicationDirectory</code> is used in conjunction with <code>Url</code> evidence during policy resolution, which we discuss in Chapter 8 .
<code>Hash</code>	Represents the hash value of the assembly contents. The runtime provides <code>Hash</code> evidence for all assemblies.

Evidence class	Description
<code>Publisher</code>	Represents the Authenticode X509 digital certificate used to sign the assembly. The runtime only provides <code>Publisher</code> evidence for assemblies signed with a digital certificate.
<code>Site</code>	Represents the host portion of the URL from which the assembly was loaded. The runtime provides <code>Site</code> evidence for all assemblies loaded from non- <i>file://</i> protocol URLs.
<code>StrongName</code>	Represents the strong name of the assembly. The runtime provides <code>StrongName</code> evidence only when the assembly has a strong name.
<code>Url</code>	Represents the URL from which the assembly was loaded. The runtime provides <code>Url</code> evidence for all assemblies. For assemblies loaded from the local disk, the URL protocol will be <i>file://</i> whereas assemblies downloaded from a web site have the protocol <i>http://</i> or <i>https://</i> .
<code>Zone</code>	Represents the Internet Explorer security zone from where the assembly was loaded. The runtime provides <code>Zone</code> evidence for all assemblies.

In addition to the classes listed in [Table 6-1](#), custom hosts and managed code that dynamically load assemblies can provide any type of object as host evidence; but unless you extend the policy resolution mechanism to understand the custom evidence classes, the runtime will ignore them. We demonstrate the creation of custom evidence in [Section 6.3](#).

6.1.1.2 Assembly evidence

The purpose of assembly evidence is to allow you to serialize custom evidence objects and embed them into your assembly at build time. When the runtime loads your assembly, it extracts and deserializes the embedded objects and assigns them to your assembly as assembly evidence.

You are free to use assembly evidence to represent any additional assembly characteristics you require; the example we use later in "Extending the .NET Framework" identifies the programmer who created the assembly.

You can use any type of serializable object as assembly evidence. However, you cannot use any of the standard evidence types listed in [Table 6-1](#); the runtime will ignore them when it loads the assembly.

As we mentioned earlier, the default configuration of the .NET Framework is to ignore assembly evidence. With some customization of CAS, assembly evidence can affect the resolution of *code-access* permissions in the same way as the standard evidence classes. However, the runtime never grants identity permissions to your assembly based on assembly evidence; only host evidence results in the granting of identity permissions. This does limit the usefulness of assembly evidence, but is necessary to stop malicious code from manipulating CAS and granting itself identity permissions in an attempt to increase its level of trust. We discuss code-access and identity permissions more in [Chapter 7](#).

[\[Team LiB \]](#)

[\[Team LiB \]](#)

6.2 Programming Evidence

In the normal course of application development, you have little need to work directly with evidence; the default evidence supplied by the runtime is sufficient to correctly drive the policy resolution process. If you do need to base runtime decisions on the evidence of an assembly, you'll most likely do so using identity permissions.

In the following sections, we show you how to use the advanced .NET support for directly manipulating evidence, which you will need when performing any of the following tasks:

- Dynamically loading or creating assemblies at runtime
- Creating application domains in which to isolate assemblies
- Developing a custom runtime host
- Developing custom evidence classes

6.2.1 Introduction to Evidence Programming

The runtime establishes the identity and grant set of an assembly when the assembly is loaded. You cannot alter the identity of an already loaded assembly to change the permissions granted to it. Therefore, you must provide all evidence at the time you load the assembly. Note that the runtime loads each assembly only once into a given application domain. Subsequent load requests result in the runtime using the already loaded assembly. If you try to reload an already loaded assembly but alter its identity by specifying additional items of evidence, the runtime will throw a `System.IO.FileLoadException`.

To provide evidence when you load an assembly, your code must have the `ControlEvidence` permission.

If you do not have the `ControlEvidence` permission, the runtime will throw a `System.Security.SecurityException` when you try to load the assembly. You can use *permission requests* to ensure that the runtime will not attempt to execute your code unless it has the `ControlEvidence` permission, or you can simply attempt to load your assembly and be prepared to handle the `SecurityException` that will occur if you do not have it.

In order to provide your evidence, you must create the set of evidence objects and add them to a `System.Security.Policy.Evidence` collection. You then pass the `Evidence` collection as an argument to the method that loads the assembly. Your evidence takes precedence over the default evidence provided by the assembly loader. When specifying the evidence, you can provide both host and assembly evidence. However, given the limitations of assembly evidence outlined earlier in [Section 6.1.1](#), there is little value in providing anything but host evidence, and so that is where we

will focus our discussions.

You can add as many evidence types as you want to the `Evidence` collection, but only the last one of each type affects the policy resolution process. If we add two pieces of `Url` evidence to an `Evidence` collection, the first being `http://www.oreilly.com` and the second being `http://www.somecompany.com`, then the permissions for the assembly will be based on the `http://www.somecompany.com` evidence.

6.2.2 Using the Evidence Class

The `System.Security.Policy.Evidence` class is a specialized collection that contains two sets of evidence objects: a collection of host evidence and a collection of assembly evidence. The `Evidence` class has three constructors, as shown in the following declarations:

```
# C#

public Evidence( );

public Evidence(Evidence evidence);

public Evidence(
    object[] hostEvidence,
    object[] assemblyEvidence
);

# Visual Basic .NET

Public Sub New( )

Public Sub New(ByVal evidence As Evidence)

Public Sub New(
    ByVal hostEvidence( ) As Object, _
    ByVal assemblyEvidence( ) As Object _
)
```

The first constructor creates an empty `Evidence` object. The second constructor takes an existing `Evidence` object and makes a copy of it. Note that this is a *shallow* copy, meaning that the new `Evidence` collection references the same evidence objects contained in the source `Evidence` collection; no new evidence objects are created. This is not an issue when using the standard evidence types because they are immutable. The third constructor takes two `Object` arrays: the `hostEvidence` argument contains the evidence objects for the host evidence collection and the `assemblyEvidence` argument contains the evidence objects for the assembly evidence collection.

The `Evidence` class implements the `System.Collections.ICollection` collection to facilitate the manipulation of the evidence objects it contains. In addition, `Evidence` implements members that support direct manipulation of the host and assembly evidence subcollections. [Figure 6-2](#) illustrates the structure of the `Evidence` class and shows the key methods you use to work with contained evidence elements.

Figure 6-2. Structure of the Evidence class

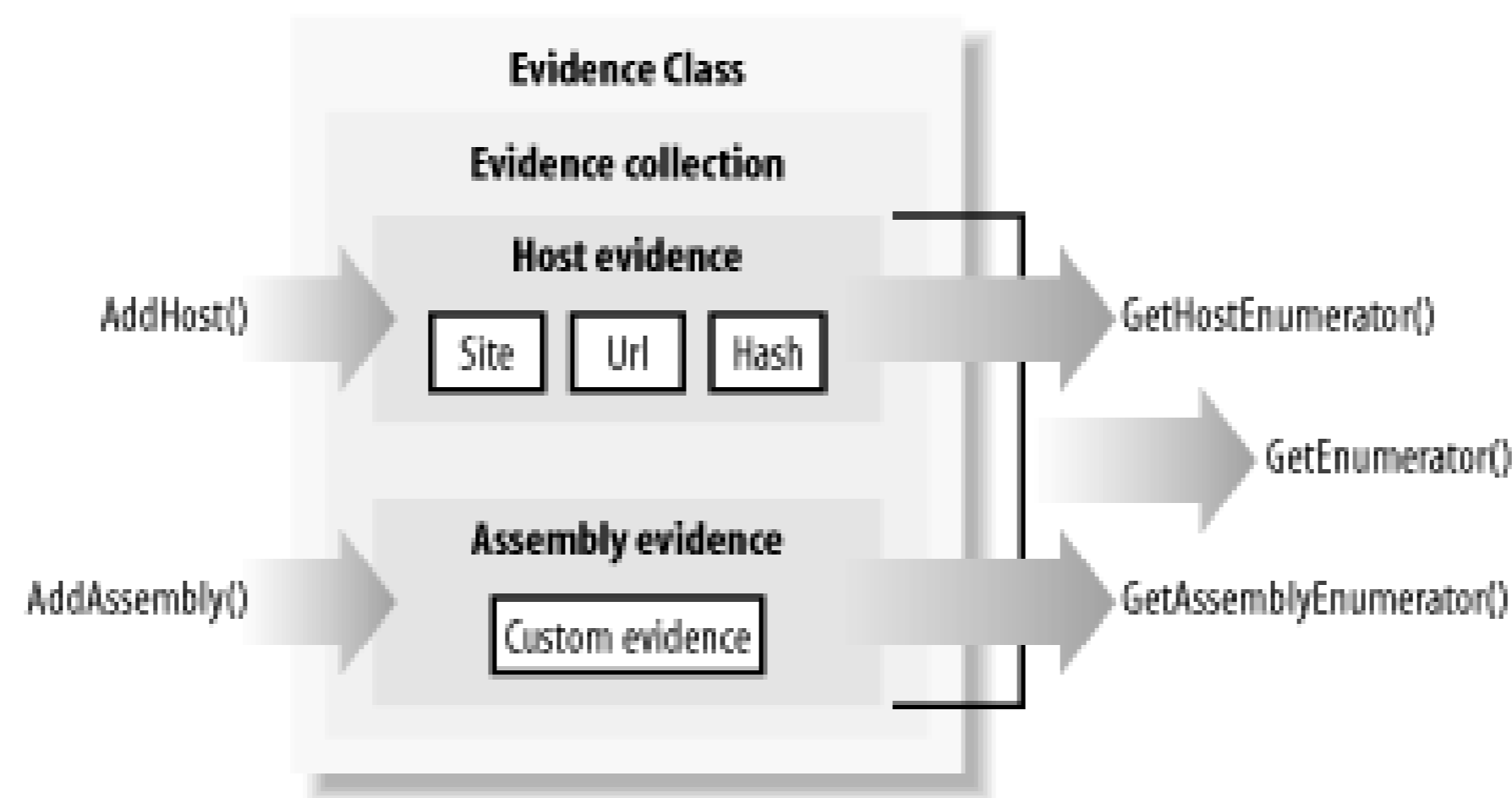


Table 6-2 summarizes the members of the `Evidence` class; the members that require your code to have the `ControlEvidence` permission are indicated in the far right column titled "C".

Table 6-2. Members of the Evidence class

Member	Description	C
Properties		
<code>Count</code>	Gets the total number of evidence objects in the <code>Evidence</code> collection.	
<code>IsReadOnly</code>	Always returns <code>false</code> because .NET does not (currently) support read-only evidence sets.	
<code>Locked</code>	Gets or sets the value indicating whether <code>Evidence</code> is locked. <code>ControlEvidence</code> is required to set this property to <code>false</code> .	
Methods		
<code>AddAssembly</code>	Adds an evidence object to the assembly evidence subcollection.	
<code>AddHost</code>	Adds an evidence object to the host evidence subcollection. Requires the caller to have the <code>ControlEvidence</code> permission when <code>Locked</code> is <code>true</code> .	
<code>CopyTo</code>	Copies all of the contained evidence objects to an array.	
<code>GetAssemblyEnumerator</code>	Returns an <code>IEnumerator</code> for enumerating across the assembly evidence subcollection.	
<code>GetEnumerator</code>	Returns an <code>IEnumerator</code> for enumerating across the entire evidence collection.	
<code>GetHostEnumerator</code>	Returns an <code>IEnumerator</code> for enumerating across the host evidence subcollection.	

Member	Description	C
<code>Merge</code>	Merges the specified <code>Evidence</code> collection into the current collection. Requires the caller to have the <code>ControlEvidence</code> permission when <code>Locked</code> is <code>true</code> and the <code>Evidence</code> collection contains host evidence.	

6.2.3 Using the Standard Evidence Classes

The evidence objects you will most commonly use to define the identity of your assemblies are the seven standard evidence classes from the `System.Security.Policy` namespace:

- `ApplicationDirectory`
- `Hash`
- `Publisher`
- `Site`
- `StrongName`
- `Url`
- `Zone`

Each of these classes is relatively simple and independent; they do not extend a common base class (other than `System.Object`), nor do they implement a common interface. However, their designs all follow a similar pattern, resulting in the following commonalities:

- All of the classes are marked as `sealed` in C#, or `NotInheritable` in Visual Basic .NET, ensuring that malicious code cannot subclass from them to manipulate CAS.
- All classes are serializable through the application of the `System.SerializableAttribute` attribute (the `Hash` class also implements the `System.Runtime.Serialization.ISerializable` interface). Being serializable is a key requirement of evidence classes.
- The content of all of the classes is immutable once it is created.
- All of the evidence classes except `Hash` implement a `Copy` method that creates a clone of the evidence object.
- Most of the evidence classes override the following members inherited from `System.Object` to base the result on the content of the evidence object:

`Equals`

All evidence classes except the `Hash` class perform equality evaluation based on the value of the evidence and not on object references.

`GetHashCode`

All evidence classes except the `Hash` class return a hash code based on the value of the

evidence.
ToString

All evidence classes override `ToString` to return a string containing an XML representation of the evidence object's content.

The following sections discuss each of the evidence classes in detail and demonstrate how to use them.

6.2.3.1 The ApplicationDirectory class

The `System.Security.Policy.ApplicationDirectory` class contains a URL that represents the root installation directory of an application. `ApplicationDirectory` evidence is unique amongst the standard evidence classes in that the runtime does not evaluate it in isolation. If an assembly has `ApplicationDirectory` evidence, the runtime also looks for `Url` evidence to determine if the `Url` evidence represents a child location specified in the `ApplicationDirectory` evidence. This will become clearer when we discuss membership conditions and policy resolution in [Chapter 8](#).

The `ApplicationDirectory` class constructor shown takes a `System.String` argument specifying the URL of the application root:

```
# C#

public ApplicationDirectory(String name);

# Visual Basic .NET

Public Sub New(ByVal name As String)
```

The read-only `Directory` property returns the URL string used in construction.

6.2.3.2 The Hash class

The `System.Security.Policy.Hash` class provides evidence based on a hash value of an assembly. A *hash* is a numeric value calculated using cryptographic algorithms that take the bytes contained in the assembly file as input. The nature of hashing algorithms means that it is very unlikely for the contents of two assemblies to result in the same hash value; even changing a single bit will result in a very different hash value. This means that a hash value provides a unique identity for an assembly (similar to a fingerprint). You can use a hash value to distinguish between two different assemblies, but by itself, a hash does not guarantee the source or integrity of an assembly; this is the purpose of strong names and publisher certificates (see [Chapter 2](#)). We provide a detailed discussion of hashing in [Chapter 13](#).

The `Hash` class is different from other evidence classes in that it provides multiple evidence values (hash values) from a single set of source data. The `Hash` class constructor takes a `System.Reflection.Assembly` object that identifies the assembly from which to generate hash values:

```
# C#
```

```
public Hash(Assembly assembly);
```

```
# Visual Basic .NET
```

```
Public Sub New(ByVal assembly as Assembly)
```

By default, the `Hash` class supports two hashing algorithms: MD5 and SHA1. The read-only `MD5` property returns an MD5 hash value based on the assembly content, and the read-only `SHA1` property does the same but using the SHA1 algorithm.

In addition, the `GenerateHash` method allows you to compute an assembly's hash using other hashing algorithms. The `GenerateHash` method takes a `System.Security.Cryptography.HashAlgorithm` argument containing an object that implements the required hashing algorithm. There are a number of hashing-algorithm implementations provided in the .NET class library (see [Chapter 13](#)).

The following code demonstrates the use of `Hash` to calculate the MD5, SHA1, and SHA512 hash values for an assembly. We use the `System.Security.Cryptography.SHA512Managed` class to provide the SHA512 hashing implementation:

```
# C#
```

```
using System.Reflection;
```

```
using System.Security.Policy;
```

```
using System.Security.Cryptography;
```

```
public class HashEvidenceDemo {
```

```
    public static void Main( ) {
```

```
        // Create a new Hash object for the currently executing
        // assembly.
```

```
        Hash hashEvidence = new Hash(Assembly.GetExecutingAssembly( ));
```

```
        // Use the Hash class to generate hash values for
        // the current assembly
```

```
        byte[] md5 = hashEvidence.MD5;
```

```
        byte[] sha1 = hashEvidence.SHA1;
```

```
        byte[] sha512 = hashEvidence.GenerateHash(new SHA512Managed( ));
```

```
    }
```

```
}
```

```
# Visual Basic .NET
```

```
Imports System.Security.Policy
```

```
Imports System.Security.Cryptography
```

```
Public Class HashEvidenceDemo
```

```
    Public Shared Sub Main( )
```

```
        ' Create a new Hash object for the currently executing
```



```

' assembly. Use fully qualified name of Assembly class
' to avoid conflict with Visual Basic .NET keyword.
Dim hashEvidence As Hash = _
    New Hash(System.Reflection.Assembly.GetExecutingAssembly( ))

' Use the Hash class to generate hash values for
' the current assembly
Dim md5( ) As Byte = hashEvidence.MD5
Dim sha1( ) As Byte = hashEvidence.SHA1
Dim sha512( ) As Byte = _
    hashEvidence.GenerateHash(New SHA512Managed( ))
End Sub
End Class

```



Unlike other evidence classes, the `Hash.ToString` method does not return a representation of the evidence value. Instead, it displays an XML representation of the source from which it calculates the hash values. This means that whenever you call `ToString`, you get the string representation of the bytes contained in your assembly-this is potentially very large.

6.2.3.3 The Publisher class

The `System.Security.Policy.Publisher` class provides evidence based on the Authenticode X.509 v3 digital certificate used to sign an assembly. The publisher (producer, author, or creator) of an assembly signs it to prove to consumers that they created the assembly. We discuss signing assemblies in [Chapter 2](#).

Because the `Publisher` class implements the `IIdentityPermissionFactory` interface, its presence in the *host* evidence collection of an assembly causes a `System.Security.Permissions.PublisherIdentityPermission` to be included in the assembly's grant set. Refer to [Chapter 7](#) for an explanation of identity permissions.

The `Publisher` class constructor shown here takes a `System.Security.Cryptography.X509Certificates.X509Certificate` object that contains the publisher's code-signing certificate. There are many simple ways to create an `X509Certificate` object, including loading it from a certificate file or extracting it from an existing signed file; refer to the .NET Framework SDK documentation for details:

```

# C#

public Publisher (X509Certificate cert);

# Visual Basic .NET

Public Sub New(ByVal cert As X509Certificate)

```

The read-only `Certificate` property returns the `X509Certificate` object provided during creation.

The following code demonstrates the creation of `Publisher` evidence using an X509 certificate

contained in a certificate file named *C:\MyCert.cer* and displays the certificate principal using the `GetName()` method of the `X509Certificate` object retrieved via the `Certificate` property:

```
# C#

using System;
using System.Security.Policy;
using System.Security.Cryptography.X509Certificates;

public class PublisherEvidenceDemo {

    public static void Main( ) {

        // Create a new X509Certificate object from the
        // test certificate file C:\MyCert.cer.
        X509Certificate cert =
            X509Certificate.CreateFromCertFile(@"C:\MyCert.cer");

        // Create a new Publisher evidence object from the
        // loaded X509Certificate object
        Publisher publisherEvidence = new Publisher(cert);

        // Use the Certificate property to get the publisher
        // certificate and display the principal name.
        Console.WriteLine(publisherEvidence.Certificate.GetName( ));
    }
}

# Visual Basic .NET

Imports System
Imports System.Security.Policy
Imports System.Security.Cryptography.X509Certificates

Public Class PublisherEvidenceDemo

    Public Shared Sub Main( )

        ' Create a new X509Certificate object from the
        ' test certificate file C:\MyCert.cer.
        Dim cert As X509Certificate = _
            X509Certificate.CreateFromCertFile("C:\MyCert.cer")

        ' Create a new Publisher evidence object from the
        ' loaded X509Certificate object
        Dim publisherEvidence As Publisher = New Publisher(cert)

        ' Use the Certificate property to get the publisher
        ' certificate and display the principal name.
        Console.WriteLine(publisherEvidence.Certificate.GetName( ))
    End Sub
End Class
```

6.2.3.4 The Site class

The `System.Security.Policy.Site` class provides evidence based on the fully qualified host name portion of the URL from where the assembly was loaded (for example, the site from the URL `https://www.somecompany.com:8080/dev/HelloWorld.exe` is `www.somecompany.com`).



Because the `Site` class implements the `IIdentityPermissionFactory` interface, its presence in the `host` evidence collection of an assembly causes a `System.Security.Permissions.SiteIdentityPermission` to be included in the assembly's grant set. Refer to [Chapter 7](#) for an explanation of identity permissions.

The `Site` class constructor shown takes a `System.String` argument containing the name of the site to use as evidence. After construction, the site name is available through the read-only `Name` property:

```
# C#

public Publisher (string name);

# Visual Basic .NET

Public Sub New(ByVal name As String)
```

The `Site` class also provides the static `CreateFromUrl` factory method to simplify the creation of `Site` objects from URL strings. The value of `siteEvidence.Name` in the following example is `www.somecompany.com`.

```
# C#

String url = "https://www.somecompany.com:8080/dev/HelloWorld.exe";
Site siteEvidence = Site.CreateFromUrl(url);

# Visual Basic.NET

Dim url As String = "https://www.somecompany.com:8080/dev/HelloWorld.exe"
Dim siteEvidence As Site = Site.CreateFromUrl(url)
```

The runtime does not provide `Site` evidence for assemblies loaded from file protocol URLs such as `file:///C:/Dev/HelloWorld.exe`. `CreateFromUrl` will actually throw a `System.ArgumentException` if you pass it such a URL.

6.2.3.5 The StrongName class

The `System.Security.Policy.StrongName` class provides evidence based on the strong name of an assembly. A strong name provides the assembly with a unique versioned identity and is a guarantee of the integrity of an assembly. However, a strong name does not guarantee the integrity of the source of an assembly, which is the purpose of publisher certificates. We discuss assembly strong names and publisher certificates in [Chapter 2](#).



Because the `StrongName` class implements the `IIdentityPermissionFactory` interface, its presence in the *host* evidence collection of an assembly causes a `System.Security.Permissions.StrongNameIdentityPermission` to be included in the assembly's grant set. Refer to [Chapter 7](#) for an explanation of identity permissions.

The `StrongName` class constructor takes three arguments:

- A `StrongNamePublicKeyBlob` object that represents the public key information for a strong name. The `StrongNamePublicKeyBlob` class is a simple utility class from the `System.Security.Permissions` namespace used to facilitate the passing of public key information. See the .NET Framework SDK documentation for complete descriptions
- A `System.String` object containing the name portion of the strong name.
- A `System.Version` object that contains the strong name's version. `Version` is a utility class that greatly simplifies the manipulation and comparison of assembly version numbers. See the .NET Framework SDK documentation for complete descriptions. Here is the code for the `StrongName` class constructor:

```
# C#

public StrongName (
    StrongNamePublicKeyBlob blob,
    string name,
    Version version
);

# Visual Basic .NET

Public Sub New( _
    ByVal blob As StrongNamePublicKeyBlob, _
    ByVal name As String, _
    ByVal version As Version _
)
```

Each of the arguments provided during construction is available through the read-only `Name`, `PublicKey`, and `Version` properties of the `StrongName` class.

To obtain the strong-name information required to create a `StrongName` object you can use the `Secutil` tool (*Secutil.exe*) to extract the public key, name, and version from a strong-named assembly. There is no simple way to extract this information using the .NET class library. The following command will extract the strong-name information from the assembly *HelloWorld.exe* and write it to the *strongname.txt* file in a format that you can easily incorporate into your program:

```
secutil -array -cmode -strongname HelloWorld.exe > strongname.txt
```

The contents of *strongname.txt* will look something like this:

```
Microsoft (R) .NET Framework SecUtil 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
```

```

Public Key =
{ 0, 36, 0, 0, 4, 128, 0, 0, 148, 0, 0, 0, 6, 2, 0, 0, 0, 36, 0, 0,
82, 83, 65, 49, 0, 4, 0, 0, 1, 0, 1, 0, 169, 206, 164, 8, 66, 197,
231, 138, 148, 74, 99, 125, 171, 203, 120, 143, 240, 155, 104, 138,
4, 123, 15, 55, 85, 255, 183, 20, 111, 10, 217, 58, 127, 15, 236, 86,
16, 121, 222, 35, 161, 14, 122, 246, 85, 226, 162, 221, 46, 215, 161,
151, 183, 38, 31, 150, 198, 119, 109, 94, 11, 65, 208, 33, 122, 172,
106, 62, 192, 4, 35, 255, 220, 10, 43, 90, 92, 183, 29, 136, 57, 235,
30, 5, 127, 72, 210, 108, 215, 226, 65, 197, 184, 28, 129, 184, 191,
211, 159, 69, 8, 84, 116, 65, 186, 179, 35, 116, 174, 223, 167, 217,
116, 8, 178, 232, 213, 155, 172, 87, 181, 187, 61, 43, 133, 105, 10,
187 }
Name =
HelloWorld
Version =
1.1578.0.0
Success

```

With the strong-name information extracted by `Secutil`, it is a simple matter to create `StrongName` evidence, as shown below:

```

# C#

using System;
using System.Security.Policy;
using System.Security.Permissions;

public class StrongNameEvidenceDemo {

    // Create a byte array containing the strong name public key
    // data.
    static byte[] publickey = { 0, 36, 0, 0, 4, 128, 0, 0, 148,
0, 0, 0, 6, 2, 0, 0, 0, 36, 0, 0, 82, 83, 65, 49, 0, 4, 0,
0, 1, 0, 1, 0, 169, 206, 164, 8, 66, 197, 231, 138, 148, 74,
99, 125, 171, 203, 120, 143, 240, 155, 104, 138, 4, 123, 15,
55, 85, 255, 183, 20, 111, 10, 217, 58, 127, 15, 236, 86, 16,
121, 222, 35, 161, 14, 122, 246, 85, 226, 162, 221, 46, 215,
161, 151, 183, 38, 31, 150, 198, 119, 109, 94, 11, 65, 208,
33, 122, 172, 106, 62, 192, 4, 35, 255, 220, 10, 43, 90, 92,
183, 29, 136, 57, 235, 30, 5, 127, 72, 210, 108, 215, 226, 65,
197, 184, 28, 129, 184, 191, 211, 159, 69, 8, 84, 116, 65, 186,
179, 35, 116, 174, 223, 167, 217, 116, 8, 178, 232, 213, 155,
172, 87, 181, 187, 61, 43, 133, 105, 10, 187 };

    public static void Main( ) {

        // Create a StrongNamePublicKeyBlob object from the
        // publickey byte array.
        StrongNamePublicKeyBlob blob =
            new StrongNamePublicKeyBlob(publickey);

        // Create a Version object based on the assembly version

```

```

        // number
        Version version = new Version("1.1578.0.0");

        // Create the new StrongName evidence object
        StrongName sn = new StrongName(blob, "HelloWorld", version);
    }
}

# Visual Basic .NET

Imports System
Imports System.Security.Policy
Imports System.Security.Permissions

Public Class StrongNameEvidenceDemo

    ' Create a byte array containing the strong name public key
    ' data.
    Shared publickey( ) As Byte = { 0, 36, 0, 0, 4, 128, 0, 0, 148, _
    0, 0, 0, 6, 2, 0, 0, 0, 36, 0, 0, 82, 83, 65, 49, 0, 4, 0, _
    0, 1, 0, 1, 0, 169, 206, 164, 8, 66, 197, 231, 138, 148, 74, _
    99, 125, 171, 203, 120, 143, 240, 155, 104, 138, 4, 123, 15, _
    55, 85, 255, 183, 20, 111, 10, 217, 58, 127, 15, 236, 86, 16, _
    121, 222, 35, 161, 14, 122, 246, 85, 226, 162, 221, 46, 215, _
    161, 151, 183, 38, 31, 150, 198, 119, 109, 94, 11, 65, 208, _
    33, 122, 172, 106, 62, 192, 4, 35, 255, 220, 10, 43, 90, 92, _
    183, 29, 136, 57, 235, 30, 5, 127, 72, 210, 108, 215, 226, 65, _
    197, 184, 28, 129, 184, 191, 211, 159, 69, 8, 84, 116, 65, 186, _
    179, 35, 116, 174, 223, 167, 217, 116, 8, 178, 232, 213, 155, _
    172, 87, 181, 187, 61, 43, 133, 105, 10, 187}

    Public Shared Sub Main( )

        ' Create a StrongNamePublicKeyBlob object from the
        ' publickey byte array.
        Dim blob As StrongNamePublicKeyBlob = _
            New StrongNamePublicKeyBlob(publickey)

        ' Create a Version object based on the assembly version
        ' number
        Dim version As Version = New Version("1.1578.0.0")

        ' Create the new StrongName evidence object
        Dim sn As StrongName = New StrongName(blob, "HelloWorld", version)

    End Sub

End Class

```

6.2.3.6 The Url class

The `System.Security.Policy.Url` class provides evidence based on the URL from where an assembly was loaded. If the assembly is loaded using a file path on the local machine, the `Url` evidence contains a `file://` protocol URL; other possibilities are `http://` and `https://` if the assembly was loaded from a web site.



Because the `Url` class implements the `IIdentityPermissionFactory` interface, its presence in the `host` evidence collection of an assembly causes a `System.Security.Permissions.UrlIdentityPermission` to be included in the assembly's grant set. Refer to [Chapter 7](#) for an explanation of identity permissions.

The `Url` class constructor shown takes a `System.String` argument specifying the URL from where the assembly was loaded:

```
# C#

public Url(String name);

# Visual Basic .NET

Public Sub New(ByVal name As String)
```

The `Url` constructor performs some validation on the URL string to ensure that `file://`-based URLs are correctly formatted but does not limit the scope of the protocol you can use.

The read-only `Value` property returns the URL string used in construction.

6.2.3.7 The Zone class

The `System.Security.Policy.Zone` class provides evidence based on the Internet Explorer security zone from where an assembly was loaded. The runtime maps the URL of an assembly against the Internet Explorer zone configuration to determine the type of `Zone` evidence to create.

Because the `Zone` class implements the `IIdentityPermissionFactory` interface, its presence in the `host` evidence collection of an assembly causes a `System.Security.Permissions.ZoneIdentityPermission` to be included in the assembly's grant set. Refer to [Chapter 7](#) for an explanation of identity permissions.

The `Zone` class constructor shown below takes as an argument a member of the `System.Security.SecurityZone` enumeration which specifies the type of `Zone` evidence to create; we list the possible values in [Table 6-3](#). The read-only `SecurityZone` property returns the `SecurityZone` member specified during construction:

```
# C#

public Zone (SecurityZone zone);

# Visual Basic .NET
```

```
Public Sub New(ByVal zone As SecurityZone)
```

Table 6-3. Members of the SecurityZone enumeration

Member	Description
<code>Internet</code>	All web sites that do not belong to one of the other zones
<code>Intranet</code>	Web sites located on the local intranet
<code>MyComputer</code>	The local computer
<code>NoZone</code>	No zone specified
<code>Trusted</code>	Specific web sites configured in Internet Explorer to be trusted
<code>UnTrusted</code>	Specific web sites configured in Internet Explorer to be untrusted

The `Zone` class also provides the static `CreateFromUrl` factory method to simplify the creation of `Zone` objects from URL strings. When you call `CreateFromUrl` the runtime maps the specified URL against the Internet Explorer zone configuration and returns the appropriate `Zone` object.

The following code demonstrates how to create `Zone` evidence representing the `Trusted` zone, as well as how to create a zone from the URL

```
https://www.somecompany.com:8080/dev/HelloWorld.exe.
```

```
# C#
```

```
String url = "https://www.somecompany.com:8080/dev/HelloWorld.exe";
Zone z1 = Zone.CreateFromUrl(url);
Zone z2 = new Zone(SecurityZone.Trusted);
```

```
# Visual Basic.NET
```

```
Dim url As String = "https://www.somecompany.com:8080/dev/HelloWorld.exe"
Dim z1 As Zone = Zone.CreateFromUrl(url)
Dim z2 As Zone = New Zone(SecurityZone.Trusted)
```

6.2.4 Viewing Evidence

The ability to view an assembly's evidence is extremely useful when debugging CAS-related programming issues and will help you to understand the concepts discussed previously. The `System.Reflection.Assembly` and `System.AppDomain` classes have read-only `Evidence` properties that contain the evidence CAS used during policy resolution to determine the assembly's or application domain's permissions. The `Evidence` property is read-only because, as we described earlier, you can only assign evidence when you load an assembly or create an application domain.

If you try to add new evidence objects to the `Evidence` collection of an existing `Assembly` or `AppDomain` object, no error will occur, but the evidence collection will not be modified.

It is unlikely that you will need to use an evidence in order to make security decisions at runtime. Most of the standard evidence classes result in the inclusion of an appropriate identity permission in the grant set of the assembly. Identity permissions, which we discuss in [Chapter 7](#), are tightly integrated with the CAS framework and allow you to base security decisions on the identity of an assembly.

[Example 6-1](#) shows how to display an assembly's evidence, using a class named `LoadAndList`, which you should build into an executable named `LoadAndList.exe`. The `LoadAndList` class takes the name of an assembly as a command-line argument, loads the specified assembly, and lists the assembly's host and assembly evidence to the console. The `LoadAndList` class uses the `Assembly.Evidence` property to get the assembly's evidence and uses two loops to enumerate first the host and then the assembly evidence:

Example 6-1. Displaying an assembly's evidence

```
# C#

using System;
using System.Reflection;
using System.Collections;
using System.Security.Policy;

public class LoadAndList {

    public static void Main(string[] args) {

        // Load the specified assembly
        Assembly a = Assembly.LoadFrom(args[0]);

        // Get the Evidence collection from the
        // loaded assembly
        Evidence e = a.Evidence;

        // Display the Host Evidence
        IEnumerator x = e.GetHostEnumerator( );
        Console.WriteLine("HOST EVIDENCE:");
        while(x.MoveNext( )) {
            Console.WriteLine(x.Current.ToString( ));
        }

        // Display the Assembly Evidence
        x = e.GetAssemblyEnumerator( );
        Console.WriteLine("ASSEMBLY EVIDENCE:");
        while(x.MoveNext( )) {
            Console.WriteLine(x.Current.ToString( ));
        }
    }
}

# Visual Basic .NET
```

```

Imports System
Imports System.Collections
Imports System.Security.Policy

Public Class LoadAndList

    Public Shared Sub Main(ByVal args( ) As String)

        ' Load the specified assembly
        Dim a As System.Reflection.Assembly = _
            System.Reflection.Assembly.LoadFrom(args(0))

        ' Get the Evidence collection from the
        ' loaded assembly
        Dim e As Evidence = a.Evidence

        ' Display the Host Evidence
        Dim x As IEnumerator = e.GetHostEnumerator( )
        Console.WriteLine("HOST EVIDENCE:")
        While x.MoveNext( )
            Console.WriteLine(x.Current.ToString( ))
        End While

        ' Display the Assembly Evidence
        x = e.GetAssemblyEnumerator( )
        Console.WriteLine("ASSEMBLY EVIDENCE:")
        While x.MoveNext( )
            Console.WriteLine(x.Current.ToString( ))
        End While
    End Sub
End Class

```

The `LoadAndList` utility provides you with the opportunity to demonstrate an important point we made earlier in this chapter: the identity of an assembly can vary greatly based on the location from which it is loaded, resulting in the assembly being granted a different set of permissions.

Instead of the path of an assembly file, pass `LoadAndList` the URL of an assembly to load. If you place an assembly in the root directory of a Microsoft Internet Information Services (IIS) server running on your local machine, the evidence assigned to the assembly will be very different if you access it using an HTTP URL as opposed to the a file path.

For example, if you take any assembly (use one named *HelloWorld.exe*) and place it in the root directory of the IIS server running on our Windows 2000 machine, executing the command `LoadAndList C:\Inetpub\wwwroot\HelloWorld.exe` results in the following display of host evidence (the `Hash` evidence is abbreviated):

```

HOST EVIDENCE:
<System.Security.Policy.Zone version="1">
  <Zone>MyComputer</Zone>
</System.Security.Policy.Zone>

<System.Security.Policy.Url version="1">

```

```

    <Url>file:///C:/Inetpub/wwwroot/HelloWorld.exe</Url>
</System.Security.Policy.Url>

<System.Security.Policy.Hash version="1">
  <RawData>4D5A90000300000004000000FFFF000</RawData>
</System.Security.Policy.Hash>

```

Alternatively, executing the command `LoadAndList http://localhost/HelloWorld.exe` results in a very different set of host evidence despite the fact that you are looking at the same assembly. Notice the changes (highlighted below) to the value of the `Zone` and `Url` evidence, and the addition of the new `Site` evidence. The `Hash` evidence (based on the content of the assembly) remains unchanged, as would `StrongName` and `Publisher` evidence if it were present. Notice that in both cases, the assembly evidence collection is empty:

HOST EVIDENCE:

```

<System.Security.Policy.Zone version="1">
  <Zone>Intranet</Zone>
</System.Security.Policy.Zone>

<System.Security.Policy.Url version="1">
  <Url>http://localhost/HelloWorld.exe</Url>
</System.Security.Policy.Url>

<System.Security.Policy.Site version="1">
  <Name>localhost</Name>
</System.Security.Policy.Site>

<System.Security.Policy.Hash version="1">
  <RawData>4D5A90000300000004000000FFFF000</RawData>
</System.Security.Policy.Hash>

```

6.2.5 Assigning Evidence to Assemblies

[Table 6-4](#) lists the most commonly used methods of the `AppDomain` and `Assembly` classes that directly or indirectly result in the runtime loading an assembly. Each of these methods has one or more overloads that allow you to provide an `Evidence` collection for the loaded assembly.

Table 6-4. AppDomain and Assembly methods that take Evidence collections

Method	Description
System.AppDomain	
<code>CreateInstance</code>	Instantiate a class contained in the specified assembly
<code>CreateInstanceAndUnwrap</code>	
<code>CreateInstanceFrom</code>	

Method	Description
<code>CreateInstanceFromAndUnwrap</code>	
<code>DefineDynamicAssembly</code>	Defines a dynamic assembly
<code>ExecuteAssembly</code>	Loads and executes the specified assembly
<code>Load</code>	Loads the specified assembly into the application domain
<code>System.Reflection.Assembly</code>	
<code>Load</code>	Load the specified assembly into the current application domain
<code>LoadFrom</code>	
<code>LoadWithPartialName</code>	

Although the purpose and signatures of the methods listed in [Table 6-4](#) vary, the steps necessary to supply evidence to a new assembly are the same:

1. Create an `Evidence` collection object.
2. Instantiate the individual evidence objects and add them to the `Evidence` collection using the `AddHost` method.
3. Execute the method to load the assembly, passing it the `Evidence` collection as an argument.

[Example 6-2](#) shows how to assign evidence to an assembly, using a modified version of the `LoadAndList` class (from [Example 6-1](#)) named `LoadAndAssign`. The `LoadAndAssign` class loads a named assembly, and assigns it new `Site`, `Url`, and `Zone` host evidence. Because we do not discuss the use of permission requests until [Chapter 7](#), we catch all exceptions and display a message to the user. If you execute `LoadAndAssign` from your local machine, there should be no problem with permissions as long as you have not modified the default security policy. However, you should always handle any security exceptions that you know might be thrown.

Example 6-2. Assigning evidence to an assembly

```
# C#

using System;
using System.Reflection;
using System.Collections;
using System.Security;
using System.Security.Policy;

public class LoadAndAssign {

    public static void Main(string[] args) {

        // Create a new Evidence collection
        Evidence e = new Evidence( );
    }
}
```

```

// Add host evidence to the Evidence collection
e.AddHost(new Site("www.oreilly.com"));
e.AddHost(new Url(@"file://C:/Dev/Test.exe"));
e.AddHost(new Zone(SecurityZone.Trusted));

// Create a new Assembly reference
Assembly a = null;

try {

    // Load the specified assembly and provide the Evidence
    // collection
    a = Assembly.LoadFrom(args[0], e);

} catch (Exception ex) {

    Console.WriteLine("Could not load assembly - {0}.", args[0]);
    Console.WriteLine(ex);
    return;
}

// Display the Host Evidence of the assembly
IEnumerator x = a.Evidence.GetHostEnumerator( );
Console.WriteLine("HOST EVIDENCE:");
while(x.MoveNext( )) {
    Console.WriteLine(x.Current.ToString( ));
}

// Display the Assembly Evidence of the assembly
x = a.Evidence.GetAssemblyEnumerator( );
Console.WriteLine("ASSEMBLY EVIDENCE:");
while(x.MoveNext( )) {
    Console.WriteLine(x.Current.ToString( ));
}
}
}

```

```
# Visual Basic .NET
```

```
Imports System
Imports System.Collections
Imports System.Security
Imports System.Security.Policy
```

```
Public Class LoadAndAssign
```

```
    Public Shared Sub Main(ByVal args( ) As String)
```

```
        ' Create a new Evidence collection
        Dim e As Evidence = new Evidence( )
```

```
        ' Add host evidence to the Evidence collection
```

```

    e.AddHost(new Site("www.oreilly.com"))
    e.AddHost(new Url("file://C:/Dev/Test.exe"))
    e.AddHost(new Zone(SecurityZone.Trusted))

    ' Create a new Assembly reference
    Dim a As System.Reflection.Assembly = Nothing

    Try
        ' Load the specified assembly and provide the Evidence
        ' collection
        a = System.Reflection.Assembly.LoadFrom(args(0), e)
    Catch ex As SecurityException
        Console.WriteLine("Could not load assembly - {0}.", args(0))
        Console.WriteLine(ex)
        Return
    End Try

    ' Display the Host Evidence of the assembly
    Dim x As IEnumerator = a.Evidence.GetHostEnumerator( )
    Console.WriteLine("HOST EVIDENCE:")
    While x.MoveNext( )
        Console.WriteLine(x.Current.ToString( ))
    End While

    ' Display the Assembly Evidence of the assembly
    x = a.Evidence.GetAssemblyEnumerator( )
    Console.WriteLine("ASSEMBLY EVIDENCE:")
    While x.MoveNext( )
        Console.WriteLine(x.Current.ToString( ))
    End While
End Sub
End Class

```

From the command line, entering the command `LoadAndAssign C:\Inetpub\wwwroot\HelloWorld.exe` to load the *HelloWorld.exe* assembly you used in [Example 6-1](#) results in the display of host evidence similar to that listed below. The `Url` and `Zone` evidence you specified has replaced the default evidence. There is also `Site` evidence, which was previously present only when you used an HTTP URL to specify the location of the assembly:

```

HOST EVIDENCE:
<System.Security.Policy.Hash version="1">
  <RawData>4D5A90000300000004000000FFFF000</RawData>
</System.Security.Policy.Hash>

<System.Security.Policy.Site version="1">
  <Name>www.oreilly.com</Name>
</System.Security.Policy.Site>

<System.Security.Policy.Url version="1">
  <Url>file://C:/Dev/Test.exe</Url>
</System.Security.Policy.Url>

```

```
<System.Security.Policy.Zone version="1">
  <Zone>Trusted</Zone>
</System.Security.Policy.Zone>
```

6.2.6 Assigning Evidence to Application Domains

Many of the standard evidence types are more applicable to assemblies than to application domains, but there is nothing to stop you assigning any of the evidence types of application domains.

For example, although it is impossible to sign an application domain with a publisher certificate, you could still assign `Publisher` evidence to an application domain manually. This may be appropriate on a server that runs code on behalf of many different companies. You may decide to run all code from a single company in its own application domain. Assigning the application domain `Publisher` evidence ensures that any code running within the application domain is limited to the permissions you assign to that particular company. The benefits of application domain evidence will become clearer when we discuss security demands and stack walks in [Chapter 7](#).

When you create an application domain using the static `AppDomain.CreateDomain` method, two of the method overloads give you the option of providing an `Evidence` collection for the new application domain:

```
# C#

public static AppDomain CreateDomain(
    string friendlyName,
    Evidence securityInfo,
    AppDomainSetup info
);

public static AppDomain CreateDomain(
    string friendlyName,
    Evidence securityInfo,
    string appBasePath,
    string appRelativeSearchPath,
    bool shadowCopyFiles
);

# Visual Basic .NET

Overloads Public Shared Function CreateDomain( _
    ByVal friendlyName As String, _
    ByVal securityInfo As Evidence, _
    ByVal info As AppDomainSetup _
) As AppDomain

Overloads Public Shared Function CreateDomain( _
    ByVal friendlyName As String, _
    ByVal securityInfo As Evidence, _
    ByVal appBasePath As String, _
    ByVal appRelativeSearchPath As String, _
    ByVal shadowCopyFiles As Boolean _
```

```
) As AppDomain
```

The process of specifying evidence for an application domain is the same as for assemblies except that you pass the `Evidence` collection populated with evidence objects to the `CreateDomain` method:

```
# C#

// Create an empty Evidence collection
Evidence e = New Evidence( );

// Add Host and Assembly Evidence
e.AddHost(new Site("www.oreilly.com"));
e.AddHost(new Zone(SecurityZone.Trusted));

// Create a new application domain reference.
AppDomain d = null;

try {

    // Create the new application domain
    d = AppDomain.CreateDomain("New Domain", e, null);

} catch (Exception ex) {

    Console.WriteLine("Could not create application domain");
    Console.WriteLine(ex);
}

# Visual Basic .NET

' Create an empty Evidence collection
Dim e As Evidence = New Evidence( )

' Add Host and Assembly Evidence
e.AddHost(New Site("www.oreilly.com"))
e.AddHost(New Zone(SecurityZone.Trusted))

' Create a new application domain reference.
Dim d As AppDomain = Nothing

Try

    ' Create the new application domain
    d = AppDomain.CreateDomain("New Domain", e, Nothing)

Catch ex As Exception

    Console.WriteLine("Could not create application domain")
    Console.WriteLine(ex)
End Try
```

Be aware of the following behavior:

- The runtime host determines the evidence assigned to the initial application domain. In the case of the shell runtime host, the default application domain is assigned the same set of evidence assigned to the assembly whose execution caused the CLR to be loaded.
- If you specify `null` (C#) or `Nothing` (Visual Basic .NET) as the `Evidence` argument to the `CreateDomain` method, the runtime assigns the new application domain the same evidence that is assigned to the current application domain.
- If you pass an empty `Evidence` collection to the `CreateDomain` method, the application domain will have no evidence. This does not mean that code running in the application domain has no permissions; in fact, it means the application domain is transparent to security permission checks (which means that the application domain has no effect on the permissions of the code running in it). See [Chapter 7](#) for details.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

6.3 Extending the .NET Framework

The standard .NET evidence classes represent the most commonly available and useful characteristics of an assembly. For most situations, these classes provide enough reliable information from which to determine a unique identity for an assembly, enabling you to configure your security policy. However, there are times when the systems you develop need to provide users, administrators, and programmers with additional criteria on which to base their security policy decisions. CAS supports the use of custom evidence to meet this requirement.

The creation of custom evidence is a simple task, but there are more steps before you can drive the policy resolution process using your custom evidence. In the following sections, we discuss the creation of a custom evidence class. We then continue our customization example with the development of custom membership conditions in [Chapter 8](#).

6.3.1 Creating Custom Evidence

You can use any serializable class as evidence. There are no evidence-specific interfaces to implement, nor do you need to derive from a common base class (other than `System.Object`). However, here are some guidelines to consider when implementing custom evidence classes:

- Make evidence simple. Evidence classes tend to be relatively simple classes that represent a single piece or type of information. Creating evidence classes that represent multiple or complex sets of data will result in security policy that is difficult for security administrators and users to configure and understand. If you need to represent multiple pieces of information, then create separate evidence classes for each.
- Make evidence lightweight. When the runtime loads an assembly, it must instantiate the objects that represent the assembly's evidence. These objects stay in memory until the assembly is unloaded or the runtime terminates. Consider the amount of memory used by your evidence classes as well as avoid lengthy processing tasks, such as network, file, and database access during instantiation.
- Make evidence serializable. Evidence classes must be serializable. Because of their simplicity, application of the `System.SerializableAttribute` is usually sufficient. Those evidence classes that have more complex serialization requirements must implement the `System.Runtime.Serialization.ISerializable` interface.

Unless you implement `ISerializable`, deserialization of an object does not result in the execution of a constructor, and you should not depend on constructor logic to configure or modify the state of the de-serialized object.

- Make evidence noninheritable. It is safest to mark your evidence classes as `sealed` in C# and `NotInheritable` in Visual Basic .NET so that malicious code cannot subclass your evidence

class in an attempt to change its behavior and subvert CAS.

- Don't assume default values. You should consider whether it is appropriate for your evidence class to assume default evidence values if the caller does not explicitly set them during creation. The approach taken by the standard evidence classes is never to provide default values for evidence. None of the standard evidence classes implements a default (empty) constructor, nor do they make assumptions about the value of incorrectly specified arguments. In all cases, the evidence classes' constructor throws a `System.ArgumentNullException` if passed null arguments. If arguments have invalid values, the constructor throws a `System.ArgumentException`.
- Make evidence immutable. Once created, the information held within an evidence class should be immutable. The evidence assigned to an assembly or application domain is accessible to many classes during policy resolution. It would cause problems if these classes could change the values of the evidence part way through the policy resolution process.
- Override `Object.ToString`. All standard evidence classes override the `Object.ToString` method to display a simple XML representation of the evidence object. For consistency, you should consider taking the same approach. Being able to display the contents of evidence is invaluable when testing and debugging.
- Grant identity permissions. To create evidence that will result in the granting of custom identity permissions, your evidence class must implement the `System.Security.Policy.IIdentityPermissionFactory` interface, which we discuss in [Chapter 7](#).

6.3.1.1 Defining the Author evidence class

To demonstrate the creation and use of custom evidence, create the `Author` class shown in [Example 6-3](#). The `Author` class provides evidence containing the name of the programmer who developed an assembly. Our ultimate goal is to assign code-access permissions to assemblies based on the assembly's `Author` evidence and to make runtime security decisions based on the assembly's `Author` using identity permissions.

In reality, you would be foolish to place any trust in evidence such as the `Author` class because it is susceptible to easy falsification. There is nothing to stop someone from assigning `Author` evidence that represents someone else to an assembly. However, `Author` is more than adequate for the purpose of our customization demonstration:

Example 6-3. Creating a custom evidence class

```
# C#

using System;
using System.Security;
using System.Reflection;

[assembly:AssemblyKeyFile("Keys.snk")]
[assembly:AssemblyVersion("1.0.0.0")]
```

```

namespace ORA.DotNetSecurity.Policy {

    [Serializable]
    public sealed class Author {

        private readonly string AuthorName = "";

        public Author (string author) {

            if (author == null) {
                throw new ArgumentNullException("author");
            } else {
                this.AuthorName = author;
            }
        }

        public string Name {

            get {
                return AuthorName;
            }
        }

        // Return string representation of the Author object
        public override string ToString( ) {

            // Create a new "Author" element
            SecurityElement se =
                new SecurityElement(this.GetType( ).FullName);

            // Add version of "1"
            se.AddAttribute("version", "1");

            // Add a child element to contain the author name
            if(AuthorName != "") {
                se.AddChild(new SecurityElement("Author", AuthorName));
            }

            // Render the SecurityElement to a string and return it
            return se.ToString( );
        }
    }
}

# Visual Basic .NET

Imports System
Imports System.Security
Imports System.Reflection

<assembly:AssemblyKeyFile("Keys.snk")>
<assembly:AssemblyVersion("1.0.0.0")>

```

```

Namespace ORA.DotNetSecurity.Policy

<Serializable> _
Public NotInheritable Class Author

    Private AuthorName As String = ""

    Public Sub New(ByVal author As String)
        If author Is Nothing Then
            Throw New ArgumentNullException("author")
        Else
            Me.AuthorName = author
        End If
    End Sub

    Public ReadOnly Property Name( ) As String
        Get
            Return AuthorName
        End Get
    End Property

    ' Return string representation of the Author object
    Public Overrides Function ToString( ) As String

        ' Create a new "Author" element
        Dim se As SecurityElement = _
            New SecurityElement(Me.GetType( ).FullName)

        ' Add version of "1"
        se.AddAttribute("version", "1")

        ' Add a child element to contain the author name
        If AuthorName <> "" Then
            se.AddChild(New SecurityElement("Author", AuthorName))
        End If

        ' Render the SecurityElement to a string and return it
        Return se.ToString( )
    End Function
End Class
End Namespace

```

The `Author` class implements many of the guidelines we outlined earlier, such as:

- `Author` is `sealed` (C#) and `NotInheritable` (Visual Basic .NET), ensuring that nobody can create a subclass that behaves differently than we intended.
- `Author` contains a single data member: a `String` named `AuthorName`, which contains the name of the author represented by the evidence object. `AuthorName` is a read-only private member that can be set only on instantiation and is retrievable only through the `Name` property, ensuring

that `Author` is immutable.

- `Author` is made serializable using `SerializableAttribute`. Because `Author` contains only a single `String` member, the default serialization capabilities provided by the `SerializableAttribute` are sufficient.
- `Author` overrides the `Object.ToString` method to render an `Author` object to XML in a format consistent with the standard evidence types. `Author` builds the XML representation using the `System.Security.SecurityElement` class, which we discuss in the next section, before returning it as a `String`. The output of `ToString` for an `Author` object representing the author "Peter" is shown below. The version attribute identifies the format of XML output in case you decide to represent future versions of the `Author` class differently.

```
<ORA.DotNetSecurity.Policy.Author version="1">
  <Author>Peter</Author>
</ORA.DotNetSecurity.Policy.Author>
```

6.3.1.2 Using the SecurityElement Class

`System.Security.SecurityElement` is a utility class that implements a simple, lightweight XML object model for encoding .NET security objects. `SecurityElement` lacks the functionality required for general-purpose XML processing but is sufficient for use within the security system where only simple XML representations are required. When extending the .NET security system, you will frequently need to use `SecurityElement`, which is why understanding how it works is essential.

A `SecurityElement` object represents a single XML element and provides members that allow you to specify the following characteristics:

- The element name or tag
- Attributes of the element
- Child elements (also represented by `SecurityElement` objects)
- Text within the body of the element

`SecurityElement` also includes methods that allow you to perform simple searches of your XML element and its children, as well as static utility methods for manipulating and testing the validity of string values used within `SecurityElement` objects. The `Author.ToString` method demonstrates the use of `SecurityElement`, and we include additional examples in [Chapter 7](#) and [Chapter 8](#), where we develop further CAS extensions. [Table 6-5](#) lists the members of `SecurityElement`.

`SecurityElement` provides flexibility in how you model the XML representation of your data. The .NET SDK documentation recommends that in the interest of readability and portability, you use attributes instead of child elements wherever possible and avoid text nested within tags. You will notice however, that all of the standard evidence classes use nested text values.

Table 6-5. Members of the SecurityElement class

Member	Description
Properties	
Attributes	Gets and sets the attributes of a <code>SecurityElement</code> using a <code>System.Collections.Hashtable</code> containing a collection of name/value string pairs representing the attributes.
Children	Gets and sets the child elements by providing a <code>System.Collections.ArrayList</code> of <code>SecurityElements</code> .
Tag	Gets or sets the tag of the <code>SecurityElement</code> .
Text	Gets or sets the text of the <code>SecurityElement</code> .
Methods	
AddAttribute	Adds an attribute with the specified name and value to the <code>SecurityElement</code> .
AddChild	Adds the specified <code>SecurityElement</code> as a child element.
Methods	
Attribute	Finds an attribute by name and returns its value, returning <code>null</code> (C#) or <code>Nothing</code> (Visual Basic .NET) if the attribute does not exist.
Equal	Tests two <code>SecurityElement</code> objects for equality by comparing tags, attributes, text, and child elements. This is different than the inherited <code>Object.Equals</code> method.
Escape	Static utility method for encoding invalid characters (that are invalid within XML) as valid escape sequences. Use the resulting string as a tag, attribute, or text value within the <code>SecurityElement</code> .
IsValidAttributeName	Static utility method that tests if a string represents a valid attribute name.
IsValidAttributeValue	Static utility method that tests if a string represents a valid attribute value.
IsValidTag	Static utility method that tests if a string represents a valid element tag.
IsValidText	Static utility method that tests if a string represents valid element text.
SearchForChildByTag	Searches the <code>SecurityElement</code> for the first child with the specified tag, returning the child as a <code>SecurityElement</code> . This method does <i>not</i> perform a recursive search; it searches only the immediate child elements.
SearchForTextOfTag	Searches the <code>SecurityElement</code> for the first child with the specified tag and returns the child's text. This method searches recursively through all child elements.
ToString	Returns a <code>System.String</code> containing the XML representation of the <code>SecurityElement</code> and its children.

6.3.1.3 Building the Author evidence class

For reasons that we will explain shortly, `Author.dll` needs a strong name, so we have included the `AssemblyKeyFile` and `AssemblyVersion` attributes. We explained the purpose and creation of assembly strong names in [Chapter 2](#). For this example, the `AssemblyKeyFile` attribute references a key file named `Keys.snk`, which we generated solely for this demonstration using the .NET Strong Name tool (`Sn.exe`). You should create the `Keys.snk` file and compile the `Author` class into a library named `Author.dll` using the following commands:

```
# C#

sn -k Keys.snk
csc /target:library Author.cs

# Visual Basic .NET

sn -k Keys.snk
vbc /target:library Author.vb
```

6.3.2 Using Custom Evidence

The previous section explains that the runtime recognizes two categories of evidence: host evidence and assembly evidence. Using custom evidence as host evidence at runtime is no different from using standard evidence; refer to [Section 6.2](#) for details.

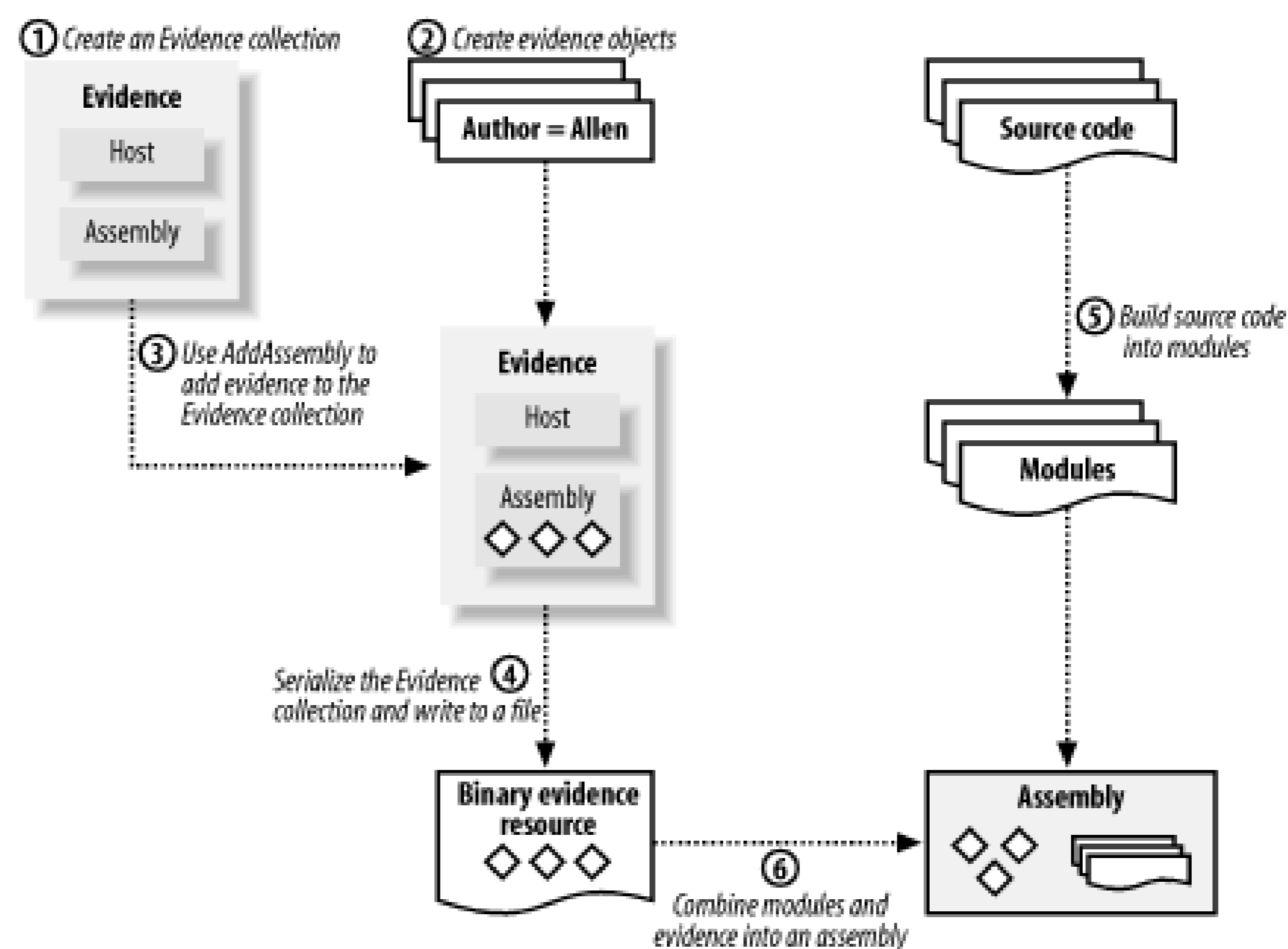
Because you must embed assembly evidence in the target assembly file, the use of custom evidence as assembly evidence involves additional steps during the build process of the assembly. You must prepare the custom assembly evidence programmatically, which you would normally do with a separate utility program, as we demonstrate later in this section. The process, illustrated in [Figure 6-3](#), is as follows:

1. Create a `System.Security.Policy.Evidence` collection.
2. Create the objects you want to assign as assembly evidence.
3. Add the evidence objects to the `Evidence` collection using the `AddAssembly` method.

There is no benefit in using the `AddHost` method to add evidence, because the runtime simply ignores host evidence when loading an assembly.

4. Serialize the `Evidence` collection using an instance of the `System.Runtime.Serialization.Formatter.Binary.BinaryFormatter` class and write it to a file.
5. Build your code into a set of modules using the `/target:module` flag on the C# or Visual Basic .NET compiler.
6. Combine the modules and the evidence resource into an assembly using the Assembly Linker tool (`al.exe`). Use the `/evidence:file` flag to specify the file containing the serialized assembly evidence.

Figure 6-3. Embedding custom evidence in an assembly



In the following sections, we expand on this summary and demonstrate exactly how to use `Author` evidence as embedded assembly evidence, but first we must make `Assembly.dll` a *fully trusted assembly*.

6.3.2.1 Making the Author assembly a fully trusted assembly

Assemblies that provide CAS extensions (such as the `Author.dll` in the previous section) must be fully trusted by the security system. This avoids problems when the runtime loads the assembly during the policy resolution process. We discuss the need for fully trusted assemblies in [Chapter 8](#)—for now, type the following commands:

```
gacutil -i Author.dll
caspol -user -addfulltrust Author.dll
caspol -machine -addfulltrust Author.dll
caspol -enterprise -addfulltrust Author.dll
```

The first command installs `Author.dll` into the global assembly cache; which you must do before you can make it a fully trusted assembly. This is why you had to give `Author.dll` a strong name when you built it. The other commands make `Author.dll` a fully trusted assembly in the `user`, `machine`, and `enterprise` policy levels. We explain policy levels in [Chapter 8](#) and the `Caspol.exe` utility in [Chapter 9](#).

6.3.2.2 Serializing evidence

To embed evidence in an assembly, you must serialize an `Evidence` collection that contains the evidence objects you want to use as assembly evidence. The .NET class library includes the

`System.Runtime.Serialization.Formatters.Binary.BinaryFormatter` class, which makes the serialization of serializable objects a straightforward process.

[Example 6-4](#) contains a simple utility named `CreateAuthorEvidenceResource` that creates an `Author` object using a name provided on the command line, adds the `Author` object to an `Evidence` collection, and then serializes the `Evidence` collection to a file. Running the command `CreateAuthorEvidenceResource Peter` results in the creation of the file named `Peter.evidence` that contains the serialized `Evidence` collection; you can then embed that file into the assembly.

Example 6-4. The `CreateAuthorEvidenceResource` utility

```
# C#

using System;
using System.IO;
using System.Security.Policy;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using ORA.DotNetSecurity.Policy;

public class CreateAuthorEvidenceResource {

    public static void Main(string[] args) {

        // Create a new Evidence collection
        Evidence ev = new Evidence( );

        // Create and configure new Author object
        Author auth = new Author(args[0]);

        // Add the new Author object to the assembly evidence
        // collection of the Evidence object
        ev.AddAssembly(auth);

        // Generate the name of the output file
        String file = auth.Name + ".evidence";

        // Serialize the Evidence object
        IFormatter fmtr = new BinaryFormatter( );
        Stream strm = new FileStream(file,
            FileMode.Create,
            FileAccess.Write,
            FileShare.None);
        fmtr.Serialize(strm, ev);
        strm.Close( );

        // Display result
        Console.WriteLine("Created author evidence resource : " +
            file);
    }
}
```

```

# Visual Basic .NET

Imports System
Imports System.IO
Imports System.Security.Policy
Imports System.Runtime.Serialization
Imports System.Runtime.Serialization.Formatters.Binary
Imports ORA.DotNetSecurity.Policy

Public Class CreateAuthorEvidenceResource

    Public Shared Sub Main(ByVal args( ) As String)

        ' Create a new Evidence collection
        Dim ev As Evidence = New Evidence( )

        ' Create and configure new Author object
        Dim auth As Author = New Author(args(0))

        ' Add the new Author object to the assembly evidence
        ' collection of the Evidence object
        ev.AddAssembly(auth)

        ' Generate the name of the output file
        Dim file As String = auth.Name & ".evidence"

        ' Serialize the Evidence object
        Dim fmtr As IFormatter = New BinaryFormatter( )
        Dim strm As Stream = New FileStream(file, _
            FileMode.Create, _
            FileAccess.Write, _
            FileShare.None)
        fmtr.Serialize(strm, ev)
        strm.Close( )

        ' Display result
        Console.WriteLine("Created author evidence resource : " & _
            file)
    End Sub
End Class

```

If you use custom evidence classes, or provide them to third parties to use in their own application development, provision of a utility similar to `CreateAuthorEvidenceResource` makes everyone's life a lot easier.

Build the `CreateAuthorEvidenceResource` class into an executable using the following command; remember, there is a dependency on the `Author.dll` assembly:

```

# C#

csc /reference:Author.dll CreateAuthorEvidenceResource.cs

```

```
# Visual Basic .NET

vbc /reference:Author.dll CreateAuthorEvidenceResource.vb
```

6.3.2.3 Embedding evidence in an assembly

We now have a mechanism for creating serialized `Evidence` collections, but we need a target assembly in which to embed the evidence. The simple `HelloWorld` class listed here will do for the purpose of this example:

```
# C#

using System;

public class HelloWorld {
    public static void Main( ) {
        Console.WriteLine("HelloWorld");
    }
}

# Visual Basic .NET

Imports System

Public Class HelloWorld
    Public Shared Sub Main( )
        Console.WriteLine("HelloWorld")
    End Sub
End Class
```

Embedding serialized evidence into your target assembly requires the use of the Assembly Linker tool (*al.exe*), which comes with the .NET Framework SDK. The Assembly Linker tool takes a number of modules and resources and combines them to create an assembly. You must first build your source into modules using the C# or Visual Basic .NET compilers and then combine the resulting modules, along with your evidence, into an assembly.

Assuming you have already built the `Author.dll` library and the `CreateAuthorEvidenceResource.exe` executable (as demonstrated in the previous sections), the following series of commands creates the `HelloWorld.exe` assembly complete with assembly evidence representing the author `Peter`:

```
# C#

CreateAuthorEvidenceResource Peter
csc /target:module HelloWorld.cs

al /target:exe /out:HelloWorld.exe /main:HelloWorld.Main
/evidence:Peter.evidence HelloWorld.netmodule

# Visual Basic .NET

CreateAuthorEvidenceResource Peter
vbc /target:module HelloWorld.vb
```

```
al /t:exe /out:HelloWorld.exe /main:HelloWorld.Main
/evidence:Peter.evidence HelloWorld.netmodule
```

The first command calls our `CreateAuthorEvidenceResource` utility, which creates a binary security resource file containing an `Author` evidence object for the author named `Peter`. Then you compile the `HelloWorld` source file into a module. Finally, you use the Assembly Linker tool to combine `HelloWorld.netmodule` and the `Peter.evidence` security resource to form the executable assembly named `HelloWorld.exe`.

Now run the `LoadAndList` utility developed in [Example 6-1](#) to view the evidence assigned to the `HelloWorld.exe` assembly when it is loaded. For example, if you place `HelloWorld.exe` in the directory `C:\dev`, and run the command `LoadAndList C:\Dev\HelloWorld.exe`, you see output similar to that shown below (the `Hash` evidence is abbreviated). Notice the inclusion of the `Author` evidence in the assembly evidence collection:

HOST EVIDENCE:

```
<System.Security.Policy.Zone version="1">
  <Zone>MyComputer</Zone>
</System.Security.Policy.Zone>

<System.Security.Policy.Url version="1">
  <Url>file:///C:/Dev/HelloWorld.exe</Url>
</System.Security.Policy.Url>

<System.Security.Policy.Hash version="1">
  <RawData>4D5A90000300000004000000FFFF0000B80000000000000040</RawData>
</System.Security.Policy.Hash>
```

ASSEMBLY EVIDENCE:

```
<ORA.DotNetSecurity.Policy.Author version="1">
  <Author>Peter</Author>
</ORA.DotNetSecurity.Policy.Author>
```

If you were to modify the `CreateAuthorEvidenceResource` class to write the `Author` evidence into the host subcollection of the `Evidence` class (using the `AddHost` method instead of the `AddAssembly` method), not only would the assembly evidence collection be empty, but the `Author` evidence would not be present in the host evidence of `HelloWorld.exe` because the runtime would ignore it.

6.3.3 The Next Steps in Customization

Now you have the `Author` evidence class that represents the programmer who developed an assembly. You can assign `Author` evidence as host evidence at runtime and embed it as assembly evidence during the build process. Unfortunately, you are not yet able to grant code-access permission based on the `Author` of an assembly. You will continue your extensions in [Chapter 8](#), where you implement a custom membership condition so that you can assign permissions to assemblies using `Author` evidence.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 7. Permissions

In this chapter, we explain permissions and discuss their role in the implementation of CAS. We describe the mechanisms that the runtime uses to enforce code-level security, and explain how you can use permissions to manipulate these mechanisms. We examine the functionality common to all permissions, and demonstrate the use of key permission classes provided in the .NET class library. Finally, we show you how to extend CAS by implementing your own custom permissions.

[\[Team LiB \]](#)

[[Team LiB](#)]

7.1 Permissions Explained

The goal of CAS is to give security administrators (and users) fine-grained control over the actions and resources to which code has access. For example, a security-conscious administrator may want to stop applications run from the Internet from starting new processes or allow only code written by Microsoft to be able to write to the Windows registry.

Instead of defining a fixed set of operations and resources to which access can be controlled, CAS provides a flexible and extensible framework that uses objects called permissions to define and enforce security. Permission objects serve the following purposes:

- When the runtime loads an assembly, it assigns the assembly a set of permission objects that represent the authority the runtime has granted to the assembly. Most permission objects represent access to actions or resources that are subject to security control, such as the ability to create application domains or the ability to write to the Windows event log. Other permission represent elements of code identity derived from the assembly's evidence, such as publisher certificates and strong names, which we discussed in [Chapter 6](#).
- Code that provides functionality or accessibility to resources that must be protected can use the security framework to demand that any code calling it has a specific set of permission objects. The .NET runtime and class library use this capability extensively to protect access to operation: and resources, including the ability to run unmanaged code or access the hard disk and Windows registry.

[Figure 7-1](#) illustrates the use of permission objects to enforce code-level security and demonstrates the most common CAS operation—the *security demand*. The runtime grants each of the application's assemblies a set of permissions at load time, which the runtime represents with a set of permission objects that it associated with the assemblies. When an assembly's code calls a protected library member, the library uses a permission object to demand that the runtime ensure that the calling assembly has the equivalent permission. The runtime evaluates the permission objects associated with the assembly and confirms to the library whether the assembly has the specified permission. Based on the response, the library determines whether to complete the application's request.

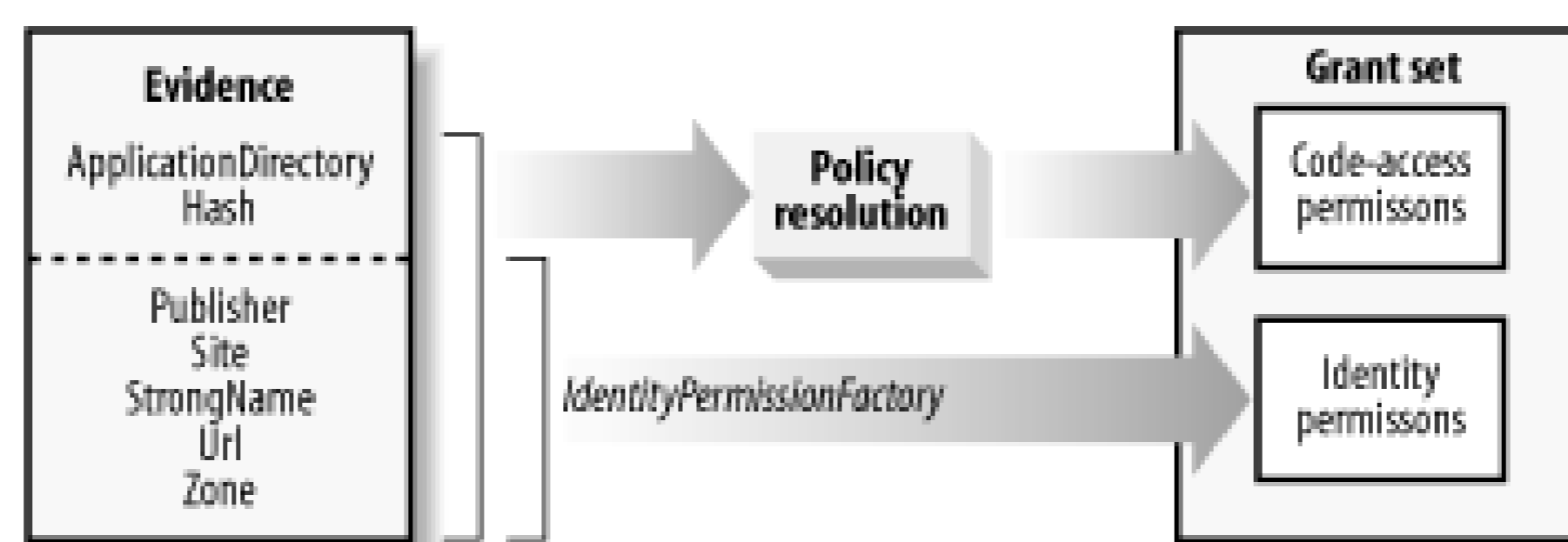
Figure 7-1. The security demand

As [Figure 7-1](#) shows, it is the library's responsibility to invoke the facilities provided by CAS to ensure that the calling application has the necessary permission to access its functionality. The library must also determine the appropriate action to take if the security demand fails. We will discuss this process in [Section 7.1.4](#), later in this chapter.

7.1.1 Granting Permissions

The functionality accessible to the code in an assembly is represented by the set of permission objects assigned to it by the runtime. The runtime determines which permissions to grant to an assembly (the *grant set*) based on the assembly's identity. The runtime establishes the identity of an assembly using the evidence the assembly presents when it is loaded; we discuss evidence and code identity in [Chapter 6](#). As shown in [Figure 7-2](#), the runtime uses two different mechanisms to translate evidence into permissions. [Figure 7-2](#) also shows that there are two types of permissions, which we discuss in [Section 7.1.3](#), later in this chapter.

Figure 7-2. Sources of permissions



The two mechanisms depicted in [Figure 7-2](#) work as follows:

1. The runtime compares all of the assembly's evidence with the .NET security policy, in a process named *policy resolution*, to determine which code-access permissions to grant to the assembly. We describe security policy and the policy resolution process in [Chapter 8](#). For now, it is enough to understand that security policy is a configurable set of rules that provides a mapping between evidence and permissions, and that policy resolution is the process of applying security policy to an assembly to determine the code-access permissions to grant to it.
2. Any *host* evidence that implements the `System.Security.Policy.IIdentityPermissionFactory` interface results in the runtime granting the assembly an identity permission that represents the type and value of the evidence. Of the seven standard evidence types discussed in [Chapter 6](#), `Publisher`, `Site`, `StrongName`, `Url`, and `Zone` evidence result in the creation of identity permissions.

Security policy can never grant identity permissions; the runtime only grants identity permission in response to an assembly or application domain presenting the necessary types of host evidence.

7.1.2 Requesting Permissions

Configuring security policy can be complicated, and problems arise when the security policy does not grant legitimate assemblies enough permission to function correctly. For example, without permission to establish network connections to an email server, an email client application cannot send and receive email. In an ideal security configuration, it is also important to make sure assemblies do not have more permissions than they need. This reduces the chance that bugs in the assembly can damage important resources. It also reduces the chance that malicious code can use the assembly as a gateway through which to access protected functionality. CAS supports the following three *permission request* operations to address the issues associated with assemblies having too many or too few permissions:

Requesting minimum permissions

Based on the premise that it is better for an application not to run than to start and crash because of insufficient permissions, CAS allows assemblies to include minimum permission requests. A minimum permission request is a statement contained within an assembly's metadata that specifies a set of permissions that the runtime must grant to an assembly in order for it to function correctly. After the runtime resolves the permissions for an assembly, if the grant set does not contain all of the requested permissions, then the runtime will throw an exception and will not load the assembly.

To use the email client application as an example, the programmer of the client builds a minimum permission request into the application assembly stating that the application must have permission to establish network connections. If the runtime does not grant permission to create network connections, then the application does not run and the user receives an error message. A security administrator can inspect the permission requests made by the application and adjust his security policy accordingly; we discuss the *Permvview.exe* tool used to do this in [Chapter 9](#).

Requesting optional permissions

Over and above core functionality, some applications offer additional but nonessential features. CAS allows assemblies to specify in their metadata a set of permissions that is useful for the assembly to have. The runtime loads the assembly even if it cannot grant these optional permissions, and so the assembly must accommodate situations in which it does not have them. For example, an email client application may allow users to export copies of email to disk if it has permission to write to the hard drive. Without the permission, the mail client still functions. However, it does not present the export option to the user nor does it display an appropriate message when the function is invoked.

The main effect of requesting optional permissions is that the runtime never grants the assembly any permission *not* listed in the optional-permission set; the maximum grant set possible is the union of the assembly's minimum permissions and optional permissions. The runtime discards any permissions granted to the assembly that fall outside these two groups.

Refusing permissions

The ability to request optional permissions enables an assembly to place an upper limit on the permissions that the runtime will grant it. However, sometimes you may want to refuse a specific set of permissions. CAS allows an assembly to specify a set of permissions in its metadata that the runtime must never grant it. Even if policy resolution normally results in the granting of a permission, the runtime will remove it when it loads the assembly. This allows you to ensure that nobody can use your code to perform any action controlled by the refused permissions. In the case of the email client application, the programmer may want to ensure that the client never calls unmanaged code and builds the refusal request into the assembly.

We describe the techniques used to make these requests in [Section 7.2.4](#).

7.1.3 Permission Types

The .NET Framework class library uses permissions to control access to the functionality it provides. The designers of the .NET class library have made decisions about which members of which classes must have restricted access and have implemented a set of permission classes through which they secure that functionality. The runtime also uses permissions to restrict access to a handful of operations that are critical to the security of the overall runtime environment. These operations include the ability of code to execute, to run unmanaged code, and to skip verification.

The permission classes implemented in the class library of the .NET Framework fall into three categories: code-access permissions, identity permissions, and role-based permissions.

Role-based permissions represent the user context in which code is executing. For example, the user **Gary** is running the application or an application is running as the inbuilt *Windows\system* account. We discuss role-based permissions in [Chapter 10](#) along with the other role-based security features provided by the .NET runtime.

In the following sections, we introduce the code-access and identity permission classes contained in the .NET class library.

It is important to understand that access to the vast majority of the functionality contained in the .NET class library is not protected using CAS. If .NET used CAS to protect every method in its class library, applications would run significantly slower and administration of security policy would be so complex as to be impossible. The class and method documentation of the .NET Framework SDK indicates where permissions are required to access functionality.

7.1.3.1 Code-access permissions

Code-access permission classes represent actions and resources that are subject to security control. The .NET class library includes the code-access permission classes listed in [Table 7-1](#). Some permission classes are members of the `System.Security.Permissions` namespace, while others are members of the namespace containing the functionality to which they apply. We break down the classes by namespace and highlight those permissions new to Version 1.1 of the .NET Framework.

All of these classes implement a common set of interfaces and adhere to a common design pattern. It is through the methods exposed by the code-access permission classes that code invokes the runtime to enforce code-level security. We discuss the specific functionality of code-access permission classes later in [Section 7.2.2](#).

Table 7-1. .NET code-access permission classes

Namespace and class	Description
System.Data.Common	
<code>DBDataPermission</code>	An abstract base class that provides common functionality for data provider permission classes, including the <code>OdbcPermission</code> , <code>OleDbPermission</code> , and <code>SqlClientPermission</code> classes. Note that <code>OraclePermission</code> is not derived from <code>DBDataPermission</code> .
System.Data.Odbc	
<code>OdbcPermission</code> (1.1)	Controls access to data sources through the ODBC data provider.
System.Data.OleDb	
<code>OleDbPermission</code>	Controls access to data sources through the OLE DB data provider.
System.Data.OracleClient	
<code>OraclePermission</code> (1.1)	Controls access to Oracle databases through the Oracle data provider.
System.Data.SqlClient	
<code>SqlClientPermission</code>	Controls access to Microsoft SQL Server databases through the SQL Client data provider.
System.Diagnostics	
<code>EventLogPermission</code>	Controls access to the Windows event log.
<code>PerformanceCounterPermission</code>	Controls access to Windows performance counters.
System.DirectoryServices	
<code>DirectoryServicesPermission</code>	Controls access to directory services, such as LDAP and Microsoft Active Directory.
System.Drawing.Printing	
<code>PrintingPermission</code>	Controls access to printers.
System.Net	
<code>DnsPermission</code>	Controls access to network-based domain name servers (DNSs).
<code>WebPermission</code>	Controls access to Internet resources.
<code>SocketPermission</code>	Controls access to socket-based network communications.
System.Messaging	
<code>MessageQueuePermission</code>	Controls access to the Microsoft message queue.
System.Security.Permissions	

Namespace and class	Description
<code>EnvironmentPermission</code>	Controls access to read, create, and change environment variables.
<code>FileDialogPermission</code>	Controls access to files and folders by restricting the accessibility and functionality of the standard Windows file dialog box.
<code>FileIOPermission</code>	Controls access to the local hard disk by restricting access to create, change, and delete files and folders.
<code>IsolatedStoragePermission</code>	Controls access to isolated storage, which we discuss in Chapter 11 .
<code>ReflectionPermission</code>	Controls access to the reflection functionality provided by the runtime.
<code>RegistryPermission</code>	Controls access to the Windows registry.
<code>ResourcePermissionBase</code>	An abstract base class that provides common functionality for Windows resource-oriented permission classes, including the <code>EventLogPermission</code> , <code>PerformanceCounterPermission</code> , <code>DirectoryServicesPermission</code> , and <code>ServiceControllerPermission</code> classes.
<code>SecurityPermission</code>	Controls access to the set of security features relating to application domains, policy, evidence, threading, principles, execution, infrastructure, .NET Remoting configuration, serialization, verification, and unmanaged code.
<code>UIPermission</code>	Controls access to manipulate the user interface and the clipboard.
<code>System.ServiceProcess</code>	
<code>ServiceControllerPermission</code>	Controls access to Windows services entries.
<code>System.Web</code>	
<code>AspNetHostingPermission</code> (1.1)	Controls access to ASP.NET-hosted environments.

Each of the standard code-access permission classes controls access to a range of related functionality and provides fine-grained control over the actions and resources a particular permission object represents. For example, the `SecurityPermission` class represents access to a set of discrete security related functions, such as the ability to control evidence or the ability to execute unmanaged code. In contrast, because the `FileIOPermission` class must represent different types of access (read, write, append) to everything from a specific named file or directory to the entire hard disk, it can represent an infinite variety of resource access.

Each of the standard code-access permission classes can also represent a completely restricted state and a completely unrestricted state. For example, an unrestricted `FileIOPermission` represents full access to all files and directories on the local hard drives, whereas a completely restricted `FileIOPermission` represents no access.

7.1.3.2 Identity permissions

Identity permissions represent the value of certain types of *host* evidence an assembly presents to the runtime when it was loaded. [Table 7-2](#) lists the standard identity permission classes provided in the .NET class library and shows the type of evidence they represent; all identity permission classes are members of the `System.Security.Permissions` namespace. We discussed each of the evidence classes in [Chapter 6](#).



Only *host* evidence is a source of identity permissions; *assembly* evidence never results in their creation. See [Chapter 6](#) for a description of the difference between *host* and *assembly* evidence.

Table 7-2. Evidence classes that implement `IIdentityPermissionFactory`

Identity permission class	Evidence type represented
<code>PublisherIdentityPermission</code>	<code>Publisher</code>
<code>SiteIdentityPermission</code>	<code>Site</code>
<code>StrongNameIdentityPermission</code>	<code>StringName</code>
<code>UrlIdentityPermission</code>	<code>Url</code>
<code>ZoneIdentityPermission</code>	<code>Zone</code>

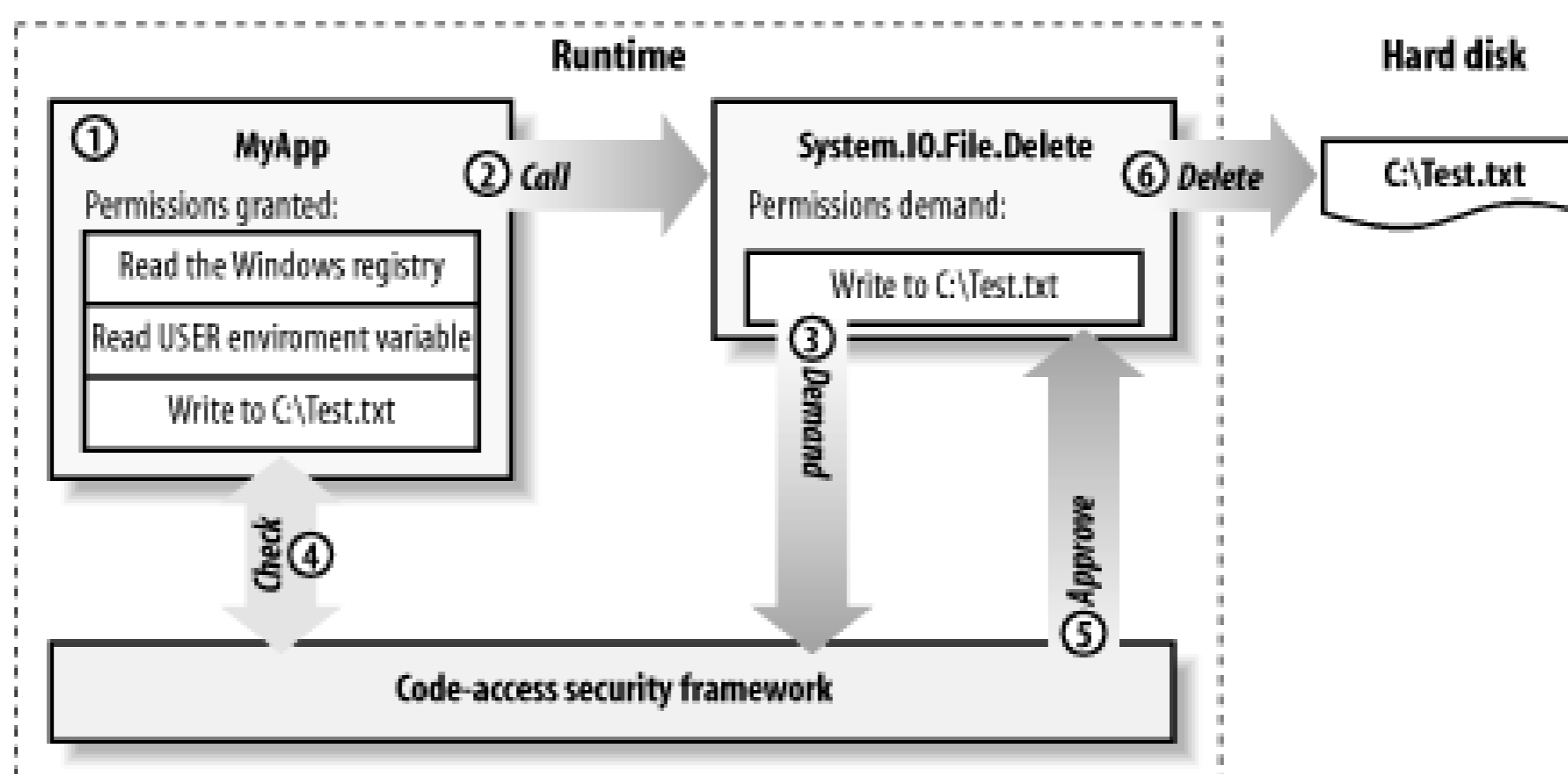
Because identity permissions implement the same interfaces as code-access permissions, you can use them in the same ways. Identity permissions are a powerful and convenient way to leverage the functionality of the CAS security framework and make security decisions based on the identity of an assembly, avoiding the need to work directly with evidence objects. For example, you can use identity permissions to enforce the following types of security restrictions:

- To allow only code run from the `LocalIntranet` security zone to execute your method
- To allow only code in an assembly signed with your company's publisher certificate to inherit from your class
- To run your application only if it was executed from the web site `www.oreilly.com`

7.1.4 Enforcing Code-Access Security

The security demand, which we mentioned in the introduction to this section, is the principal mechanism used to enforce code-level security. When code makes a security demand, it instructs the runtime to ensure that the code that called it has a specified permission object (or set of permissions). [Figure 7-3](#) shows what happens when a simple console application named `MyApp.exe` tries to delete a file named `C:\Test.txt` using the static `System.IO.File.Delete` method from the .NET class library.

Figure 7-3. The security demand process



The security demand process shown in [Figure 7-3](#) works as follows:

1. When you run `MyApp`, the runtime loads the `MyApp.exe` assembly, evaluates its evidence, and determines what permissions to grant it. In this example, the runtime grants `MyApp.exe` the following three permissions, which it creates permission objects for and assigns to the assembly:
 - o A `RegistryPermission` object that represents permission to read the Windows registry
 - o An `EnvironmentPermission` object that represents permission to read the `USER` environment variable
 - o A `FileIOPermission` object that represents permission to write to the `C:\Test.txt` file
2. At some point during its execution, `MyApp` calls `File.Delete` to delete the `C:\Test.txt` file.
3. The `File.Delete` method creates a `FileIOPermission` object that represents permission to write to the `C:\Test.txt` file and calls its `Demand` method to request that the runtime ensure that `MyApp` has the equivalent permission.
4. The runtime checks the permission objects assigned to `MyApp` and confirms that they include a `FileIOPermission` that represents permission to write to the `C:\Test.txt` file.
5. The runtime responds with a positive result to the `File.Delete` method, indicating that `MyApp` has the demanded permission. The positive response is *not* to throw a `System.Security.SecurityException`.
6. Given that `MyApp` has the necessary permission, the `File.Delete` method attempts to delete the `C:\Test.txt` file; only if the access controls of the underlying operating system allow the active user to delete the `C:\Test.txt` file will the operation succeed; we discussed the relationship between runtime and operating system security in [Chapter 5](#).

In this example, `MyApp` is the only executing application and is the only assembly that you are concerned with. However, when code makes a security demand, the runtime must ensure that not only the immediate caller has the specified permission, but that the whole chain of previous callers have the necessary permission as well. This ensures that less-trusted code does not exploit more

highly trusted code to perform operations to which it does not have permission (commonly known as a *luring attack*). The runtime checks the permission of all callers using a *stack walk*, which we describe in the next section. We discuss the implementation of security demands in [Section 7.2.5](#).

7.1.4.1 Stack walks explained

The runtime maintains a call stack for each thread of execution. When an active thread calls a method or function, the runtime adds a new *activation record* (also called a *stack frame*) to the top of the thread's call stack. The activation record contains details of the method that was called, the arguments passed to the method, and the address to return to when the method completes. Upon completion of the called method, the runtime removes the activation record from the top of the stack and returns control to the instruction specified in the record.

The runtime also records application domain transitions on the call stack. As code running in one application domain calls a method in another application domain, the runtime records the transition by adding a record to the call stack. We discussed application domains in [Chapter 3](#).

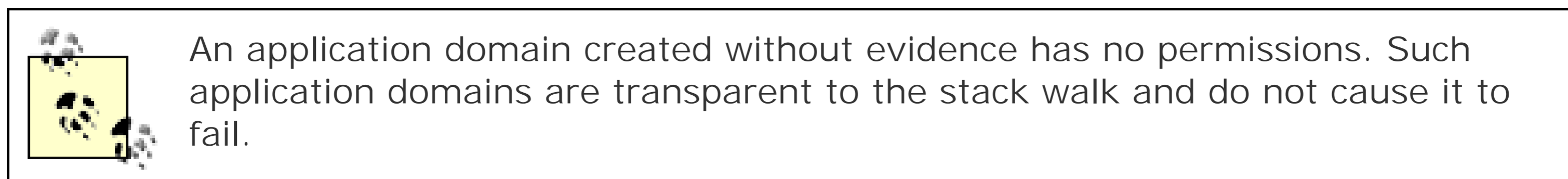
The call stack grows and shrinks constantly throughout the life of an application, but at any instant, the call stack contains an accurate record of the sequence of method calls and application domain transitions that have brought the code to its current point of execution. [Figure 7-4](#) shows the state of call stack after a sequence of method calls (shown with arrows), which cross assembly and application domain boundaries.

Figure 7-4. Structure of a call stack

When code makes a security demand, the runtime walks back up the call stack (hence the name stack walk), from the most recent activation record to the oldest. Importantly, stack walks do not include the activation record of the method making the security demand; the first activation record evaluated is that of the immediate caller.

As the runtime walks the stack, it compares the permissions granted to each method and application domain to see whether they contain the demanded permission. Remember that the permissions of a method are the same as the permissions of the assembly in which it is contained.

If every method and application domain on the call stack has the demanded permission, then the stack walk completes without event; execution returns to the method that made the security demand, which in turn carries on to complete the protected operation. However, if the runtime encounters a method or application domain without the demanded permission, the runtime throws a `System.Security.SecurityException` back to the method that made the security demand, which would normally pass this through to the code that called it.



[Figure 7-5](#) shows a stack walk that results when `MethodD` from [Figure 7-4](#) calls the `File.Delete` method. The `Delete` method demands that callers have the `FileIOPermission` permission to write to a specified file. In response to the security demand, the runtime walks the stack and inspects the permissions granted to first `MethodD`, and then `MethodC`, `MethodB`, `AppDomainY`, `MethodA`, and finally `AppDomainX`. As long as all of them have the necessary `FileIOPermission`, the `Delete` call is permitted.

Figure 7-5. Walking the stack

Some important points to note:

- `MethodC` and `MethodD` have the same grant set, because they are both contained in `AssemblyB` and are executing from the same application domain. It is still necessary for the runtime to evaluate each method independently because of the features we discuss in [Section 7.2.6](#).
- Even though both `MethodA` and `MethodB` are contained in `AssemblyA` they may not have the same permissions. `AssemblyA` has been loaded into two different application domains. This can result in each instance having different permission sets depending on the security policy of `AppDomainY`; we discuss security policy in [Chapter 8](#).

The stack walk process described here is the default behavior of the runtime in response to a security demand. You can override the default behavior of the stack walk process in three ways:

`Assert`

`Assert` allows a method to specify a set of permissions for which it is willing to vouch for the callers' authority above it on the stack. The stack walk terminates with a positive result at the frame of the method invoking `Assert` and does not look at the remaining stack frames.

`Deny`

`Deny` is effectively the opposite of `Assert`. Any stack walk looking for the denied set of permissions terminates with a negative result at the frame of the method invoking `Deny`, regardless of the permissions held by the remaining methods on the stack.

`PermitOnly`

`PermitOnly` is similar to `Deny`, but it allows a method to specify the set of permissions that should not terminate the stack walk. A security demand for any other permission results in the stack walk terminating as if it has encountered a `Deny` override.

We discuss each of the `Assert`, `Deny`, and `PermitOnly` overrides further in [Section 7.2.6](#), where we provide implementation examples.

[\[Team LiB \]](#)

[Team LiB]

7.2 Programming Code-Access Security

So far, we have discussed permissions and introduced the features provided by CAS to enforce and control code-level security. In this section, we look at how you implement these features in your programs. Your level of interest in this section will depend on the type of code you intend to write. As a developer of applications, you will be interested in understanding the implementation and functionality of the standard permission classes, as well as how to make security requests to ensure that your application has the permissions it needs to execute. If you are writing libraries, you will also be interested in how to use permissions to control access to your functionality, as well as the features available to control the stack walk process.

We begin by explaining the general syntax of security statements, followed by a look at the standard permission classes included in the .NET class library. There are more than 25 different code-access and identity permission classes in Version 1.1 of the .NET Framework, and we do not cover them all in detail. We provide detailed coverage of only the `SecurityPermission` class, which is an important permission for performing many of the security-related operations we discuss in this book. Once you understand how to use permissions in general, everything you need to know about specific permission classes can be found easily in the .NET Framework SDK documentation.

We conclude this section by looking at how to apply different security syntax to build permission requests, security demands, and stack walk overrides into your code.

7.2.1 Security Statement Syntax

The statements to control CAS do not form part of the C# or Visual Basic .NET languages. It is through the permission classes and their associated attribute class counterparts that you control CAS although this makes the syntax and use of CAS more complex and initially harder to understand, it means that CAS is flexible, extensible, and to a large degree language-independent.

There are two different ways to express security statements. The first, *imperative security*, uses the standard methods implemented by the permission classes. The second, *declarative security*, uses the attribute class counterparts of each permission class.

7.2.1.1 Imperative security statements

Imperative security statements appear in the body of your methods and functions, which means they end up forming part of the compiled intermediate language (IL) code contained in your assembly. We discuss IL code in Chapter 2 in our discussion of assemblies.

The first steps in using imperative security are to instantiate a permission object of the type you require and configure it to represent the permission you want to work with. Then you call one of the following methods, which are implemented by all code-access and identity permission classes to execute the desired security statement. For security demands, you will use the `Demand` method; for stack walk manipulation, you will typically use the `Assert`, `Deny`, and `PermitOnly` methods.

We discuss the details of these methods later in this chapter; for now all we are concerned with

understanding is the general syntax of imperative security statements. Example 7-1 creates a `FileIOPermission` object, configures it to represent write access to the file `C:\SomeFile.txt`, and then calls its `Demand` method to initiate an imperative security demand. Unless the code calling the `CreateFile` method (and all previous callers on the call stack) has been granted write access to the `C:\SomeFile.txt` file, the runtime throws a `System.Security.SecurityException`.

Example 7-1. Imperative security statement

```
# C#
public void CreateFile( ) {

    // Create a new FileIO permission object
    FileIOPermission perm = new FileIOPermission(
        FileIOPermissionAccess.Write, @"C:\SomeFile.txt");

    try {
        // Demand the FileIOPermission
        perm.Demand( );
    } catch (SecurityException se) {
        // Callers do not have necessary permission
    }

    // Method implementation...
}

# Visual Basic .NET

Public Function CreateFile ( )

    ' Create a new FileIO permission object
    Dim perm As FileIOPermission = New FileIOPermission( _
        FileIOPermissionAccess.Write, "C:\SomeFile.txt")

    Try
        ' Demand the FileIOPermission
        perm.Demand( )
    Catch (se As SecurityException)
        ' Callers do not have necessary permission
    End Try

    ' Method implementation...

End Function
```

You cannot express all security operations using imperative syntax. However, imperative security offers a level of flexibility and control that you cannot achieve using declarative syntax, which we discuss in Section 7.2.1.2.

Because imperative security statements appear in the body of your program, you can use them in conjunction with the normal program control constructs supported in the language, such as conditional and iterative statements. In addition, imperative security allows you to base security decisions on information that is available only at runtime, such as user input, and you can use

structured exception handling to process security exceptions that result from your imperative security statements.

7.2.1.2 Declarative security statements

Declarative security statements are expressed using attributes, which are statements compiled to form part of an assembly's metadata. For every code-access and identity permission class, the .NET class library contains a corresponding attribute class, which enables the use of declarative security syntax with that permission. Each attribute class is a member of the same namespace as its corresponding permission class and has the same name as the permission, but with the word "Attribute" appended. For example, the corresponding attribute for the `FileIOPermission` class is the `FileIOPermissionAttribute` class, and they are both members of the `System.Security.Permissions` namespace.



Because the C# and Visual Basic .NET compilers both allow you to omit the standard "Attribute" appendix from attributes when you use them in your code, it looks as if you are using the permission class which aids readability.

All permission attribute classes extend the `System.Security.Permissions.CodeAccessSecurityAttribute` class, including the attribute class counterparts of identity permissions. The `CodeAccessSecurityAttribute` class and every subclass implement a single constructor with the signature shown here:

```
# C#

public CodeAccessPermissionAttribute(
    SecurityAction action
);

# Visual Basic .NET

Public Sub New( _
    ByVal action As SecurityAction _
)
```

`System.Security.Permissions.SecurityAction` is an enumeration that contains members representing each of the security operations supported by declarative syntax.

Permission requests

- `RequestMinimum`
- `RequestOptional`
- `RequestRefuse`

Security demands

- `Demand`

- `LinkDemand`
- `InheritanceDemand`

Stack walk manipulation

- `Assert`
- `Deny`
- `PermitOnly`

The selection of operations supported by declarative security is more than that supported using imperative security. We discuss the details of each operation later in this chapter; for now we are concerned only with the general syntax of declarative security statements.

Each attribute class defines a set of properties that are specific to the type of permission. These allow you to configure the attribute to represent the permission state to which you want to invoke the declarative security statement. The following code demonstrates the declarative syntax alternative to that listed in Example 7-1. The `Write` property is specific to the `FileIOPermissionAttribute` class and specifies the full path of the file to which write access is granted, as in `C:\SomeFile.txt`. To change the security operation, it is a simple matter of changing the value of the `SecurityAction` argument:

```
# C#

[FileIOPermission(SecurityAction.Demand, Write = @"C:\SomeFile.txt")]
public void CreateFile( ) {

    // Method implementation...
}

# Visual Basic .NET

<FileIOPermission(SecurityAction.Demand, Write = "C:\SomeFile.txt"> _
Public Sub CreateFile ( )

    ' Method implementation...
End Sub
```

As metadata, the runtime and other tools can extract and view the declarative security statements contained in your assembly without loading or running the assembly's IL. *Permview.exe* is one such tool supplied with the .NET Framework. We discuss the use of *Permview.exe* in Chapter 9, but in summary, it allows security administrators to inspect all the declarative security statements contained in your assembly. This provides them with valuable information when they are configuring security policy.

The primary limitation of declarative syntax when compared with imperative syntax is that the permission definitions are fixed at build time and cannot be based on runtime state. The only way to change the permissions is to modify the code and recompile it.

7.2.2 Programming Permissions

The use you make of the standard permission classes in your own development will vary depending on what type of code you are writing. If you are writing applications that call protected functionality of the .NET class library, you will be concerned primarily with ensuring that your code has the necessary permissions to call the functionality it needs and fails gracefully if it does not. Your focus will be on the correct configuration of security policy (see Chapter 8), making the necessary permission requests and correctly handling any security exceptions thrown at runtime.

If you are writing library code that will be used by others, use the standard evidence classes in the same way that the .NET class library does to protect the functionality you provide. Be aware that if you are providing a wrapper around existing .NET class library functionality, you may not need to implement your own security because the underlying methods you call could already be protected. If you are creating a library that calls unmanaged code or provides access to a new type of resource, you may decide to protect it using an appropriate standard permission class. However, permission classes tend to be task-specific, and you might find yourself wanting to implement a completely new permission class to represent your needs, which we discuss in Section 7.3.1.

Each of the standard permission classes represents a different type of secured functionality or code identity. Some code-access permission classes represent access to a set of discrete actions grouped together into a single container (for example, `SecurityPermission`), but many represent access to an infinite variety of resources (like `FileIOPermission`). Despite this variety and scope of purpose, all permission classes integrate with the security framework to provide the same base functionality and pattern of use.

7.2.2.1 Common permission class functionality

All code-access and identity permissions extend directly or indirectly the abstract `System.Security.CodeAccessPermission` class, which implements three interfaces that define the functionality all permission classes must provide:

`System.Security.IPermission`

`IPermission` defines methods to initiate security demands as well as to create, compare, and combine permission objects.

`System.Security.IStackWalk`

`IStackWalk` defines methods that allow code to invoke security demands and override the default stack-walk process.

`System.Security.ISecurityEncodable`

`ISecurityEncodable` defines methods that allow permission objects to be converted to and from XML representations.

In addition to the methods defined by these interfaces, `CodeAccessPermission` implements four static methods that are used to undo previously implemented stack-walk overrides. Table 7-3 summarizes the methods of the `CodeAccessPermission` class broken down by interface.

Table 7-3. Methods of the `CodeAccessPermission` class

Method	Description
IPermission interface	
<code>Copy</code>	Returns a permission object that is a copy of the current permission.
<code>Demand</code>	Invokes a security demand for the current permission object-see Section 7.2.5 in this chapter for details.
<code>Intersect</code>	Returns a permission that is the intersection the current permission and the specified permission.
<code>IsSubsetOf</code>	Determines whether the current permission is a subset of the specified permission.
<code>Union</code>	Creates a permission that is the union of the current permission and the specified permission.
IStackWalk interface	
<code>Assert</code>	Adds an <code>Assert</code> override to the current stack frame-see Section 7.2.6 in this chapter for details.
<code>Demand</code>	See <code>IPermission.Demand</code> .
<code>Deny</code>	Adds a <code>Deny</code> override to the current stack frame-see Section 7.2.6 in this chapter for details.
<code>PermitOnly</code>	Adds a <code>PermitOnly</code> override to the current stack frame-see Section 7.2.6 in this chapter for details.
<code>FromXml</code>	Reconstructs a permission object from a <code>System.Security.SecurityElement</code> that contains an XML object model representing the permissions state.
<code>ToXml</code>	Returns a <code>SecurityElement</code> that contains an XML object model of the permission object, including state information.
ISecurityEncodable	
<code>RevertAll</code>	Removes all <code>Assert</code> , <code>Deny</code> , and <code>PermitOnly</code> overrides from the current stack frame- see Section 7.2.6 in this chapter for details.
<code>RevertAssert</code>	Removes all <code>Assert</code> overrides from the current stack frame-see Section 7.2.5 in this chapter for details.
<code>RevertDeny</code>	Removes all <code>Deny</code> overrides from the current stack frame-see Section 7.2.6 in this chapter for details.
<code>RevertPermitOnly</code>	Removes all <code>PermitOnly</code> overrides from the current stack frame-see Section 7.2.6 in this chapter for details.

All code-access permission classes also implement the `System.Security.Permissions.IUnrestrictedPermission` interface, which includes a single method named `IsUnrestricted`. The `IsUnrestricted` method returns true if a permission object represents the unrestricted state of the permission class; otherwise, it returns false.



`CodeAccessPermission` does not implement `IUnrestrictedPermission` because identity permissions, which also extend `CodeAccessPermission`, never represent unrestricted states.

Each concrete code-access permission class implements its own internal state to represent the functionality to which it controls access. Likewise, each concrete identity permission class implements internal state to represent values appropriate to the type of evidence on which they are based. Therefore, all permission classes implement a unique set of constructors, properties, and methods necessary to manipulate their own internal state information. One constructor signature common to all code-access permission classes is shown here for the `SecurityPermission` class:

```
# C#

public SecurityPermission(
    PermissionState state
);

# Visual Basic .NET

Public Sub New( _
    ByVal state As PermissionState _
)
```

The `PermissionState` enumeration contains two member values:

`None`

Represents a completely restricted state (no permission)

`Unrestricted`

Represents an unrestricted state (all permissions)

7.2.2.2 Using the `SecurityPermission` class

The `SecurityPermission` class is used by both the runtime and methods in the .NET Framework class library to control access to a set of 14 discrete operations that are critical to the security of the runtime environment. Many of the activities we discuss in this book require permissions represented by the `SecurityPermission` class. These include things like controlling evidence, modifying security policy, and creation application domains.

The `SecurityPermission` class implements two constructors with the following signatures:

```
# C#

public SecurityPermission(
    PermissionState state
);

public SecurityPermission(
    SecurityPermissionFlag flag
```



```
);

# Visual Basic .NET

Public Sub New( _
    ByVal state As PermissionState _
)

Public Sub New( _
    ByVal flag As SecurityPermissionFlag _
)
```

The first constructor takes a member of the `PermissionState` enumeration, which we discussed in the previous section. Passing the value `PermissionState.None` creates a `SecurityPermission` that grants access to none of the contained 14 operations, while `PermissionState.Unrestricted` grants access to them all.

The second constructor takes a member of the `System.Security.Permissions.SecurityPermissionFlag` enumeration, which defines the set of component operations represented by the `SecurityPermission` object. Table 7-4 lists the members of the `SecurityPermissionFlag` enumeration. Because `SecurityPermissionFlag` has the `FlagsAttribute` attribute, you can specify more than one value simultaneously using bitwise combination.

Table 7-4. Members of the `SecurityPermissionFlag` enumeration

Member	Description
<code>AllFlags</code>	All the permissions listed in this table except for <code>NoFlags</code> .
<code>Assertion</code>	The ability to call <code>Assert</code> on the permissions that have been granted to the code, which we discuss in Section 7.2.6.
<code>BindingRedirects</code>	The ability to perform binding redirection of assemblies in the application configuration file.
<code>ControlAppDomain</code>	The ability to create and manipulate application domains, which we discuss in Chapter 3.
<code>ControlDomainPolicy</code>	The ability to specify application domain policy, which we discuss in Chapter 8.
<code>ControlEvidence</code>	The ability to provide evidence for assemblies and application domains, which we discuss in Chapter 6.
<code>ControlPolicy</code>	The ability to view and edit security policy, which we discuss in Chapter 8.
<code>ControlPrincipal</code>	The ability to manipulate the principal object associated with application domains and threads, which we discuss in Chapter 10.
<code>ControlThread</code>	Advanced threading control.

Member	Description
<code>Execution</code>	The ability of code to execute.
<code>Infrastructure</code>	The ability to plug certain types of code extensions into the .NET Framework.
<code>NoFlags</code>	No permissions.
<code>RemotingConfiguration</code>	The ability to configure .NET Remoting types and channels.
<code>SerializationFormatter</code>	The ability to provide serialization services.
<code>SkipVerification</code>	The ability for code to skip verification, which we discuss in Chapter 4.
<code>UnmanagedCode</code>	The ability to execute unmanaged code. Calls to unmanaged code through either PInvoke or COM Interoperability result in an implicit security demand for the <code>SecurityPermission.UnmanagedCode</code> permission. This is an expensive operation when code calls unmanaged code frequently. Managed libraries that provide access to unmanaged code can use the <code>System.Security.SuppressUnmanagedCodeAttribute</code> to limit the performance impact; refer to the .NET Framework SDK documentation for further information.

Example 7-2 demonstrates how to create two `SecurityPermission` objects. The first represents unrestricted access to all `SecurityPermission` operations, and the second specifies only the `Assertion` and `ControlEvidence` permissions.

Example 7-2. Constructing `SecurityPermission` objects

```
# C#

SecurityPermission p1 =
    new SecurityPermission(PermissionState.Unrestricted);

SecurityPermission p2 = new SecurityPermission(
    SecurityPermissionFlag.Assertion |
    SecurityPermissionFlag.ControlEvidence
);

# Visual Basic .NET

Dim p1 As SecurityPermission = _
    New SecurityPermission(PermissionState.Unrestricted)

Dim p2 As SecurityPermission = _
    New SecurityPermission(SecurityPermissionFlag.Assertion Or _
    SecurityPermissionFlag.ControlEvidence)
```

Using these `SecurityPermission` objects, you can execute imperative security statements. For example, your code could use an imperative security demand to control the functionality it exposes to the calling code based on the permissions the callers have:

```

# C#

// Declare boolean on which to base available functionality
bool unrestricted = false;

// Test to see if callers have unrestricted security access
try {
    p1.Demand( );
    unrestricted = true;
} catch (SecurityException) {
    // Don't have unrestricted access, check for Assertion and
    // ControlEvidence
    try {
        p2.Demand( );
    } catch (SecurityException) {
        // No Assertion or ControlEvidence permission either
        // Take appropriate action
    }
}

```

```

# Visual Basic.NET

```

```

' Declare boolean on which to base available functionality
Dim unrestricted As Boolean = False

```

```

' Test to see if callers have unrestricted security access
Try

```

```

    p1.Demand( )
    unrestricted = True

```

```

Catch

```

```

    ' Don't have unrestricted access, check for Assertion and
    ' ControlEvidence

```

```

    Try

```

```

        p2.Demand( )

```

```

    Catch

```

```

        ' No Assertion or ControlEvidence permission either
        ' Take appropriate action

```

```

    End Try

```

```

End Try

```

Given the nature of the actions protected by `SecurityPermission`, it is more likely that you will want to use it in a permission request. The `SecurityPermissionAttribute` class is the attribute counterpart of the `SecurityPermission` class. It implements a single constructor that takes a `SecurityAction` argument, which defines what type of security action the attribute represents:

```

# C#

```

```

public SecurityPermissionAttribute(
    SecurityAction action
);

```

```

# Visual Basic .NET

```

```
Public Sub New( _
    ByVal action As SecurityAction _
)
```

The configuration of `SecurityPermissionAttribute` is specified using its properties. `SecurityPermissionAttribute` defines a set of properties with the same name and purpose as the component permissions listed in Table 7-4. Each of these properties takes a Boolean value to indicate whether the attribute configuration contains the specified component permission. There is also an `Unrestricted` property, which allows you to represent an unrestricted `SecurityPermission`, and a `Flags` attribute, which allows you to logically combine the values of `SecurityPermissionFlag` listed in Table 7-4, as you would in a normal `SecurityPermission` constructor.

The following code demonstrates the use of the `SecurityPermissionAttribute` to define the equivalent `SecurityPermission` instances as shown in Example 7-2. In this case, we are issuing a minimum-permission request for the specified `SecurityPermission`, as indicated by the `SecurityAction.RequestMinimum` argument:

```
# C#

[SecurityPermission(SecurityAction.RequestMinimum, Unrestricted = true)]

[SecurityPermission(SecurityAction.RequestMinimum, Assertion = true,
    ControlEvidence = true)]

# Visual Basic .NET

<SecurityPermission(SecurityAction.RequestMinimum, _
Unrestricted := True)> _

<SecurityPermission(SecurityAction.RequestMinimum, _
Assertion := True, ControlEvidence := True)> _
```

7.2.3 Programming Permission Sets

When implementing CAS features in your code, you will often need to work with more than one permission at a time. For example, a method can make a security demand for both `FileIOPermission` and `SecurityPermission`. The `System.Security.PermissionSet` class provides the capability to create permission sets that can contain an arbitrary number of individual permission objects.

Permission sets expose much of the same functionality as the individual permission classes, and you can use an entire permission set in the same way as you can an individual permission to enforce code-level security. The `PermissionSet` class implements two constructors with the following signatures:

```
# C#

public PermissionSet(
    PermissionSet permSet
);
```

```

public PermissionSet(
    PermissionState state
);

# Visual Basic .NET

Public Sub New( _
    ByVal permSet As PermissionSet _
)

Public Sub New( _
    ByVal state As PermissionState _
)

```

The first constructor populates the new `PermissionSet` with *copies* of the permission objects contained in the specified `PermissionSet`. The second constructor takes a member of the `PermissionState` enumeration, which we discussed in Section 7.2.2.1. Specifying `PermissionState.None` creates an empty `PermissionSet` with no permissions. Passing `PermissionState.Unrestricted` creates a permission set that is equivalent to having all permissions that implement the `IUnrestricted` interface (this includes all of the standard code-access permissions but not the identity permissions).

`PermissionSet` implements the same `IStackWalk` interface that the individual permission classes do, which allows you to execute the `Assert`, `Demand`, `Deny`, and `PermitOnly` methods against the entire set of permissions with a single call. `PermissionSet` also implements the `System.Collections ICollection` and `System.Collections.IEnumerator` interfaces to facilitate the management and manipulation of the contained permission objects. Table 7-5 summarizes the members of the `PermissionSet` class.

Table 7-5. Members of the `PermissionSet` class

Member	Description
Properties	
<code>Count</code>	Gets the number of <code>IPermission</code> objects in the permission set.
<code>IsReadOnly</code>	Always returns false; read-only permission sets are not supported.
Methods	
<code>AddPermission</code>	Adds an <code>IPermission</code> to the <code>PermissionSet</code> . If the <code>PermissionSet</code> already contains an <code>IPermission</code> of the same type, the result is the union of the existing <code>IPermission</code> and the new one. The resulting <code>IPermission</code> added to the <code>PermissionSet</code> is also returned to the caller.
<code>Assert</code>	Adds an <code>Assert</code> override to the current stack frame for all permission objects in the <code>PermissionSet</code> -see Section 7.2.6 in this chapter for details.

Member	Description
<code>ContainsNonCodeAccessPermissions</code>	Determines if the <code>PermissionSet</code> contains any <code>IPermission</code> objects that do not extend <code>CodeAccessPermission</code> .
<code>Copy</code>	Creates a copy of the <code>PermissionSet</code> .
<code>CopyTo</code>	Copies all of the <code>IPermission</code> objects in the <code>PermissionSet</code> to an array.
<code>Demand</code>	Invokes a security demand for all permission objects in the <code>PermissionSet</code> -see Section 7.2.5 in this chapter for details.
<code>Deny</code>	Adds a <code>Deny</code> override to the current stack frame for all permission objects in the <code>PermissionSet</code> -see Section 7.2.6 in this chapter for details.
<code>FromXml</code>	Reconstructs the state of the permission set from a <code>System.Security.SecurityElement</code> , which contains an XML object model representing the desired <code>PermissionSet</code> state.
<code>GetEnumerator</code>	Returns a <code>System.Collections.IEnumerator</code> for the <code>IPermission</code> objects contained in the <code>PermissionSet</code> .
<code>GetPermission</code>	Returns a copy of the <code>IPermission</code> of the specified <code>System.Type</code> contained in the <code>PermissionSet</code> .
<code>Intersect</code>	Creates a new <code>PermissionSet</code> that is the intersection of the current <code>PermissionSet</code> and another.
<code>IsEmpty</code>	Returns true if the <code>PermissionSet</code> is empty; otherwise, returns false.
<code>IsSubsetOf</code>	Determines if the <code>PermissionSet</code> is a subset of the specified <code>PermissionSet</code> .
<code>IsUnrestricted</code>	Returns true if the <code>PermissionSet</code> represents an unrestricted permission set; otherwise, returns false.
<code>PermitOnly</code>	Adds a <code>PermitOnly</code> override to the current stack frame for all permission objects in the <code>PermissionSet</code> -see Section 7.2.6 in this chapter for details.
<code>RemovePermission</code>	Removes the permission of the specified <code>System.Type</code> from the <code>PermissionSet</code> and returns it to the caller.
<code>SetPermission</code>	Adds a specified <code>IPermission</code> to the <code>PermissionSet</code> . If an <code>IPermission</code> of the specified type already exists, it is replaced by the new one.
<code>ToString</code>	Returns a <code>System.String</code> containing an XML representation of the <code>PermissionSet</code> and its contents.
<code>ToXml</code>	Returns a <code>SecurityElement</code> containing an XML object model representing the state of the <code>PermissionSet</code> .

Member	Description
Union	Creates a new <code>PermissionSet</code> that is the union of the current <code>PermissionSet</code> and another.

The runtime uses permission sets extensively during policy resolution (which we discuss in Chapter 8); you will find this functionality invaluable as you work more with permissions.

Example 7-3 creates and populates two `PermissionSet` objects:

- `PermissionSetOne`, which contains a `SecurityPermission` representing the ability to control evidence (`SecurityPermissionFlag.ControlEvidence`), a `FileIOPermission` representing the ability to read from directory `C:\`.
- `PermissionSetTwo`, which contains an unrestricted `SecurityPermission`, and a `FileIOPermission` representing the ability to read and write to the file `C:\Test.txt`.

Example 7-3. Creating a permission set from an intersection of permission sets

```
# C#

// Create and populate PermissionSetOne
PermissionSet PermissionSetOne =
    new PermissionSet(PermissionState.None);

PermissionSetOne.AddPermission(
    new SecurityPermission(SecurityPermissionFlag.ControlEvidence));
PermissionSetOne.AddPermission(
    new FileIOPermission(FileIOPermissionAccess.Read, @"C:\"));

// Create and populate PermissionSetTwo
PermissionSet PermissionSetTwo =
    new PermissionSet(PermissionState.None);

PermissionSetTwo.AddPermission(
    new SecurityPermission(PermissionState.Unrestricted));
PermissionSetTwo.AddPermission(
    new FileIOPermission(FileIOPermissionAccess.Read
        | FileIOPermissionAccess.Write, @"C:\Test.txt"));

# Visual Basic .NET

' Create and populate PermissionSetOne
Dim PermissionSetOne As PermissionSet = _
    New PermissionSet(PermissionState.None)

PermissionSetOne.AddPermission( _
    New SecurityPermission(SecurityPermissionFlag.ControlEvidence))
PermissionSetOne.AddPermission( _
```

```
New FileIOPermission(FileIOPermissionAccess.Read, "C:\")
```

```
' Create and populate PermissionSetTwo
Dim PermissionSetTwo As PermissionSet = _
New PermissionSet(PermissionState.None)
```

```
PermissionSetTwo.AddPermission( _
New SecurityPermission(PermissionState.Unrestricted))
PermissionSetTwo.AddPermission( _
New FileIOPermission(FileIOPermissionAccess.Read _
Or FileIOPermissionAccess.Write, "C:\Test.txt"))
```

Example 7-4 creates a third permission set named `PermissionSetThree` from the union of `PermissionSetOne` and `PermissionSetTwo`.

Example 7-4. Creating a permission set from a union of permission sets

```
# C#
```

```
// Create PermissionSetThree
PermissionSet PermissionSetThree =
    PermissionSetOne.Union(PermissionSetTwo);
```

```
# Visual Basic .NET
```

```
' Create PermissionSetThree
Dim PermissionSetThree As PermissionSet = _
PermissionSetOne.Union(PermissionSetTwo)
```

Calling `PermissionSetThree.ToString` produces the following output, which, incidentally, is the same as calling `PermissionSetThree.ToXml.ToString`. You can see that the resulting `FileIOPermission` lists both of the permissions we specified, whereas the `SecurityPermission` shows an unrestricted state, which includes the `ControlEvidence` ability:

```
<PermissionSet class="System.Security.PermissionSet"
    version="1">
    <IPermission class="System.Security.Permissions.FileIOPermission,
mscorlib, Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
        version="1"
        Read="C:\"
        Write="C:\Test.txt"/>
    <IPermission class="System.Security.Permissions.SecurityPermission,
mscorlib, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"
        version="1"
        Unrestricted="true"/>
</PermissionSet>
```

If instead you had created `PermissionSetThree` as the intersection of `PermissionSetOne` and `PermissionSetTwo` using the `Intersect` method, as in Example 7-5, the output from `PermissionSetThree.ToString` would be very different.

Example 7-5. Creating a permission set from an intersection of permission sets

```
# C#

// Create PermissionSetThree
PermissionSet PermissionSetThree =
    PermissionSetOne.Intersect(PermissionSetTwo);

# Visual Basic .NET

' Create PermissionSetThree
Dim PermissionSetThree As PermissionSet = _
    PermissionSetOne.Intersect(PermissionSetTwo)
```

Notice that `FileIOPermission` now includes only the ability to `Read` from `C:\Test.txt` and not to write to it. Also, you are left only with the ability to implement `ControlEvidence` from the `SecurityPermission`:

```
<PermissionSet class="System.Security.PermissionSet"
    version="1">
    <IPermission class="System.Security.Permissions.FileIOPermission, mscorlib,
Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"
        version="1"
        Read="C:\Test.txt"/>
    <IPermission class="System.Security.Permissions.SecurityPermission, mscorlib,
Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"
        version="1"
        Flags="ControlEvidence"/>
</PermissionSet>
```

When creating the union and intersection of permission sets, the `PermissionSet` object relies on the `Union` and `Intersect` methods of the individual `IPermission` objects. The logic used by each `IPermission` class to calculate unions and intersections is class-dependent; consult the .NET Framework SDK documentation for details.

The `NamedPermissionSet` class extends `PermissionSet` but includes only two additional members: the `Name` property gets and sets a name for the `NamedPermissionSet`, and the `Description` property gets and sets the description for the `NamedPermissionSet`. You will use `NamedPermissionSet` objects primarily when working with security policy, which we discuss in Chapter 8.

The `System.Security.Permissions.PermissionSetAttribute` class enables you to use permission sets with declarative security syntax. As with permission attributes, you must use the values of the `SecurityAction` enumeration to specify the security statement you want to invoke. To specify the contents of the permission set, you use the `Name`, `XML`, and `File` properties of `PermissionSetAttribute`.

The `Name` property allows you to specify the name of one of the immutable named permission sets included in .NET's default security policy, which we discuss in Chapter 9. Table 7-6 lists the possible

values and gives an indication of the permissions each set contains.

Table 7-6. Named permission sets

Permission set	Description
<code>FullTrust</code>	Unrestricted access to all operations and resources
<code>SkipVerification</code>	No access to any operation or resource other than permission to skip verification, which we discuss in Chapter 4
<code>Execution</code>	No access to any operation or resource other than the permission to execute
<code>Nothing</code>	No access to any operation or resource, not even the permission to execute
<code>LocalIntranet</code>	A set of permissions defined in the default security policy deemed by Microsoft to be appropriate for code loaded from the local intranet
<code>Internet</code>	A set of permissions defined in the default security policy deemed by Microsoft to be appropriate for code loaded from the Internet

The following example demonstrates a declarative security demand using the `Internet` named permission set:

```
# C#

[assembly:PermissionSet(SecurityAction.Demand,Name = "Internet")]

# Visual Basic .NET

<assembly: PermissionSet(SecurityAction. Demand,Name := "Internet")> _
```

The `XML` property allows you to provide a `String` containing an XML representation of the permission set you want to use. The correct format of the XML can be seen using `PermissionSet.ToString`, which we did earlier. The following example executes a declarative security demand on a permission set containing an unrestricted `FileIOPermission`. Although you cannot show it correctly here because of the length of the text, you should not have line breaks in the middle of the quoted strings within the XML text:

```
# C#

[PermissionSet(SecurityAction.Demand, XML = "<PermissionSet class=\"System.Security.
PermissionSet\" version=\"1\"><IPermission class=\"System.Security.Permissions.
FileIOPermission, mscorlib, Version=1.0.3300.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089\" version=\"1\" Unrestricted=\"true\"/></
PermissionSet>")]

# Visual Basic .NET

<PermissionSet(SecurityAction.Demand, XML := "<PermissionSet class="System.Security.
PermissionSet" version="1"><IPermission class="System.Security.Permissions.
FileIOPermission, mscorlib, Version=1.0.3300.0, Culture=neutral,
```

```
PublicKeyToken=b77a5c561934e089" version="1" Unrestricted="true"/></
PermissionSet">)> _
```

Notice that the use of the `XML` property can be a little unwieldy even for small permission sets. The `File` property allows you to specify the name of a file that contains the XML representation of the permission set. The following example uses the contents of the `Test.xml` file from the local directory to execute a declarative security demand. The `Test.xml` file content is used during compilation of your code but is not required afterwards:

```
# C#

[PermissionSet(SecurityAction.Demand, File = "Test.xml")]

# Visual Basic .NET

<PermissionSet(SecurityAction.Demand, File = "Test.xml")> _
```

7.2.4 Programming Permission Requests

Earlier in Section 7.1.2, we explained that CAS supports three types of permission requests, which are represented by the following members of the `SecurityAction` enumeration: `RequestMinimum`, `RequestOptional`, and `RequestRefuse`.

Permission requests are evaluated as the runtime loads an assembly and are expressed only using declarative security syntax. Because permission requests define the security requirements of your assembly, they must be declared using assembly scope attributes. Each statement takes a permission, or set of permissions, that can include any of the standard code-access and identity permission classes. After the runtime calculates the grant set of an assembly, it looks at any permission request statements in the assembly and applies them, as described in the following sections.

7.2.4.1 Requesting minimum permission

If you are writing code that must have access to a certain set of permissions in order to function correctly, the best tactic is to configure minimum-permission requests in your assembly. This is especially true if your code requires permissions that are not normally granted under default security policy. For example, if your code writes to the Windows registry but you expect people to run it over the Internet, many people will have problems running your code, because default Internet policy does not give access to write to the registry. We discuss the permission granted by the default security policy in Chapter 9.

By including a `RequestMinimum` statement in your assembly, you can specify that your code requires permission to write to the registry. This will not alter the fact that your code does not have the permission, but it will stop your application from loading; the runtime will throw a `System.Security.Policy.PolicyException` and identify the permission that is required. Knowing that there is a problem, the user or security administrator can view the assembly's permission requests with the `Permview.exe` tool (see Chapter 9) and decide whether to make the necessary security policy changes.

The use of `RequestMinimum` statements also allows you to simplify your code. Because the runtime will not load your assembly if it does not have the specified minimum set of permissions, you do not

need to place `SecurityException` handling code around every secured operation you perform. The fact that the code is running is proof that it has the required permissions.

The following code demonstrates how to declare a minimum-permission request for the `FileIOPermission` to read from the `C:\config` directory and have full access to the `C:\development` directory. Note that you cannot include both the `Read` and `All` properties in a single statement; the `FileIOPermissionAttribute` requires that you use a separate attribute for each path requested:

```
# C#

[assembly:FileIOPermission(SecurityAction.RequestMinimum,
    Read = @"C:\config")]
[assembly:FileIOPermission(SecurityAction.RequestMinimum,
    All = @"C:\development")]

# Visual Basic .NET

<assembly:FileIOPermission(SecurityAction.RequestMinimum, _
Read := "C:\config")> _

<assembly:FileIOPermission(SecurityAction.RequestMinimum, _
All := "C:\development")> _
```

7.2.4.2 Requesting optional permissions

The `RequestOptional` statement specifies the maximum set of permissions that the runtime should grant to your assembly regardless of security policy configuration. The runtime will still load your assembly if it cannot grant all of the permissions specified in the `RequestOptional` statement, but it will never grant more than those permissions specified.

The `RequestOptional` statement serves two principal purposes. First, you can request permissions that your application will make use of if they are granted. This could be to support noncore functionality that is made available only if the runtime grants the necessary permissions. Second, `RequestOptional` statements provide an excellent mechanism to limit the security exposure of your code. By minimizing the functionality and resources to which your code has access, you can ensure that bugs in your code cannot inadvertently destroy or modify protected resources. In addition, you reduce the chance that malicious users can use your code as a gateway to execute protected operations surreptitiously.

The major problem with optional permission requests is that the runtime does not indicate to the assembly which (if any) of the optional permissions requested it actually granted. You must either use exception handling around the code where you rely on the optional permissions or test whether the assembly has each of the permissions using the `SecurityManager.IsGranted` method, which we discuss in Chapter 8.

The following `RequestOptional` statement requests full access to the hard drive through an unrestricted `FileIOPermission`:

```
# C#

[assembly:FileIOPermission(PermissionState.Unrestricted)]
```

```
# Visual Basic .NET

<assembly:FileIOPermission(PermissionState.Unrestricted)> _
```

7.2.4.3 Refusing permissions

The `RequestOptional` statement enables you to define an upper limit on the permission set that the runtime will grant to your assembly. However, if there is a specific permission or set of permissions you are concerned about, then it is easy to ensure that they are never granted to your assembly using a `RequestRefuse` statement. Even if the runtime would have normally granted the permissions to the assembly based on policy resolution, the permission will be removed if specified in the `RequestRefuse` statement:

The following `RequestRefuse` statement refuses all access to the hard drive using an unrestricted `FileIOPermission` :

```
# C#

[assembly:FileIOPermission(PermissionState.Unrestricted)]

# Visual Basic .NET

<assembly:FileIOPermission(PermissionState.Unrestricted)> _
```

7.2.5 Programming Permission Demands

In Section 7.1.4 , we discussed the basic security demand `Demand` , which is the most commonly used mechanism of enforcing code-access security. In addition, CAS supports two other kinds of security demands: `LinkDemand` and `InheritanceDemand`

You will use `LinkDemand` and `InheritanceDemand` less frequently than `Demand` , but under certain circumstances, they provide invaluable mechanisms to protect your code from misuse. In the following sections, we describe the purpose of the `LinkDemand` and `InheritanceDemand` statements and demonstrate how to use them. First, we complete our coverage of `Demand` statements.

7.2.5.1 Demand

The `Demand` statement causes the runtime to walk the stack and ensures that not only the immediate caller, but all previous callers have a set of demanded permissions. You can use either declarative or imperative syntax to invoke a `Demand` . A declarative `Demand` occurs as soon as code invokes the protected method, whereas an imperative `Demand` occurs where you position the statement in your code.

You can apply declarative demands to classes or individual functional members, such as methods, properties, and events. If you use declarative syntax to specify a `Demand` on a class declaration, the same `Demand` is enforced on every class member. To override a class-level `Demand` , simply specify a different `Demand` on the individual members.

The following code demonstrates the use and syntax of the `Demand` statement. We have applied a declarative `Demand` for the "Internet" permission set to the `SomeClass` class, but `MethodA` overrides

this `Demand` with its own declarative `Demand` for an unrestricted `FileIOPermission`. The class-level `Demand` is still enforced on `MethodB`, but within the body of `MethodB`, we make an additional imperative `Demand` for an unrestricted `FileIOPermission`:

```
# C#

using System.Security;
using System.Security.Permissions;

[PermissionSet(SecurityAction.Demand, Name = "Internet")]
public class SomeClass {

    [FileIOPermission(SecurityAction.Demand, Unrestricted = true)]
    public void MethodA( ) {

        //...
    }

    public void MethodB( ) {

        // Create a new unrestricted FileIOPermission object
        FileIOPermission f =
            new FileIOPermission(PermissionState.Unrestricted);

        try {

            // Invoke security demand
            f.Demand( );

        } catch (SecurityException se) {

            // Handle exception condition
        }
    }
}

# Visual Basic .NET

Imports System.Security
Imports System.Security.Permissions

<PermissionSet(SecurityAction.Demand, Name := "Internet")> _
Public Class SomeClass

    <FileIOPermission(SecurityAction.Demand, Unrestricted := True)> _
    Public Sub MethodA( )
        '...
    End Sub

    Public Sub MethodB( )

        ' Create a new unrestricted FileIOPermission object
```

```

Dim f As FileIOPermission = _
New FileIOPermission(PermissionState.Unrestricted)

Try
    ' Invoke security demand
    f.Demand( )
Catch se As SecurityException
    ' Handle exception condition
End Try
End Sub
End Class

```

7.2.5.2 LinkDemands

A `LinkDemand` is similar to a `Demand`, but it is evaluated only when a caller first links to your code and only checks the permissions of the immediate caller as opposed to the permissions of all callers on the call stack. `LinkDemand` provides a lightweight alternative to the `Demand` and is used to reduce the performance hit of performing stack walks on frequently called and time-critical methods. However, because the permissions of all callers are not checked, `LinkDemand` leaves your code susceptible to attacks.

Because the runtime resolves `LinkDemand` security before your code is running, you can only invoke `LinkDemand` statements using declarative syntax. Like the `Demand`, if you apply a `LinkDemand` to a class declaration, it is enforced on all class members. If you also apply `LinkDemand` statements to individual class members, the runtime evaluates the class- and member-level `LinkDemand` statements sequentially. The following code demonstrates the syntax of `LinkDemand` to allow only code loaded from the `MyComputer` Internet Explorer Zone to call method `MethodA`:

```

# C#

[ZoneIdentityPermission(SecurityAction.LinkDemand,
Zone = SecurityZone.MyComputer)]
public void MethodA( ) {
// ...
}

# Visual Basic .NET

<ZoneIdentityPermission(SecurityAction.LinkDemand, _
Zone := SecurityZone.MyComputer)> _
Public Sub MethodA( )
' ...
End Sub

```

Calls to the public and protected members of strong-named assemblies result in an implicit `LinkDemand` for the `FullTrust` permission set. This means that only fully trusted code can call the methods of any strong-named assembly, which is problematic for most code not run from the local machine. Applying the `System.Security.AllowPartiallyTrustedCallers` attribute to a strong-named assembly disables this implicit `LinkDemand`. See Chapter 2 for more details on using the `AllowPartiallyTrustedCallers` attribute.

7.2.5.3 Inheritance demands

The `InheritanceDemand` statement allows you to restrict which code can extend your classes. For example, you may want to allow only code that is signed with your company's publisher certificate to extend a set of business-critical classes. In this case, apply the following `InheritanceDemand` to your class definition (assuming `certificate.cer` contains your company's publisher certificate):

```
# C#

[PublisherIdentityPermission(SecurityAction.InheritanceDemand,
    CertFile = "certificate.cer")]
public class SomeClass {
//...
}

# Visual Basic .NET

<PublisherIdentityPermission(SecurityAction.InheritanceDemand, _
CertFile := "certificate.cer")> _
Public Class SomeClass
'...
End Class
```

The runtime enforces the `InheritanceDemand` when it loads an assembly containing a class that extends the protected class. If the loaded assembly does not have the demanded permission (in this case, the necessary `PublisherIdentityPermission`), the runtime will throw a `SecurityException`.

You can also apply an `InheritanceDemand` to individual methods, properties, and events to define the permissions code it must have if it wants to override the protected member.

Whether applied at the class or member level, the `InheritanceDemand` is transitive—all classes derived directly or indirectly from the protected class must meet the requirements specified in the `InheritanceDemand`.

7.2.6 Manipulating Stack Walks

In Section 7.1.4.1, we described how the runtime uses the call stack in response to a security demand to ensure that every caller on the stack has the demanded permission or set of permissions. This is normally desirable, but in some instances, you will need to use stack-walk overrides to change this default behavior. CAS supports three different stack-walk overrides: `Assert`, `Deny`, and `PermitOnly`.

You can use either declarative or imperative syntax to invoke these overrides, and can specify either individual permissions or a permission set. Each statement causes the runtime to annotate the stack frame of the current method with the type of override and the set of permissions to which it applies. In the course of execution, when a stack walk occurs, the runtime uses the annotation information to modify the default stack walk behavior. If a single stack frame has different types of overrides in effect, the runtime applies the `PermitOnly`, then the `Deny`, and finally the `Assert`.

There can be only one of each type of override in effect on each stack frame. Before adding a new

override of the same type, you must *revert* the existing override or the runtime throws a `SecurityException`; see Section 7.2.6.4 later in this section for details.

When using declarative syntax to express `Assert`, `Deny`, and `PermitOnly` operations, apply them at the class, method, or property level. If you specify the security demand at the class level, the same override is applied to every member of the class. Applying a different override to an individual method will replace the default class-level override.



The .NET Framework SDK documentation states that stack-walk overrides should not be used in class constructors. The inconsistent state of the call stack during object construction may lead to undesirable results.

7.2.6.1 Assert

If you write code that accesses a protected resource or action, it is unreasonable to expect that security policy will grant all code that uses your code the same level of permission. `Assert` allows a method to override the normal stack-walk process and vouch for the authority of the callers above it on the stack. During a stack walk, if the demanded permission matches or is a subset of the asserted permissions, then the stack walk terminates successfully at the stack frame of the method that made the assertion. For example, in Figure 7-6 `MethodB` asserts an unrestricted `FileIOPermission`.

Figure 7-6. The effect of `Assert` on a stack walk

When `MethodD` calls the static `System.IO.File.Delete` method to delete the `C:\Test.txt` file, `Delete` makes a security demand for the permission to write to `C:\Test.txt`, which is similar to the following code:

```
# C#

FileIOPermission p =
    new FileIOPermission(FileIOPermissionAccess.Write, @"C:\Test.txt");
p.Demand( );

# Visual Basic .NET

Dim p As New FileIOPermission(FileIOPermissionAccess.Write, "C:\Test.txt")
p.Demand( )
```

Assuming that both `MethodD` and `MethodC` have the necessary permission, when the stack walk reaches the stack frame for `MethodB`, it compares the demanded `FileIOPermission` with the asserted unrestricted `FileIOPermission`. Because permission to write to the `C:\Test.txt` file is a subset of the unrestricted `FileIOPermission` permission, the stack walk terminates as if all callers or the call stack had been found to have the demanded permission. The true grant sets of `AppDomainY`, `MethodA`, and `AppDomainX` are irrelevant.

If the security demand that initiated the stack walk was called on a permission set and the asserted permissions satisfy only part of the security demand, the stack walk will continue, but the remaining stack frames will only be compared against the subset of permissions not satisfied by the `Assert`.

The following code shows the imperative security statement to `Assert` the unrestricted `FileIOPermission` described in this example:

```
# C#

public void MethodB( ) {

    try {

        // Instantiate and Assert FileIOPermission
        FileIOPermission p = new FileIOPermission(
            PermissionState.Unrestricted);
        p.Assert( );

    } catch (SecurityException se) {

        // Handle SecurityException
    }
}

# Visual Basic .NET

Public Sub MethodB( )

    Try

        ' Instantiate and Assert FileIOPermission
        Dim p As FileIOPermission = _
        New FileIOPermission(PermissionState.Unrestricted)
        p.Assert( )

    Catch se As SecurityException

        ' Handle SecurityException
    End Try
End Sub
```

Note that the `Assert` call can throw a `System.Security.SecurityException` in three instances:

- If the assembly containing the asserting method does not have the

`SecurityPermission.Assertion` permission

- If the assembly containing the asserting method does not have the permission it is asserting
- If an `Assert` override is already present on the callers stack frame, in which case the caller must first revert to the existing `Assert`

The following code shows the equivalent declarative syntax to `Assert` an unrestricted `FileIOPermission`:

```
# C#

[FileIOPermission(SecurityAction.Assert, Unrestricted = true)]
public void MethodB( ) {}

# Visual Basic .NET

<FileIOPermission(SecurityAction.Assert, Unrestricted := True)> _
Public Sub MethodB( )
End Sub
```

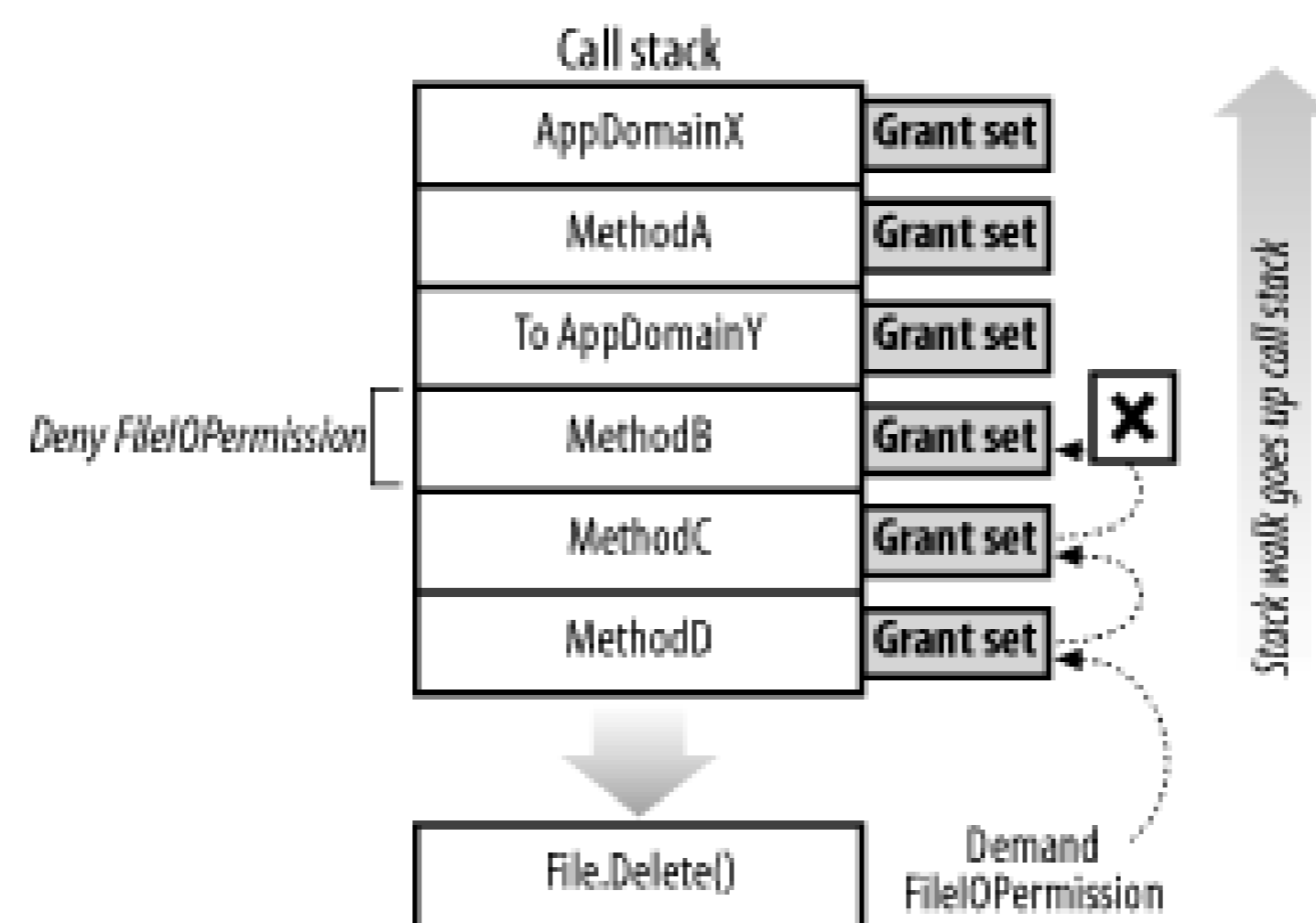
Before using `Assert`, you should be confident that your code cannot be used as a gateway through which malicious code can access protected resources. This is particularly true because the `Assert` override is frequently used with highly trusted permissions not normally given to all applications, such as the ability to execute unmanaged code. Used carelessly, `Assert` can open up gaping holes in the overall security of the runtime.

7.2.6.2 Deny

The `Deny` is the opposite of an `Assert`. Instead of vouching for the authority of the callers higher on the call stack and terminating a stack walk with a positive response, the runtime throws a `SecurityException` to the code that initiated the `Demand` and terminates the stack walk.

For example, in Figure 7-7 `MethodB` denies an unrestricted `FileIOPermission`. This causes any stack walk looking for access to the hard drive to fail. When `MethodD` calls the static `System.IO.File.Delete` method to delete the `C:\Test.txt` file, `Delete` makes a security demand for the permission to write to the `C:\Test.txt` file.

Figure 7-7. The effect of Deny on a stack walk



Assuming the stack walk reaches `MethodB`, it compares the demanded `FileIOPermission` with the denied unrestricted `FileIOPermission`. Because permission to write to `C:\Test.txt` file is a subset of the unrestricted `FileIOPermission` permission, the stack walk terminates and throws a `SecurityException` in the `Delete` method.

The following code shows the imperative security syntax to `Deny` the unrestricted `FileIOPermission` described in the example. Note that the `Deny` call can throw a `SecurityException` if there is already a `Deny` override in place for the current stack frame:

```
# C#
```

```
public void MethodB( ) {
    try {
        // Instantiate and Deny FileIOPermission
        FileIOPermission p = new FileIOPermission(
            PermissionState.Unrestricted);
        p.Deny( );
    } catch (SecurityException se) {
        // Handle SecurityException
    }
}
```

```
# Visual Basic .NET
```

```
Public Sub MethodB( )
    Try
        ' Instantiate and Deny FileIOPermission
        Dim p As FileIOPermission = _
        New FileIOPermission(PermissionState.Unrestricted)
        p.Deny( )
    Catch se As SecurityException
```

```

        ' Handle SecurityException
    End Try
End Sub

```

The following example shows the equivalent declarative syntax to `Deny` an unrestricted `FileIOPermission`:

```

# C#

[FileIOPermission(SecurityAction.Deny, Unrestricted = true)]
public void MethodB( ) {}

# Visual Basic .NET

<FileIOPermission(SecurityAction.Deny, Unrestricted := True)> _
Public Sub MethodB( )
End Sub

```

7.2.6.3 PermitOnly

The `PermitOnly` stack-walk override is like an inverted `Deny`. Instead of specifying the permissions that should be denied, you specify only those permissions that permit the stack walk to continue unaffected. This provides a more concise way of denying large sets of permissions without the need to specify them all in a `Deny` override.

In the following code, `MethodB` will cause a stack walk to terminate (and a `SecurityException` to be thrown) if methods further down the call stack demand any permission other than a `FileIOPermission`. In this instance, the stack walk illustrated in Figure 7-7 passes through the overrides of `MethodB` without effect, because the permission to write to the `C:\Test.txt` file is a subset of the unrestricted `FileIOPermission` permission:

```

# C#

public void MethodB( ) {

    try {

        // Instantiate and PermitOnly FileIOPermission
        FileIOPermission p = new FileIOPermission(
            PermissionState.Unrestricted);
        p.PermittOnly( );

    } catch (SecurityException se) {

        // Handle SecurityException
    }
}

# Visual Basic .NET

Public Sub MethodB( )

```

```

Try

    ' Instantiate and PermitOnly FileIOPermission
    Dim p As FileIOPermission = _
    New FileIOPermission(PermissionState.Unrestricted)
    p.PermittOnly ( )

Catch se As SecurityException

    ' Handle SecurityException
End Try
End Sub

```

The following example shows the equivalent declarative syntax to `PermitOnly` an unrestricted `FileIOPermission`:

```

# C#

[FileIOPermission(SecurityAction.PermittOnly, Unrestricted = true)]
public void MethodB( ) {}

# Visual Basic .NET

<FileIOPermission(SecurityAction.PermittOnly, Unrestricted := True)> _
Public Sub MethodB( )
End Sub

```

7.2.6.4 Reverting overrides

Using the static methods of the `CodeAccessPermission` class listed in Table 7-7, code removes the stack frame annotations made by previous calls to `Assert`, `Deny`, and `PermitOnly`. If you call any of these methods and there is none of the specified override in effect, the runtime throws a `System.ExecutionEngineException` that you must catch and handle appropriately.

Table 7-7. Removing stack-walk overrides

Static method	Description
<code>RevertAll</code>	Causes the removal of all previous <code>Assert</code> , <code>Deny</code> , or <code>PermitOnly</code> overrides on the current stack frame
<code>RevertAssert</code>	Causes the removal of all previous <code>Assert</code> overrides on the current stack frame
<code>RevertDeny</code>	Causes the removal of all previous <code>Deny</code> overrides on the current stack frame
<code>RevertPermitOnly</code>	Causes the removal of all previous <code>PermitOnly</code> overrides on the current stack frame

Because the revert statements are all issued through static methods of the `CodeAccessPermission` class, there are no declarative syntax alternatives.

[Team LiB]

[\[Team LiB \]](#)

7.3 Extending the .NET Framework

CAS is fully extensible and allows you to create your own permission classes that integrate with the security framework to provide capabilities equivalent to the standard permission classes. The creation of custom permissions is relatively straightforward. The complexity of the identity, actions, or resources that your permission class represents defines how difficult the development task is. In the following sections, we demonstrate how to create and implement custom code-access permissions.

7.3.1 Creating Custom Code-Access Permissions

All code-access permissions must meet the following requirements:

- Implement the `IPermission` interface
- Implement the `IStackWalk` interface
- Implement the `ISecurityEncodable` interface
- Implement the `IUnrestrictedPermission` interface
- Provide a constructor that takes a `PermissionState` argument
- Be serializable

The simplest way to meet these requirements is to derive your permission class from the abstract `CodeAccessPermission` class, which we discussed in [Section 7.2.2.1](#) earlier in this chapter. `CodeAccessPermission` declares implementation of the `IPermission`, `IStackWalk`, and `ISecurityEncodable` interfaces, but only provides concrete implementations of the `IStackWalk` interface methods. Because the logic of the following methods depends on permission-specific state, you must provide implementations of them in your class:

`IPermission` methods

- `Copy`
- `Intersect`
- `IsSubsetOf`
- `Union`

`ISecurityEncodable` methods

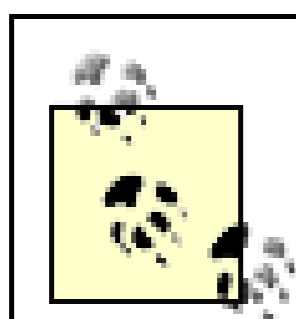
- `FromXml`

- ToXml

IUnrestrictedPermission method

- IsUnrestrictedPermission

To meet the requirements of serialization, it is usually enough to apply the `SerializableAttribute` to your class definition. However, if your permission class requires nonstandard serialization processing because of the complex internal state it maintains, implement the `ISerializable` interface.



The `CodeAccessPermission` class defines an `InheritanceDemand` for the `ControlEvidence` and `ControlPolicy` elements of the `SecurityPermission` class. The assembly containing your subclass of `CodeAccessPermission` must have these permissions, or the runtime throws a `SecurityException`.

To support declarative security syntax, create an attribute counterpart to your permission class. This involves creating a class that extends the `System.Security.Permissions.CodeAccessSecurityAttribute` class. Your class must:

- Implement a single constructor that takes a `SecurityAction` argument
- Override the `CreatePermission` method
- Implement public properties that enable the configuration of the attribute's state

7.3.1.1 Designing the RadioPermission class

To demonstrate the creation and use of custom code-access permissions, imagine that we need to develop a managed library that allows an application to control a PC-based radio tuner. The radio tuner could be a hardware attachment to the PC or software that plays Internet-based radio stations; the implementation details are unimportant, but the library must allow code to:

- Turn the radio on and off
- Adjust the volume of the radio
- Change the station the radio is tuned to

The goal is to create a custom code-access permission that you can use to control access to each of these actions. After all, you do not want to allow malicious code downloaded from competitive radio stations to casually change the tuning of the radio.

The best way to achieve fine-grained access control is to implement a code-access permission that represents a discrete set of actions; this is the same model used by the `SecurityPermission` class that we described in [Section 7.2.2.2](#) earlier in this chapter. Because the number of possible actions is small and well-defined, this model fits your needs well. Another common approach is to implement a permission hierarchy, where permission to perform a highly trusted action gives permission to

perform all lesser-trusted actions; however, the actions you need to represent in this case do not fit this model well.

To give the programmers using the custom permission flexibility, you want to provide both imperative and declarative syntax support. This will also allow application developers to build permission requests for your permission into their assemblies. Create two classes and one enumeration:

`RadioPermission`

The custom code-access permission that controls access to the managed library for the radio tuner

`RadioPermissionAttribute`

The attribute counterpart to `RadioPermission` that provides declarative syntax support

`RadioAction`

An enumeration that defines values that represent each of the controlled actions

For convenience, we include all three types in a single source file named `Radio.cs` (C#) and `Radio.vb` (Visual Basic .NET). We will build this into a library named `Radio.dll`, which programmers will use in the development of the managed radio tuner library, the configuration of .NET security policy, and to make permission requests in assemblies that access the managed library. Although the code is straightforward, it is lengthy, so we break the listings into manageable sections for explanation.

7.3.1.2 Defining imports and assembly scope attributes

Aside from importing the necessary namespace references, the first thing to do is to specify the assembly scope attributes `AssemblyKeyFile` and `AssemblyVersion` to give `Radio.dll` a strong name. Because `Radio.dll` contains security classes used to extend CAS, you must configure it as a fully trusted assembly in the runtime's security policy; we discuss fully trusted assemblies in [Chapter 8](#). For this example, the `AssemblyKeyFile` attribute references a key file named `Keys.snk`, which you can generate later using the .NET Strong Name tool (`Sn.exe`):

```
# C#

using System;
using System.Reflection;
using System.Security;
using System.Security.Permissions;

[assembly:AssemblyKeyFile("Keys.snk")]
[assembly:AssemblyVersion("1.0.0.0")]

namespace OReilly.ProgrammingDotNetSecurity {

# Visual Basic .NET

Imports System
Imports System.Reflection
Imports System.Security
```

```
Imports System.Security.Permissions

<assembly:AssemblyKeyFile("Keys.snk")>
<assembly:AssemblyVersion("1.0.0.0")>

Namespace OReilly.ProgrammingDotNetSecurity
```

7.3.1.3 Defining the RadioAction enumeration

`RadioPermission` represents control of a discrete set of actions. The `RadioAction` enumeration represents each of the actions to which `RadioPermission` controls access. `RadioAction` also includes a value to represent no permission and a value to represent full or unrestricted permission.

To simplify the implementation of your permission model, apply the `Flags` attribute to the `RadioAction` enumeration. The `Flags` attribute allows you to use the values of the `RadioAction` enumeration as bit fields, enabling you to manipulate and compare `RadioAction` values with bitwise operators. [Table 7-8](#) lists the members of `RadioAction` and shows their binary and hex values. From the binary values, it is easy to see how `RadioAction` identifies the individual actions to which a `RadioPermission` object represents access.

Table 7-8. Values of the RadioAction enumeration

Value	Binary value	Hex value	Description
None	000	0	No access to perform any actions with the radio
StartStop	001	1	Permission to turn the radio on and off
Channel	010	2	Permission to change the radio's channel
Volume	100	4	Permission to change the radio's volume
All	111	7	Permission to perform any of the above actions

Finally, because you use `RadioAction` values inside `SecurityPermission` objects (which must be serializable), you also mark `RadioAction` as `Serializable`:

```
# C#

// Define enumeration to represent the set of possible permissions.
[Flags, Serializable]
public enum RadioAction {
    None          = 0x00,      // No permissions
    StartStop    = 0x01,      // Permission to start and stop the radio
    Channel      = 0x02,      // Permission to change the channel
    Volume       = 0x04,      // Permission to change the volume
    All          = 0x07       // All permissions
}

# Visual Basic .NET
```

```
' Define enumeration to represent the set of possible permissions.
<Flags, Serializable> _
Public Enum RadioAction
    None          = &H00    ' No permissions
    StartStop    = &H01    ' Permission to start and stop the radio
    Channel      = &H02    ' Permission to change the channel
    Volume       = &H04    ' Permission to change the volume
    All          = &H07    ' All permissions
End Enum
```

7.3.1.4 Defining the RadioPermission class

The class declaration for `RadioPermission` specifies that it is `sealed` (C#) and `NotInheritable` (Visual Basic .NET). There is no need to allow anybody to extend the `RadioPermission` class, and it stops malicious code from subclassing `RadioPermission` in an attempt to subvert security restrictions.

`RadioPermission` extends `CodeAccessPermission`, implements the `IUnrestricted` interface, and is `Serializable`. The `CodeAccessPermission` base class already implements the functionality defined in the `IStackWalk` interface so you do not have to implement the `Assert`, `Demand`, `Deny`, and `PermitOnly` methods; this simplifies the development of custom code-access permissions. However, you still need to implement the permission-specific `Copy`, `Intersect`, `IsSubsetOf`, and `Union` methods defined in `IPermission`, and the `FromXml` and `ToXml` methods defined in `ISecurityEncodable`, all of which `CodeAccessPermission` declares as abstract.

Declare a single private data member named `RadioActions` of type `RadioAction`. This holds a bit field representing the actions to which a specific `RadioPermission` object grants access. Implement the `Actions` property to provide controlled access to the private `RadioActions` member:

```
# C#

// Define the RadioPermission class.
[Serializable]
public sealed class RadioPermission :
    CodeAccessPermission, IUnrestrictedPermission {

    // Private member to signal permitted actions.
    private RadioAction RadioActions;

    // Property to provide access to the enabled radio actions.
    public RadioAction Actions {

        get {
            return RadioActions;
        }
        set {
            if ((value & (~RadioAction.All)) != 0) {
                throw new
                    ArgumentException("Invalid RadioAction value");
            } else {
                RadioActions = value;
            }
        }
    }
}
```

```
    }
}
```

```
# Visual Basic .NET
```

```
' Define the RadioPermission class.
<Serializable> _
Public NotInheritable Class RadioPermission
    Inherits CodeAccessPermission

    ' Private member to signal permitted actions.
    Private RadioActions As RadioAction

    ' Property to provide access to the enabled radio actions.
    Public Property Actions( ) As RadioAction
        Get
            Return RadioActions
        End Get
        Set (ByVal Value As RadioAction)
            If (value And (Not RadioAction.All)) <> 0 Then
                Throw New ArgumentException _
                    ("Invalid RadioAction value")
            Else
                RadioActions = value
            End If
        End Set
    End Property
```

Declare two constructors for `RadioPermission`. All code-access permission classes that extend `CodeAccessPermission` must implement a constructor that takes a `PermissionState` argument. Passing the constructor the value `PermissionState.None` creates a `RadioPermission` that represents no permissions by setting `RadioActions` to `RadioAction.None`, whereas passing the `PermissionState.Unrestricted` value creates a `RadioPermission` object representing full permission by setting `RadioActions` to `RadioAction.All`.

Because you use a bit field to represent each permitted action, it is easy to represent an unrestricted permission simply by switching on all action bits (`RadioAction.All` equals 111 in binary). In other permission models, you may need to implement a separate Boolean data member to represent an unrestricted state.

The second constructor takes a member of the `RadioAction` enumeration, confirms that it is valid, and sets the private `RadioActions` member to the value of the argument:

```
# C#
```

```
// Constructor that takes a PermissionState.
public RadioPermission (PermissionState state) {

    if (state == PermissionState.None) {
        this.RadioActions = RadioAction.None;
    } else if (state == PermissionState.Unrestricted) {
        this.RadioActions = RadioAction.All;
    }
}
```

```

        } else {
            throw new ArgumentException("state");
        }
    }

    // Constructor that takes a RadioAction specifying the set
    // of actions permitted by this RadioPermission object.
    public RadioPermission (RadioAction actions) {

        // Ensure we have a valid actions value.
        if ((actions & (~RadioAction.All)) != 0) {
            throw new ArgumentException("Invalid RadioAction value");
        } else {
            RadioActions = actions;
        }
        this.RadioActions = actions;
    }

```

```
# Visual Basic .NET
```

```

' Constructor that takes a PermissionState.
Public Sub New(ByVal state As PermissionState)

    If state = PermissionState.None Then
        Me.RadioActions = RadioAction.None
    Else If state = PermissionState.Unrestricted Then
        Me.RadioActions = RadioAction.All
    Else
        Throw New ArgumentException("state")
    End If
End Sub

' Constructor that takes a RadioAction specifying the set
' of actions permitted by this RadioPermission object.
Public Sub New(ByVal actions As RadioAction)

    ' Ensure we have a valid actions value.
    If (actions And (Not RadioAction.All)) <> 0 Then
        Throw New ArgumentException("Invalid RadioAction value")
    Else
        RadioActions = actions
    End If
    Me.RadioActions = actions
End Sub

```

7.3.1.5 Implementing the IUnrestrictedPermission interface

The `IUnrestrictedPermission` interface defines the `IsUnrestricted` method, which returns a Boolean indicating whether a permission object represents an unrestricted state. In the case of `RadioPermission`, you can easily calculate the response by comparing the value of the private

`RadioActions` member with the value `RadioAction.All` and returning the result:

```
# C#

// Return whether permission represents unrestricted access.
public bool IsUnrestricted( )
{
    return RadioActions == RadioAction.All;
}

# Visual Basic .NET

' Return whether permission represents unrestricted access.
Public Function IsUnrestricted( ) As Boolean
    Return RadioActions = RadioAction.All
End Function
```

7.3.1.6 Implementing the `IPermission` interface

Both the `IStackWalk` and `IPermission` interfaces define a method named `Demand`. The `CodeAccessPermission` base class implements all members of the `IStackWalk` interface, and you do not need to implement the `Demand` method from the `IPermission` interface, leaving you with the `Copy`, `Intersect`, `Union`, and `IsSubsetOf` methods to implement.

The `Copy` method returns a clone of the permission object. In the case of `RadioPermission`, this is a simple matter of calling one of the constructors and passing the current `RadioActions` value. If your permission class uses objects to represent state, `Copy` must perform a deep copy and create copies of all contained objects:

```
# C#

// Create a copy the RadioPermission.
public override IPermission Copy( ) {

    return new RadioPermission(RadioActions);
}

# Visual Basic .NET

' Create a copy the RadioPermission.
Public Overrides Function Copy( ) As IPermission

    Return New RadioPermission(RadioActions)
End Function
```

The `Intersect` method returns a new `IPermission` that represents the logical intersection of the current `RadioPermission` and another. Because `RadioPermission` represents permission to perform a discrete set of actions, the intersection of two `RadioPermission` objects is the set of actions permitted by *both* objects. For example, if the current `RadioPermission` grants `Volume` and `Channel` control and the other grants only `Volume` control, the intersection of the two is a `RadioPermission` that grants only `Volume` control. Because the `Flags` attribute allows you to treat values of the

`RadioAction` enumeration as bit fields, you can express the intersection logic using a bitwise `And` on the `RadioActions` members of the two original `RadioPermission` objects.

```
# C#
```

```
// Return an intersection of this and another RadioPermission
// object.
public override IPermission Intersect(IPermission target) {

    // Return null if the target argument is null.
    if (target == null) {
        return null;
    }

    // Ensure the target argument is another RadioPermission
    // object, if not throw an ArgumentException.
    if (target is RadioPermission) {

        RadioPermission r = (RadioPermission)target;

        // Calculate the intersection of the radio actions
        // from this and the target RadioPermission.
        RadioAction i =
            this.RadioActions & r.RadioActions;

        // Return a new RadioPermission.
        return new RadioPermission(i);

    } else {
        throw new ArgumentException("target");
    }
}
```

```
# Visual Basic .NET
```

```
' Return an intersection of this and another RadioPermission
' object.
Public Overrides Function Intersect(ByVal target As IPermission) _
As IPermission

    ' Return null if the target argument is null.
    If target Is Nothing Then
        Return Nothing
    End If

    ' Ensure the target argument is another RadioPermission
    ' object, if not throw an ArgumentException.
    If TypeOf target Is RadioPermission Then

        Dim r As RadioPermission = _
            CType(target, RadioPermission)
```



```

        ' Calculate the intersection of the radio actions
        ' from this and the target RadioPermission.
        Dim i As RadioAction = Me.RadioActions And _
            r.RadioActions

        ' Return a new RadioPermission.
        Return New RadioPermission(i)

    Else
        Throw New ArgumentException("target")
    End If
End Function

```

The `Union` method returns a new `IPermission` that represents the logical union of the current `RadioPermission` and another. For example, if one `RadioPermission` grants `Channel` control and the other grants `Volume` control, the union of the two is a `RadioPermission` that grants both `Volume` and `Channel` control. The implementation of the `Union` method is similar to the `Intersect` method, but uses a bitwise `Or` operator on the `RadioActions` members of the two original `RadioPermission` objects to calculate the union:

```
# C#
```

```

// Return the union of this and another RadioPermission object.
public override IPermission Union(IPermission other) {

    // Return null if the other argument is null.
    if (other == null) {
        return null;
    }

    // Ensure the other argument is another RadioPermission object
    // if not throw an ArgumentException.
    if (other is RadioPermission) {

        RadioPermission r = (RadioPermission)other;

        // Calculate the union of the radio actions
        // from this and the other RadioPermission.
        RadioAction u =
            this.RadioActions | r.RadioActions;

        // Return a new RadioPermission
        return new RadioPermission(u);
    } else {
        throw new ArgumentException("other");
    }
}

```

```
# Visual Basic .NET
```

```

' Return the union of this and another RadioPermission object.
Public Overrides Function Union(ByVal other As IPermission) _
As IPermission

    ' Return null if the other argument is null.
    If other Is Nothing Then
        Return Nothing
    End If

    ' Ensure the other argument is another RadioPermission object
    ' if not throw an ArgumentException.
    If TypeOf other Is RadioPermission Then

        Dim r As RadioPermission = CType(other, RadioPermission)

        ' Calculate the union of the radio actions
        ' from this and the other RadioPermission.
        Dim u As RadioAction = Me.RadioActions _
        Or r.RadioActions

        ' Return a new RadioPermission
        Return New RadioPermission(u)

    Else
        Throw New ArgumentException("other")
    End If
End Function

```

The `IsSubsetOf` method returns a Boolean value indicating whether the current `RadioPermission` is a subset of the `RadioPermission` passed as an argument to the method. For example, if the current `RadioPermission` allowed `Volume` control, and the one specified by the argument allowed both `Volume` and `Channel` control, then the current `RadioPermission` is a subset of the other. As with the `Intersect` and `Union` methods, we perform the subset relationship test using bitwise operators.

```
# C#
```

```

// Determines if this RadioPermission is a subset of the provided
// RadioPermission.
public override bool IsSubsetOf(IPermission target) {

    // Return false if the target argument is null.
    if (target == null) {

        return false;
    }

    // Ensure the target argument is another RadioPermission
    // object, if not throw an ArgumentException.
    if (target is RadioPermission) {

        RadioPermission r = (RadioPermission)target;
    }
}

```

```

        // Determine if the provided set of actions are
        // a subset of the current permissions actions.
        return (this.RadioActions
            & (~r.RadioActions)) == 0;

    } else {

        throw new ArgumentException("target");
    }
}

# Visual Basic .NET

' Determines if this RadioPermission is a subset of the provided
' RadioPermission.
Public Overrides Function IsSubsetOf _
    (ByVal target As IPermission) As Boolean

    ' Return false if the target argument is null.
    If target Is Nothing Then

        Return False
    End If

    ' Ensure the target argument is another RadioPermission
    ' object, if not throw an ArgumentException.
    If TypeOf target Is RadioPermission Then

        Dim r As RadioPermission = _
            CType(target, RadioPermission)

        ' Determine if the provided set of actions are
        ' a subset of the current permissions actions.
        Return (Me.RadioActions And _
            (Not r.RadioActions)) = 0
    Else
        Throw New ArgumentException("target")
    End If
End Function

```

7.3.1.7 Implementing the ISecurityEncodable interface

The `ToXml` method returns a `SecurityElement`, which we discussed in [Chapter 6](#). The `SecurityElement` contains a simple XML object model representing the state of the `RadioPermission` object. The runtime uses the `Tostring` method of `SecurityElement` to render permission objects to XML for writing to the security policy files, which we discuss in [Chapter 9](#). The XML representation is also useful when debugging CAS programming issues.

The XML representation of all code-access permissions must have a root element named "IPermission." The internal structure of this element is up to you, as long as the `FromXml` method

(discussed next) can accurately recreate the state of a permission object from the XML. Here is an example of the XML used to represent a `RadioPermission` object that represents access to the `StartStop` action only.

```
<IPermission class="OReilly.ProgrammingDotNetSecurity.RadioPermission,
Radio, Version=1.0.0.0, Culture=neutral, PublicKeyToken=cc5e18bc387194b3"
    version="1"
    StartStop="true"/>
```

The XML structure you have implemented for `RadioPermission` is consistent with that used by the standard permission classes. Use attributes, as opposed to child elements, to store state data, and include active state only, assuming missing values have their default values. If the `RadioPermission` object represents access to all actions, include a single `Unrestricted = true` attribute instead of including values for each of the individual actions. The `version` attribute identifies the XML format to provide backward compatibility in case future versions of the `RadioPermission` use different XML representations:

```
# C#
```

```
// Convert the Radio Permission to a SecurityElement.
public override SecurityElement ToXml( ) {

    // Create a new "IPermission" element.
    SecurityElement se = new SecurityElement("IPermission");

    // Add fully qualified type name for the RadioPermission.
    se.AddAttribute("class",this.GetType( ).AssemblyQualifiedName);

    // Add version of "1" to be consistent with other permission
    // classes
    se.AddAttribute("version", "1");

    // Add the radio action state, only write out active
    // permissions.
    if (this.IsUnrestricted( )) {

        // Write only the unrestricted attribute if the
        // permission is unrestricted.
        se.AddAttribute("Unrestricted", "true");
    } else {

        // Write out the individual actions that are granted.
        if ((RadioActions & RadioAction.StartStop) != 0) {
            se.AddAttribute("StartStop", "true");
        }
        if ((RadioActions & RadioAction.Channel) != 0) {
            se.AddAttribute("Channel", "true");
        }
        if ((RadioActions & RadioAction.Volume) != 0) {
            se.AddAttribute("Volume", "true");
        }
    }
}
```

```

        // Return the new SecurityElement.
        return se;
    }

# Visual Basic .NET

' Convert the Radio Permission to a SecurityElement.
Public Overrides Function ToXml( ) As SecurityElement

    ' Create a new "IPermission" element.
    Dim se As SecurityElement = New SecurityElement("IPermission")

    ' Add fully qualified type name for the RadioPermission.
    se.AddAttribute("class",Me.GetType( ).AssemblyQualifiedName)

    ' Add version of "1" to be consistent with other permission
    ' classes
    se.AddAttribute("version", "1")

    ' Add the radio action state, only write out active
    ' permissions.
    If Me.IsUnrestricted( ) Then

        ' Write only the unrestricted attribute if the
        ' permission is unrestricted.
        se.AddAttribute("Unrestricted", "true")
    Else

        ' Write out the individual actions that are granted.
        If (RadioActions And RadioAction.StartStop) <> 0 Then
            se.AddAttribute("StartStop", "true")
        End If
        If (RadioActions And RadioAction.Channel) <> 0 Then
            se.AddAttribute("Channel", "true")
        End If
        If (RadioActions And RadioAction.Volume) <> 0 Then
            se.AddAttribute("Volume", "true")
        End If
    End If

    ' Return the new SecurityElement.
    Return se
End Function

```

The `FromXml` method recreates the state of a `RadioPermission` object that was rendered previously to XML using the `ToXml` method. Check first to see if the XML represents an unrestricted permission; if it does not, look for each individual action:

```

# C#

// Extract state from a SecurityElement.

```



```

        ' Determine if the permission is unrestricted
        If e.Attribute("Unrestricted") = "true" Then
            RadioActions = RadioAction.All
        Else
            ' Look for each individual action attribute to
            ' build the set of permissions
            RadioActions = RadioAction.None

            If e.Attribute("StartStop") = "true" Then
                RadioActions = _
                RadioActions Or RadioAction.StartStop
            End If
            If e.Attribute("Channel") = "true" Then
                RadioActions = _
                RadioActions Or RadioAction.Channel
            End If
            If e.Attribute("Volume") = "true" Then
                RadioActions = _
                RadioActions Or RadioAction.Volume
            End If
        End If
    End If
End Sub
End Class

```

7.3.1.8 Defining the RadioPermissionAttribute class

We begin the declaration of the `RadioPermissionAttribute` class by using the `AttributeUsage` attribute to define the program elements you can apply the `RadioPermissionAttribute` to.

`RadioPermissionAttribute` extends the `CodeAccessSecurityAttribute` class, which provides the base class from which all permission attributes must extend. Implement a single constructor that all permission attributes extending `CodeAccessSecurityAttribute` must define. The constructor takes a value from the `SecurityAction` enumeration, which describes the CAS operation invoked by a declarative security statement; see [Section 7.2.1.2](#) for details:

```

# C#

[AttributeUsage(AttributeTargets.Method | AttributeTargets.Constructor
| AttributeTargets.Class | AttributeTargets.Struct
| AttributeTargets.Assembly, AllowMultiple = true,
Inherited = false )]
[Serializable]
public sealed class RadioPermissionAttribute :
    CodeAccessSecurityAttribute {

    // Constructor to take SecurityAction, simply calls base
    // class constructor.
    public RadioPermissionAttribute(SecurityAction action):
        base( action ) {}
}

```

```
# Visual Basic .NET
```

```
<AttributeUsage(AttributeTargets.Method Or _
AttributeTargets.Constructor Or AttributeTargets.Class Or _
AttributeTargets.Struct Or AttributeTargets.Assembly, _
AllowMultiple := True, Inherited := False), Serializable> _
Public NotInheritable Class RadioPermissionAttribute
    Inherits CodeAccessSecurityAttribute

    ' Constructor to take SecurityAction, simply calls base
    ' class constructor.
    Public Sub New(ByVal action As SecurityAction)
        MyBase.New(action)
    End Sub
```

As with `RadioPermission`, the `RadioPermissionAttribute` class contains a private data member named `RadioActions` (of type `RadioAction`) to identify which actions the `RadioPermissionAttribute` object represents. Four properties provide controlled access to configure the value of the private `RadioActions` member. The first property, named `Actions`, allows you to provide a `RadioAction` value to configure the state of all actions in a single call. The other three attributes, named `StartStop`, `Channel`, and `Volume`, allow you to configure the individual bits of `RadioActions` to set whether an individual action is permitted or not:

```
# C#
```

```
// Private member to signal permitted actions.
private RadioAction RadioActions = RadioAction.None;

// Property allows all actions to be configured in one call
// using values from the RadioAction enumeration
public RadioAction Actions {
    get { return RadioActions; }
    set { RadioActions = value; }
}

// Property to allow StartStop permission to be switched on
// and off
public bool StartStop {
    get { return (RadioActions & RadioAction.StartStop) != 0; }
    set {
        if (value) {
            RadioActions = RadioActions | RadioAction.StartStop;
        } else {
            RadioActions = RadioActions & ~RadioAction.StartStop;
        }
    }
}

// Property to allow Channel permission to be switched on
// and off
public bool Channel {
    get { return (RadioActions & RadioAction.Channel) != 0; }
}
```



```

        set {
            if (value) {
                RadioActions = RadioActions | RadioAction.Channel;
            } else {
                RadioActions = RadioActions & ~RadioAction.Channel;
            }
        }
    }

    // Property to allow Volume permission to be switched on
    // and off
    public bool Volume {
        get { return (RadioActions & RadioAction.Volume) != 0; }
        set {
            if (value) {
                RadioActions = RadioActions | RadioAction.Volume;
            } else {
                RadioActions = RadioActions & ~RadioAction.Volume;
            }
        }
    }
}

```

```
# Visual Basic .NET
```

```

' Private member to signal permitted actions.
Private RadioActions As RadioAction = RadioAction.None

' Property allows all actions to be configured in one call
' using values from the RadioAction enumeration
Public Property Actions( ) As RadioAction
    Get
        Return RadioActions
    End Get
    Set (ByVal Value As RadioAction)
        RadioActions = value
    End Set
End Property

' Property to allow StartStop permission to be switched on
' and off
Public Property StartStop( ) As Boolean
    Get
        Return (RadioActions And RadioAction.StartStop) <> 0
    End Get
    Set (ByVal Value As Boolean)
        If value Then
            RadioActions = RadioActions Or RadioAction.StartStop
        Else
            RadioActions = RadioActions And _
                (Not RadioAction.StartStop)
        End If
    End Set
End Property

```

```

        End Set
    End Property

    ' Property to allow Channel permission to be switched on
    ' and off
    Public Property Channel( ) As Boolean
        Get
            Return (RadioActions And RadioAction.Channel) <> 0
        End Get
        Set (ByVal Value As Boolean)
            If value Then
                RadioActions = RadioActions Or RadioAction.Channel
            Else
                RadioActions = RadioActions And _
                    (Not RadioAction.Channel)
            End If
        End Set
    End Property

    ' Property to allow Volume permission to be switched on
    ' and off
    Public Property Volume( ) As Boolean
        Get
            Return (RadioActions And RadioAction.Volume) <> 0
        End Get
        Set (ByVal Value As Boolean)
            If value Then
                RadioActions = RadioActions Or RadioAction.Volume
            Else
                RadioActions = RadioActions And _
                    (Not RadioAction.Volume)
            End IF
        End Set
    End Property

```

Despite its simplicity, the `CreatePermission` method is the most important method of a permission attribute. It allows the runtime to create permission objects from the declarative security statements contained in your code. The `CreatePermission` method returns a newly created `RadioPermission` using the value of attributes `RadioActions` member:

```

# C#

    // Creates and returns an RadioPermission object
    // based on the configured radio actions.
    public override IPermission CreatePermission( ) {

        return new RadioPermission(RadioActions);
    }
}

# Visual Basic .NET

```

```

        ' Creates and returns an RadioPermission object
        ' based on the configured radio actions.
        Public Overrides Function CreatePermission( ) As IPermission
            Return New RadioPermission(RadioActions)
        End Function
    End Class
End Namespace

```

7.3.1.9 Building the Radio.dll library

The `RadioAction`, `RadioPermission`, and `RadioPermissionAttribute` classes are all contained in a single source file named `Radio.cs` (C#) or `Radio.vb` (Visual Basic .NET). Before you can build the `Radio.dll` library containing the new security classes, you must create a key file named `Keys.snk` to satisfy the reference contained in the `AssemblyKeyFile` attribute. Create the `Keys.snk` file and compile the `Radio.dll` library using the following commands:

```

# C#

sn -k Keys.snk
csc /target:library Radio.cs

# Visual Basic .NET

sn -k Keys.snk
vbc /target:library Radio.vb

```

7.3.1.10 Using RadioPermission to enforce security

`RadioPermission` is now ready to protect the methods in the managed library that provides access to the radio tuner. As demonstrated in the following code fragments, you can use `RadioPermission` in exactly the same ways as the standard permission classes:

```

# C#

// An imperative assert of RadioPermission granting Volume control
RadioPermission r = new RadioPermission(RadioAction.Volume);
r.Assert( );

// A declarative demand for permission to change the radio channel
[RadioPermission(SecurityAction.Demand, Channel = true)]

// A minimum permission request for unrestricted access to the radio
[assembly:RadioPermission(SecurityAction.RequestMinimum,
Unrestricted = true)]

# Visual Basic .NET

' An imperative assert of RadioPermission granting Volume control
Dim r As RadioPermission = New RadioPermission(RadioAction.Volume)
r.Assert( )

```

```
' A declarative demand for permission to change the radio channel  
<RadioPermission(SecurityAction.Demand, Channel := True)> _
```

```
' A minimum permission request for unrestricted access to the radio  
<assembly:RadioPermission(SecurityAction.RequestMinimum, _  
Unrestricted := True)>
```

Once you have developed the secure class library that enables managed code to control the PC radio tuner, you must configure the security policy to grant access to the appropriate code. First, you must understand how security policy calculates the set of permissions to grant to an assembly. Then you need to understand how to use the security administration tools supplied with the .NET Framework to configure security policy. We discuss security policy in [Chapter 8](#) and the .NET administration tools in [Chapter 9](#).

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 8. Security Policy

This chapter explains how the runtime uses *security policy* to determine which permissions to grant an assembly or application domain based on its identity. We begin with a high-level explanation of security policy and clarify its relationship to evidence and permissions. We describe the structure of security policy and explain how the component elements interact at runtime. We explain how to manipulate security policy programmatically and demonstrate the use of application domain policy. Finally, we continue the development of the CAS extensions started in [Chapter 6](#), showing you how to use custom evidence in security policy configuration.

[\[Team LiB \]](#)

[[Team LiB](#)]

8.1 Security Policy Explained

Security policy is the set of configurable rules that provide a mapping between evidence and permissions. Specifically, the runtime uses security policy to determine which *code-access* permissions to grant an assembly or application domain based on the set of evidence that the assembly or application domain presents—a process known as *policy resolution*.



Security policy only determines the code-access permissions assigned to an assembly or application domain. The runtime assigns *identity* permissions as a direct result of the assembly or application domain presenting certain types of evidence. *Role-based* permissions are based on the identity of the user under which the application is executing. See [Chapter 7](#) for a complete description of the different permission types.

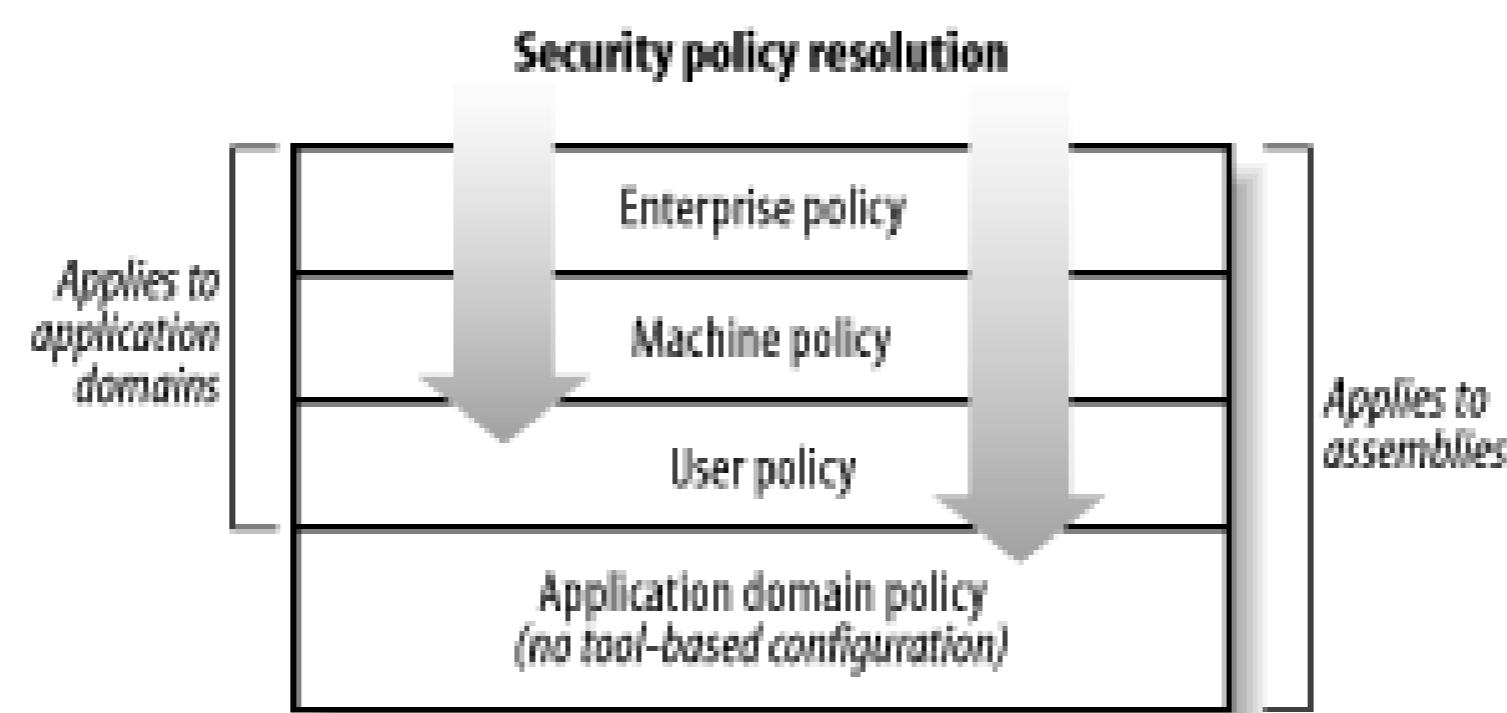
The security policy mechanism is flexible and extensible. It gives administrators and users fine-grained control over the operations and resources to which code has access. With a properly configured security policy, users can confidently run managed code from any source, knowing that the runtime will stop the code from performing undesired actions.

The flexibility provided by the security policy mechanism is essential; no single set of security restrictions can meet everyone's requirements. One user may never want to allow code from the Internet to write to the hard disk, another may only want applications written by Microsoft to write to the Windows registry, yet another may want an environment where all applications can do everything. The security policy mechanism is flexible enough to cater to all of these situations.

8.1.1 Security Policy Levels

As shown in [Figure 8-1](#), .NET divides security policy into four levels: enterprise, machine, user, and application domain. The enterprise, machine, and user levels are configurable independently using administrative tools provided with the .NET Framework, which we discuss in [Chapter 9](#). Because application domains exist only at runtime, it is not possible to configure them statically using tools; you must configure application domain policy programmatically, which we explain in [Section 8.2](#) of this chapter.

Figure 8-1. Security policy levels



When the runtime loads an assembly or creates an application domain, it determines the permissions granted by the enterprise policy first, followed by the machine policy, and finally the user policy. In the case of an assembly, the runtime will also determine the permissions granted by the application domain to which the assembly is loaded. The runtime intersects the sets of permissions granted by each policy level to determine the final code-access permission set for the assembly or application domain. This means that each policy level can further restrict the permissions granted by previous levels but can never grant additional permissions. We provide a more complete description of this in [Section 8.2.1.2](#).

The purpose of enterprise policy is to provide a corporate-wide security policy that is enforced on all machines, but both the enterprise and machine policies are machine-specific. Although we will discuss mechanisms for distributing enterprise security policy in [Chapter 9](#), there are no mechanisms built into the .NET Framework to manage enterprise policy from a central location. User policy is user-specific; on a machine used by different people, each user will have his own user policy configuration.

The division of security policy into different levels complicates administration, but the flexibility it provides far outweighs the complexity, especially in corporate environments. With multiple levels of security policy, it is possible to delegate the task of security management. For example, the central IT or security department can use enterprise policy to enforce the minimum security standards of the organization, while giving departments, teams, or even individuals the autonomy to further restrict permissions to suit local needs with machine and user policy.

Each policy level contains three key elements:

- Code groups
- Named permission sets
- Fully trusted assemblies

We discuss each of these elements in detail in the following sections.

8.1.1.1 Code groups

Code groups are the basic building blocks of security policy; they embody the logic that controls the policy resolution process. More specifically, code groups provide the mapping between evidence and permissions that the policy resolution process uses to determine which code-access permissions to grant an assembly or application domain.

Each policy level consists of a set of code groups organized into a tree structure as shown in [Figure 8-](#)

2. During policy resolution, the runtime traverses the tree of code groups in each policy level and compares the evidence presented by the assembly or application domain with the *membership condition* of each code group. If the evidence meets the code group's membership condition, then the runtime grants the assembly or application domain the permissions contained in the code group's *permission set*.

Figure 8-2. Code group hierarchies

We provide a detailed explanation of how the runtime uses code groups to resolve policy in [Section 8.1.2](#), but first we explain a little more about the structure of code groups. Each code group has a name and a description and contains the following elements:

Membership condition

As shown in [Figure 8-3](#), the membership condition defines the evidence that an assembly or application domain must have to qualify for membership to a code group. Membership conditions support all of the standard evidence types and can be extended to support custom evidence types such as the `Author` class developed in [Chapter 6](#). Possible membership conditions include tests like `Site = "*.oreilly.com,"` `Zone = Internet,` or `Author = "Peter."` A membership condition can also specify "all code," in which case any assembly or application domain is a member of the code group regardless of the evidence it presents.

Figure 8-3. Membership conditions

Permission set

The permission set is the set of permissions to grant an assembly or application domain that qualifies for membership of the code group. When configuring security policy using the .NET administrative tools, this is a *named permission set*, which we discuss in the next section. If you are manipulating the code group programmatically, you can use any permission set, which we will demonstrate in [Section 8.2.1](#).

Child code groups

As shown in [Figure 8-2](#), each policy level consists of a single tree of code groups with each code group having one or more child code groups. An assembly or application domain that is a member of a code group is tested for membership of the code group's children and so on down through the tree. We discuss the relationship between parent and child code groups when determining grant sets in "Policy Resolution." The use of code group trees and simple membership conditions is a flexible model that allows you to create complex security policies that remain easy to manage and understand.

Attributes

There are two optional attributes that you can assign to a code group in order to modify the normal policy resolution process:

Exclusive

The code group's permission set defines the maximum set of permissions that the assembly or application domain can get from the current policy level regardless of what other code groups it is a member of. Because the final permission set of an assembly or application domain is the union of the sets of permissions granted by each policy level, an **Exclusive** code group effectively limits the overall permissions available from the policy resolution.

LevelFinal

The runtime will not evaluate any policy level below the current level other than the application domain level (when resolving assemblies).

We explain the effects of these attributes on policy resolution more in [Section 8.1.2.1](#) of "Policy Resolution."

8.1.1.2 Named permission sets

As the name suggests, named permission sets are simply groups of permissions to which you assign a name. Each policy level maintains its own set of named permission sets that are valid only within the scope of that policy level.

When using the .NET administrative tools (discussed in [Chapter 9](#)) to configure the permission set granted by a code group, you must use a named permission set from the policy level to which the code group belongs. Named permission sets simplify the administration and auditing of security policy, and although this approach may initially seem restrictive, when you look at the interaction of code groups in [Section 8.1.2](#), you will see that the model maintains a high degree of flexibility.

The default security policy defines a standard set of named permission sets for the enterprise, machine, and user policy levels, which we list in [Table 8-1](#). All permission sets except the **Everything**

set are immutable.

Table 8-1. Standard named permission sets

Permission set	Description
<code>FullTrust</code>	Unrestricted access to all operations and resources.
<code>SkipVerification</code>	No access to any operation or resource other than permission to skip verification, which we discuss in Chapter 4 .
<code>Execution</code>	No access to any operation or resource other than permission to execute.
<code>Nothing</code>	No access to any operation or resource, not even the permission to execute.
<code>LocalIntranet</code>	A set of permissions defined in the default security policy deemed by Microsoft to be appropriate for code loaded from the local intranet. See Chapter 9 for a more detailed description of the permissions included in this permission set.
<code>Internet</code>	A set of permissions defined in the default security policy deemed by Microsoft to be appropriate for code loaded from the Internet. See Chapter 9 for a more detailed description of the permissions included in this permission set.
<code>Everything</code>	The only modifiable standard permissions set. By default, <code>Everything</code> contains all permissions with unrestricted access except the <code>SkipVerification</code> element of the <code>SecurityPermission</code> .

8.1.1.3 Fully trusted assemblies

.NET uses classes to represent all key elements of CAS. Some elements that we have discussed so far include evidence, policy levels, code groups, membership conditions, and permissions. During policy resolution, the runtime creates objects to represent the security information it is working with; this can involve the instantiation of classes from many different assemblies.

Under normal circumstances, as the runtime loads these assemblies, it would need to resolve the policy for each one to determine their permissions. However, if the policy resolution of these assemblies required the runtime to instantiate security classes contained within the same assemblies the runtime would need to resolve the policy for the same assembly again, resulting in a never-ending policy resolution loop.

To overcome this problem, each policy level contains a list of *fully trusted assemblies*. When the runtime loads any of these assemblies during policy resolution, it automatically assigns them full trust within that policy level; they are not subject to the normal policy resolution process.

Making an assembly fully trusted in one policy level does not mean that that assembly has full trust during program execution. The final set of permissions granted to the assembly is still calculated by intersecting the permission sets granted by each policy level.

The default security policy defines a standard set of fully trusted assemblies for the enterprise,

machine, and user policy levels. These lists include all of the standard .NET assemblies that contain classes required in the policy resolution process. You do not need to make any changes to the default lists unless you install extensions to CAS containing custom security classes. See [Chapter 9](#) for a description of the default security policy.

To use a CAS extension, you must add the assembly containing your custom security classes and all the assemblies it references to the fully trusted assemblies list of the policy level in which you will use your extensions. If you create a custom evidence class, as we did in [Chapter 6](#), you must add the assembly to the fully trusted list of every policy level, because every policy level will use it. Before you can add an assembly to the fully trusted list of a policy level, you must install the assembly in the *global assembly cache*, this requires the assembly to have a strong name. We discuss the global assembly cache in [Chapter 2](#), and describe how to configure the fully trusted assembly lists using both graphical and command-line tools in [Chapter 9](#).

8.1.2 Policy Resolution

The runtime uses only the enterprise, machine, and user policy levels when resolving the grant set for application domains. Other than this, the policy resolution process for assemblies and application domains is the same. To simplify the following explanation, refer only to assemblies.

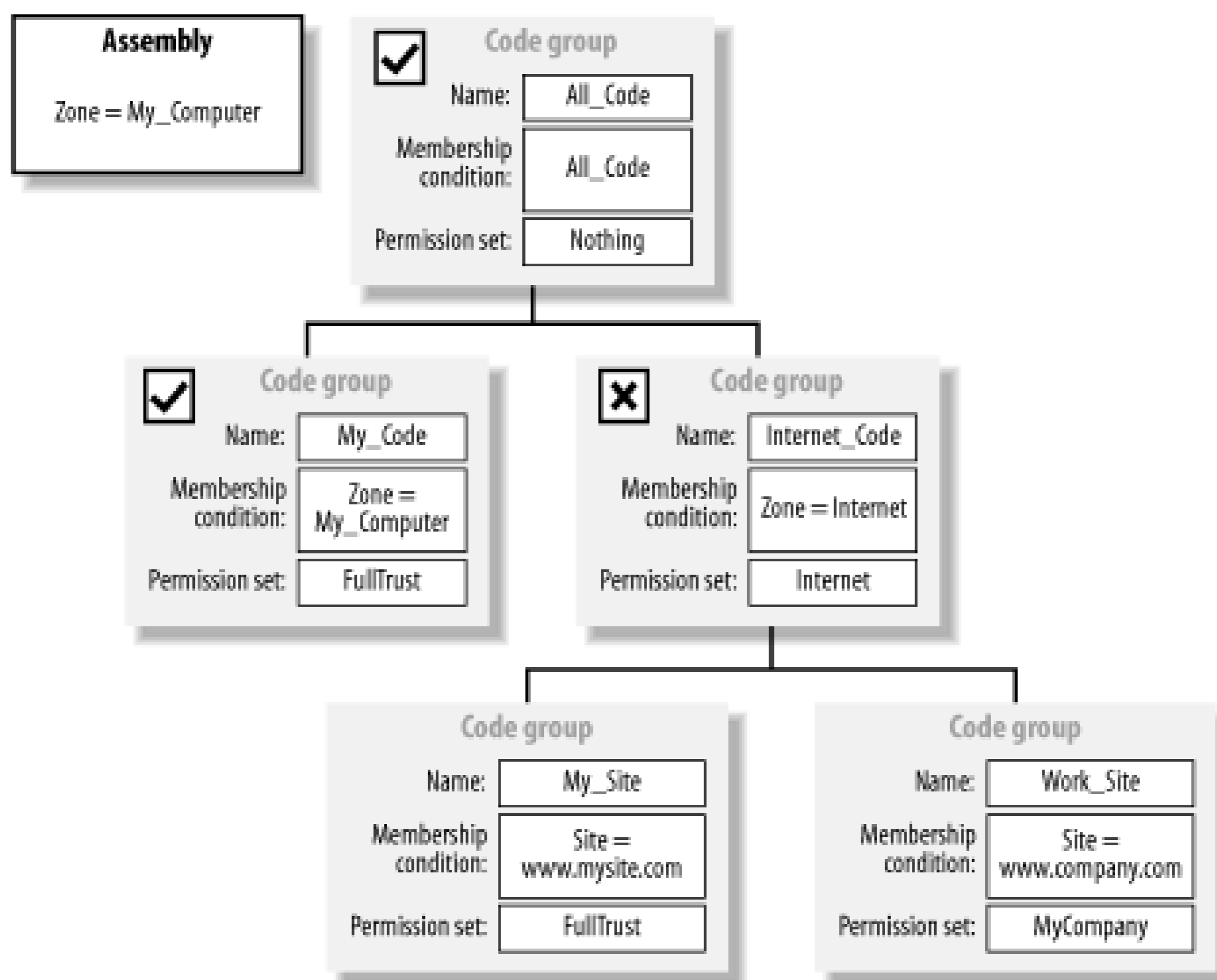
There are different types of code groups that affect the policy resolution process in different ways. The policy resolution we describe here uses the most common type of code group and the only one with direct support in the security administration tools. We discuss each type of code group and provide details of how they affect policy resolution in [Section 8.2.1](#).

When resolving policy for an assembly, the runtime starts at the enterprise policy's root code group and checks the assembly's evidence against the code group's membership condition. If the evidence meets the membership condition, the runtime grants the permissions specified in the code group's permission set to the assembly.

The runtime then traverses the code group tree by comparing the assembly's evidence with each child code group of the current code group. At any stage, if the assembly does not qualify for membership of a code group, then the runtime does not grant the code group's permission set to the assembly, and policy resolution moves on to the next peer-level code group, ignoring the current code group's children.

For example, [Figure 8-4](#) shows a code group hierarchy and an assembly with the evidence `Zone = My_Computer`. Policy resolution begins by comparing the assembly's evidence collection to the membership condition of the root code group `All_Code`. The membership condition of `All_Code` is `All_Code`, and the assembly qualifies for membership, but receives no permissions because the permission set of `All_Code` is `Nothing`.

Figure 8-4. Policy resolution

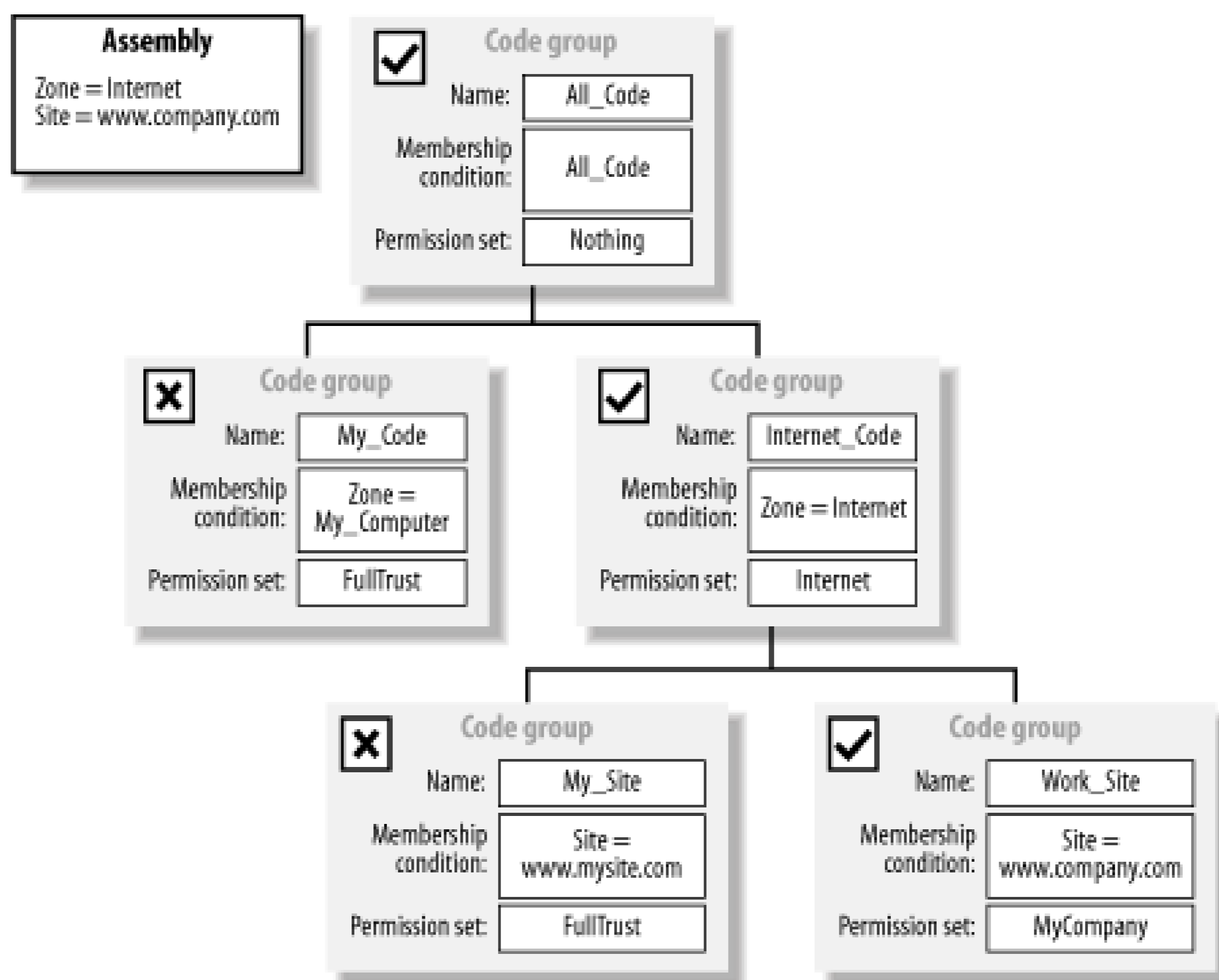


Because the assembly is a member of the `All_Code` group, the runtime compares the assembly for membership of each child code group: `My_Code` followed by `Internet_Code`. The `Zone` evidence of the assembly matches the membership condition of the `My_Code` group, and the runtime grants the assembly the `FullTrust` permission set. Even though the assembly now has `FullTrust`, policy resolution must continue in case another code group implements the `Exclusive` attribute, which may limit the assembly grant set—see [Section 8.1.2.1](#) for details.

When the runtime compares the assembly to the `Internet_Code` group, the assembly's evidence does not meet the membership condition. The runtime ignores the children of the `Internet_Code` group, and because there are no further peer-level groups, policy resolution is complete. The final grant set for the policy level is the union of `Nothing` (granted by `All_Code`) and `FullTrust` (granted by `My_Code`), which is `FullTrust`.

In [Figure 8-5](#), we show the same code group hierarchy, but this time the policy resolution is for an assembly with `Zone = Internet` and `Site = www.company.com` evidence. Policy resolution begins, as in the previous example, by comparing the assembly's evidence with the membership condition of `All_Code`. Again, the assembly meets the membership condition (`All_Code`), and the runtime compares the assembly for membership with the `My_Code` and `Internet_Code` child groups.

Figure 8-5. Policy resolution



In this case, the assembly does not qualify for membership to the `My_Code` group, but its `Zone = Internet` evidence means it is a member of the `Internet_Code` group, which grants it the `Internet` permission set. When compared to the child `My_Site` and `Work_Site` groups, the assembly's `Site = www.company.com` evidence qualifies it for membership to the `Work_Site` code group, and therefore the runtime grants the assembly the custom `MyCompany` permission set.

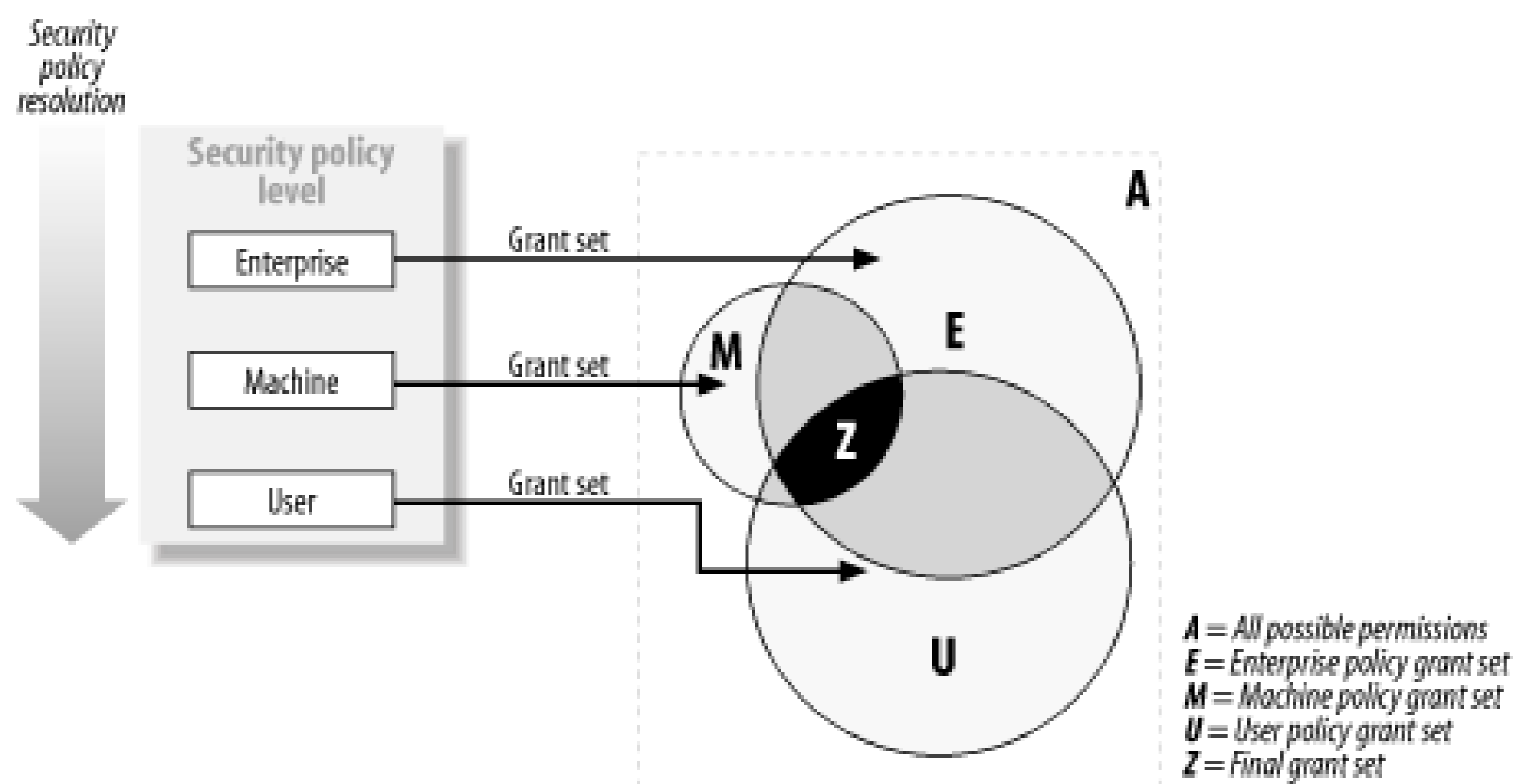
In this example, the assembly is a member of three code groups: `All_Code`, `Internet_Code`, and `Work_Site`. The shaded area in [Figure 8-6](#) shows the final permission set for this policy level calculated as the union of the permissions contained in the `Nothing`, `Internet`, and `MyCompany` permission sets.

Figure 8-6. Calculating policy level permissions

The runtime uses this process to determine the permissions granted by each policy level and then intersects them to calculate the final code-access permission set for the assembly. [Figure 8-7](#)

illustrates the intersection of permission sets from the enterprise, machine, and user policy levels (areas E, M, and U) to create a final grant set (area Z) for the assembly. For clarity, we have omitted the application domain policy.

Figure 8-7. Intersecting policy level grant sets



8.1.2.1 Code group attributes

We have mentioned that the **Exclusive** and **LevelFinal** code group attributes affect the policy resolution process. A code group can have none, one, or both of the attributes, but their effect is significant and you should consider their impact thoroughly before implementing either of them.

Exclusive

The permissions specified in the code group's permission set are the maximum set of permissions that the assembly can receive from the current policy level. For example, in [Figure 8-5](#), if we make the **Internet_Code** code group **Exclusive**, even though the assembly is also a member of the **Work_Site** group, it will receive only the permissions specified in the **Internet** permission set.

As shown in [Figure 8-7](#), to determine the assembly's final grant set, the permissions granted by each policy level intersect to determine the assembly's final grant set, and a code group with the **Exclusive** attribute defines the absolute maximum set of permissions an assembly can be granted.

Exclusive code groups are useful if you want to single out applications with a specific characteristic (the Publisher or source web site, for example) and assign them an absolute set of permissions. This avoids the runtime inadvertently granting them additional permissions based on the other evidence they present.

Be careful when configuring **Exclusive** code groups. If an assembly resolves to be a member of more than one **Exclusive** code group, the runtime will throw a **System.Security.Policy.PolicyException**, and the assembly will not be loaded.

LevelFinal

If an assembly or application domain is a member of a `LevelFinal` code group, the runtime will not evaluate any policy levels below the current level. If resolving the permissions for an assembly, the runtime still evaluates the application domain policy regardless of whether the assembly is the member of a `LevelFinal` code group. For example, if the runtime resolves an assembly to be a member of a `LevelFinal` code group in the machine policy level, the runtime will skip the user policy level, but will still evaluate the application domain policy.

In environments that distribute security policy management between multiple people or groups, `LevelFinal` gives security administrators the ability to disable lower levels of security policy. This enables security administrators to stop people from using security policy to break applications. For example, if every machine in your organization needed to run a particular virus checker, you could create an enterprise-level `LevelFinal` code group based on `Publisher` or `StrongName` evidence that grants the virus checker the permission it requires. Nobody could use machine or user policy to restrict security policy further in order to stop the virus checker from running. By also making the code group `Exclusive`, you could ensure that the enterprise policy would not grant the virus checker more permissions than it needed.

8.1.3 Configuring Security Policy

Security policy is stored in XML files; there is one file for each of the enterprise, machine, and user policy levels. We discuss the location of these files in [Chapter 9](#). There are four ways to configure security policy:

- Using the .NET Framework Configuration tool (*Mscorcfg.msc*), a Microsoft Management Console (MMC) plug-in provided with the .NET Framework that provides a graphical interface with which to administer security policy
- Using the Code Access Security Policy tool (*Caspol.exe*), a command-line tool provided with the .NET Framework
- Programmatically, using the security classes contained in the .NET class library
- Manually, by editing the XML contained in the individual security policy files

We suggest that you do not try to edit your security policy files manually. The structure of policy files is poorly documented, and they can easily become long and confusing. A simple typing error can change your security configuration to the point where no code at all can run. Even worse, you may introduce a subtle and difficult-to-spot security hole, which grants code more permissions than you intended.

The only way to configure application domain policy is at runtime using the .NET security classes. The techniques for the programmatic manipulation of security policy are the same for all policy levels, and we discuss these in the next section. We explain how to use the .NET Framework Configuration tool and the Code Access Security Policy tool in [Chapter 9](#).

[\[Team LiB \]](#)

[\[Team LiB \]](#)

8.2 Programming Security Policy

The .NET security administration tools (discussed in [Chapter 9](#)) are sufficient for most users and security administrators to configure their security policy, but to have complete control over all security policy features, you must do so programmatically. As with evidence and permissions, .NET represents each of the security policy elements with classes, meaning that you can manipulate them in your own code very easily.

In the following sections, we provide explanations of how to program the key components of security policy, starting with code groups and membership conditions, then moving on to policy levels, and finally the security manager. We do not discuss individual permissions or permission sets, because we covered them in [Chapter 7](#). We conclude with an example that brings together all elements of security policy programming; we demonstrate how to manipulate the policy of an application domain to control the policy resolution process applied to the assemblies loaded into it.

8.2.1 Programming Code Groups

The abstract `System.Security.Policy.CodeGroup` class provides the base representation of a code group and defines the functionality that lies at the heart of the policy resolution process. Four noninheritable subclasses extend `CodeGroup` to provide concrete implementations that you can use in security policy programming; these are shown in [Figure 8-8](#).

Figure 8-8. Concrete subclasses of CodeGroup

[Figure 8-9](#) illustrates the structure of the `CodeGroup` class, showing that `CodeGroup` is a container for all of the elements we discussed in [Section 8.1.1.1](#). We summarize how `CodeGroup` implements each of these component elements here and provide additional details on how to use them later in this section.

Membership condition

An object that implements the `System.Security.Policy.IMembershipCondition` interface provides the functionality to determine whether an assembly or application domain qualifies for membership to the code group. You must specify the membership condition at `CodeGroup`

instantiation but can change it later using the `MembershipCondition` property.

Policy statement

The policy statement contains values that specify the effect the `CodeGroup` has on assemblies and application domains that are members of the group. The `System.Security.Policy.PolicyStatement` class represents a code group's policy statement. It contains a value from the `System.Security.Policy.PolicyStatementAttribute` enumeration to represent the code group's attributes, and a `System.Security.PermissionSet` that specifies the set of permissions granted by the code group to its members.

Children

Each `CodeGroup` contains an ordered list of child `CodeGroup` objects. In turn, each child `CodeGroup` can have its own children, creating a tree structure. The `Children` property gets or sets the code group's children using a `System.Collections.IList` containing the set of child `CodeGroup` objects.

Figure 8-9. Structure of the `CodeGroup` class

The most important method of the `CodeGroup` class is `Resolve`, which takes an `Evidence` collection as an argument. Policy resolution of an assembly (or application domain) consists of the runtime calling the `Resolve` method on the root `CodeGroup` in each policy level and passing it the assembly's `Evidence` collection. In the `Resolve` method, the `CodeGroup` is responsible for determining if the assembly's evidence qualifies it for membership, how to apply any attributes, and how or if the `CodeGroup` should use its children to continue the policy resolution process. `Resolve` returns a `PolicyStatement` that represents the net effect of all code groups in the tree to which the assembly qualified for membership. The key difference between each of the `CodeGroup` subclasses is how they process the `Resolve` method:

`UnionCodeGroup`

This is the most commonly used code group type, which implements policy resolution, as we explained in [Section 8.1.2](#). Members are tested against all child code groups for membership. The resulting `PolicyStatement` contains the union of the code group's permission set and the permission sets of each child of which the assembly is also a member. If a code group has the `Exclusive` attribute, the `PolicyStatement` will include only those permissions granted by that group, as we described in [Section 8.1.2.1](#).

`FileCodeGroup`

This operates the same as `UnionCodeGroup` with respect to matching members against its

children and how the resulting `PolicyStatement` is calculated. However, `FileCodeGroup` does not support attributes and does not contain a statically defined permission set. With each call to `Resolve`, if the evidence of the assembly contains "file://" -based `Url` evidence, the `FileCodeGroup` dynamically generates a permission set that contains a `System.Security.Permissions.FileIOPermission` granting access to the directory specified in the `Url` evidence. We discussed `Url` evidence in [Chapter 6](#) and `FileIOPermission` in [Chapter 7](#).

`NetCodeGroup`

This operates the same as `FileCodeGroup`, but with each call to `Resolve`, if the evidence of the assembly contains "http://" or "https://" `Url` (or `Site`) evidence, the `NetCodeGroup` dynamically generates a permission set that contains a `System.Security.Permissions.WebPermission` granting connect access to the specified web site. We discussed `WebPermission` in [Chapter 7](#).

`FirstMatchCodeGroup`

This operates the same `UnionCodeGroup` but evaluates members against its children only until it finds the first matching child group. The resulting `PolicyStatement` contains the permission granted by the `FirstMatchCodeGroup` and at most one of its child groups; this makes the order of the child elements important.

As important as `Resolve` is to the runtime, both `Resolve` and the similar `ResolveMatchingCodeGroups` method provide minimal value when used directly. You cannot use them to drive the policy resolution process manually, and therefore they are useful only for testing and debugging. [Table 8-2](#) summarizes the properties and methods of `CodeGroup` and highlights any special behavior implemented by the subclasses. No specific code-access permissions are required to use the members of the code group classes.

Table 8-2. Members of the code group classes

Member	Description
Properties	
<code>AttributeString</code>	Returns a <code>String</code> representing the code group's attributes. <code>FileCodeGroup</code> and <code>NetCodeGroup</code> do not use attributes and always return <code>null</code> (C#) or <code>Nothing</code> (Visual Basic .NET).
<code>Children</code>	Gets or sets the code group's child code groups. The child <code>CodeGroup</code> elements are contained in an <code>ICollection</code> . For <code>FirstMatchCodeGroup</code> groups, the order of the children in the <code>ICollection</code> is important.
<code>Description</code>	Gets or sets the description of the code group as a <code>String</code> .
<code>MembershipCondition</code>	Gets or sets the membership condition for the code group using a <code>System.Security.Policy.IMembershipCondition</code> instance.

Member	Description
<code>MergeLogic</code>	Returns a <code>String</code> that describes the logic used to merge the permissions of the code group and its children; returns "First Match" for <code>FirstMatchCodeGroup</code> and "Union" for the other three types of code groups.
<code>Name</code>	Gets or sets the name of the code group using a <code>String</code> .
<code>PermissionSetName</code>	Gets a <code>String</code> containing the name of the code group's permission set. This is <code>null</code> (C#) or <code>Nothing</code> (Visual Basic .NET) if the permission set does not have a name. The <code>NetCodeGroup</code> class always returns the name "Same site Web" and the <code>FileCodeGroup</code> always returns the name "Same directory FileIO-" appended with the type of file access granted.
<code>PolicyStatement</code>	Gets or sets the <code>PolicyStatement</code> of the code group.
Methods	
<code>AddChild</code>	Creates a copy of the specified <code>CodeGroup</code> and adds it to the end of the current list of children.
<code>Copy</code>	Creates a deep copy of the <code>CodeGroup</code> , including all child code groups.
<code>Equals</code>	Basic equality is determined by comparing the <code>Name</code> , <code>Description</code> , and <code>MembershipCondition</code> of two <code>CodeGroup</code> objects. It is also possible to force a comparison that includes all child code groups.
<code>FromXml</code>	Reconstructs a <code>CodeGroup</code> from XML previously generated using the <code>ToXml</code> method.
<code>RemoveChild</code>	Removes the specified <code>CodeGroup</code> from the list of immediate children.
<code>Resolve</code>	Returns a <code>PolicyStatement</code> containing the permissions granted by the <code>CodeGroup</code> based on a specified <code>Evidence</code> collection. As discussed earlier, the type of code group determines how the permissions of child code groups affect the final <code>PolicyStatement</code> .
<code>ResolveMatchingCodeGroups</code>	Returns a <code>CodeGroup</code> tree containing all of the child code groups to which a specified <code>Evidence</code> collection qualifies for membership. The type of code group determines the set of child code groups contained in the final collection.
<code>ToXml</code>	Returns a <code>SecurityElement</code> containing the XML object model for the <code>CodeGroup</code> . <code>ToXml</code> is useful for writing the contents of a code group to an XML file so that you can import it when using the administrative tools. See Chapter 9 for further details.

Before demonstrating the creation and manipulation of code groups, we provide details of membership conditions and policy statements.

8.2.1.1 Programming membership conditions

Membership conditions are classes that implement the `IMembershipCondition` interface. All four types of code group contain a single `IMembershipCondition` instance that you must specify as an argument to the code group's constructor; you can get and set the `IMembershipCondition` through the `CodeGroup.MembershipCondition` property after construction.

The `IMembershipCondition` interface defines a method named `Check`, which takes an `Evidence` collection argument and returns a Boolean value indicating whether the values of the contained evidence objects satisfy a configurable condition:

```
# C#
```

```
bool Check(Evidence evidence);
```

```
# Visual Basic .NET
```

```
Function Check(ByVal evidence As Evidence) As Boolean
```

A `CodeGroup` calls the `Check` method of its `IMembershipCondition` during policy resolution to evaluate whether an assembly or application domain qualifies for membership to the code group.

The .NET Framework includes the eight standard membership condition classes listed in [Table 8-3](#); all are members of the `System.Security.Policy` namespace. Seven of these classes contain the logic necessary to test the values of standard evidence classes, which we discussed in [Chapter 6](#). One additional membership condition class, named `AllMembershipCondition` always returns true when `Check` is called, regardless of the evidence provided. This means a code group using `AllMembershipCondition` as its membership condition will contain all assemblies and application domains tested against it.

Table 8-3. Standard membership condition classes

Membership class	Membership condition
<code>AllMembershipCondition</code>	All code irrespective of evidence.
<code>ApplicationDirectoryMembershipCondition</code>	<code>Evidence</code> collection contains both <code>ApplicationDirectory</code> and <code>Url</code> evidence. The <code>Url</code> evidence represents a location that is a child of that represented by the <code>ApplicationDirectory</code> evidence.
<code>HashMembershipCondition</code>	<code>Evidence</code> collection contains a <code>Hash</code> class with the specified hash value.
<code>PublisherMembershipCondition</code>	<code>Evidence</code> collection contains a <code>Publisher</code> class with the specified publisher certificate.
<code>SiteMembershipCondition</code>	<code>Evidence</code> collection contains a <code>Site</code> class with the specified site name.

Membership class	Membership condition
<code>StrongNameMembershipCondition</code>	<code>Evidence</code> collection contains a <code>StrongName</code> class with the specified hash value.
<code>UrlMembershipCondition</code>	<code>Evidence</code> collection contains a <code>Url</code> class with the specified URL location.
<code>ZoneMembershipCondition</code>	<code>Evidence</code> collection contains a <code>Zone</code> class with the specified security zone.

The constructors for each type of membership condition vary depending on the evidence types they evaluate, but all of them are relatively straightforward to use if you understand the standard evidence types. You should consult the .NET Framework SDK documentation for complete details, but here are some examples of how to create membership conditions:

```
# C#
```

```
// Create a membership condition to match all code.
IMembershipCondition m1 = new AllMembershipCondition( );

// Create a membership condition to match all code with
// Internet Zone evidence.
IMembershipCondition m2 =
    new ZoneMembershipCondition(SecurityZone.Internet);

// Create a membership condition to match all code from
// all "oreilly.com" Sites.
IMembershipCondition m3 =
    new SiteMembershipCondition("*.oreilly.com");

// Create a membership condition to match all code with
// the same Publisher certificate as was used to sign
// the SomeFile.exe assembly.
IMembershipCondition m4 =
    new PublisherMembershipCondition(
        X509Certificate.CreateFromSignedFile("SomeFile.exe")
    );
```

```
# Visual Basic .NET
```

```
' Create a membership condition to match all code.
Dim m1 As IMembershipCondition = New AllMembershipCondition( )

' Create a membership condition to match all code with
' Internet Zone evidence.
Dim m2 As IMembershipCondition = _
New ZoneMembershipCondition(SecurityZone.Internet)

' Create a membership condition to match all code from
' all "oreilly.com" Sites.
Dim m3 As IMembershipCondition = _
```

```

New SiteMembershipCondition("*.oreilly.com")

' Create a membership condition to match all code with
' the same Publisher certificate as was used to sign
' the SomeFile.exe assembly.
Dim m4 As IMembershipCondition = _
New PublisherMembershipCondition( _
X509Certificate.CreateFromSignedFile("SomeFile.exe"))

```

8.2.1.2 Programming policy statements

For the `UnionCodeGroup` and `FirstMatchCodeGroup` classes, a `PolicyStatement` class represents the effect the code group has on its members, including the code group's attributes and its permission set. The `FileCodeGroup` and `NetCodeGroup` classes do not require you to set a policy statement, because they generate their permission sets dynamically and do not support attributes.

Provide a `PolicyStatement` as an argument to the `UnionCodeGroup` and `FirstMatchCodeGroup` constructors, and you can get and set the `PolicyStatement` after construction through the `CodeGroup.PolicyStatement` property.

The `PolicyStatement` class provides two constructors. The first takes a `System.Security.PermissionSet` argument specifying the permissions a code group grants to its members. The second constructor takes both a `PermissionSet` and a member of the `System.Security.Policy.PolicyStatementAttribute` enumeration, representing the code group's set of attributes. We list the possible values of `PolicyStatementAttribute` in [Table 8-4](#).

Table 8-4. Members of the `PolicyStatementAttribute` enumeration

Member	Description
<code>All</code>	The code group has both the <code>Exclusive</code> and <code>LevelFinal</code> attributes
<code>Exclusive</code>	The code group has only the <code>Exclusive</code> attribute. See "Code Group Attributes" for details
<code>LevelFinal</code>	The code group has only the <code>LevelFinal</code> attribute. See "Code Group Attributes" for details
<code>Nothing</code>	The code group has no attributes

The properties listed in [Table 8-5](#) provide access to the content of the `PolicyStatement` after instantiation.

Table 8-5. Properties of `PolicyStatement`

Property	Description
<code>Attributes</code>	Gets or sets the attributes contained in the policy statement as a member of the <code>PolicyStatementAttribute</code> enumeration.
<code>AttributeString</code>	Gets a human-readable <code>String</code> representing the attributes contained in the <code>PolicyStatement</code> .
<code>PermissionSet</code>	Gets or sets the permission contained in the <code>PolicyStatement</code> as a <code>PermissionSet</code> .

The following examples demonstrate how to create `PolicyStatement` objects. See [Chapter 7](#) for a description of how to create `PermissionSet` objects:

```
# C#

// Create a PolicyStatement that grants Unrestricted access
// to everything
PolicyStatement p1 = new PolicyStatement(
    new PermissionSet(PermissionState.Unrestricted)
);

// Create a PolicyStatement that grants read access to the
// file "C:\File.txt" and specifies the LevelFinal attribute.
PermissionSet pset = new PermissionSet(
    new FileIOPermission(FileIOPermissionAccess.Read, @"C:\File.txt"));
PolicyStatement p2 = new PolicyStatement(
    pset, PolicyStatementAttribute.LevelFinal);

# Visual Basic .NET

' Create a PolicyStatement that grants Unrestricted access
' to everything
Dim p1 As PolicyStatement = New PolicyStatement( _
    New PermissionSet(PermissionState.Unrestricted))

' Create a PolicyStatement that grants read access to the
' file "C:\File.txt" and specifies the LevelFinal attribute.
Dim pset As PermissionSet = New PermissionSet( _
    New FileIOPermission(FileIOPermissionAccess.Read, "C:\File.txt"))

Dim p2 As PolicyStatement = New PolicyStatement _
    (pset, PolicyStatementAttribute.LevelFinal)
```

8.2.1.3 Creating code groups

With the knowledge of how to create policy statements and membership conditions, creating code groups is straightforward. You create `UnionCodeGroup` and `FirstMatchCodeGroup` objects by providing an `IMembershipCondition` and a `PolicyStatement` as constructor arguments. The following example creates a `UnionCodeGroup` with the `Exclusive` attribute that matches all code downloaded from any web site in the *oreilly.com* domain and grants it unrestricted access to the filesystem:

```
# C#

// Create the permission set and add unrestricted file access.
PermissionSet pset = new PermissionSet(PermissionState.None);
pset.AddPermission(new FileIOPermission(PermissionState.Unrestricted));

// Create the policy statement and set the Exclusive attribute.
PolicyStatement pstate =
    new PolicyStatement(pset, PolicyStatementAttribute.Exclusive);

// Create the membership condition to match all "*.oreilly.com" sites.
IMembershipCondition mcon =
    new SiteMembershipCondition("*.oreilly.com");

// Create the UnionCodeGroup
UnionCodeGroup cg = new UnionCodeGroup(mcon, pstate);
```

```
# Visual Basic .NET
```

```
' Create the permission set and add unrestricted file access.
Dim pset As PermissionSet = New PermissionSet(PermissionState.None)
pset.AddPermission(New FileIOPermission(PermissionState.Unrestricted))

' Create the policy statement and set the Exclusive attribute.
Dim pstate As PolicyStatement = _
    New PolicyStatement(pset, PolicyStatementAttribute.Exclusive)

' Create the membership condition to match all "*.oreilly.com" sites.
Dim mcon As IMembershipCondition = _
    New SiteMembershipCondition("*.oreilly.com")

' Create the UnionCodeGroup
Dim cg As UnionCodeGroup = New UnionCodeGroup(mcon, pstate)
```

The `NetCodeGroup` class constructor takes only an `IMembershipCondition`, because it calculates dynamically its permission set and does not support attributes; therefore, there is no need for a `PolicyStatement`. The following example creates a `NetCodeGroup` that allows all code signed with the publisher certificate contained in the `Publisher.cer` file to connect back to the web site from where it is loaded. Details of the `WebPermission` class are in [Chapter 7](#):

```
# C#
```

```
NetCodeGroup cg = new NetCodeGroup(
    new PublisherMembershipCondition(
        X509Certificate.CreateFromCerFile("Publisher.cer")
    ));
```

```
# Visual Basic .NET
```

```
Dim cg As NetCodeGroup = New NetCodeGroup( _
    New PublisherMembershipCondition( _
        X509Certificate.CreateFromCerFile("Publisher.cer"))) )
```


The `FileCodeGroup` constructor takes an `IMembershipCondition` and a `System.Security.Permissions.FileIOPermissionAccess` argument. The `FileIOPermissionAccess` specifies the type of file access permission the `FileCodeGroup` should include in its dynamically generated permission set; see [Table 8-6](#) for possible values. Details of the `FileIOPermission` class are in [Chapter 7](#).

Table 8-6. Members of the `FileIOPermissionAccess` enumeration

Property	Description
<code>AllAccess</code>	All access
<code>Append</code>	Access to append content to an existing file or directory
<code>NoAccess</code>	No access
<code>PathDiscovery</code>	Access to the information contained in the path of a directory or file
<code>Read</code>	Access to read from a file or directory
<code>Write</code>	Access to write to a file or directory, including overwriting and deleting existing files and directories

The following code creates a `FileCodeGroup` that allows all code signed with the publisher certificate contained in the `Publisher.cer` file to write to the directory from which it was loaded:

```
# C#

FileCodeGroup cg = new FileCodeGroup(
    new PublisherMembershipCondition(
        X509Certificate.CreateFromCertFile("Publisher.cer")),
    FileIOPermissionAccess.Write
);

# Visual Basic .NET

Dim cg As FileCodeGroup = New FileCodeGroup( _
New PublisherMembershipCondition( _
X509Certificate.CreateFromCertFile("Publisher.cer")),
FileIOPermissionAccess.Write)
```

8.2.2 Programming Policy Levels

The .NET class library contains the `System.Security.Policy.PolicyLevel` class to represent all security policy levels: enterprise, machine, user, and application domain. The `PolicyLevel` class is a container for the component elements we described in "Security Policy Levels": fully trusted assemblies, named permission sets, and code groups.

You cannot create new `PolicyLevel` objects using constructors. To manipulate the enterprise, machine, or user policy levels, you must obtain a reference to the desired level using the static

methods of the `System.Security.SecurityManager` class, which we discuss in the next section. To create a new application domain policy level, use the static `PolicyLevel.CreateAppDomainLevel` factory method, which we demonstrate in [Section 8.2.4](#).

[Table 8-7](#) summarizes the members of the `PolicyLevel` class. None of the members is protected with code-access permissions. .NET protects the methods you use to obtain and update policy levels, not the methods that manipulate them once you have a reference.

Table 8-7. Members of the `PolicyLevel` class

Member	Description
Properties	
<code>FullTrustAssemblies</code>	Gets an <code>ICollection</code> of <code>StrongNameMembershipCondition</code> objects representing the fully trusted assemblies of the policy level.
<code>Label</code>	Gets a <code>String</code> containing a description of the <code>PolicyLevel</code> .
<code>NamedPermissionSets</code>	Gets an <code>ICollection</code> of <code>NamedPermissionSet</code> objects representing the fully trusted assemblies of the policy level.
<code>RootCodeGroup</code>	Gets or sets the root <code>CodeGroup</code> of the policy levels code group tree.
<code>StoreLocation</code>	Gets a <code>String</code> containing the path of the file in which the <code>PolicyLevel</code> is stored. Returns <code>null</code> (C#) or <code>Nothing</code> (Visual Basic .NET) if the <code>PolicyLevel</code> does not have a storage location.
Methods	
<code>AddFullTrustAssembly</code>	Adds a <code>StrongNameMembershipCondition</code> to the fully trusted assembly list.
<code>AddNamedPermissionSet</code>	Adds a <code>NamedPermissionSet</code> to the policy level's set of named permission sets.
<code>ChangeNamedPermissionSet</code>	Replaces the <code>PermissionSet</code> of the specified <code>NamedPermissionSet</code> .
<code>CreateAppDomainLevel</code>	Returns a <code>PolicyLevel</code> configured for use as an application domain policy level.
<code>FromXml</code>	Reconstructs a <code>PolicyLevel</code> from correctly formatted XML, which is normally generated using the <code>ToXml</code> method.
<code>GetNamedPermissionSet</code>	Returns a <code>NamedPermissionSet</code> with the specified name.
<code>Recover</code>	Reverts the file where the policy level is stored to the previously stored version.
<code>RemoveFullTrustAssembly</code>	Removes the specified fully trusted assembly.
<code>RemoveNamedPermissionSet</code>	Removes the specified named permission set.

Member	Description
<code>Reset</code>	Returns the policy level to its default state. See Chapter 9 for details on the default state of the various policy levels.
<code>Resolve</code>	Returns a <code>PolicyStatement</code> containing the permissions granted by the <code>PolicyLevel</code> based on a specified <code>Evidence</code> collection. This is the same as calling <code>Resolve</code> on the <code>CodeGroup</code> contained in the policy levels <code>RootCodeGroup</code> property.
<code>ResolveMatchingCodeGroups</code>	Returns a <code>CodeGroup</code> tree containing all of the child code groups to which a specified <code>Evidence</code> collection qualifies for membership. This is the same as calling <code>ResolveMatchingCodeGroups</code> on the <code>CodeGroup</code> contained in the policy levels <code>RootCodeGroup</code> property.
<code>ToXml</code>	Returns a <code>SecurityElement</code> containing an XML object model of the <code>PolicyLevel</code> and its contents.

8.2.2.1 Managing fully trusted assemblies

The `PolicyLevel` class represents fully trusted assemblies by maintaining a list of `StrongNameMembershipCondition` objects configured to match the strong names of the trusted assembly. You can manage the fully trusted assembly list by providing `StrongName` or `StrongNameMembershipCondition` objects to the `AddFullTrustAssembly` and `RemoveFullTrustAssembly` methods. The read-only `FullTrustAssemblies` property gets a `System.Collections.IList` containing the list of fully trusted assemblies.

[Example 8-1](#) creates a `StrongNameMembershipCondition` object to add an entry to the fully trusted assembly list for the `HelloWorld` assembly.

Example 8-1. Managing a fully trusted assembly

```
# C#

// Create a byte array containing the strong name public key
// data.
byte[] publickey = { 0, 36, 0, 0, 4, 128, 0, 0, 148,
    0, 0, 0, 6, 2, 0, 0, 0, 36, 0, 0, 82, 83, 65, 49, 0, 4, 0,
    0, 1, 0, 1, 0, 169, 206, 164, 8, 66, 197, 231, 138, 148, 74,
    99, 125, 171, 203, 120, 143, 240, 155, 104, 138, 4, 123, 15,
    55, 85, 255, 183, 20, 111, 10, 217, 58, 127, 15, 236, 86, 16,
    121, 222, 35, 161, 14, 122, 246, 85, 226, 162, 221, 46, 215,
    161, 151, 183, 38, 31, 150, 198, 119, 109, 94, 11, 65, 208,
    33, 122, 172, 106, 62, 192, 4, 35, 255, 220, 10, 43, 90, 92,
    183, 29, 136, 57, 235, 30, 5, 127, 72, 210, 108, 215, 226, 65,
    197, 184, 28, 129, 184, 191, 211, 159, 69, 8, 84, 116, 65, 186,
    179, 35, 116, 174, 223, 167, 217, 116, 8, 178, 232, 213, 155,
    172, 87, 181, 187, 61, 43, 133, 105, 10, 187 };

// Create a StrongNamePublicKeyBlob object from the
// public key byte array.
StrongNamePublicKeyBlob blob = new StrongNamePublicKeyBlob(publickey);
```

```

// Create a Version object based on the assembly version
// number
Version version = new Version("1.1578.0.0");

// Create the new StrongNameMembershipCondition
StrongNameMembershipCondition mc =
    new StrongNameMembershipCondition (blob, "HelloWorld", version);

// Create a new application domain policy level
PolicyLevel p = PolicyLevel.CreateAppDomainLevel( );

// Add the StrongNameMembershipCondition to the fully trusted
// assembly list
p.AddFullTrustAssembly(mc);

# Visual Basic .NET

' Create a byte array containing the strong name public key
' data.
Dim publickey( ) As Byte = { 0, 36, 0, 0, 4, 128, 0, 0, 148, _
    0, 0, 0, 6, 2, 0, 0, 0, 36, 0, 0, 82, 83, 65, 49, 0, 4, 0, _
    0, 1, 0, 1, 0, 169, 206, 164, 8, 66, 197, 231, 138, 148, 74, _
    99, 125, 171, 203, 120, 143, 240, 155, 104, 138, 4, 123, 15, _
    55, 85, 255, 183, 20, 111, 10, 217, 58, 127, 15, 236, 86, 16, _
    121, 222, 35, 161, 14, 122, 246, 85, 226, 162, 221, 46, 215, _
    161, 151, 183, 38, 31, 150, 198, 119, 109, 94, 11, 65, 208, _
    33, 122, 172, 106, 62, 192, 4, 35, 255, 220, 10, 43, 90, 92, _
    183, 29, 136, 57, 235, 30, 5, 127, 72, 210, 108, 215, 226, 65, _
    197, 184, 28, 129, 184, 191, 211, 159, 69, 8, 84, 116, 65, 186, _
    179, 35, 116, 174, 223, 167, 217, 116, 8, 178, 232, 213, 155, _
    172, 87, 181, 187, 61, 43, 133, 105, 10, 187}

' Create a StrongNamePublicKeyBlob object from the
' publickey byte array.
Dim blob As StrongNamePublicKeyBlob = New StrongNamePublicKeyBlob(publickey)

' Create a Version object based on the assembly version
' number
Dim version As Version = New Version("1.1578.0.0")

' Create the new StrongNameMembershipCondition
Dim mc As StrongNameMembershipCondition = _
New StrongNameMembershipCondition(blob,"HelloWorld",version)

' Create a new application domain policy level
Dim p As PolicyLevel = PolicyLevel.CreateAppDomainLevel( )

' Add the StrongNameMembershipCondition to the fully trusted
' assembly list
p.AddFullTrustAssembly(mc)

```

8.2.2.2 Managing named permission sets

As described in [Section 8.1.1](#), each policy level contains a default set of named permissions sets. These default sets cannot be deleted, and, apart from the `Everything` group, you cannot edit them.

To manage a policy level's named permission sets use the `AddNamedPermissionSet` and `RemoveNamedPermissionSet` methods. `AddNamedPermissionSet` takes a `NamedPermissionSet` argument, whereas `RemoveNamedPermissionSet` can take either a `NamedPermissionSet` or a `String` containing the name of the `NamedPermissionSet` to remove. You can also change the permission set of an existing `NamedPermissionSet` without having to add and remove it by calling the `ChangeNamedPermissionSet` method and passing it the name of the `NamedPermissionSet` to change, and a `PermissionSet` containing the new set of permissions.

The `GetNamedPermissionSet` method returns a `NamedPermissionSet` with the specified name, and the `NamedPermissionSets` property gets an `IList` containing the set of `NamedPermissionSet` objects. [Example 8-2](#) demonstrates the manipulation of named permission sets.

Example 8-2. Manipulating named permission sets

```
# C#
// Create a new application domain policy level
PolicyLevel p = PolicyLevel.CreateAppDomainLevel( );

// Get a copy of the default permission set named "Internet" and
// call it "NewPermissionSet"
NamedPermissionSet ps =
    p.GetNamedPermissionSet("Internet").Copy("NewPermissionSet");

// Add the new permission set
p.AddNamedPermissionSet(ps);

// Modify the permission set "NewPermissionSet" to grant unrestricted
// access
p.ChangeNamedPermissionSet("NewPermissionSet",
    new PermissionSet(PermissionState.Unrestricted));

// Remove the NewPermissionSet permission set
p.RemoveNamedPermissionSet("NewPermissionSet");

# Visual Basic .NET

' Create a new application domain policy level
Dim p As PolicyLevel = PolicyLevel.CreateAppDomainLevel( )

' Get a copy of the default permission set named "Internet" and
' call it "NewPermissionSet"
Dim ps As NamedPermissionSet = _
    p.GetNamedPermissionSet("Internet").Copy("NewPermissionSet")

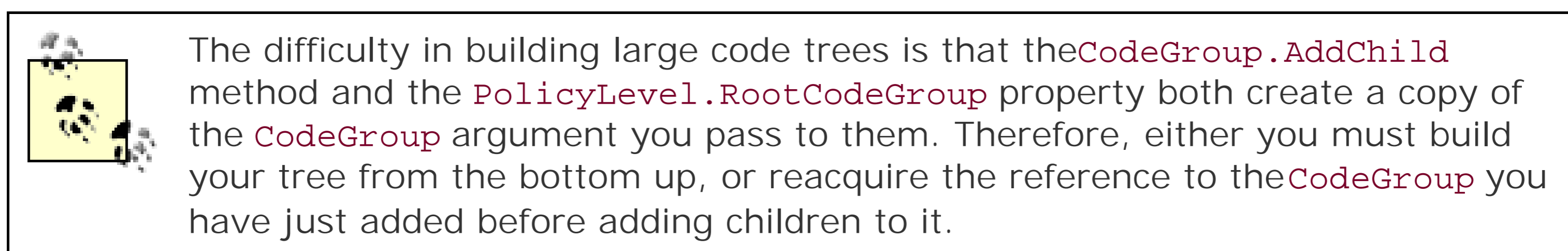
' Add the new permission set
p.AddNamedPermissionSet(ps)
```

```
' Modify the permission set "NewPermissionSet" to grant unrestricted
' access
p.ChangeNamedPermissionSet("NewPermissionSet", _
New PermissionSet(PermissionState.Unrestricted))

Console.WriteLine(p.ToXml( ))
' Remove the NewPermissionSet permission set
p.RemoveNamedPermissionSet("NewPermissionSet")
```

8.2.2.3 Managing the code group tree

You get and set the root code group of the policy level's code group tree using the `RootCodeGroup` property (see [Example 8-3](#)). You must then use the methods and properties of `CodeGroup`, which we discussed in [Section 8.2.1](#), to build and configure the tree hierarchy. The following code creates the policy level we used in [Section 8.1.2](#) to demonstrate the policy resolution process (see [Figure 8-4](#)).



Example 8-3. Manipulating the code group tree of a policy level

```
# C#

// Create a new application domain policy level.
PolicyLevel p = PolicyLevel.CreateAppDomainLevel( );

// Create the MyCompany named permission set as a copy of
// the default LocalIntranet named permission set
p.AddNamedPermissionSet(
    p.GetNamedPermissionSet("LocalIntranet").Copy("MyCompany")
);

// Create the My_Site code group that matches all code
// run from the "www.mysite.com" Site and grants it FullTrust.
UnionCodeGroup MySite = new UnionCodeGroup(
    new SiteMembershipCondition("www.mysite.com"),
    new PolicyStatement(p.GetNamedPermissionSet("FullTrust"))
);
MySite.Name = "My_Site";

// Create the Work_Site code group that matches all code
// run from the "www.company.com" Site and grants it the
// MyCompany, permission set.
UnionCodeGroup WorkSite = new UnionCodeGroup(
    new SiteMembershipCondition("www.company.com"),
    new PolicyStatement(p.GetNamedPermissionSet("MyCompany"))
```

```

);
WorkSite.Name = "Work_Site";

// Create the Internet_Code code group that matches all code
// run from the Internet Zone and grants it Internet permissions.
UnionCodeGroup Internet = new UnionCodeGroup(
    new ZoneMembershipCondition(SecurityZone.Internet),
    new PolicyStatement(p.GetNamedPermissionSet("Internet"))
);
Internet.Name = "Internet_Code";

// Add the My_Site and Work_Site code groups as children of the
// Internet code group
Internet.AddChild(MySite);
Internet.AddChild(WorkSite);

// Create the My_Code code group that matches all code
// run from the My_Computer Zone and grants it FullTrust.
UnionCodeGroup MyCode = new UnionCodeGroup(
    new ZoneMembershipCondition(SecurityZone.MyComputer),
    new PolicyStatement(p.GetNamedPermissionSet("FullTrust"))
);
MyCode.Name = "My_Code";

// Create the root UnionCodeGroup that matches all code,
// but grants no permissions.
UnionCodeGroup Root = new UnionCodeGroup(
    new AllMembershipCondition( ),
    new PolicyStatement(p.GetNamedPermissionSet("Nothing"))
);
Root.Name = "All_Code";

// Add the My_Code and Internet_Code groups as children of the
// Root code group
Root.AddChild(MyCode);
Root.AddChild(Internet);

// Assign the code group tree to the PolicyLevel
p.RootCodeGroup = Root;

# Visual Basic .NET

' Create a new application domain policy level.
Dim p As PolicyLevel = PolicyLevel.CreateAppDomainLevel( )

' Create the MyCompany named permission set as a copy of
' the default LocalIntranet named permission set
p.AddNamedPermissionSet( _
p.GetNamedPermissionSet("LocalIntranet").Copy("MyCompany"))

' Create the My_Site code group that matches all code
' run from the "www.mysite.com" Site and grants it FullTrust.

```

```

Dim MySite As UnionCodeGroup = New UnionCodeGroup( _
New SiteMembershipCondition("www.mysite.com"), _
New PolicyStatement(p.GetNamedPermissionSet("FullTrust")))
MySite.Name = "My_Site"

' Create the Work_Site code group that matches all code
' run from the "www.company.com" Site and grants it the
' MyCompany, permission set.
Dim WorkSite As UnionCodeGroup = New UnionCodeGroup( _
New SiteMembershipCondition("www.company.com"), _
New PolicyStatement(p.GetNamedPermissionSet("MyCompany")))
WorkSite.Name = "Work_Site"

' Create the Internet_Code code group that matches all code
' run from the Internet Zone and grants it Internet permissions.
Dim Internet As UnionCodeGroup = New UnionCodeGroup( _
New ZoneMembershipCondition(SecurityZone.Internet), _
New PolicyStatement(p.GetNamedPermissionSet("Internet")))
Internet.Name = "Internet_Code"

' Add the My_Site and Work_Site code groups as children of the
' Internet code group
Internet.AddChild(MySite)
Internet.AddChild(WorkSite)

' Create the My_Code code group that matches all code
' run from the My_Computer Zone and grants it FullTrust.
Dim MyCode As UnionCodeGroup = New UnionCodeGroup( _
New ZoneMembershipCondition(SecurityZone.MyComputer), _
New PolicyStatement(p.GetNamedPermissionSet("FullTrust")))
MyCode.Name = "My_Code"

' Create the root UnionCodeGroup that matches all code,
' but grants no permissions.
Dim Root As UnionCodeGroup = New UnionCodeGroup( _
New AllMembershipCondition( ), _
New PolicyStatement(p.GetNamedPermissionSet("Nothing")))
Root.Name = "All_Code"

' Add the My_Code and Internet_Code groups as children of the
' Root code group
Root.AddChild(MyCode)
Root.AddChild(Internet)

' Assign the code group tree to the PolicyLevel
p.RootCodeGroup = Root

```

8.2.3 Programming the Security Manager

The `System.Security.SecurityManager` class contains a set of static members that provide access to critical security system functionality and data. Most members of `SecurityManager` require the

caller to have the `ControlPolicy` permission; `ControlPolicy` is an element of `System.Security.Permissions.SecurityPermission`, which we discussed in [Chapter 7](#). The `SecurityManager` functionality protected by `ControlPolicy` means that `ControlPolicy` is one of the highest trust permissions you can grant to code; you must have absolute confidence in the source and integrity of the code to which you grant the `ControlPolicy` permission.

[Table 8-8](#) summarizes the members of the `SecurityManager` class. The far right column titled "C" identifies those members that require the `ControlPolicy` permission.

Table 8-8. Members of the `SecurityManager` class

Member	Description	C
Properties		
<code>CheckExecutionRights</code>	Controls whether the runtime enforces the <code>Execution</code> permission.	
<code>SecurityEnabled</code>	Controls whether the runtime enforces code-access security.	
Methods		
<code>IsGranted</code>	Checks if the calling code has a specified permission.	
<code>LoadPolicyLevelFromFile</code>	Replaces the specified policy level with a policy level loaded from a file; the file must contain an XML representation of the policy level. Use a member of the <code>PolicyLevelType</code> enumeration to identify the current policy level to replace.	
<code>LoadPolicyLevelFromString</code>	Like <code>LoadPolicyLevelFromFile</code> but reads the XML definition of the new policy level from a <code>String</code> .	
<code>PolicyHierarchy</code>	Provides access to the currently active policy levels via an <code>IEnumerator</code> .	
<code>ResolvePolicy</code>	Returns a <code>PermissionSet</code> containing all of the permissions the policy resolution process would grant based on a specified <code>Evidence</code> collection.	
<code>ResolvePolicyGroup</code>	Returns an <code>IEnumerator</code> containing a set of <code>CodeGroup</code> objects representing the code groups to which a specified <code>Evidence</code> collection would grant membership.	
<code>SavePolicy</code>	Saves the security configuration, including all policy levels.	
<code>SavePolicyLevel</code>	Saves the specified policy level to the same location from which it was loaded.	

The `SecurityEnabled` property is critical to the overall operation of the security system. `SecurityEnabled` is the master switch for code access security. Setting `SecurityEnabled` to `false` causes all demands for code-access or identity permissions to succeed; role-based security is not affected. This is effectively the same as giving all code `FullTrust` but has less overhead, because the runtime bypasses the security system when code makes security demands.



`SecurityManager` implements one of the most important members of the whole security system. The `SecurityEnabled` property allows you to turn off all code-access security, meaning that any code can perform any operation regardless of evidence and permissions.

The `CheckExecutionRights` property controls whether the runtime enforces the `Execution` permission, which is also an element of the `SecurityPermission` class. When `CheckExecutionRights` is `false`, any code can run regardless of whether it has the `Execution` permission.

The runtime will not store changes to the `SecurityEnabled` and `CheckExecutionRights` properties until you call the `SavePolicy` method. Even after calling `SavePolicy`, the effect on the current process is unpredictable, and only new processes will work reliably with the new settings:

```
# C#

// Turn on security, turn off execution checking, and save the settings
SecurityManager.SecurityEnabled = true;
SecurityManager.CheckExecutionRights = false;
SecurityManager.SavePolicy( );

# Visual Basic.NET

' Turn on security, turn off execution checking, and save the settings
SecurityManager.SecurityEnabled = True
SecurityManager.CheckExecutionRights = False
SecurityManager.SavePolicy( )
```

The reason you are most likely to be tempted to use both `SecurityEnabled` and `CheckExecutionRights` is performance. In an environment where security is not required, it makes sense to remove the overhead of runtime security checks, but the performance gains will vary greatly depending on the nature of the application. In addition, you should consider thoroughly what other factors are affecting application performance before you decide to turn off code-access security.

The `IsGranted`, `ResolvePolicy`, and `ResolvePolicyGroup` members are useful for testing and debugging but are not as useful as they might initially seem. `IsGranted` only determines if the grant set of the calling code contains a specified permission—you should not use it as a substitute for the normal security control statements we discussed in [Chapter 7](#). `ResolvePolicy` and `ResolvePolicyGroup` both take an `Evidence` collection and run it through the policy resolution process. `ResolvePolicy` returns a grant set based on the evidence provided, and `ResolvePolicyGroup` returns the code groups to which the evidence qualifies for membership.

Of more use is the `PolicyHierarchy` method that returns a `System.Collections.IEnumerator` containing `PolicyLevel` objects for each of the active policy levels. Stepping through the enumerator, you can obtain a `PolicyLevel` object representing one of the active policy levels, allowing you to edit its content. Use the `PolicyLevel.Label` property to identify the desired `PolicyLevel`:

```
# C#

// Get the enumeration of policy levels.
IEnumerator e = SecurityManager.PolicyHierarchy( );
```

```

// Step through the PolicyLevel objects, find the User policy
// and display it to the console.
while(e.MoveNext( )) {

    PolicyLevel p = e.Current as PolicyLevel;

    if (p.Label == "User") {
        Console.WriteLine(p.ToXml( ).ToString( ));
    }
}

# Visual Basic .NET

' Get the enumeration of policy levels.
Dim e As IEnumerator = SecurityManager.PolicyHierarchy( )

' Step through the PolicyLevel objects, find the User policy
' and display it to the console.
While e.MoveNext( )

    Dim p As PolicyLevel = CType(e.Current,PolicyLevel)

    If p.Label = "User" Then
        Console.WriteLine(p.ToXml( ).ToString( ))
    End If
End While

```

After editing an active `PolicyLevel`, you must call `SavePolicy` or `SavePolicyLevel` method if you want to save the changes to disk so that they affect future processes. `SavePolicy` saves all policy levels and policy settings, but `SavePolicyLevel` saves the configuration of a specified `PolicyLevel`. `SavePolicyLevel` saves the specified `PolicyLevel` back to its source location, which is contained in the `PolicyLevel.StoreLocation` property.

8.2.4 Programming Application Domain Policy

Application domain policy is the final layer of security policy evaluated when determining the code-access permissions to grant an assembly. Regardless of the permissions granted to an assembly based on the enterprise, machine, and user policies, you can further lock down its permissions using application domain policy. Application domain policy is resolved even if the assembly is a member of a `LevelFinal` code group in a higher policy level.

You configure application domain policy by calling the `AppDomain.SetAppDomainPolicy` method and passing a configured `PolicyLevel` object. Application domain policy can be set only once per application domain—setting it a second time results in a `System.Security.Policy.PolicyException`. Your code must have the `ControlDomainPolicy` permission to call the `SetAppDomainPolicy` method. `ControlDomainPolicy` is an element of `SecurityPermission`, which we discussed in [Chapter 7](#):

```
# C#
```

```

public void SetAppDomainPolicy(
    PolicyLevel domainPolicy
);

# Visual Basic .NET

NotOverridable Public Sub SetAppDomainPolicy( _
    ByVal domainPolicy As PolicyLevel
)

```

Any assemblies loaded into an application domain before you configure the application domain policy are subject to the standard policy resolution results from the enterprise, machine, and user levels. Application domain policy affects only those assemblies loaded after it's set. This allows you to load assemblies that require higher trust, before locking down the application domain through the additional layer of policy.

The need to implement application domain policy will not be an everyday occurrence, but when writing applications that are responsible for loading other code, the capability is invaluable.

[Example 8-4](#) uses the `SetDomainPolicy` method to create a policy level with the code group structure shown in [Figure 8-10](#) and assign it to a specified application domain. The root code group matches all code but grants the `Nothing` permission set. The second layer of code groups determines the permissions to grant based on `Url` evidence of a loaded assembly. You grant assemblies loaded from the `Trusted` folder the `FullTrust` permission set, assemblies loaded from the `Untrusted` folder the `Internet` permission set, and assemblies loaded from the `Demo` the `Execution` permission set. You grant all other assemblies no permissions. Because we expect an assembly to have only one piece of `Url` evidence, we make the root code group of type `FirstMatchCodeGroup`, saving the runtime from performing unnecessary policy resolution processing.

Figure 8-10. Sample application domain policy

Example 8-4. Creating a policy level and assigning it to an application domain

```

# C#

private static void SetDomainPolicy(AppDomain domain) {

```

```

        // Create a new PolicyLevel for the application domain
        PolicyLevel policy = PolicyLevel.CreateAppDomainLevel( );

        // Create a new FirstMatchCodeGroup that matches all code and
        // grants the named permission set "Nothing".
        policy.RootCodeGroup = new FirstMatchCodeGroup(
            new AllMembershipCondition( ),
            new PolicyStatement(policy.GetNamedPermissionSet("Nothing"))
        );

        // Determine the base path name for the URL evidence
        String basePath = "file://" + AppDomain.CurrentDomain.BaseDirectory;

        // Create the code groups that apply to the assemblies loaded
        // from specific folders. Make each of the the code groups
        // Exclusive so that they set the absolut maximum permisisions a
        // plugin can have.
        policy.RootCodeGroup.AddChild(new UnionCodeGroup(
            new UrlMembershipCondition(basePath + "Trusted/*"),
            new PolicyStatement(policy.GetNamedPermissionSet("FullTrust"),
                PolicyStatementAttribute.Exclusive)
        ));

        policy.RootCodeGroup.AddChild(new UnionCodeGroup(
            new UrlMembershipCondition(basePath + "Untrusted/*"),
            new PolicyStatement(policy.GetNamedPermissionSet("Internet"),
                PolicyStatementAttribute.Exclusive)
        ));

        policy.RootCodeGroup.AddChild(new UnionCodeGroup(
            new UrlMembershipCondition(basePath + "Demo/*"),
            new PolicyStatement(policy.GetNamedPermissionSet("Execution"),
                PolicyStatementAttribute.Exclusive)
        ));

        // Assign the policy to the specified application domain.
        domain.SetAppDomainPolicy(policy);
    }

```

```
# Visual Basic .NET
```

```
Private Shared Sub SetDomainPolicy(ByVal domain As AppDomain)
```

```

    ' Create a new PolicyLevel for the application domain
    Dim policy As PolicyLevel = _
        PolicyLevel.CreateAppDomainLevel( )

```

```

    ' Create a new FirstMatchCodeGroup that matches all code and
    ' grants the named permission set "Nothing".
    policy.RootCodeGroup = New FirstMatchCodeGroup( _
        New AllMembershipCondition( ), _

```

```
New PolicyStatement(policy.GetNamedPermissionSet("Nothing"))

' Determine the base path name for the URL evidence
Dim basePath As String = _
"file://" & AppDomain.CurrentDomain.BaseDirectory;

' Create the code groups that apply to the assemblies loaded
' from specific folders. Make each of the the code groups
' Exclusive so that they set the absolut maximum permisisions a
' plugin can have.
policy.RootCodeGroup.AddChild(New UnionCodeGroup( _
New UrlMembershipCondition(basePath & "Trusted/*"), _
New PolicyStatement(policy.GetNamedPermissionSet("FullTrust"), _
PolicyStatementAttribute.Exclusive)))

policy.RootCodeGroup.AddChild(New UnionCodeGroup( _
New UrlMembershipCondition(basePath & "Untrusted/*"), _
New PolicyStatement(policy.GetNamedPermissionSet("Internet"), _
PolicyStatementAttribute.Exclusive)))

policy.RootCodeGroup.AddChild(New UnionCodeGroup( _
New UrlMembershipCondition(basePath & "Demo/*"), _
New PolicyStatement(policy.GetNamedPermissionSet("Execution"), _
PolicyStatementAttribute.Exclusive)))

' Assign the policy to the specified application domain.
domain.SetAppDomainPolicy(policy)
End Sub
```

[\[Team LiB \]](#)

[[Team LiB](#)]

8.3 Extending the .NET Framework

In [Section 6.3.1.1](#), we developed the `Author` evidence class, which identifies the developer of an assembly. In this section, we extend CAS so that you can base policy resolution on the presence of `Author` evidence.

8.3.1 Custom Membership Conditions Explained

To enable `Author` evidence to drive policy resolution, first you create a custom membership condition class. Using the techniques described in [Section 8.2](#), you can then use the custom membership condition in code groups to test the values of any `Author` evidence presented by an assembly or application domain. Here are the key requirements of a custom membership condition class:

Implement `IMembershipCondition`

Membership condition classes must implement the `System.Security.Policy.IMembershipCondition` interface, which extends both the `System.Security.ISecurityPolicyEncodable` and `System.Security.ISecurityEncodeable` interfaces. We summarize the members of each interface in [Table 8-9](#).

Implement default constructor

Membership condition classes must implement a default (empty) constructor. The runtime uses this constructor when it needs to create a membership condition object from XML contained in the security policy files.

Make serializable

Membership condition classes must be serializable. Because of their simplicity, application of the `System.SerializableAttribute` is usually sufficient. Those membership condition classes that have more complex serialization requirements must implement the `System.Runtime.Serialization.ISerializable` interface.

Unless you implement `ISerializable`, deserialization of an object does not result in the execution of a constructor, and therefore you should not depend on constructor logic to configure or modify the state of the deserialized object.

Table 8-9. Members of the `IMembershipCondition`, `ISecurityPolicyEncodable`, and `ISecurityEncodeable` interfaces

Member	Description
<code>IMembershipCondition</code> interface	

Member	Description
<code>Check</code>	Determines whether an <code>Evidence</code> collection contains evidence that satisfies the membership criterion
<code>Copy</code>	Returns a copy of the membership condition
<code>Equals</code>	Tests a <code>System.Object</code> against the <code>IMembershipCondition</code> object
<code>ToString</code>	Returns a <code>System.String</code> representation of the membership condition
<code>ISecurityEncodable</code> interface	
<code>FromXml</code>	Reconstructs the <code>IMembershipCondition</code> object's state from a specified <code>SecurityElement</code>
<code>ToXml</code>	Creates and returns a <code>SecurityElement</code> containing an XML object model representing the <code>IMembershipCondition</code> object's state
<code>ISecurityPolicyEncodable</code>	
<code>FromXml</code>	The same as <code>ISecurityEncodable.ToXml</code> but handles any policy level-specific state
<code>ToXml</code>	The same as <code>ISecurityEncodable.FromXml</code> but includes any policy level-specific state

In addition to these requirements, many of the guidelines we outlined for evidence in [Chapter 6](#) also hold true for custom membership conditions. These include:

- Make membership conditions simple. Membership conditions classes have access to the entire `Evidence` collection of an assembly or application domain in the `Check` method, and therefore it is possible to create custom membership conditions that test complex membership criteria spanning multiple items of evidence. You must consider who will use your membership condition class and design it appropriately. Creating membership classes that test multiple or complex sets of data can result in security policy that is difficult for security administrators and users to configure and understand.
- Make membership conditions lightweight. You should consider the amount of memory used by your membership condition classes as well as avoid lengthy processing tasks, such as network, file, and database access during membership evaluation.
- Make membership conditions noninheritable. It is safest to mark your membership condition classes as `sealed` in C# and `NotInheritable` in Visual Basic .NET so that malicious code cannot subclass them in an attempt to change its behavior and subvert CAS, or confuse security administrators.
- Override `Object.ToString`. The security administration tools use `ToString` to display a message that describes the condition and value that a membership is configured to match against. The use of this information necessitates that the output be concise. For example, "Site = *.oreilly.com" or "Author = Peter."

8.3.2 Defining the AuthorMembershipCondition Class

Following the naming pattern used in the .NET class library, we name the membership condition class `AuthorMembershipCondition`. The `AuthorMembershipCondition` class is relatively simple; however, there is still a fair amount of code involved because we have to implement the eight members defined in the `IMembershipCondition` interface. We will break the `AuthorMembershipCondition` class into manageable sections for explanation.

The class declaration for `AuthorMembershipCondition` specifies that it is `sealed` (C#) and `NotInheritable` (Visual Basic .NET) to stop malicious code from subclassing it in order to subvert CAS. You should also specify the implementation of `IMembershipCondition` and annotate `AuthorMembershipCondition` with the `System.Serializable` attribute-satisfying the two major requirements of membership condition classes.

We define a single private `System.String` data member named `AuthorName`, which will contain the name of the author on which an `AuthorMembershipCondition` object will base its membership test:

```
# C#

using System;
using System.Security;
using System.Collections;
using System.Reflection;
using System.Security.Policy;

[assembly:AssemblyKeyFile("Keys.snk")]
[assembly:AssemblyVersion("1.0.0.0")]

namespace ORA.DotNetSecurity.Policy {

    [Serializable]
    public sealed class AuthorMembershipCondition : IMembershipCondition {

        private string AuthorName = "";

# Visual Basic .NET

Imports System
Imports System.Security
Imports System.Collections
Imports System.Reflection
Imports System.Security.Policy

<assembly:AssemblyKeyFile("Keys.snk")>
<assembly:AssemblyVersion("1.0.0.0")>

Namespace ORA.DotNetSecurity.Policy

    <Serializable> _
    Public NotInheritable Class AuthorMembershipCondition
        Implements IMembershipCondition
```

```
Private AuthorName As String = ""
```

We define two constructors for `AuthorMembershipCondition`. The first is a default (empty) constructor. The runtime uses this constructor to create `AuthorMembershipCondition` objects before calling the `FromXml` method to recreate the state of an `AuthorMembershipCondition` from XML stored in the policy configuration files. The second constructor is for general use and takes a `System.String` argument containing the name of the author on which the `AuthorMembershipCondition` should base its membership test:

```
# C#
```

```
public AuthorMembershipCondition( ) {
    this.AuthorName = "";
}

public AuthorMembershipCondition(string author) {
    if (VerifyAuthorName(author)) {
        this.AuthorName = author;
    }
}
```

```
# Visual Basic.NET
```

```
Public Sub New( )
    Me.AuthorName = ""
End Sub

Public Sub New(ByVal author As String)
    If VerifyAuthorName(author) Then
        Me.AuthorName = author
    End If
End Sub
```

We include a `Name` property to provide controlled access to the private `AuthorName` data member and the private `VerifyAuthorName` utility method, which we use in both the `Name` property and class constructor to validate the author name provided. To simplify `AuthorMembershipCondition` you check only if the author name is `null` (C#) or `Nothing` (Visual Basic .NET)-in which case, you throw a `System.ArgumentNullException`. In a production-quality membership condition class, it is good practice to verify the condition data more thoroughly and throw a `System.ArgumentException` if the data is invalid. For example, it would be a good idea to check for invalid characters and limit the size of the name provided:

```
# C#
```

```
// Property to get/set the author name
public string Name {
    get {
```

```

        return AuthorName;
    }

    set {
        if (VerifyAuthorName(value)) {
            this.AuthorName = value;
        }
    }
}

// Utility method to verify that the author name
// is not null.
private bool VerifyAuthorName(string author) {

    if (author == null) {
        throw new ArgumentNullException("author");
    } else {
        return true;
    }
}

```

Visual Basic .NET

```

' Property to get/set the author name
Public Property Name( ) As String
    Get
        Return AuthorName
    End Get
    Set (ByVal Value As String)
        If VerifyAuthorName(Value) Then
            Me.AuthorName = Value
        End If
    End Set
End Property

' Utility method to verify that the author name
' is not null.
Private Function VerifyAuthorName(ByVal author As String) _
    As Boolean

    If author Is Nothing Then
        Throw New ArgumentNullException("author")
    Else
        Return True
    End If
End Function

```

The real work of the `AuthorMembershipCondition` class takes place in the `Check` method. The `Check` method takes an `Evidence` collection and enumerates the contained evidence objects to determine if any of them are `Author` objects. If `Check` finds an `Author` object, it compares the `Author.Name` property with its own private `AuthorName` data member. If the two names match, `Check` returns

true; otherwise, Check returns false.

For simplicity, we have implemented a straightforward case-sensitive string comparison, but you might also consider support for non-case-sensitive comparisons and wildcard matches. The membership condition logic can be arbitrarily complex, but remember that the runtime may call Check many times during policy resolution, and time-consuming membership condition evaluations will affect application performance:

C#

```
// Determines whether the Evidence meets the membership
// condition.
public bool Check( Evidence evidence ) {

    // Return false if the Evidence is null
    if (evidence == null) {
        return false;
    }

    Author auth = null;

    // Enumerate across the host and assembly evidence
    // collections.
    IEnumerator enumerator = evidence.GetEnumerator( );
    while (enumerator.MoveNext( ))
    {
        auth = enumerator.Current as Author;

        // If the evidence is of type Author and the
        // author names match, the evidence meets the
        // membership condition.
        if (auth != null) {
            if (auth.Name == this.AuthorName) {
                return true;
            }
        }
    }
    return false;
}
```

Visual Basic .NET

```
' Determines whether the Evidence meets the membership
' condition.
Public Function Check(ByVal evidence As Evidence) As Boolean _
    Implements IMembershipCondition.Check

    ' Return false if the Evidence is Nothing
    If evidence Is Nothing Then
        Return False
    End If
```

```

Dim auth As Author = Nothing

' Enumerate across the host and assembly evidence
' collections.
Dim enumerator As IEnumerator = evidence.GetEnumerator( )
While enumerator.MoveNext( )

    auth = CType(enumerator.Current, Author)

    ' If the evidence is of type Author and the
    ' author names match, the evidence meets the
    ' membership condition.
    If Not auth Is Nothing Then
        If auth.Name = Me.AuthorName Then
            Return True
        End If
    End If
End While

Return False
End Function

```

The remaining members of the `IMembershipCondition` interface are straightforward. The `Copy` method returns a clone of the current `AuthorMembershipCondition` object, and the `Equals` method compares two `AuthorMembershipCondition` objects for equality based on the value of their private `AuthorName` data members instead of comparing object references. Because `Object` also defines an `Equals` method, you must override it with the one defined in the `IMembershipCondition` interface. As a result, you must also override the `GetHashCode` method:

```
# C#
```

```

// Creates a copy of the membership condition
public IMembershipCondition Copy( ) {

    return new AuthorMembershipCondition(this.AuthorName);
}

// Compares an object for equality based on the author's
// name, not the object reference.
public override bool Equals(object obj) {

    AuthorMembershipCondition that =
        (obj as AuthorMembershipCondition);

    if (that != null) {
        if (this.AuthorName == that.AuthorName) {
            return true;
        }
    }
    return false;
}

```

```

// We must override GetHashCode because we override
// Object.Equals. Returns a hash code based on the
// author's name.
public override int GetHashCode( ) {

    return this.AuthorName.GetHashCode( );
}

```

```
# Visual Basic .NET
```

```

' Creates a copy of the membership condition
Public Function Copy( ) As IMembershipCondition _
    Implements IMembershipCondition.Copy

    Return New AuthorMembershipCondition(Me.AuthorName)
End Function

' Compares an object for equality based on the author's
' name, not the object reference.
Public Overloads Function Equals(ByVal obj As Object) As Boolean _
    Implements IMembershipCondition.Equals

    Dim that As AuthorMembershipCondition = _
        CType(obj, AuthorMembershipCondition)

    If Not that Is Nothing Then
        If Me.AuthorName = that.AuthorName Then
            Return True
        End If
    End If
    Return False
End Function

' We must override GetHashCode because we override
' Object.Equals. Returns a hash code based on the
' author's name.
Public Overrides Function GetHashCode( ) As Integer

    Return Me.AuthorName.GetHashCode( )
End Function

```

All of the standard membership condition classes override `Object.ToString` to return a simple human-readable representation of the membership condition they define. The administration tools display this information, and it should be relatively concise. For consistency, we take the same approach:

```
# C#
```

```

// Returns a simple string representation of the
// membership condition
public override string ToString( ) {

```

```

        return "Author - " + AuthorName;
    }

```

```
# Visual Basic .NET
```

```

' Returns a simple string representation of the
' membership condition
Public Overrides Function ToString( ) As String _
    Implements IMembershipCondition.ToString

    Return "Author - " & AuthorName
End Function

```

Both the `ISecurityEncodable` and `ISecurityPolicyEncodable` interfaces define a `ToXml` method that returns a `System.Security.SecurityElement` containing an XML object model of the `AuthorMembershipCondition` object. The `AuthorMembershipCondition` class does not differentiate between different policy levels and therefore implements both methods the same.

Most importantly, the root XML element must be "IMembershipCondition" or the administrative tools will not be able to import the XML describing the custom membership condition. The XML representation of all standard membership condition classes includes a `version` attribute that represents the XML format used to represent the membership condition. We have taken the same approach, which gives flexibility should future versions of the `AuthorMembershipCondition` class require a different structure:

```
# C#
```

```

// Return a SecurityElement containing an XML object
// model representing the membership condition.
public SecurityElement ToXml( ) {

    return this.ToXml(null);
}

// Return a SecurityElement containing an XML object
// model representing the membership condition. We have
// no need to differentiate between policy levels and so
// we ignore the "level" argument.
public SecurityElement ToXml(PolicyLevel level) {

    // Create a new "IMembershipCondition" element
    SecurityElement se =
        new SecurityElement("IMembershipCondition");

    // Add fully qualified type name for membership condition
    se.AddAttribute(
        "class",
        this.GetType( ).AssemblyQualifiedName
    );
}

```

```

        // Add an XML version number of "1"
        se.AddAttribute("version", "1");

        // Add the author name
        se.AddAttribute("name", AuthorName);

        // Return the new SecurityElement
        return se;
    }

```

```
# Visual Basic .NET
```

```

' Return a SecurityElement containing an XML object
' model representing the membership condition.
Public Function ToXml( ) As SecurityElement _
    Implements ISecurityEncodable.ToXml

    Return Me.ToXml(Nothing)
End Function

' Return a SecurityElement containing an XML object
' model representing the membership condition. We have
' no need to differentiate between policy levels and so
' we ignore the "level" argument.
Public Function ToXml(ByVal level As PolicyLevel) _
    As SecurityElement Implements ISecurityPolicyEncodable.ToXml

    ' Create a new "IMembershipCondition" element
    Dim se As SecurityElement = _
        New SecurityElement("IMembershipCondition")

    ' Add fully qualified type name for membership condition
    se.AddAttribute( _
        "class", _
        Me.GetType( ).AssemblyQualifiedName _
    )

    ' Add an XML version number of "1"
    se.AddAttribute("version", "1")

    ' Add the author name
    se.AddAttribute("name", AuthorName)

    ' Return the new SecurityElement
    Return se
End Function

```

The reverse of the `ToXml` method is `FromXml`, which takes a `SecurityElement` and reconstructs the object state. As with `ToXml`, both the `ISecurityEncodable` and `ISecurityPolicyEncodable` interfaces define a `FromXml` method. Again, you implement both methods the same, because you do not need to perform policy level-specific state configuration:


```
# C#
```

```
// Reconstruct the state of the membership condition
// object from the SecurityElement provided.
// fromExtract state from a SecurityElement
public void FromXml(SecurityElement e) {

    this.FromXml(e, null);
}

// Reconstruct the state of the membership condition
// object from the SecurityElement provided. We have
// no need to differentiate between policy levels and so
// we ignore the "level" argument.
public void FromXml(SecurityElement e, PolicyLevel level) {

    // Ensure we have a SecurityElement to work with
    if (e == null) throw new ArgumentNullException("e");

    // Ensure the SecurityElement is an AuthorMembershipCondition
    if (e.Tag != "IMembershipCondition") {
        throw new ArgumentException
            ("Element must be IMembershipCondition");
    } else {
        // Extract the author name from the SecurityElement
        this.AuthorName = e.Attribute("name");
    }
}
}
}
```

```
# Visual Basic .NET
```

```
' Reconstruct the state of the membership condition
' object from the SecurityElement provided.
' fromExtract state from a SecurityElement
Public Sub FromXml(ByVal e As SecurityElement) _
    Implements ISecurityEncodable.FromXml

    Me.FromXml(e, Nothing)
End Sub

' Reconstruct the state of the membership condition
' object from the SecurityElement provided. We have
' no need to differentiate between policy levels and so
' we ignore the "level" argument.
Public Sub FromXml(ByVal e As SecurityElement, _
    ByVal level As PolicyLevel) _
    Implements ISecurityPolicyEncodable.FromXml
```

```

    ' Ensure we have a SecurityElement to work with
    If e Is Nothing Then
        Throw New ArgumentNullException("e")
    End If

    ' Ensure the SecurityElement is an AuthorMembershipCondition
    If e.Tag <> "IMembershipCondition" Then
        Throw New ArgumentException _
            ("Element must be IMembershipCondition")
    Else
        ' Extract the author name from the SecurityElement
        Me.AuthorName = e.Attribute("name")
    End If
End Sub
End Class
End Namespace

```

8.3.3 Building the AuthorMembershipCondition Assembly

The `AuthorMembershipCondition` class has a dependency on the `Author` class you created in the [Chapter 7](#). For convenience, you could build all of the `Author`-related security classes into a single assembly. This would simplify security administration if you needed to distribute CAS extensions to third parties, but because the `Author` class is evidence, you would need to add the assembly to the fully trusted assembly list of every policy level. This would grant the assembly full trust at runtime, which many people would find unacceptable.

For the purpose of an example, build `AuthorMembershipCondition` into its own assembly using the following command. Remember to ensure that you place a copy of the `Author.dll` assembly from [Chapter 7](#) in the directory where you build `AuthorMembershipCondition`:

```

# C#

csc /t:library /reference:Author.dll AuthorMembershipCondition.cs

# Visual Basic .NET

vbc /t:library /reference:Author.dll AuthorMembershipCondition.vb

```

8.3.4 Using the AuthorMembershipCondition Membership Condition

The Microsoft .NET Framework Configuration tool (*Mscorcfg.msc*) provides a graphical interface through which you can configure security policy. *Mscorcfg.msc* implements specific support for the standard membership condition classes and provides user-friendly interfaces through which you can configure their membership condition parameters. However, the support for using custom membership conditions is rudimentary, requiring you to import an XML fragment describing the membership condition to use. This XML is the result of calling `ToXml().ToString()` on a custom membership condition object.

In this example, we will use the Code Access Security Policy tool (*Caspol.exe*) to configure a code group in the user policy level that uses `AuthorMembershipCondition` to evaluate membership. We will describe the *Caspol.exe* commands necessary to perform this configuration, but you should look

at [Chapter 9](#) for complete details of both the command-line and graphical administration tools.

8.3.4.1 Installing security assemblies

As we discussed in [Section 8.1.1.3](#), assemblies that provide CAS extensions (such as the *AuthorMembershipCondition.dll* assembly) must be fully trusted by the policy level in which they are used. To make *AuthorMembershipCondition.dll* fully trusted by the user policy level, execute the following commands from the directory where the *AuthorMembershipCondition* is located:

```
gacutil -i AuthorMembershipCondition.dll
caspol -user -addfulltrust AuthorMembershipCondition.dll
```

The first command installs *AuthorMembershipCondition* into the global assembly cache (which you must do before you can make it a fully trusted assembly). The second command makes *AuthorMembershipCondition* a fully trusted assembly in the user policy level.

8.3.4.2 Generating AuthorMembershipCondition XML

You must create an XML representation of the membership condition you want to assign to the code group. Although the required XML is simple enough that you could create it manually, it is much easier and safer to instantiate an *AuthorMembershipCondition* object and write the contents of the *SecurityElement* returned by the *ToXml* to disk.

If you distribute custom membership condition classes to users or customers, we advise you to create a simple utility to perform the creation of the required XML fragments. The *CreateAuthorMembership* class takes an author name as a command-line argument, instantiates an *AuthorMembershipCondition* object, and writes an XML representation of it to disk:

```
# C#

using System;
using System.IO;
using ORA.DotNetSecurity.Policy;

public class CreateAuthorMembership {

    public static void Main(string[] args) {

        // Create a new AuthorMembershipCondition based
        // on the name of the author provided
        AuthorMembershipCondition amc =
            new AuthorMembershipCondition(args[0]);

        // Generate the name of the XML output file
        String file = args[0] + ".xml";

        // Render the AuthorMembershipCondition to
        // XML and write it to a file
        StreamWriter strm = new StreamWriter(file);
```

```

        strm.Write(amc.ToXml( ).ToString( ));
        strm.Close( );

        // Display result
        Console.WriteLine("Created author membership condition : " +
            file);
    }
}

# Visual Basic .NET

Imports System
Imports System.IO
Imports ORA.DotNetSecurity.Policy

Public Class CreateAuthorMembership

    Public Shared Sub Main(ByVal args( ) As String)

        ' Create a new AuthorMembershipCondition based
        ' on the name of the author provided
        Dim amc As AuthorMembershipCondition = _
            New AuthorMembershipCondition(args(0))

        ' Generate the name of the XML output file
        Dim file As String = args(0) + ".xml"

        ' Render the AuthorMembershipCondition to
        ' XML and write it to a file
        Dim strm As StreamWriter = New StreamWriter(file)
        strm.Write(amc.ToXml( ).ToString( ))
        strm.Close( )

        ' Display result
        Console.WriteLine("Created author membership condition : " & _
            file)
    End Sub
End Class

```

Build the `CreateAuthorMembership` utility using the following command, remembering the dependency on the *AuthorMembershipCondition.dll* library:

```
# C#
```

```
csc /reference:AuthorMembershipCondition.dll CreateAuthorMembership.cs
```

```
# Visual Basic .NET
```

```
vbc /reference:AuthorMembershipCondition.dll CreateAuthorMembership.vb
```

Run the command `CreateAuthorMembership Peter` to create the XML representation of an `AuthorMembershipCondition` that matches `Author` evidence for the author "Peter." Here is the

content of the *Peter.xml*/file. The `PublicKeyToken` value will change based on the keys you used to give `create` a strong name:

```
<IMembershipCondition class="ORA.DotNetSecurity.Policy.
AuthorMembershipCondition, AuthorMembershipCondition,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=cc5e18bc387194b3"
    version="1"
    name="Peter" />
```

8.3.4.3 Configuring security policy

With the *Author.dll* assembly configured as a fully trusted assembly in the user policy level, you are ready to create a code group whose membership condition is based on `Author` evidence. Type the following command:

```
caspol -user -addgroup All_Code -custom Peter.xml FullTrust
-name "Peter's code" -description "Code group grants all
code written by Peter full trust"
```

Here is a summary of the arguments used in this command:

-user

Specifies that the command is to affect the user policy level

-addgroup All_Code

Specifies that you want to add a new child code group to the existing group with the label "All_Code"

-custom Peter.xml/FullTrust

Specifies that you are adding a custom membership condition contained in the file *Peter.xml* and that the permission set granted by the code group is the named permission set "FullTrust"

-name "Peter's code"

Specifies the name of the new code group

-description "Code group grants all code written by Peter full trust"

Specifies a description for the new code group

Depending on how *Caspol.exe* is configured, it may prompt you to confirm the change you are about to make with the following message:

```
The operation you are performing will alter security policy.
Are you sure you want to perform this operation? (yes/no)
```

Entering "yes" updates the user security policy and commits the changes to disk; you will see the following message:

```
Added union code group with "-custom" membership condition to the User level.
Success
```

8.3.5 Testing Custom Membership Conditions

The easiest way to test the extensions to the policy resolution process is to use the *Caspol.exe* tool. In [Chapter 6](#), we embedded **Author** evidence in an assembly named *HelloWorld.exe*. We can use *Caspol.exe* and the *HelloWorld.exe* assembly to show that the "Peter's code" code group does detect **Author** evidence and evaluate it correctly. Type the following command from the directory where *HelloWorld.exe* is located:

```
caspol -user -resolvegroup HelloWorld.exe
```

Caspol.exe produces the following output, showing that *HelloWorld.exe* is a member of the code group "Peter's code":

```
Microsoft (R) .NET Framework CasPol 1.0.3705.288  
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
```

```
Level = User
```

```
Code Groups:
```

1. All code: FullTrust
 - 1.1. Author - Peter: FullTrust

```
Success
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 9. Administering Code-Access Security

The final aspect of code-access security (CAS) comprises the tools provided with the .NET Framework SDK that allow you to configure and test the security system. This chapter is not a comprehensive administrative reference for the configuration of .NET Framework security; instead we focus on the information and tools you as a programmer will need during the testing and debugging of your programs. These tools are relevant no matter what type of code you develop, but are particularly useful when developing systems that rely on or manipulate the CAS features we have discussed in the previous four chapters. If you require a more general discussion of these tools and .NET security administration, we refer you to the .NET Framework SDK documentation.

First, we outline the default security policy implemented by the .NET Framework. If you work in a corporate environment, it is likely that a central IT or security group will define a standard security policy. However, outside such environments many average users are unable or reluctant to change the default security policy. It is important that you understand the security restrictions enforced by the most common environment in which your code is likely to run. You can then either program around these restrictions or communicate any special security requirements to the users of your code.

We look at three tools provided with the .NET Framework SDK and show you how to use them in the configuration and testing of security policy:

Permissions View tool

Allows you to view an assembly's declarative security statements

The .NET Framework Configuration tool

Provides a graphical interface through which to configure aspects of the .NET Framework

The Code-Access Security Policy tool

Provides a command-line interface through which to configure the .NET Framework security system

[\[Team LiB \]](#)

[[Team LiB](#)]

9.1 Default Security Policy

The .NET Framework provides a simple and conservative default security policy that makes use of Microsoft Internet Explorer security zones, which map to the `Zone` evidence type that we discussed in [Chapter 6](#).

9.1.1 Security Policy Files

The configuration information for the enterprise, machine, and user policy levels is stored in separate files. Each file contains an XML description of the complete policy level including code groups, named permission sets, and fully trusted assemblies. The runtime uses the XML description to create the policy level's object hierarchy for use during policy resolution, which we discussed in [Chapter 8](#).

[Table 9-1](#) lists the default names and storage locations of the policy files on the Windows 2000, Windows NT, Windows 98, and Windows Me platforms. The location of the enterprise and machine policy files depends on the location of the .NET Framework runtime installation. User policy files are user-specific, and their default location depends on where the operating system stores user files. To support users that run multiple versions of the .NET Framework, the location of user policy files also depends on the version of the .NET Framework to which they apply.

Table 9-1. Security policy file locations

Policy level	Default location
Enterprise	<i>runtime_install_path</i> \Config\Enterprisesec.config
Machine	<i>runtime_install_path</i> \Config\Security.config
User (Windows 2000 and NT)	%USERPROFILE%\Application Data\Microsoft\CLR Security Config\ <i>version</i> \security.config
User (Windows 98 and Me)	%WINDIR%\ <i>username</i> \CLR Security Config\ <i>version</i> \security.config

When the runtime starts, it looks for the enterprise, machine, and user policy files. If any of the files are missing, the runtime will recreate the missing files automatically using the default configuration settings that we discuss in the following sections. This behavior ensures that the runtime never operates without a complete security policy. However, this approach can also give surprising and undesirable results if you are unaware that one of the policy files has been removed. For example, if somebody manages to delete the enterprise policy file in which you have defined a series of security constraints, the new one created by the runtime will contain the default enterprise-level configuration, which gives all code full trust.



Policy files are stored as clear-text XML files and are susceptible to both manual and programmatic manipulation. Because policy files are so important to the security of the .NET runtime, it is essential that you protect them from inadvertent or malicious modification and deletion by using operating system file-access controls, such as those provided by NTFS.

9.1.2 Named Permission Sets

The default enterprise, machine, and user policy levels all contain the set of named permission sets listed in [Table 9-2](#). We summarize the permissions granted by each permission set, but refer you to the .NET Framework SDK documentation for exact details. All of these permission sets except *Everything* are immutable.

Table 9-2. Default named permission sets

Permission set	Description
<code>FullTrust</code>	Unrestricted access to all operations and resources.
<code>SkipVerification</code>	Includes permission for code to skip verification, but nothing else. This set includes only the <code>SecurityPermission.SkipVerification</code> permission. We discuss the importance of verification in Chapter 4 .
<code>Execution</code>	Includes permission for code to execute, but nothing else. This set includes only the <code>SecurityPermission.Execution</code> permission.
<code>Nothing</code>	No access to any operation or resource, not even the permission to execute.
<code>LocalIntranet</code>	A set of permissions deemed by Microsoft to be appropriate for code loaded from the local intranet. This includes the following permissions: <code>DnsPermisison.Unrestricted</code> <code>EnvironmentPermission.Read(Username)</code> <code>EventLogPermission.Instrument</code> <code>FileDialogPermission.Unrestricted</code> <code>IsolatedStoragePermssion.AssemblyIsolaitionByUser</code> <code>PrintingPermission.DefaultPrinting</code> <code>ReflectionPermission.ReflectionEmit</code> <code>SecurityPermission.Execution</code> <code>SecurityPermission.Assertion</code> <code>UIPermission.Unresticted</code>
<code>Internet</code>	A set of permissions deemed by Microsoft to be appropriate for code loaded from the Internet. This includes the following permissions: <code>FileDialogPermission.Open</code> <code>IsolatedStoragePermssion.DomainIsolaitionByUser</code> <code>PrintingPermission.SafePrinting</code> <code>SecurityPermission.Execution</code> <code>UIPermission.SafeTopLevelWindows</code>

Permission set	Description
	<code>UIPermission.OwnClipboard</code>
<code>Everything</code>	When created, the <code>Everything</code> group contains an unrestricted version of each standard code-access permission contained in the .NET class library except the <code>SkipVerification</code> element of the <code>SecurityPermission</code> class.

9.1.3 Enterprise and User Policy Code Groups

The default security policy does not rely on the enterprise and user policy levels to enforce code-access security. Both the default enterprise and user policy levels consist of a single code group named `All_Code`, which has a membership condition of `AllCode` and grants its members the `FullTrust` permission set. This means that without the default machine policy (discussed next), any code regardless of the evidence it presents can perform any action and access any resource.

9.1.4 Machine Policy Code Groups

In the interest of simplicity, the default security policy concentrates all security decisions in the machine policy level. The code group hierarchy of the machine policy level is oriented around Microsoft Internet Explorer security zones. [Figure 9-1](#) shows the group hierarchy of the default machine policy. The root code group, named `All_Code`, gives all code the `Nothing` permission set. Below the `All_Code` root node are five code groups—one for each value of `Zone` evidence. `Internet_Zone` and `Trusted_Zone` use child code groups of type `NetCodeGroup` and `FileCodeGroup` to grant code permission to the URL and web site from where they were downloaded; see [Chapter 8](#) for a description of the different types of code groups.

Figure 9-1. Default machine policy code group hierarchy



Although the default machine policy code group hierarchy remains the same between Versions 1.0 and 1.1 of the .NET Framework, the permissions granted to the `Internet_Zone` code group are different. The .NET Framework Version 1.0 security policy is extremely conservative and grants the `Internet_Zone` code group the `Nothing` permission set. As of Version 1.0 SP1 and including Version 1.1 of the .NET Framework, the `Internet_Zone` code group grants the `Internet` permission set to its members.

9.1.5 Fully Trusted Assemblies

The following is a list of assemblies contained in the fully trusted assemblies list of the default enterprise, machine, and user policy levels. Each of these assemblies contains classes that may be used during the policy-resolution process. You should not remove any of these assemblies from the fully trusted list, as this will almost certainly cause problems during policy resolution:

- `mscorlib.resources`
- `System`
- `System.resources`
- `System.Data`
- `System.Data.resources`
- `System.Drawing`
- `System.Drawing.resources`
- `System.Messaging`
- `System.Messaging.resources`
- `System.ServiceProcess`
- `System.ServiceProcess.resources`
- `System.DirectoryServices`
- `System.DirectoryServices.resources`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

9.2 Inspecting Declarative Security Statements

The Permissions View tool (*Permview.exe*) allows you to view the declarative security statements used in an assembly. This is particularly useful when configuring security policy, as it allows you to view the permission requests contained in the assembly. *Permview.exe* is located in the *lbin* subdirectory of the .NET Framework SDK installation directory. Note that the *lbin* directory is not added to the `Path` environment variable by the SDK-installation process.



Permview.exe only shows declarative security statements and cannot extract the imperative security demands. It overrides from the body of an assembly.

To demonstrate *Permview.exe*, use the `MessageUtil` class shown here, which allows any caller to display a message box containing the value of the `Message` environment variable. `MessageUtil` uses three `RequestMinimum` permission requests to ensure that it has permission to read the `Message` environment variable, called `Assert`, and display safe top-level windows (such as message boxes). The `DisplayMessage` method uses declarative syntax to `Assert` `UIPermission` and `EnvironmentPermission`. This allows any client code to use `DisplayMessage` regardless of the caller's permissions:

```
# C#

using System;
using System.Windows.Forms;
using System.Security.Permissions;

// Request read access to the Message environment variable.
[assembly:EnvironmentPermission(SecurityAction.RequestMinimum,
    Read = "Message")]

// Request permission to Assert.
[assembly:SecurityPermission(SecurityAction.RequestMinimum,
    Assertion = true)]

// Request permission to display safe top level windows.
[assembly:UIPermission(SecurityAction.RequestMinimum,
    Window = UIPermissionWindow.SafeTopLevelWindows)]

public class MessageUtil {

    // Assert the permission to read the Message environment variable and
    // to display top level windows.
    [EnvironmentPermission(SecurityAction.Assert, Read = "Message")]
    [UIPermission(SecurityAction.Assert,
        Window = UIPermissionWindow.SafeTopLevelWindows)]
    public static void DisplayMessage( ) {
```

```

        // Display the value of the Message environment variable
        // in a message box.
        MessageBox.Show(Environment.GetEnvironmentVariable("Message"));
    }
}

```

```
# Visual Basic .NET
```

```
Imports System
Imports System.Windows.Forms
Imports System.Security.Permissions
```

```
' Request read access to the Message environment variable.
<assembly:EnvironmentPermission(SecurityAction.RequestMinimum, _
    Read := "Message")> _
```

```
' Request permission to Assert.
<assembly:SecurityPermission(SecurityAction.RequestMinimum, _
    Assertion := True)> _
```

```
' Request permission to display safe top level windows.
<assembly:UIPermission(SecurityAction.RequestMinimum, _
    Window := UIPermissionWindow.SafeTopLevelWindows)> _
```

```
Public Class MessageUtil
```

```
    ' Assert the permission to read the Message environment variable and
    ' to display top level windows.
```

```
    <EnvironmentPermission(SecurityAction.Assert, Read := "Message"), _
    UIPermission(SecurityAction.Assert, _
    Window := UIPermissionWindow.SafeTopLevelWindows)> _
```

```
    Public Shared Sub DisplayMessage( )
```

```
        ' Display the value of the Message environment variable
        ' in a message box.
```

```
        MessageBox.Show(Environment.GetEnvironmentVariable("Message"))
```

```
    End Sub
```

```
End Class
```

If you build `MessageUtil` into a library named `MessageUtil.dll` and then run the command `permview MessageUtil.dll`, you will see the following XML descriptions of the permission request statements:

```
Microsoft (R) .NET Framework Permission Request Viewer. Version 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
```

minimal permission set:

```
<PermissionSet class="System.Security.PermissionSet"
    version="1">
    <IPermission class="System.Security.Permissions.EnvironmentPermission, mscorl
ib, Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
        version="1"
        Read="Message"/>
    <IPermission class="System.Security.Permissions.SecurityPermission, mscorlib,
```

```

Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
    version="1"
    Flags="Assertion"/>
    <IPermission class="System.Security.Permissions.UIPermission, mscorlib, Versi
on=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
        version="1"
        Window="SafeTopLevelWindows"/>
</PermissionSet>

```

optional permission set:
Not specified

refused permission set:
Not specified

This is not the easiest format to read, but it contains all the information you need to configure your security policy correctly. Unfortunately, the output is not pure XML, and therefore creating a utility to parse the output and create a more readable report is not as straightforward as it could be.

Using the command `permview /decl MessageUtil.dll` extracts and displays *all* declarative security demands and stack walk overrides in addition to the permission requests. You will see the following information in addition to the permission request information we have already shown. Be aware that for large libraries the output from `Permview` may be significant:

```

Method MessageUtil::DisplayMessage( ) Assert permission set:
<PermissionSet class="System.Security.PermissionSet"
    version="1">
    <IPermission class="System.Security.Permissions.EnvironmentPermission, mscorl
ib, Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
        version="1"
        Read="Message"/>
    <IPermission class="System.Security.Permissions.UIPermission, mscorlib, Versi
on=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
        version="1"
        Window="SafeTopLevelWindows"/>
</PermissionSet>

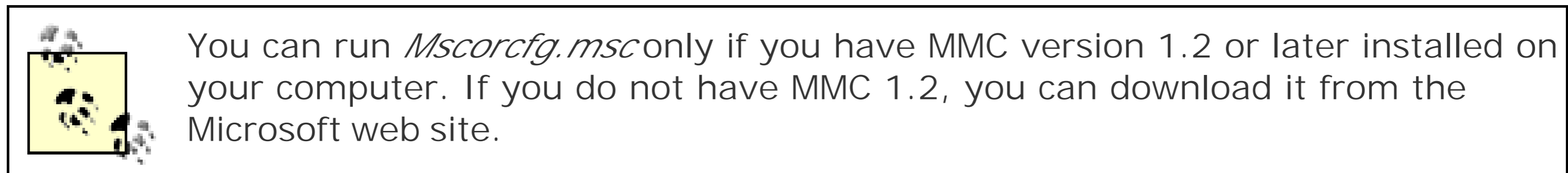
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

9.3 Using the .NET Framework Configuration Tool

The .NET Framework Configuration tool (*Mscorcfg.msc*) is a Microsoft Management Console (MMC) snap-in supplied with the .NET Framework that provides a graphical interface to manage various aspects of .NET configuration. We are concerned with the administration of security policy only and will not discuss the other features (refer to the .NET Framework documentation).



The steps to execute *Mscorcfg.msc* from the Windows user interface depend on the operating system you are using:

Windows 2000 Professional

Run Microsoft .NET Framework Configuration from the Administrative Tools group of the Control Panel.

Windows 2000 Server or Windows .NET Server

Click Start, Programs, Administrative Tools, and finally Microsoft .NET Framework Configuration

Windows XP Professional

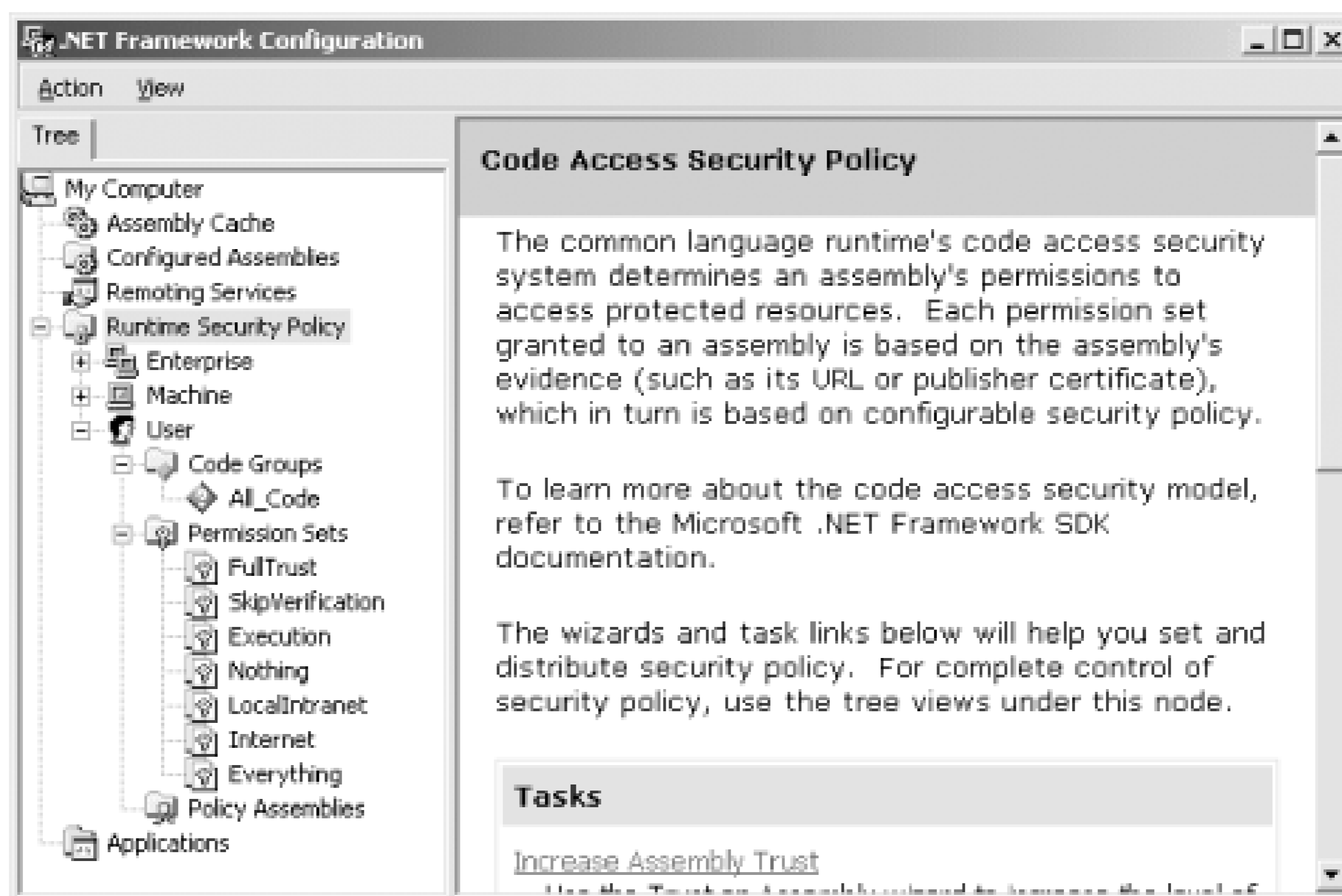
From the Control Panel, click Performance and Maintenance then Administrative Tools and run Microsoft .NET Framework Configuration

Regardless of the operating system you choose, you can start *Mscorcfg.msc* from the command line using the following command, where *version* is the version of the .NET framework, such as "v1.0.3705" for Version 1.0:

```
%Systemroot%\Microsoft.NET\Framework\version\Mscorcfg.msc
```

[Figure 9-2](#) shows the main window of *Mscorcfg.msc*. Expanding the Runtime Security Policy node of the console tree, you will see child nodes for each of the enterprise, machine, and user policy levels.

Figure 9-2. The Mscorcfg.msc main window



By expanding the node representing the policy level you want to administer, you can see three subtrees, one for each of the following policy-level elements we discussed in [Chapter 8](#):

- Fully trusted assemblies list
- Named permission sets
- Code group hierarchy

The following sections focus on the management of these three elements.

9.3.1 Managing Fully Trusted Assemblies

In [Chapter 8](#), we explained the requirement for policy levels to define a list of fully trusted assemblies. By right-clicking the Policy Assemblies node, as shown in [Figure 9-3](#), a shortcut menu provides you with the following options:

Add...

Allows you to add an assembly to the fully trusted assembly list, which we discuss further in [Section 9.3.1.1](#)

View

Allows you to control the righthand console pane to display either the current list of fully trusted assemblies (shown in [Figure 9-3](#)) or help information that describes the purpose of fully trusted assemblies

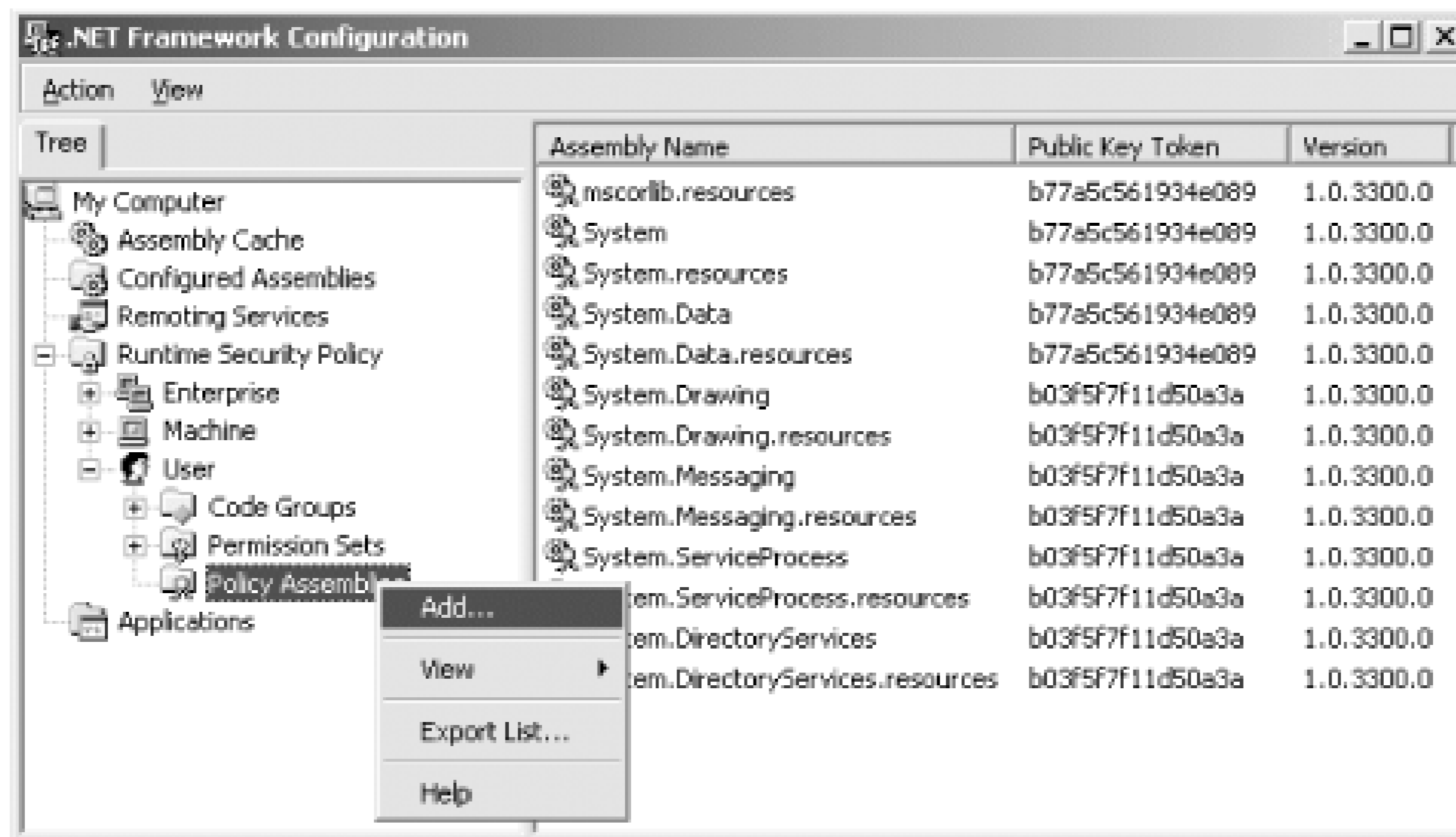
Export List...

Allows you to export the list of fully trusted assemblies to a text file, including version numbers and public key token values

Help

Displays MMC help information

Figure 9-3. Managing fully trusted assemblies

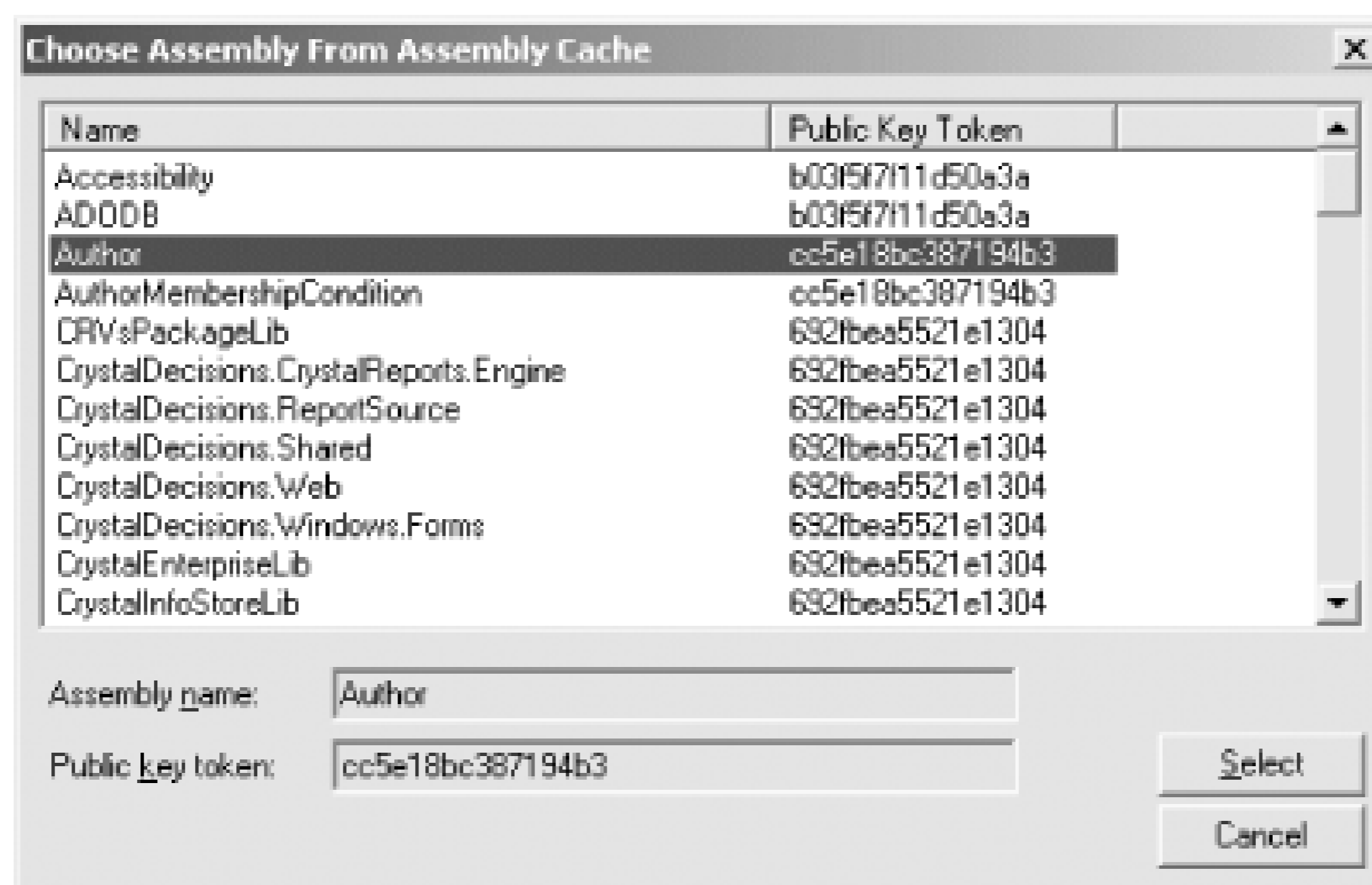


9.3.1.1 Adding a fully trusted assembly

Before you can add a new assembly to the fully trusted assemblies list, you must install the assembly into the global assembly cache. In [Chapter 2](#), we discussed the global assembly cache and how to add assemblies to it.

After you have installed the assembly in the global assembly cache, right-click the Policy Assemblies node of the console tree, as shown in [Figure 9-3](#) and select Add... from the shortcut menu. The Choose Assembly From Assembly Cache dialog box (shown in [Figure 9-4](#)) is displayed, listing all of the assemblies installed in the global assembly cache. Select the assembly you want to make fully trusted and click the Select button.

Figure 9-4. List of assemblies in the global assembly cache



You must make an assembly fully trusted in every policy level in which you will use the security classes that the assembly contains; see [Chapter 8](#) for more details.

9.3.1.2 Deleting a fully trusted assembly

To remove an assembly from the fully trusted assemblies list, make sure the righthand pane of the console displays the current list of fully trusted assemblies, as previously shown in [Figure 9-3](#). To configure this, right-click the Policy Assemblies node in the console tree and click View followed by Assemblies from the shortcut menus.

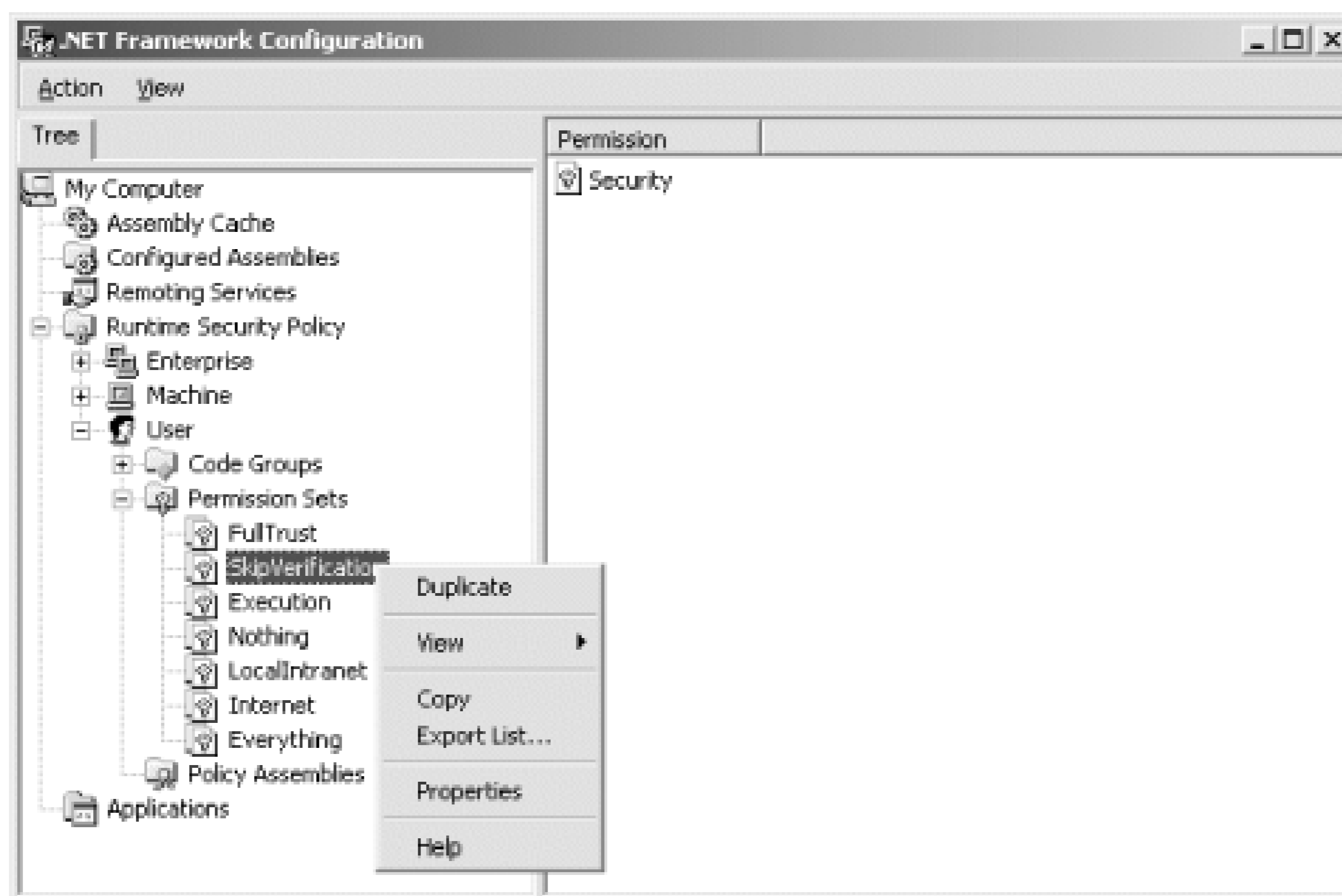
With the list displayed, right-click the assembly that you want to remove and select Delete from the shortcut menu that appears. You will be prompted to confirm the delete action; click the Yes button and the assembly is removed.

Security policy resolution will not function correctly if you remove an assembly containing classes on which the active policy configuration depends; see [Chapter 8](#) for details.

9.3.2 Managing Named Permission Sets

By expanding a policy level's Permission Sets node in the console tree, you will see the list of named permission sets defined in that policy level. [Figure 9-5](#) shows the list of named permission sets defined in the default user policy.

Figure 9-5. Managing named permission sets



As shown in [Figure 9-5](#), right-clicking any of the default permission set nodes displays a shortcut menu containing the following options:

Duplicate

Creates a copy of the selected permission set in the current policy level. For example, if you right-click the **Internet** permission set and select Duplicate, you create a new permission set named Copy of Internet in the user policy level.

View

Allows you to control the contents of the righthand console pane to display either the permissions contained in the selected permission set (shown in [Figure 9-5](#)) or help information that describes the permission set.

Properties

Displays a dialog box containing the name and description of the permission set. Because the default permission sets (except **Everything**, which we discuss shortly) are immutable, the dialog box does not allow you to edit the property values.

Copy

Copies the permission set to the clipboard, allowing you to make copies of the permission set in other policy levels using the clipboard's Paste facility.

Help

Displays MMC help information.

The shortcut menu for the default **Everything** permission set or any other non-default permission set includes these additional options:

Change Permissions...

Brings up a dialog box that allows you to change the permissions contained in the permission

set. The dialog boxes for editing the permission sets' content are the same dialog boxes used to create permission sets; we will discuss them in [Section 9.3.2.1](#).

Cut

Copies the permission set to the clipboard and deletes it from the current policy level. You cannot cut permission sets that are in use by code groups.

Delete

Allows you to delete the permission set as long as it is not in use by any code group.

Rename

Allows you to rename the permission set as long as it is not is in use by any code group.

Properties

Displays a dialog box that allows you to edit the name and description of the permission set. As with Rename, you cannot change the name of a permission set that is currently in use by a code group.

9.3.2.1 Creating named permission sets

To create a new named permission set, right-click Permission Sets node in the console tree and select New... from the shortcut menu.

The Create Permission Set dialog box, shown in [Figure 9-6](#), appears. You have two options for creating the new permission set: manual creation using the graphical interface or importing the definition of the permission set from an XML file.

Figure 9-6. Creating a new permission set

To import the named permission set from a file, select the Import a permission set from an XML file

radio button, enter the name of the file containing the definition, and press the Next button. The XML description of the permission set must have the structure shown in the following example, which describes a permission set named `TestSet` containing the unrestricted `FileIOPermission`, as well as the `Assertion` and `Execution` permissions from `SecurityPermission`:

```
<PermissionSet class="System.Security.NamedPermissionSet"
    version="1"
    Name="TestSet">
    <IPermission class="System.Security.Permissions.FileIOPermission, mscorlib,
Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
    version="1"
    Unrestricted="true"/>
    <IPermission class="System.Security.Permissions.SecurityPermission, mscorlib,
Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
    version="1"
    Flags="Assertion, Execution"/>
</PermissionSet>
```

The complexity of information in the XML description of a named permission set means that it is easier to create this information programmatically. This is easy to do by creating a `System.Security.NamedPermission` object, populating it with permission objects, and calling the `ToString` method on the `System.Security.SecurityElement` returned by the `ToXml` method. The following code demonstrates the creation of the `TestSet` named permission set (whose XML description we just listed) and displays it to the console:

```
# C#

// Create an empty NamedPermissionSet
NamedPermissionSet ps = new NamedPermissionSet("TestSet", PermissionState.None);

// Add an unrestricted FileIOPermission
ps.AddPermission(new FileIOPermission(PermissionState.Unrestricted));

// Add a SecurityPermission with Assertion and Execution permissions
ps.AddPermission(new SecurityPermission(SecurityPermissionFlag.Assertion |
    SecurityPermissionFlag.Execution));

// Display an XML version of the named permission set to the console.
Console.WriteLine(ps.ToXml().ToString());

# Visual Basic .NET

' Create an empty NamedPermissionSet
Dim ps As NamedPermissionSet = _
New NamedPermissionSet("TestSet", PermissionState.None)

' Add an unrestricted FileIOPermission
ps.AddPermission(New FileIOPermission(PermissionState.Unrestricted))

' Add a SecurityPermission with Assertion and Execution permissions
ps.AddPermission(New SecurityPermission(SecurityPermissionFlag.Assertion Or
SecurityPermissionFlag.Execution))
```

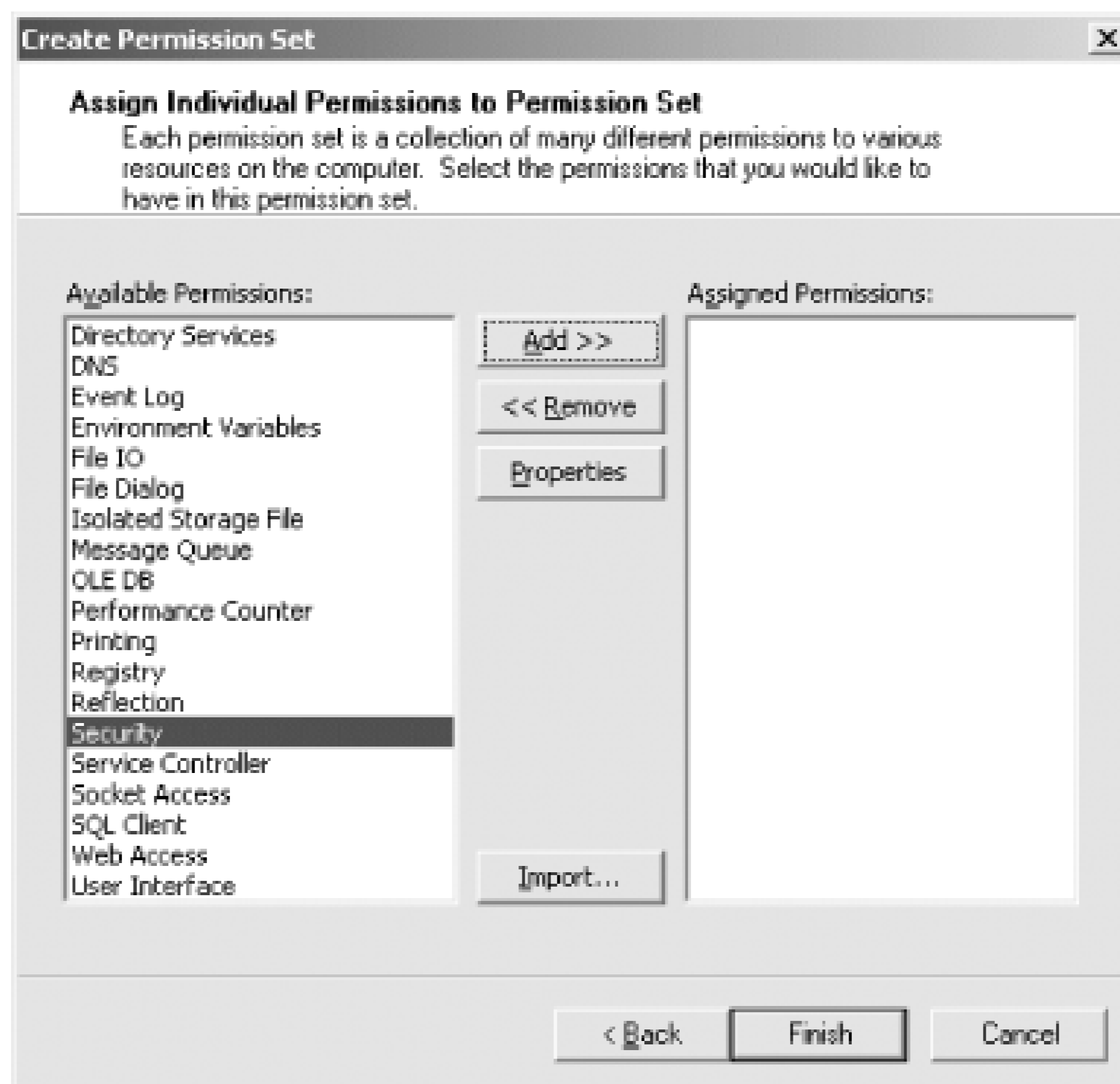
```
' Display an XML version of the named permission set to the console.
Console.WriteLine(ps.ToXml( ).ToString( ))
```



Imported permission set definitions can include entries for custom code-access permission classes, which we discussed in [Chapter 7](#). Before you import custom code-access permissions, make sure you have added the assembly that contains the custom class to the fully trusted assemblies list of the current policy level; see "Managing Fully Trusted Assemblies" earlier in this chapter for details.

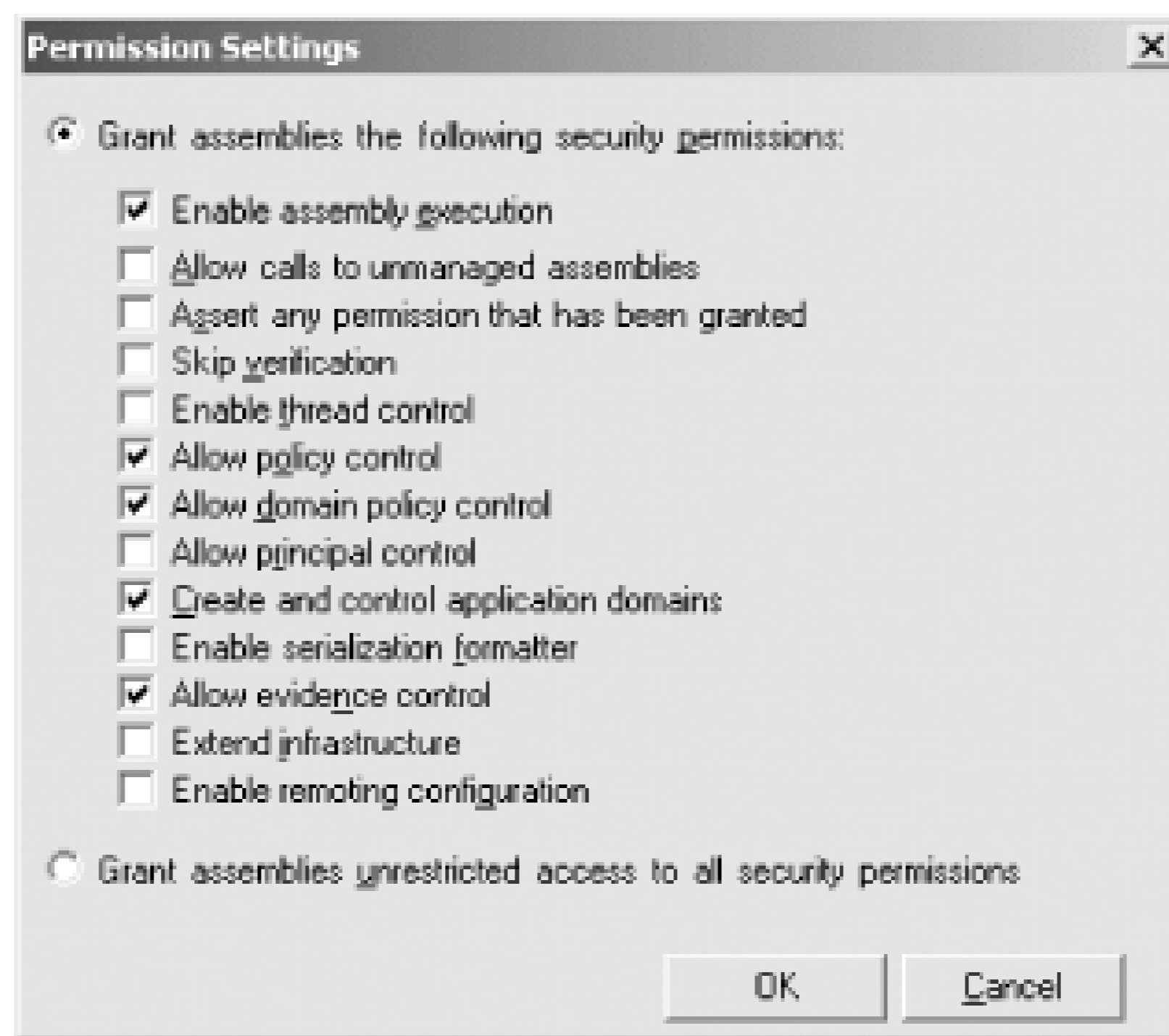
Instead of importing the named permission set, you can create it manually by selecting the Create a new permission set radio button. Enter a name and a description for the new permission set. In [Figure 9-6](#), we entered the name "CAS_Manipulation" because we are creating a permission set that includes all of the permissions necessary to manipulate CAS programmatically. Press the Next button to display the dialog box shown in [Figure 9-7](#), which allows you to select the set of permissions to include in the permission set.

Figure 9-7. Assigning permissions



Select the permission you want to include in your permission set from the Available Permissions list and click the Add button. You will see a Permission Settings dialog box. The contents of this dialog box depend on which permission you selected to add. [Figure 9-8](#) shows the Permission Settings dialog box for the Security permission, which represents the `System.Security.Permissions.SecurityPermission` class we described in [Chapter 7](#).

Figure 9-8. Configuring permissions



In [Figure 9-8](#), we select the individual elements of the `SecurityPermission` that we want to include in the new permission set. The permissions we have selected here map to the `Execution`, `ControlPolicy`, `ControlDomainPolicy`, `ControlAppDomain`, and `ControlEvidence` members of the `System.Security.Permissions.SecurityPermissionFlag` enumeration, which is also discussed in [Chapter 7](#). Pressing the OK button takes you back to the list of available permissions shown in [Figure 9-7](#). You can add more permissions or remove any that you have already added.

[Figure 9-7](#) also shows a button named Import... (which allows you to import an individual custom permission to your permission set). Pressing the Import... button displays a dialog box that allows you to select the file containing the XML description of the custom permission. The XML description of the permission must be of the appropriate structure to allow the runtime to create an instance of the permission using the permission class's `FromXml` method. The following example contains the XML description of the custom `RadioPermission` class that we developed in [Chapter 7](#). This specific example grants access to turn the radio on and off:

```
<IPermission class="OReilly.ProgrammingDotNetSecurity.RadioPermission,
Radio, Version=1.0.0.0, Culture=neutral, PublicKeyToken=cc5e18bc387194b3"
    version="1"
    StartStop="true"/>
```

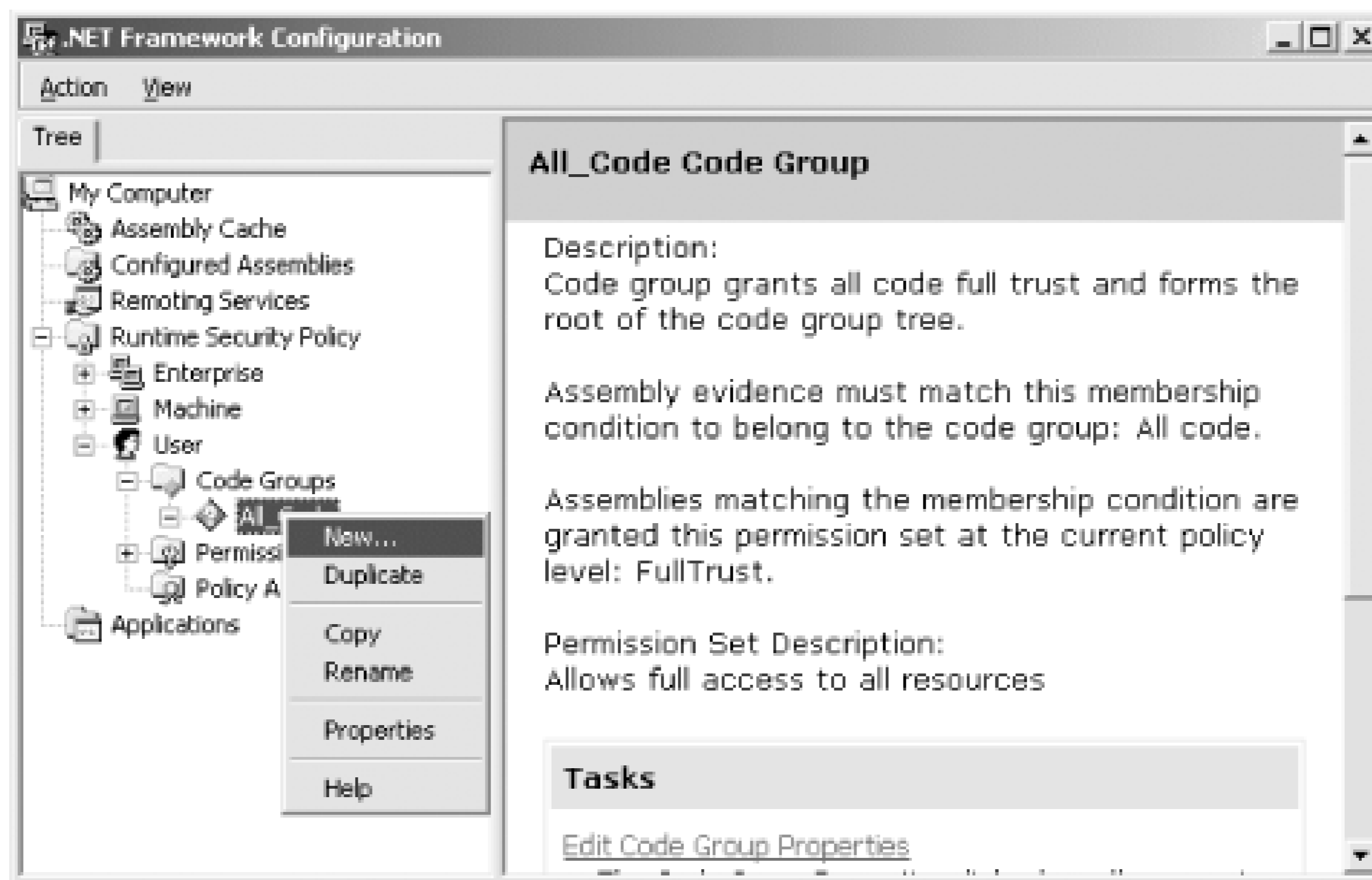
After configuring the contents of the new permission set, clicking Finish returns you to the main `Mscorcfg.msc` window.

Before you import custom code-access permissions, make sure you have added the assembly that contains the custom class to the fully trusted assemblies list of the current policy level; see "Managing Fully Trusted Assemblies" earlier in this chapter for details.

9.3.3 Managing Code Groups

By expanding a policy level's Code Groups node in the console tree, you will see the root code group for the policy level, as shown in [Figure 9-9](#).

Figure 9-9. Managing code groups



Code group names must be unique within a policy level.

Right-clicking the root code group brings up a shortcut menu with the following options:

New...

Allows you to create a new child code group, which we discuss in [Section 9.3.3.1](#).

Duplicate

Creates a deep copy (including all child code groups) of the selected code group and adds it as a child node to the current code group. For example, if you right-click on the `All_Code` group and select Duplicate, you create a new child group under `All_Code` named Copy of `All_Code`.

Copy

Copies the code group to the clipboard, allowing you to create copies of the code group as children of other code groups using the clipboard's Paste facility.

Paste

Pastes a previously copied code group as a child of the selected code group.

Rename

Allows you to edit the name of the code group.

Properties

Displays a dialog box that allows you to edit the configuration of the code group. With the exception of the code group's attributes, the property pages contain the same settings as the

dialog boxes used during the creation of the code group. We detail these settings in [Section 9.3.3.1](#).

The key use of the property pages is to configure the `Exclusive` and `LevelFinal` attributes for the code group, which we discussed in [Chapter 8](#). However, the interface does not use the attribute names and provides check boxes with the following human-readable interpretations from which to choose:

`Exclusive`

This policy level will only have the permissions from the permission set associated with this code group.

`LevelFinal`

Policy levels below this level will not be evaluated.

Help

Displays MMC help information.

By expanding the node representing the root code group, and the subsequently displayed child nodes, you can navigate through the code group hierarchy of the policy level. Right-clicking any code group node (except the root node) displays a shortcut menu the same as that we have already described, but with the following two additional options:

Cut

Copies the code group and its children to the clipboard and deletes it from the current policy level

Delete

Deletes the code group and its children.

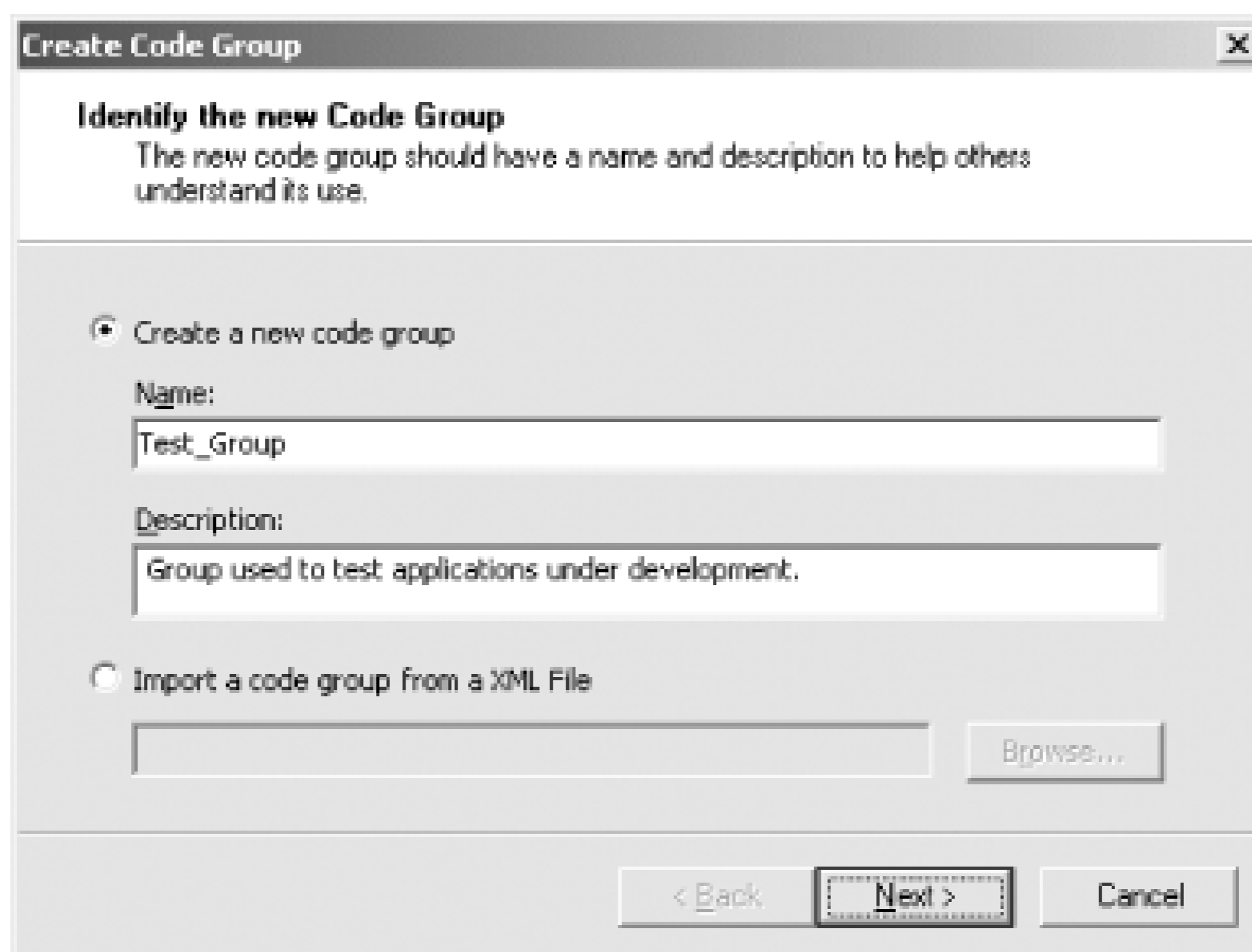
9.3.3.1 Creating code groups

To create a new code group, right-click the existing code group under which you want to create the new child code group and select `New...` from the shortcut menu.

You can only create code groups of type `System.Security.Policy.UnionCodeGroup` using the graphical interface of `Mscorcfg.msc`. To create code groups of other types, you must import the XML description of the code group. See [Chapter 8](#) for a description of the different types of code groups.

In the Create Code Group dialog box shown in [Figure 9-10](#), you have the option to create the new code group manually or to import an XML description of the code group from a file.

Figure 9-10. Identifying the new code group



To import a code group from a file select, the Import a code group from a XML file radio button, enter the name of the file containing its XML description, and press the Next button. The XML description of the code group must have the structure shown in the following example, which defines a code group with the configuration listed in [Table 9-3](#):

```
<CodeGroup class="System.Security.Policy.UnionCodeGroup, mscorlib, Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
  version="1"
  Attributes="Exclusive"
  Name="TestGroup"
  Description="A code group for testing">
  <IMembershipCondition class="System.Security.Policy.SiteMembershipCondition, mscorlib, Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
    version="1"
    Site="*.oreilly.com"/>
  <PermissionSet class="System.Security.NamedPermissionSet"
    version="1"
    Name="FileAccess">
    <IPermission class="System.Security.Permissions.FileIOPermission, mscorlib, Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
      version="1"
      Unrestricted="true"/>
  </PermissionSet>
</CodeGroup>
```

Table 9-3. Parameter values of the TestGroup code group

Parameter	Setting
Name	TestGroup

Parameter	Setting
Description	A code group for testing
Membership Condition	All code downloaded from a <code>Site</code> in the oreilly.com domain
Permission Set	A custom permission set named <code>FileAccess</code> , containing unrestricted access to the filesystem
Attributes	<code>Exclusive</code>

As with the permission set you imported in [Section 9.3.2.1](#), the complexity of a code group's XML description means it is easier and less error-prone to create. We detail the programmatic creation and configuration of code groups in [Chapter 8](#). As an example, here is the code that will generate the `TestGroup` code group we just described and print an XML representation of it to the console:

```
# C#

// Create the named permission set and add unrestricted file access.
NamedPermissionSet pset =
    new NamedPermissionSet("FileAccess",PermissionState.None);
pset.AddPermission(new FileIOPermission(PermissionState.Unrestricted));

// Create the policy statement and set the Exclusive attribute.
PolicyStatement pstate =
    new PolicyStatement(pset, PolicyStatementAttribute.Exclusive);

// Create the membership condition to match all "*.oreilly.com" sites.
IMembershipCondition mcon = new SiteMembershipCondition("*.oreilly.com");

// Create and configure the UnionCodeGroup
UnionCodeGroup cg = new UnionCodeGroup(mcon, pstate);
cg.Name = "TestGroup";
cg.Description = "A code group for testing";

// Display the CodeGroup to the console
Console.WriteLine(cg.ToXml( ).ToString( ));

# Visual Basic .NET

' Create the named permission set and add unrestricted file access.
Dim pset As NamedPermissionSet = _
    New NamedPermissionSet("FileAccess",PermissionState.None)
pset.AddPermission(New FileIOPermission(PermissionState.Unrestricted))

' Create the policy statement and set the Exclusive attribute.
Dim pstate As PolicyStatement = _
    New PolicyStatement(pset,PolicyStatementAttribute.Exclusive)

' Create the membership condition to match all "*.oreilly.com" sites.
Dim mcon As IMembershipCondition = _
    New SiteMembershipCondition("*.oreilly.com")
```

```
' Create and configure the UnionCodeGroup
Dim cg As UnionCodeGroup = New UnionCodeGroup(mcon,pstate)
cg.Name = "TestGroup"
cg.Description = "A code group for testing"

' Display the CodeGroup to the console
Console.WriteLine(cg.ToXml( ).ToString( ))
```



Imported code group definitions can include entries for custom code group types, membership conditions, and permissions. Before you import custom code groups, make sure you have added the assembly that contains any custom classes to the fully trusted assemblies list of the current policy level; see "Managing Fully Trusted Assemblies" earlier in this chapter for details.

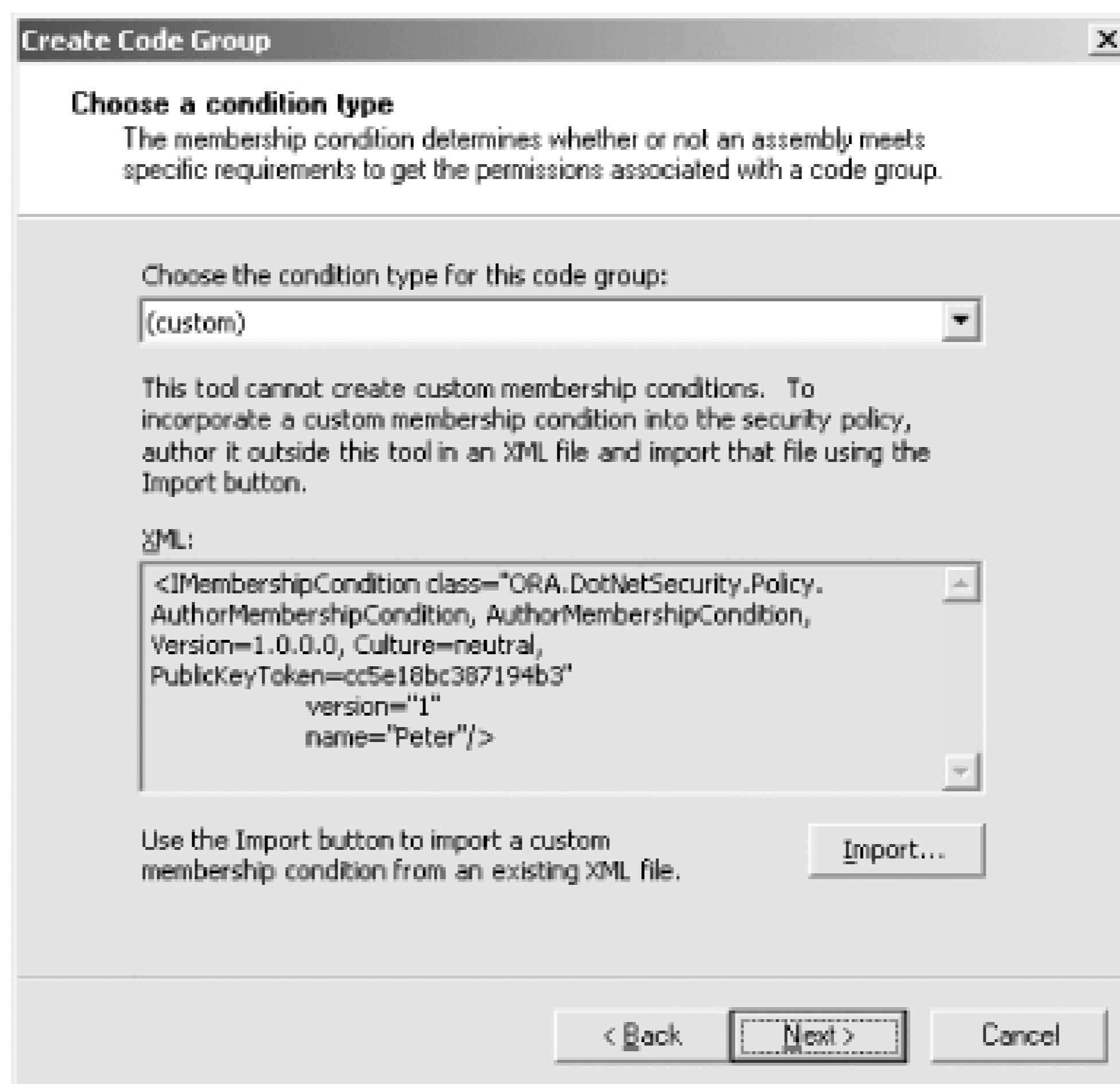
Instead of importing the code group, you can create it manually using a series of dialog boxes by selecting the Create a new code group radio button, and entering a name and a description for the new code group. In [Figure 9-10](#), we entered the name "Test_Group" to create a code group that we will use to grant permission to applications we are in the process of developing.

Press the Next button to display the dialog box shown in [Figure 9-11](#), which allows you to configure the membership condition for the new code group. The graphical interfaces allow you to base membership on any of the standard evidence types we discussed in [Chapter 6](#). [Figure 9-11](#) shows the use of `Url` evidence to match any code loaded from the `c:\development\cas_project` directory.

Figure 9-11. Specifying a membership condition

You can also choose to import a custom membership condition such as the `AuthorMembershipCondition` class that we developed in [Chapter 8](#). Selecting (custom) from the list of membership condition types causes the dialog box to change its appearance to that shown in [Figure 9-12](#). Pressing the Import button allows you to specify the file that contains an XML description of the custom membership condition. In [Figure 9-12](#), we have imported the XML description of an `AuthorMembershipCondition` that matches any code with `Author = Peter` evidence.

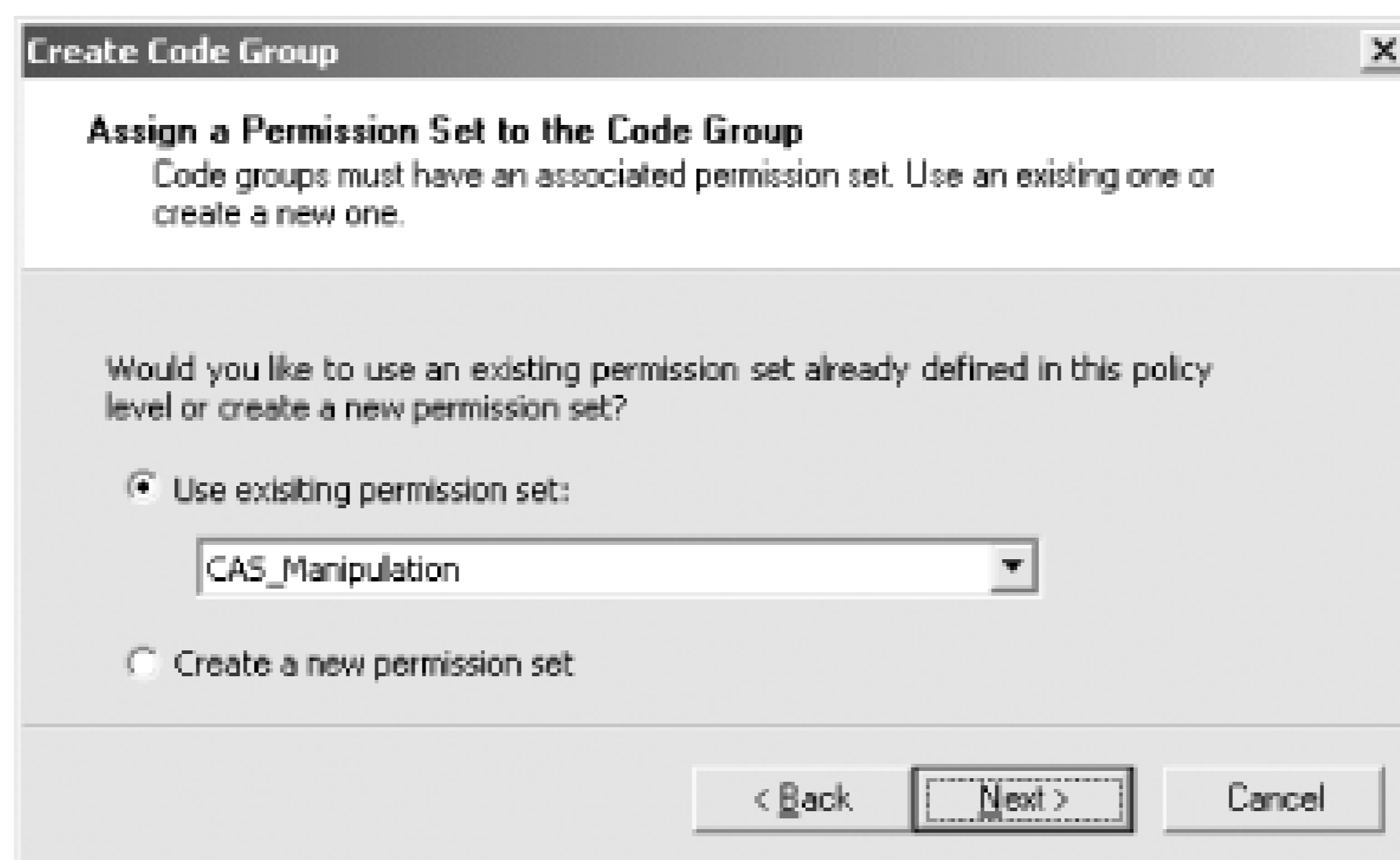
Figure 9-12. Importing a custom membership condition



Before you import custom membership conditions, make sure you have added the assembly that contains the custom class to the fully trusted assemblies list of the current policy level; see "Managing Fully Trusted Assemblies" earlier in this chapter for details.

After configuring the code group's membership condition, press the Next button to display the dialog box shown in [Figure 9-13](#), which allows you to select the permission set that the code group grants to its members. You can choose an existing named permission set from the current policy level, or you can choose to create a new code group. If you choose to create a new code group, you will begin the process we described earlier in [Section 9.3.2.1](#).

Figure 9-13. Selecting the code group's permission set



Selecting the named permission set for the code group and pressing the Next button completes the creation of the code group and returns you to the *Mscorcfg.msc* main window.

9.3.4 Other Security Policy Administration Options

The previous sections have shown you how to perform the security policy configuration tasks you will use most frequently during the development and testing process. Here we outline a number of other operations available through *Mscorcfg.msc* that you may find useful. By right-clicking the Runtime Security Policy node in the console tree, you will see the shortcut menu shown in [Figure 9-14](#).

Figure 9-14. The shortcut menu available through Mscorcfg.msc

The commands available to you are:

New...

Creates a new policy level based on the default security configuration described in [Section 9.1](#) earlier in this chapter. You can also specify the location where the policy file will be stored.

Open...

Opens a new policy level from a specified file.

Reset All...

Allows you to reset all security policy to the default configuration described in [Section 9.1](#) earlier in this chapter. You can also reset individual policy levels to the default settings by right clicking the individual policy level nodes in the console tree and selecting "Reset" from the shortcut menu.

Adjust Security...

Provides a simple interface through which to adjust the default machine security policy, which we discussed earlier in [Section 9.1](#). You can configure the level of trust you want to grant to each of the Internet Explorer zones, which modifies the named permission set assigned to that code group.

Evaluate Assembly...

Allows you to test an assembly to see what code groups the assembly is a member of and what permissions policy resolution would grant to the assembly.

Trust Assembly...

Allows you to define the minimum level of trust that will be assigned to an assembly.

Create Deployment Package...

Allows you to create a Windows Installer (*.msi*) file containing the definition of a security policy level from the current machine. Running this file on a target machine will configure the policy level of that machine. This simplifies the task of configuring security policy on a large number of machines, but .NET contains no mechanism to perform the distribution process. You can simply send the file to people that need it via email or use the normal software distribution processes used by your organization.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

9.4 Using the Code-Access Security Policy Tool

The Code Access Security Policy tool (*Caspol.exe*) is a utility provided with the .NET Framework that allows you to administer .NET's code-access security system from the command line and from batch files. You can run *Caspol.exe* from the following location, where *version* is the version of the .NET framework, such as "v1.0.3705" for Version 1.0:

```
%Systemroot%\Microsoft.NET\Framework\version\Caspol.exe
```

When performing some administrative tasks *Caspol.exe* commands can become long and confusing, but for many tasks, *Caspol.exe* provides a much quicker way of configuring and inspecting security policy, as long as you are happy to work from the command line. Before we look at how to configure security policy using *Caspol.exe*, there is one *Caspol.exe* configuration setting that you will want to change if you use *Caspol.exe* frequently.

Whenever you run a *Caspol.exe* command that changes the security policy configuration, before *Caspol.exe* commits the change to disk, it will prompt you to confirm the change with the following message:

```
The operation you are performing will alter security policy.  
Are you sure you want to perform this operation? (yes/no)
```

You must enter "y" or "yes" for the change to proceed. Although it serves a valuable purpose, you will find this level of protection frustrating if you use *Caspol.exe* frequently. To turn this prompt on and off use the following statements, but remember that if you turn it off, any changes you make will be committed automatically:

```
caspol -polchgprompt on  
caspol -polchgprompt off
```

Each *Caspol.exe* command has a short version as well as the full version we use throughout this section. For example, the short version of the `polchgprompt` flag we just used is `-pp`. Consult the .NET SDK Documentation for the short versions of each command.

9.4.1 Controlling the Security System

Not only does *Caspol.exe* allow you to manage security policy, but it also allows you to manage some higher-level configuration settings of the security system. These features map directly to methods of the `System.Security.SecurityManager` class, which we discussed in [Chapter 8](#).

9.4.1.1 Switching code-access security on and off

To switch code-access security on and off, use the `-security` flag, as shown here:


```
caspol -security on
caspol -security off
```

Turning off code-access security bypasses the security system, meaning that all code has full permission to perform any action. You should consider this only after ensuring that other security mechanisms, such as operating system security, are in place and properly configured to protect your system and resources.



The `-security` flag does not control the enforcement of role-based security checks, which we discuss in [Chapter 10](#).

Use of the `-security` flag maps directly to the static `SecurityManager.SecurityEnabled` property we discussed in [Chapter 8](#); it is protected by the `SecurityPermission.ControlPolicy` permission.

9.4.1.2 Switching execution permission checks on and off

Normally, the .NET Framework runtime checks to see if an assembly has the `SecurityPermission.Execution` permission before it will load and run the assembly. You can switch execution permission checking on and off using the `-execution` flag shown here:

```
caspol -execution on
caspol -execution off
```

There are performance benefits from turning off execution checking; your assembly will load and start more quickly, but you must weigh those savings against the decreased security. Once an assembly is running, there are many undesirable things that malicious code can do that do not require permissions, such as making heavy use of your CPU. Also' once loaded, malicious code is free to probe the security you have in place to try to identify any weaknesses that exist.

Use of the `-execution` flag maps directly to the static `SecurityManager.CheckExecutionRights` property we discussed in [Chapter 8](#); it is protected by the `SecurityPermission.ControlPolicy` permission.

9.4.2 Administering Policy Levels

With respect to the administration of security policy, the functionality of *Caspol.exe* is much the same as what we described for *Mscorfg.msc* in [Section 9.3](#). In the following sections, we describe how to manage each of the key elements of security policy, but first we must describe how to target your *Caspol.exe* commands to affect the appropriate policy level and code group.

9.4.2.1 Specifying the target policy level

You must target each policy-administration command at the policy levels you want it to affect using one of the following flags:

```
-all
```

Affects the enterprise and machine policies as well as the user policy for the currently logged-on user

`-enterprise`

Affects only the enterprise policy

`-machine`

Affects only the machine policy

`-user`

Affects only the user policy for the currently logged on user

`-customuser path`

Affects only the user policy contained in the security policy file located at the specified path

`-customall path`

Affects the enterprise and machine policies as well as the user policy contained in the security policy file located at the specified path

Not every target value is valid for every *Caspol.exe* command. In addition, if you do not specify a target policy level, the result will depend on the command you execute; some commands assume a default `-machine` setting, others use `-all`. We describe the default behavior when we discuss the individual commands.

9.4.2.2 Specifying the target code group

Some *Caspol.exe* commands require you to specify an existing code group as the target of the command. For example, you must identify the code group you want to delete, or the code group under which you want to add a new child code group. *Caspol.exe* provides two ways in which you can uniquely reference a code group.

First, because code group names must be unique within their policy levels, you can use the name of the code group. For example, when manipulating code groups from the default machine policy that we described in [Section 9.1](#) earlier, you could use names like `All_Code`, `Trusted_Zone`, or `Microsoft_Strong_Name` to identify the target of your command.

Second, you can use a numeric label that uniquely represents the position of the target code group within the code group hierarchy of a policy group. The root code group has a label of "1.", and every code group below it has a label that identifies its position among its peers—for example, "1.1.", "1.2.", "1.3.", and so on. The labels extend to accommodate all the levels in the code group hierarchy—for example, "1.1.", "1.1.1.", "1.1.1.1.", and so on. You must always include the final decimal point when specifying the label of a code group.

9.4.2.3 Managing fully trusted assemblies

To list the fully trusted assemblies in a policy level use the `-listfulltrust` flag. The default target for `-listfulltrust` is `-machine`, but you can specify any of the target values; we show some examples here:

```
caspol -listfulltrust
caspol -all -listfulltrust
caspol -customuser "c:\someuser\security.config" -listfulltrust
```

The start of the output from these commands will look similar to the following. The output shows some general *Caspol.exe* settings, identifies the target policy level, and then lists the fully trusted assemblies, along with strong-name information.

```
Microsoft (R) .NET Framework CasPol 1.0.3705.288
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
```

```
Security is ON
Execution checking is ON
Policy change prompt is ON
```

```
Level = Machine
```

```
Full Trust Assemblies:
```

```
1. mscorlib.resources 1.0.3300.0 =
StrongName - 00000000000000004000000000000000 name = mscorlib.resources version
= 1.0.3300.0
2. System 1.0.3300.0 =
StrongName - 00000000000000004000000000000000 name = System version = 1.0.3300.0
```

To add an assembly to the fully trusted list of a policy level, use the `-addfulltrust` flag. You cannot specify `-all` or `-customall` as the target, and if you do not specify any target, `-machine` is used. You must specify the name and path of the assembly that you want to add to the fully trusted list, as shown in the following examples:

```
caspol -addfulltrust SomeAssembly.dll
caspol -user -addfulltrust C:\Development\SomeOtherAssembly.dll
```

The assembly specified in the `-addfulltrust` command must already be installed in the global assembly cache, or *Caspol.exe* will throw an error; see [Chapter 2](#) for details.

To remove an assembly from the fully trusted assembly list, use the `-remfulltrust` flag and specify the name of the assembly to remove, as shown here; note that you do not need to include any path information. Again, the default target is `-machine`, and you cannot specify `-all` or `-customall` as the target of the command:

```
caspol -remfulltrust SomeAssembly.dll
caspol -user -remfulltrust SomeOtherAssembly.dll
```

Security policy resolution will not function correctly if you remove an assembly containing classes that the active policy configuration depends on; see [Chapter 8](#) for an explanation of why assemblies need to be fully trusted.

9.4.2.4 Managing named permission sets

To list the named permission sets currently defined in a policy level, use the `-listpset` flag. The default target for `-listpset` is `-machine`, but you can specify any target, as shown in these examples:

```
caspol -listpset
caspol -user -listpset
caspol -customall "c:\someuser\security.config" -listpset
```

The start of the output from these commands will look something like this, showing the XML representation of every code group defined in the policy level. This information is long and difficult to read in this format, but you cannot get *Caspol.exe* to list the definition of a specific permission set:

```
Microsoft (R) .NET Framework CasPol 1.0.3705.288
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
```

```
Security is ON
Execution checking is ON
Policy change prompt is ON
```

```
Level = Machine
```

```
Named Permission Sets:
```

```
1. FullTrust (Allows full access to all resources) =
<PermissionSet class="System.Security.NamedPermissionSet"
  version="1"
  Unrestricted="true"
  Name="FullTrust"
  Description="Allows full access to all resources"/>

2. SkipVerification (Grants right to bypass the verification) =
<PermissionSet class="System.Security.NamedPermissionSet"
  version="1"
  Name="SkipVerification"
  Description="Grants right to bypass the verification">
  <IPermission class="System.Security.Permissions.SecurityPermission, mscorlib,
    Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
    version="1"
    Flags="SkipVerification"/>
</PermissionSet>
```

To add a new named permission set to a policy level, use the `-addpset` flag. You must specify the name of the file that contains the XML description of the permission set to add. This is equivalent to importing the permission set using *Mscorcfg.msc*, which we discussed earlier in [Section 9.3.2.1](#), and the XML format is the same. The following statements demonstrate how to create permission sets from an XML description contained in the file *pset.xml*. You cannot use `-all` or `-customall` as the target of the command, and the default is `-machine`:

```
caspol -addpset pset.xml
caspol -enterprise -addpset pset.xml
```

If the XML description of the permission set does not include a `Name` attribute, *Caspol.exe* will throw

an error and refuse to create the permission set. In this case, you must specify the name of the permission set as an argument to the `-addpset` command, as shown here where we name the permission set "TestSet":

```
caspol -addpset pset.xml TestSet
caspol -enterprise -addpset pset.xml TestSet
```

You can change an existing permission set using the `-chgpset` flag and specifying the name of the existing permission set to change and the name of the file containing the new XML description for the permission set. The `-chgpset` command simply overwrites the existing permission set definition with the new one. For example, to change the TestSet permission set that we just created using the contents of a file named *newpset.xml*, we use these commands:

```
caspol -chgpset newpset.xml TestSet
caspol -enterprise -chgpset newpset.xml TestSet
```

To remove a named permission set, use the `-rempset` flag and specify the name of the permission set to remove. Here we show how to remove the TestSet permission set that we just created. Again, the default target is `-machine`, and you cannot specify `-all` or `-customall` as the target of the command:

```
caspol -rempset TestSet
caspol -enterprise TestSet
```

Caspol.exe will not remove permission sets that are in use by code groups.

9.4.2.5 Managing code groups

To list the code groups in a policy level, use the `-listgroups` flag. The default target for `-listgroups` is `-machine`, but you can specify any of the target settings, as demonstrated in the following examples:

```
caspol -listgroups
caspol -user -listgroups
caspol -customuser "c:\someuser\security.config" -listgroups
```

When executed against the default machine policy, which we described in [Section 9.1](#), the output of the `-listgroups` command looks like this; the numeric label we discussed in [Section 9.4.2.2](#) precedes the membership condition, and the name of the permission set each code group grants:

```
Microsoft (R) .NET Framework CasPol 1.0.3705.288
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
```

```
Security is ON
Execution checking is ON
Policy change prompt is ON
```

```
Level = Machine
```

Code Groups:

1. All code: Nothing
 - 1.1. Zone - MyComputer: FullTrust
 - 1.1.1. StrongName - 00240000048000009400000006020000002400005253413100040000100010007D1FA57C4AED9F0A32E84AA0FAEFD0DE9E8FD6AEC8F87FB03766C834C99921EB23BE79AD9D5DCC1DD9AD236132102900B723CF980957FC4E177108FC607774F29E8320E92EA05ECE4E821C0A5EFE8F1645C4C0C93C1AB99285D622CAA652C1DFAD63D745D6F2DE5F17E5EAF0FC4963D261C8A12436518206DC093344D5AD293: FullTrust
 - 1.1.2. StrongName - 00000000000000000400000000000000: FullTrust
 - 1.2. Zone - Intranet: LocalIntranet
 - 1.2.1. All code: Same site Web.
 - 1.2.2. All code: Same directory FileIO - Read, PathDiscovery
 - 1.3. Zone - Internet: Nothing
 - 1.4. Zone - Untrusted: Nothing
 - 1.5. Zone - Trusted: Internet
 - 1.5.1. All code: Same site Web.

Success

To add a new code group to a policy level, use the `addgroup` flag. Because of the complexity of code group configuration, the `addgroup` command is the most complex *Caspol.exe* statement, and takes multiple arguments with many optional values. The general form of the `addgroup` command is shown here:

```
caspol -target -addgroup {label|name} membership pset [flags]
```

The `target` argument specified the target policy level, which defaults to `-machine`, and cannot be `-all` or `-customall`. The `label` and `name` arguments allow you to specify the parent of the new code group using either its numeric label or name, as we discuss earlier in [Section 9.4.2.2](#). The `membership`, `pset`, and `flags` arguments specify the membership condition, permission set, and properties of the code group respectively. We summarize the possible values of the `membership` argument in [Table 9-4](#), and those of the `flags` argument in [Table 9-5](#). The types of evidence listed in [Table 9-4](#) are discussed in [Chapter 6](#), and we provide a more complete discussion of membership conditions in [Chapter 8](#).

Table 9-4. Possible values of the membership argument

Value	Description
-all	Specifies that all code is a member of the code group.
-appdir	Looks for both <code>ApplicationDirectory</code> and <code>Url</code> evidence and tests to see if the location specified in the <code>Url</code> evidence represents a location that is a child of the location specified in the <code>ApplicationDirectory</code> evidence.
-custom xmlfile	Specifies the use of a custom membership condition to determine membership of the code group. The <code>xmlfile</code> argument specifies the name of the file containing the XML description of the custom membership condition. See Chapter 8 for details on creating custom membership conditions.

Value	Description
-hash hashAlg { -hex value -file assembly }	Specifies that all code with Hash evidence of a specified value is a member of the code group. The hashAlg specifies the hashing algorithm to use, and the hex and file arguments provide different ways of providing the hash value; see the .NET Framework SDK documentation for details.
-pub { -cert file -file file -hex string }	Specifies that all code with the specified Publisher evidence is a member of the code group. The cert , file , and hex arguments provide different ways of specifying the publisher certificate to test for; see the .NET Framework SDK documentation for details.
-site sitename	Specifies that all code with the Site evidence specified by the sitename argument is a member of the code group.
-strong -file file_name name version	Specifies that all code with the specified StrongName evidence is a member of the code group. The file_name , name , and version arguments specify the component parts of the strong name; see the .NET Framework SDK documentation for details.
-url URL	Specifies that all code with the Url evidence specified by the URL argument is a member of the code group.
-zone zonename	Specifies that all code with the Zone evidence specified by the zonename argument is a member of the code group.

Table 9-5. Possible values of the flags argument

Value	Description
-description "description"	Allows you to specify the description for the code group, which you must enclose in double quotes.
-exclusive { on off }	Allows you to set the code group's Exclusive attribute on or off, see Chapter 8 for a description of this attribute.
-levelfinal { on off }	Allows you to set the code group's LevelFinal attribute on or off, see Chapter 8 for a description of this attribute.
-name "name"	Allows you to specify the name of the code group, which you must enclose in double quotes, and which must be unique within the target policy level.

The following is an example of the **-addgroup** command that we used in [Chapter 8](#) to create a code group using the custom **AuthorMembershipCondition** class:

```
caspol -user -addgroup All_Code -custom Peter.xml FullTrust
-name "Peter's code" -description "Code group grants all
code written by Peter full trust"
```

The command to change an existing code group is also complex but uses the same arguments as the **-addgroup** command, with which you are now familiar. The general structure of a **chggroup** command is as follows:

```
caspol -target -chgggroup {label|name} {membership|pset|flags}
```

In the following example, we change the name of the machine-level code group with label "1.5." to "SomeTestGroup":

```
caspol -enterprise -chgggroup 1.5. -name SomeTestGroup
```

Here, we target a machine-level code group named "MyCode" to change its membership condition and its permission set, and make the code group **Exclusive**; notice that we do not specify the target level because **-machine** is the default:

```
caspol -chgggroup MyCode -url "file:///c:\dev\*" FullTrust -exclusive on
```

To remove a code group, use the **-remgroup** flag and specify the code group to remove. The examples here show how to remove a code group in the enterprise level using both its name and numeric label. Again, the default target is **-machine**, and you cannot specify **-all** or **-customall** as the target of the command:

```
caspol -enterprise -remgroup 1.5.
caspol -enterprise -remgroup SomeTestGroup
```

9.4.3 Evaluating Security Policy

Caspol.exe also allows you to test an assembly to determine which code groups it is a member of and, consequentially, the runtime would grant to it when loaded. To determine which groups an assembly is a member of, use the **-resolvegroup** flag and specify the assembly name as follows:

```
caspol -resolvegroup HelloWorld.exe
```

Depending on the evidence of the assembly and the configuration of security policy, the output will look similar to this:

```
Microsoft (R) .NET Framework CasPol 1.0.3705.288
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
```

```
Level = Enterprise
```

```
Code Groups:
```

1. All code: FullTrust
 - 1.1. Url - file:///C:/development/*: FullTrust (Exclusive)
 - 1.2. All code: FullTrust
 - 1.2.1. All code: Internet

```
Level = Machine
```

```
Code Groups:
```

1. All code: Nothing
 - 1.1. Zone - MyComputer: FullTrust


```
Level = User
```

```
Code Groups:
```

1. All code: Nothing
 - 1.1. Zone - MyComputer: FullTrust

```
Success
```

To view the permission granted to an assembly, use the `-resolveperm` flag, as shown here:

```
caspol -resolveperm HelloWorld.exe
```

The `-resolveperm` command will display the XML description of the permission set granted to the assembly, similar to that shown here. Notice that the permission set also lists the identity permissions granted to the assembly based on the evidence it presented, as well as the permissions granted by the security policy:

```
Microsoft (R) .NET Framework CasPol 1.0.3705.288
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
```

```
Resolving permissions for level = Enterprise
Resolving permissions for level = Machine
Resolving permissions for level = User
```

```
Grant =
```

```
<PermissionSet class="System.Security.PermissionSet"
  version="1"
  Unrestricted="true">
  <IPermission class="System.Security.Permissions.UrlIdentityPermission, mscorl
ib, Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
  version="1"
  Url="file:///C:/Development/HelloWorld.exe"/>
  <IPermission class="System.Security.Permissions.ZoneIdentityPermission, mscor
lib, Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
  version="1"
  Zone="MyComputer"/>
</PermissionSet>
```

```
Success
```

Both the `-resolvegroup` and `-resolveperm` commands allow you to specify a target policy level, in which case the assembly is only evaluated against that policy level. If you do not specify a policy level, the default target `-all` is used.

9.4.4 Forcing Security Changes

Caspol.exe is a managed application that relies on obtaining a high level of trust from the runtime in order to perform management of the security system. When you execute any of the commands we

discussed in the previous sections, *Caspol.exe* will test to see if the changes will result in *Caspol.exe* being unable to run correctly in the future; *Caspol.exe* will refuse to make such changes and display the following message:

```
Microsoft (R) .NET Framework CasPol 1.0.3705.288  
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
```

This operation will make some or all caspol functionality cease to work. If you are sure you want to do this operation, use the '-force' option before the option you just executed. For example:

```
caspol -force -machine -remgroup 1.6
```

Policy save aborted.

If you want to force the change despite this warning, you must use the `force` flag. The following command makes the `All_Code` code group of the machine policy exclusive. In the default security policy, this will stop all code from running:

```
caspol -force -machine -chggroup All_Code -exclusive on
```

9.4.5 Resetting Security Policy

A feature of *Caspol.exe* you will find useful during development is the ability to reset the security policy to its default value. When developing software that manipulates security policy directly, it is not difficult to break security policy to a point where your code will not run. To reset policy levels, use the `-reset` flag and specify the levels you want to reset:

```
caspol -user -reset  
caspol -all -reset
```

You can also undo the last change you made with *Caspol.exe* using the `-recover` flag, as shown here:

```
caspol -machine -recover  
caspol -all -recover
```

If your security policy is broken to such a degree that *Caspol.exe* will not run, the final resort is to delete the security policy files to force the runtime to create new ones with default settings, as we described in [Section 9.1.1](#).

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 10. Role-Based Security

The most significant security enhancement added to the .NET Framework is code-access security, which we discussed in the previous chapters. However, at times it is still necessary to make security decisions based on the user running the code, as opposed to the identity of the code itself. For this purpose, the .NET Framework supports a simple role-based security model. In this chapter, we explain what role-based security is and discuss the .NET Framework's implementation. We describe the classes used to access role-based security and demonstrate how to use them in your programs.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

10.1 Role-Based Security Explained

Role-based security (RBS) is a common security model in contemporary computing. When users wish to access a computer system, they must first prove their identity—a process known as *authentication*. Authentication requires the user to provide a set of *credentials* that uniquely identify him. These credentials are commonly a name and password but could be a physical token, such as a key card, or a biological attribute, such as a thumbprint. The computer system consults an *authority* to determine if the supplied credentials represent a known user and whether that user should have access to the system. During operation, the system relies on the user's authenticated identity when performing *authorization*—the process of determining what actions and resources a user has authority to access. A person's authority is expressed in terms of *roles*. A role is a logical categorization that grants members of the role specific permissions.

To use an example with which you should be familiar, the Windows NT, Windows 2000, and Windows XP user account systems provide a form of RBS. Each user authenticates with Windows by providing a username and password that must match a predefined user account; to the Windows operating system, the user account represents the user's identity. The Windows groups that the user account is a member of represent the user's roles and determine the resources and operations to which the user has access.

The most common types of roles mirror the jobs and activities we perform in everyday life, such as teller, manager, or administrator, and grant their members the authority to perform the actions and access the resources that they need to carry out their daily duties. However, roles are an abstract concept that you can use to represent any kind of grouping to which you want to assign a set of permissions (for example, members of a certain project that need access to a special color printer).

10.1.1 .NET Role-Based Security Explained

.NET provides a general-purpose RBS abstraction that you can use in conjunction with a variety of authentication and authorization mechanisms. In .NET RBS, an *identity* represents the user on whose behalf code is running. Normally, this is the currently logged-on Windows user, but this is not always the case. If your application authenticates users against an authority other than the Windows account system, then the identity may be different to the currently logged-on Windows user.

A *principal* encapsulates an identity and the roles to which the identity belongs; we illustrate this relationship in [Figure 10-1](#). If the identity represents a Windows user account, the roles will identify the Windows groups to which the user belongs. Because a principal encapsulates both the identity and roles of a user, it is the primary reference point on which the .NET runtime bases any role-based security decisions.

Figure 10-1. The relationship between principals, identities, and roles



.NET role-based security is independent of the role-based security mechanism provided by COM+. See [Chapter 19](#) for a discussion of how to use COM+ security.

10.1.2 Comparing .NET Role-Based Security and Windows Security

.NET RBS makes it easy to integrate with the Windows user accounts system, but .NET RBS is not just an extension of Windows user security into the .NET runtime. .NET RBS and the Windows user account system are completely independent security mechanisms that operate concurrently and serve different purposes.

Windows security protects the integrity of the entire system by controlling user access to important operating system operations and resources. Every Windows process and thread has an associated *access token* that represents the user on whose behalf the process or thread is running. The access token is an operating system object that contains information about the permissions and privileges of the user it represents, and the groups to which the user belongs. Before Windows allows code to perform a protected operation or access a protected resource, it evaluates the thread's access token to ensure that it contains the permission necessary to perform the operation. Windows enforces security regardless of whether an action is initiated by a managed .NET application or native code.

.NET RBS operates at the application level, providing a convenient mechanism through which you can control access to functionality based on the user's identity and roles. For example, you may only allow users who are members of certain roles to call an important method or you may make different menu items visible depending on the roles of the current user. Each thread running .NET code has a principal associated with it; the principal serves a similar purpose to the Windows access token, allowing .NET to determine if the user on whose behalf a thread is running is a member of a specific role. Although it is common for both the principal and access token of a thread to represent the same user, they are independent objects and there is no fixed relationship between the two. The use of .NET RBS is entirely optional; you are responsible for deciding what is protected by .NET RBS, and you must express these requirements in your code.

A .NET application can change the Windows access token of the current operating system thread in order to impersonate another Windows user—see [Section 10.2.3.2](#) later in this chapter for details.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

10.2 Programming Role-Based Security

The .NET runtime enforces role-based security using techniques and syntax similar to those we described in [Chapter 7](#) for code-access security. In your applications, you protect functionality by making role-based security demands that specify the identity or role that the thread's principal must contain. If the thread's principal does not contain the demanded identity and role, then the demand causes an exception.

10.2.1 Introducing the `IIdentity` and `IPrincipal` Interfaces

The `System.Security.Principal` namespace includes the `IIdentity` and `IPrincipal` interfaces to represent identities and principals. By using interfaces to represent identities and principals, .NET provides flexibility, which means that it is relatively easy to create concrete role-based security implementations to support many different authentication and authorization mechanisms. The .NET class library contains four concrete RBS implementations that use the `IIdentity` and `IPrincipal` interfaces:

Forms

Provides a role-based authentication mechanism for use in ASP.NET applications. Forms authentication only provides an implementation of `IIdentity`; we discuss Forms authentication in [Chapter 18](#).

Generic

Provides a generic role-based security implementation that is independent of any specific authentication and authorization mechanism. See [Section 10.2.5](#) for details.

Passport

Provides a role-based authentication mechanism that relies on the Microsoft Passport .NET web-based service to authenticate users. Passport authentication only provides an implementation of `IIdentity`. A discussion of Passport authentication is beyond the scope of this book, and we refer you to the .NET Framework SDK and Passport SDK documentation for further details.

Windows

Provides a role-based security implementation that is based on the users defined in the Windows user account system. See [Section 10.2.3](#) for details.

We describe the members of the `IIdentity` and `IPrincipal` interfaces in [Table 10-1](#). In later sections, when we discuss the Windows and Generic implementations of these interfaces, we highlight any implementation-specific behavior.

Table 10-1. Members of the `IIdentity` and `IPrincipal` interfaces

Member	Description
IIdentity interface	
<code>AuthenticationType</code>	Property that returns a string specifying the type of authentication used to identify the user represented by the <code>IIdentity</code> object
<code>IsAuthenticated</code>	Property that returns true if the user represented by the <code>IIdentity</code> has been authenticated; otherwise it returns false
<code>Name</code>	Property that returns a string containing the name of the user represented by the <code>IIdentity</code> object
IPrincipal interface	
<code>Identity</code>	Property that returns the <code>IIdentity</code> object contained in the <code>IPrincipal</code> object
<code>IsInRole</code>	Method that returns true if the <code>IIdentity</code> contained in the <code>IPrincipal</code> is a member of the role with the specified name; otherwise, it returns false



The `IPrincipal` interface makes no provision for enumerating or accessing all of a principal's roles. The only access provided is to test individual role names against the principal's role set using the `IsInRole` method.

10.2.2 Determining the Current Principal

Every thread executing in the .NET runtime has an `IPrincipal` object associated with it. The thread's `IPrincipal` object represents the user on whose behalf the thread is running, and allows the runtime to make role-based security decisions for the thread based on the user's identity and roles. However, many applications do not use the RBS features of .NET, and so in the interest of performance and conserving resources, the .NET runtime does not automatically assign an `IPrincipal` object to every thread. If you intend to use RBS, you must either assign an `IPrincipal` to a thread manually or configure the runtime to create one automatically the first time it is needed.

You can use the `System.AppDomain.SetThreadPrincipal` method shown in the following code to specify an `IPrincipal` object that the runtime will automatically assign to each thread running in the application domain. You can call the `SetThreadPrincipal` method only once on each application domain; otherwise, an instance of `System.Security.Policy.PolicyException` is thrown:

```
# C#

public void SetThreadPrincipal(
    IPrincipal principal
);

# Visual Basic .NET

NotOverridable Public Sub SetThreadPrincipal( _
    ByVal principal As IPrincipal _
)
```

Instead of using `SetThreadPrincipal` to set a default `IPrincipal` for an entire application domain, you can set the current thread's `IPrincipal` manually through the `System.Threading.Thread.CurrentPrincipal` property. `CurrentPrincipal` is `static` (C#) or `Shared` (Visual Basic .NET) and always affects the `IPrincipal` of the currently executing thread. You also use `CurrentPrincipal` to obtain the `IPrincipal` of the current thread.



Code must have the `ControlPrincipal` permission of the `System.Security.Permissions.SecurityPermission` class in order to call the `AppDomain.SetThreadPrincipal` method or set the `Thread.CurrentPrincipal` property. We discuss the `SecurityPermission` class in [Chapter 7](#).

If you do not use `AppDomain.SetThreadPrincipal` or `Thread.CurrentPrincipal` to assign an `IPrincipal` to a `Thread`, the *principal policy* of the application domain in which the thread is running determines what happens when code tries to obtain the thread's `CurrentPrincipal`. You can configure the principal policy of an application domain using the `System.AppDomain.SetPrincipalPolicy` method, which has the following signature:

```
# C#

public void SetPrincipalPolicy(
    PrincipalPolicy policy
);

# Visual Basic .NET

NotOverridable Public Sub SetPrincipalPolicy( _
    ByVal policy As PrincipalPolicy _
)
```

The `policy` argument is a member of the `System.Security.Principal.PrincipalPolicy` enumeration, whose values we list in [Table 10-2](#). The default principal policy for all application domains is `UnauthenticatedPrincipal`, which creates an empty `IPrincipal` that is not useful for making role-based security decisions.

To call `SetPrincipalPolicy`, code must have the `ControlPrincipal` permission of the `System.Security.Permissions.SecurityPermission` class.

Table 10-2. Members of the `PrincipalPolicy` enumeration

Value	Description
<code>NoPrincipal</code>	No principal or identity object is created. Getting the <code>Thread.CurrentPrincipal</code> property will return <code>null</code> (C#) or <code>Nothing</code> (Visual Basic .NET). All role-based security demands will fail.
	The runtime creates a <code>GenericPrincipal</code> containing a <code>GenericIdentity</code> with its <code>Name</code> and <code>AuthenticationType</code> properties

Value	Description
<code>UnauthenticatedPrincipal</code>	set to empty strings (""), and its <code>IsAuthenticated</code> property set to false. The <code>GenericIdentity</code> is a member of no roles. We discuss the <code>GenericIdentity</code> and <code>GenericPrincipal</code> classes in Section 10.2.3 later in this chapter.
<code>WindowsPrincipal</code>	The runtime creates a <code>WindowsPrincipal</code> containing a <code>WindowsIdentity</code> based on the Windows access token of the current thread. The roles contained in the <code>WindowsPrincipal</code> are the Windows groups of the current user. We discuss the <code>WindowsIdentity</code> and <code>WindowsPrincipal</code> classes in Section 10.2.3 later in this chapter.

As you can see from the three principal policy options, if you want to use an RBS implementation in your programs that does not rely the current Windows user, principal policy provides no benefit. It is your responsibility to ensure that the current thread has the correct `IPrincipal` associated with it. You must create the `IPrincipal` object and assign it to the thread using the `AppDomain.SetThreadPrincipal` method or the `Thread.CurrentPrincipal` property.

10.2.3 Programming the Windows Role-Based Security Implementation

The `WindowsIdentity` and `WindowsPrincipal` classes of the `System.Security.Principal` namespace provide an RBS implementation that allows you to base security decisions on the identity and roles of Windows user accounts. The `WindowsIdentity` class implements the `IIdentity` interface and represents a Windows user account. The Windows user's name, accessible through the `WindowsIdentity.Name` property, is of the form "Domain\User."

The `WindowsPrincipal` class implements `IPrincipal` and contains the user's `WindowsIdentity` object, along with the names of the Windows groups to which the user belongs. The name of any built-in Windows user groups has the prefix "BUILTIN\"-for example, "BUILTIN\Administrators" or "BUILTIN\Users". This is important to remember when using the `WindowsPrincipal.IsInRole` method to test role membership. Non-built-in group names are prefixed with the name of the domain to which the group belongs, or the machine name if the group exists on a standalone machine-for example, "MyDomain\Developers" or "MyMachine\Developers".

The `WindowsIdentity` and `WindowsPrincipal` do not provide access to the Windows operating system permissions granted to the user and groups they represent. You must base your RBS decisions solely on the username and the names of the groups to which the user belongs.

The `WindowsIdentity` class implements members in addition to those defined in the `IIdentity` interface. These additional members provide useful functionality for creating `WindowsIdentity` objects, testing what type of Windows account a `WindowsIdentity` object represents, and impersonating Windows users. We summarize the members of the `WindowsIdentity` class in [Table 10-3](#).

Table 10-3. Members of the WindowsIdentity class

Member	Description
Properties	
<code>AuthenticationType</code>	Defined in <code>IIdentity</code> . Returns a string identifying the mechanism used to authenticate the user represented by the <code>WindowsIdentity</code> object.
<code>IsAnonymous</code>	Returns true if the Windows account type represented by the <code>WindowsIdentity</code> object is an anonymous account. Normally, you will see only anonymous identities within ASP.NET applications; see Chapter 18 for a discussion of ASP.NET security.
<code>IsAuthenticated</code>	Defined in <code>IIdentity</code> . Returns true of the user represented by the <code>WindowsIdentity</code> object was authenticated; otherwise, it returns false.
<code>IsGuest</code>	Returns true if the Windows account type represented by the <code>WindowsIdentity</code> object is a guest account.
<code>IsSystem</code>	Returns true if the Windows account type represented by the <code>WindowsIdentity</code> object is a system account.
<code>Name</code>	Defined in <code>IIdentity</code> . Returns the Windows logon name of the user. The <code>Name</code> property returns a string of the form "Domain\User."
<code>Token</code>	Returns a <code>System.IntPtr</code> containing the handle of the Windows account token for the user represented by the <code>WindowsIdentity</code> object.
Methods	
<code>GetAnonymous</code>	Static method that returns a <code>WindowsIdentity</code> object that represents an anonymous Windows user account.
<code>GetCurrent</code>	Static method that returns a <code>WindowsIdentity</code> object that represents the currently logged-on Windows user account.
<code>Impersonate</code>	Allows code to impersonate other Windows users. See "Impersonating Windows Users."

The `WindowsPrincipal` class implements no role-based functionality other than that specified by the `IPrincipal` interface. The `Identity` property returns the contained `WindowsIdentity` as an `IIdentity` object; therefore, you must cast it to `WindowsIdentity` before you can use the additional methods listed in [Table 10-3](#).

Of note is the implementation of the `IsInRole` method, which has three overloads:

```
# C#

public virtual bool IsInRole(
    int rid
);

public virtual bool IsInRole(
    string role
);
```

```

public virtual bool IsInRole(
    WindowsBuiltInRole role
);

# Visual Basic .NET

Overridable Overloads Public Function IsInRole( _
    ByVal rid As Integer _
) As Boolean

Overridable Overloads Public Function IsInRole( _
    ByVal role As String _
) As Boolean

Overridable Overloads Public Function IsInRole( _
    ByVal role As WindowsBuiltInRole _
) As Boolean

```

In the first overload, the `rid` argument specifies a Windows Role Identifier (RID). RIDs are a component of a Windows group's security identifier and provide a way to identify groups independent of language localization. The second overload takes a case-insensitive string that specifies the name for which to test; this is the standard form of `IsInRole` defined in `IPrincipal`. The final overload takes a member of the `System.Security.Principal.WindowsBuiltInRole` enumeration as an argument; `WindowsBuiltInRole` contains values that represent the standard Windows groups. [Table 10-4](#) compares the role names, RIDs, and `WindowsBuiltInRole` values used to test for membership of the most commonly used Windows groups.

Table 10-4. Commonly used role names, RIDs, and `WindowsBuiltInRole` members

Name	RID (hex)	WindowsBuiltInRole
BUILTIN\Account Operators	0x224	AccountOperator
BUILTIN\Administrators	0x220	Administrator
BUILTIN\Backup Operators	0x227	BackupOperator
BUILTIN\Guests	0x222	Guest
BUILTIN\Power Users	0x223	PowerUser
BUILTIN\Print Operators	0x226	PrintOperator
BUILTIN\Replicator	0x228	Replicator
BUILTIN\Server Operators	0x225	SystemOperator
BUILTIN\Users	0x221	User

10.2.3.1 Configuring the current `WindowsPrincipal`

The easiest way to make the current principal represent the active Windows user is by calling the `System.AppDomain.SetPrincipalPolicy` method and passing it the value `PrincipalPolicy.WindowsPrincipal`. The first time code requests the thread's `IPrincipal`, the runtime creates a `WindowsIdentity` and a `WindowsPrincipal` for a thread based on the currently active Windows access token:

```
# C#

// Configure the current application domain's principal policy
// to represent the active Windows user
AppDomain.CurrentDomain.SetPrincipalPolicy(
    PrincipalPolicy.WindowsPrincipal);
```

```
# Visual Basic .NET
```

```
' Configure the current application domain's principal policy
' to represent the active Windows user
AppDomain.CurrentDomain.SetPrincipalPolicy( _
PrincipalPolicy.WindowsPrincipal)
```

You can also create `WindowsIdentity` and `WindowsPrincipal` objects manually and pass them to the `AppDomain.SetThreadPrincipal` method or the `Thread.CurrentPrincipal` property. The `AppDomain.SetThreadPrincipal` defines the default principal that the runtime assigns to any thread in the application domain, whereas the `Thread.CurrentPrincipal` property sets the principal of the current thread.

The `WindowsIdentity.GetCurrent` method returns a `WindowsIdentity` object that represents the current Windows user. Once you have a `WindowsIdentity` object, you can pass it to the `WindowsPrincipal` constructor and assign the new `WindowsPrincipal` to the current thread, as demonstrated by the following statements:

```
# C#
```

```
// Create a WindowsIdentity for the active Windows user
WindowsIdentity wi = WindowsIdentity.GetCurrent( );

// Create a new WindowsPrincipal
WindowsPrincipal wp = new WindowsPrincipal(wi);

// Assign the WindowsPrincipal to the active thread.
Thread.CurrentPrincipal = wp;
```

```
# Visual Basic .NET
```

```
' Create a WindowsIdentity for the active Windows user
Dim wi As WindowsIdentity = WindowsIdentity.GetCurrent( )

' Create a new WindowsPrincipal
Dim wp As WindowsPrincipal = New WindowsPrincipal(wi)

' Assign the WindowsPrincipal to the active thread.
Thread.CurrentPrincipal = wp
```

Using the `Thread.CurrentPrincipal` property or the `AppDomain.SetThreadPrincipal` method allows you to change the thread's current principal if you need your code to operate on behalf of a different principal—a process known as *impersonation*. However, changing the current principal does not change the current Windows access token.

10.2.3.2 Impersonating Windows users

Sometimes, you will want your code to act at the operating system level as though it is a different user than the one currently logged on. This is particularly common in server applications that process requests from different users and need to access resources, such as databases, on behalf of the user

The `WindowsIdentity` class provides the mechanism through which you can impersonate another Windows user. However, first you must obtain a Windows access token that represents the user you want to impersonate. Unfortunately, the .NET Framework class library does not currently contain classes that provide managed access to the Windows account database, and therefore you must call the `LogonUser` method of the unmanaged `advapi32.dll/Win32` library to obtain the access token. The `LogonUser` method takes username and password arguments, along with other parameters that control the authentication process, and provides access to an access token for the user. If the `LogonUser` fails, you will also need to call the `System.Runtime.InteropServices.Marshal.GetLastWin32Error` method to determine what the problem is. The `GetLastWin32Error` method exposes the `GetLastError` method of the `kernel32.dll` library and avoids possible problems arising from the CLR making internal calls to the WIN32 APIs that would overwrite the last error code.

On the Windows NT and Windows 2000 platforms, the account under which a program is running requires the Windows `SE_TCB_NAME` privilege to call `LogonUser` in order to obtain an access token. Without this privilege, calls to `LogonUser` fail and calling `GetLastError` returns the code `ERROR_PRIVILEGE_NOT_HELD` (value 1314). To grant an account the `SE_TCB_NAME` privilege, you must configure the local security policy to allow the account to "Act as part of the operating system." This highly trusted privilege allows the account to create access tokens that represent any user or permission set. You should only grant this permission to accounts created specifically to run your software; never grant this permission to user accounts.

Once you have the access token, you use it to create a new `WindowsIdentity` object and call its `Impersonate` method. The `Impersonate` method changes the Windows access token of the current thread. Any operating system operation you perform after calling `Impersonate` takes place as the impersonated user. The `Impersonate` method returns a `System.Security.Principal.WindowsImpersonationContext` object that represents the Windows user prior to impersonation. When you want to revert to the original user, call the `WindowsImpersonationContext.Undo` method.

[Example 10-1](#) demonstrates the impersonation of a Windows user named "Bob." Although the example demonstrates how to call `LogonUser` from managed code, a complete discussion of unmanaged code interoperability and the `LogonUser` method is beyond the scope of this book. You should consult the .NET Framework SDK documentation for details on calling unmanaged code and the Windows Platform SDK for details of the `LogonUser` method.

Example 10-1. Impersonating a Windows user

```

# C#

using System;
using System.IO;
using System.Security.Principal;
using System.Security.Permissions;
using System.Runtime.InteropServices;

// Make sure we have permission to execute unmanaged code
[assembly:SecurityPermission(SecurityAction.RequestMinimum,
    UnmanagedCode=true)]

public class WindowsImpersonationTest {

    // Define the external LogonUser method from advapi32.dll.
    [DllImport("advapi32.dll", SetLastError=true)]
    static extern int LogonUser(String UserName, String Domain,
        String Password, int LogonType, int LogonProvider,
        ref IntPtr Token);

    public static void Main( ) {

        // Create a new initialized IntPtr to hold the access token
        // of the user to impersonate.
        IntPtr token = IntPtr.Zero;

        // Call LogonUser to obtain an access token for the user
        // "Bob" with the password "treasure". We authenticate against
        // the local accounts database by specifying a "." as the Domain
        // argument.
        int ret = LogonUser(@"Bob", ".", "treasure", 2, 0, ref token);

        // If the LogonUser return code is zero an error has occurred.
        // Display it and exit.
        if (ret == 0) {
            Console.WriteLine("Error {0} occurred in LogonUser",
                Marshal.GetLastWin32Error( ));
            return;
        }

        // Create a new WindowsIdentity from Bob's access token
        WindowsIdentity wi = new WindowsIdentity(token);

        // Impersonate Bob, saving a reference to the returned
        // WindowsImpersonationContext.
        WindowsImpersonationContext impctx = wi.Impersonate( );

        // !!! Perform actions as Bob !!!
        // We create a file that Windows will show is owned by Bob.
        StreamWriter file = new StreamWriter("test.txt");
        file.WriteLine("Bob's test file.");
    }
}

```

```

file.Close( );

// Revert back to the original Windows user using the
// WindowsImpersonationContext object.
impctx.Undo( );
}
}

# Visual Basic .NET

Imports System
Imports System.IO
Imports System.Security.Principal
Imports System.Security.Permissions
Imports System.Runtime.InteropServices

' Make sure we have permission to execute unmanaged code
<assembly:SecurityPermission(SecurityAction.RequestMinimum, _
UnmanagedCode:=True)> _

Public Class WindowsImpersonationTest

    ' Define the external LogonUser method from advapi32.dll.
    <DllImport("advapi32.dll", SetLastError := True)> _
    Public Shared Function LogonUser(Username As String, _
    Domain As String, Password As String, LogonType As Integer, _
    LogonProvider As Integer, ByRef Token As IntPtr) As Integer
    End Function

    Public Shared Sub Main( )

        ' Create a new initialized IntPtr to hold the access token
        ' of the user to impersonate.
        Dim token As IntPtr = IntPtr.Zero

        ' Call LogonUser to obtain an access token for the user
        ' "Bob" with the password "treasure". We authenticate against
        ' the local accounts database by specifying a "." as the Domain
        ' argument.
        Dim ret As Integer = LogonUser("Bob",".","treasure",2,0, token)

        ' If the LogonUser return code is zero an error has occurred.
        ' Display it and exit.
        If ret = 0 Then
            Console.WriteLine("Error {0} occurred in LogonUser", _
            Marshal.GetLastWin32Error( ))
            Return
        End If

        ' Create a new WindowsIdentity from Bob's access token
        Dim wi As WindowsIdentity = New WindowsIdentity(token)

```

```

' Impersonate Bob, saving a reference to the returned
' WindowsImpersonationContext.
Dim impctx As WindowsImpersonationContext = wi.Impersonate( )

' !!! Perform actions as Bob !!!
' We create a file that Windows will show is owned by Bob.
Dim file As StreamWriter = New StreamWriter("test.txt")
file.WriteLine("Bob's test file.")
file.Close( )

' Revert back to the original Windows user using the
' WindowsImpersonationContext object.
impctx.Undo( )
End Sub
End Class

```

10.2.4 Making Role-Based Security Demands

Role-based security demands function similarly to code-access security demands. The key difference is that role-based security demands never result in a stack walk. The result of a role-based security demand is based solely on the identity and roles of the active thread's principal. If the thread's principal does not contain the demanded identity or roles, the runtime throws a `System.Security.SecurityException`. If the principal meets the demanded requirements, execution continues unaffected.

The `PrincipalPermission` class, with its attribute counterpart named `PrincipalPermissionAttribute`, provides the mechanisms through which you invoke role-based security demands in your programs. The `PrincipalPermission` class allows you to use imperative security syntax, and the `PrincipalPermissionAttribute` class enables the use of declarative syntax; both classes are members of the `System.Security.Permissions` namespace. We discussed the syntax and use of both imperative and declarative security demands in [Chapter 7](#), and therefore we will only cover them briefly in the following sections.

Unlike code-access security, which is enforced exclusively through permissions and attributes, it is commonplace to make role-based security decisions by calling the members of a thread's principal and identity objects directly. This is particularly useful if the RBS implementation includes functionality in addition to that defined in the `IIdentity` or `IPrincipal` interfaces. For example, the implementation may expose features specific to the underlying authentication or authorization mechanism, as is the case with the `WindowsIdentity` object that we discussed in [Section 10.2.3](#).

10.2.4.1 Using imperative role-based security statements

To make an imperative RBS demand, you must instantiate a `PrincipalPermission` object and call its `Demand` method. The most commonly used `PrincipalPermission` constructor has the signature shown here:

```
# C#
```



```

public PrincipalPermission(
    string name,
    string role
);

# Visual Basic .NET

Public Sub New( _
    ByVal name As String, _
    ByVal role As String _
)

```

The `name` and `role` arguments specify the values that the current principal must have for the `Demand` to succeed. Each `PrincipalPermission` can specify only a single role name. The current principal must match both the specified `name` and `role` values; however, you can specify `null` (C#) or `Nothing` (Visual Basic .NET) for either argument to match any value. [Table 10-5](#) explains how `PrincipalPermission` uses these arguments to determine the result of the `Demand`.

Table 10-5. Arguments of the `PrincipalPermission` constructors

Argument	Description
<code>name</code>	Must match the value held in the <code>Thread.CurrentPrincipal.Identity.Name</code> property. Both the <code>GenericPrincipal</code> and <code>WindowsPrincipal</code> classes implement case-insensitive comparisons of the <code>Name</code> property.
<code>role</code>	Used as an argument to the <code>Thread.CurrentPrincipal.IPrincipal.IsInRole</code> method to test if the current principal is a member of the specified role. Both the <code>GenericPrincipal</code> and <code>WindowsPrincipal</code> classes implement case-insensitive comparisons for <code>IsInRole</code> .

The following statements demonstrate the use of the `PrincipalPermission` class to invoke a variety of imperative RBS demands:

```

# C#

// Demand that the active principal has the identity name "Peter".
PrincipalPermission p1 = new PrincipalPermission("Peter", null);
p1.Demand( );

// Demand that the active principal is a member of the "Developers"
// group with any identity.
PrincipalPermission p2 = new PrincipalPermission(null, "Developers");
p2.Demand( );

// Demand that the active principal have the identity name "Bart"
// and be a member of the "Managers" group.
PrincipalPermission p3 = new PrincipalPermission("Bart", "Managers");
p3.Demand( );

```

```

# Visual Basic .NET

' Demand that the active principal has the identity name "Peter".
Dim p1 As PrincipalPermission = _
New PrincipalPermission("Peter",Nothing)
p1.Demand( )

' Demand that the active principal is a member of the "Developers"
' group with any identity.
Dim p2 As PrincipalPermission = _
New PrincipalPermission(Nothing,"Developers")
p2.Demand( )

' Demand that the active principal have the identity name "Bart"
' and be a member of the "Managers" group.
Dim p3 As PrincipalPermission = _
New PrincipalPermission("Bart","Managers")
p3.Demand( )

```

`PrincipalPermission` implements the `System.Security.IPermission` interface, and defines the `Copy`, `Intersect`, `IsSubsetOf`, and `Union` methods to manipulate `PrincipalPermission` objects. The composition of `PrincipalPermission` objects means the `Intersect` and `IsSubSetOf` methods are of little value; `PrincipalPermission` primarily implements them to satisfy the requirements of the `IPermission` interface. The `Union` method, however, does provide useful functionality, enabling you to make security demands that test against multiple sets of name/role pairs.

The `Union` of two `PrincipalPermission` objects is a `PrincipalPermission` object that contains the name and role elements of both source objects. For example if the two source `PrincipalPermission` objects contained the name/role values "Alice"/"managers" and "Bob"/"developers," the union would contain both name/role pairs. Calling `Demand` on the resulting `PrincipalPermission` would succeed if the current principal matches either name/role pair. However, `PrincipalPermission` maintains and compares the contained name/role pairs independently. Comparing the union against a principal with the name/role pair "Alice"/"developers" would fail. To understand the contents of a `PrincipalPermission` resulting from a `Union`, look at the following output from the `PrincipalPermission.ToString` method:

```

<Permission class="System.Security.Permissions.PrincipalPermission,
mscorlib, Version=1.0.5000.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"
    version="1">
  <Identity Authenticated="true"
    ID="Alice"
    Role="managers"/>
  <Identity Authenticated="true"
    ID="Bob"
    Role="developers"/>
</Permission>

```

10.2.4.2 Using declarative role-based security statements

You can apply `PrincipalPermissionAttribute` to classes, methods, properties, or events to force declarative security demands. The key difference between `PrincipalPermissionAttribute` and the

permission attributes (see [Chapter 7](#)) is that you cannot apply the `PrincipalPermissionAttribute` at the assembly level. This means you cannot make role-based permission requests, such as `RequestMinimum`, `RequestOptional`, and `RequestRefuse`. In addition, because RBS does not use or affect the call stack, there are no declarative stack override statements available. Thus, you can only invoke the following three role-based security statements using declarative syntax: `Demand`, `LinkDemand`, and `InheritanceDemand`.

The `PrincipalPermissionAttribute` class defines the `Name` and `Role` properties that you use to specify the name and role values that the current principal must have for the declarative RBS demand to succeed. The following statements demonstrate the equivalent `PrincipalPermissionAttribute` syntax used to make the same demands we made in the previous section:

```
# C#

// Demand that the active principal has the identity name "Peter".
[PrincipalPermission(SecurityAction.Demand, Name = "Peter")]

// Demand that the active principal is a member of the "Developers"
// group with any identity.
[PrincipalPermission(SecurityAction.Demand, Role = "Developers")]

// Demand that the active principal have the identity name "Bart"
// and be a member of the "Managers" group.
[PrincipalPermission(SecurityAction.Demand, Name = "Bart", Role = "Managers")]

# Visual Basic .NET

' Demand that the active principal has the identity name "Peter".
<PrincipalPermission(SecurityAction.Demand, Name := "Peter")> _

' Demand that the active principal is a member of the "Developers"
' group with any identity.
<PrincipalPermission(SecurityAction.Demand, Role := "Developers")> _

' Demand that the active principal have the identity name "Bart"
' and be a member of the "Managers" group.
<PrincipalPermission(SecurityAction.Demand, Name:="Bart", Role:="Managers")> _
```

10.2.5 Programming the Generic Role-Based Security Implementation

If you rely on an authority other than the Windows accounts system to authenticate and authorize your users, it is possible that the manufacturers of the system will provide `IIdentity` and `IPrincipal` implementations that work directly with the authority. If not, the simplest approach is to obtain identity and role information from the custom authority and use the `GenericIdentity` and `GenericPrincipal` classes from the `System.Security.Principal` namespace to enable role-based security support in your managed applications. The `GenericIdentity` and `GenericPrincipal` classes provide a generic RBS implementation that you can use independently of the Windows user accounts database. The `GenericIdentity` class implements `IIdentity`, and `GenericPrincipal` implements `IPrincipal`.

10.2.5.1 Configuring the current `GenericPrincipal`

The techniques for setting the current principal to a `GenericPrincipal` object are more limited than those we described earlier in [Section 10.2.3.1](#). It is impossible for an application domain's principal policy to create useful `GenericPrincipal` objects automatically; therefore, you have two options:

- Create a `GenericPrincipal` object and pass it to the `AppDomain.SetThreadPrincipal` method. This defines the default principal that the runtime will assign to any thread executing in the application domain.
- Create a `GenericPrincipal` object and use it to set the `Thread.CurrentPrincipal` property. This sets the principal of the current thread.

To create a `GenericPrincipal`, you must first create a `GenericIdentity`. The `GenericIdentity` class provides two constructors with the following signatures:

```
# C#

public GenericIdentity(
    string name
);

public GenericIdentity(
    string name,
    string type
);

# Visual Basic .NET

Public Sub New( _
    ByVal name As String _
)

Public Sub New( _
    ByVal name As String, _
    ByVal type As String _
)
```

The first constructor takes a single `name` argument, in which you specify the name of the user represented by the identity. In addition to the `name` argument, the second constructor takes a `type` argument. The `type` argument allows you to specify a string that identifies the type of authentication mechanism used to authenticate the user. The `GenericIdentity` class enforces no restrictions on the format or content of the `name` and `type` arguments as long as they contain only valid string characters. This provides maximum flexibility, which allows you to use the `GenericIdentity` class in conjunction with any custom authentication mechanisms. The following statements demonstrate the use of the `GenericIdentity` constructors:

```
# C#

GenericIdentity i1 = new GenericIdentity("Peter");
GenericIdentity i2 = new GenericIdentity("Peter", "SmartCard");
```

```
# Visual Basic .NET
```

```
Dim i1 As GenericIdentity = New GenericIdentity("Peter")
Dim i2 As GenericIdentity = New GenericIdentity("Peter", "SmartCard")
```

The `GenericPrincipal` class provides a single constructor that takes an `IIdentity` object and a string array as arguments:

```
# C#
```

```
public GenericPrincipal(
    IIdentity identity,
    string[] roles
);
```

```
# Visual Basic .NET
```

```
Public Sub New( _
    ByVal identity As IIdentity, _
    ByVal roles( ) As String _
)
```

The `identity` argument can be of any type that implements `IIdentity` and represents the user that the `GenericPrincipal` should represent. The `roles` argument takes a string array containing the set of role names to which the user belongs. You must specify the user roles in the `GenericPrincipal` object constructor because the `GenericPrincipal` class is independent of any underlying user-authorization mechanism. The following statements demonstrate the creation of a `GenericPrincipal` from a `GenericIdentity`, and then make the new `GenericPrincipal` the current principal using the `Thread.CurrentPrincipal` property:

```
# C#
```

```
// Create a GenericIdentity for the user Peter
GenericIdentity gi = new GenericIdentity("Peter");

// Create a GenericPrincipal for Peter and specify membership of the
// Developers and Managers roles
String[] roles = new String[]{"Developers", "Managers"};
GenericPrincipal gp = new GenericPrincipal(gi, roles);

// Assign the new principal to the current thread
Thread.CurrentPrincipal = gp;
```

```
# Visual Basic .NET
```

```
' Create a GenericIdentity for the user Peter
Dim gi As GenericIdentity = New GenericIdentity("Peter")

' Create a GenericPrincipal for Peter and specify membership of the
' Developers and Managers roles
Dim roles( ) As String = New String( ) {"Developers", "Managers"}
Dim gp As GenericPrincipal = New GenericPrincipal(gi, roles)
```

```
' Assign the new principal to the current thread
Thread.CurrentPrincipal = gp;
```

Both the `GenericIdentity` and `GenericPrincipal` classes are simple, offering no role-based functionality other than that defined in their respective `IIdentity` and `IPrincipal` interfaces. In [Table 10-6](#), we describe the `GenericIdentity` and `GenericPrincipal` classes' implementations of the `IIdentity` and `IPrincipal` interfaces.

Table 10-6. Members of `GenericIdentity` and `GenericPrincipal` classes

Member	Description
<code>GenericIdentity</code> class	
<code>AuthenticationType</code>	Read-only property that gets the authentication type specified in the <code>type</code> argument of the <code>GenericIdentity</code> constructor. If <code>type</code> was not specified, an empty string is returned.
<code>IsAuthenticated</code>	Read-only property that returns true as long as the username specified in the <code>name</code> argument of the <code>GenericIdentity</code> constructor was not an empty string; otherwise, it returns false.
<code>Name</code>	Read-only property that gets the username specified in the <code>name</code> argument of the <code>GenericIdentity</code> constructor.
<code>GenericPrincipal</code> class	
<code>Identity</code>	Read-only property that gets the <code>IIdentity</code> object specified in the <code>identity</code> argument of the <code>GenericPrincipal</code> constructor. You must cast it to the right type, which will normally be <code>GenericIdentity</code> .
<code>IsInRole</code>	Method that takes a string argument containing the name of a role and returns true if the specified role name matches one of those contained in the <code>roles</code> argument of the <code>GenericPrincipal</code> constructor. The comparison is case-insensitive.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 11. Isolated Storage

Isolated storage is the final aspect of .NET runtime security we discuss. Isolated storage is a data storage mechanism that allows managed code to store user and application-specific data to disk. The use of isolated storage simplifies programming and minimizes the security risks associated with allowing code to store data to a computer's hard drive. In this chapter, we describe what isolated storage is and explain where it provides benefits over existing data storage options. We demonstrate how to use isolated storage in your own programs, as well as how to administer and control access to it.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

11.1 Isolated Storage Explained

Many applications need to write data to a persistent store so that it's available each time the application runs. Data, such as user preferences and application state, is generally user-specific and needs to be stored in such a way that other users, and possibly other applications, cannot access and modify it. On the Windows platform, there has been no standard method of doing this, and therefore different applications have taken different approaches. These include using the Windows registry, configuration files (such as *.ini* files), or databases.

Windows features such as role-based security, ACL-based file permissions, user profiles, and the registry's `HKEY_CURRENT_USER` hive make it relatively easy for code to store data that other users cannot access. However, it is not so easy to ensure that applications run by the same user cannot read and modify each other's data. This presents significant security risks, especially with the increase in the use of mobile code and dynamically downloaded components.

Isolated storage provides a controlled and structured mechanism through which code can write data to the hard drive in a way that prevents access by other users and managed applications. When you use isolated storage in your programs, you do not have to worry about the mechanics of isolation and you do not have to choose a storage location that will not conflict with those used by other applications. In addition, isolated storage can leverage the capabilities of Windows-roaming user profiles, ensuring that a user's data is available to them regardless of which machine they use.

Before managed code can save data to isolated storage, it must obtain a *store*. A store is a compartment within isolated storage that has an identity derived from the identity of the user and code that created it. When code tries to obtain a store, isolated storage first looks for an existing store with the correct identity to pass to the code. If one does not exist, isolated storage creates a new store automatically. The use of stores means that each time code runs, it has access to any data that it previously saved to isolated storage. Because different applications run by the same user have different identities, they each obtain unique stores. We describe how isolated storage determines the user and code's identity later in [Section 11.1.1.1](#).

Isolated storage provides a more secure data storage alternative than granting code direct access to selected areas of the hard drive. Different code-access permissions control access to isolated storage and normal hard disk access, simplifying security administration and ensuring that code granted access to isolated storage can't gain unforeseen access to other areas of the hard disk. In addition, each store appears to be a root-level container for the code using it. Within its own store, code is free to create files and directories as if it was working directly with the filesystem, but code cannot manipulate file or directory paths to access data in other stores, or other areas of the hard drive.

Isolated storage also allows security administrators to configure quotas that limit the maximum size of each store, ensuring that no application can create a store so large that it fills the entire hard disk. Isolated storage enforces these quotas as code writes data to its store. In summary, it is a relatively low-risk decision to give any code permission to use isolated storage, even code downloaded from unknown and untrusted sources. Isolated storage provides a perfect data storage solution for use by controls downloaded from the Internet.

11.1.1 Levels of Isolation

Isolated storage manages the isolation of application data automatically, always ensuring that one user cannot access the data of another. However, the level of isolation (isolation scope) between different applications run by the same user is configurable when the application obtains a store. Isolated storage supports two primary levels of isolation scope:

- Isolation by user and assembly
- Isolation by user, assembly, and application domain

You can configure each of these isolation scopes to support roaming users as long as the application is running on a platform that supports Windows-roaming profiles. In the following sections, we discuss these two isolation scopes and describe where each is preferable. First, we describe how isolated storage determines the identity of users and code on which it bases the isolation of stores.

11.1.1.1 Determining user and code identity

User-level isolation is implicit in isolated storage because of the structure of the underlying file store. A user's isolated storage data is stored in his Windows profile directory, and is always separate from all other user's data and protected by Windows file permissions. The storage location of a user's profile is dependant on the version of Windows and whether roaming profiles are being used. The .NET Framework SDK documentation lists the storage location used for each version of Windows.

Isolated storage achieves isolation at the assembly and application domain level by assigning each store an identity derived from the code that created it. Isolated storage uses the evidence possessed by the creating code to establish the store's identity; we discussed evidence in [Chapter 6](#). Only code that presents the same evidence values can obtain the store in the future.

With user and assembly isolation, one piece of evidence from the assembly is used. When isolating by user, assembly, and application domain, isolated storage uses one piece, of evidence from the assembly and one from the application domain for the store's identity. This list shows the order in which isolated storage will use available evidence objects (all from the `System.Security.Policy` namespace):

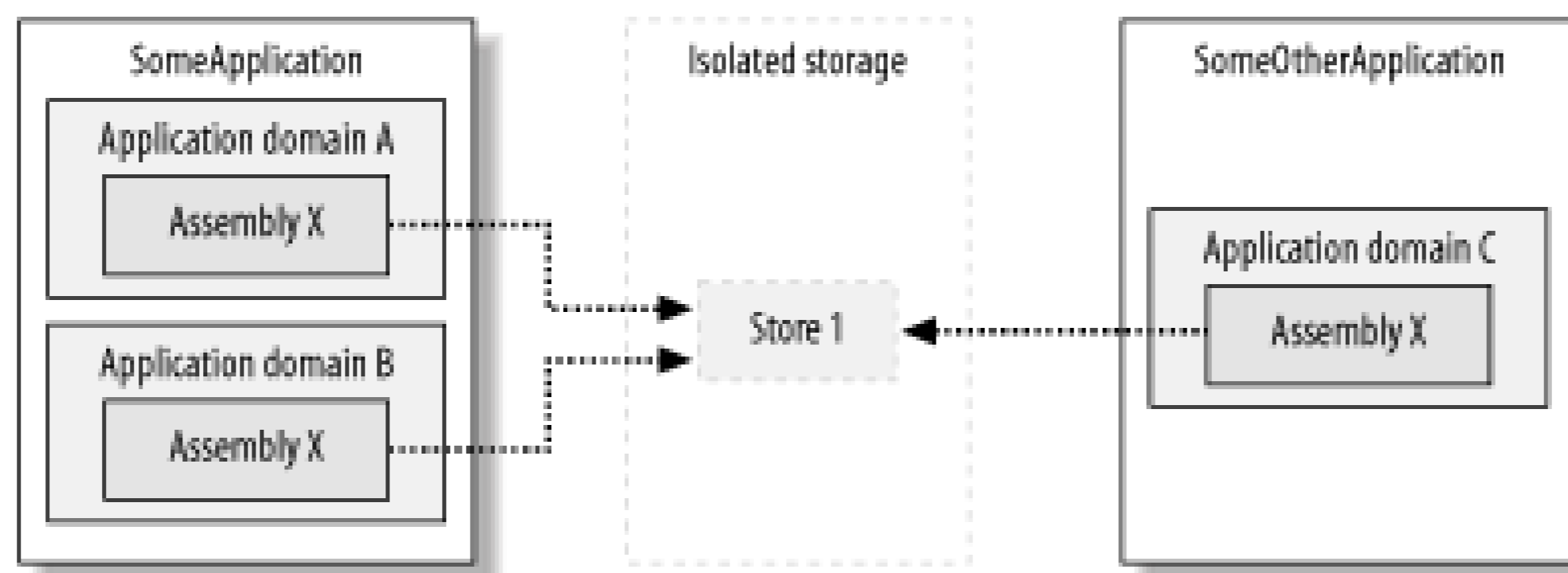
1. `Publisher`
2. `StrongName`
3. `Url`
4. `Site`
5. `Zone`

In the unlikely event that an assembly or application domain has none of these evidence objects, the runtime will throw a `System.IO.IsolatedStorage.IsolatedStorageException` when code tries to obtain a store.

11.1.1.2 Isolation by user and assembly

When code obtains a store isolated by user and assembly, isolated storage returns a store with an identity based on the current user and the assembly that made the call to obtain the store. As shown in [Figure 11-1](#), if an assembly is loaded into different application domains within a single application, the assembly will obtain the same store as long as it presents the same evidence. Even if the assembly is loaded into entirely different applications or Windows processes, as long as it presents the same evidence, it will obtain the same store. This kind of store sharing is most likely to happen if the assembly presents `Publisher` or `StrongName` evidence.

Figure 11-1. Isolation by user and assembly



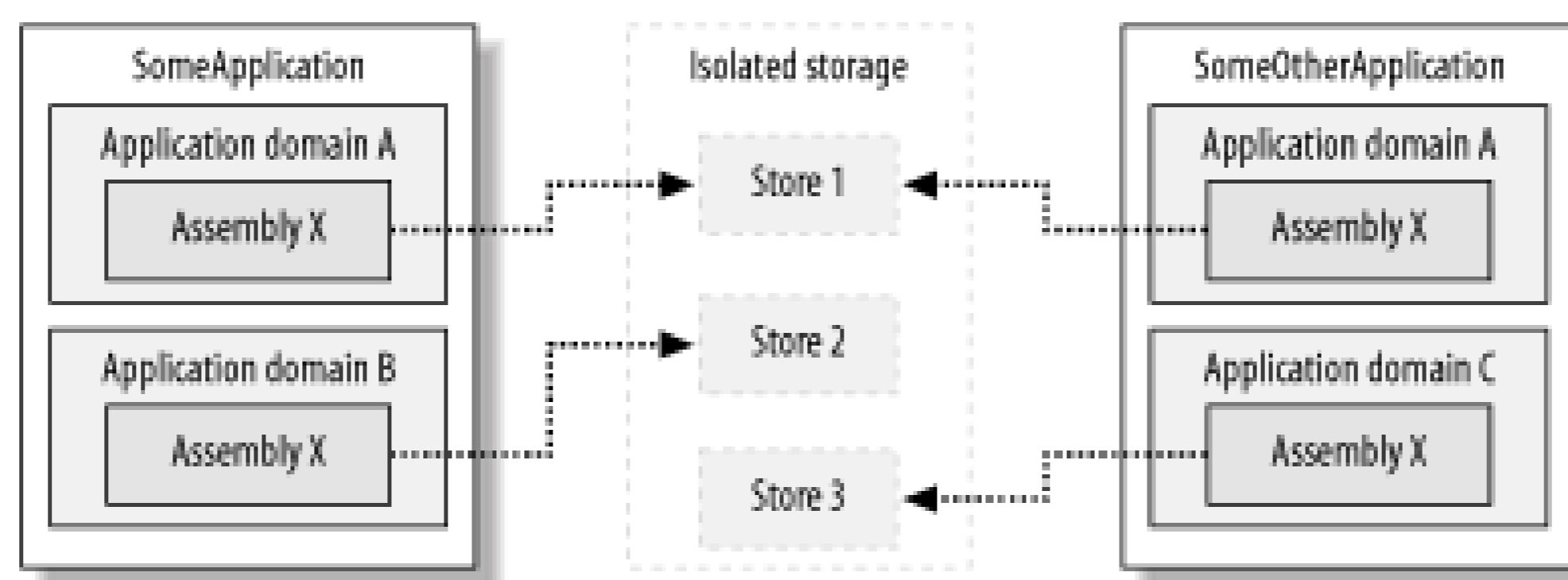
Isolation by user and assembly enables an assembly used in multiple applications to share data across those applications. In many instances, this is desirable behavior. For example, an assembly that implements spellchecker functionality that is used in both a word processor and a spreadsheet can save user preferences to affect the spellchecker behavior in both applications.

Used inappropriately, isolation by user and assembly can cause data leaks between applications. Depending on how the assembly uses isolated storage, this can result in an application gaining access to data that it should not have, or unpredictable behavior as one application overwrites the settings of another. Because of the risk of data leaks, the ability to obtain a store isolated by user, assembly, and application domain requires a higher level of trust than obtaining a store isolated by user, assembly, and application domain. We discuss code-access permission requirements in [Section 11.2.1](#) later in this chapter.

11.1.1.3 Isolation by user, assembly, and application domain

Isolation by user, assembly, and application domain limits the ability of an assembly to share or leak data across multiple applications. When code obtains a store isolated by user, assembly, and application domain, isolated storage returns a store with an identity based on the current user, the assembly that made the call to obtain the store, and the application domain in which the assembly is loaded. As shown in [Figure 11-2](#), even if the same assembly tries to obtain a store, because each instance is loaded into an application domain with unique identities, isolated storage returns a different store. Unless there is a specific need to share data across multiple applications, you are best to use isolation by user assembly and application domain for the extra guarantee of isolation it provides.

Figure 11-2. Isolation by user, assembly, and application domain



As [Figure 11-2](#) also shows, if two application domains have the same identity, then it is still possible for applications to share data; see [Section 11.1.1.1](#) for more details. However, this is not common and is unlikely to happen by coincidence, as it requires a high level of trust for code to manipulate the identity of an application domain (see [Chapter 6](#) for details).

11.1.2 Limitations of Isolated Storage

As useful as isolated storage is, it is not the answer to every managed application's data storage requirements. Here are some examples of when the use of isolated storage is inappropriate:

When valuable or secret data is used

Isolated storage is not secure and you should not use it to store sensitive or secret data without first encrypting that data. Although the .NET runtime ensures isolation of data between managed applications, any application that has direct access to the area of the hard disk that contains the isolated storage files can access the data they contain. This includes both native applications and managed applications with permission to access the hard drive. In addition, if data needs to be backed up, isolated storage should not be used unless you have a distributed backup tool, or use roaming profiles and accept the risk that a user's most recent data may not always be contained in the centralized roaming-profile store.

When data between users is shared

The only way one user can access another user's isolated storage data is by using impersonation, which we discussed in [Chapter 10](#). This is not a suitable mechanism for most data-sharing solutions, so you should use a shared network drive, or a database instead of isolated storage.

When managed and nonmanaged code need application access

If both managed and nonmanaged code need to access shared data, storing it in isolated storage is not a good solution. Although the nonmanaged application can access the isolated storage files through direct access to the hard drive, isolated storage manages the structure of the file store to meet its own needs, and is not easy to work with directly.

When there is no roaming-profile support

If you need the isolated storage data to be available to a user regardless of the machine he uses, but you do not have a network with Windows-roaming profiles enabled, you should not use isolated storage. Without roaming profiles, a user will have different data stored on every machine he uses.

When used as a primary file store

You should not use isolated storage as the primary storage area for datafiles produced by your

application. For example, it is inappropriate for email clients, word processors, and the like to store a user's files in isolated storage. The primary purpose of isolated storage is to store small amounts of noncritical data.

When large amounts of data are used

You must take the number of different people that use an application and the use of roaming profiles into consideration, but it is generally not a good idea to store large amounts of data in isolated storage. If many people use an application that consumes large amounts of isolated storage, it can easily fill up the disk space because isolated storage enforces quotas at the individual store level, not across all stores cumulatively.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

11.2 Programming Isolated Storage

Isolated storage is simple to use and requires knowledge of relatively few classes. The most important class is `System.IO.IsolatedStorage.IsolatedStorageFile`. `IsolatedStorageFile` objects represent individual stores and provide methods to manage the files and directories contained within the stores, as well as properties to access information, such as the store's isolation scope, current size, and maximum size. The `IsolatedStorageFile` class also provides static factory methods through which you initially obtain store objects.

Also important is the `System.IO.IsolatedStorage.IsolatedStorageFileStream` class, which extends `System.IO.FileStream` and provides the mechanism through which you read and write data to files in isolated storage.

In the following sections, we demonstrate how to use these classes to implement the use of isolated storage in your programs. First, we look at the code-access permissions used to control access to isolated storage.

11.2.1 Isolated Storage and Code-Access Security

The `System.Security.Permissions.IsolatedStorageFilePermission` code-access permission class controls access to isolated storage. Unlike the `System.Security.Permissions.FileIOPermission` class that controls direct access to the hard drive, the `IsolatedStorageFilePermission` class does not control the type of access (read or write) to specific directories or even specific stores. `IsolatedStorageFilePermission` controls access to isolated storage by restricting the scope of isolation at which code can obtain a store. Permission to obtain a store of a specified scope implies the ability for code to read and write data in the store. The `IsolatedStorageFilePermission` class also controls the maximum size that code can make a store.

If different code obtains the same store, the maximum store size granted to each assembly may be different based on the configuration of the `IsolatedStorageFilePermission` granted to the each assembly.

When configuring `IsolatedStorageFilePermission` instances, the `System.Security.Permissions.IsolatedStorageContainment` enumeration represents the different isolation scopes available, as well as some levels of access that transcend the individual isolation scopes. [Table 11-1](#) lists the members of the `IsolatedStorageContainment` enumeration and describes the access they represent.

Table 11-1. Members of the `IsolatedStorageContainment` enumeration

Member	Description
None	No permission to access isolated storage
DomainIsolationByUser	Permission to obtain a nonroaming store isolated by user, assembly, and application domain
AssemblyIsolationByUser	Permission to obtain a nonroaming store isolated by user and assembly
DomainIsolationByRoamingUser	Permission to obtain a roaming store isolated by user, assembly, and application domain
AssemblyIsolationByRoamingUser	Permission to obtain a roaming store isolated by user and assembly
AdministerIsolatedStorageByUser	Permission to enumerate and obtain all stores for the current user, but not the ability to modify the stores contents
UnrestrictedIsolatedStorage	Grants code unrestricted access to all stores isolated by the current user



The `IsolatedStorageFilePermission` treats the `DomainIsolationByUser`, `AssemblyIsolationByUser`, `DomainIsolationByRoamingUser`, and `AssemblyIsolationByRoamingUser` values of the `IsolatedStorageContainment` enumeration as a hierarchy of permissions. This introduces an important inconsistency in the control of access to isolated storage. Granting code the `DomainIsolationByRoamingUser` permission results in the code also obtaining a store isolated by user and assembly (`AssemblyIsolationByUser`) only. This introduces a significant risk because an assembly used by multiple applications can leak data between those applications, when the expected behavior allows code to obtain a roaming store isolated by user, assembly, and application domain.

Despite the relative safety of isolated storage, .NET's default security policy (discussed in [Chapter 9](#)) restricts the isolation scope of stores some code can obtain. In particular, code run from the Internet can obtain only a store isolated by user, assembly, and application domain. This minimizes the chance that applications downloaded from the Internet can share data between each other.

If you intend to use isolated storage in your code, it is prudent to use a permission request, which we discussed in [Chapter 7](#), to ensure the runtime grants your code the isolation scope it requires. The following code demonstrates the statements that request permission to create stores of some sample isolation scopes:

```
# C#

// Minimum permission request to obtain a non-roaming store isolated by
// user, assembly, and application domain
[assembly:IsolatedStorageFilePermission(SecurityAction.RequestMinimum,
UsageAllowed = IsolatedStorageContainment.DomainIsolationByUser)]

// Minimum permission request to obtain a roaming store isolated by
// user and assembly with a quota of 2048 bytes
```

```
[assembly:IsolatedStorageFilePermission(SecurityAction.RequestMinimum,
UsageAllowed = IsolatedStorageContainment.AssemblyIsolationByRoamingUser,
UserQuota = 2048)]

// Minimum permission request for unrestricted access to isolated storage
[assembly:IsolatedStorageFilePermission(SecurityAction.RequestMinimum,
Unrestricted = true)]

# Visual Basic .NET

' Minimum permission request to obtain a non-roaming store isolated by
' user, assembly, and application domain
<Assembly:IsolatedStorageFilePermission(SecurityAction.RequestMinimum, _
UsageAllowed := IsolatedStorageContainment.DomainIsolationByUser)>

' Minimum permission request to obtain a roaming store isolated by
' user and assembly with a quota of 2048 bytes
<Assembly:IsolatedStorageFilePermission(SecurityAction.RequestMinimum, _
UsageAllowed:=IsolatedStorageContainment.AssemblyIsolationByRoamingUser, _
UserQuota := 2048)>

' Minimum permission request for unrestricted access to isolated storage
<Assembly:IsolatedStorageFilePermission(SecurityAction.RequestMinimum, _
Unrestricted := true)>
```

You cannot `Assert` the `IsolatedStorageFilePermission` in your code; attempts to `Assert` are ignored.

11.2.2 Obtaining a Store

Two static methods of the `IsolatedStorageFile` class provide the simplest and most common way of obtaining a store:

`GetUserStoreForAssembly`

Returns a nonroaming store isolated by user and assembly. To invoke `GetUserStoreForAssembly`, the calling code must have the `AssemblyIsolationByUser` element of the `IsolatedStorageFilePermission` permission.

`GetUserStoreForDomain`

Returns a nonroaming store isolated by user, assembly, and application domain. To invoke `GetUserStoreForDomain`, the calling code must have the `DomainIsolationByUser` element of the `IsolatedStorageFilePermission` permission.

Both methods take no arguments and return an `IsolatedStorageFile` object representing the store with the appropriate identity for the current user and code. If a store with the necessary identity does not already exist, isolated storage creates a new one. The following statements demonstrate the use of both methods:

```
# C#
```

```

// Obtain a store isolated by user and assembly
IsolateStorageFile iso1 = IsolatedStorageFile.GetUserStoreForAssembly( );

// Obtain a store isolated by user, assembly, and application domain
IsolateStorageFile iso2 = IsolatedStorageFile.GetUserStoreForDomain( );

# Visual Basic .NET

' Obtain a store isolated by user and assembly
Dim iso1 As IsolateStorageFile = _
IsolatedStorageFile.GetUserStoreForAssembly( )

' Obtain a store isolated by user, assembly, and application domain
Dim iso2 As IsolateStorageFile = _
IsolatedStorageFile.GetUserStoreForDomain( )

```

The static `IsolatedStorageFile.GetStore` method also obtains a store, but allows the caller to specify the scope and identity of the store to obtain. `GetStore` has three overloads that provide different ways to specify the identity of the store. We show the most commonly used signature here; refer to the .NET Framework documentation for a discussion of all available overloads:

```

# C#

public static IsolatedStorageFile GetStore {
    IsolatedStorageScope scope,
    object domainIdentity,
    object assemblyIdentity
};

# Visual Basic .NET

Overloads Public Shared Function GetStore( _
    ByVal scope As IsolatedStorageScope, _
    ByVal domainIdentity As Object, _
    ByVal assemblyIdentity As Object _
) As IsolatedStorageFile

```

The `domainIdentity` and `assemblyIdentity` arguments represent the evidence objects for isolated storage to use when determining the identity of the store to obtain. Specifying `null` (C#) or `Nothing` (Visual Basic .NET) for both arguments means isolated storage will use its default approach to determine code identity, which we discussed earlier in [Section 11.1.1.1](#).

The `scope` argument takes a value of the `System.IO.IsolatedStorage.IsolatedStorageScope` enumeration that identifies the desired isolation scope for the store. The `IsolatedStorageScope` enumeration contains the following five values:

- Assembly
- Domain
- None

- Roaming
- User

By themselves, these values serve no purpose; only by combining them using a logical `Or` operation do they have any meaning to the `GetStore` method. [Table 11-2](#) lists the valid combination of values. The `|` symbol is the C# equivalent of the Visual Basic `.NET` keyword `Or`.

Table 11-2. Valid `IsolatedStorageScope` values for `GetStore`

Values	Description
<code>User Assembly</code>	Requests a store isolated by user and assembly, the same as that returned by the <code>GetUserStoreForAssembly</code> method
<code>User Assembly Domain</code>	Requests a store isolated by user, assembly, and application domain, the same as that returned by the <code>GetUserStoreForDomain</code> method
<code>Roaming User Assembly</code>	Requests a roaming store isolated by user and assembly
<code>Roaming User Assembly Domain</code>	Requests a roaming store isolated by user, assembly, and application domain

`GetStore` is the only way to obtain stores backed by Windows-roaming profiles. If you use `GetUserStoreForAssembly`, `GetUserStoreForDomain`, or do not specify the `Roaming` flag when calling `GetStore`, the store will exist only on the machine on which it was created and will not be accessible to the user when she uses other machines.

Because the `GetStore` method allows you to override the default evidence used to identify the store to obtain, you can obtain another application's store if you specify the correct evidence values—a relatively easy task. This breaks the normal code-level isolation provided by isolated storage; therefore, the calling code must have the more highly trusted `AdministerIsolatedStorageByUser` element of the `IsolatedStorageFilePermission` to call `GetStore` with evidence arguments that are not `null` (C#) or `Nothing` (Visual Basic `.NET`).

The following statements demonstrate the use of the `GetStore` method, including method calls that are equivalent to calling the `GetUserStoreForAssembly` and `GetUserStoreForDomain` methods:

```
# C#

// GetStore equivalent of GetUserStoreForAssembly
IsolatedStorageFile iso1 = IsolatedStorageFile.GetStore(
    IsolatedStorageScope.User | IsolatedStorageScope.Assembly,
    null, null
);

// GetStore equivalent of GetUserStoreForDomain
IsolatedStorageFile iso2 = IsolatedStorageFile.GetStore(
    IsolatedStorageScope.User | IsolatedStorageScope.Assembly |
    IsolatedStorageScope.Domain, null, null
```

```

);

// Get a roaming store isolated by user and assembly. Specify
// new Url evidence for the assembly identity
IsolatedStorageFile iso3 = IsolatedStorageFile.GetStore(
    IsolatedStorageScope.User | IsolatedStorageScope.Assembly |
    IsolatedStorageScope.Roaming, null, new Url(@"http://test.com")
);

# Visual Basic .NET

' GetStore equivalent of GetUserStoreForAssembly
Dim iso1 As IsolatedStorageFile = IsolatedStorageFile.GetStore( _
    IsolatedStorageScope.User Or IsolatedStorageScope.Assembly, _
    Nothing, Nothing)

' GetStore equivalent of GetUserStoreForDomain
Dim iso2 As IsolatedStorageFile = IsolatedStorageFile.GetStore( _
    IsolatedStorageScope.User Or IsolatedStorageScope.Assembly Or _
    IsolatedStorageScope.Domain, Nothing, Nothing)

' Get a roaming store isolated by user and assembly. Specify
' new Url evidence for the assembly identity
Dim iso3 As IsolatedStorageFile = IsolatedStorageFile.GetStore( _
    IsolatedStorageScope.User Or IsolatedStorageScope.Assembly Or _
    IsolatedStorageScope.Roaming, Nothing, New Url("http://test.com"))

```

The final way to obtain a store is to enumerate all stores for the current user. This breaks the data isolation provided by isolated storage completely and requires the caller to have the `AdministerIsolatedStorageByUser` element of the `IsolatedStorageFilePermission` permission. The `IsolatedStorageFile.GetEnumerator` method has the following signature:

```

# C#
public static IEnumerable GetEnumerator(
    IsolatedStorageScope scope
);

# Visual Basic .NET

Public Shared Function GetEnumerator( _
    ByVal scope As IsolatedStorageScope _
) As IEnumerable

```

The `scope` argument takes a value of the `IsolatedStorageScope` enumeration; [Table 11-3](#) lists the valid member combinations.

Table 11-3. Valid `IsolatedStorageScope` values for `GetEnumerator`

Values	Description
User	Returns an enumeration across all nonroaming stores for the current user
Roaming User	Returns an enumeration across all roaming stores for the current user

You can read data from `IsolatedStorageFile` objects obtained through the `GetEnumerator` method, but you cannot write to them. Methods that write data to stores ensure that no write operation increases the size of the store beyond that specified by the `IsolatedStorageFilePermission` permission granted to the executing code. When code obtains a store through the `GetStore`, `GetUserStoreForAssembly`, or `GetUserStoreForDomain` methods, the code's permissions determine the maximum size of the store. This figure is stored in the `IsolatedStorageFile.MaximumSize` property. However, code that executes `GetEnumerator` has no defined quota and so no maximum size is set. The lack of a maximum size value does not, however, result in an unrestricted store quote. Any attempt by code that obtains a store using `GetEnumerator` to read or modify the `MaximumSize` property will cause a `System.InvalidOperationException` to be thrown.

[Example 11-1](#) demonstrates the use of the `GetEnumerator` method to obtain all nonroaming stores for the current user. We then step through the stores and display the stores' isolation scope, current size, and the evidence of the code that created the stores using the `Scope`, `CurrentSize`, `AssemblyIdentity`, and `DomainIdentity` properties:

Example 11-1. Enumerating and displaying the characteristics of isolated stores

```
# C#

using System;
using System.Collections;
using System.IO.IsolatedStorage;
using System.Security.Permissions;

// Ensure we have permission to enumerate stores
[assembly:IsolatedStorageFilePermission(SecurityAction.RequestMinimum,
UsageAllowed = IsolatedStorageContainment.AdministerIsolatedStorageByUser)]

public class EnumerateStores {

    public static void Main( ) {

        // Obtain an IEnumerator across all non roaming stores for the
        // current user
        IEnumerator e =
            IsolatedStorageFile.GetEnumerator(IsolatedStorageScope.User);

        // Iterate through the enumerator
        while (e.MoveNext( )) {

            IsolatedStorageFile i = (IsolatedStorageFile)e.Current;
```

```

// Display the scope of the current store
Console.WriteLine("STORE SCOPE: {0}", i.Scope);

// Display the maximum and current size for the store
Console.WriteLine("Current size={0}", i.CurrentSize);

// Display the assembly identity of the current store
Console.WriteLine("Assembly ID:");
Console.WriteLine(i.AssemblyIdentity);

// Display the domain identity of the current store if
// the store is isolated by domain
if ((i.Scope & IsolatedStorageScope.Domain) != 0) {

    Console.WriteLine("Domain ID:");
    Console.WriteLine(i.DomainIdentity);
}
}
}
}

# Visual Basic .NET

Imports System
Imports System.Collections
Imports System.IO.IsolatedStorage
Imports System.Security.Permissions

' Ensure we have permission to enumerate stores
<assembly:IsolatedStorageFilePermission(SecurityAction.RequestMinimum, _
UsageAllowed := IsolatedStorageContainment.AdministerIsolatedStorageByUser _
)> _

Public Class EnumerateStores

    Public Shared Sub Main( )

        ' Obtain an IEnumerator across all non roaming stores for the
        ' current user
        Dim e As IEnumerator = _
        IsolatedStorageFile.GetEnumerator(IsolatedStorageScope.User)

        ' Iterate through the enumerator
        While e.MoveNext( )

            Dim i As IsolatedStorageFile = _
            CType(e.Current, IsolatedStorageFile)

            ' Display the scope of the current store
            Console.WriteLine("STORE SCOPE: {0}", i.Scope)

            ' Display the maximum and current size for the store

```

```

        Console.WriteLine("Current size={0}", i.CurrentSize)

        ' Display the assembly identity of the current store
        Console.WriteLine("Assembly ID:")
        Console.WriteLine(i.AssemblyIdentity)

        ' Display the domain identity of the current store if
        ' the store is isolated by domain
        If (i.Scope And IsolatedStorageScope.Domain) <> 0 Then

            Console.WriteLine("Domain ID:")
            Console.WriteLine(i.DomainIdentity)
        End If
    End While
End Sub
End Class

```

11.2.3 Creating Directories

Once you have an `IsolatedStorageFile` object representing a store, you can create files and directories in the store. .NET's isolated storage implementation does not provide an object that represents directories; this simplifies the isolated storage class structure but also forces you to constantly work with strings that represent directory paths. The `CreateDirectory` method, with the following signature, creates a directory with the specified fully qualified name:

```

# C#

public void CreateDirectory(
    string dir
);

# Visual Basic .NET

Public Sub CreateDirectory( _
    ByVal dir As String _
)

```

You can create more than one level of the directory hierarchy in a single call. For example, the following statements create a directory named `Dir1`, as well as two subdirectories named `Dir2` and `Dir3` in the store represented by the `iso` object:

```

# C#

IsolatedStorageFile iso = IsolatedStorageFile.GetUserStoreForDomain( );
iso.CreateDirectory(@"Dir1\Dir2");
iso.CreateDirectory("Dir1/Dir3");

# Visual Basic .NET

Dim iso As IsolatedStorageFile = IsolatedStorageFile.GetUserStoreForDomain( )
iso.CreateDirectory("Dir1\Dir2")

```

```
iso.CreateDirectory("Dir1/Dir3")
```

On the first call, `CreateDirectory` creates both the `Dir1` and `Dir2` directories. If you specify a directory name that already exists, as is the case with `Dir1` in the second call to `CreateDirectory`, no error occurs. If you specify a directory name that contains illegal characters, `CreateDirectory` throws an `IsolatedStorageException`. As this code shows, you can use either the forward slash / or backward slash \ characters as the path separators.

11.2.4 Creating, Reading, and Writing Files

The `IsolatedStorageFileStream` class represents a file in a store. `IsolatedStorageFileStream` extends `System.IO.FileStream` and apart from its `Handle` property, supports all of the normal file I/O capabilities that you are familiar with, including asynchronous I/O. If you use the `Handle` property, it throws an `IsolatedStorageException`.

You do not create an `IsolatedStorageFileStream` through an `IsolatedStorageFile` object as you may expect; instead, you pass the `IsolatedStorageFile` instance in which you want to create or open the file to one of the `IsolatedStorageFileStream` constructors. The `IsolatedStorageFileStream` class provides eight overloaded constructors. We show the most specific `IsolatedStorageFileStream` constructor signature here; the other seven variations take subsets of the arguments required by this version and provide defaults for the arguments not specified:

```
# C#
```

```
public IsolatedStorageFileStream(
    string path,
    FileMode mode,
    FileAccess access,
    FileShare share,
    int bufferSize,
    IsolatedStorageFile isf
);
```

```
# Visual Basic .NET
```

```
Public Sub New( _
    ByVal path As String, _
    ByVal mode As FileMode, _
    ByVal access As FileAccess, _
    ByVal share As FileShare, _
    ByVal bufferSize As Integer, _
    ByVal isf As IsolatedStorageFile _
)
```

All of the arguments to the `IsolatedStorageFileStream` constructor serve the same purpose as those to the `FileStream` constructor, and we do not cover them in detail here. Of special interest are the first and last arguments: `path` and `isf`. The `path` argument is a `String` that contains the name of the file to create and must include the full path (within the store) of the file. If you create a file in a directory, the directory must exist before you create the file. The `isf` argument is a reference to the `IsolatedStorageFile` in which to create or open the file.



Some constructors of the `IsolatedStorageFileStream` class do not require you to obtain an `IsolatedStorageFile` object first. When you call them, they automatically open a store isolated by user, assembly, and application domain as if you had first called the `GetUserAssemblyForDomain` method. Refer to the .NET Framework SDK documentation for details on the available constructors.

After creating or opening a file in a store, you can use the `IsolatedStorageFileStream` object as you would with any other `FileStream` object. [Example 11-2](#) demonstrates two methods that demonstrate the simplicity with which you can read and write XML data to isolated storage—a common approach when storing user and application data. In `WriteConfig`, we demonstrate the creation of an XML file named `Config.xml`, to which we write the contents of a `System.Xml.XmlDocument`. In the `ReadConfig` method, read the `Config.xml` file and return a new `XmlDocument` created from the file's contents. Consult the .NET Framework SDK documentation for details of the `XmlDocument` class.

Example 11-2. Reading and writing XML data to isolated storage

```
# C#

public static void WriteConfig(IsolatedStorageFile store,
    XmlDocument xml) {

    //Ensure there is a config directory
    store.CreateDirectory("/config");

    // Create or Open the Config.xml file
    IsolatedStorageFileStream str = new IsolatedStorageFileStream(
        "/config/Config.xml",
        FileMode.OpenOrCreate,
        FileAccess.Write,
        FileShare.None,
        100,
        store
    );

    // Write the XmlDocument to the isolated storage file
    xml.Save(str);

    // Close the Config.xml file
    str.Close( );
}

public static XmlDocument ReadConfig(IsolatedStorageFile store) {

    // Open the Config.xml file
    IsolatedStorageFileStream str = new IsolatedStorageFileStream(
        "config/Config.xml",
        FileMode.Open,
        FileAccess.Read,
        FileShare.None,
```

```
        100,
        store
    );

    // Create a new XmlDocument and load the config file
    XmlDocument xml = new XmlDocument( );
    xml.Load(str);

    // Close the Config.xml file
    str.Close( );

    // return the XmlDocument
    return xml;
}

# Visual Basic .NET

Public Shared Sub WriteConfig(ByVal store As IsolatedStorageFile, _
    ByVal xml As XmlDocument)

    'Ensure there is a config directory
    store.CreateDirectory("/config")

    ' Create or Open the Config.xml file
    Dim str As IsolatedStorageFileStream = New IsolatedStorageFileStream( _
        "/config/Config.xml", FileMode.OpenOrCreate, FileAccess.Write, _
        FileShare.None, 100, store)

    ' Write the XmlDocument to the isolated storage file
    xml.Save(str)

    ' Close the Config.xml file
    str.Close( )
End Sub

Public Shared Function ReadConfig(ByVal store As IsolatedStorageFile) _
    As XmlDocument

    ' Open the Config.xml file
    Dim str As IsolatedStorageFileStream = New IsolatedStorageFileStream( _
        "config/Config.xml", FileMode.Open, FileAccess.Read, FileShare.None, 100, _
        store)

    ' Create a new XmlDocument and load the config file
    Dim xml As XmlDocument = New XmlDocument( )
    xml.Load(str)

    ' Close the Config.xml file
    str.Close( )

    ' return the XmlDocument
    Return xml
End Function
```


End Function

11.2.5 Searching for Files and Directories

Isolated storage does not provide any mechanism through which you can work directly with objects that represent the individual directories within a store, but does provide search capabilities that simplify the discovery of files and directories.

The `GetDirectoryNames` and `GetFileNames` methods both return a `String` array containing a list of files or directories contained in a store with names that match a specified pattern. Both methods take a single `String` argument containing the pattern for which to search. The `GetDirectoryNames` and `GetFileNames` methods have the following signatures:

```
# C#

public string[] GetDirectoryNames(
    string searchPattern
);

public string[] GetFileNames(
    string searchPattern
);

# Visual Basic .NET

Public Function GetDirectoryNames( _
    ByVal searchPattern As String _
) As String( )

Public Function GetFileNames( _
    ByVal searchPattern As String _
) As String( )
```

The `searchPattern` argument must specify the full path of the directory in which you want to search and can include both single-character ("?") and multicharacter ("*") wildcards, but only in the final element. [Table 11-4](#) lists some example values of `searchPattern` for both the `GetDirectoryNames` and `GetFileNames` methods and describes their effects.

Table 11-4. Example `searchPattern` values

Values	Description
<code>GetDirectoryNames</code>	
"*"	Returns the names of all directories contained in the root directory
"/data/*"	Returns the names of all directories contained in the <i>data</i> directory
"/data?/*"	Error; wildcards can occur only in the last element

Values	Description
<code>GetFileNames</code>	
"*"	Returns the names of all files contained in the root directory
"config/*.xml"	Returns the names of all <i>.xml</i> files contained in the <i>config</i> directory
"data/SomeApp.???"	Returns the names of all files in the <i>data</i> directory with the name <i>SomeApp</i> and any three-letter extension

We demonstrate the use of both the `GetDirectoryNames` and `GetFileNames` methods in the following section.

11.2.6 Deleting Files and Directories

You delete files and directories from a store through the methods of the `IsolatedStorageFile` object that represents the containing store. Both the `DeleteDirectory` and `DeleteFile` methods take a single `String` argument that specifies the fully qualified name of the file to delete:

```
# C#

public void DeleteDirectory(
    string dir
);

public void DeleteFile(
    string file
);

# Visual Basic .NET

Public Sub DeleteDirectory( _
    ByVal dir As String _
)

Public Sub DeleteFile( _
    ByVal file As String _
)
```

If you try to delete a directory that does not exist or you include wildcard characters in the directory name, `DeleteDirectory` throws an `IsolatedStorageFile` exception. Attempting to delete a file that does not exist also results in an `IsolatedStorageFile` exception, but including wildcard characters in the filename will cause a `System.Argument` exception.

The major difficulty in using `DeleteDirectory` is that the directory must be empty of files and subdirectories before you can delete it. Because you cannot specify wildcard characters to delete groups of files and directories, you have to implement the program logic necessary to delete all of the files in the directory. [Example 11-3](#) contains a static method named `DeleteDirectory`, which demonstrates the techniques necessary to delete an isolated storage directory tree. `DeleteDirectory` takes two arguments: an `IsolatedStorageFile` object that represents the store

containing the directory to delete, and a `System.String`, which specifies the fully qualified name of the directory to delete—for example, `"/config/gary"`:

Example 11-3. Deleting an isolated storage directory tree

```
# C#

public static void DeleteDirectory(IsolatedStorageFile store, String root) {

    // Ensure we have a directory name ending in a "/" character
    String dir = root.EndsWith("/") ? root : root + "/";

    // Get a list of all files in the current directory, iterate
    // through them and delete them.
    foreach (String file in store.GetFilesNames(dir + "*")) {

        store.DeleteFile(file);
    }

    // Get a list of all sub-directories in the current directory,
    // iterate through them and delete them.
    foreach (String subdir in store.GetDirectoryNames(dir + "*")) {

        DeleteDirectory(store, dir + subdir + "/");
    }

    // Delete the current directory
    store.DeleteDirectory(dir);
}

# Visual Basic .NET

Public Shared Sub DeleteDirectory(ByVal store As IsolatedStorageFile, _
ByVal root As String)

    ' Ensure we have a directory name ending in a "/" character
    Dim dir As String = IIf(root.EndsWith("/"), root, root & "/")

    ' Get a list of all files in the current directory, iterate
    ' through them and delete them.
    Dim file As String
    For Each file In store.GetFilesNames(dir + "*")

        store.DeleteFile(file)
    Next

    ' Get a list of all sub-directories in the current directory,
    ' iterate through them and delete them.
    Dim subdir As String
    For Each subdir In store.GetDirectoryNames(dir + "*")
```

```

        DeleteDirectory(store, dir + subdir + "/")
    Next

    ' Delete the current directory
    store.DeleteDirectory(dir)
End Sub

```

11.2.7 Deleting Stores

To delete a store, use the `Remove` method of the `IsolatedStorageFile` class. `Remove` has two overloads, with the signatures shown here:

```

# C#
public override void Remove( );

public static void Remove(
    IsolatedStorageScope scope
);

# Visual Basic .NET

Overrides Overloads Public Sub Remove( )

Overloads Public Shared Sub Remove( _
    ByVal scope As IsolatedStorageScope _
)

```

The first overload is an instance method that takes no arguments and simply deletes the store represented by the current `IsolatedStorageFile` object. If you try to use the `IsolatedStorageFile` object after calling `Remove`, a `System.InvalidOperationException` exception is thrown because the store is considered to be closed. The second `Remove` overload is a static method that takes an `IsolatedStorageScope` argument, which represents the isolation scope of the stores to delete. [Table 11-5](#) lists the valid combination of `IsolatedStorageScope` values you can use for the `scope` argument.

Once you delete a store, its contents are not recoverable.

Table 11-5. Valid `IsolatedStorageScope` values for `GetEnumerator`

Values	Description
<code>User</code>	Removes all nonroaming stores for the current user
<code>Roaming User</code>	Removes all roaming stores for the current user

[\[Team LiB \]](#)

[\[Team LiB \]](#)

11.3 Administering Isolated Storage

There are two aspects to the administration of isolated storage:

- Configuring security policy to control access to isolated storage
- Managing existing stores

In the following sections, we discuss the administration of isolated storage, beginning with the configuration of security policy.

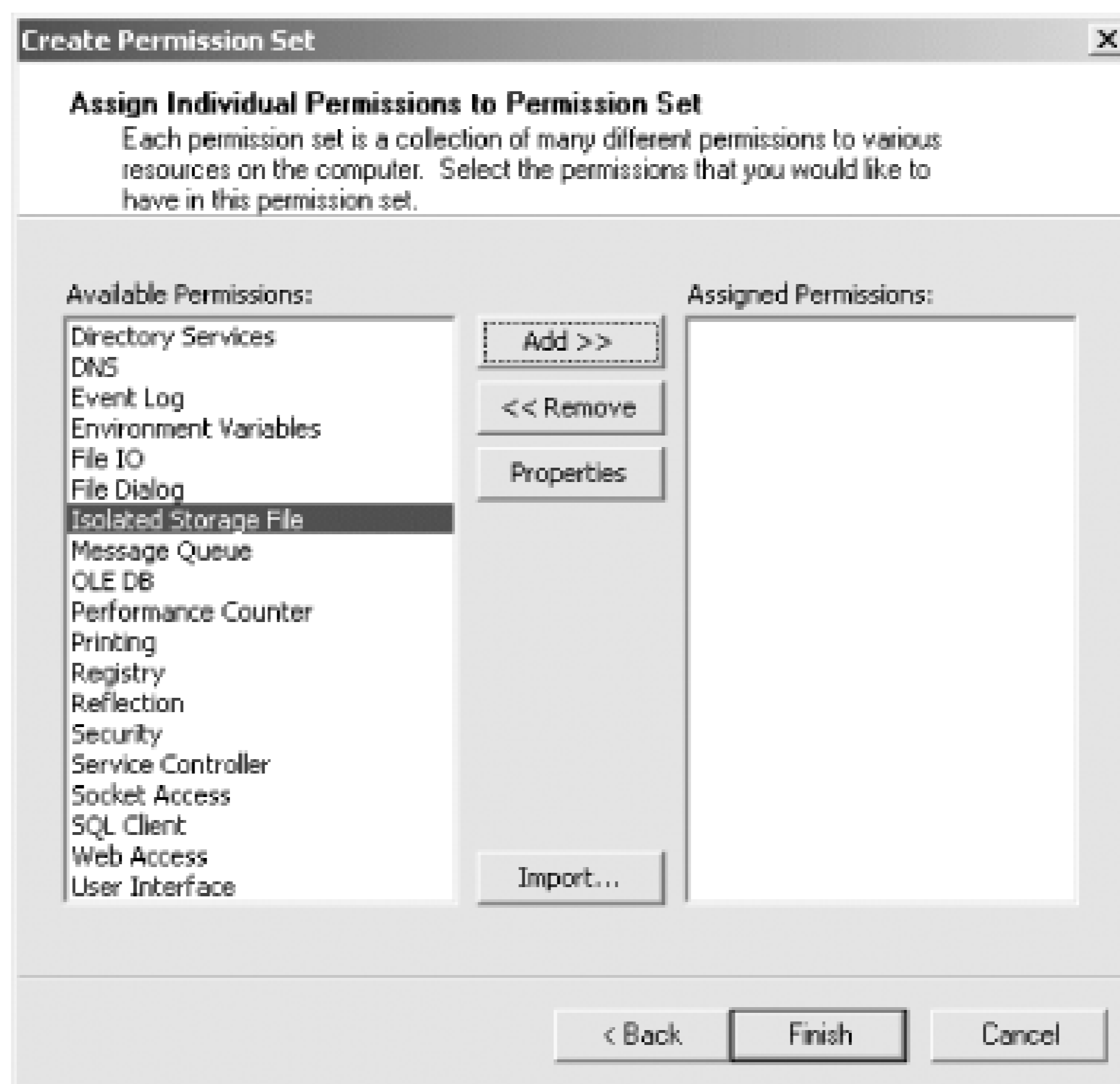
11.3.1 Configuring Security Policy

You can use both the .NET Framework Configuration tool (*Mscorcfg.msc*) and the Code Access Security Policy tool (*Caspol.exe*) to administer security policy to grant access to isolated storage. We discussed the general use of *Mscorcfg.exe* and *Caspol.exe* in [Chapter 9](#). In the following sections, we describe how to use *Mscorcfg.exe* and *Caspol.exe* specifically to control access to isolated storage.

11.3.1.1 Granting isolated storage permissions with Mscorcfg.msc

To grant code access to isolated storage, you must create a named permission set that grants its members the `IsolateStorageFilePermission` class, and you must assign this permission set to a code group. To create a permission set containing `IsolateStorageFilePermission`, you should follow the process we described in [Section 11.1.1.1](#) of [Chapter 9](#). [Figure 11-3](#) shows the dialog box through which you select the permissions to add to the new permission set. The Isolated Storage File entry (highlighted in the diagram) represents the `IsolateStorageFilePermission` class.

Figure 11-3. Selecting IsolatedStorageFilePermission graphically



When you press the Add >> button to include the `IsolateStorageFilePermission` in your permission set, the dialog box shown in [Figure 11-4](#) appears and allows you to configure the specific level of isolated storage access to grant.

Figure 11-4. Configuring `IsolatedStorageFilePermission` graphically

You can grant unrestricted access to isolated storage by pressing the bottom radio button, or grant a specific level of access by pressing the top radio button and choosing the desired level from the Usage Allowed drop-down list. The configuration maps to the values of the `IsolatedStorageContainment` enumeration, which we discussed in [Section 11.2.1](#) earlier in this

chapter.

11.3.1.2 Granting isolated storage permissions with Caspol.exe

In [Chapter 9](#), we explained how to use *Caspol.exe* to administer code-access security. To create a named permission set that grants access to isolated storage, you must generate an XML file that contains the description of a `System.Security.NamedPermissionSet` object containing a configured `IsolatedStorageFilePermission` object. Because of the complexity of the XML description, we recommend that you don't try to create these XML files manually; you should create the permission set programmatically, and then write its XML description to a file, as demonstrated in [Example 11-4](#):

Example 11-4. Programmatically creating an XML description of a permission set

```
# C#

using System;
using System.IO;
using System.Security;
using System.Security.Permissions;

public class WritePermSet {

    public static void Main( ) {

        // Create an empty NamedPermissionSet
        NamedPermissionSet nps =
            new NamedPermissionSet("IsoTestSet",PermissionState.None);

        // Create an IsolatedStorageFilePermission and add it to the
        // NamedPermissionSet
        IsolatedStorageFilePermission p =
            new IsolatedStorageFilePermission(PermissionState.None);
        p.UsageAllowed = IsolatedStorageContainment.DomainIsolationByUser;
        p.UserQuota = 2048;
        nps.AddPermission(p);

        // Write the NamedPermissionSet to a file
        StreamWriter sw = new StreamWriter("IsoTestSet.xml");
        sw.Write(nps.ToString( ));
        sw.Close( );
    }
}

# Visual Basic .NET

Imports System
Imports System.IO
Imports System.Security
Imports System.Security.Permissions
```

```

Public Class WritePermSet

    Public Shared Sub Main( )

        ' Create an empty NamedPermissionSet
        Dim nps As NamedPermissionSet = _
            New NamedPermissionSet("IsoTestSet",PermissionState.None)

        ' Create an IsolatedStorageFilePermission and add it to the
        ' NamedPermissionSet
        Dim p As IsolatedStorageFilePermission = _
            New IsolatedStorageFilePermission(PermissionState.None)
        p.UsageAllowed = IsolatedStorageContainment.DomainIsolationByUser
        p.UserQuota = 2048
        nps.AddPermission(p)

        ' Write the NamedPermissionSet to a file
        Dim sw As StreamWriter = new StreamWriter("IsoTestSet.xml")
        sw.Write(nps.ToString( ))
        sw.Close( )
    End Sub
End Class

```

Running this utility produces a file named *IsoTestSet.xml* that contains the following XML:

```

<PermissionSet class="System.Security.NamedPermissionSet"
    version="1"
    Name="IsoTestSet">
    <IPermission class="System.Security.Permissions.IsolatedStorageFilePermission,
mscorlib, Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
        version="1"
        Allowed="DomainIsolationByUser"
        UserQuota="2048"/>
</PermissionSet>

```

You can use the *IsoTestSet.xml* file to create a new permission set in the machine policy level using the following command:

```
caspol -machine -addpset IsoTestSet.xml
```

11.3.2 Managing Isolated Storage Stores

The Isolated Storage tool (*Storeadm.exe*) is a command-line tool that comes with the .NET Framework SDK and is located in the */bin* subdirectory of the SDK installation directory. *Storeadm.exe* allows you to list or remove the stores of the current user; no tool lets you manage isolated storage for all users, and you must manage the user's roaming and nonroaming stores independently.

To list the nonroaming stores of the current user, use the command `storeadm /list`. Specifying the `/roaming` flag lists the roaming stores—for example, `storeadm /list /roaming`. Either of these commands displays a list of stores but not content, similar to that shown here. We have abbreviated

the `StrongName.Key` and `Publisher.X509Certificate` elements in the interest of brevity:

Microsoft (R) .NET Framework Store Admin 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

Record #1

[Assembly]

```
<StrongName version="1"
  Key="002400000480000094000000060200000024000052534131
00040000010001005308669520E62E2E1F08F07B0D5DA0B33C0F82AC093C54E26
852080E8C5B7"
  Name=" SomeOtherApp"
  Version="1.5.0.0"/>
```

Size : 0

Record #2

[Domain]

```
<System.Security.Policy.Url version="1">
  <Url>file://C:/Development/projects</Url>
</System.Security.Policy.Url>
```

[Assembly]

```
<System.Security.Policy.Publisher version="1">
  <X509v3Certificate>308201713082011BA0030201020210E5D7C7E57FD9B9B347A7EE
AC3221100A39899EB866979DBBB6430F5A52A024FC7FCEE9</X509v3Certificate>
</System.Security.Policy.Publisher>
```

Size : 0

The example output contains information about two stores: **Record #1** and **Record #2** (which we have highlighted in boldface). **Record #1** represents a store isolated by user and assembly; the assembly's `StrongName` evidence identifies the creating code. **Record #2** represents a store isolated by user, assembly, and application domain; the application domain's `Url` evidence and the assembly's `Publisher` evidence identify the creating code.

Storeadm.exe also allows you to remove stores for the current user. However, you must remove all roaming or nonroaming stores at once; there is no way to remove individual stores. To remove all nonroaming stores, use the command `storeadm /remove`. To remove all roaming stores, use the command `storeadm /roaming /remove`.

Storeadm.exe will not prompt you to confirm the removal of stores, and you cannot recover any stores once you remove them.

[[Team LiB](#)]

[\[Team LiB \]](#)

Part III: .NET Cryptography

Provides a description of modern cryptographic techniques and details the implementation of these techniques provided by the .NET Framework class library. These chapters demonstrate the use of each implementation and show you how to extend the functionality of the .NET class library by implementing your own cryptographic algorithms:

[Chapter 12](#)

[Chapter 13](#)

[Chapter 14](#)

[Chapter 15](#)

[Chapter 16](#)

[Chapter 17](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 12. Introduction to Cryptography

The previous chapters shown how to apply security to an application; the topics that follow demonstrate how to use cryptography to secure the data that your applications use.

We begin with an overview of cryptography and discuss some of the dangers and limitations of which you should be aware when putting it to use. Consider this chapter as a simple primer for those who have not previously encountered cryptography. In the chapters that follow, we examine specific aspects of cryptography in detail, and illustrate how to use and extend the .NET Framework's cryptographic support.

You should seriously consider using cryptography in any application that processes sensitive or valuable data. However, contrary to common opinion, cryptography itself is vulnerable to attack and is not something that can be "plugged into" an application and then forgotten. Throughout this book, we have shown you that software security is an ongoing process that requires careful planning and management. This is especially true for cryptography, where it is easy to create disastrous mistakes.

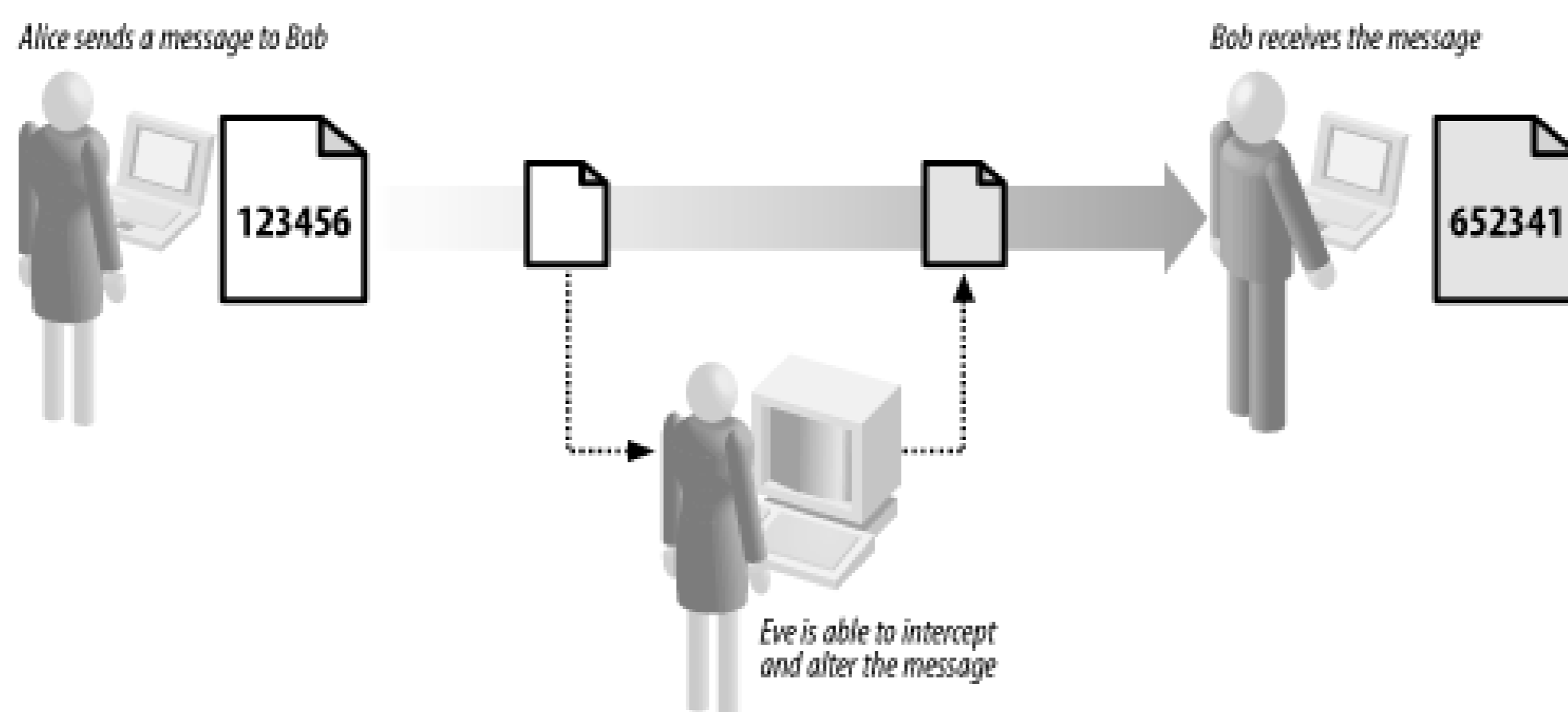
[\[Team LiB \]](#)

[[Team LiB](#)]

12.1 Cryptography Explained

The most common way to describe cryptographic systems is to assume that one person (usually named Alice) wants to send a message to another person (Bob). However, a third person (Eve) is determined to eavesdrop on their conversation. [Figure 12-1](#) shows the relationship between Alice, Bob, and Eve.

Figure 12-1. Alice, Bob, and Eve are commonly used to illustrate cryptography.



Let's assume that Eve will intercept and alter messages between Alice and Bob irrespective of how those messages are composed and sent. Although this might seem paranoid, there are benefits to this way of thinking. By assuming that Eve can intercept each message, you are, in effect, forced to regard the communications channel as insecure—this is a realistic approach because communications techniques that seem secure today may be compromised in the future. By assuming that Eve can read any message, we focus on the security of the message itself.

Cryptography has a long history, and different techniques for protecting messages have been in use for thousands of years. For most of that time, Alice, Bob, and Eve have represented real people or organizations. With advances in technology, cryptography has become irrevocably bound to the world of computers. Computers have facilitated stronger cryptographic techniques but also allowed people like Eve to mount stronger attacks on cryptography. Although we find it useful to talk about Alice, Bob, and Eve, these names are often a metaphor for computers.

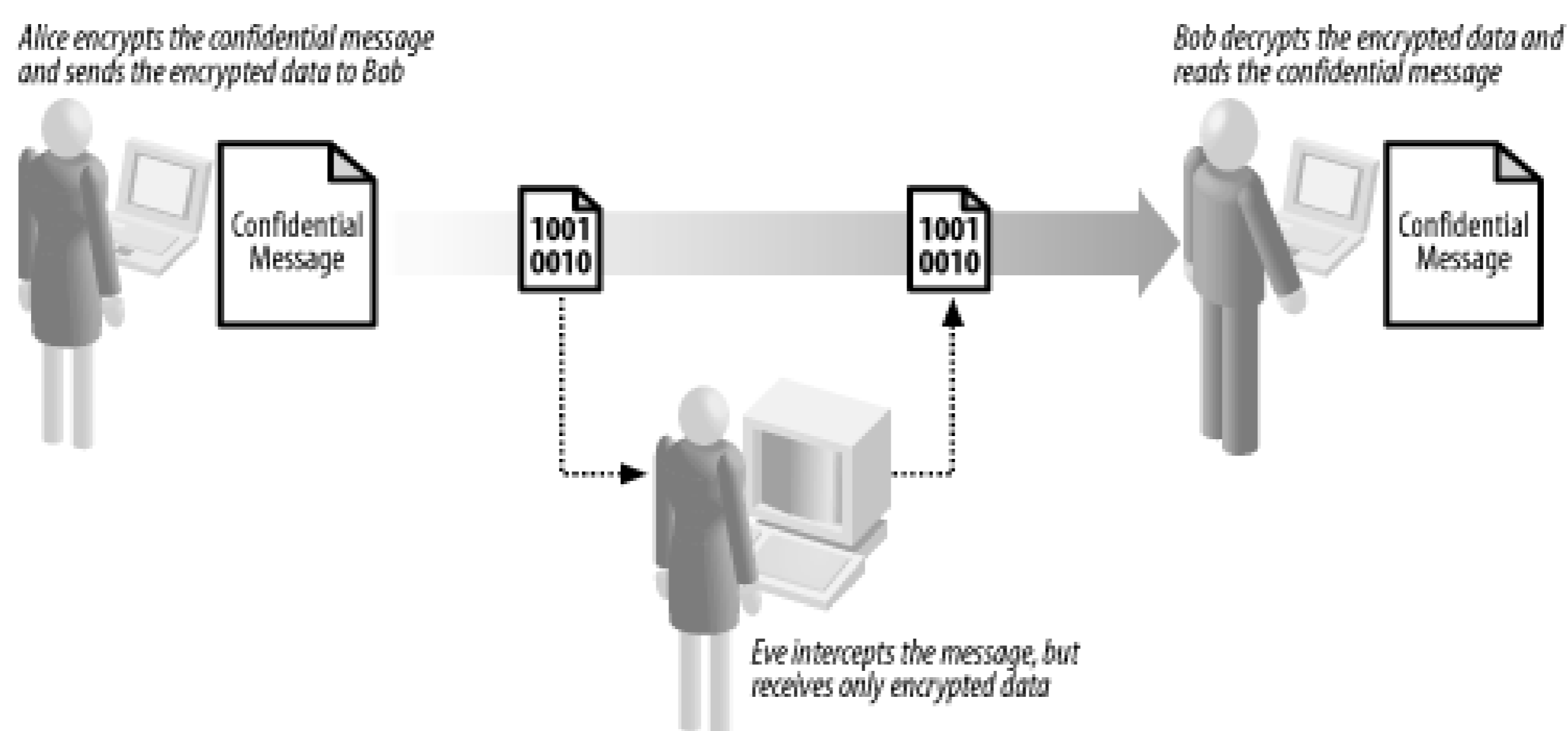
When we say "Alice sends a message to Bob," we do not consider the details of how she does this, and as a result, we will be discussing techniques that are applicable in many different situations. For example, Alice may use her email client to compose and send a message, or she may have written an application that will communicate automatically with her bank to pay her bills while she is on vacation. Irrespective of how Alice communicates with Bob, her need for cryptography can be broken down into the three categories that we discuss in [Section 12.1.1](#), [Section 12.1.2](#), and [Section 12.1.3](#).

12.1.1 Confidentiality

Confidentiality becomes an issue when Alice wants to send a message containing a secret to Bob. We assume that Eve can intercept and read the message, and that Alice and Bob do not want Eve to know the secret contained in the message.

The cryptographic solution to this problem is to obscure the contents of the message using encryption. Encryption ensures that Eve cannot understand the messages sent between Alice and Bob, even though she is able to intercept them. [Figure 12-2](#) illustrates the role of encryption in confidentiality.

Figure 12-2. Alice uses encryption to ensure that her messages to Bob remain confidential



Encryption works on the basis that there is one piece of information that Eve has not been able to acquire, known as the *key*. The key is used as part of the encryption process and is kept secret. Alice selects an encryption algorithm and uses the secret key to create the encrypted data, as shown in [Figure 12-3](#). When Bob receives the encrypted text, he uses the secret key to restore the confidential message so that he can read it.

Figure 12-3. Alice uses the confidential message and the secret key to create the encrypted data.

Using encryption changes the nature of the confidentiality problem-Alice and Bob can exchange confidential messages as long as Eve does not acquire the secret key. When Alice and Bob exchange messages, they send the encrypted data only-they never send the key, because they know that Eve can intercept it and then read their messages.

This is the very essence of encryption-Alice uses one secret that Eve does not know (the key) to protect another (the confidential message). However, Eve can read the confidential messages if she discovers or guesses the key.

In practice, using encryption becomes centered on protecting and managing these secret keys. Some types of encryption require Alice and Bob to know the key and are called *symmetric* encryption (because Alice and Bob have the same knowledge). The problem with symmetric encryption is that Alice and Bob need to agree on what the secret key will be before sending any messages.

Another approach is to use *asymmetric* encryption, where only Bob has to keep a secret. Bob creates a special pair of keys, one of which he keeps secret (known as the *private key*) and one that is given out to anyone who wants to send him a message (the *public key*). Bob can send Alice his public key openly, because he does not care if Eve intercepts it.

When Alice wants to send Bob a confidential message, she uses an asymmetric encryption technique to encrypt the data. This is similar to the model shown in [Figure 12-3](#); but instead of using the secret key, Alice uses Bob's public key and the contents of the message as inputs.

When Bob receives the message, he decrypts it using his private key (which he always keeps secret) and restores the confidential message from the encrypted data. Eve cannot read the message unless she acquires Bob's secret key. Using this technique, Alice and Bob do not have to agree on a key that they both know, further reducing the chances that Eve can acquire the key during the agreement process. The use of asymmetrical encryption is also known as *public key cryptography*.

We discuss the .NET Framework's support for symmetric encryption in [Chapter 14](#), and provide more information about public key cryptography in [Chapter 15](#) and [Chapter 16](#).

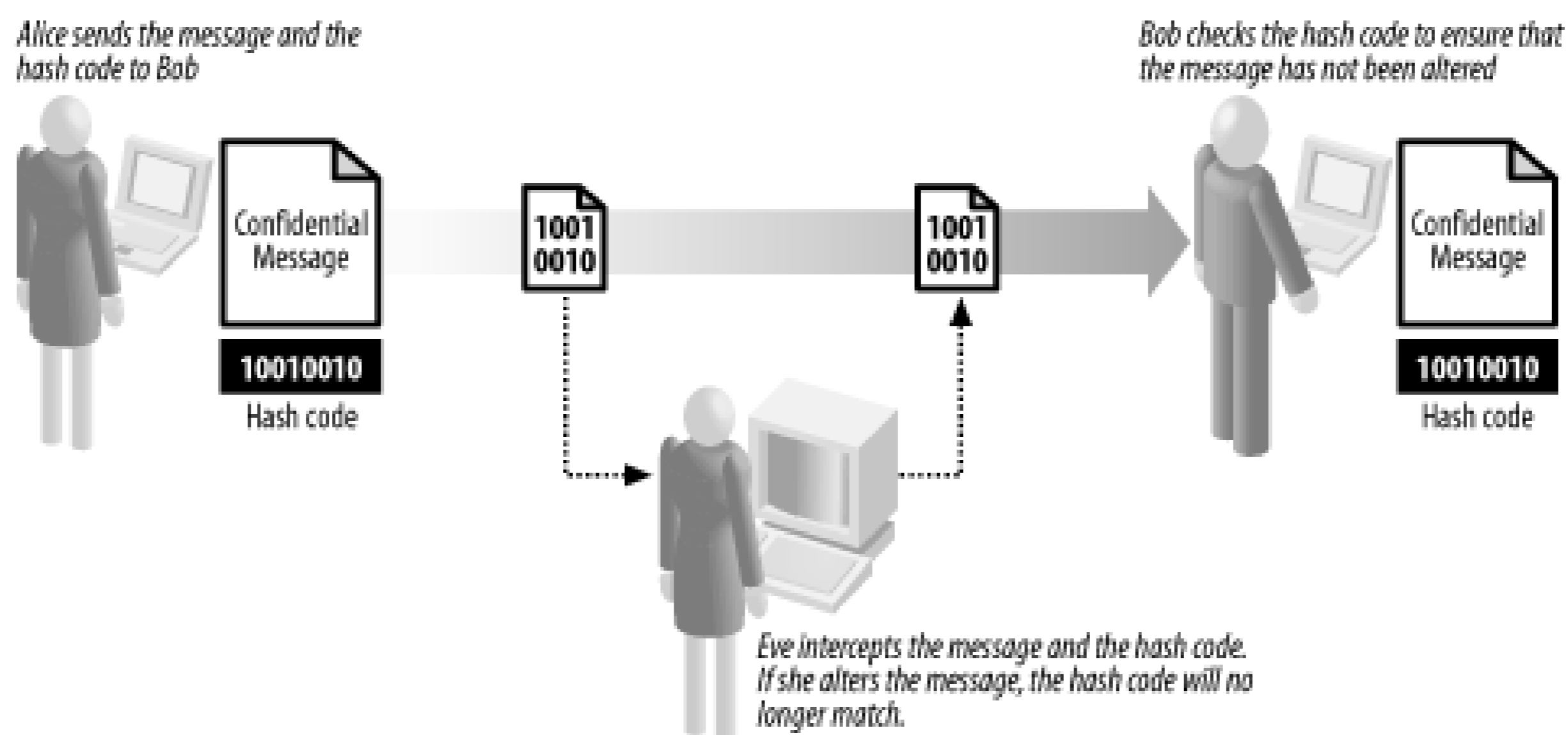
12.1.2 Integrity

Integrity becomes an issue when Alice wants to send a message to Bob but is concerned that Eve will tamper with the message and change the contents. In this case, Alice does not care if Eve can read the message-she only wants to make sure that Bob can detect any changes made by Eve.

Alice takes the message that she has written and creates a cryptographic hash code, which is a numeric code based on the contents of the message that acts as a "fingerprint" to identify the message. Changing even the smallest portion of the message results in a different hash code. Alice sends the message and the hash code to Bob.

Bob takes the message that he has received and creates his own hash code using the same technique that Alice used. If his code matches the one that Alice sent, Bob can assume that Eve has not modified the message-if the hash codes do not match, then Bob can assume that Eve has altered the message en route. [Figure 12-4](#) illustrates the use of hash codes.

Figure 12-4. Alice and Bob use hash codes to check the integrity of their messages



Eve will be able to change the message without detection if she changes the hash code as well. Eve creates a new message and a new hash code, both of which she sends to Bob. When Bob creates a hash code for the message, it will match the hash code he has received, leading him to think that the message has not been tampered with.

The solution to this problem is to use a "keyed" hash code, which uses the contents of the message and a secret key to create the hash code, as illustrated in [Figure 12-5](#). Eve can still modify the message, but she can no longer create a valid hash code, because she lacks the key used to create the original code.

Figure 12-5. Keyed hash codes are created using the contents of the message and a secret key.

Unless Eve is able to discover the key, she will be unable to create hash codes that will fool Bob; if Eve acquires the key, she will be able to generate valid hash codes for her own messages, rendering Alice and Bob's efforts to ensure message integrity useless. As with confidentiality, the issue of integrity becomes one of protecting secret keys from Eve. We discuss the .NET Framework support for cryptographic hash codes in [Chapter 13](#).

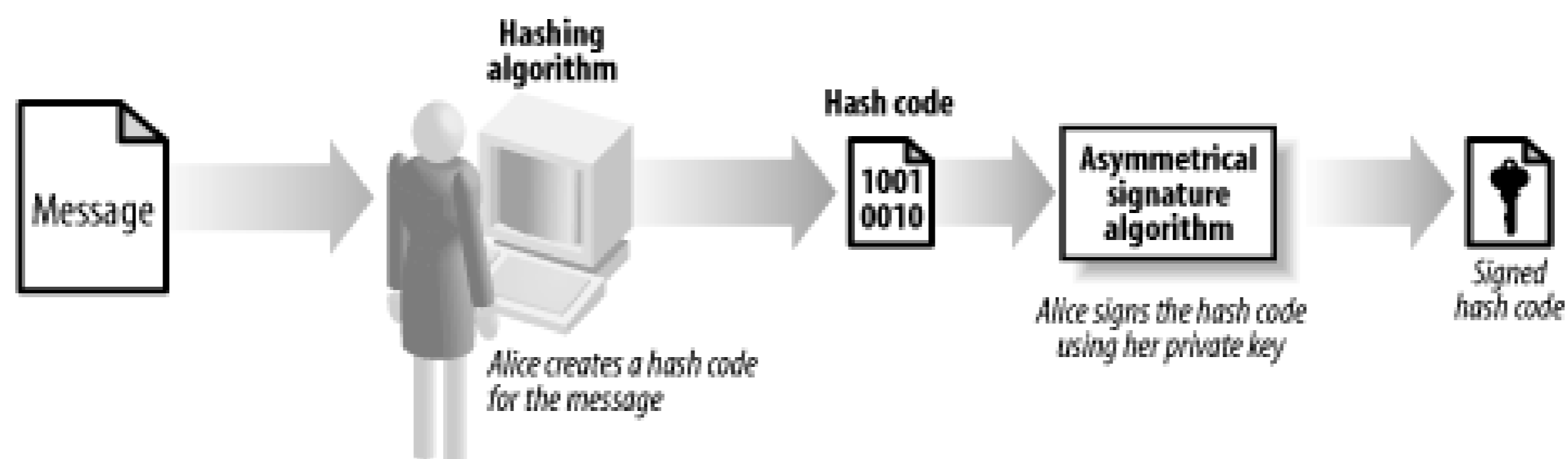
12.1.3 Authentication

The goal of authentication is to allow Bob to establish that Alice is the author of a message. For our purposes, this means that Alice should be able to create a "digital signature" for the message and that Bob should be able to check the signature to ensure that it is valid. In addition, Eve should not be able to create and sign messages as though she were Alice.

Digital signatures rely on asymmetric encryption techniques, although they are applied differently than we discussed earlier. Alice creates a pair of keys, one of which she makes public and one of

which she keeps private. To sign the message, Alice creates a cryptographic hash code of the message that she wants to send to Bob, as discussed earlier. Alice then signs the hash code using her private key, as shown in [Figure 12-6](#). This creates a digital signature that is unique to the combination of the document and Alice's private key.

Figure 12-6. Alice uses public key cryptography to create a digital signature



When Bob receives the message, he verifies the signature using Alice's public key. If the signature is valid, then Alice has "signed" the message, and Bob can assume that Eve has not forged the message.

Because digitally signing a document requires the creation of a hash code, it also addresses the issue of integrity that we discussed earlier. If Eve modifies the message that Alice has sent, then Alice's signature will not be valid. Bob will generate his own hash code for the message, and it will not be the same as the one Alice signed. [Figure 12-7](#) shows the protocol for message signatures.

Figure 12-7. Alice uses a digital signature to authenticate a message

Eve is unable to forge messages from Alice, because she does not know Alice's secret key and is unable to create digital signatures that will fool Bob. However, as with the areas of confidentiality and integrity, if Eve is able to acquire the secret key, she will be able to sign messages as though she were Alice.

[[Team LiB](#)]

[\[Team LiB \]](#)

12.2 Cryptography Is Key Management

It should be evident from the earlier sections that although cryptography can be classified according to whether it will be used to provide confidentiality, integrity and authentication, each of these approaches relies on the protection of secret keys to work properly.

For symmetrical encryption and keyed hash codes, Alice and Bob must agree on a key before sending any messages and ensure that the key does not fall into Eve's hands. For asymmetric encryption and digital signatures, only one person has to keep the secret (the private key), which is an improvement over needing an agreed key, but still requires either Alice or Bob to remain vigilant.

Alice and Bob have no way of determining if Eve has been able to learn the secrets that they are protecting. When Eve intercepts a message, she is careful to send on a copy (or a replacement message) to Bob. If Eve has learned the keys, then she is able to impersonate Alice or Bob and interfere with their messages at will.

To minimize the damage Eve can cause if she discovers the secret keys, Alice and Bob may agree to change their keys regularly. Eve may be able to obtain or guess the key, but if Alice and Bob change the keys every week, Eve will only be able to meddle with up to seven days worth of messages before she is back in the dark. Selecting and changing keys is an important process, and the problems of storing and distributing keys securely are two of the most difficult aspects of implementing cryptographic solutions. We will discuss the issues that arise in the following chapters.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

12.3 Cryptographic Attacks

In general, there are three ways to attack cryptography:

1. Acquire or guess the secret key
2. Identify and exploit weaknesses in the cryptographic algorithms
3. Identify and exploit weaknesses in the software that implements the cryptographic solution

We discuss each of these techniques in the following sections, but the key point to remember is that cryptography is vulnerable to attack, and does not provide invincible security. The passage of time weakens cryptographic security. Once Eve has intercepted a message from Alice, she can attack the cryptography at her leisure; therefore, Alice and Bob must assume that at some point Eve will be able to read the contents of that message.

12.3.1 Brute Force Attacks, Theft, and Guessing

A brute force attack is the process of trying every possible key to determine which key Alice used. Modern cryptography is based on mathematical algorithms using numerical keys. The length of the key (the number of digits) determines how many possible keys there can be. If there are only 10 possible keys, Eve has to try to decrypt the message at most 10 times before she finds the key that Alice uses.

The cryptographic algorithms that we discuss in this book have different key sizes, some of them allowing ranges that are so big that it could take many hundreds of thousands of years to check all of the possible keys that Alice may have used. However, Eve has history on her side; the speed at which computers can test keys has increased enormously in recent years, and there is no reason why this should change in the future. Computing-performance increases will allow Eve to check more keys each day, or completely new techniques may be devised for attempting to guess keys, allowing her to test all of the combinations in a fraction of the time. Alternately, Eve may create a global network of computers and coordinate the guesses to speed up the attack using current technology.

Instead of using brute force, Eve may try to steal the key from Alice or Bob, either by tricking them into revealing the secret, or by intimidation and violence. While it is more likely that trickery would be used, we have to bear in mind that some information is of limitless value, and using violence to force Alice or Bob into revealing the key is not inconceivable.

Finally, Eve may just guess the key. Eve has to be lucky only once to be able to beat huge odds and guess the right key; the odds of her guessing correctly are extremely low, but that is the very nature of being lucky.

12.3.2 Cryptanalysis

Cryptanalysis is the study of the algorithms that Alice and Bob have used, looking for weaknesses that can be exploited to reveal the messages or the keys that have been used. By the time that new cryptographic algorithms enter commercial use, they have been subjected to extensive cryptanalysis by security researchers and interested mathematicians, and are generally regarded to contain no obvious flaws.

However, just because there are no publicly known flaws does not mean that none will be discovered in the future or even that some flaws are already known, but they are being kept secret by people like Eve in order to lull Alice and Bob into a false sense of security. Cryptography is a dynamic area; new algorithms are developed and old ones are found to be insecure. Alice and Bob cannot rely on their selected algorithm to remain secure forever.

12.3.3 Software Bugs

Even if Eve is unable to discover the key or find a weakness in the cryptographic algorithm, she may be able to exploit a bug in the software that Alice and Bob use to perform their cryptography.

Although new cryptographic theories are rigorously debated and tested before they are considered to be secure, the software that implements the theory is often untested and insecure. If Eve can find a weakness in the software, she may be able to exploit it to obtain the keys that Alice and Bob are using.

The most common problem with security software is that complex algorithms are not correctly implemented or are used in such a way as to make them much less secure than they should be, as if inadvertently discarding some of the key that Alice has created, thereby reducing the number of guesses that Eve has to make.

When Alice and Bob select cryptographic software, they must be confident that their security will not be compromised by programming errors.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 13. Hashing Algorithms

In the previous chapter, we showed you that Alice and Bob could ensure the integrity of their messages by using data hash codes. Alice creates a hash code, which she sends to Bob along with her message. When Bob receives the message, he creates his own hash code and compares it to the one that he received from Alice. If the two hash codes are the same, then Bob can assume that Eve has not altered the contents of the message.

In this chapter, we examine hash codes in-depth and show you how to create and validate your own hash codes using the .NET Framework classes. We also show you how you can extend the .NET Framework by adding a new hashing algorithm.

[\[Team LiB \]](#)

[[Team LiB](#)]

13.1 Hashing Algorithms Explained

A hashing algorithm creates a hash code, also called a "message digest" or "message fingerprint." As we explained in [Chapter 12](#), hash codes are of limited use for communications security, because Eve can replace both the hash code *and* the message that Bob receives, but they are an essential element of digital signatures, which we discuss in [Chapter 16](#).

In this section, we explain how hashing algorithms work, and provide some practical insight into choosing a suitable algorithm for your project.

13.1.1 Creating a Hash Code

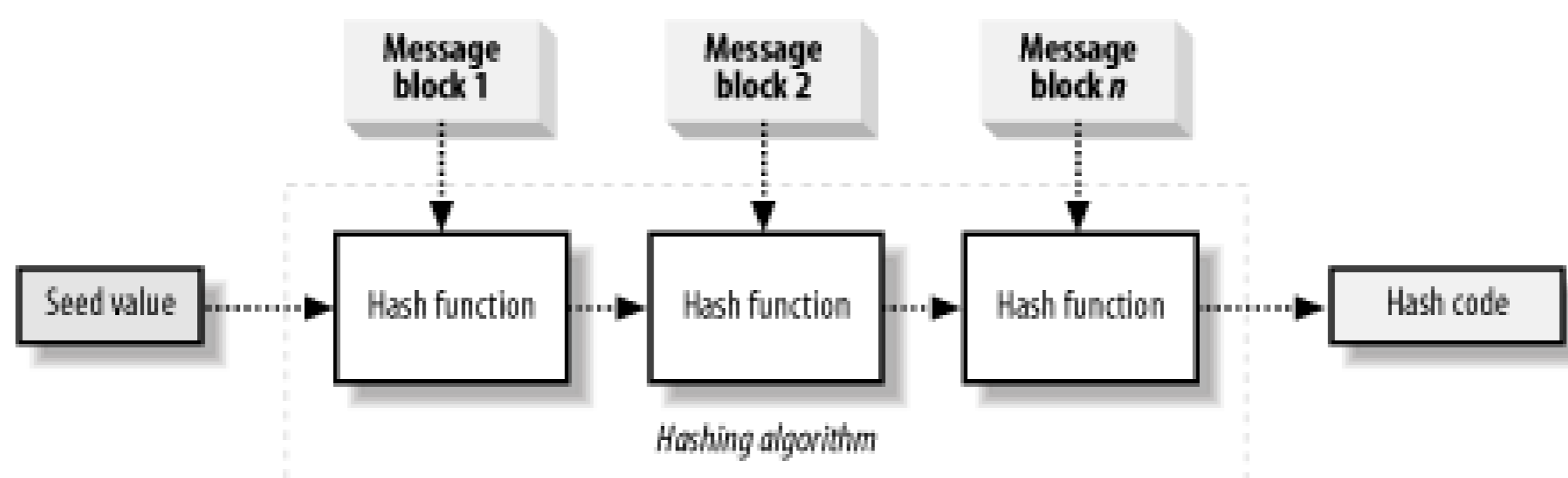
At the heart of a hashing algorithm is a mathematical function that operates on two fixed-size blocks of data to create a hash code, as shown in [Figure 13-1](#).

Figure 13-1. The hash function operates on fixed-size blocks of data

We break up Alice's message into blocks that are the same size as the input for the hash function. The size of each data block varies depending on the algorithm but the blocks tend to be small—the algorithms included in the .NET Framework break the messages into blocks of 512 or 1024 bits. The design of the hash function differs for each hashing algorithm, but they all share the same basic approach.

The algorithm specifies a "seed" value that feeds into the hash function along with the first block of message data, thus producing our first hash code. Take this hash code and feed it into the hash function along with the second block of message data, creating a second hash code. Feed the second hash code into the function along with the third message block, and repeat the process of hashing a data block along with the hash code of the previous block until you have processed all of the message data, as shown in [Figure 13-2](#).

Figure 13-2. Creating the hash code by "chaining" the hash function



The idea of a hash code acting as a message "fingerprint" is reasonable, because making even the smallest change in the message data changes the value of the final hash code. By "chaining" repeated calls to the hashing function, you can create a hash code that relies on the value of every single bit of the message. The output of hashing the first data block becomes an input to the next operation, which will alter the value of the second hash code, which will affect the result of the third operation, and so on. Known as a "ripple effect" or an "avalanche," the huge impact that the smallest change has on the final hash code provides the fingerprint. Two messages that differ by a single bit of data produce very different hash codes.

It is important not to confuse the hashing algorithm with the hash function. The hash function generates a hash code by operating on two blocks of fixed-length binary data. The hashing algorithm describes the process of using the hash function to create a hash code for a message—the algorithm is a protocol for using the hash function, specifying how the message will be broken up and how the results from previous message blocks are chained together.

13.1.2 Message Limits

There is a limit to the size of message that a hashing algorithm can process before the hash code is no longer secure. A very large message causes the initial bytes of message data to have less of an effect on the hash code than the bytes at the end of the message. This means that it is easier for Eve to find a message with the same hash code as Alice sent to Bob, because we have increased the likelihood of being able to find two messages with slightly different initial bytes that result in the same hash code.

The problem arises because the hash function is called so many times for large messages—for example, a 1 GB message will cause the hash function to be called more than 6,000,000 times. So many hashing operations lessen the "ripple effect."

Most hashing algorithms impose a message size limit of 2^{64} bits. This limit is not a practical consideration for most projects but can be an issue when calculating hash codes for data generated over a long period. Some algorithms are able to process 2^{128} bits of data securely.

All of the hashing algorithms in the .NET Framework can handle message limits of at least 2^{64} bits, which is sufficient for most projects. However, we recommend that you bear the limit in mind when considering hashing algorithms provided by third parties.

13.1.3 Hash Code Length

It is common to refer to a hash code as "uniquely" identifying a message, but the reality is slightly different. The reason that hash codes can be used to protect message integrity is that it is very

difficult (but not impossible) to find another message that produces the same hash code.

Irrespective of the size or content of the message processed, hash algorithms produce a fixed-length hash code, meaning that there is a finite set of hash code values. We measure the length of a hash code in binary bits, meaning that a 64-bit hash code can represent 2^{64} different numeric hash values.

You can create an infinite number of different messages, and there are only a finite number of possible hash codes; therefore, it must be possible to find a pair of messages that produce the same hash code. In this section, we consider the security of a 64-bit hash code which was, until quite recently, thought of as very secure.

For Eve to find a message that produces the same hash code that Alice has sent to Bob, we expect that she will have to create and test 2^{64} (18,446,744,073,709,551,616) different messages before she finds a match (assume that hash codes are evenly distributed). If Eve can check 50 messages per second (around what you can achieve using our desktop PCs), then it will take her 11.6 billion years to test enough messages. If Eve were able to access very powerful computers, she might be able to check 10 million messages each second, but we would still expect her to take 58,494 years to find a match. This is the basic security of hash codes; realistically, Eve cannot wait almost 60,000 years, meaning that finding a matching message for a specific hash code is "computationally infeasible"-a phrase that is used by cryptographers to mean "possible, but extremely unlikely."

However, if Eve can control the contents of both messages, then she has a much easier job of finding two matching hash codes, known as a message collision. If Eve wants to find *any* pair of messages that create the same hash code, then she only has to create and test 2^{32} different messages before she has a better than 50% chance of finding two that match. In fact, if she is able to check 50 messages a second, she is likely to find a match in less than 3 years, which is a lot less than the 11.6 billion years it would take to handle 2^{64} messages. If Eve is able to check 10 million messages per second, she is likely to find a pair of messages in a little over 7 minutes.

Eve now has a pair of messages that she has written, both of which result in the same hash code. She cannot use these messages to bother Alice and Bob, because she is unlikely to get Alice to hash either of her messages and send them to Bob. However, Eve can use the following attack to defraud Joe, who happens to owe her some money:

1. Eve creates two versions of a message. The first message instructs Joe's bank to pay Eve \$100, which is the amount he owes her. The second message instructs Joe's bank to pay Eve \$1,000,000.
2. Eve generates 2^{32} variations of the innocuous message by making minor editorial changes that do not affect the meaning of the message (for example, adding spaces or tabs, or altering the formatting).
3. Eve also generates 2^{32} variations of her dishonest message. Because she is in control of the content of both messages, the chances of her finding a version of the innocuous message and the dishonest message with the same hash code are better than 50%.
4. Eve sends the innocuous message to Joe, who creates a hash code and digitally signs it (see [Chapter 16](#) for more information about digital signatures). Joe sends the signed hash code to Eve so that she may present the signed message to his bank.
5. Eve presents her dishonest message to Joe's bank along with the signed hash code. The bank checks the digital signature and finds that the hash code matches the message and that Joe

signed it. Eve was able to forge a signature without learning Joe's secret keys.

Luckily, for Joe at least, most banks would not honor a request for a million dollars in this way. Nonetheless, Eve was able to create a forged message with very little effort—in fact, if Joe has used a 64-bit hash code for his signature, it would have taken Eve somewhere between 7 minutes and 3 years to create the messages. Looking for collisions in this way is a "birthday" attack and drastically reduces the effectiveness of a hashing algorithm, forcing you to treat a hash code as though it was only half of its actual length.

Birthday Attacks

How many people must be in a room before the probability of one of them sharing your birthday is greater than 50%? The answer is 183. How many people have to be in the room before the probability of *any two* people sharing the same birthday is greater than 50%? The answer is 23. It is easier to find any shared birthday than to find a specific one.

Eve is relying on being able to create two messages that produce the same hash code, knowing that it is simpler than trying to find a match for a specific message from Alice. Any attack along these lines is a "birthday attack."

If this kind of attack is possible in your project, then you will need to select an algorithm that creates hash codes with twice the length you might expect. If this kind of attack is not possible, then you should select a hash code based on its actual length. As a rule, a longer hash code will take more time to attack, but takes longer to generate.

13.1.4 The .NET Framework Hashing Algorithms

The .NET Framework includes classes for five different hashing algorithms, although four of them are closely related, being variations of the same basic premise to create hash codes of different length. [Table 13-1](#) lists the hashing algorithms available.

Table 13-1. Summary of .NET Framework hashing algorithms

Name	Input block size	Message limit (bits)	Hash code size (bits)
MD5	512	2^{64}	128
SHA-1	512	2^{64}	160
SHA-256	512	2^{64}	256
SHA-384	1024	2^{128}	384
SHA-512	1024	2^{128}	512

Developed in 1991, MD5 has been used in all sorts of systems and is an integral part of many standards for communications security. MD stands for "Message Digest," and MD5 was the fifth such

algorithm that Rivest had designed.

MD5 is the fastest hashing algorithm included in the .NET Framework, but the relatively small hash code size makes it more susceptible to brute force and birthday attacks.

The National Security Agency (NSA) designed the Secure Hash Algorithm (SHA) in 1993, and the National Institute of Standards and Technology (NIST) published it as Federal Information Processing Standard (FIPS) 180, permitting its use in government projects. NSA withdrew the standard shortly after publication, replacing it with SHA-1, which contained a slight modification to the hash function.

The NSA stated that the change addressed a flaw in the original algorithm that reduced its cryptographic security. The NSA has never described this flaw, leading the paranoid world of cryptographers to spend many thousands of hours analyzing the algorithm, looking for any weaknesses deliberately introduced into SHA-1 to facilitate government skullduggery. To date, no weaknesses have been found, and SHA-1 is considered a secure algorithm.

NIST published FIPS 180-2 in 2001, which defined the SHA-256, SHA-384, and SHA-512 algorithms, named after the length of the hash code that each produces. These new algorithms are variations of SHA-1, but are sufficiently recent that their cryptographic security remains open.

Consider how much you know about an algorithm before selecting one for use in a project. For example, although SHA-256, SHA-384, and SHA-512 have longer hash codes, MD5 and SHA-1 are tried and tested and are known to be reliable. Remember that a longer hash code does not provide greater security if the underlying algorithm is flawed.

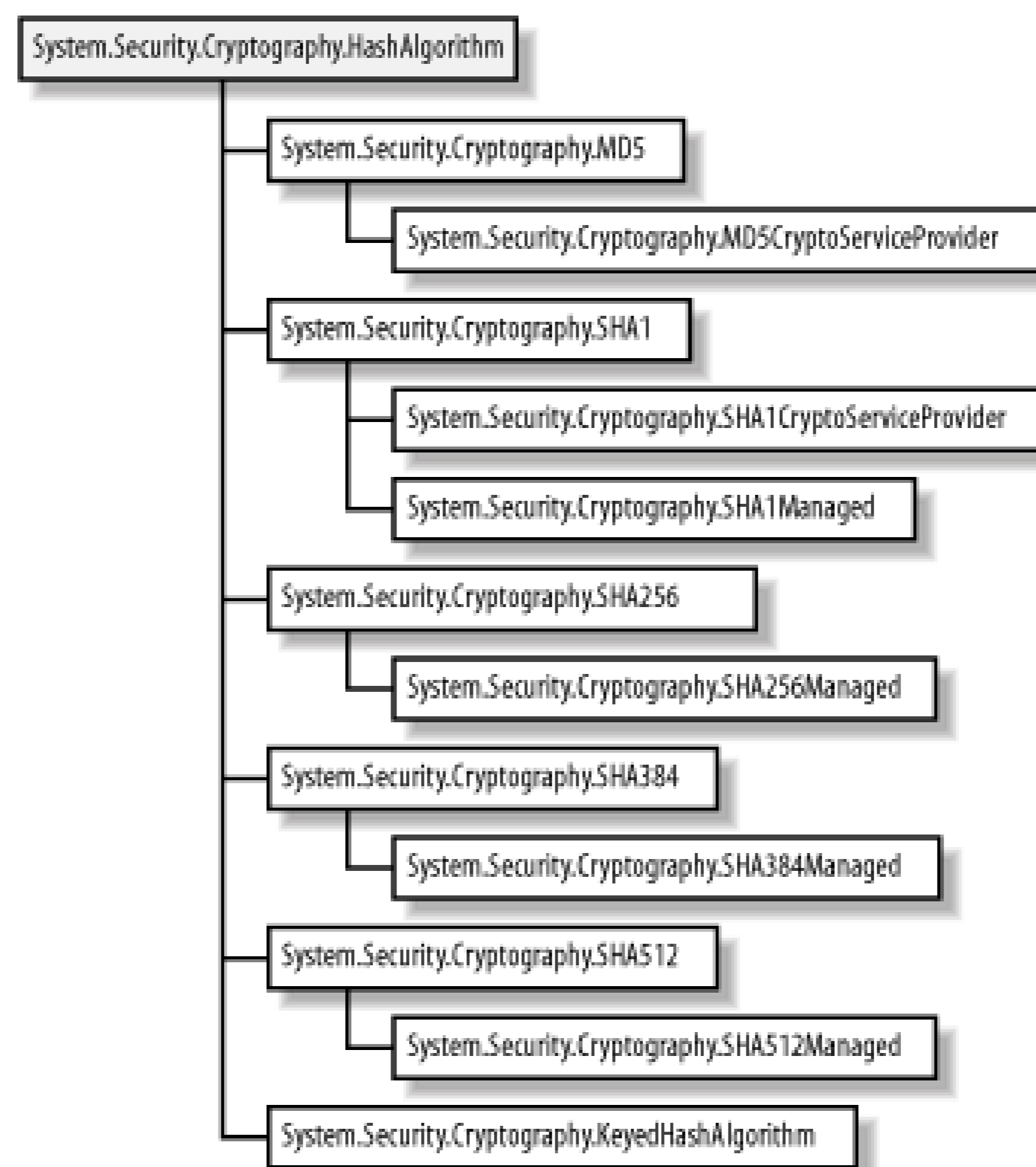
[\[Team LiB \]](#)

[[Team LiB](#)]

13.2 Programming Hashing Algorithms

The .NET Framework represents all hashing algorithms with the abstract `System.Security.Cryptography.HashAlgorithm` class. An abstract class represents each specific algorithm, extended by individual implementation classes, as shown in [Figure 13-3](#). This approach supports selecting between several implementations of the same algorithm.

Figure 13-3. The .NET Framework class hierarchy for hashing algorithms



There are two types of implementation class included with the .NET Framework. Those whose names end with `Managed` (for example, `SHA1Managed`) were written in a managed .NET language (for example, C# or Visual Basic .NET). Those classes whose names end in `CryptoServiceProvider` (for example, `SHA1CryptoServiceProvider`) rely on the Windows Crypto API. Where there is a choice between a managed implementation and a Crypto API implementation, our (unscientific) testing shows that the Crypto API versions are slightly faster and consume less system memory.

The Windows Crypto API

The Crypto API is a component of the Windows operating system that contains cryptographic functionality. Many of the .NET cryptographic classes rely on features of the Crypto API for tasks such as key management or providing implementations of algorithms. As a .NET programmer, you do not need to have an understanding of the details of the Crypto API. We do not explore the Crypto API in this book; for more information, consult the Windows API documentation.

13.2.1 The HashAlgorithm Class

The design of the `HashAlgorithm` class makes it very simple to generate hash codes for any of the hashing algorithms that the .NET Framework supports. In this section, we show you how to create an instance of a given hashing algorithm and the techniques used to create hash codes for different types of data. [Table 13-2](#) lists the members of the `HashAlgorithm` class.

Table 13-2. Members of the HashAlgorithm Class

Member	Description
Properties	
<code>Hash</code>	This property returns the value of the computed hash code. It is not commonly used. See the following sections for instructions on using the <code>ComputeHash</code> method to create hash codes.
<code>HashSize</code>	This property returns the size of the hash code (in bits) that the hashing algorithm produces. See Table 13-1 for details of the hashing algorithms implemented in the .NET Framework and the sizes of the hash codes that they produce.
<code>CanReuseTransform</code>	These properties are from the <code>ICryptoTransform</code> interface, which is not directly related to hashing, but is implemented by the <code>HashAlgorithm</code> class. See Chapter 14 for more information.
<code>CanTransformMultipleBlocks</code>	
<code>InputBlockSize</code>	
<code>OutputBlockSize</code>	
Methods	
<code>Create</code>	This static method creates a new instance of the <code>HashAlgorithm</code> class by name. See Section 13.2.2 for further details.
<code>ComputeHash</code>	The <code>ComputeHash</code> method is used to create hash codes from byte arrays and data streams. See Section 13.2.3 and Section 13.2.4 in this chapter for full details.

Member	Description
<code>Initialize</code>	This is used by algorithm-implementation classes to initialize their state before use.
<code>Clear</code>	This releases the resources held by the <code>HashAlgorithm</code> instance.
<code>TransformBlock</code>	These methods are from the <code>ICryptoTransform</code> interface, which is not directly related to hashing, but is implemented by the <code>HashAlgorithm</code> class. See Chapter 14 for more information.
<code>TransformFinalBlock</code>	

13.2.2 Instantiating the Algorithm

The first step toward creating a hash code is to create an instance of an implementation class for the algorithm that you want to use. The simplest way of doing this is to use the name of the class directly:

```
# C#
SHA1Managed x_hash_alg = new SHA1Managed( );

# Visual Basic .NET
Dim x_hash_alg As SHA1Managed = New SHA1Managed( )
```

This statement creates an instance of the managed-code implementation of the SHA-1 algorithm. This is the clearest and most direct approach to instantiating hashing algorithms. Another approach is to use the static `HashAlgorithm.Create` method to instantiate the class indirectly:

```
# C#
HashAlgorithm x_hash_alg = HashAlgorithm.Create("SHA1");

# Visual Basic .NET
Dim x_hash_alg As HashAlgorithm = HashAlgorithm.Create("SHA1")
```

The `Create` method instantiates an algorithm based on the value of the string argument. [Table 13-3](#) lists the mapping between string values and implementation classes.

Table 13-3. Mapping string values to algorithm classes

String value	Algorithm-implementation class
MD5	<code>MD5CryptoServiceProvider</code>

String value	Algorithm-implementation class
<code>System.Security.Cryptography.MD5</code>	<code>MD5CryptoServiceProvider</code>
SHA	<code>SHA1CryptoServiceProvider</code>
SHA1	<code>SHA1CryptoServiceProvider</code>
<code>System.Security.Cryptography.SHA1</code>	<code>SHA1CryptoServiceProvider</code>
SHA256	<code>SHA256Managed</code>
SHA-256	<code>SHA256Managed</code>
<code>System.Security.Cryptography.SHA256</code>	<code>SHA256Managed</code>
SHA384	<code>SHA384Managed</code>
SHA-384	<code>SHA384Managed</code>
<code>System.Security.Cryptography.SHA384</code>	<code>SHA384Managed</code>
SHA512	<code>SHA512Managed</code>
SHA-512	<code>SHA512Managed</code>
<code>System.Security.Cryptography.SHA512</code>	<code>SHA512Managed</code>

If you do not supply a string as an argument to the `Create` method or the string is not one of the values listed in [Table 13-3](#), then the `System.Security.Cryptography.SHA1CryptoServiceProvider` algorithm will be used as a default.

This approach is less direct but does have a number of benefits. First, we only have to change the value of the string to change the algorithm that is used. Second, the systems administrator can alter the mappings listed in [Table 13-3](#) by editing the .NET configuration files, potentially allowing a customer to use a different algorithm for specific tasks. For example, a customer may want to use specialized hardware to generate hash codes rather than relying on the default software implementations. We recommend that you use the `HashAlgorithm.Create` method for the flexibility that it offers.

13.2.3 Hashing Data from Memory

The simplest way to generate a hash code is to use data held in memory. The overloaded `ComputeHash` method is able to create a hash code from a byte array, as shown by the following signature:

```
# C#

public byte[] ComputeHash(
    byte[] buffer
);

# Visual Basic .NET
```

```
Overloads Public Function ComputeHash( _
    ByVal buffer( ) As Byte _
) As Byte( )
```

The `ComputeHash` returns a byte array that contains the hash code for the message data. The following class demonstrates how to create a hash code for a `String` value; note that you should use the `System.Text.Encoding` class to convert the string to a byte array:

```
# C#

using System;
using System.Text;
using System.Security.Cryptography;

class StringHash {

    static void Main(string[] args) {

        // define the string that we will
        // create a hash code for
        String x_str = "Programming .NET Security";

        // create a byte array from the string
        byte[] x_message_data = Encoding.Default.GetBytes(x_str);

        // create an instance of the MD5 hashing algorithm
        HashAlgorithm x_hash_alg = HashAlgorithm.Create("MD5");

        // obtain the hash code from the HashAlgorithm by
        // using the ComputeHash method
        byte[] x_hash_code = x_hash_alg.ComputeHash(x_message_data);

        // print out the hash code to the console
        foreach (byte x_byte in x_hash_code) {
            Console.Write("{0:X2} ", x_byte);
        }
    }
}

# Visual Basic .NET

Imports System
Imports System.Text
Imports System.Security.Cryptography

Module StringHash

    Sub Main( )
        ' define the string that we will
        ' create a hash code for
        Dim x_str As String = "Programming .NET Security"
```

```

' create a byte array from the string
Dim x_message_data( ) As Byte = Encoding.Default.GetBytes(x_str)

' create an instance of the MD5 hashing algorithm
Dim x_hash_alg As HashAlgorithm = HashAlgorithm.Create("MD5")

' obtain the hash code from the HashAlgorithm by
' using the ComputeHash method
Dim x_hash_code( ) As Byte = x_hash_alg.ComputeHash(x_message_data)

' print out the hash code to the console
Dim x_byte As Byte
For Each x_byte In x_hash_code
    Console.Write("{0:X2} ", x_byte)
Next
End Sub

End Module

```

The output from this example is:

```
E1 62 9F C2 96 85 C3 A4 5B 94 97 57 D8 9C 65 78
```

To make the hash code easier to read, convert each byte to a hexadecimal value and print a space between each one. When you used the `ComputeHash` method in the previous example, you wanted to process an entire byte array; however, the `HashAlgorithm` class includes a different version of the method, which can process a region of a byte array:

```

# C#

public byte[] ComputeHash(
    byte[] buffer,
    int offset,
    int count
);

# Visual Basic .NET

Overloads Public Function ComputeHash( _
    ByVal buffer( ) As Byte, _
    ByVal offset As Integer, _
    ByVal count As Integer _
) As Byte( )

```

13.2.4 Hashing Streamed Data

You can also create a hash code from a stream of data; this is useful for processing datafiles or for reading data from a network connection. The `HashAlgorithm` class contains an overloaded version of the `ComputeHash` method for working with streams:

```
# C#
```

```

public byte[] ComputeHash(
    Stream inputStream
);

# Visual Basic .NET

Overloads Public Function ComputeHash( _
    ByVal inputStream As Stream _
) As Byte( )

```

The following class demonstrates how to create a hash code using a stream, in this case streaming data from a file called *myfile.txt*, which contains the string "Programming .NET Security":

```

# C#

using System;
using System.IO;
using System.Security.Cryptography;

class StreamHash {

    static void Main(string[] args) {

        // create the file stream
        Stream x_stream = new FileStream("mydata.txt", FileMode.Open);

        // create an instance of the MD5 hashing algorithm
        HashAlgorithm x_hash_alg = HashAlgorithm.Create("MD5");

        // obtain the hash code from the HashAlgorithm by
        // using the ComputeHash method
        byte[] x_hash_code = x_hash_alg.ComputeHash(x_stream);

        // print out the hash code to the console
        foreach (byte x_byte in x_hash_code) {
            Console.Write("{0:X2} ", x_byte);
        }

        // close the stream
        x_stream.Close( );
    }
}

# Visual Basic .NET

Imports System
Imports System.IO
Imports System.Security.Cryptography

Module StreamHash

    Sub Main( )

```



```

' create the file stream
Dim x_stream As Stream = New FileStream("mydata.txt", FileMode.Open)

' create an instance of the MD5 hashing algorithm
Dim x_hash_alg As HashAlgorithm = HashAlgorithm.Create("MD5")

' obtain the hash code from the HashAlgorithm by
' using the ComputeHash method
Dim x_hash_code( ) As Byte = x_hash_alg.ComputeHash(x_stream)

' print out the hash code to the console
Dim x_byte As Byte
For Each x_byte In x_hash_code
    Console.WriteLine("{0:X2} ", x_byte)
Next

' close the stream
x_stream.Close( )

End Sub

```

End Module

As you might expect, the output from the example, shown below, is the same hash code produced by our first example:

```
E1 62 9F C2 96 85 C3 A4 5B 94 97 57 D8 9C 65 78
```

13.2.5 Validating Hash Codes

Now that you have seen how to create hash codes (playing the role of Alice), we will show you how to validate them, playing the role of Bob. Remember that Bob receives a hash code and a message from Alice. He generates his own hash code and checks that it matches the one that he received. [Example 13-1](#) shows a simple approach to validating hash codes:

Example 13-1. Validating a hash code

```

# C#

using System;
using System.Text;
using System.Security.Cryptography;

class HashCodeValidation {

    private static byte[] ParseHashCodeString(string p_hash_code_string) {
        // split the hash code on spaces
        string[] x_elements = p_hash_code_string.Split(' ');
        // create a byte array to hold the elements
        byte[] x_hash_code_array = new byte[x_elements.Length];
        // parse each string element into a byte

```

```

        for (int i = 0; i < x_elements.Length; i++) {
            x_hash_code_array[i] = Byte.Parse(x_elements[i],
                System.Globalization.NumberStyles.HexNumber);
        }
        // return the byte array
        return x_hash_code_array;
    }

private static bool CompareHashCodes(byte[] x_hash_code1,
    byte[] x_hash_code2) {

    // check that the hash codes are the same length
    if (x_hash_code1.Length == x_hash_code2.Length) {
        // run through the hash code and check
        // each value in turn
        for (int i = 0; i < x_hash_code1.Length; i++) {
            if (x_hash_code1[i] != x_hash_code2[i]) {
                // the byte at this location is different
                // in each hash code
                return false;
            }
        }
        // the hash codes are the same
        return true;
    } else {
        // the hash codes contain different numbers of
        // bytes and so cannot be the same
        return false;
    }
}

public static bool ValidateHashCode(string p_hash_algorithm,
    string p_hash_string, byte[] p_data) {

    // parse the hash code string into a byte array
    byte[] x_hash_code = ParseHashCodeString(p_hash_string);

    // create the hashing algorithm object using the
    // name argument
    HashAlgorithm x_hash_alg = HashAlgorithm.Create(p_hash_algorithm);

    // compare the hash codes
    return CompareHashCodes(x_hash_code, x_hash_alg.ComputeHash(p_data));
}

static void Main(string[] args) {
    // define the message data that Alice sent to Bob
    string x_message_data = "Programming .NET Security";
    // covert the message data into a byte array
    byte[] x_message_bytes = Encoding.Default.GetBytes(x_message_data);
    // define the hash code that Alice sent to Bob
    string x_hash_string =

```

```

        "E1 62 9F C2 96 85 C3 A4 5B 94 97 57 D8 9C 65 78";
// define the name of the hashing algorithm that Alice used
string x_algorithm_name = "MD5";

// validate the hash code and write out the result
bool x_result = ValidateHashCode(x_algorithm_name,
    x_hash_string, x_message_bytes);

if (x_result) {
    Console.WriteLine("The hash code is valid");
} else {
    Console.WriteLine("The hash code is invalid");
}
}
}
# Visual Basic .NET

Imports System.Security.Cryptography
Imports System.Text

Module HashCodeValidation

    Public Function ParseHashCodeString(ByVal p_hash_code_string _
As String) As Byte( )
        ' split the hash code on spaces
        Dim x_elements( ) As String = p_hash_code_string.Split(" "c)
        ' create a byte array to hold the elements
        Dim x_hash_code_array( ) As Byte = New Byte(x_elements.Length - 1) {}
        ' parse each string element into a byte
        Dim i As Integer
        For i = 0 To x_elements.Length - 1 Step i + 1
            x_hash_code_array(i) = Byte.Parse(x_elements(i), _
                System.Globalization.NumberStyles.HexNumber)
        Next
        ' return the byte array
        Return x_hash_code_array
    End Function

    Private Function CompareHashCodes(ByVal x_hash_code1( ) As Byte, _
ByVal x_hash_code2( ) As Byte) As Boolean
        ' check that the hash codes are the same length
        If x_hash_code1.Length = x_hash_code2.Length Then
            ' run through the hash code and check
            ' each value in turn
            Dim i As Integer
            For i = 0 To x_hash_code1.Length - 1 Step i + 1
                If x_hash_code1(i) <> x_hash_code2(i) Then
                    ' the byte at this location is different
                    ' in each hash code
                    Return False
                End If
            Next
        End If
    End Function

```

```

        ' the hash codes are the same
        Return True
    Else
        ' the hash codes contain different numbers of
        ' bytes and so cannot be the same
        Return False
    End If
End Function

Public Function ValidateHashCode(ByVal p_hash_algorithm As String, _
    ByVal p_hash_string As String, ByVal p_data( ) As Byte) As Boolean

    ' parse the hash code string into a byte array
    Dim x_hash_code( ) As Byte = ParseHashCodeString(p_hash_string)

    ' create the hashing algorithm object using the
    ' name argument
    Dim x_hash_alg As HashAlgorithm = HashAlgorithm.Create(p_hash_algorithm)

    ' compare the hash codes
    Return CompareHashCodes(x_hash_code, x_hash_alg.ComputeHash(p_data))
End Function

Sub Main( )
    ' define the message data that Alice sent to Bob
    Dim x_message_data As String = "Programming .NET Security"
    ' covert the message data into a byte array
    Dim x_message_bytes( ) As Byte = Encoding.Default.GetBytes(x_message_data)
    ' define the hash code that Alice sent to Bob
    Dim x_hash_string As String = _
        "E1 62 9F C2 96 85 C3 A4 5B 94 97 57 D8 9C 65 78"
    ' define the name of the hashing algorithm that Alice used
    Dim x_algorithm_name As String = "MD5"

    ' validate the hash code and write out the result
    Dim x_result As Boolean = ValidateHashCode(x_algorithm_name, _
        x_hash_string, x_message_bytes)

    If (x_result) Then
        Console.WriteLine("The hash code is valid")
    Else
        Console.WriteLine("The hash code is invalid")
    End If
End Sub

End Module

```

The first step in validating the hash code is to parse the information that Bob has received. The format we have used to express the hash codes is the most commonly used, but you may encounter other formats.

The `ParseHashCodeString` method accepts a string containing a formatted hash code and produces

a `byte` array. The `CompareHashCodes` method takes care of comparing two arrays of `bytes` to ensure that they contain the same hash code.

The `ValidateHashCode` method uses the other two methods to validate a hash code, taking three arguments:

1. The name of the algorithm that Alice used to create the hash code
2. The hash code Alice created, expressed as a string of hexadecimal bytes
3. The message from Alice, expressed as a string

Parse the hash code string into a byte array, and then create an instance of the `HashAlgorithm` class based on the algorithm name. Assume that Alice and Bob agree on the algorithm that was used to create the hash codes, or that Alice picks one and indicates which algorithm she has used by sending some information along with the message. Notice how you cannot create an instance of any supported algorithm using the same code statements; this is another benefit of instantiating algorithms by name.

Finally, calculate a hash code for the message data and compare it to the one that Bob received from Alice. The example includes a `Main` method to demonstrate how to validate the hash code that you created previously.

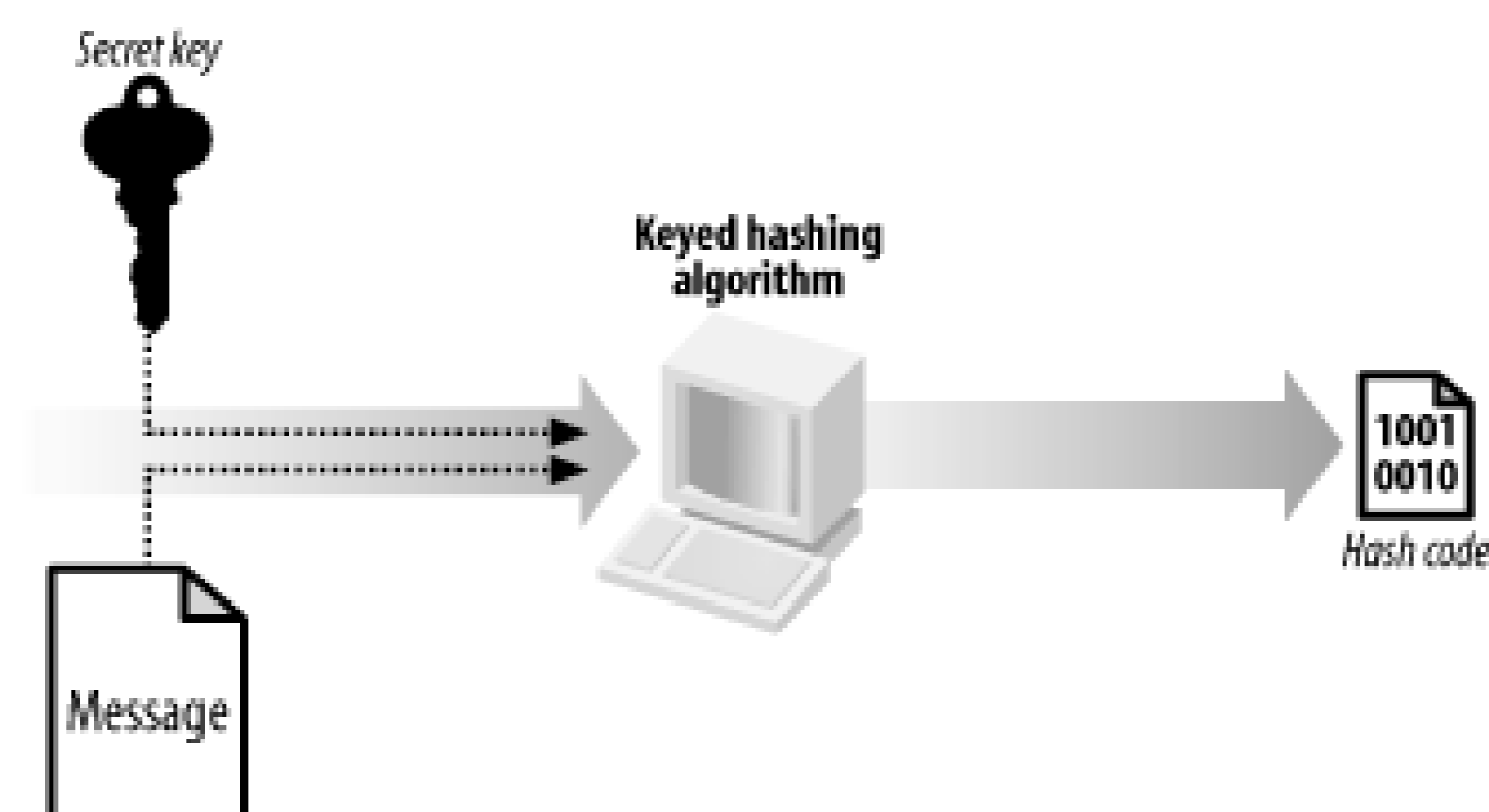
[\[Team LiB \]](#)

[[Team LiB](#)]

13.3 Keyed Hashing Algorithms Explained

Keyed hashing algorithms address the problem of Eve intercepting and replacing both the message and the hash code that Alice sent to Bob. Keyed hashing algorithms mix a secret key in with the message data blocks when creating a hash code, as illustrated in [Figure 13-4](#). Eve cannot replace a keyed hash code, because she does not know the secret key.

Figure 13-4. Using a secret key to create a hash code



13.3.1 Creating the Keyed Hash Code

There are two main approaches to creating keyed hash codes, also known as Message Authentication Codes (MACs). The first, and most commonly used, approach is to combine the secret key and the message data together, and use a normal cryptographic hashing algorithm to create a hash code for the combination. The details of how to combine the key and the message depend on your choice of algorithm, but one commonly used approach is the HMAC standard, which we discuss in the following section.

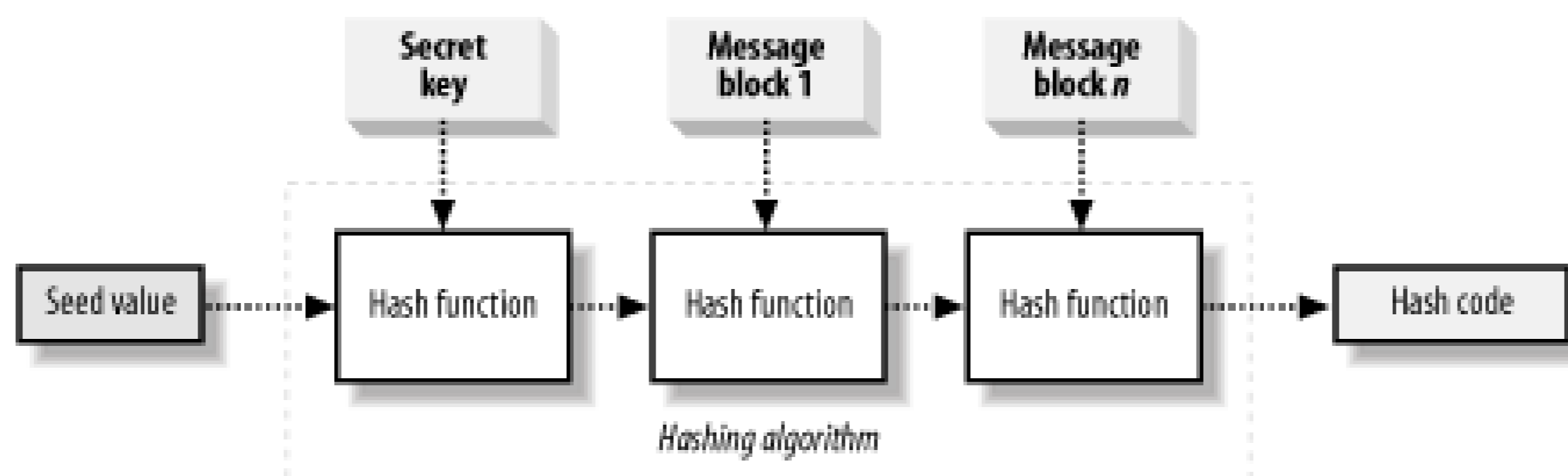
The second approach does not involve a hashing algorithm at all. The message data is encrypted with a symmetrical algorithm, and all but the last few bits of the encrypted data are discarded-what remains is used as the keyed hash code. Note that even though a normal hashing algorithm is not used, the protocol of using encryption to create a keyed hash code is still classified as a keyed hashing algorithm.

13.3.1.1 HMAC algorithms

The HMAC standard specifies how the message data and the secret key are combined, but does not specify which hashing algorithm should be used to create the hash code, meaning that the standard can be applied to any algorithm. It is customary to refer to HMAC algorithms by including the name of the hashing algorithm that was used. Therefore, HMAC-SHA-1 is the combination of the key and message, using the HMAC standard and the SHA-1 algorithm to create the hash code.

The HMAC standard specifies that the key is used as the first data block processed by the hash function, as shown in [Figure 13-5](#). This is equivalent to adding the key to the front of the message before it is processed.

Figure 13-5. The HMAC standard specifies that the secret key is the first data block processed by the hash function



13.3.1.2 Block cipher hash codes

Block cipher hash codes are generated by encrypting the message data and discarding all but the last few bytes. For example, if you wanted to create a 64-bit hash code, you would encrypt the data and discard all but the last 8 bytes of data, as shown in [Figure 13-6](#). In some schemes, you would encrypt the retained data again.

Figure 13-6. MACs are created by encrypting the message and discarding all but the last few bytes

The word "block" refers to the fact that the encryption algorithm works by breaking the message into fixed-sized blocks, in the same way that normal hashing algorithms operate (not all encryption algorithms work in this way, meaning that they cannot be used to create secure hash codes). See [Chapter 14](#) and [Chapter 15](#) for more information about encryption.

13.3.2 The .NET Framework Keyed Hashing Algorithms

The .NET Framework includes only two keyed hashing algorithms, but they are the most commonly used ones. The HMAC-SHA-1 algorithm combines the HMAC standard with the SHA-1 hashing algorithm. This is the most popular combination for commercial software and communications

security.

The MAC-Triple-DES algorithm uses the Triple-DES encryption algorithm to create a block cipher hash code. This technique has been widely used to protect the integrity of financial transactions, but is gradually being replaced by HMAC hash codes. See [Chapter 14](#) for more information about Triple-DES encryption.

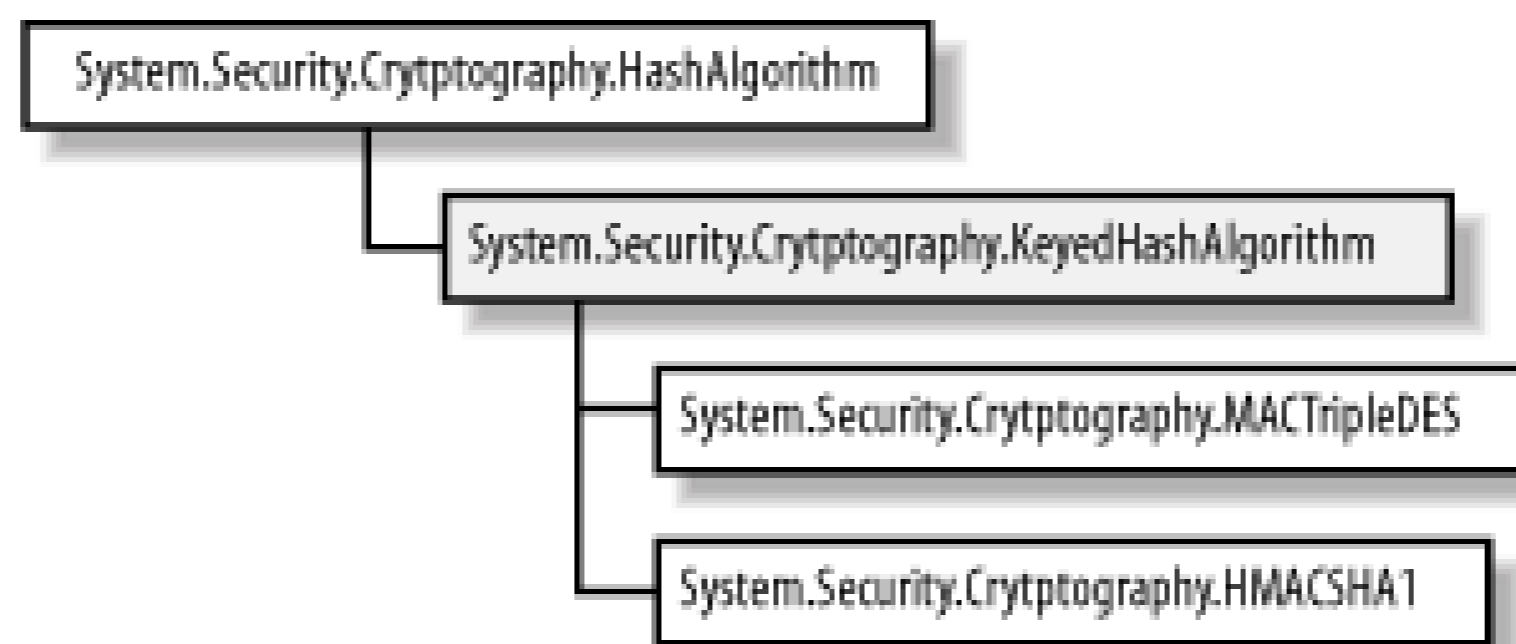
[\[Team LiB \]](#)

[[Team LiB](#)]

13.4 Programming Keyed Hashing Algorithms

The .NET Framework has a much more direct model for representing keyed hashing algorithms. There are no abstract classes for different algorithms, meaning that there can be only one implementation of each keyed algorithm. [Figure 13-7](#) shows the class hierarchy for keyed hashing algorithms.

Figure 13-7. The .NET Framework class hierarchy for keyed hashing algorithms.



13.4.1 The KeyedHashAlgorithm Class

The `KeyedHashAlgorithm` class extends `HashAlgorithm`, adding a new property called `Key`, which is used to get and set the secret key as a byte array. Aside from this addition, the `KeyedHashAlgorithm` class provides the same abstraction benefits as using `HashAlgorithm` for creating normal hash codes.

13.4.2 Instantiating the Algorithm

The process of creating instances of keyed algorithms is much the same as for normal hashing algorithms. You can create algorithms directly using the implementation class name, or indirectly using the `Create` method of the `KeyedHashingAlgorithm` class.

When using the direct approach, you can supply the secret key as an argument to the class constructor. In the following fragment, use the `System.Text.Encoding` class to convert a `string` into a `byte` array and to initialize the keyed hashing algorithm:

```

# C#

// define the key as a string
string x_key_string = "This is my secret key";
// convert the string to a byte array
byte[] x_key_bytes
    = System.Text.Encoding.Default.GetBytes(x_key_string);
  
```

```

// create the keyed hashing algorithm, using the
// byte array as the constructor argument
KeyedHashAlgorithm x_hash_alg = new HMACSHA1(x_key_bytes);

# Visual Basic .NET

' define the key as a string
Dim x_key_string As String = " This is my secret key "
' convert the string to a byte array
Dim x_key_bytes As Byte( ) _
    = System.Text.Encoding.Default.GetBytes(x_key_string)

' create the keyed hashing algorithm, using the
' byte array as the constructor argument
Dim x_hash_alg As KeyedHashAlgorithm = New HMACSHA1(x_key_bytes)

```

These statements create an instance of the HMAC-SHA-1 implementation class, using the key "This is my secret key." When using the `Create` method, the key value is set separately, using the `Key` property. The following fragment demonstrates how to do this:

```

# C#

// create the keyed hashing algorithm, using the
// byte array as the constructor argument
KeyedHashAlgorithm x_hash_alg = KeyedHashAlgorithm.Create("HMACSHA1");

// define the key as a string
string x_key_string = " This is my secret key ";
// convert the string to a byte array
byte[] x_key_bytes
    = System.Text.Encoding.Default.GetBytes(x_key_string);

// set the keyed hash algorithm key
x_hash_alg.Key = x_key_bytes;

# Visual Basic .NET

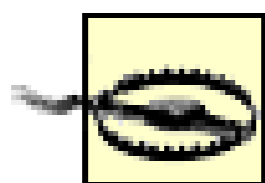
' create the keyed hashing algorithm, using the
' byte array as the constructor argument
Dim x_hash_alg As KeyedHashAlgorithm = KeyedHashAlgorithm.Create("HMACSHA1")

' define the key as a string
Dim x_key_string As String = " This is my secret key "
' convert the string to a byte array
Dim x_key_bytes As Byte( ) _
    = System.Text.Encoding.Default.GetBytes(x_key_string)

' set the keyed hash algorithm key
x_hash_alg.Key = x_key_bytes

```

The `Create` method creates an algorithm class based on the value of the string argument. [Table 13-4](#) lists the mapping between string values and implementation classes for keyed hashing algorithms.



If you do not supply a string as an argument to the `Create` method, the `System.Security.Cryptography.HMACSHA1` algorithm will be used as a default. If you use a string that is not one of the values listed in [Table 13-4](#), then a `System.NullReferenceException` will be thrown.

Table 13-4. Mapping string values to keyed algorithm classes

String value	Algorithm implementation class
HMACSHA1	HMACSHA1
System.Security.Cryptography.HMACSHA1	HMACSHA1
MACTripleDES	MACTripleDES
System.Security.Cryptography.MACTripleDES	MACTripleDES

13.4.3 Hashing Data and Validating Hash Codes

Because the `KeyedHashAlgorithm` class derives from `HashAlgorithm`, the processes for creating and validating keyed hash codes is the same as for normal hash codes, with the exception that the value of the key should be set either as a constructor argument or as using the `Key` property. For reference, [Example 13-2](#) demonstrates how to create a keyed hash code from a byte array of data:

You must set the secret key value before creating the hash code; otherwise, the `KeyedHashAlgorithm` will create a random key. See [Chapter 17](#) for more information about creating and managing keys.

Example 13-2. Creating a keyed hash code from a byte array of data

```
# C#

using System;
using System.Text;
using System.Security.Cryptography;

class ByteArrayHash {

    static void Main(string[] args) {

        // define the string that we will
        // create a hash code for
        String x_str = "Programming .NET Security";
        // create a byte array from the string
        byte[] x_message_data = Encoding.Default.GetBytes(x_str);

        // define the secret key as a string
        string x_key_string = "This is my secret key";
```

```
// create a byte array containing the key
byte[] x_key_bytes = Encoding.Default.GetBytes(x_key_string);

// create an instance of the HMAC-SHA-1 keyed algorithm
KeyedHashAlgorithm x_hash_alg = KeyedHashAlgorithm.Create("HMACSHA1");

// set the secret key for the hashing algorithm
x_hash_alg.Key = x_key_bytes;

// obtain the hash code from the HashAlgorithm by
// using the ComputeHash method
byte[] x_hash_code = x_hash_alg.ComputeHash(x_message_data);

// print out the hash code to the console
foreach (byte x_byte in x_hash_code) {
    Console.WriteLine("{0:X2} ", x_byte);
}
}

# Visual Basic .NET

Imports System
Imports System.Text
Imports System.Security.Cryptography

Module Module1

    Sub Main( )

        ' define the string that we will
        ' create a hash code for
        Dim x_str As String = "Programming .NET Security"
        ' create a byte array from the string
        Dim x_message_data( ) As Byte = Encoding.Default.GetBytes(x_str)

        ' define the secret key as a string
        Dim x_key_string As String = "This is my secret key"
        ' create a byte array containing the key
        Dim x_key_bytes( ) As Byte = Encoding.Default.GetBytes(x_key_string)

        ' create an instance of the HMAC-SHA-1 keyed algorithm
        Dim x_hash_alg As KeyedHashAlgorithm = _
            KeyedHashAlgorithm.Create("HMACSHA1")

        ' set the secret key for the hashing algorithm
        x_hash_alg.Key = x_key_bytes

        ' obtain the hash code from the HashAlgorithm by
        ' using the ComputeHash method
        Dim x_hash_code( ) As Byte = x_hash_alg.ComputeHash(x_message_data)
```

```
' print out the hash code to the console
Dim x_byte As Byte
For Each x_byte In x_hash_code
    Console.WriteLine("{0:X2} ", x_byte)
Next
End Sub
End Module
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

13.5 Extending the .NET Framework

In this section, we extend the .NET Framework by adding a new algorithm. Unfortunately, cryptographic hashing algorithms usually include large tables of random numbers, making them unsuitable for listing in a book such as this.

Because we want to focus on the process of adding an algorithm rather than on the algorithm itself, we have selected a simple, noncryptographic hashing scheme to use, known as Adler32.

13.5.1 The Adler32 Algorithm Explained

The Adler32 algorithm generates a 32-bit hash code and is included as part of GZIP-compressed files to guard against inadvertent modification (for example, errors introduced in copying a compressed file over a network connection).

Adler32 is a simple algorithm, which operates on 8-bit blocks of message data to create hash codes, as follows:

1. Define a 16-bit value called Sum1 with an initial value of 1.
2. Define a 16-bit value called Sum2 with an initial value of 0.
3. Process each 8-bit message block in turn:
4. Add the numeric value of the message block to Sum 1, modulo 65521.
5. Add the value of Sum 1 to Sum 2, modulo 65521.
6. Concatenate Sum 2 and Sum 1 together to create a 32-bit hash code.

The Adler32 algorithm is not suitable for cryptographic uses because it produces hash codes that are susceptible to attack. Adler32 produces short hash codes that easily succumb to birthday attacks and, more importantly, it is easy to generate a message that results in a specific Adler32 hash code.

13.5.2 Defining the Abstract Class

The first step is to create an abstract class representing the Adler32 algorithm. The abstract class extends the `System.Security.Cryptography.HashAlgorithm` class but does not need to implement any members. Here is the definition of the abstract `Adler32` algorithm class:

```
# C#  
using System.Security.Cryptography;
```

```

public abstract class Adler32 : HashAlgorithm {

}

# Visual Basic .NET

Imports System.Security.Cryptography

Public MustInherit Class Adler32
    Inherits HashAlgorithm

End Class

```

We have created the abstract class to conform to the .NET class model, allowing you to add other Adler32 implementations in the future. You can omit the abstract class, but we feel that the extra flexibility of this model is well worth the effort.

13.5.3 Defining the Implementation Class

The abstract `HashAlgorithm` class contains a lot of functionality that makes writing a new algorithm a simple process. For example, the details of the `ComputeHash` method are hidden from the implementer, and only a small number of methods need to be written in order to create a fully functional implementation.

13.5.3.1 Creating the implementation class

We have named our implementation class `Adler32Managed`, because it is written using a managed .NET language. The following listing shows the initial version of the implementation class extending the abstract `Adler32` class. We have defined the two 16-bit values as `o_sum_1` and `o_sum_2`. For Visual Basic .NET, we have used 32-bit integers to represent the 16-bit sums, because Visual Basic .NET does not support unsigned data types:

```

# C#

using System;
using System.Security.Cryptography;

public class Adler32Managed : HashAlgorithm {
    private ushort o_sum_1;
    private ushort o_sum_2;

}

# Visual Basic .NET
Imports System
Imports System.Security.Cryptography

Public Class Adler32Managed
    Inherits Adler32

```

```

    Private o_sum_1 As Integer
    Private o_sum_2 As Integer

```

```
End Class
```

13.5.3.2 Implementing the Initialize method

You should use the `Initialize` method to set the initial state of any algorithm; an algorithm will produce unexpected hash code values unless correctly reset between invocations. For Adler32, we use this method to set the initial sum values. We have also added a class constructor that calls the `Initialize` method to ensure that the values are correctly set before the class is first used:

```

# C#

using System;
using System.Security.Cryptography;

public class Adler32Managed : HashAlgorithm {
    private ushort o_sum_1;
    private ushort o_sum_2;

    public Adler32Managed( ) {
        Initialize( );
    }

    public override void Initialize( ) {
        // reset the sum values
        o_sum_1 = 1;
        o_sum_2 = 0;
    }
}

# Visual Basic .NET

Imports System
Imports System.Security.Cryptography

Public Class Adler32Managed
    Inherits Adler32

    Private o_sum_1 As Integer
    Private o_sum_2 As Integer

    Public Sub New( )
        Initialize( )
    End Sub

    Public Overrides Sub Initialize( )
        ' reset the sum values
        o_sum_1 = 1
        o_sum_2 = 0
    End Sub

```



```
End Sub
```

```
End Class
```

13.5.3.3 Implementing the HashCore and HashFinal methods

The `HashCore` method receives message data into the algorithm class; the amount of data to process can vary from a single byte to very large arrays. Always ensure that you process only the region of the array indicated by the `p_start_index` and `p_count` parameters. This method may be invoked multiple times, and you must make sure that your algorithm can process data blocks incrementally, rather than assuming that all of the message data will arrive at once:

```
# C#
```

```
protected override void HashCore(byte[] p_array, int p_start_index, int p_count) {
    // process each byte in the array
    for (int i = p_start_index; i < p_count; i++) {
        o_sum_1 = (ushort)((o_sum_1 + p_array[i]) % 65521);
        o_sum_2 = (ushort)((o_sum_1 + o_sum_2) % 65521);
    }
}
```

```
# Visual Basic .NET
```

```
Protected Overrides Sub HashCore(ByVal p_array( ) As Byte, _
    ByVal p_start_index As Integer, ByVal p_count As Integer)

    ' process each byte in the array
    Dim i As Integer
    For i = p_start_index To p_count - 1 Step i + 1
        o_sum_1 = (o_sum_1 + p_array(i)) Mod 65521
        o_sum_2 = (o_sum_1 + o_sum_2) Mod 65521
    Next
End Sub
```

The `HashFinal` method is called when all of the message data has been processed by the `HashCore` method and returns the computed hash code value as a `byte` array. This method allows you to complete any pending calculations before computing the final hash code. For Alder32, you simply need to concatenate the two sum values together to form the 32-bit hash code, and convert the result into a `byte` array:

```
# C#
```

```
protected override byte[] HashFinal( ) {
    // concat the two 16 bit values to form
    // one 32-bit value
    uint x_concat_value = (uint)((o_sum_2 << 16) | o_sum_1);
    // use the bitconverter class to render the
    // 32-bit integer into an array of bytes
    return BitConverter.GetBytes(x_concat_value);
}
```

```

# Visual Basic .NET

Protected Overrides Function HashFinal( ) As Byte( )
    ' concat the two 16 bit values to form
    ' one 32-bit value
    Dim x_concat_value As Integer = (o_sum_2 * 65536) Or o_sum_1
    ' use the bitconverter class to render the
    ' 32-bit integer into an array of bytes
    Return BitConverter.GetBytes(x_concat_value)
End Function

```

13.5.3.4 Completing the class

The final step is to implement the `HashSize` property to specify the size (in bits) of the hash code that you are generating. [Example 13-3](#) is the complete listing for the `Alder32Managed` class:

Example 13-3. algorithm

```

# C#

using System;
using System.Security.Cryptography;

public class Adler32Managed : HashAlgorithm {
    private ushort o_sum_1;
    private ushort o_sum_2;

    public Adler32Managed( ) {
        Initialize( );
    }

    public override int HashSize {
        get {
            return 32;
        }
    }

    public override void Initialize( ) {
        // reset the sum values
        o_sum_1 = 1;
        o_sum_2 = 0;
    }

    protected override void HashCore(byte[] p_array, int p_start_index,
int p_count) {
        // process each byte in the array
        for (int i = p_start_index; i < p_count; i++) {
            o_sum_1 = (ushort)((o_sum_1 + p_array[i]) % 65521);
            o_sum_2 = (ushort)((o_sum_1 + o_sum_2) % 65521);
        }
    }
}

```

```

protected override byte[] HashFinal( ) {
    // concat the two 16 bit values to form
    // one 32-bit value
    uint x_concat_value = (uint)((o_sum_2 << 16) | o_sum_1);
    // use the bitconverter class to render the
    // 32-bit integer into an array of bytes
    return BitConverter.GetBytes(x_concat_value);
}

# Visual Basic .NET

Imports System
Imports System.Security.Cryptography

Public Class Adler32Managed
    Inherits Adler32

    Private o_sum_1 As Integer
    Private o_sum_2 As Integer

    Public Sub New( )
        Initialize( )
    End Sub

    Public Overrides ReadOnly Property HashSize( ) As Integer
        Get
            Return 32
        End Get
    End Property

    Public Overrides Sub Initialize( )
        ' reset the sum values
        o_sum_1 = 1
        o_sum_2 = 0
    End Sub

    Protected Overrides Sub HashCore(ByVal p_array( ) As Byte, _
        ByVal p_start_index As Integer, ByVal p_count As Integer)

        ' process each byte in the array
        Dim i As Integer
        For i = p_start_index To p_count - 1 Step i + 1
            o_sum_1 = (o_sum_1 + p_array(i)) Mod 65521
            o_sum_2 = (o_sum_1 + o_sum_2) Mod 65521
        Next
    End Sub

    Protected Overrides Function HashFinal( ) As Byte( )
        ' concat the two 16 bit values to form
        ' one 32-bit value

```

```

        Dim x_concat_value As Integer = (o_sum_2 * 65536) Or o_sum_1
        ' use the bitconverter class to render the
        ' 32-bit integer into an array of bytes
        Return BitConverter.GetBytes(x_concat_value)
    End Function

```

```
End Class
```

13.5.3.5 Testing the implementation

Now that you have created the implementation class, you can use the Adler32 algorithm in exactly the same way as the other classes covered in this chapter. The following listing details how to create an Adler32 hash code for the message "Programming .NET Security":

```

# C#

// define the message string that we will create a hash code for
string x_message_string = "Programming .NET Security";
// convert the message string into bytes
byte[] x_message_bytes = Encoding.Default.GetBytes(x_message_string);

HashAlgorithm x_hash_alg = new Adler32Managed( );
byte[] x_hash_code = x_hash_alg.ComputeHash(x_message_bytes);

// print out the hash code
foreach (byte x_byte in x_hash_code) {
    Console.WriteLine("{0:X2} ", x_byte);
}

# Visual Basic .NET

' define the message string that we will create a hash code for
Dim x_message_string As String = "Programming .NET Security"
' convert the message string into bytes
Dim x_message_bytes( ) As Byte = Encoding.Default.GetBytes(x_message_string)

Dim x_hash_alg As HashAlgorithm = New Adler32Managed( )
Dim x_hash_code( ) As Byte = x_hash_alg.ComputeHash(x_message_bytes)

' print out the hash code
Dim x_byte As Byte
For Each x_byte In x_hash_code
    Console.WriteLine("{0:X2} ", x_byte)
Next

```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 14. Symmetric Encryption

In [Chapter 13](#), we showed you how Alice can use data hash codes to prevent Eve from secretly modifying the messages that she sends to Bob. Hash codes are useful for message integrity, but they do not provide any confidentiality; although Eve cannot alter the messages from Alice without Bob knowing, she *is* able to read the messages, and Alice and Bob are unable to keep secrets from her.

In this chapter, we examine how to achieve confidentiality with data encryption, and how to encrypt and decrypt data using the .NET Framework. We also show you how to extend the .NET Framework by adding a new encryption algorithm.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

14.1 Encryption Revisited

Encryption is the process of transforming a message so that it cannot be read if it is intercepted but can still be understood by the intended recipient. If Alice encrypts and sends a confidential message to Bob, Eve will not be able to make sense of what it says, and the contents of the message will remain confidential. Bob, as the intended recipient, is able to decrypt the message, which is the process of restoring the original message from the encrypted data. Like almost all types of cryptography, encryption relies on keeping a key secret from Eve.

[Figure 14-1](#) shows the basic model for modern data encryption. Alice uses an encryption algorithm to convert her message into encrypted data, which she sends to Bob. Bob uses a secret key to decrypt the data, transforming it back into the original message, which he then reads. Eve can intercept the encrypted data, but she is unable to read the original message unless she can acquire the secret key that Bob uses to decrypt the data or find a weakness in the encryption algorithm Alice used.

Figure 14-1. Alice uses encryption to ensure message confidentiality

If Eve acquires the key, she will be able to decrypt the data in the same way that Bob does and read the message. She may be able to steal the key, or she may keep guessing until she finds the right key. If Eve is able to find a weakness in the algorithm Alice used, then she may be able to determine the contents of the confidential message without needing the secret key. In this chapter, we discuss the first of the two basic types of encryption, known as "symmetric" or "secret key" encryption. We cover the second type of encryption in [Chapter 15](#).

[\[Team LiB \]](#)

[[Team LiB](#)]

14.2 Symmetric Encryption Explained

Symmetric encryption is the oldest and most established way of ensuring message confidentiality, dating back for many thousands of years. [Figure 14-2](#) illustrates the basic approach, which is as follows:

1. Alice and Bob agree on a secret key.
2. Alice composes a confidential message to Bob and encrypts it using the secret key.
3. Alice sends the encrypted data to Bob.
4. Bob uses the secret key to decrypt the encrypted data, and reads the confidential message.

The most important aspect of symmetric encryption is that Alice and Bob agree on a secret key without Eve eavesdropping. If Eve acquires the secret key, she will be able to read, modify, and forge messages between Alice and Bob. The traditional means of agreeing on secret key has been to meet in person at a private location, or to rely on a secure and trusted courier service, both of which can be difficult to arrange securely.

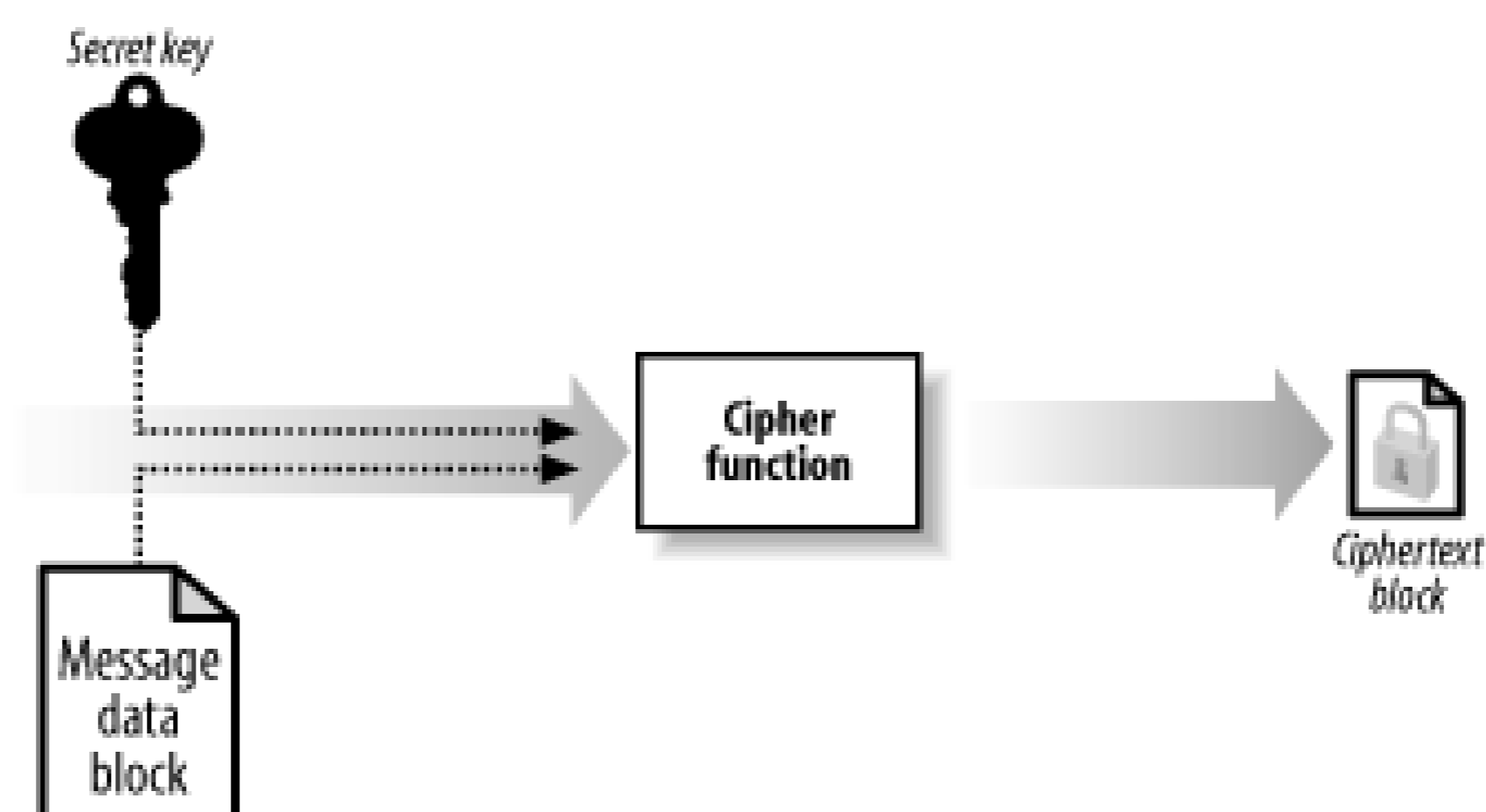
Figure 14-2. Alice and Bob use symmetric encryption to ensure message confidentiality

14.2.1 Creating the Encrypted Data

At the heart of a symmetric encryption algorithm is a cipher function that transforms fixed-size

blocks of message data, known as plaintext, into blocks of encrypted data, known as ciphertext, illustrated by [Figure 14-3](#). This type of function is a block cipher, because it processes blocks of data sequentially and independently. The output of a block cipher is deterministic; this means that a given block of plaintext encrypted using a given key will always result in the same block of ciphertext. Different block ciphers process differently sized blocks of plaintext.

Figure 14-3. A cipher function uses a secret key and a fixed-size message block to create a block of ciphertext



You can restore a block of plaintext from a block of ciphertext by reversing the transformation, as illustrated by [Figure 14-4](#). Both the transformation into ciphertext and the restoration of the plaintext require the secret key; therefore, Bob can decrypt the data only if he uses the same secret key that Alice used to encrypt the message. This is different from the hash functions we examined in [Chapter 13](#); encryption is a reversible, or "two-way," process, where we can use the secret key to "reverse" the encryption effect and restore the plaintext. Creating a hash code is a "one-way" process, and the original message data cannot be determined from the hash code data easily.

Figure 14-4. Cipher functions also transform ciphertext back into plaintext, which is known as decryption

An encryption algorithm is the protocol for using a cipher function to encrypt plaintext, in the same way that a hashing algorithm specifies how to use a hash function to create a hash code. The algorithm specifies how the plaintext is broken into blocks and how to use the output of the cipher function in creating the ciphertext; the details of how to perform these tasks differ between algorithms.

On a practical basis, the .NET Framework provides a standardized model for using symmetric encryption algorithms, and you do not have to understand the inner workings of any specific block ciphers. However, you should understand some of the different ways that you can configure an algorithm, which we discuss in the following sections.

14.2.2 Cipher Modes

Most symmetric encryption algorithms operate in different cipher modes, which specify variations in the way that data is prepared before being processed by the cipher function. The following sections explain the most commonly used modes, but you should bear in mind that there are a large number of variations and not all algorithms can support all cipher modes.

14.2.2.1 Electronic Codebook mode

The Electronic Codebook (ECB) mode is the simplest cipher mode and builds ciphertext by concatenating the output of the cipher function, as shown in [Figure 14-5](#). We can summarize the process as follows:

1. Break the plaintext into blocks that are the same size as the input of the cipher function.
2. Use the secret key to encrypt each plaintext data block in turn with the cipher function.
3. Append the output of the cipher function to the ciphertext.

Figure 14-5. The ECB mode is the simplest way to encrypt data with a block cipher function

Although this seems like the obvious way to encrypt data, the deterministic nature of the cipher function exposes the ciphertext to a special kind of attack whenever there is repetition in the confidential message.

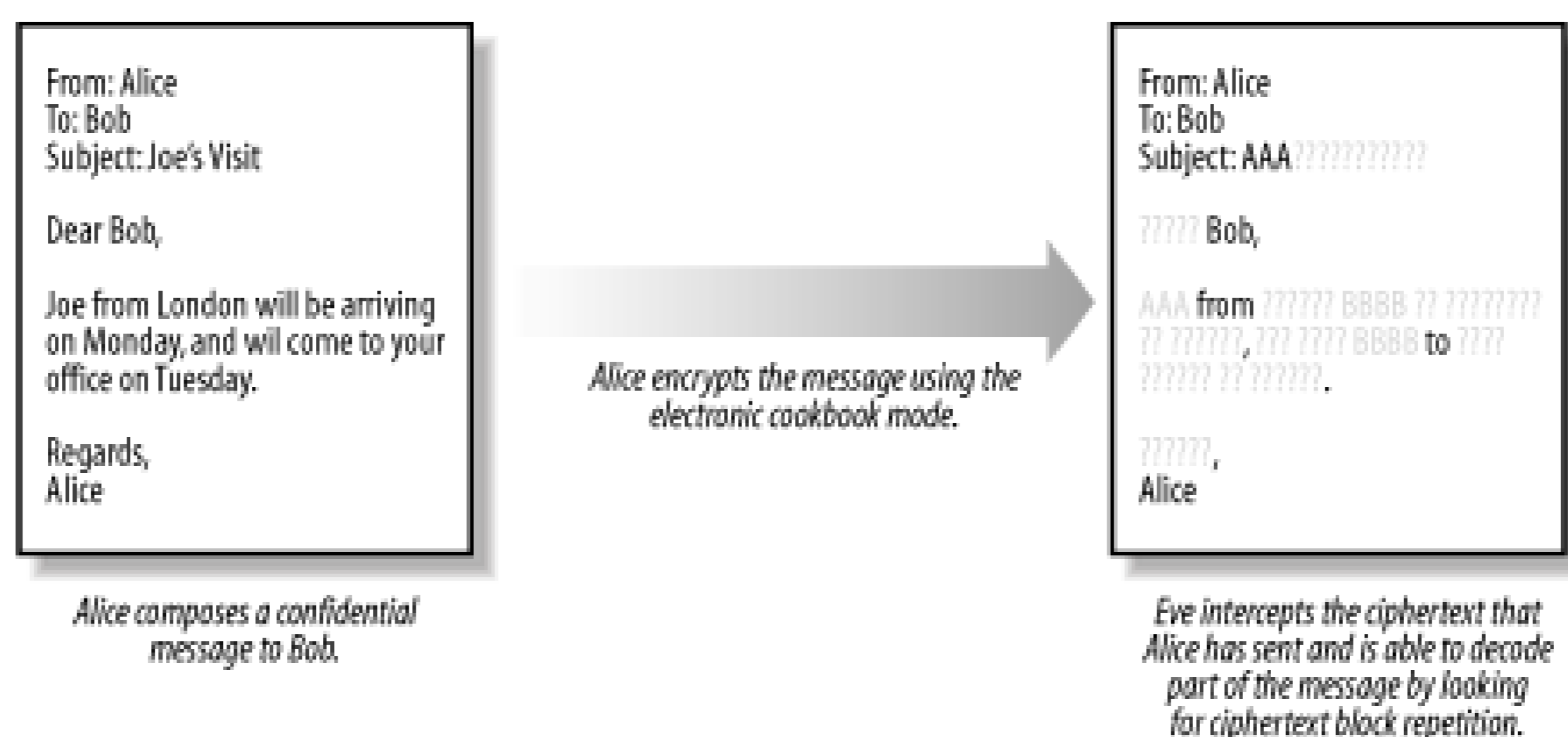
The cipher function is consistent, and using a secret key to encrypt two identical blocks of data results in identical blocks of ciphertext. If there are any repeated blocks of data in the plaintext, you will end up with repeated blocks of data in the ciphertext, which presents a weakness that Eve can exploit.

There is a natural structure to many types of messages; for example, Alice's letters often begin with "Dear Bob" and emails will often start with "From," "To," or "Received." Imagine that Alice sends a

company memo to Bob, and that Eve knows (or can guess at) the structure of such messages.

Figure 14-6 shows a simplified representation of the effect of building the ciphertext from the output of the cipher function directly-simplified because we have assumed that each word is encrypted as a plaintext block in a case-insensitive manner. Real encryption algorithms work on binary data, but our example is useful for understanding the issue at hand. Eve knows that all of Alice's memos begin with "From," and therefore knows that the first block of ciphertext represents that word. Eve knows that every time she finds that particular ciphertext block in the encrypted data she has located another instance of the word "From." By applying her knowledge of the message structure, Eve deduces several words in the confidential message, including the words "To," "Bob," "Subject," and "Alice."

Figure 14-6. Eve uses plaintext repetition to build a codebook of known ciphertext blocks



Eve finds repeated ciphertext blocks for which she can't deduce the meaning, because the word that the block represents is not included in the standard structure of Alice's memo; AAA and BBBB indicate these blocks. Regardless, Eve keeps a record of these blocks, because she may intercept another message that reveals their meaning later and shed more light on this message—a process of building a codebook that will help her recognize the meaning of known ciphertext blocks as she intercepts them. Eve's "codebook" is useful only while Alice and Bob use the same secret key. If they agree on a new key, the ciphertext blocks for known words will be different, and Eve will have to start from scratch.

The ECB mode is the simplest way to create ciphertext, although the risks associated with encrypting duplicate plaintext blocks mean that you should use it with caution. The following sections describe alternate cipher modes that address the problem of block repetition, and we recommend that you use these instead of ECB.

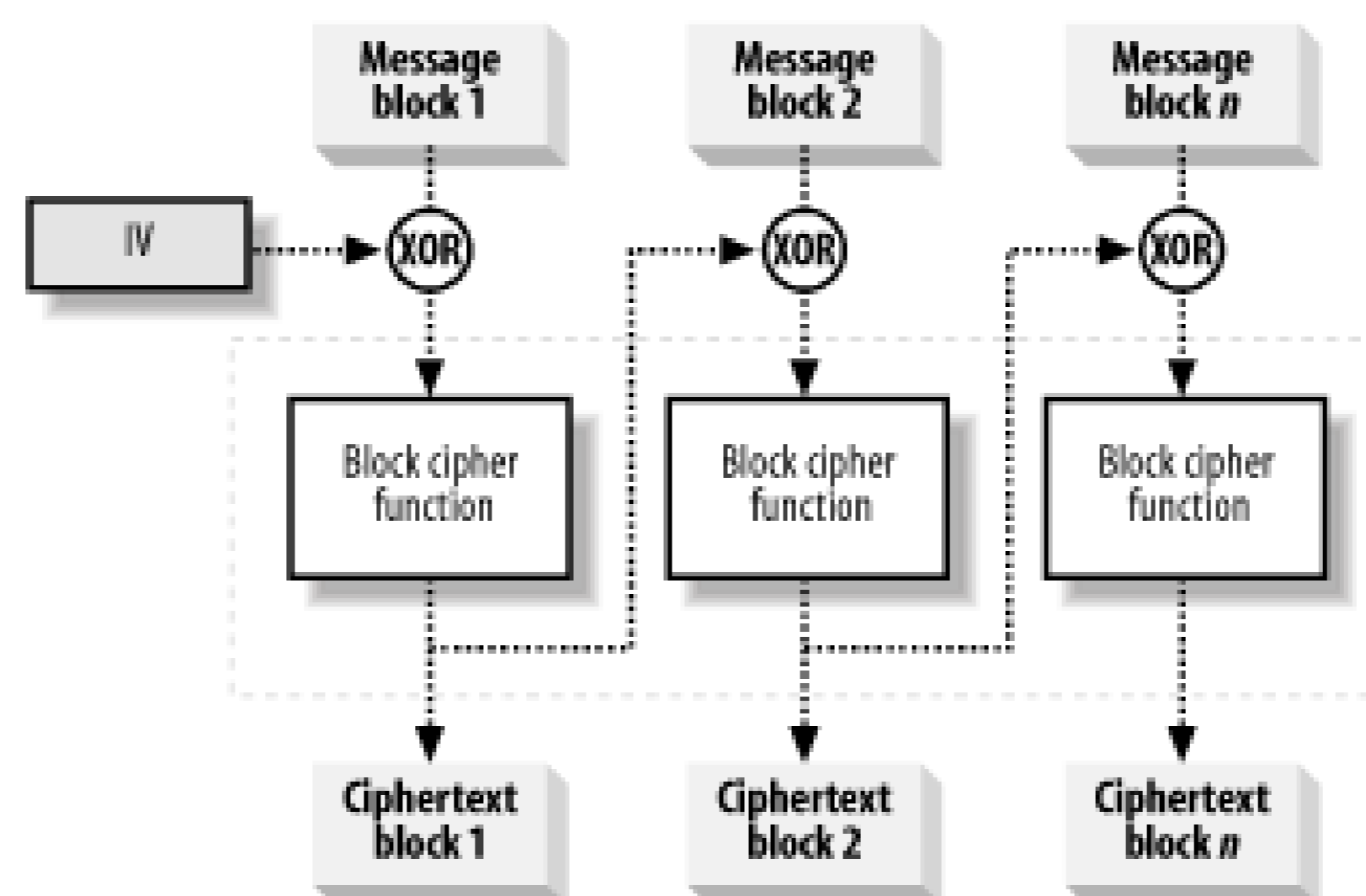
14.2.2.2 Cipher block chaining

The most common way to overcome the block repetition weakness of ECB is to add "feedback" into the cipher function. Adding feedback produces a ciphertext in which the result of encrypting a data block affects the results of encrypting all subsequent blocks. Figure 14-7 illustrates the most widely used way of doing this, called Cipher Block Chaining (CBC), which works as follows:

1. Break the plaintext into blocks that are the same size as the input for the cipher function.

2. Process the first message block:
 - a. XOR the message block with the "seed" data to create a combined data block.
 - b. Encrypt the result to produce the first block of ciphertext.
3. Process remaining message blocks in turn:
 - a. XOR the plaintext block with most recently created ciphertext block to create a combined data block.
 - b. Encrypt the combined data block and append the result to the ciphertext.

Figure 14-7. CBC uses the result of previous encryptions to transform plain text blocks



For example, to encrypt the 10th plaintext block you XOR it with the 9th ciphertext block and encrypt the result, producing the 10th ciphertext block, which you XOR with the 11th plaintext block, and so on.

The seed data for symmetric encryption is called the initialization vector (IV) and is used to ensure that the first block of ciphertext is not open to ECB-style attacks. The IV is a block of random data that is the same size as the cipher function input block.

Using CBC means that Bob has to use the same IV to decrypt the data as Alice used to encrypt the message; fortunately, the value of the IV is not a secret, and there is no danger in allowing it to fall into Eve's hands. Alice should select a new IV for each message that she encrypts, and she will normally send the IV to the recipient along with the encrypted data.

14.2.2.3 Cipher feedback mode

Processing plaintext data block by block is not suitable for all projects, especially those where the data to encrypt becomes available over time (for example, received via a network stream). In such cases, the confidential data has to be held in memory until a complete block is received and

encrypted, which can present a security risk.

Cipher Feedback (CFB) mode is a technique for using a block cipher function to encrypt plaintext in fragments smaller than the function block size, allowing data to be encrypted as it arrives. Like CBC, CFB uses an IV, but in this mode, it represents the initial value for a queue, which is a data block the same size as the cipher function input. If you are using a cipher that operates on 64-bit data blocks but the data becomes available in 8-bit chunks, then the CFB protocol is as follows:

1. Create a 64-bit queue, and fill it with a 64-bit IV.
2. As each 8-bit chunk of plaintext becomes available:
 - a. Encrypt the queue.
 - b. XOR the leftmost eight bits of the encrypted queue with the eight bits of plaintext to create the cipher data chunk.
 - c. Append the cipher data chunk to the ciphertext output.
 - d. Shift the queue eight bits to the left and discard the leftmost eight bits.
 - e. Set the rightmost eight bits of the queue to be the eight bits of ciphertext.

This seems a lot more complicated than it actually is. Start with a 64-bit queue that contains the IV, and as you progress, the queue gradually fills up with 8-bit chunks of ciphertext. [Figure 14-8](#) illustrates the way in which CFB works.

Figure 14-8. CFB mode uses a queue to process small chunks of data

CFB encrypts data one bit at a time if required, although that would use a lot of CPU (because the cipher function encrypts a complete block of data for each message bit). In the queue, the gradual

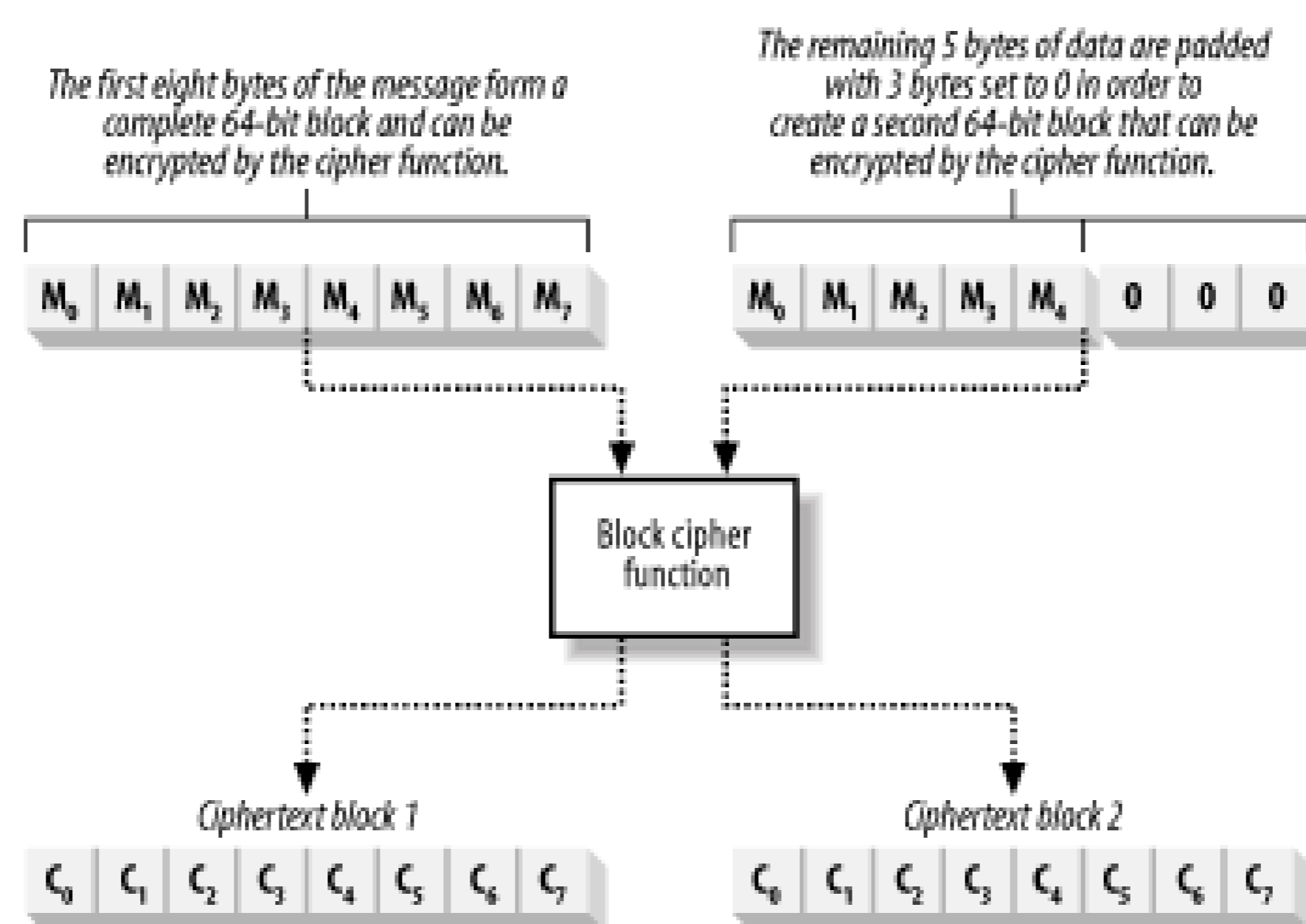
shifting of the data protects against ECB attacks.

14.2.3 Block Padding

Most messages do not divide neatly into the fixed-size blocks required by the cipher function, and there is usually a partial data block left over at the end. The cipher function cannot process partial blocks, and the algorithm adds "padding" to the leftover data to create a complete block.

Padding, unlike other aspects of encryption, is expressed in bytes rather than bits. For example, if you are using a cipher that processes 64-bit blocks, a 13-byte (104-bit) message breaks down into one complete data block with 40 bits remaining. We need to add 3 bytes (24 bits) of data to the remainder to create a second block that the cipher function can process. There are two commonly used approaches to block padding, and the .NET Framework supports both. The first approach is simply to set the remaining bytes to 0, as shown in [Figure 14-9](#).

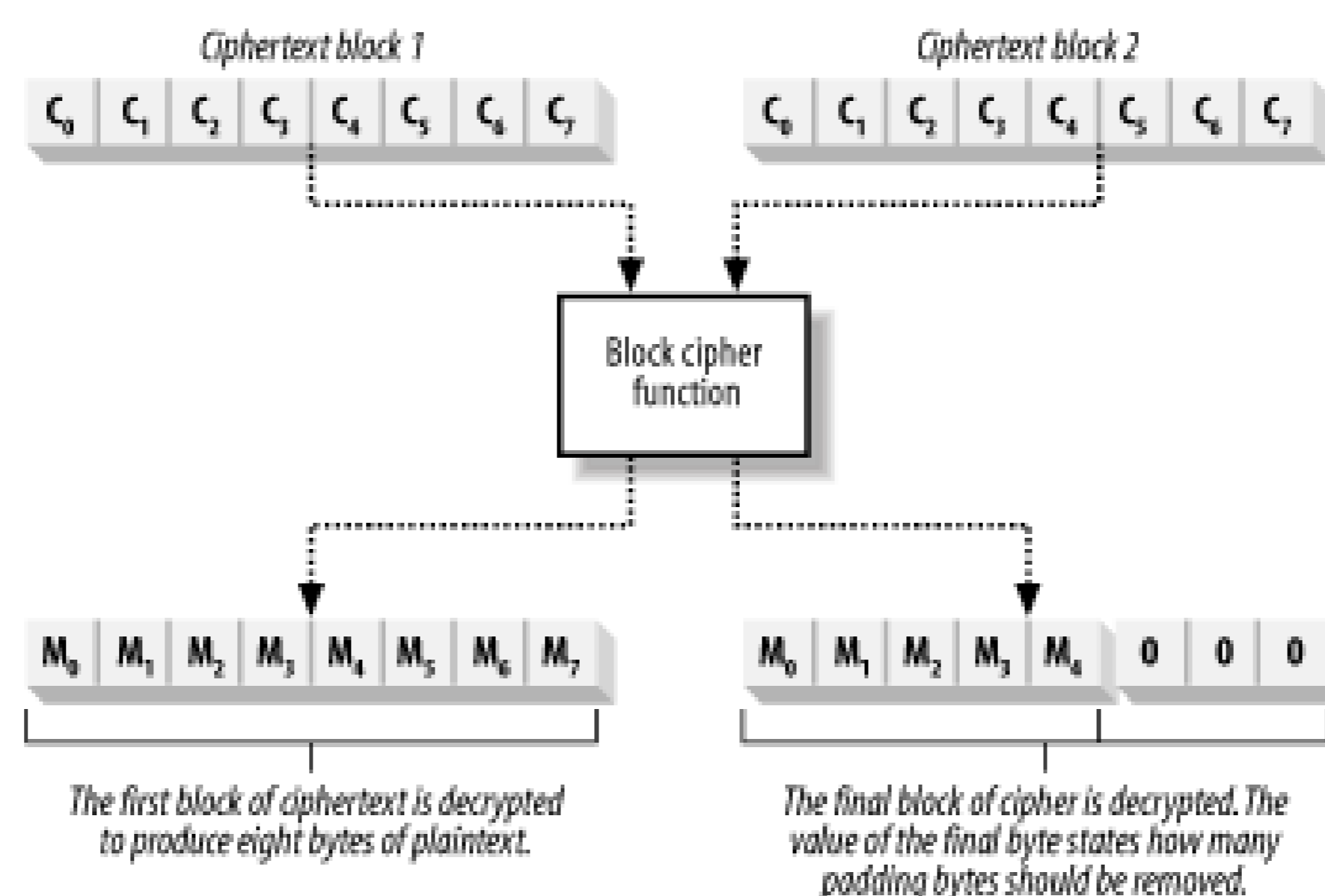
Figure 14-9. Using zeros to pad a partial data block



Encrypt the padded plaintext block, and the result is appended to the ciphertext; note that the ciphertext will be three bytes larger than the plaintext. When you decrypt the ciphertext, remove the padding values to return to the original plaintext message. The problem with this approach is that it is difficult to be sure that you are removing the padding correctly; some algorithms naturally generate strings of zeros, and you may remove bytes that are part of the message.

A more reliable approach is to use PKCS #7 padding, where you set the value of the padding bytes to the number of bytes you added. For our example, the padding would be three bytes set to a value of "3". When you remove the padding, simply read the value of the last byte and you can tell which bytes you have to remove, as shown in [Figure 14-10](#).

Figure 14-10. Using PKCS #7 padding to complete a partial data block



You still have to add PKCS #7 padding even when there are no partial plaintext blocks to process. This is because many algorithm implementations examine the numeric value of the last byte, and do not check to ensure that all of the padding bytes are set to the same value. When the plaintext results in only whole blocks, add a complete block of padding data, with all of the byte values set to be 1/8th of the block size; for a 64-bit block cipher, this would be 8 bytes of value "8". When the ciphertext is decrypted, remove the entire final block to leave the original plaintext. PKCS #7 padding is limited to 256 bytes of padding, which is the largest numeric value a single byte of data can represent.

14.2.4 Symmetric Encryption Key Lengths

The security of symmetric encryption comes from the fact that Eve does not have the secret key. If she did, she would be able to read any of Alice's messages to Bob and even forge new messages by using the secret key to encrypt them.

We describe secret keys by their length, measured in bits. A 64-bit key can represent 2^4 different key values, and a 256-bit key can represent 2^{256} different key values. Symmetric encryption keys are typically attacked using brute force, where Eve tries all of the possible keys until she finds the one that Alice and Bob have selected. Protect the confidential messages by making it hard for Eve to try all of the possible keys; the longer you make the key, the longer it takes Eve to check all of the possible key values.

Probability and Brute Force Attacks

When you see calculations that tell you how long it takes to guess a secret key, you should be aware of two assumptions that are rarely stated:

1. Alice and Bob select the secret key randomly, and Eve has no information that would lead her to suspect that checking a specific range of keys is more likely to yield a result than any other range.
2. Eve will have to check all of the possible key values before she finds the correct one.

If you consider these assumptions, you will notice that there is a fundamental conflict. For a randomly selected key, Eve may be lucky and find that the first key that she checks is the one that she is looking for.

It will take Eve 11.6 billion years to try *all* the possible values for a 64-bit key if she can test 50 keys per second, but there is a 50% chance that she will find a match after checking only half the key space and a 1 in 2^{64} chance that the first key she checks will be the one that Alice and Bob have used. When assessing the security of keys, always bear in mind that brute force attacks are subject to the laws of probability, and it's a very optimistic expectation to hope that Eve will have to check all possible keys.

If Eve knows that the key is contained in a specific range, then the situation is even worse: not only is early success still possible but she is also able to reduce the total number of keys to check. See [Chapter 17](#) for more information about selecting and managing keys.

Assess the length of the key you need by estimating how many candidate keys Eve can test per second then compare this with how long you need to maintain confidentiality. For example, if Eve could check 50 candidate keys per second, it would take her 11.6 billion years to check all of the possible values for a 64-bit key. Symmetric cryptography is not subject to the birthday attacks we mentioned in [Chapter 13](#), and calculating the relative security of a key is a simple process. If Eve checks more keys per second, then you will have to adjust your projections accordingly.

The trade-off is that encrypting and decrypting data using longer keys requires more computation, and using very long keys is practical only when you are using high-end computers. Make a reasonable estimate of your confidentiality requirements when selecting a key length and include an estimation of how Eve's ability to perform brute force attacks will change over time. History shows us that improvements in design and manufacturing halve the cost of processing power every 18 months; therefore, over time Eve will be able to increase her ability to check candidate keys for each dollar that she spends.

14.2.5 The .NET Framework Encryption Algorithms

The .NET Framework provides classes for four different symmetric encryptions. [Table 14-1](#) summarizes the encryption algorithms available and the possible secret key lengths.

Table 14-1. Summary of .NET symmetric encryption algorithms

Name	Block size	Key length (bits)
DES	64	56 (although conventionally expressed as a 64-bit number)
RC2	64	40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120, 128
Triple-DES	64	Two or three 56-bit keys, expressed as 64-bit numbers
Rijndael (AES)	128, 192, 256	128, 192, 256

Based on the Lucifer cipher developed by IBM, the Data Encryption Standard (DES) algorithm was published in 1976 and has been used in a huge range of applications. The design has aged remarkably well, given its rocky beginnings. To produce DES, the National Security Agency modified the Lucifer cipher in a way that was thought to compromise security. (A common theory throughout recent cryptographic history is that the National Security Agency is tinkering behind the scenes to limit the efficacy of cryptography.) The limit of a 56-bit key means that DES is becoming increasingly susceptible to exhaustive key searches using modern computers, but various efforts have been made to prolong the life of this venerable cipher, most notably the Triple-DES standard. The Advanced Encryption Standard (AES) officially succeeded DES in 2001 as the principal U.S. federal standard for data encryption.

The RC2 algorithm was developed by Ron Rivest of RSA (also responsible for the MD5 hashing algorithm discussed in [Chapter 13](#)) as a direct replacement for DES. Controversy dogged the acceptance of RC2, because RSA decided to keep the details of the cipher secret, preventing proper security analysis. In 1997, RSA published the details of the algorithm to foster the adoption of the S/MIME standard, which acts as a framework for cryptographically secure email. A wide range of products use RC2, and it has become one of the most commonly used symmetrical ciphers. RC2 operates with key lengths ranging from 40 to 128 bits.

Designed to prolong the life of DES, Triple-DES encrypts data three times using the DES algorithm. A different 56-bit key can be used for each encryption step, or two 56-bit keys can be used and the third encryption step will be performed using the first key. Triple-DES has found a lasting niche in the international financial markets for encrypting banking data.

Rijndael (pronounced "RAIN-Dahl" or "REIN-Daal") was the winner of the U.S. Government's selection process for the AES, which is the successor to the aging DES algorithm. Unlike the other algorithms included in the .NET Framework, the Rijndael cipher function operates on a range of data block sizes. Although tested rigorously during selection, cryptographers have not yet established the long-term security of the Rijndael/AES algorithm. You should exercise caution before using Rijndael, and remember that it takes a great deal of time and analysis before a new algorithm can be fully trusted.

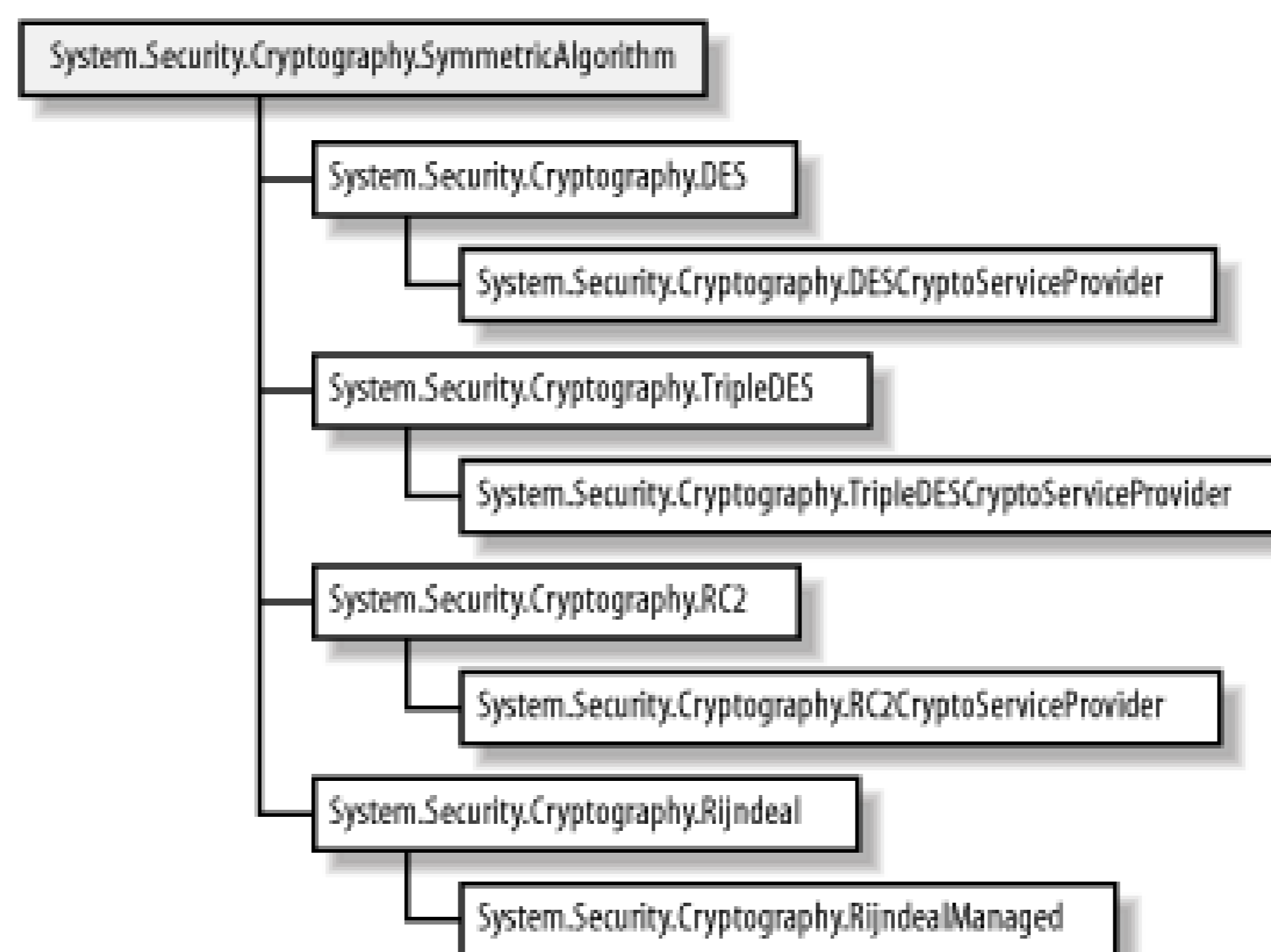
[\[Team LiB \]](#)

[[Team LiB](#)]

14.3 Programming Symmetrical Encryption

The .NET Framework takes the same basic approach to defining symmetric algorithms as the one you saw in [Chapter 13](#) for hashing algorithms; abstract classes extend the `System.Security.Cryptography.SymmetricAlgorithm` class for each of the supported algorithms. Individual implementations of the algorithms extend the abstract class, supporting the possibility of more than one implementation of an algorithm, as represented by [Figure 14-11](#).

Figure 14-11. The .NET Framework class hierarchy for symmetric encryption algorithms



14.3.1 The SymmetricAlgorithm Class

The `SymmetricAlgorithm` class allows you to configure an algorithm (select the block size, padding mode, etc.) and create instances of the classes that encrypt and decrypt data; this class, and the derived implementation classes, are not used to process data directly; see [Section 14.3.4](#) for more details. [Table 14-2](#) summarizes the public members of the `SymmetricAlgorithm` class.

Table 14-2. Members of the SymmetricAlgorithm Class

Member	Description
Properties	
<code>BlockSize</code>	Gets or sets the block size used by the cipher function.

Member	Description
<code>FeedbackSize</code>	Gets or sets the block size used to create feedback when encrypting data.
<code>KeySize</code>	Gets or sets the size in bits of the secret key used by the algorithm.
<code>IV</code>	Get and set the values of the secret key and initialization vector, expressed as an array of bytes.
<code>Key</code>	
<code>LegalBlockSizes</code>	Return the range of block and secret key sizes that the algorithm supports.
<code>LegalKeySizes</code>	
<code>Mode</code>	Gets and sets the cipher mode used to prepare data.
<code>Padding</code>	Gets or sets the padding mode that will fill out partial blocks of data.
<code>Methods</code>	
<code>Create</code>	Creates a new instance of the <code>SymmetricAlgorithm</code> class by name. See the following section for further details.
<code>CreateEncryptor</code>	Create instances of the classes used to encrypt and decrypt data. See Section 14.3.4 for details.
<code>CreateDecryptor</code>	
<code>GenerateIV</code>	Generate random secret keys and initialization vectors.
<code>GenerateKey</code>	
<code>ValidKeySize</code>	Determines if a key of a given length is valid for the algorithm.

14.3.2 Instantiating the Algorithm

You can instantiate the implementation classes for symmetric algorithms in the same way as the hashing algorithms we examined in [Chapter 13](#). The preferred way to create an instance is by using the `Create` method of the `SymmetricAlgorithm` class:

```
# C#
```

```
SymmetricAlgorithm x_alg = SymmetricAlgorithm.Create("RC2");
```

```
# Visual Basic .NET
```

```
Dim x_alg As SymmetricAlgorithm = SymmetricAlgorithm.Create("RC2")
```

The `Create` method instantiates an implementation class based on the value of the argument; [Table 14-3](#) shows the list of supported argument strings and the implementation classes that they create. If no argument is supplied, the `Create` method will instantiate the `RijndaelManaged` class and return `null` (C#) or `Nothing` (Visual Basic .NET) if the argument is not one of those listed in [Table 14-3](#). The system administrator configures the mapping between the string values and the instantiated classes.

Algorithms can be instantiated directly, an approach that is useful if you wish to ensure that a specific implementation class is used or if you are using algorithms for which no mappings are available:

```
# C#

SymmetricAlgorithm x_alg = new RC2CryptoServiceProvider( );

# Visual Basic .NET

Dim x_alg As SymmetricAlgorithm = New RC2CryptoServiceProvider( )
```

Table 14-3. Mapping string values to algorithm classes

String value	Algorithm-implementation class
DES	DESCryptoServiceProvider
System.Security.Cryptography.DES	DESCryptoServiceProvider
3DES	TripleDESCryptoServiceProvider
TripleDES	TripleDESCryptoServiceProvider
Triple DES	TripleDESCryptoServiceProvider
System.Security.Cryptography.TripleDES	TripleDESCryptoServiceProvider
RC2	RC2CryptoServiceProvider
System.Security.Cryptography.RC2	RC2CryptoServiceProvider
Rijndael	RijndaelManaged
System.Security.Cryptography.Rijndael	RijndaelManaged

14.3.3 Configuring the Algorithm

The public properties of the `SymmetricAlgorithm` class provide a mechanism that allows you to configure an algorithm before data is encrypted or decrypted. The following sections describe the configuration options and the impact that each option has.

14.3.3.1 Block and key sizes

The `KeySize` and `BlockSize` properties represent the size (in bits) of the secret key and the size of block processed by the cipher function. Inspect and change the current settings as follows:

```
# C#

SymmetricAlgorithm x_alg = SymmetricAlgorithm.Create("Rijndael");

// print out the current values
```

```

Console.WriteLine("Block Size: {0}", x_alg.BlockSize);
Console.WriteLine("Key Size: {0}", x_alg.KeySize);

// change the values
x_alg.BlockSize = 192;
x_alg.KeySize = 128;

# Visual Basic .NET

Dim x_alg As SymmetricAlgorithm = SymmetricAlgorithm.Create("Rijndael")

' print out the current values
Console.WriteLine("Block Size: {0}", x_alg.BlockSize)
Console.WriteLine("Key Size: {0}", x_alg.KeySize)

' change the values
x_alg.BlockSize = 192
x_alg.KeySize = 128

```

Each algorithm supports a different set of block and key sizes; refer to [Table 14-1](#) for a summary of the sizes supported by the algorithms included in the .NET Framework. Inspect the valid sizes programmatically using the `LegalKeySizes` and `LegalBlockSizes` properties, which rely on the `KeySizes` structure, summarized in [Table 14-4](#).

Table 14-4. The KeySizes structure

Property	Description
<code>MinSize</code>	The smallest valid key size in bits
<code>SkipSize</code>	The interval between valid key sizes in bits
<code>MaxSize</code>	The largest valid key size in bits

Both properties return an array of the `KeySizes` structure, where each element represents a range of keys. To determine the valid key lengths, begin with the smallest valid size and append multiples of the skip size until you reach the largest valid size; for example, if `MinSize` is 16 bits, `SkipSize` is 8 bits, and `MaxSize` is 32 bits, then the valid key lengths would be 16, 24, and 32 bits. The following code demonstrates how to determine all of the valid key lengths for a `SymmetricAlgorithm` instance:

```

# C#

// create the encryption algorithm instance
SymmetricAlgorithm x_alg = SymmetricAlgorithm.Create("Rijndael");

// get the valid sizes for the secret key
KeySizes[] x_size_ranges = x_alg.LegalKeySizes;

// iterate through each KeyRange in the array
foreach (KeySizes x_range in x_size_ranges) {

```

```

// only iterate through if there is a range to process
if (x_range.SkipSize > 0) {
    // calculate each valid size
    for (int i = x_range.MinSize; i <= x_range.MaxSize; i+= x_range.SkipSize) {
        Console.WriteLine("Valid Secret Key Size: {0}", i);
    }
} else {
    // there is no range - only a single value
    Console.WriteLine("Valid Secret Key Size: {0}", x_range.MinSize);
}
}

# Visual Basic .NET

' create the encryption algorithm instance
Dim x_alg As SymmetricAlgorithm = SymmetricAlgorithm.Create("Rijndael")

' get the valid sizes for the secret key
Dim x_size_ranges( ) As KeySizes = x_alg.LegalKeySizes

' iterate through each KeyRange in the array
Dim x_range As KeySizes
For Each x_range In x_size_ranges
    ' only iterate through if there is a range to process
    If x_range.SkipSize > 0 Then
        ' calculate each valid size
        Dim i As Integer
        For i = x_range.MinSize To x_range.MaxSize Step x_range.SkipSize
            Console.WriteLine("Valid Secret Key Size: {0}", i)
        Next
    Else
        ' there is no range - only a single value
        Console.WriteLine("Valid Secret Key Size: {0}", x_range.MinSize)
    End If
Next

```

14.3.3.2 Cipher and padding modes

The .NET Framework supports the two padding modes that we described in [Section 14.2.3](#) earlier in the chapter. A member of the `System.Security.Cryptography.PaddingMode` enumeration, as summarized in [Table 14-5](#), represents each padding mode. The `Padding` property of the `SymmetricAlgorithm` class allows the padding mode to be determined and changed.

Table 14-5. The members of the `PaddingMode` enumeration

Member	Description
--------	-------------

Member	Description
<code>PKCS7</code>	Represents the PKCS #7 padding style, where the value of the padding bytes is the total number of padding bytes added to the partial data block
<code>Zeros</code>	Represents the use of padding bytes that are set to 0

The .NET Framework supports block cipher modes that include those we describe in the "Cipher Modes" section of this chapter, and some additional variations. The members of the `System.Security.Cryptography.CipherMode` enumeration, listed in [Table 14-6](#), represent each cipher mode. The `Mode` property of the `SymmetricAlgorithm` class configures the cipher mode; note that not all encryption algorithms support all of the cipher modes.

Table 14-6. The members of the CipherMode enumeration

Member	Description
<code>ECB</code>	These members represent the modes described in Section 14.2.2 of this chapter.
<code>CBC</code>	
<code>CFB</code>	
<code>CTS</code>	This member represents the "Cipher Text Stealing" mode, which is a variation of the CBC mode that computes the last block of ciphertext in such a way as to ensure that the plaintext and the ciphertext are the same size.
<code>OFB</code>	This member represents the "Output Feedback" mode, which is a variation of the CFB mode, using a different technique to fill the queue.

The following code example demonstrates how to inspect the padding and cipher modes:

```
# C#

// create the encryption algorithm instance
SymmetricAlgorithm x_alg = SymmetricAlgorithm.Create("Rijndael");

// view the current settings
Console.WriteLine("Padding Mode: {0}", x_alg.Padding);
Console.WriteLine("Cipher Mode: {0}", x_alg.Mode);

// change the padding and cipher modes
x_alg.Padding = PaddingMode.Zeros;
x_alg.Mode    = CipherMode.ECB;

# Visual Basic .NET

' create the encryption algorithm instance
Dim x_alg As SymmetricAlgorithm = SymmetricAlgorithm.Create("Rijndael")

' view the current settings
```

```

Console.WriteLine("Padding Mode: {0}", x_alg.Padding)
Console.WriteLine("Cipher Mode: {0}", x_alg.Mode)

' change the padding and cipher modes
x_alg.Padding = PaddingMode.Zeros
x_alg.Mode = CipherMode.ECB

```

14.3.3.3 Keys and initialization vectors (IVs)

The .NET Framework expresses secret keys and initialization vectors (IVs) as arrays of bytes. The `Key` and `IV` properties of the `SymmetricAlgorithm` class allow you to get and set the values.

If you set values using the `Key` and `IV` properties, the `SymmetricAlgorithm` class will use them for encryption or decryption. If you wish to generate random values for the key and IV, simply get the values from the related properties, as illustrated in the following code fragment:

```

# C#

// create the encryption algorithm instance
SymmetricAlgorithm x_alg = SymmetricAlgorithm.Create("DES");

// we are "getting" the value of the secret key, which
// will lead the SymmetricAlgorithm class to create a
// new random key
byte[] x_secret_key = x_alg.Key;

// we are "setting" the value of the secret key, which
// will now be used for any subsequent encryption or
// decryption operations
x_alg.Key = new byte[] {0xD0, 0x8C, 0xD3, 0xEB, 0x10, 0x60, 0x41, 0x59};

# Visual Basic .NET

' create the encryption algorithm instance
Dim x_alg As SymmetricAlgorithm = SymmetricAlgorithm.Create("DES")

' we are "getting" the value of the secret key, which
' will lead the SymmetricAlgorithm class to create a
' new random key
Dim x_secret_key( ) As Byte = x_alg.Key

' we are "setting" the value of the secret key, which
' will now be used for any subsequent encryption or
' decryption operations
x_alg.Key = New Byte( ) { &HD0, &H8C, &HD3, &HEB, &H10, &H60, &H41, &H59 }

```

You do not have to get the values of the `key` and `IV` explicitly to create random values; any encryption or decryption operation will lead to the `SymmetricAlgorithm` implementation class creating new `Key` and `IV` values if they have not been set. If you are using randomly generated values, you must be sure to take note of the `key` and `IV` values used to encrypt the plaintext or you won't be able to decrypt the ciphertext.

14.3.4 Encrypting and Decrypting Data

The `SymmetricAlgorithm` class delegates the process of encrypting and decrypting data to the `ICryptoTransform` interface, which exposes the details of handling data in blocks.

An instance of `ICryptoTransform` transforms plaintext to ciphertext or transforms ciphertext to plaintext; each `ICryptoTransform` is "one-way" and can be used only for the purpose for which it was created. The following statements demonstrate how to create transformations, using the `CreateEncryptor` and `CreateDecryptor` methods:

```
# C#

// create the encryption algorithm
SymmetricAlgorithm x_alg = SymmetricAlgorithm.Create("Rijndael");

// create an ICryptoTransform that can be used to encrypt data
ICryptoTransform x_encryptor = x_alg.CreateEncryptor( );

// create an ICryptoTransform that can be used to decrypt data
ICryptoTransform x_decryptor = x_alg.CreateDecryptor( );

# Visual Basic .NET

' create the encryption algorithm
Dim x_alg As SymmetricAlgorithm = SymmetricAlgorithm.Create("Rijndael")

' create an ICryptoTransform that can be used to encrypt data
Dim x_encryptor As ICryptoTransform = x_alg.CreateEncryptor( )

' create an ICryptoTransform that can be used to decrypt data
Dim x_decryptor As ICryptoTransform = x_alg.CreateDecryptor( )
```

The transformations that these statements create will transform data using the key and IV from the `SymmetricAlgorithm` instance; there are overloaded forms of the `CreateEncryptor` and `CreateDecryptor` classes that accept byte arrays to specify particular key and IV values. [Table 14-7](#) summarizes the members of the `ICryptoTransform` interface.

Table 14-7. The members of the `ICryptoTransform` interface

Member	Description
Properties	
<code>InputBlockSize</code>	Returns the number of bytes that the cipher function operates on
<code>OutputBlockSize</code>	Returns the number of bytes that the cipher function produces when processing a data block of <code>InputBlockSize</code> bytes

Member	Description
<code>CanReuseTransform</code>	Indicates if a transformation can be used to process more than one ciphertext/plaintext
<code>CanTransformMultipleBlocks</code>	If true, the <code>TransformBlock</code> method can accept data in multiples of <code>InputBlockSize</code> bytes
Methods	
<code>TransformBlock</code>	Transforms a region of a byte array and copies the result to a region of an output buffer
<code>TransformFinalBlock</code>	Transforms the final block of data

Instances of the `ICryptoTransform` interface are not useful on their own; the .NET Framework provides the `CryptoStream` companion class, which is the basis for using instances of `ICryptoTransform`.

The `CryptoStream` class acts as a wrapper around a stream and automatically transforms blocks of data using an `ICryptoTransform`. The `CryptoStream` class transforms data read from a stream (for example, decrypting ciphertext stored in a file) or written to a stream (for example, encrypting programmatically generated data and storing the result in a file).

Creating instances of `CryptoStream` requires a real stream, an `ICryptoTransform`, and a value from the `CryptoStreamMode` enumeration, which defines whether to transform the data as it is read from the stream (`CryptoStreamMode.Read`) or as it is written to the stream (`CryptoStreamMode.Write`).

The `CryptoStream` class extends `System.IO.Stream` and exposes all of the functionality of the stream it is wrapped around; requests made to the `CryptoStream` `Read` or `Write` methods are serviced by the underlying stream, and then transformed using an `ICryptoTransform`. [Figure 14-12](#) shows how the `CryptoStream` class transforms data read from a file stream.

Figure 14-12. Using the `CryptoStream` class to transform data

The following statements, [Example 14-1](#), demonstrate how to use the `CryptoStream` class to encrypt data as it is written to a `MemoryStream`, which provides a stream that is backed by an in-memory byte array:

Example 14-1. using the `CryptoStream` class to encrypt data in a stream

```
# C#
```

```
using System;
using System.Security.Cryptography;
using System.IO;
using System.Text;

class MemoryEncryptionExample {

    static void Main( ) {

        // define the message that we will encrypt
        string x_message = "Programming .NET Security";

        // get the bytes representing the message
        byte[] x_plaintext = Encoding.Default.GetBytes(x_message);

        // create the memory stream
        MemoryStream x_memory_stream = new MemoryStream( );

        // create the encryption algorithm
        SymmetricAlgorithm x_alg = SymmetricAlgorithm.Create("RC2");

        // create an ICryptoTransform that can be used to encrypt data
        ICryptoTransform x_encryptor = x_alg.CreateEncryptor( );

        // create the CryptoStream that ties together the FileStream and
        // the ICryptoTransform
        CryptoStream x_cryptostream = new CryptoStream(x_memory_stream,
                                                    x_encryptor,
                                                    CryptoStreamMode.Write);

        // write the plaintext out to the cryptostream
        x_cryptostream.Write(x_plaintext, 0, x_plaintext.Length);

        // close the CryptoStream
        x_cryptostream.Close( );

        // get the ciphertext from the MemoryStream
        byte[] x_ciphertext = x_memory_stream.ToArray( );

        // print out the cipher text bytes
        foreach (byte b in x_ciphertext) {
            Console.Write("{0:X2} ", b);
        }
    }

}

# Visual Basic .NET

Imports System
Imports System.Security.Cryptography
Imports System.IO
Imports System.Text
```

```
Class MemoryEncryptionExample
```

```
    Shared Sub Main( )
```

```
        ' define the message that we will encrypt
        Dim x_message As String = "Programming .NET Security"

        ' get the bytes representing the message
        Dim x_plaintext( ) As Byte = Encoding.Default.GetBytes(x_message)

        ' create the memory stream
        Dim x_memory_stream As MemoryStream = New MemoryStream( )

        ' create the encryption algorithm
        Dim x_alg As SymmetricAlgorithm = SymmetricAlgorithm.Create("RC2")

        ' create an ICryptoTransform that can be used to encrypt data
        Dim x_encryptor As ICryptoTransform = x_alg.CreateEncryptor( )

        ' create the CryptoStream that ties together the FileStream and
        ' the ICryptoTransform
        Dim x_cryptostream As CryptoStream = New CryptoStream(x_memory_stream, _
                                                             x_encryptor, _
                                                             CryptoStreamMode.Write)

        ' write the plaintext out to the cryptostream
        x_cryptostream.Write(x_plaintext, 0, x_plaintext.Length)

        ' close the CryptoStream
        x_cryptostream.Close( )

        ' get the ciphertext from the MemoryStream
        Dim x_ciphertext( ) As Byte = x_memory_stream.ToArray( )

        ' print out the cipher text bytes
        Dim b As Byte
        For Each b In x_ciphertext
            Console.WriteLine("{0:X2} ", b)
        Next

    End Sub
End Class
```

This example provides the general model for encrypting and decrypting data using symmetric algorithms; substitute any stream in the example to write encrypted data using the `CryptoStream` class. Note that the .NET symmetric encryption support makes no special provision for handling data held in memory; the `MemoryStream` class provides the means to process in-memory data using streams.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

14.4 Extending the .NET Framework

In this section, we extend the .NET Framework by implementing the Extended Tiny Encryption Algorithm (XTEA). We have selected XTEA because it is simple to express, allowing us to concentrate on the details of adding a new algorithm rather than the specifics of the algorithm itself. Most symmetric encryption algorithms rely on large lookup tables that speed up encryption, but these are not suitable for listing in this book.

We have provided only a C# implementation of the XTEA algorithm. Like almost all encryption algorithms, XTEA relies on using bitwise shift operations on unsigned data types; this is not possible in Visual Basic .NET without creating additional support functions to compensate for the limited numeric support Visual Basic .NET provides.

14.4.1 The Extended Tiny Encryption Algorithm Explained

The Tiny Encryption Algorithm (TEA) was developed by David Wheeler and Roger Needham of Cambridge University, England. They designed TEA to be simple, quick, and easily expressed in a range of programming languages. Wheeler and Needham released the XTEA in response to a weakness discovered in the original algorithm. Since that time, further weaknesses have emerged with TEA, leading the designers to make more modifications (often referred to as XTEA).

To complicate matters, poor documentation has made differing implementations incompatible. This confusion has restricted the adoption of XTEA, and it is not frequently used. There has been little analysis of the XTEA algorithm, and the security of the design is unknown.

We have selected the XTEA algorithm for clarity, not security. We recommend that you do not use this implementation to protect confidential data.

14.4.2 Defining the Abstract Class

First, create an abstract class representing the XTEA algorithm. The abstract class extends the `System.Security.Cryptography.SymmetricAlgorithm` class and is the foundation for our managed code implementation:

```
using System.Security.Cryptography;

public abstract class XTEA : SymmetricAlgorithm {

}
```

You have created the abstract class to conform to the .NET class model, allowing you to add other XTEA implementations in the future. You can omit the abstract class, but we feel that the extra flexibility of this model is well worth the effort.

14.4.3 Defining the Implementation Class

The `SymmetricAlgorithm` class has two roles: the first is to describe an algorithm and the second is to provide a mechanism for creating instances of `ICryptoTransform` to encrypt or decrypt data. [Table 14-8](#) lists the `SymmetricAlgorithm`-protected fields, each of which, with the exception of `LegalKeySizesValue` and `LegalBlockSizesValue`, can also be get and set using a corresponding property; each property holds data that describes an aspect of the encryption algorithm configuration.

Table 14-8. The protected fields of the `SymmetricAlgorithm` class

Name	Data type	Description
<code>BlockSizeValue</code>	<code>int</code>	The cipher function block size in bits
<code>FeedbackSizeValue</code>	<code>int</code>	The number of data bits fed back into the cipher function in chained cipher modes
<code>IVValue</code>	<code>byte[]</code>	The value of the initialization vector
<code>KeySizeValue</code>	<code>int</code>	The size of the secret key in bits
<code>KeyValue</code>	<code>byte[]</code>	The value of the secret key
<code>LegalBlockSizeValue</code>	<code>KeySize</code>	The range of data block sizes that can be processed by the cipher function
<code>LegalKeySizeValue</code>	<code>KeySize</code>	The range of key sizes supported by the algorithm
<code>ModeValue</code>	<code>CipherMode</code>	Represents the cipher mode
<code>PaddingValue</code>	<code>PaddingMode</code>	Specifies the technique to use in padding incomplete data blocks

We define the `XTEAManaged` class as our implementation of the XTEA algorithm. The XTEA cipher operates on 64-bit blocks of data and uses a 128-bit secret key. Use the class constructor to set these values, implementing the `LegalKeySizeValue` and `LegalBlockSizeValue` fields:

```
using System.Security.Cryptography;

public class XTEAManaged : XTEA {

    public XTEAManaged( ) {
        // define the valid block and key sizes
        LegalKeySizesValue = new KeySizes[] {new KeySizes(128, 128, 0)};
        LegalBlockSizesValue = new KeySizes[] {new KeySizes(64, 64, 0)};
    }
}
```

The `GenerateKey` and `GenerateIV` methods are marked as abstract in the `SymmetricAlgorithm` class; therefore, they must be included in the implementation class; these methods must create a random secret key or initialization vector that is subject to the design of the algorithm. Create a 128-

bit secret key and a 64-bit IV using the random number generator; see [Chapter 17](#) for more information about creating and managing keys.

```

public override void GenerateIV( ) {
    // create the PRNG
    RandomNumberGenerator x_random = RandomNumberGenerator.Create( );
    // create the data array that will hold the IV
    IVValue = new byte[8];
    x_random.GetBytes(IVValue);
}

public override void GenerateKey( ) {
    // create the PRNG
    RandomNumberGenerator x_random = RandomNumberGenerator.Create( );
    // create the data array that will hold the IV
    KeyValue = new byte[16];
    x_random.GetBytes(KeyValue);
}

public override ICryptoTransform CreateDecryptor(byte[] p_key, byte[] p_iv) {
    return new XTEAManagedDecryptor(p_key);
}

public override ICryptoTransform CreateEncryptor(byte[] p_key, byte[] p_iv) {
    return new XTEAManagedEncryptor(p_key);
}
}

```

The remaining methods create implementations of the `ICryptoTransform` interface, called `XTEAManagedEncryptor` and `XTEAManagedDecryptor`. Notice that you do not pass the `p_iv` parameter into the constructor for the `ICryptoTransform` implementations; this is because our XTEA implementation supports only the ECB cipher mode; therefore, you do not need to use an IV as a seed value for cipher function chaining.

14.4.4 Defining an Abstract Transformation

We have defined an abstract class that you used as the basis for the `ICryptoTransform` implementation classes. The main purpose of this class is to provide a common location for utility methods and `ICryptoTransform` interface members that are the same in both the encryption and decryption classes:

```

using System;
using System.Security.Cryptography;

public abstract class XTEAManagedAbstractTransform : ICryptoTransform {

    protected uint[] o_key;    // the key, expressed as a series of 32bit words

    public XTEAManagedAbstractTransform(byte[] p_key) {
        // convert the key from an array of bytes into
        // and array of unsigned 32-bit words, as required
    }
}

```

```

        // by the XTEA specification
        o_key = ConvertBytesToUints(p_key, 0, p_key.Length);
    }

```

The constructor simply converts the secret key from a byte array, as expressed by the .NET Framework, to an array of unsigned 32-bit integers as required by the XTEA algorithm; store the key as an instance array. The XTEA algorithm has been designed with unsigned 32-bit integers in mind, and you can use the `ConvertBytesToUints` and `ConvertUintsToBytes` methods in the encryption and decryption transformations to convert to and from the byte representation that the .NET Framework uses:

```

protected uint[] ConvertBytesToUints(byte[] p_data, int p_offset,
int p_count) {

    // allocate an array - each uint requires 4 bytes
    uint[] x_result = new uint[p_count/4];

    // run through the data and create the unsigned ints from
    // the array of bytes
    for (int i = p_offset, j = 0; i < p_offset + p_count; i+=4, j++) {
        x_result[j] = BitConverter.ToUInt32(p_data, i);
    }

    // return the array of 32-bit unsigned ints
    return x_result;
}

protected byte[] ConvertUintsToBytes(uint[] p_data) {
    // convert the uint[] into a byte[]
    byte[] x_result = new byte[p_data.Length * 4];

    // run through the data and create the bytes from
    // the array of unsigned ints
    for (int i = 0, j = 0; i < p_data.Length; i++, j+=4) {
        byte[] x_interim = BitConverter.GetBytes(p_data[i]);
        Array.Copy(x_interim, 0, x_result, j, x_interim.Length);
    }

    // return the array of 8-bit bytes
    return x_result;
}

```

The `ICryptoTransform` interface requires the following properties, which describe the algorithm used by the transformation. Some encryption algorithms (for example, Rijndael/AES) support a range of block and key sizes, and it can be useful to examine the settings for a specific transformation, especially if the `SymmetricAlgorithm` instance used to create it is not available.

The XTEA algorithm accepts and produces 64-bit blocks of data, and so the `InputBlockSize` and `OutputBlockSize` properties both return "8", representing the number of bytes-note that unlike the `SymmetricAlgorithm` class, the `ICryptoTransform` interface expresses blocks in bytes rather than bits.

The `CryptoStream` class uses the `CanTransformMultipleBlocks` property when breaking up data for processing. If this property returns `true`, then the implementation of the `TransformBlock` method will process data arrays that are multiples of `InputBlockSize`. The implementations of XTEA processes only 64-bit blocks, and for simplicity, return `false` from this method, indicating that `CryptoStream` should only request processing for data in 64-bit blocks and not multiples thereof. Set the `CanReuseTransform` property always to return `false` so that our XTEA transformation can be used only to process one set of data-to either encrypt a single plaintext or decrypt a single ciphertext:

```
public bool CanReuseTransform {
    get {
        return false;
    }
}

public bool CanTransformMultipleBlocks {
    get {
        return false;
    }
}

public int InputBlockSize {
    get {
        return 8;
    }
}

public int OutputBlockSize {
    get {
        return 8;
    }
}
```

The remaining members of this class are included simply to comply with the interface requirements; the `Dispose` method is useful if your algorithm uses unmanaged code (for accessing specialized acceleration hardware or key readers), but the XTEA implementation is written using only managed code; therefore you have no need to use statements to clean up resources. The definitions for `TransformBlock` and `TransformFinalBlock` are `abstract`, because they represent the point of specialization for the encryption and decryption transformations:

```
public virtual void Dispose( ) {
    // we use no unmanaged resources
}

public abstract int TransformBlock(byte[] p_buffer, int p_offset,
    int p_count, byte[] p_output, int p_output_offset);

public abstract byte[] TransformFinalBlock(byte[] p_buffer, int p_offset,
    int p_count);
}
```

14.4.5 Defining the Encryption Transformation

The `XTEAManagedEncryptor` class is instantiated by the `XTEAManaged.CreateEncryptor` method (see [Section 14.4.2](#)) and is used to transform blocks of plaintext into ciphertext:

```
using System;
using System.Security.Cryptography;

public class XTEAManagedEncryptor : XTEAManagedAbstractTransform {

    public XTEAManagedEncryptor(byte[] p_key) : base(p_key) {
        // do nothing
    }
}
```

The class constructor does nothing other than set the key in the base class for future use. The private `XTEAEncrypt` method is the encrypting version of the XTEA cipher function; this method converts a series of bytes to unsigned 32-bit integers, encrypts them, and then converts the results back into a byte array. We have listed the `XTEAEncrypt` method for completeness, but we will treat the working of the XTEA cipher as a black box and will not explain the way that the cipher works:

```
private byte[] XTEAEncrypt(byte[] p_data, int p_offset, int p_count) {

    // defintions that are used to create the cipher text
    uint x_sum = 0, x_delta = 0x9e3779b9, x_count = 32;

    // convert the plaintext data into 32 bit words
    uint[] x_words = ConvertBytesToUints(p_data, p_offset, p_count);

    // main part of the XTEA Cipher
    while(x_count-->0) {
        x_words[0] += (x_words[1] << 4 ^ x_words[1] >> 5) + x_words[1] ^
            x_sum + o_key[x_sum & 3];
        x_sum += x_delta;
        x_words[1] += (x_words[0] << 4 ^ x_words[0] >> 5) + x_words[0] ^
            x_sum + o_key[x_sum >> 11 & 3];
    }

    // convert the cipher block into bytes and return the result
    return ConvertUintsToBytes(x_words);
}
```

The `CryptoStream` class uses the `TransformBlock` method to encrypt blocks of data that are the same size as the return value of the `InputBlockSize` property—for XTEA this means that you will process eight bytes of data at a time.

If the `CanTransformMultipleBlocks` property returns `true`, then `CryptoStream` may request that the transformation encrypt bytes in multiples of `InputBlockSize`, but will never request that the `TransformBlock` method process a partial data block. The implementation only transforms single data blocks; therefore, you work on the basis that `CryptoStream` will only request a transformation for eight bytes, which is the input for the `XTEAEncrypt` cipher function:

```
public override int TransformBlock(byte[] p_buffer, int p_offset, int p_count,
    byte[] p_output, int p_output_offset) {
```

```

// get the cipher text for the block
byte[] x_ciphertext = XTEAEncrypt(p_buffer, p_offset, p_count);

// copy the ciphertext into the output buffer
Array.Copy(x_ciphertext, 0, p_output, p_output_offset,
           x_ciphertext.Length);

// return the number of bytes written out
return x_ciphertext.Length;
}

```

The `CryptoStream` class calls the `TransformFinalBlock` after the `TransformBlock` method has processed all of the complete data blocks. This method allows you to pad out any partial data and is called even if there is no partial data to process (in which case the value of `p_count` is 0). The XTEA algorithm requires padding (because the cipher function works only on complete 64-bit blocks), so use "PKCS#7" padding.



Algorithm implementations are directly responsible for supporting cipher modes, which is an unexpected omission from the .NET abstract classes, given the way that the `Cryptostream` class is used by the .NET Framework to break data into fixed-sized blocks. If you are implementing a new algorithm, you must cater to all of the cipher modes that you wish to support. For simplicity, use the ECB cipher mode to process data for our XTEA implementation only.

```

public override byte[] TransformFinalBlock(byte[] p_buffer,
                                           int p_offset, int p_count) {

    // create an array which is the size of an input block
    byte[] x_result = new byte[InputBlockSize];

    // copy in the data from the buffer to the array
    Array.Copy(p_buffer, p_offset, x_result, 0, p_count);

    // work out the number of bytes we need to add to the
    // array to ensure that we have a complete block
    int x_shortfall = InputBlockSize - ((p_count - p_offset));

    // pad out the data by adding bytes
    for (int i = p_count - p_offset; i < x_result.Length; i++) {
        x_result[i] = (byte)x_shortfall;
    }

    // encrypt and return the data block
    return XTEAEncrypt(x_result, 0, x_result.Length);
}
}

```

The effect of padding out partial data blocks is that the ciphertext you produce can be longer than the plaintext, most notably, add a complete block of padding (8 bytes) when the plaintext contains only full data blocks.

14.4.6 Defining the Decryption Transformation

The `XTEAManagedDecryptor` class is instantiated by the `XTEAManaged.CreateDecryptor` method (see [Section 14.4.3](#)) and is used to transform blocks of ciphertext into plaintext, a task complicated by the way that we pad partial blocks of data during encryption.

This padding strategy means that you will have to decrypt complete data blocks only, but you have to process the final block to remove the padding. The problem is that the `CryptoStream` class will call the `TransformFinalBlock` method only after the `TransformBlock` method has processed all of the data blocks.

The solution to this problem is to hold onto each block of plaintext until the next time that the `TransformBlock` or `TransformFinalBlock` methods are called. You know that you have retained the last data block if the `CryptoStream` has invoked the `TransformFinalBlock` method and you can safely remove the padding:

```
using System;

public class XTEAManagedDecryptor : XTEAManagedAbstractTransform {
    private byte[] o_data_block;

    public XTEAManagedDecryptor(byte[] p_key) : base(p_key) {
        // initialize the data block
        o_data_block = new byte[0];
    }

    private byte[] XTEADecrypt(byte[] p_data, int p_offset, int p_count) {
        // definitions that are used to restore the plaintext text
        uint x_count = 32, x_sum = 0xC6EF3720, x_delta = 0x9E3779B9;

        // convert the data into 32 bit words
        uint[] x_words = ConvertBytesToUints(p_data, p_offset, p_count);

        // main part of the XTEA Cipher
        while(x_count-- > 0) {
            x_words[1] -= (x_words[0] << 4 ^ x_words[0] >> 5) + x_words[0]
                ^ x_sum + o_key[x_sum >> 11 & 3];
            x_sum -= x_delta;
            x_words[0] -= (x_words[1] << 4 ^ x_words[1] >> 5) + x_words[1]
                ^ x_sum + o_key[x_sum & 3];
        }

        // convert the uint[] into a byte[]
        return ConvertUintsToBytes(x_words);
    }
}
```

The class constructor sets the key in the base class and initializes the byte array that you use to hold back plaintext blocks. The private `XTEADecrypt` method is the decryption version of the XTEA cipher function; this method converts a series of bytes to unsigned 32-bit integers, decrypts them and then converts the results back into a byte array.

The `TransformBlock` implementation decrypts the ciphertext block and assigns the result to the

`o_data_block` field. Write any data held from previous invocations to the output buffer:

```
public override int TransformBlock(byte[] p_buffer, int p_offset, int p_count,
    byte[] p_output, int p_output_offset) {

    // define the number of data bytes we copied
    int x_result = 0;

    if (o_data_block.Length > 0) {
        // there is some data from a previous call - write it out
        Array.Copy(o_data_block, 0, p_output, p_output_offset,
            o_data_block.Length);
        x_result = o_data_block.Length;
    }

    // decrypt the ciphertext block we are bring passed and store it
    o_data_block = XTEADecrypt(p_buffer, p_offset, p_count);

    // return the number of bytes that we have written
    // to the output buffer
    return x_result;
}
```

You know that the "held" data is the final plaintext block when the `TransformFinalBlock` method is invoked. Remove the PKCS #7 padding and copy the rest of the data into a new byte array, which returns as the method result, completing our implementation of the XTEA cipher:

```
public override byte[] TransformFinalBlock(byte[] p_buffer, int p_offset,
    int p_count) {

    // get the value of the last byte, which tells us
    // how many padding bytes there are
    int x_padding_count = (int)o_data_block[o_data_block.Length - 1];

    // create the shortened data block
    byte[] x_final_block = new byte[o_data_block.Length - x_padding_count];

    // copy in the data from the block
    Array.Copy(o_data_block, 0, x_final_block, 0, x_final_block.Length);

    // return the final plaintext block
    return x_final_block;
}
}
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 15. Asymmetric Encryption

In the previous chapter, we discussed symmetric encryption and the support available in the .NET Framework. The word "symmetric" is an appropriate description because both Alice and Bob must know the secret key used to encrypt data in order to read it, leading to the problem of agreeing on the key in such a way that Eve is unable to eavesdrop. In this chapter, we discuss an alternative approach to message confidentiality known as "asymmetric encryption." We'll explain what it is, how it works, and how it overcomes the problem of exchanging secret keys.

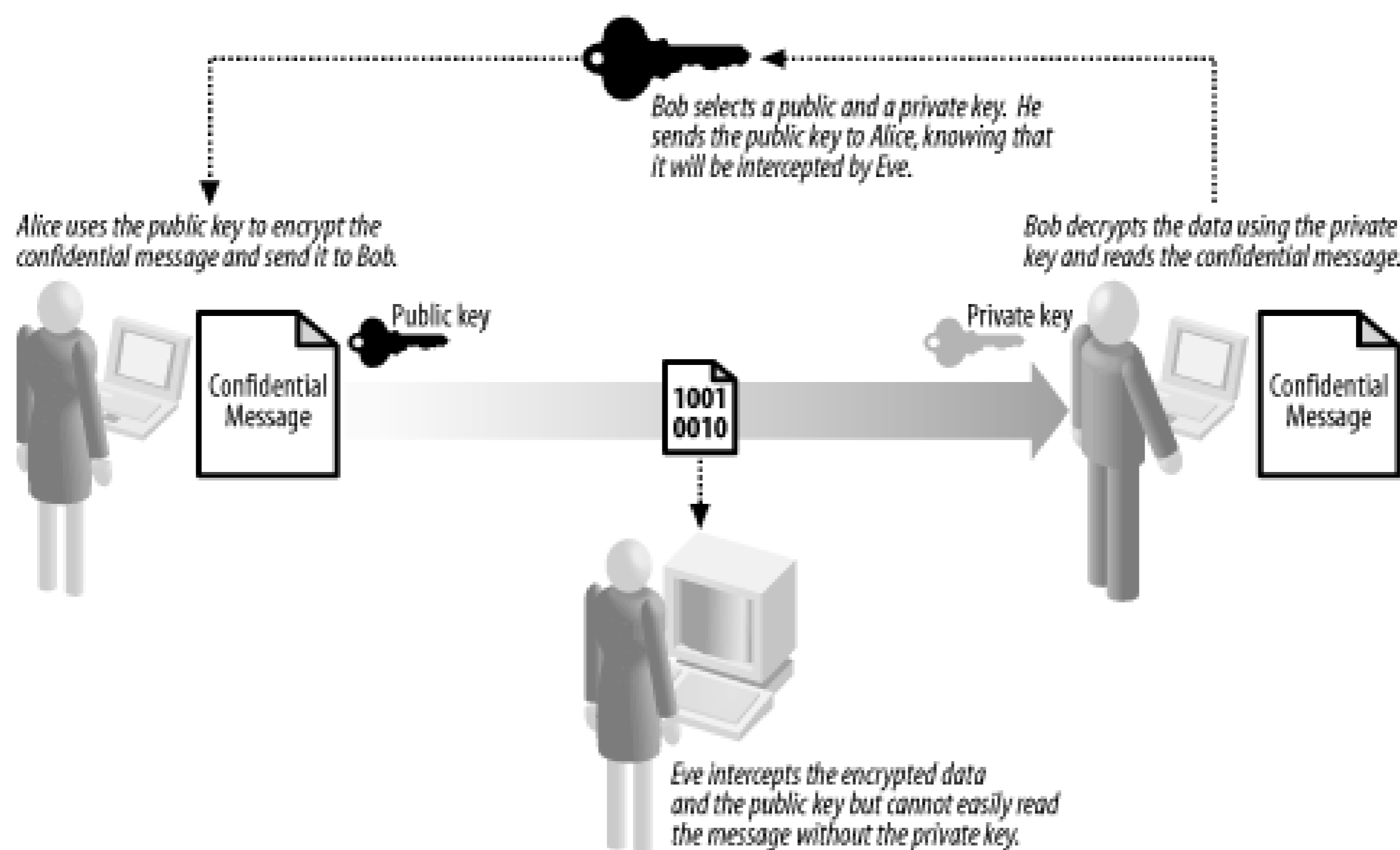
[\[Team LiB \]](#)

[[Team LiB](#)]

15.1 Asymmetric Encryption Explained

Asymmetric encryption, often called "public key" encryption, allows Alice to send Bob an encrypted message without a shared secret key; there *is* a secret key, but only Bob knows what it is, and he does not share it with anyone, including Alice. [Figure 15-1](#) provides an overview of this asymmetric encryption, which works as follows:

Figure 15-1. Asymmetric encryption does not require Alice and Bob to agree on a secret key



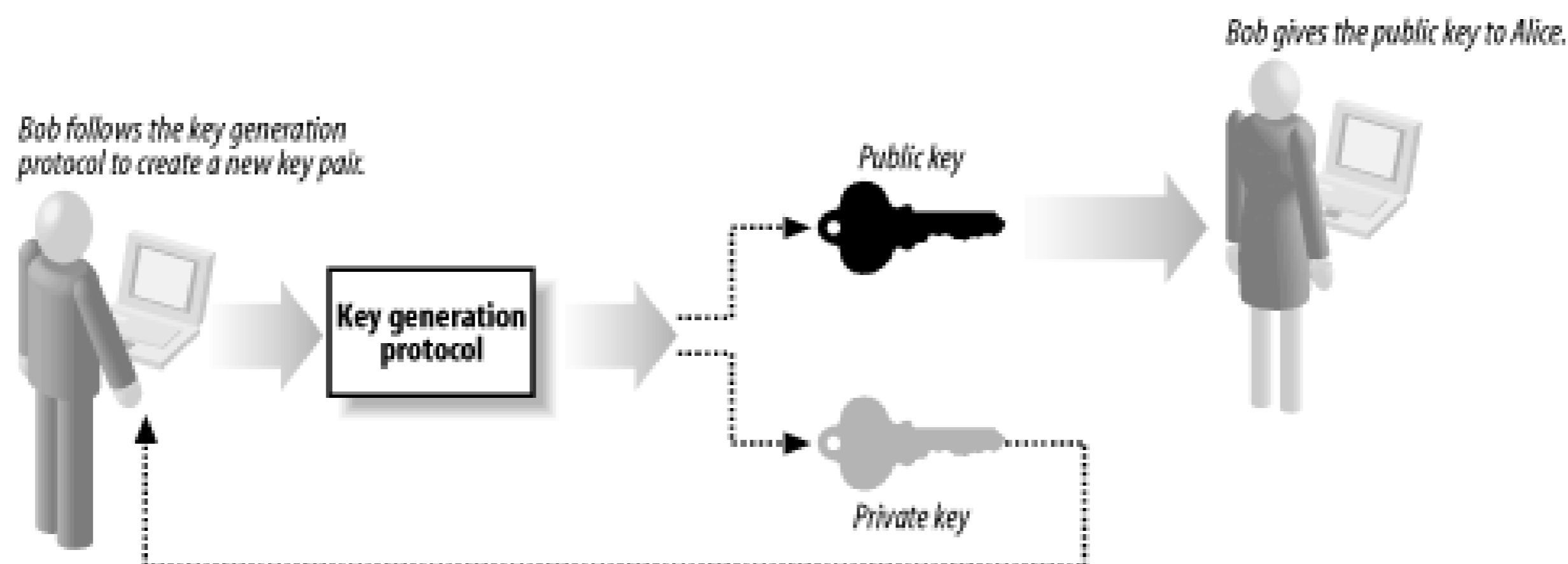
1. Bob creates a pair of keys, one of which he keeps secret and one of which he sends to Alice.
2. Alice composes a confidential message and encrypts it using the key that Bob has sent to her.
3. Alice sends the encrypted data to Bob.
4. Bob uses his secret key to decrypt the data and reads the confidential message.

The key that Bob sends to Alice is the public key, and the key he keeps to himself is the "private" key; jointly, they form Bob's "key pair."

The most striking aspect of asymmetric encryption is that Alice is not involved in selecting the key: Bob creates a pair of keys without any input or agreement from Alice and simply sends her the public key. Bob retains the private key and keeps it secret. Alice uses an asymmetric algorithm to encrypt a message with Bob's public key and sends him the encrypted data, which he decrypts using the private key.

Asymmetric algorithms include a "key generation" protocol that Bob uses to create his key pair, as shown by [Figure 15-2](#). Following the protocol results in the creation of a pair of keys that have a mathematical relationship—the exact detail of the protocol and the relationship between the keys is different for each algorithm.

Figure 15-2. Bob uses a key generation protocol to create a new key pair



When we talk about an asymmetric encryption algorithm, we are actually referring to two related functions that perform specific tasks: an encryption function that encrypts a message using a public key, and a decryption function that uses a secret key to decrypt a message encrypted with the corresponding public key.

The encryption function can *only* encrypt data. Alice cannot decrypt ciphertext that she has created using the encryption function. This means that Bob can send his public key to several people, and each of them can create ciphertext that only Bob's secret key can decrypt, as shown in [Figure 15-3](#).

Figure 15-3. Alice and Anthony are able to use the same public key to create ciphertext that can only be decrypted using Bob's secret key

The one-way nature of the encryption function means that messages created by one sender cannot

be read by another (i.e., Alice cannot decrypt the ciphertext that Anthony has created, even though they both have Bob's public key). Bob can give out the public key to anyone who wants to send him a message, and he can even print his public key on his business card and hand it out to anyone who might want to send him a message. He can add the public key to an Internet directory of keys, allowing people Bob has never met to create messages that only he can read.

If Bob suspects that Eve has guessed his private key, he simply creates a new key pair and sends out the new public key to anyone who might send him a message. This is a lot easier than arranging to meet in a secure location to agree on a new symmetric secret key with every person that might want to communicate with him. In practice, the process of changing key pairs is more complex, and we have more to say on this topic in [Chapter 17](#).

Bob's pair of keys allows Alice to send him encrypted messages, but Bob cannot use them to send a message back to Alice because of the one-way nature of the encryption and decryption functions. If Bob needs to send Alice a confidential message, Alice must create her own pair of keys and send the public key to Bob, who can then use the encryption function to create ciphertext that only Alice can decrypt with her private key.

The main limitation of public key encryption is that it is very slow relative to symmetric encryption and is not practical for encrypting large amounts of data. In fact, the most common use of public key encryption is to solve the key agreement problem for symmetric encryption algorithms, which we discuss in more detail in [Chapter 17](#).

In the following sections, we demonstrate how an asymmetric encryption algorithm works. We use the RSA algorithm for our illustration because it is the only one implemented in the .NET Framework. Ronald Rivest, Adi Shamir, and Leonard Adleman created the RSA algorithm in 1977, and the name is the first letter of each of the inventors' last names. The RSA algorithm is the basis for numerous security systems, and remains the most widely used and understood asymmetrical algorithm.

15.1.1 Creating Asymmetric Keys

Most asymmetric algorithms use keys that are very large numbers, and the RSA algorithm is no exception. In this section, we demonstrate the RSA key generation protocol and provide you with some general information about the structure and usage of asymmetric keys.

We step through the RSA key generation protocol, using small test values. The protocol is as follows:

1. Choose two large random prime numbers, p and q , of equal length and multiply them together to create n , the RSA key modulus.

Select p as 23 and q as 31, so that the modulus, n , is:

$$n = p \times q = 23 \times 31 = 713$$

2. Randomly choose e , the public exponent, so that e and $(p-1)(q-1)$ are relatively prime.

Numbers are "relatively" prime when they share no common factors except 1. For these test values
select a value for e that has no common factors with 660. Select e as 19.

3. Compute the private exponent, d , where $d = e^{-1} \bmod ((p-1)(q-1))$.

For our example, we calculate d as follows:

$$d = 19^{-1} \bmod (22 \times 30) = 139$$

4. The public key consists of e and n . The private key is d . Discard p and q , but do not reveal their values.

You can see how simple it is to create an RSA key pair. Bob sends the value of e (19) and n (713) to Alice and keeps the value of d (139) secret. Most asymmetric encryption algorithms use a similar approach to key generation; we explain the protocol for a different asymmetric algorithm in [Section 15.3](#).

Random Prime Numbers

Selecting prime numbers at random is a requirement of many different key generation protocols. It is very time-consuming to check that a random number is truly a prime number, especially when dealing with numbers that have hundreds of digits.

The compromise between the need for prime numbers and the need to generate them in a reasonable time is to create numbers that are "probably" prime, which means that there is a small possibility that a number seems to be a prime number, but is actually not. Probable prime numbers are subject to a level of confidence, so that a level of 16 means that the probability that a number is a true prime number exceeds:

Some asymmetric encryption algorithms are significantly less secure if numbers that are supposed to be prime numbers turn out not to be, and so care must be taken when generating numbers with a low level of confidence. There is a balance between the confidence in a prime number and the amount of computation that is required to attain that confidence level.

15.1.2 Asymmetric Algorithm Security

Asymmetric algorithms use much longer keys than symmetric algorithms. In our examples, we selected small values to demonstrate the key generation protocol, but the numeric values used in practice contain many hundreds of digits.

Measure asymmetric key lengths in bits. The way to determine the number of bits differs between algorithms. The RSA algorithm specifies that the key length is the smallest number of bits needed to represent the value of the key modulus, n , using binary. Round up the number of bits to a factor of eight so that you can express the key using bytes; for example, consider a modulus represented by 509 bits to be a 512-bit key. Common lengths for keys are 512 and 1024-bits, but a 1024-bit asymmetric key does not provide 16 times more resistance to attack than a 64-bit symmetric key. [Table 15-1](#) lists the equivalent asymmetric and symmetric key lengths accepted as providing equivalent resistance to brute force attacks (where Eve obtains the value of the private/secret key by testing all of the possible key values).

Most asymmetric algorithms rely on some form mathematical task that is difficult or time-consuming to perform. Cryptographers consider the RSA algorithm secure because it is hard to find the factors of a large number; given our example value for n (713), it would take some time to establish that the factors are the values you selected for p (23) and q (31). The longer the value of n , the longer it takes to determine the factors; bear in mind that the example value of n has only five digits, whereas the numbers that you would use for a secure key pair will be significantly longer.

Table 15-1. Asymmetric and symmetric key lengths providing equivalent resistance to brute force attacks

Symmetric key length	Asymmetric key length
64 bits	512 bits
80 bits	768 bits
112 bits	1792 bits
128 bits	2304 bits

Therefore, the essence of security for RSA is that given only the public key e and n , it takes a lot of computation to discover the private key d . Once you know the factors p and q , it is relatively easy to calculate d and decrypt ciphertext. Bob makes it difficult for Eve to decrypt his messages by keeping secret the values of d , p and q .

The main risk with asymmetric algorithms is that someone may discover a technique to solve the mathematical problems quickly, undermining the security of the algorithm. New techniques to factor large numbers might make it a simple process to decrypt messages or to deduce the value of the private key from the public key, and this would render the RSA algorithm (and any others that rely on the same mathematical problem) insecure.

15.1.3 Creating the Encrypted Data

We have already explained how the encryption and decryption functions are at the heart of an asymmetric algorithm, and the way in which we use these functions is similar to the techniques we discussed in [Chapter 14](#). The protocol for encrypting data using an asymmetric algorithm is as follows:

1. Break the plaintext into small blocks of data.
2. Encrypt each small plaintext block by using the public key and the encryption function.
3. Concatenate the encrypted blocks to form the ciphertext.

The length of the public key determines the size of the block to which we break the plaintext. Each algorithm specifies a rule for working out how many bytes of data should be in each block, and for the RSA protocol, we subtract 1 from the key length (in bits) and divide the result by 8. The integral part of the result (the part of the number to the left of the decimal point) tells you how many bytes

of data should be in each block of plaintext passed that is to the encryption function. You can work out how many bytes should be in a plaintext block for a 1024-bit key, as follows:

$$\text{length} = (1024 - 1) \div 8 = 127.875$$

The integral value of the result is 127, meaning that when using the RSA algorithm with a 1024-bit public key we should break the plaintext into blocks of 127 bytes. The small key that you generated to demonstrate the key generation protocol is 16 bits long (713 is 1011001001 in binary and the bit length is therefore rounded up to be 16 bits) and with a 16-bit key, we must use 1-byte blocks (the integral part of $(10 - 1) / 8 = 1.875$ is 1).

Encrypt each data block by interpreting it as an integer value and compute a ciphertext block using the RSA encryption function shown below (m is the integer value of the plaintext block and c is the ciphertext block):

$$c = m^e \bmod n$$

[Figure 15-4](#) demonstrates how this process works for a 24-bit key, meaning that you process 2 bytes of plaintext at a time; the figure shows how the encryption function is applied to encrypt the string ".NET" into the ciphertext "35 7B AE 05 F1 6F."

Figure 15-4. Using the RSA encryption function to encrypt the string .NET using a 24-bit public key

For reference, we created our 24-bit key using 1901 for p and 1999 for q . We chose e to be 19, giving a secret key value, d , of 2805887. Notice that the output of the cipher function is the same length as the key modulus, making the ciphertext larger than the plaintext.

Decrypting data is the reverse of the encryption protocol:

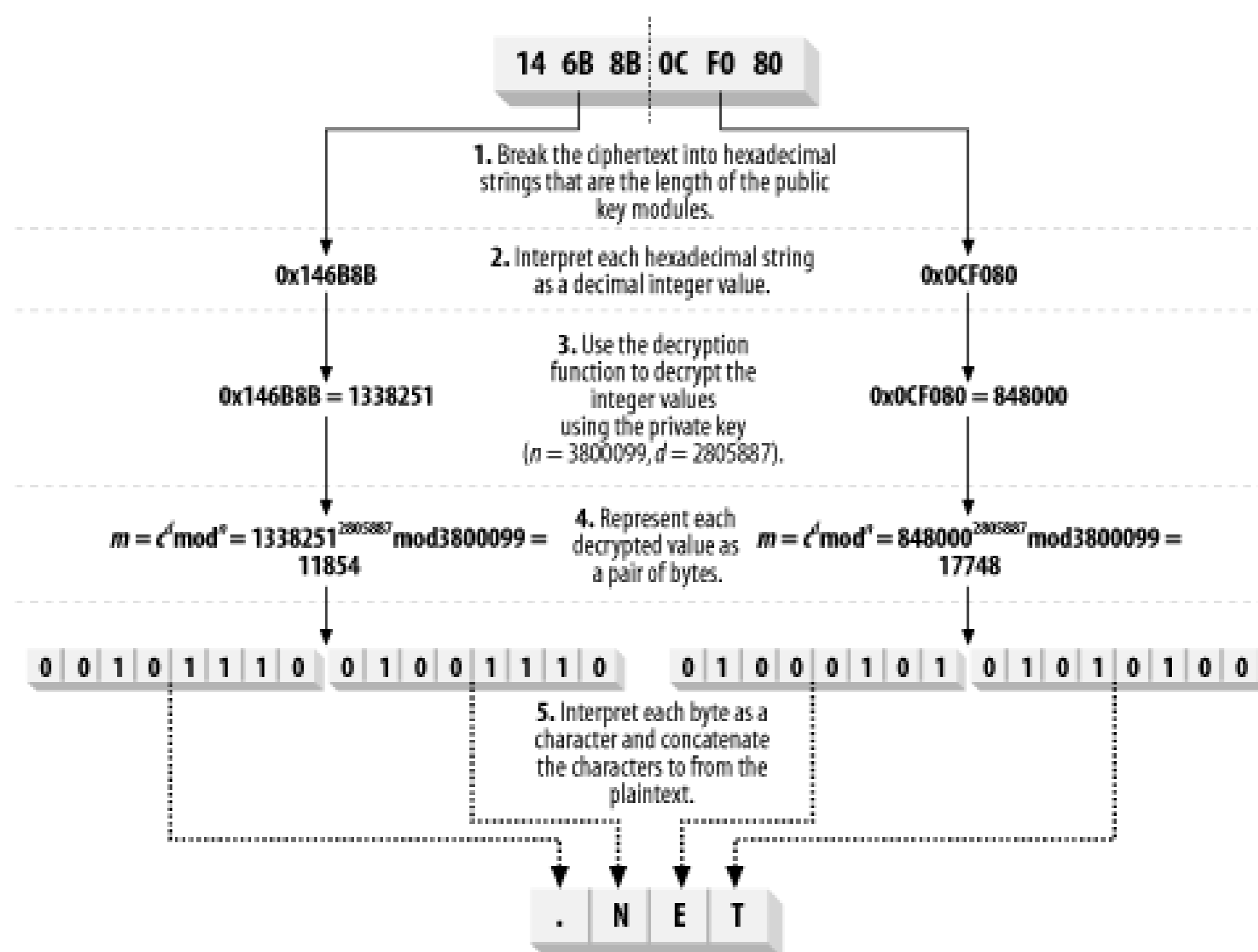
1. Break the ciphertext into small blocks of data that are the same length as the public key modulus.
2. Decrypt each small ciphertext block by using the private key and the decryption function.
3. Concatenate the decrypted blocks to form the restored plaintext.

The decryption function is as follows (c is the value of the ciphertext block and m is the plaintext block):

$$m = c^d \bmod n$$

Notice that the decryption function uses the secret key (d) and the modulus from the public key (n). [Figure 15-5](#) demonstrates how this process works for our 24-bit key, meaning that we process 3 bytes of ciphertext at a time; the figure shows how we decrypt the ciphertext that we created in [Figure 15-4](#).

Figure 15-5. Using the RSA decryption function



15.1.3.1 Asymmetric data padding

Asymmetric encryption algorithms rely on padding to protect against specific kinds of attack, in much the same way that symmetric algorithms rely on cipher feedback. Padding schemes also ensure that the encryption function does not have to process partial blocks of data.

Asymmetric padding schemes are a series of instructions that specify how to prepare data before encryption, and usually mix the plaintext with other data to create a ciphertext that is much larger than the original message. The .NET Framework supports two padding schemes for the RSA

algorithm: Optimal Asymmetric Encryption Padding (OAEP) and PKCS #1 v1.5. OAEP is a newer scheme that provides protection from attacks to which the PKCS #1 v1.5 scheme is susceptible. You should always use OAEP, unless you need to exchange encrypted data with a legacy application that expects PKCS #1 v1.5 padding.

We do not discuss the details of either padding scheme in this book; it is important only that you understand that padding is used in conjunction with the asymmetric algorithm to further protect confidential data.

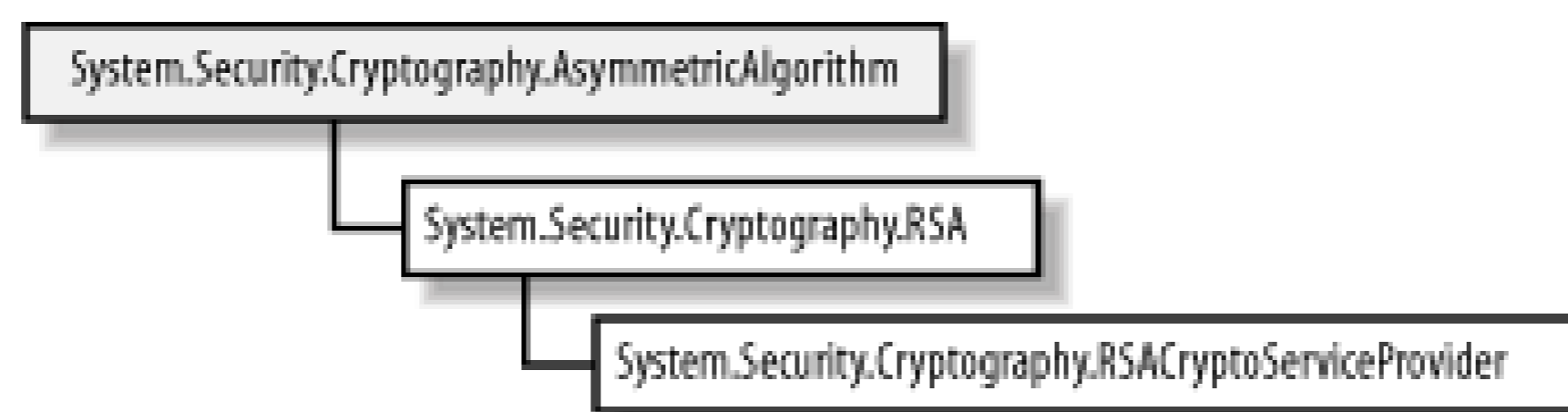
[\[Team LiB \]](#)

[Team LiB]

15.2 Programming Asymmetrical Encryption

The .NET Framework takes the same approach to representing asymmetric algorithms as it does for symmetric algorithms and hashing algorithms; abstract classes extend the `System.Security.Cryptography.AsymmetricAlgorithm` class for each of the supported algorithms. Individual implementations of the algorithms extend the abstract class, supporting the possibility of more than one implementation of an algorithm. Figure 15-6 shows the class hierarchy that supports asymmetric encryption algorithms, which is very simple because the .NET Framework supports only the RSA algorithm.

Figure 15-6. The .NET Framework class hierarchy for asymmetric encryption algorithms



15.2.1 The AsymmetricAlgorithm class

Different asymmetric algorithms can take very different approaches to encrypting data, and the abstract `AsymmetricAlgorithm` class provides very little in the way of common functionality. Table 15-2 lists the public methods of the `AsymmetricAlgorithm` class.

Table 15-2. Members of the `AsymmetricAlgorithm` class

Member	Description
Properties	
<code>KeySize</code>	Gets or sets the size in bits of the key modulus used by the algorithm
<code>LegalKeySizes</code>	Gets the key sizes supported by the algorithm
<code>KeyExchangeAlgorithm</code>	Gets the name of the key exchange algorithm
<code>SignatureAlgorithm</code>	Gets the name of the digital signature algorithm; see Chapter 16 for details of digital signatures
Methods	
<code>Create</code>	Creates a new instance of the <code>AsymmetricAlgorithm</code> class by name

Member	Description
<code>ToXmlString</code>	Creates an XML string representation of the <code>AsymmetricAlgorithm</code> instance
<code>FromXmlString</code>	Reconstructs an instance of <code>AsymmetricAlgorithm</code> from an XML string

15.2.2 The RSA class

The `RSA` class is the abstract representation of the RSA algorithm and extends `AsymmetricAlgorithm`. This class provides the common functionality shared by any implementation of the underlying algorithm and allows the algorithm to be employed without regard to the implementation details; however, as you will see in the following sections, the `RSA` class is not as useful as it should be. Table 15-3 summarizes the public members of the `RSA` class.

Table 15-3. Members of the RSA class

Method	Description
<code>DecryptValue</code>	Abstract decryption method-consult the following sections for details of decrypting data with the RSA implementation class
<code>EncryptValue</code>	Abstract encryption method-consult the following sections for details of encrypting data with the RSA implementation class
<code>ExportParameters</code>	Exports the public or private key using the <code>RSAPParameters</code> structure
<code>ImportParameters</code>	Imports a key using the <code>RSAPParameters</code> structure

The `EncryptValue` and `DecryptValue` methods are not as useful as they seem, as we will explain in the next section. The value of the `RSA` class is the addition of the `ImportParameters` and `ExportParameters` methods, which allow the public or private keys to be imported into or exported from an instance of the class. Table 15-4 details the members of the `RSAPParameters` structure, which represents RSA keys as a related series of byte arrays.

Table 15-4. Members of the RSAPParameters structure

Field	Description
<code>Modulus</code>	The key modulus value, n .
<code>Exponent</code>	The key exponent value, e .
<code>D</code>	The secret key value, d .
<code>P</code>	These values are used to increase the performance of the RSA algorithm.
<code>Q</code>	

Field	Description
DP	
DQ	
InverseQ	

You will be familiar with the purpose of the `RSAParameters` values `Modulus`, `Exponent`, and `D` from Section 15.1.1 in this chapter. The other fields represented by the structure can be used to increase the performance of the decryption function. The performance enhancement is not part of the RSA algorithm specification, but the `RSA` class only imports private keys using `RSAParameters` structures that contain all of the additional values. You do not need to understand how the performance enhancements work, but you should be aware that the .NET RSA classes always expect these additional values.

15.2.3 The `RSACryptoServiceProvider` Class

This class is the only asymmetric encryption implementation class contained in the .NET Framework. Unfortunately, while this class extends the abstract `RSA` class, it does not implement the `EncryptValue` and `DecryptValue` methods, meaning that you must work with the implementation class directly to use the RSA algorithm. Table 15-5 details the new methods introduced by the `RSACryptoServiceProvider` class.

Table 15-5. Members of the `RSACryptoServiceProvider` class

Method	Description
<code>Encrypt</code>	Encrypts plaintext, using OAEP or PKCS #1 v1.5 padding.
<code>Decrypt</code>	Decrypts OAEP or PKCS #1 v1.5 padded ciphertext.
<code>SignData</code>	These methods create and verify digital signatures, which we discuss in Chapter 16.
<code>SignHash</code>	
<code>VerifyData</code>	
<code>VerifyHash</code>	

15.2.4 Instantiating the Algorithm

The `RSACryptoServiceProvider` class does not implement all of the methods defined by the abstract `RSA` class. There is no purpose in using the `Create` method specified in the `AsymmetricAlgorithm` class; the abstraction advantages offered with symmetric and hashing algorithms do not apply to RSA encryption, because you must always create instances of the implementation class to be able to use the `Encrypt` and `Decrypt` methods:

```
# C#
```



```

RSACryptoServiceProvider x_alg = new RSACryptoServiceProvider( );

# Visual Basic .NET

Dim x_al2g As RSACryptoServiceProvider = New RSACryptoServiceProvider( )

```

15.2.5 Using RSA Keys

Even though you cannot use the abstract classes to encrypt and decrypt data, the methods that they define do allow you to configure the keys used by instances of the `RSACryptoServiceProvider` class.

The `AsymmetricAlgorithm` class provides the `LegalKeySizes` property, which returns an array of the `KeySizes` structure, specifying all of the key lengths that the algorithm supports. This is the same model used by the `SymmetricAlgorithm` class (see Section 14.3.3.1 of Chapter 14 for full details). The .NET RSA implementation supports key lengths from 384 to 16384 bits, in 8-bit steps, and the size of key you wish to use is set using the `KeySize` property.

The RSA implementation will create a new key pair automatically if you have not imported a key value before you encrypt or decrypt data. You can export the details of this key using the `ExportParameters` method, which returns an instance of the `RSAPParameters` structure. This method accepts a Boolean argument. If the argument is `true`, then the private key is exported; otherwise, only the details of the public key are included:

```

# C#

RSACryptoServiceProvider x_alg = new RSACryptoServiceProvider( );

// export only the public key
RSAPParameters x_public_params = x_alg.ExportParameters(false);

// export the private key
RSAPParameters x_private_params = x_alg.ExportParameters(true);

# Visual Basic .NET

Dim x_alg As RSACryptoServiceProvider = New RSACryptoServiceProvider( )

' export only the public key
Dim x_public_params As RSAPParameters = x_alg.ExportParameters(False)

' export the private key
Dim x_private_params As RSAPParameters = x_alg.ExportParameters(True)

```

The most common use for the `ExportParameters` method is to export details of a key from one instance of an algorithm and then import them to another using the `ImportParameters` method:

```

# C#

// create an instance of the RSA algorithm
RSACryptoServiceProvider x_alg = new RSACryptoServiceProvider( );
// export only the public key
RSAPParameters x_public_params = x_alg.ExportParameters(false);

```

```

// create a second instance of the algorithm
RSACryptoServiceProvider x_second_algorithm = new RSACryptoServiceProvider( );
// import the public key
x_second_algorithm.ImportParameters(x_public_params);

# Visual Basic .NET

' create an instance of the RSA algorithm
Dim x_alg As RSACryptoServiceProvider = New RSACryptoServiceProvider( )
' export only the public key
Dim x_public_params As RSAParameters = x_alg.ExportParameters(False)

' create a second instance of the algorithm
Dim x_second_algorithm As RSACryptoServiceProvider = New RSACryptoServiceProvider( )
' import the public key
x_second_algorithm.ImportParameters(x_public_params)

```

The `RSAParameters` structure is also useful if you need to export a key into a custom format. The following statements demonstrate how to export the public part of a key pair and write it out as a pair of hexadecimal strings:

```

# C#

// create an instance of the RSA algorithm
RSACryptoServiceProvider x_alg = new RSACryptoServiceProvider( );

// export only the public key
RSAParameters x_public_params = x_alg.ExportParameters(false);

Console.WriteLine("n: ");
foreach (byte x_byte in x_public_params.Modulus) {
    Console.WriteLine("{0:X2} ", x_byte);
}
Console.WriteLine( );
Console.WriteLine("e: ");
foreach (byte x_byte in x_public_params.Exponent) {
    Console.WriteLine("{0:X2} ", x_byte);
}

# Visual Basic .NET

' create an instance of the RSA algorithm
Dim x_alg As RSACryptoServiceProvider = New RSACryptoServiceProvider( )
' export only the public key
Dim x_public_params As RSAParameters = x_alg.ExportParameters(False)

Console.WriteLine("n: ")
Dim x_byte As Byte
For Each x_byte In x_public_params.Modulus
    Console.WriteLine("{0:X2} ", x_byte)
Next x_byte

```

```

Console.WriteLine( )
Console.Write("e: ")
For Each x_byte In x_public_params.Exponent
    Console.Write("{0:X2} ", x_byte)
Next x_byte

```

The `ToXmlString` method exports the key information as an XML string, as in the following example:

```

<RSAKeyValue>
  <Modulus>xCDce...</Modulus>
  <Exponent>AQAB</Exponent>
  <P>6heANKwqXC4d...</P>
  <Q>1nvPzfGay26Q...</Q>
  <DP>Wxyyh3M3vw...</DP>
  <DQ>i0evdC8WtFB...</DQ>
  <InverseQ>r01lw... </InverseQ>
  <D>mHKmjx5hjhZn...</D>
</RSAKeyValue>

```

The XML string represents each of the values from the `RSAParameters` structure as a Base64-encoded string; we have shortened the strings shown, which are very long even for a 384-bit key. The `ToXmlString` method accepts a Boolean argument that includes the private key when set to `true` and exports only the public key when `false`. You can import XML strings exported using this method into instances of `RSACryptoServiceProvider` using the `FromXmlString` method.

15.2.6 Encrypting and Decrypting Data

The `RSACryptoServiceProvider` class defines the `Encrypt` and `Decrypt` methods, each of which processes a byte array, as demonstrated by the following statements:

```

# C#

using System;
using System.Security.Cryptography;
using System.Text;

class RSAExample {

    static void Main( ) {

        // define the plaintext to encrypt
        byte[] x_plaintext
            = Encoding.Default.GetBytes("Programming .NET Security");

        // create an instance of the RSA algorithm
        RSACryptoServiceProvider x_alg = new RSACryptoServiceProvider( );

        // encrypt the plaintext using OAEP padding
        byte[] x_ciphertext = x_alg.Encrypt(x_plaintext, true);

        // decrypt the plaintext

```

```

        byte[] x_new_plaintext = x_alg.Decrypt(x_ciphertext, true);

        // reconstruct a string from the new plaintext and print it out
        string x_output = Encoding.Default.GetString(x_new_plaintext);
        Console.WriteLine(x_output);
    }
}

# Visual Basic .NET

Imports System
Imports System.Security.Cryptography
Imports System.Text

Class RSAExample

    Shared Sub Main( )

        ' define the plaintext to encrypt
        Dim x_plaintext As Byte( ) _
            = Encoding.Default.GetBytes("Programming .NET Security")

        ' create an instance of the RSA algorithm
        Dim x_alg As RSACryptoServiceProvider = New RSACryptoServiceProvider( )

        ' encrypt the plaintext using OAEP padding
        Dim x_ciphertext( ) As Byte = x_alg.Encrypt(x_plaintext, True)

        ' decrypt the plaintext
        Dim x_New_plaintext( ) As Byte = x_alg.Decrypt(x_ciphertext, True)

        ' reconstruct a string from the new plaintext and print it out
        Dim x_output As String = Encoding.Default.GetString(x_New_plaintext)
        Console.WriteLine(x_output)

    End Sub
End Class

```

The Boolean argument to the `Encrypt` and `Decrypt` methods specifies whether to use OAEP padding; if this value is set to `false`, then PKCS #1 v1.5 padding will be used.

The OAEP padding mode is available only on computers running Windows XP or earlier versions upgraded with the "high encryption" pack.

The RSA implementation does not support processing data using the `CryptoStream` class that we described in Chapter 14. The relatively slow performance of all asymmetric algorithms means that they are rarely used to encrypt large amounts of data and are most frequently employed for "key exchange," which we discuss in Chapter 17.

[Team LiB]

[Team LiB]

15.3 Extending the .NET Framework

In this section, we demonstrate how to extend the .NET Framework by adding a new asymmetric encryption algorithm. We have selected the ElGamal algorithm, and we will implement the base classes and the core encryption functions. In the following chapters, we will extend our implementation to include support for digital signatures and padding schemes.

We have provided only a C# implementation of the ElGamal algorithm. Like almost all encryption algorithms, ElGamal relies on mathematical operations that are not possible in Visual Basic .NET without creating additional support functions to compensate for the limited numeric support the language provides.

15.3.1 The ElGamal Algorithm Explained

Taher ElGamal published the ElGamal algorithm in 1985, and it has been widely adopted as an alternative to RSA, where licensing and patents have been an issue (the RSA algorithm was protected by a patent that has now expired, whereas no patents were obtained for ElGamal).

ElGamal relies on a different mathematical problem than the factoring of large numbers used by the RSA algorithm; this problem is finding the discrete logarithm of a large integer value. The ElGamal algorithm is as secure as RSA, but encrypts data slower; the algorithm is the basis for the Digital Signature Algorithm, which we discuss in the following chapter.

15.3.1.1 ElGamal key generation protocol

The key generation protocol for the ElGamal algorithm is as follows; we will select small values to demonstrate the key generation, and then use this key to show how the encryption and decryption functions operate:

1. Create a random prime number, p . This number is the ElGamal "modulus." The number of bits required to represent p is the size of the public key, so that if p is represented using 1024 bits, then the key created by following the protocol is a 1024-bit key. We will select 607 as our example value.
2. Select two other random numbers, g and x , both of which are less than p ; these numbers do not have to be prime. We will select 23 as our value of g , and 152 as our value of x .
3. Compute y , where $y = g^x \bmod p$

For our example, we calculate y as follows:

$$y = 23^{152} \bmod 607 = 209$$

4. The public key is p , g and y . The private key is x .

15.3.1.2 ElGamal encryption and decryption functions

The ElGamal algorithm encrypts only numeric values that are less than the modulus, p . In order to encrypt messages greater than the modulus, we must break the plaintext into smaller values, just as we did with the RSA algorithm.

The ElGamal encryption function is as follows:

1. Break the plaintext into blocks based on the length of the public key modulus.
2. Process each plaintext block, m :
 - a. Choose a random number, k , relatively prime to $p - 1$ (i.e., k and $p - 1$ have no common factors).
 - b. Compute $a = g^k \text{ mod } p$
 - c. Compute $b = (y^k m) \text{ mod } p$
 - d. Concatenate a and b to form an encrypted data block.
3. Concatenate the encrypted data blocks to form the ciphertext.

The size of the ciphertext created by the encryption function is double the size of the plaintext; this is not an issue for most projects because asymmetric algorithms are typically used to encrypt only small amounts of data; see Chapter 17 for more details.

The ElGamal decryption function is as follows:

1. Break the ciphertext into blocks that are twice the length of the key modulus.
2. Process each ciphertext block:
 - a. Break the ciphertext block in half to form a and b .
 - b. Use the private key to compute the decrypted block m where
 - c. Concatenate the decrypted blocks to form the plaintext.

These functions are the "raw" algorithm; like RSA, ElGamal is always used in conjunction with a padding scheme, usually OAEP. In this chapter, we implement the raw algorithm, setting the padding aside until Chapter 17.

15.3.2 Processing Large Integer Values

One problem that asymmetric algorithms present is the need to perform arithmetic operations on very large integer values, often containing many hundreds of digits. The .NET Framework cannot process numbers that require more than 64 bits to represent, falling short of our requirements, where 1024-

and 2048-bit numbers are commonly required.

Our ElGamal implementation relies on the `BigInteger` class written and placed in the public domain^[1] by Chew Keong Tan. The `BigInteger` class represents large integer values by using arrays of 32-bit integers and includes useful methods for many mathematical functions, including the generation of probable prime numbers.

[1] The following statements comprise the copyright declaration and the license for Chew Keong Tan's `BigInteger` class :

We have included the `BigInteger` class in the ElGamal example code, and with the exception of one small change, we use the same version that you can download from the internet. The change we have made is to double the largest value that the class can represent by increasing the `maxLength` variable to 140. You do not have to be concerned with the details of this change or the implementation of the class. Table 15-6 describes the class members that we use for our algorithm implementation.

Table 15-6. Selected members of the `BigInteger` class

Member	Description
Constructors	
<code>BigInteger(byte[])</code>	Creates a new instance from an array of signed byte values
<code>BigInteger(long)</code> <code>BigInteger(int)</code>	Creates a new instance representing the numeric value specified by the <i>long</i> or <i>int</i> argument
<code>BigInteger(BigInteger)</code>	Creates a new instance with the same value as the argument
Methods	
<code>modPow(BigInteger exp, BigInteger m)</code>	Returns the value $(\text{this}^{\text{exp}} \bmod m)$
<code>modInverse(BigInteger m)</code>	Returns the value $(\text{this}^{-1} \bmod m)$
<code>getBytes()</code>	Converts the integer into a series of signed byte values
<code>bitCount()</code>	Calculates how many bits are required to express the integer value
<code>genPseudoPrime(int bits, int confidence, Random rand)</code>	Generates a random positive <code>BigInteger</code> with the specified number of bits and confidence level, using the provided random number generator
<code>genRandomBits(int bits, Random rand)</code>	Creates a random number with the specified number of bits, using the provided random number generator

15.3.3 Defining the Abstract Class

We start by defining the `ElGamalParameters` structure, which we use to import and export keys to and

from the implementation classes without exposing our dependence on the `BigInteger` class; this is equivalent to the `RSAParameters` structure used by the RSA implementation classes:

```
[Serializable]
public struct ElGamalParameters {
    public byte[] P;
    public byte[] G;
    public byte[] Y;
    [NonSerialized] public byte[] X;
}
```

We have annotated the structure with the `Serializable` attribute so that users of the `ElGamal` implementation can store public keys persistently, but we have used the `NonSerialized` attribute to ensure that the private key cannot be persisted. We express the value of `P`, `G`, `Y` and `X` as byte arrays for consistency with the rest of the .NET Framework, even though we will represent these values internally using the `BigInteger` class:

```
public abstract class ElGamal : AsymmetricAlgorithm {

    public abstract void ImportParameters(ElGamalParameters p_parameters);
    public abstract ElGamalParameters ExportParameters(bool
        p_include_private_params);
}
```

We have called our abstract class `ElGamal`, in keeping with the standard naming model. The `ImportParameters` and `ExportParameters` methods set and get the state of the algorithm using instances of the `ElGamalParameters` structure we defined previously:

```
public abstract byte[] EncryptData(byte[] p_data);
public abstract byte[] DecryptData(byte[] p_data);
public abstract byte[] Sign(byte[] p_hashcode);
public abstract bool VerifySignature(byte[] p_hashcode, byte[] p_signature);
```

The remaining abstract methods define the asymmetric encryption and digital signature functions we support; for simplicity we will not support padding schemes in the algorithm class, which allows us to define an abstract class that can be used directly (unlike the RSA implementation, where the methods defined by the abstract class are not implemented by the Crypto API implementation).

The only methods that are not abstract are `ToXmlString` and `FromXmlString`, required by the `AsymmetricAlgorithm` class. We can implement these in the abstract class safely because the implementation gets and sets the details about the key using the abstract `ImportParameters` and `ExportParameters` methods; the implementation details do not affect the way in which the XML string is processed:

```
public override string ToXmlString(bool p_include_private) {
    ElGamalParameters x_params = ExportParameters(p_include_private);
    // create a new string builder
    StringBuilder x_sb = new StringBuilder( );
    // add the header
    x_sb.Append("<ElGamalKeyValue>");
    // add the public elements from the parameters
    x_sb.Append("<P>" + Convert.ToBase64String(x_params.P) + "</P>");
    x_sb.Append("<G>" + Convert.ToBase64String(x_params.G) + "</G>");
    x_sb.Append("<Y>" + Convert.ToBase64String(x_params.Y) + "</Y>");
}
```



```

    if (p_include_private) {
        // we need to include X, which is the part of private key
        x_sb.Append("<X>" + Convert.ToBase64String(x_params.X) + "</X>");
    }
    // add the final element
    x_sb.Append("</ElGamalKeyValue>");
    return x_sb.ToString( );
}

public override void FromXmlString(String p_string) {
    // create the params that we will use as the result
    ElGamalParameters x_params = new ElGamalParameters( );
    // create a text reader using a string reader
    XmlTextReader x_reader =
        new XmlTextReader(new System.IO.StringReader(p_string));

    // run through the elements in the xml string
    while (x_reader.Read( )) {
        // we are only interested in processing start nodes
        if (true || x_reader.IsStartElement( )) {
            switch (x_reader.Name) {
                case "P":
                    // set the value for P
                    x_params.P =
                        Convert.FromBase64String(x_reader.ReadString( ));
                    break;
                case "G":
                    // set the value for G
                    x_params.G =
                        Convert.FromBase64String(x_reader.ReadString( ));
                    break;
                case "Y":
                    // set the value for Y
                    x_params.Y =
                        Convert.FromBase64String(x_reader.ReadString( ));
                    break;
                case "X":
                    // set the value for X (this would not be found in a
                    // string that was generated by excluding the private
                    // elements.
                    x_params.X =
                        Convert.FromBase64String(x_reader.ReadString( ));
                    break;
            }
        }
    }
    // Import the result
    ImportParameters(x_params);
}
}

```

15.3.4 Defining the Implementation Class

Begin the implementation by defining a structure that you can use privately to represent the key as a series of `BigInteger`s. The `ElGamalKeyStruct` structure makes processing blocks of data simpler, because you do not need to convert the byte arrays represented by the `ElGamalParameters` to integer values each time you need to use one of the key parameters:

```
using System;
using System.Security.Cryptography;

public struct ElGamalKeyStruct {
    public BigInteger P;
    public BigInteger G;
    public BigInteger Y;
    public BigInteger X;
}
```

The name of our implementation class is `ElGamalManaged`, in keeping with the .NET cryptographic naming model. This class extends the `ElGamal` abstract algorithm class, and defines an `ElGamalKeyStruct` instance variable that you use to represent the current key in use:

```
public class ElGamalManaged : ElGamal {
    private ElGamalKeyStruct o_key_struct;
```

The first part of the constructor initializes the `BigInteger`s contained in the `ElGamalKeyStruct` to 0; this gives a known starting point, and you can assume the user has not imported a key if the public key parameter values are all set to 0:

```
    public ElGamalManaged( ) {
        // create the key struct
        o_key_struct = new ElGamalKeyStruct( );
        // set all of the big integers to zero
        o_key_struct.P = new BigInteger(0);
        o_key_struct.G = new BigInteger(0);
        o_key_struct.Y = new BigInteger(0);
        o_key_struct.X = new BigInteger(0);
```

The other task the constructor performs defines the range of key lengths that the implementation supports. We have selected 384 bits as the shortest key supported to conform to the .NET RSA implementation. We define 1088 bits as the largest value because of the size limitations when using the `BigInteger` class; even though we have doubled the largest value that this class can represent, we still exceed that limit when performing the encryption computation for larger key sizes. We set the default key size to be 1024 bits, which provides a key strength that is suitable for most projects:

```
        // set the default key size value
        KeySizeValue = 1024;
        // set the range of legal keys
        LegalKeySizesValue = new KeySizes[] {new KeySizes(384, 1088, 8)};
    }
```

The `SignatureAlgorithm` and `KeyExchangeAlgorithm` properties return a string value that indicates how the class performs encryption and creates digital signatures; we are implementing the "raw"

ElGamal algorithm, and so both of these properties simply return `ElGamal` :

```
public override string SignatureAlgorithm {
    get {
        return "ElGamal";
    }
}

public override string KeyExchangeAlgorithm {
    get {
        return "ElGamal";
    }
}
```

The `CreateKeyPair` method follows the ElGamal key generation protocol to create a new key pair; the key parameter values are set using the `ElGamalKeyStruct` instance variable; the method accepts an integer argument that specifies the required key length:

```
private void CreateKeyPair(int p_key_strength) {
    // create the random number generator
    Random x_random_generator = new Random( );

    // create the large prime number, P
    o_key_struct.P = BigInteger.genPseudoPrime(p_key_strength,
        16, x_random_generator);

    // create the two random numbers, which are smaller than P
    o_key_struct.X = new BigInteger( );
    o_key_struct.X.genRandomBits(p_key_strength -1, x_random_generator);
    o_key_struct.G = new BigInteger( );
    o_key_struct.G.genRandomBits(p_key_strength -1, x_random_generator);

    // compute Y
    o_key_struct.Y = o_key_struct.G.modPow(o_key_struct.X, o_key_struct.P);
}
}
```

The `NeedToGenerateKey` method tests the value of the public key parameters; if all of the values are 0, then we assume that the user has not imported a key and that we should create a new key pair before encrypting or decrypting data. The `KeyStruct` property simply gets or sets the `ElGamalKeyStruct` instance member:

```
private bool NeedToGenerateKey( ) {
    return o_key_struct.P == 0 && o_key_struct.G == 0 && o_key_struct.Y == 0;
}

public ElGamalKeyStruct KeyStruct {
    get {
        if (NeedToGenerateKey( )) {
            CreateKeyPair(KeySizeValue);
        }
        return o_key_struct;
    }
}
```

```

    }
    set {
        o_key_struct = value;
    }
}

```

The `ImportParameters` method accepts an instance of the `ElGamalParameters` structure and creates the `BigIntegers` that we will use to represent the key; this method allows users to import a previously created key and is used by the `FromXmlString` method in the abstract `ElGamal` class:

```

public override void ImportParameters(ElGamalParameters p_parameters) {
    // obtain the big integer values from the byte parameter values
    o_key_struct.P = new BigInteger(p_parameters.P);
    o_key_struct.G = new BigInteger(p_parameters.G);
    o_key_struct.Y = new BigInteger(p_parameters.Y);
    if (p_parameters.X != null && p_parameters.X.Length > 0) {
        o_key_struct.X = new BigInteger(p_parameters.X);
    }
    // set the length of the key based on the import
    KeySizeValue = o_key_struct.P.bitCount( );
}

```

The `ExportParameters` method creates an `ElGamalParameters` structure using the key represented by the instance `ElGamalKeyStruct` member. Notice that we check to see if we need to create a new key pair before we export the key details to ensure that there is something to export. The Boolean argument determines whether to export the private parameters, in keeping with the model used by the RSA implementation classes:

```

public override ElGamalParameters ExportParameters(bool
    p_include_private_params) {

    if (NeedToGenerateKey( )) {
        // we need to create a new key before we can export
        CreateKeyPair(KeySizeValue);
    }

    // create the parameter set
    ElGamalParameters x_params = new ElGamalParameters( );
    // set the public values of the parameters
    x_params.P = o_key_struct.P.getBytes( );
    x_params.G = o_key_struct.G.getBytes( );
    x_params.Y = o_key_struct.Y.getBytes( );

    // if required, include the private value, X
    if (p_include_private_params) {
        x_params.X = o_key_struct.X.getBytes( );
    } else {
        // ensure that we zero the value
        x_params.X = new byte[1];
    }
    // return the parameter set
    return x_params;
}

```

```
}
```

The `EncryptData` and `DecryptData` methods provide access to the ElGamal encryption and decryption functions; create a new key pair if the user has not imported a key, and then pass the data to the specialized `ElGamalEncryptor` and `ElGamalDecryptor` classes, which we detail in the next sections:

```
public override byte[] EncryptData(byte[] p_data) {
    if (NeedToGenerateKey( )) {
        // we need to create a new key before we can export
        CreateKeyPair(KeySizeValue);
    }
    // encrypt the data
    ElGamalEncryptor x_enc = new ElGamalEncryptor(o_key_struct);
    return x_enc.ProcessData(p_data);
}

public override byte[] DecryptData(byte[] p_data) {
    if (NeedToGenerateKey( )) {
        // we need to create a new key before we can export
        CreateKeyPair(KeySizeValue);
    }
    // encrypt the data
    ElGamalDecryptor x_enc = new ElGamalDecryptor(o_key_struct);
    return x_enc.ProcessData(p_data);
}
```

The `Dispose` method is useful for releasing unmanaged resources, but you do not need to do anything in this method since the implementation is written entirely in managed code, but it is a requirement of extending from the `AsymmetricAlgorithm` class:

```
protected override void Dispose(bool p_bool) {
    // do nothing - no unmanaged resources to release
}
```

The `Sign` and `VerifySignature` methods support the creation and verification of digital signatures; although you have implemented these methods to comply with the abstractions of the `ElGamal` class, we will not explain their use, or detail the classes they call until Chapter 16:

```
public override byte[] Sign(byte[] p_hashcode) {
    throw new System.NotImplementedException( );
}

public override bool VerifySignature(byte[] p_hashcode, byte[] p_signature) {
    throw new System.NotImplementedException( );
}
```

15.3.5 Defining the Abstract Cipher Function Class

We use the `ElGamalAbstractCipher` class as the basis for the encryption and decryption classes; it contains methods that are common to both functions. This class is also responsible for breaking up an array of data into blocks that can be processed using the current public key (remember that the length of the key modulus affects the amount of data that can be processed by the encryption function in one

go):

```
using System;
using System.IO;
```

```
public abstract class ElGamalAbstractCipher {
    protected int o_block_size;
    protected int o_plaintext_blocksize;
    protected int o_ciphertext_blocksize;
    protected ElGamalKeyStruct o_key_struct;

    public ElGamalAbstractCipher(ElGamalKeyStruct p_key_struct) {
        // set the key details
        o_key_struct = p_key_struct;

        // calculate the block sizes
        o_plaintext_blocksize = (p_key_struct.P.bitCount( ) - 1) / 8;
        o_ciphertext_blocksize = ((p_key_struct.P.bitCount( ) + 7) / 8) * 2;

        // set the default block for plaintext, which is suitable for encryption
        o_block_size = o_plaintext_blocksize;
    }
}
```

The constructor accepts an `ElGamalKeyStruct` structure as an argument, which is used to work out the size of the blocks that will be used by the encryption and decryption functions-the ElGamal encryption function produces ciphertext that is twice the size of the corresponding plaintext:

```
public byte[] ProcessData(byte[] p_data) {

    // create a stream backed by a memory array
    MemoryStream x_stream = new MemoryStream( );
    // determine how many complete blocks there are
    int x_complete_blocks = p_data.Length / o_block_size;

    // create an array which will hold a block
    byte[] x_block = new Byte[o_block_size];

    // run through and process the complete blocks
    int i = 0;
    for (; i < x_complete_blocks; i++) {
        // copy the block and create the big integer
        Array.Copy(p_data, i * o_block_size, x_block, 0, o_block_size);
        // process the block
        byte[] x_result = ProcessDataBlock(x_block);
        // write the processed data into the stream
        x_stream.Write(x_result, 0, x_result.Length);
    }

    // process the final block
    byte[] x_final_block = new Byte[p_data.Length -
        (x_complete_blocks * o_block_size)];
    Array.Copy(p_data, i * o_block_size, x_final_block, 0,
```

```

        x_final_block.Length);

    // process the final block
    byte[] x_final_result = ProcessFinalDataBlock(x_final_block);

    // write the final data bytes into the stream
    x_stream.Write(x_final_result, 0, x_final_result.Length);

    // return the contents of the stream as a byte array
    return x_stream.ToArray( );
}

```

The `ProcessData` method uses the block sizes calculated in the constructor to process an array of data; the array is broken down into data blocks and passed to the `ProcessDataBlock` and `ProcessFinalDataBlock` using the model laid down by the `ICryptoTransform` interface you saw in Chapter 14 .

The `ProcessDataBlock` and `ProcessFinalDataBlock` methods are abstract, and the encryption and decryption subclasses must implement them in order to process the blocks of data created by the `ProcessData` method:

```

    protected abstract byte[] ProcessDataBlock(byte[] p_block);

    protected abstract byte[] ProcessFinalDataBlock(byte[] p_final_block);
}

```

15.3.6 Defining the Encryption Class

The `ElGamalEncryptor` class extends `ElGamalAbstractCipher` and is responsible for producing ciphertext using the ElGamal encryption cipher; the `EncryptData` method in the `ElGamalManaged` class uses this class:

```

using System;

public class ElGamalEncryptor : ElGamalAbstractCipher {
    Random o_random;

    public ElGamalEncryptor(ElGamalKeyStruct p_struct) : base(p_struct) {
        o_random = new Random( );
    }
}

```

The constructor creates a new random number generator, which the `BigInteger` class uses to create the large random value K , required by the encryption function. The `ProcessDataBlock` method represents the encryption function, and computes the A and B values that we discussed in the introduction to the ElGamal algorithm; these values are concatenated to form a single ciphertext block that is the method return value:

```

    protected override byte[] ProcessDataBlock(byte[] p_block) {
        // the random number, K
        BigInteger K;
        // create K, which is the random number
        do {

```

```

        K = new BigInteger( );
        K.genRandomBits(o_key_struct.P.bitCount( ) -1, o_random);
    } while (K.gcd(o_key_struct.P -1) != 1);

    // compute the values A and B
    BigInteger A = o_key_struct.G.modPow(K, o_key_struct.P);
    BigInteger B = (o_key_struct.Y.modPow(K, o_key_struct.P)
        * new BigInteger(p_block)) % (o_key_struct.P);

    // create an array to contain the ciphertext
    byte[] x_result = new byte[o_ciphertext_blocksize];
    // copy the bytes from A and B into the result array
    byte[] x_a_bytes = A.getBytes( );
    Array.Copy(x_a_bytes, 0, x_result, o_ciphertext_blocksize / 2
        - x_a_bytes.Length, x_a_bytes.Length);
    byte[] x_b_bytes = B.getBytes( );
    Array.Copy(x_b_bytes, 0, x_result, o_ciphertext_blocksize
        - x_b_bytes.Length, x_b_bytes.Length);
    // return the result array
    return x_result;
}

```

The `ElGamalAbstractCipher` class calls the `ProcessFinalDataBlock` method to encrypt the last block of plaintext, which may not be a complete block; the block is padded with 0 as required, and then passed to the `ProcessDataBlock` method for encryption. If the data block contains no data, then assume that the plaintext is divided into blocks with no partial data remaining, and therefore there is no remaining data for encryption. In this case, simply return an empty byte array:

```

protected override byte[] ProcessFinalDataBlock(byte[] p_final_block) {
    if (p_final_block.Length > 0) {
        if (p_final_block.Length < o_block_size) {
            // create a fullsize block which contains the
            // data to encrypt followed by trailing zeros
            byte[] x_padded = new byte[o_block_size];
            Array.Copy(p_final_block, 0, x_padded, 0, p_final_block.Length);
            return ProcessDataBlock(x_padded);
        } else {
            return ProcessDataBlock(p_final_block);
        }
    } else {
        return new byte[0];
    }
}
}

```

15.3.7 Defining the Decryption Class

The `ElGamalDecryptor` class extends `ElGamalAbstractCipher` and is responsible for restoring plaintext using the ElGamal decryption cipher; the `DecryptData` method in the `ElGamalManaged` class uses this class:


```

using System;

public class ElGamalDecryptor : ElGamalAbstractCipher {

    public ElGamalDecryptor(ElGamalKeyStruct p_struct) : base(p_struct) {
        // set the default block size to be ciphertext
        o_block_size = o_ciphertext_blocksize;
    }
}

```

The `ProcessDataBlock` method represents the decryption function, and breaks the data block in half to obtain the `a` and `b` values that we discussed in the introduction to the ElGamal algorithm; these values are used to compute the plaintext block that is the method return value:

```

protected override byte[] ProcessDataBlock(byte[] p_block) {
    // extract the byte arrays that represent A and B
    byte[] x_a_bytes = new byte[o_ciphertext_blocksize / 2];
    Array.Copy(p_block, 0, x_a_bytes, 0, x_a_bytes.Length);
    byte[] x_b_bytes = new byte[o_ciphertext_blocksize / 2];
    Array.Copy(p_block, x_a_bytes.Length, x_b_bytes, 0, x_b_bytes.Length);
    // create big integers from the byte arrays
    BigInteger A = new BigInteger(x_a_bytes);
    BigInteger B = new BigInteger(x_b_bytes);
    // calculate the value M
    BigInteger M = (B *
        A.modPow(o_key_struct.X, o_key_struct.P).modInverse(o_key_struct.P))
        % o_key_struct.P;
    // return the result - take care to ensure that we create
    // a result which is the correct length
    byte[] x_m_bytes = M.getBytes( );
    // we may end up with results which are short some leading
    // bytes - add these are required
    if (x_m_bytes.Length < o_plaintext_blocksize) {
        byte[] x_full_result = new byte[o_plaintext_blocksize];
        Array.Copy(x_m_bytes, 0, x_full_result,
            o_plaintext_blocksize - x_m_bytes.Length, x_m_bytes.Length);
        x_m_bytes = x_full_result;
    }
    return x_m_bytes;
}
}

```

The decryption version of the `ProcessFinalDataBlock` method can assume that there is no partial data to process, because the encryption version will add padding as required:

```

protected override byte[] ProcessFinalDataBlock(byte[] p_final_block) {
    if (p_final_block.Length > 0) {
        return ProcessDataBlock(p_final_block);
    } else {
        return new byte[0];
    }
}
}

```

15.3.8 Testing the Algorithm

The following statements use the smallest supported key size (384 bits) to encrypt and decrypt the string "Programming .NET Security". Export the key parameters from one instance of the `ElGamalManaged` class and import them into other instances to perform the encryption and decryption:

```
using System;
using System.Text;
using System.Security.Cryptography;

public class Test {

    public static void Main( ) {
        // define the byte array that we will use
        // as plaintext
        byte[] x_plaintext
            = Encoding.Default.GetBytes("Programming .NET Security");

        // Create an instance of the algorithm and generate some keys
        ElGamal x_alg = new ElGamalManaged( );
        // set the key size - keep is small to speed up the tests
        x_alg.KeySize = 384;
        // extract and print the xml string (this will cause
        // a new key pair to be generated)
        string x_xml_string = x_alg.ToXmlString(true);
        Console.WriteLine("\n{0}\n", x_xml_string);

        // Test the basic encryption support
        ElGamal x_encrypt_alg = new ElGamalManaged( );
        // set the keys - note that we export without the
        // private parameters since we are encrypting data
        x_encrypt_alg.FromXmlString(x_alg.ToXmlString(false));
        byte[] x_ciphertext = x_encrypt_alg.EncryptData(x_plaintext);

        // create a new instance of the algorithm to decrypt
        ElGamal x_decrypt_alg = new ElGamalManaged( );
        // set the keys - note that we export with the
        // private parameters since we are decrypting data
        x_decrypt_alg.FromXmlString(x_alg.ToXmlString(true));
        // restore the plaintext
        byte[] x_candidate_plaintext = x_decrypt_alg.DecryptData(x_ciphertext);

        Console.WriteLine("BASIC ENCRYPTION: {0}",
            CompareArrays(x_plaintext, x_candidate_plaintext));
    }

    private static bool CompareArrays(byte[] p_arr1, byte[] p_arr2) {
        for (int i = 0; i < p_arr1.Length; i++) {
            if (p_arr1[i] != p_arr2[i]) {
                return false;
            }
        }
    }
}
```

```
        }  
        return true;  
    }  
}
```

[Team LiB]

[\[Team LiB \]](#)

Chapter 16. Digital Signatures

Digital signatures provide a solution to the problem of message authentication, and let Bob accurately identify the author of messages he receives. For example, with digital signatures, Bob can differentiate between messages sent by Alice and messages forged by Eve. In this chapter, we explain what digital signatures are, how they work, and how you use them. We also extend the .NET Framework by adding support for digital signatures to the implementation of the ElGamal algorithm that we introduced in [Chapter 15](#).

[\[Team LiB \]](#)

[[Team LiB](#)]

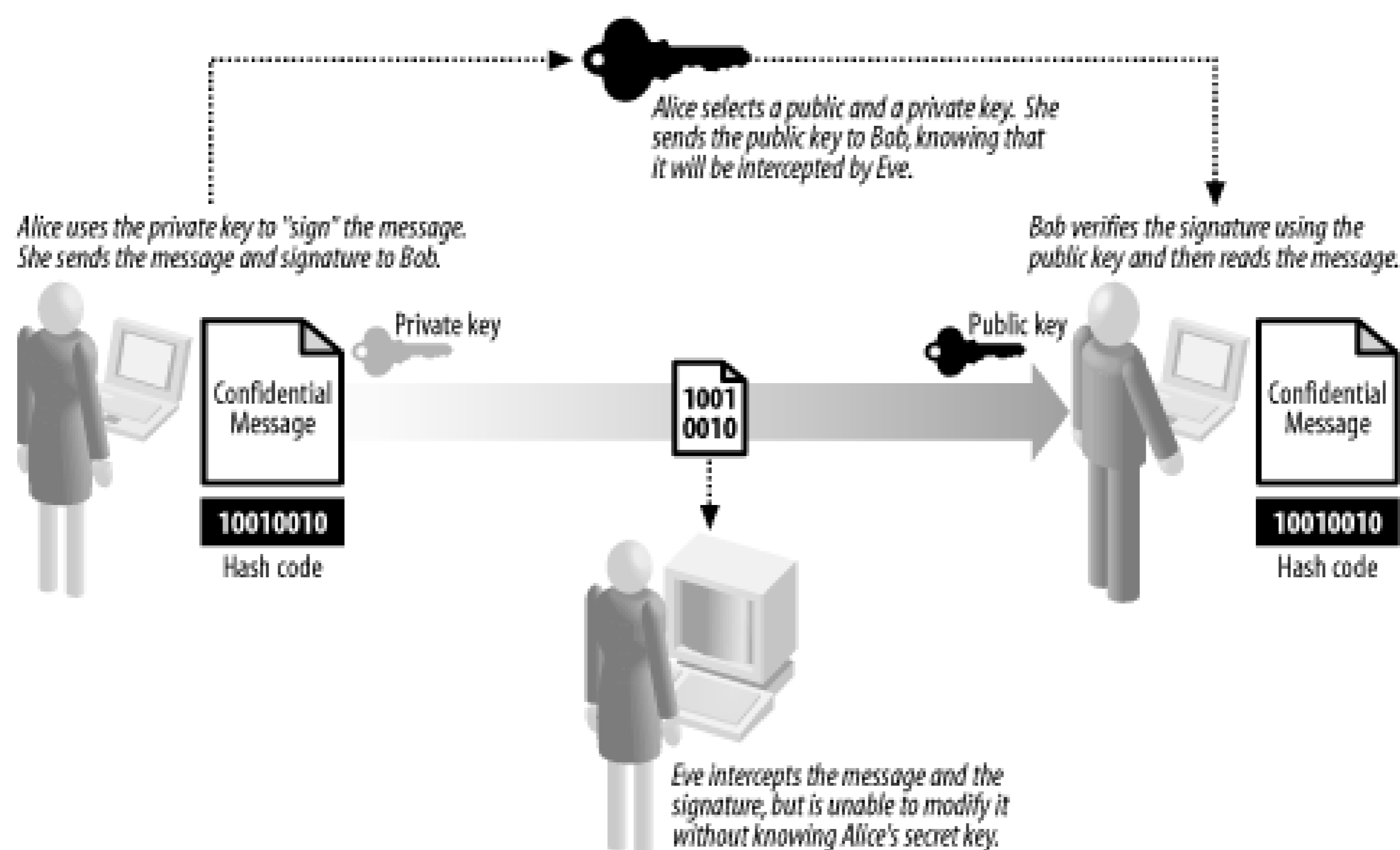
16.1 Digital Signatures Explained

Digital signatures are a different application of the asymmetric algorithms that we discussed in [Chapter 15](#). You use an asymmetric key pair to create a "signature" for a message by adding a *signature* function to the asymmetric algorithm. The recipient can verify the signature using a *verification* function. Asymmetric algorithms that support signature and verification functions are *digital signature algorithms*. Each digital signature is specific to an individual message or document, and if Alice signs a message, Bob can be confident that Eve has not sent him a forgery.

[Figure 16-1](#) provides an overview of using digital signatures, which work as follows:

1. Alice selects an asymmetric signature algorithm and follows the key generation protocol to create a new pair of keys. She keeps her private key secret and sends the public key to Bob.
2. Alice composes a message and creates a signature using her private key.
3. Alice sends the message and the signature to Bob.
4. Bob uses Alice's public key to verify the signature. If the signature is valid, Bob can be confident that:
 - a. Alice is truly the author of the message.
 - b. Eve has not altered the message contents.

Figure 16-1. Using asymmetric signature algorithms to provide message authentication



Notice that Alice, the sender of the message, creates the key pair and retains the private key; this is different from asymmetric encryption, where the recipient (Bob) is responsible for key creation. Digitally signing a message creates a separate piece of data, which Alice sends to Bob along with the message.

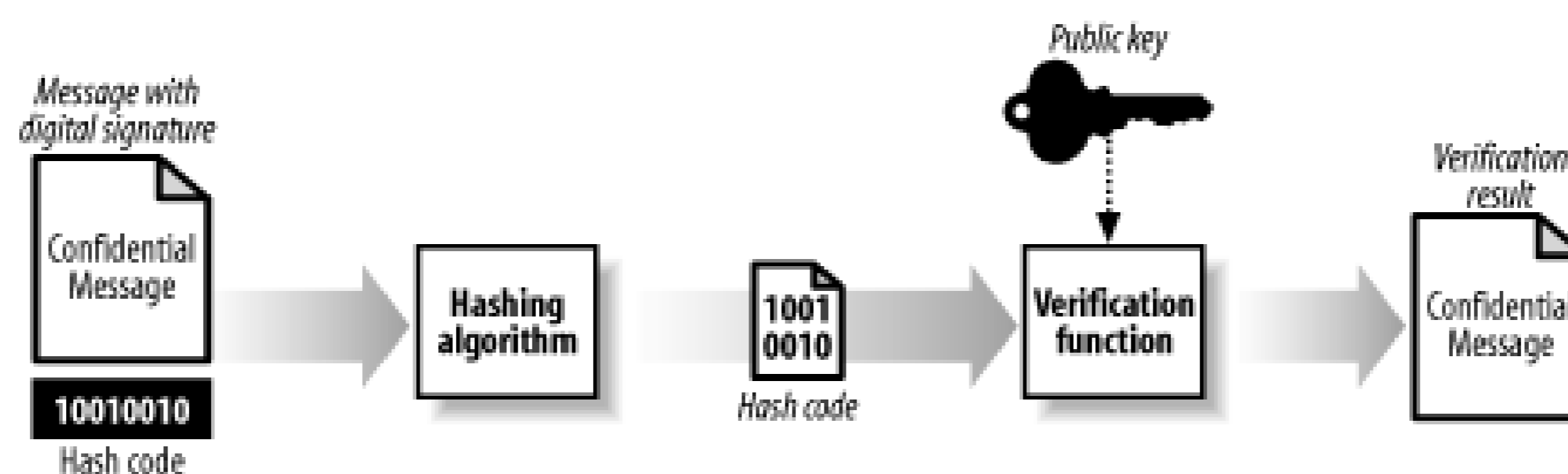
The relatively slow performance of asymmetric algorithms means that Alice does not sign the entire message she wants to send; instead, she creates a cryptographic hash code for the message and signs this instead. Subject to the limitations of hash code security (discussed in [Chapter 13](#)), signing hash code for a message is equivalent to signing the message itself, but is much faster, because the hash code is less data to process with the asymmetric algorithm. [Figure 16-2](#) illustrates the way that Alice creates the signature by using her private key, the data hash code, and the asymmetric signature function.

Figure 16-2. Using a hash code as the basis for creating a digital signature

Bob verifies the signature by creating his own hash code and using the asymmetric signature verification and Alice's public key. Bob cannot verify the signature if the message is altered in any way, meaning that Eve cannot secretly tamper with the contents and Bob cannot alter a message and later claim that Alice had signed the altered version. Alice and Bob must use the same hash code algorithm; otherwise, Bob will not be able to verify valid signatures. [Figure 16-3](#) illustrates how Bob

verifies a signature.

Figure 16-3. Using a hash code as the basis for verifying a digital signature



Digital signatures allow more than one person to sign a message or document, such as a contract. Each person signs the message using the basic protocol we described, and sends their signature to a of the other signatories. If Alice and Bob wish to sign a digital contract, the basic protocol that they use is as follows:

1. Alice and Bob agree on the contents of the contract they will sign.
2. Alice creates a hash code for the contract and signs it with her private key.
3. Bob creates a hash code for the contract and signs it with his private key.
4. Alice and Bob exchange digital signatures and public keys.
5. Alice checks that Bob has used the correct hash code (and therefore has signed the expected version of the contract) and verifies the signature using his public key.
6. Bob checks that Alice has used the correct hash code and verifies the signature using her public key.

Should there be any dispute, a third party (perhaps a judge or other arbitrator) can use Alice and Bob's public keys to verify that both Alice and Bob have signed the same contract. Neither party can alter the contract (because then the hash code would not match) nor deny that they signed the contract (because the third party can verify each separate signature).

16.1.1 Digital Signature Security

Digital signatures have two objectives: they must be cryptographically secure and they must provide equivalent security features to an ink signature on a printed document. We summarize the security features of digital signatures as follows:

- Eve should not be able to forge Alice's digital signature. Eve cannot create a fake signature unless she knows Alice's private key; as in almost all aspects of cryptography, digital signatures rely on the careful protection and management of keys. If Eve could forge Alice's signature, Eve could make it appear that Alice has signed any document.

- Eve should not be able to reuse one of Alice's digital signatures. Eve can only take Alice's signature from one document and associate it with another if both documents produce the same data hash code. See [Chapter 13](#) for details of why this is not a simple thing to achieve.
- Eve should not be able to alter a signed document. If Eve alters the content of a document, then Alice's signature will no longer be valid, because the altered document will produce a different data hash code to the one that Alice signed. Eve could alter a document successfully if it resulted in the same data hash code as the original (which is, in effect, the same as reusing Alice's signature).
- Alice should not be able to deny that she has signed a document. Alice cannot sign a document and then later claim that she did not, because we assume that she is the only person who knows her secret key. If she suspects that her key has been compromised, then she must create a new key pair.

From these features, you can see that digital signatures provide a reasonable analogue to a physical signature. You can also see that the entire security of digital signatures is dependent on the following assumptions:

- Eve is unable to guess or otherwise obtain Alice's secret key.
- Eve is unable to find a document that results in a specific data hash value.

We know that these assumptions are optimistic. Eve can find Alice's secret key by testing every possible value and find a document that results in a specific hash code if she creates and tests enough variations. In [Chapter 13](#) and [Chapter 14](#), we found that Eve *can* do these things, but that they can take her a very long time to accomplish. The security of digital signatures, like every other aspect of cryptography, is based on probabilities and effective key management. When using digital signatures, you should be aware that they are susceptible to the same attacks as data hash codes and asymmetric keys.

The final point raises a very serious issue. Alice should not be able to sign a document and then later claim that she did not. Relying on digital signatures means making another assumption: Alice is responsible for all signatures created with her private key, even if Eve is able to guess the secret value.

The problem that this presents is that Eve may be able to guess Alice's private key by brute force without Alice being aware that her private key is compromised. Eve could then sign documents using Alice's signature, but Alice would not find out that this was happening until it was too late. For example, if Eve signs a digital money order to herself, Alice will not know that money is missing from her account until she receives her next bank statement.

We need Alice to be responsible for signatures created with her key; otherwise, digital signatures would be worthless, and Alice could simply deny signing documents that are not in her favor. On the other hand, because it is possible to guess the value of Alice's key, we run the risk of holding her responsible for signatures that she really did not create. There is currently no solution to this problem, which is a fundamental barrier to the adoption of digital signatures.

16.1.2 The .NET Framework Digital Signature Algorithms

The .NET Framework includes two asymmetric algorithms that support digital signatures, one is the RSA algorithm that we discussed in [Chapter 15](#); this algorithm defines functions for both data encryption and digital signatures.

The second algorithm is the Digital Signature Algorithm (DSA), published as a U.S. FIPS in May 1994. DSA supports digital signatures but not data encryption. Initially created to provide a standard algorithm for digital signatures for federal projects, DSA has gained support in commercial projects gradually but still lags behind RSA in terms of adoption.

The creators of DSA based their design, in part, on the ElGamal algorithm that we introduced in [Chapter 15](#). In [Section 16.4](#) of this chapter, we add support for signatures and signature formatting to our ElGamal implementation.

All asymmetric algorithms include a key generation protocol, similar to the ones that we discussed in [Chapter 15](#). If an algorithm supports both encryption and digital signatures, then the same keys can be used by the encryption, decryption, signature, and verification functions; following the key generation protocol results in keys that can be used for all asymmetric cryptographic tasks.

16.1.3 Creating Digital Signatures

The hash code created from the document to be signed is usually formatted before being processed by the algorithm's signature function; this ensures that the specified key pair pads the hash code to the maximum size that can be signed. See [Chapter 15](#) for more information about how to determine the length of a key and how the length affects the amount of data that algorithm functions can process. The act of formatting a hash code before it is signed also protects the digital signature from certain types of cryptographic attack.

The most commonly used signature-formatting protocol is PKCS #1, which we summarize as follows:

1. Create **H**, the cryptographic hash code for the document. SHA-1 and MD5 are the most commonly used hashing algorithms; see [Chapter 13](#) for details of these hashing algorithms and the .NET Framework support for them.
2. Concatenate the hashing algorithm ID and the hexadecimal representation of the hash code together to form the **DigestInfo** value. Each hashing algorithm has a unique identifier. [Table 16-1](#) lists the hexadecimal IDs for each of the algorithms that the .NET Framework supports.
3. Create an array of padding bytes, **PS**; the value of each byte is 255. The number of bytes to create depends on the length of the key and the length of **DigestInfo**. The number of bytes is $\frac{\text{key length} - \text{DigestInfo length}}{255}$, where the lengths **DigestInfo** and the key are expressed in bytes. The length of **PS** will be at least 8 bytes.
4. Combine the following to form the formatted data:
 - a. One byte with a value of 0
 - b. One byte with a value of 1
 - c. The padding string **PS**

d. `DigestInfo`, represented as bytes



The acronym PKCS stands for Public Key Cryptography Standard, which is a series of documents published by RSA Security, whose founders developed the RSA Algorithm. The PKCS #1 formatting for digital signatures is not the same as the PKCS #1 padding that we described in [Chapter 15](#).

Each PKCS standard addresses a specific aspect of asymmetric cryptography, and PKCS #1 describes how to use the RSA algorithm for encryption and digital signatures. The PKCS #1 data padding and signature formatting techniques we describe are both contained in the PKCS #1 document.

Table 16-1. Hexadecimal IDs for hashing algorithms

Algorithm	Hexadecimal ID
MD5	30 20 30 0c 06 08 2a 86 48 86 f7 0d 02 05 05 00 04 10
SHA-1	30 21 30 09 06 05 2b 0e 03 02 1a 05 00 04 14
SHA-256	30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20
SHA-384	30 41 30 0d 06 09 60 86 48 01 65 03 04 02 02 05 00 04 30
SHA-512	30 51 30 0d 06 09 60 86 48 01 65 03 04 02 03 05 00 04 40

Once you have created the formatted data, sign it using the signature function. In this section, we continue our illustrations using the RSA protocol. The RSA signature function is shown below, where s is the signature value and m is the formatted data to sign:

$$s = m^d \bmod n$$

The values d and n are parameters of the RSA key; see the previous chapter for details about RSA key pairs and how to create them. The RSA key length required to create a digital signature depends on the size of the hash code that the selected hashing algorithm produces. For example, MD5 produces a 128-bit hash code, and with the 18 bytes of the algorithm ID and the 11 bytes of data that PKCS #1 formatting adds as a minimum, you need a key that is able to process 360-bits of data. Therefore, the smallest key length that you can use to sign a PKCS #1-formatted MD5 hash code is 368 bits.

We won't walk through the computation required to create a digital signature because the numeric values that make up a 368-bit key are simply too large to print on a page; see [Chapter 15](#) for an example of using the RSA functions with small key lengths.

To verify a signature, take the document you have received and create a PKCS #1-formatted hash code, following the protocol we described previously. Use the RSA verification function, shown below, where m_2 is the verification result, s is the digital signature, and e and n are the RSA public key parameters:

$$m_2 = s^e \bmod n$$

When you verify the signature, the value of m^2 should be the PKCS #1-formatted hash code that you created from the document you received with the signature. If someone has tampered with the document in transit, the value of the hash code that you created will be different than the output of the verification function, and you should reject the signature.

Notice that the RSA signature and verification functions are the same as the encryption and decryption functions that we described in [Chapter 15](#); the signature and decryption functions are identical, as are the verification and encryption functions. This relationship does not apply to all asymmetric algorithms, and we describe a very different approach when we extend our ElGamal implementation in the [Section 16.4](#).

The DSA algorithm specification requires that PKCS #1 formatting is always used, and all hash codes are generated using the SHA-1 hashing algorithm. The SHA-1 algorithm ID is still included as part of the PKCS #1 formatting.

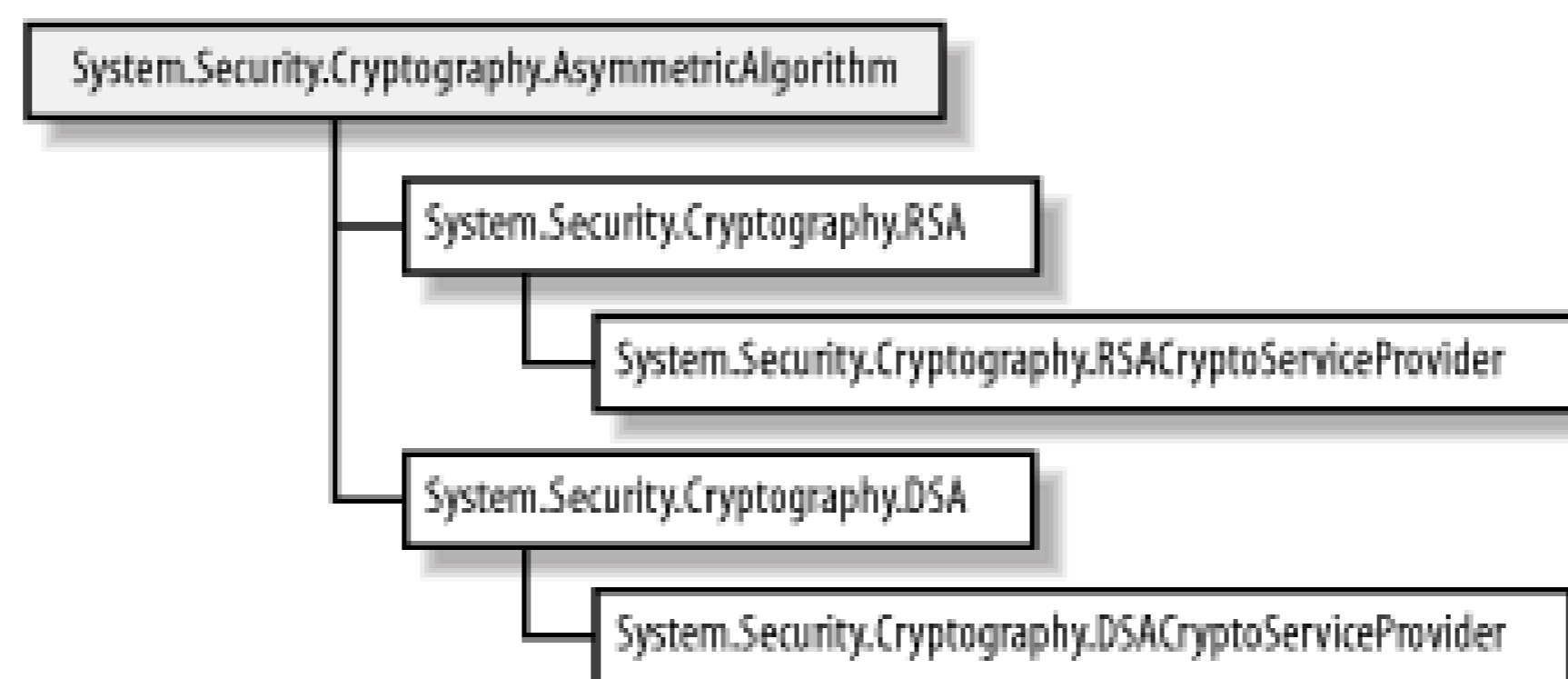
[\[Team LiB \]](#)

[[Team LiB](#)]

16.2 Programming Digital Signatures

The .NET Framework groups encryption and digital signature algorithms together as subclasses of the `AsymmetricAlgorithm` class. [Figure 16-4](#) depicts the .NET class hierarchy for digital signature algorithms, which differs from the encryption algorithm hierarchy only because of the addition of the signature-only DSA support.

Figure 16-4. The .NET Framework class hierarchy for digital signature algorithms



The general lack of consistency between the abstract algorithm classes (`RSA` and `DSA`) and their implementation counterparts (`RSACryptoServiceProvider` and `DSACryptoServiceProvider`) means that there are several equivalent ways to accomplish signature operations, which we demonstrate in the following sections.

16.2.1 Using the Abstract Class

The abstract `System.Security.Cryptography.DSA` class defines the `CreateSignature` method, which accepts a SHA-1 hash code that will be PKCS #1 formatted and signed, as the following example demonstrates (we have omitted the process of specifying the key pair to use):

```

# C#

// create the plaintext
byte[] x_plaintext = Encoding.Default.GetBytes("Programming .NET Security");

// create the SHA-1 algorithm instance and create a hash code for the plaintext
SHA1 x_sha = SHA1.Create( );
byte[] x_hashcode = x_sha.ComputeHash(x_plaintext);

// create an instance of the DSA algorithm using
// the Create method in the abstract class
DSA x_dsa = DSA.Create( );
  
```

```

// use the CreateSignature method to sign the
// SHA-1 hashcode created from the plaintext
byte[] x_signature = x_dsa.CreateSignature(x_hashcode);

# Visual Basic .NET

' create the plaintext
Dim x_plaintext As Byte( ) = Encoding.Default.GetBytes("Programming .NET Security")

' create the SHA-1 algorithm instance and create a hash code for the plaintext
Dim x_sha As SHA1 = SHA1.Create( )
Dim x_hashcode As Byte( ) = x_sha.ComputeHash(x_plaintext)

' create an instance of the DSA algorithm using
' the Create method in the abstract class
Dim x_dsa As DSA = DSA.Create( )

' use the CreateSignature method to sign the
' SHA-1 hashcode created from the plaintext
Dim x_signature As Byte( ) = x_dsa.CreateSignature(x_hashcode)

```

You must create the SHA-1 hash code yourself when using the `CreateSignature` method. The method returns the DSA signature, expressed as an array of bytes.

The DSA signature function relies on random numbers to create signatures. This means that any two signatures will be different, even when created for the same data and using the same key pair.

The `VerifySignature` method is the counterpart to `CreateSignature`, and accepts a SHA-1 hash code and the signature to verify, both expressed as an array of bytes. The following statements demonstrate how to verify a DSA signature:

```

# C#

// create the plaintext
byte[] x_plaintext
    = Encoding.Default.GetBytes("Programming .NET Security");

// define the signature to verify
byte[] x_signature = new Byte[] {0x7D, 0x2B, 0xD7, 0x3D, 0x88, 0xCB, 0x1B, 0x6B,
                                0x04, 0x62, 0x95, 0xBE, 0x28, 0x59, 0x3E, 0xC5,
                                0x40, 0xDA, 0x79, 0xFE, 0x3B, 0x25, 0x08, 0x4B,
                                0x27, 0xF1, 0x31, 0x2A, 0x6F, 0x7C, 0x6E, 0x35,
                                0x45, 0x9A, 0x49, 0x4C, 0xA4, 0x5E, 0xE6, 0xA0};

// create the SHA-1 algorithm instance and
// create a hash code for the plaintext
SHA1 x_sha = SHA1.Create( );
byte[] x_hashcode = x_sha.ComputeHash(x_plaintext);

// create an instance of the DSA algorithm using

```

```

// the Create method in the abstract class
DSA x_dsa = DSA.Create( );

// use the VerifySignature method to verify the DSA signature
bool x_signature_valid = x_dsa.VerifySignature(x_hashcode, x_signature);

# Visual Basic .NET

' create the plaintext
Dim x_plaintext As Byte( ) = Encoding.Default.GetBytes("Programming .NET Security")

' define the signature to verify
Dim x_signature As Byte( ) = New Byte( ) {&H7D, &H2B, &HD7, &H3D, &H88, &HCB, _
    &H1B, &H6B, &H4, &H62, &H95, &HBE, &H28, _
    &H59, &H3E, &HC5, &H40, &HDA, &H79, &HFE, _
    &H3B, &H25, &H8, &H4B, &H27, &HF1, &H31, _
    &H2A, &H6F, &H7C, &H6E, &H35, &H45, &H9A, _
    &H49, &H4C, &HA4, &H5E, &HE6, &HA0}

' create the SHA-1 algorithm instance and
' create a hash code for the plaintext
Dim x_sha As SHA1 = SHA1.Create( )
Dim x_hashcode As Byte( ) = x_sha.ComputeHash(x_plaintext)

' create an instance of the DSA algorithm using
' the Create method in the abstract class
Dim x_dsa As DSA = DSA.Create( )

' use the VerifySignature method to verify the DSA signature
Dim x_signature_valid As Boolean = x_dsa.VerifySignature(x_hashcode, x_signature)

```

The `VerifySignature` method returns `true` if the signature can be verified and `false` if the signature is not valid.

The abstract `RSA` class does not provide any methods to support digital signatures with the RSA algorithm.

16.2.2 Using the Implementation Class

The `RSACryptoServiceProvider` and `DSACryptoServiceProvider` classes both define four methods related to digital signatures (`DSACryptoServiceProvider` implements these methods in addition to the `CreateSignature` and `VerifySignature` methods we discussed in the previous section). [Table 16-2](#) summarizes the methods.

Table 16-2. Algorithm implementation signature methods

Method	Description
<code>SignData</code>	Creates a digital signature from the original document
<code>SignHash</code>	Creates a digital signature from a hash code
<code>VerifyData</code>	Verifies a digital signature against the original document
<code>VerifyHash</code>	Verifies a digital signature against a hash code

The `SignData` method creates a signature by generating a hash code, formatting the hash code using PKCS #1, and signing the result. The corresponding `VerifyData` method creates a PKCS #1-formatted hash code and uses it to verify a signature.

For the RSA algorithm, the hash codes are generated using an instance of `System.Security.Cryptography.HashAlgorithm`, provided as an argument to the `SignData` and `VerifyData` methods; see [Chapter 13](#) for details of the `HashAlgorithm` class. For the DSA algorithm, the SHA-1 hashing algorithm is always used to generate the hash codes.

The following statements demonstrate how to use the `SignData` method to create a signature for a byte array, and then verify the signature using the `VerifyData` method:

```
# C#
// create the plaintext
byte[] x_plaintext = Encoding.Default.GetBytes("Programming .NET Security");

// create an instance of the DSA implementation class
DSACryptoServiceProvider x_dsa = new DSACryptoServiceProvider( );
// create a signature for the plaintext
byte[] x_dsa_signature = x_dsa.SignData(x_plaintext);
// verify the signature, using the plaintext
bool x_dsa_sig_valid = x_dsa.VerifyData(x_plaintext, x_dsa_signature);

// create an instance of the RSA implementation class
RSACryptoServiceProvider x_rsa = new RSACryptoServiceProvider( );
// create an instance of the SHA-1 hashing algorithm
HashAlgorithm x_shal = HashAlgorithm.Create("SHA1");
byte[] x_rsa_signature = x_rsa.SignData(x_plaintext, x_shal);
// verify the signature, using the plaintext
bool x_rsa_sig_valid = x_rsa.VerifyData(x_plaintext, x_shal, x_rsa_signature);

# Visual Basic .NET

' create the plaintext
Dim x_plaintext As Byte( ) = Encoding.Default.GetBytes("Programming .NET Security")

' create an instance of the DSA implementation class
Dim x_dsa As DSACryptoServiceProvider = New DSACryptoServiceProvider( )
' create a signature for the plaintext
Dim x_dsa_signature As Byte( ) = x_dsa.SignData(x_plaintext)
' verify the signature, using the plaintext
Dim x_dsa_sig_valid As Boolean = x_dsa.VerifyData(x_plaintext, x_dsa_signature)
```

```

' create an instance of the RSA implementation class
Dim x_rsa As RSACryptoServiceProvider = New RSACryptoServiceProvider( )
' create an instance of the SHA-1 hashing algorithm
Dim x_shal As HashAlgorithm = HashAlgorithm.Create("SHA1")
Dim x_rsa_signature As Byte( ) = x_rsa.SignData(x_plaintext, x_shal)
' verify the signature, using the plaintext
Dim x_rsa_sig_valid As Boolean = x_rsa.VerifyData(x_plaintext, x_shal, _
x_rsa_signature)

```

The principal advantage of the `SignData` method is that the data to be signed can be read from a stream, which is useful for signing documents (which would otherwise be read into system memory). The following statements demonstrate how to use the `SignData` method to create a DSA signature by reading data from a stream. For this example, assume that we wish to sign a disk file called *mydocument.txt*.

```

# C#

// open the file as a stream
System.IO.FileStream x_stream
    = new System.IO.FileStream("mydocument.txt", System.IO.FileMode.Open);

// create an instance of the DSA implementation class
DSACryptoServiceProvider x_dsa = new DSACryptoServiceProvider( );

// create a signature using the stream
byte[] x_dsa_signature = x_dsa.SignData(x_stream);

// close the stream
x_stream.Close( );

# Visual Basic .NET

' open the file as a stream
Dim x_stream As System.IO.FileStream _
= New System.IO.FileStream("mydocument.txt", System.IO.FileMode.Open)

' create an instance of the DSA implementation class
Dim x_dsa As DSACryptoServiceProvider = New DSACryptoServiceProvider( )

' create a signature using the stream
Dim x_dsa_signature As Byte( ) = x_dsa.SignData(x_stream)

' close the stream
x_stream.Close( )

```

The `SignHash` and `VerifyHash` methods format and sign a pre-existing hash code, which is passed into the methods as an argument; the methods also require a string argument representing the identifier of the hashing algorithm that has been used to create the hash code (this is so the correct PKCS #1 algorithm ID is included in the signature). The static `MapNameToOID` method of the `CryptoConfig` class returns the ID of a hashing algorithm, so that the following statements obtain the ID for the SHA-1 algorithm:


```
# C#

string x_id = CryptoConfig.MapNameToOID("SHA1");
Console.WriteLine(x_id);
# Visual Basic .NET

Dim x_id As String = CryptoConfig.MapNameToOID("SHA1")
Console.WriteLine(x_id)
```

These statements produce the following output, which is a decimal representation of part of the ID that we listed in [Table 16-1](#):

```
1.3.14.3.2.26
```

The following statements demonstrate how to create and verify a signature using the `SignHash` and `VerifyHash` methods and the DSA algorithm. Notice that the DSA implementation requires the ID of the hashing algorithm, even though the DSA specification demands that SHA-1 is always used:

```
# C#

// create the plaintext
byte[] x_plaintext = Encoding.Default.GetBytes("Programming .NET Security");

// create a hash code for the plaintext, using the SHA-1 algorithm
byte[] x_hashcode = HashAlgorithm.Create("SHA1").ComputeHash(x_plaintext);

// create an instance of the DSA implementation class
DSACryptoServiceProvider x_dsa = new DSACryptoServiceProvider( );

// create a DSA signature using the hash code
byte[] x_signature = x_dsa.SignHash(x_hashcode, CryptoConfig.MapNameToOID("SHA1"));

// verify the signature
bool x_sig_valid = x_dsa.VerifyHash(x_hashcode, CryptoConfig.MapNameToOID("SHA1"),
    x_signature);

# Visual Basic .NET

' create the plaintext
Dim x_plaintext As Byte( ) = Encoding.Default.GetBytes("Programming .NET Security")

' create a hash code for the plaintext, using the SHA-1 algorithm
Dim x_hashcode As Byte( ) = HashAlgorithm.Create("SHA1").ComputeHash(x_plaintext)

' create an instance of the DSA implementation class
Dim x_dsa As DSACryptoServiceProvider = New DSACryptoServiceProvider( )

' create a DSA signature using the hash code
Dim x_signature As Byte( ) = x_dsa.SignHash(x_hashcode, _
    CryptoConfig.MapNameToOID("SHA1"))

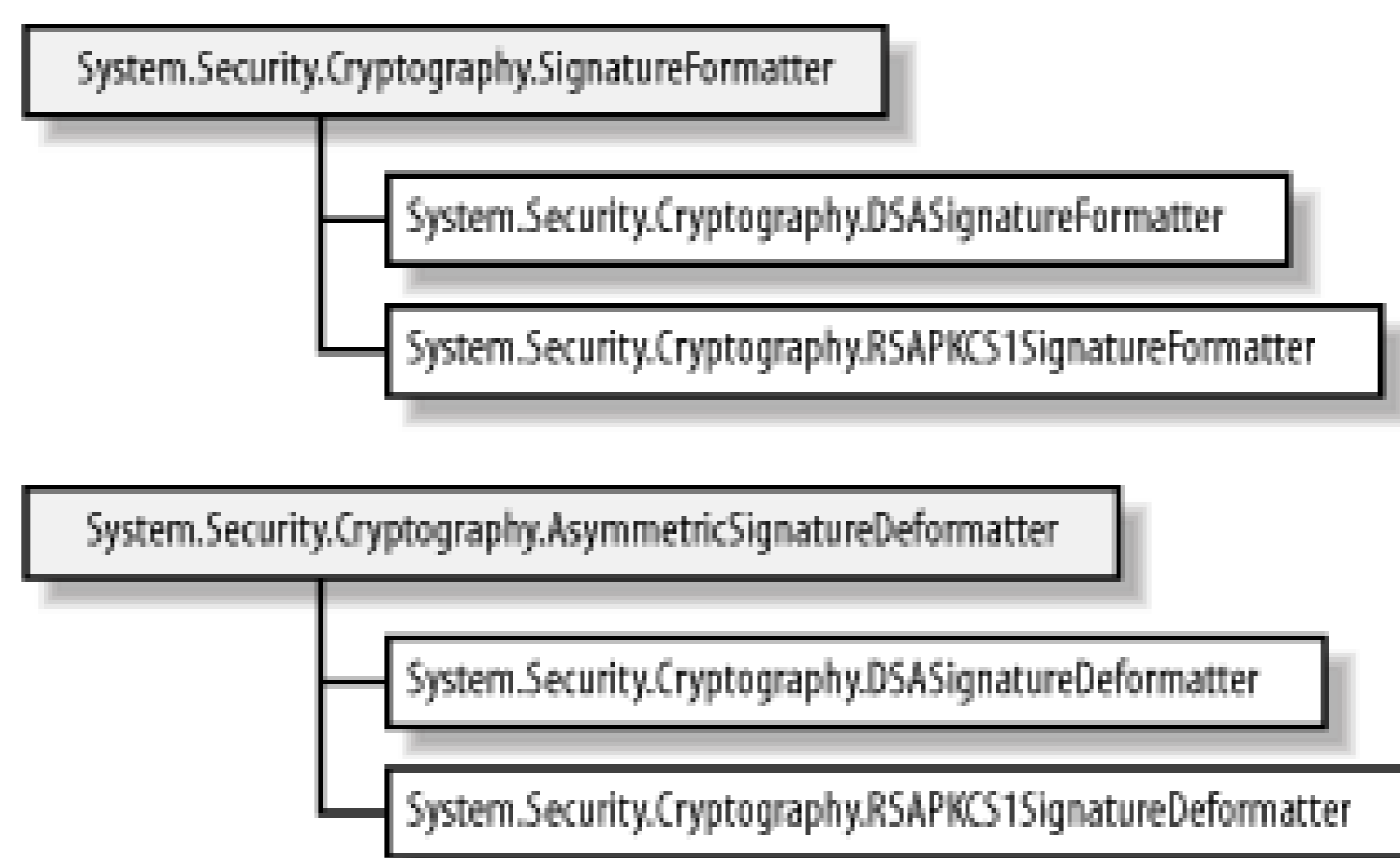
' verify the signature
```

```
Dim x_sig_valid As Boolean = x_dsa.VerifyHash(x_hashcode, _
CryptoConfig.MapNameToOID("SHA1"), x_signature)
```

16.2.3 Using the Signature Formatter Classes

The .NET Framework specifies separate formatter and deformatter classes, as shown by the hierarchy in [Figure 16-5](#). These classes format a hash code using PKCS #1, and sign the result in conjunction with the abstract and implementation classes. The functionality that these classes provide is equivalent to using the methods that we have described in the preceding sections; there is no specific advantage to using the formatter and deformatter classes.

Figure 16-5. The .NET Framework class hierarchy for signature formatters



The abstract `AsymmetricSignatureFormatter` class defines the methods described in [Table 16-3](#). The abstract class does not specify how to format a hash code, and so implementation classes can define any formatting that is appropriate to an algorithm; both of the default .NET implementation classes use PKCS #1 formatting.

Table 16-3. AsymmetricSignatureFormatter members

Method	Description
<code>SetHashAlgorithm</code>	Specifies the name of the hashing algorithm that was used to create the hash code that will be signed; in the case of PKCS #1 formatting, the ID of the algorithm will be included in the signed data
<code>SetKey</code>	Specifies an instance of the asymmetric signing algorithm that will be used to create the digital signature
<code>CreateSignature</code>	Creates a digital signature from a hash code

The following statements demonstrate how to use the `RSAPKCS1SignatureFormatter` class to create a digital signature:

```
# C#
```

```

// create the plaintext
byte[] x_plaintext = Encoding.Default.GetBytes("Programming .NET Security");

// create a hash code for the plaintext, using the SHA-1 algorithm
byte[] x_hashcode = HashAlgorithm.Create("SHA1").ComputeHash(x_plaintext);

// create an instance of the DSA implementation class
DSACryptoServiceProvider x_dsa = new DSACryptoServiceProvider( );

// create the signature formatter
DSASignatureFormatter x_formatter = new DSASignatureFormatter( );
// set the instance of the DSA algorithm that will sign the data
x_formatter.SetKey(x_dsa);
// set the name of the hashing algorithm we used to create the hash code
x_formatter.SetHashAlgorithm("SHA1");
// create the formatted DSA signature
byte[] x_signature = x_formatter.CreateSignature(x_hashcode);

# Visual Basic .NET

' create the plaintext
Dim x_plaintext As Byte( ) = Encoding.Default.GetBytes("Programming .NET Security")

' create a hash code for the plaintext, using the SHA-1 algorithm
Dim x_hashcode As Byte( ) = HashAlgorithm.Create("SHA1").ComputeHash(x_plaintext)

' create an instance of the DSA implementation class
Dim x_dsa As DSACryptoServiceProvider = New DSACryptoServiceProvider( )

' create the signature formatter
Dim x_formatter As DSASignatureFormatter = New DSASignatureFormatter( )
' set the instance of the DSA algorithm that will sign the data
x_formatter.SetKey(x_dsa)
' set the name of the hashing algorithm we used to create the hash code
x_formatter.SetHashAlgorithm("SHA1")
' create the formatted DSA signature
Dim x_signature As Byte( ) = x_formatter.CreateSignature(x_hashcode)

```

The abstract `AsymmetricSignatureDeformatter` is the basis for classes that verify digital signatures created by subclasses of the `AsymmetricSignatureFormatter` class. [Table 16-4](#) summarizes the methods of the `AsymmetricSignatureDeformatter` class.

Table 16-4. `AsymmetricSignatureDeformatter` members

Method	Description
<code>SetHashAlgorithm</code>	Specifies the name of the hashing algorithm that will be used to verify the signature

Method	Description
<code>SetKey</code>	Specifies an instance of the asymmetric signing algorithm that will be used to verify the digital signature
<code>VerifySignature</code>	Verifies a digital signature

The following statements demonstrate how to verify a digital signature using the `DSASignatureDeformatter` class:

```
# C#

// create the plaintext
byte[] x_plaintext = Encoding.Default.GetBytes("Programming .NET Security");

// create a hash code for the plaintext, using the SHA-1 algorithm
byte[] x_hashcode = HashAlgorithm.Create("SHA1").ComputeHash(x_plaintext);

// define the signature to verify
byte[] x_signature = new Byte[] {0x7D, 0x2B, 0xD7, 0x3D, 0x88, 0xCB, 0x1B, 0x6B,
                                0x04, 0x62, 0x95, 0xBE, 0x28, 0x59, 0x3E, 0xC5,
                                0x40, 0xDA, 0x79, 0xFE, 0x3B, 0x25, 0x08, 0x4B,
                                0x27, 0xF1, 0x31, 0x2A, 0x6F, 0x7C, 0x6E, 0x35,
                                0x45, 0x9A, 0x49, 0x4C, 0xA4, 0x5E, 0xE6, 0xA0};

// create an instance of the DSA implementation class
DSACryptoServiceProvider x_dsa = new DSACryptoServiceProvider( );

// create the signature deformatter
DSASignatureDeformatter x_deformatter = new DSASignatureDeformatter( );
// set the instance of the DSA algorithm that will verify the signature
x_deformatter.SetKey(x_dsa);
// set the name of the hashing algorithm we used to create the hash code
x_deformatter.SetHashAlgorithm("SHA1");
// verify the DSA signature
bool x_signature_valid = x_deformatter.VerifySignature(x_hashcode, x_signature);

# Visual Basic .NET

' create the plaintext
Dim x_plaintext As Byte( ) = Encoding.Default.GetBytes("Programming .NET Security")

' create a hash code for the plaintext, using the SHA-1 algorithm
Dim x_hashcode As Byte( ) = HashAlgorithm.Create("SHA1").ComputeHash(x_plaintext)

' define the signature to verify
Dim x_signature As Byte( ) = New Byte( ) {&H7D, &H2B, &HD7, &H3D, &H88, &HCB, _
&H1B, &H6B, &H4, &H62, &H95, &HBE, &H28, _
&H59, &H3E, &HC5, &H40, &HDA, &H79, &HFE, _
&H3B, &H25, &H8, &H4B, &H27, &HF1, &H31, _
&H2A, &H6F, &H7C, &H6E, &H35, &H45, &H9A, _
&H49, &H4C, &HA4, &H5E, &HE6, &HA0}
```

```
' create an instance of the DSA implementation class
Dim x_dsa As DSACryptoServiceProvider = New DSACryptoServiceProvider( )

' create the signature deformatter
Dim x_deformatter As DSASignatureDeformatter = New DSASignatureDeformatter( )
' set the instance of the DSA algorithm that will verify the signature
x_deformatter.SetKey(x_dsa)
' set the name of the hashing algorithm we used to create the hash code
x_deformatter.SetHashAlgorithm("SHA1")

' verify the DSA signature
Dim x_signature_valid As Boolean = _
x_deformatter.VerifySignature(x_hashcode, x_signature)
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

16.3 Programming XML Signatures

The .NET Framework supports the XML Signature specification (commonly referred to as XMLDSIG), which provides a standard approach to creating and representing signatures for XML documents.

In this section, we introduce the .NET XMLDSIG classes. We do not discuss all of the options available to you or provide in-depth coverage of the standard itself. The XMLDSIG standard is complex, and full coverage is outside the scope of this book. For full details of the XML Signature specification, consult the World Wide Web Consortium web site, at <http://www.w3.org/TR/xmlsig-core>. Consult the .NET documentation for full details of the XMLDSIG classes. You should have a basic understanding of how the .NET Framework supports XML.

16.3.1 XMLDSIG Explained

The .NET Framework includes classes that support creating and verifying XMLDSIG signatures for XML documents; in other words, the .NET XMLDSIG classes allow you to sign XML documents to create signatures that are also XML documents themselves.

Throughout this section, we will base our examples on creating a signature for the following simple XML, which we assume is in a file named *book.xml*.

```
<book>
  <title>Programming .NET Security</title>
  <author>Adam Freeman</author>
  <author>Allen Jones</author>
</book>
```

All of the .NET XMLDSIG classes are contained in the `System.Security.Cryptography.Xml` namespace, which is included in *System.Security.dll*.

16.3.2 Signing an XML Document

In the following sections, we demonstrate how to create an XMLDSIG signature for our sample XML document.

16.3.2.1 Creating the reference

To create a signature, the first step is to create a reference to the XML document that you want to sign. The .NET `Reference` class allows you to create a reference using a URL string or a stream of data. When the signature is created, the XML data will be read from the URL or the stream to create the signature. The following statements demonstrate how to create a `Reference` object:

```
# C#
```

```

// specify the URL of the document we want to sign
string x_url = "http://www.mydomain.com/book.xml";
// create a reference that points to the URL
Reference x_url_ref = new Reference(x_url);

// open a file stream for the document we want to sign
FileStream x_stream = new FileStream("book.xml", FileMode.Open);
// create a Reference that will use the stream
Reference x_stream_ref = new Reference(x_stream);

# Visual Basic .NET

' specify the URL of the document we want to sign
Dim x_url As String = "http:'www.mydomain.com/book.xml"
' create a reference that points to the URL
Dim x_url_ref As Reference = New Reference(x_url)

' open a file stream for the document we want to sign
Dim x_stream As FileStream = New FileStream("book.xml", FileMode.Open)
' create a Reference that will use the stream
Dim x_stream_ref As Reference = New Reference(x_stream)

```

16.3.2.2 Creating the SignedXML

The `System.Security.Cryptography.Xml.SignedXml` class is responsible for creating the signature document. Create an instance of this class, and then configure it with the settings you wish to use in creating the signature. Create a new instance of the `SignedXml` class using the default constructor, and then use the `AddReference` method to add the `Reference` object you created in the previous section:

```

# C#

SignedXml x_signed_xml = new SignedXml( );
x_signed_xml.AddReference(x_stream_ref);

# Visual Basic .NET

Dim x_signed_xml As SignedXml = New SignedXml( )
x_signed_xml.AddReference(x_stream_ref)

```

16.3.2.3 Setting the signing algorithm

you need to create a new asymmetric signing algorithm instance and assign it to the `SignedXml` object using the `SigningKey` property, as illustrated by the following statements:

```

# C#

// create a new instance of the DSA algorithm
DSA x_dsa = DSA.Create( );

```

```

// configure the signing key
// ...

// set the algorithm for the SignedXml
x_signed_xml.SigningKey = x_dsa;

# Visual Basic .NET

' create a new instance of the DSA algorithm
Dim x_dsa As DSA = DSA.Create( )

' configure the signing key
' ...

' set the algorithm for the SignedXml
x_signed_xml.SigningKey = x_dsa

```

These statements have the effect of defining the algorithm and the keys that will be used to create the signature. The .NET classes support using the RSA and DSA algorithms for generating XML Signature documents.

16.3.2.4 Creating the signature

Create the signature by calling the `ComputeSignature` method. You can then obtain the signature by calling the `GetXml` method, and obtain a string representation by using the `OuterXml` property, as shown below:

```

# C#

x_signed_xml.ComputeSignature( );
Console.WriteLine(x_signed_xml.GetXml( ).OuterXml);

# Visual Basic .NET

x_signed_xml.ComputeSignature( )
Console.WriteLine(x_signed_xml.GetXml( ).OuterXml)

```

The result of creating a DSA signature for our sample XML is as follows:

```

<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod
      Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
    <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1" />
    <Reference>
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
      <DigestValue>eJgWGA+lXiqrOpXa+2NSBb01HS4=</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>u5zleI17IY5MV8sxsirlGIqRaZ9LgExoKdQwgCMqaPxWZX7VOAcT9g==</SignatureValue>

```



```
</Signature>
```

Notice that the XML signature document contains information about the signing algorithm (DSA), the hashing algorithm (SHA-1), and the signature value, which has been Base64-encoded. This is a formalized representation of the information that is needed to verify the signature, which would otherwise have to be communicated between parties in an ad hoc manner (for example, Alice and Bob exchange messages to agree on the hashing and signing algorithms before they exchange signed data).

16.3.3 Embedding Objects in the Signature

The signature that you created in the previous section is separate from the data it relates to, as are the other signatures you created in [Section 16.2](#). The XMLDSIG standard supports including the original data as part of the XML signature document.

To include your sample XML document, load the XML document and use it to create a new instance of the `DataObject` class:

```
# C#

// load the XML document
XmlDocument x_xml_doc = new XmlDocument( );
x_xml_doc.Load("book.xml");

// create the data object for the xml document
DataObject x_obj = new DataObject( );
x_obj.Data = x_xml_doc.ChildNodes;
x_obj.Id = "book";

# Visual Basic .NET

' load the XML document
Dim x_xml_doc As XmlDocument = New XmlDocument( )
x_xml_doc.Load("book.xml")

' create the data object for the xml document
Dim x_obj As DataObject = New DataObject( )
x_obj.Data = x_xml_doc.ChildNodes
x_obj.Uri = "book"
```

Assign the XML data to the `DataObject` using the `Data` property, and assign an ID to the object using the `URI` property; for example, choose the ID "book." Continue by creating a new reference, but instead of using a URL or a stream for the data, use a "local" reference, where you place a # symbol in front of our object ID (in this case, the local reference is `#book`):

```
# C#

// create the local reference
Reference x_local_reference = new Reference( );
x_local_reference.Uri = "#book";
```

```
# Visual Studio .NET
```

```
' create the local reference
Dim x_local_reference As Reference = New Reference( )
x_local_reference.Uri = "#book"
```

Create a new instance of the `SignedXml` class, and use the `AddReference` and `AddObjects` methods to add your reference and data:

```
# C#
```

```
// create the SignedXml instance
SignedXml x_signed_xml = new SignedXml( );
// add the local reference
x_signed_xml.AddReference(x_local_reference);
// add the data object
x_signed_xml.AddObject(x_obj);
```

```
# Visual Basic .NET
```

```
' create the SignedXml instance
Dim x_signed_xml As SignedXml = New SignedXml( )
' add the local reference
x_signed_xml.AddReference(x_local_reference)
' add the data object
x_signed_xml.AddObject(x_obj)
```

Finally, set the instance of the signing algorithm and compute the signature:

```
# C#
```

```
// create a new instance of the DSA algorithm
DSA x_dsa = DSA.Create( );

// configure the signing key
// ...

// set the algorithm for the SignedXml
x_signed_xml.SigningKey = x_dsa;

// compute the signature
x_signed_xml.ComputeSignature( );
Console.WriteLine(x_signed_xml.GetXml( ).OuterXml);
```

```
# Visual Basic .NET
```

```
' create a new instance of the DSA algorithm
Dim x_dsa As DSA = DSA.Create( )

' configure the signing key
' ...
```

```
' set the algorithm for the SignedXml
x_signed_xml.SigningKey = x_dsa

' compute the signature
x_signed_xml.ComputeSignature( )
Console.WriteLine(x_signed_xml.GetXml( ).OuterXml)
```

The resulting signature is shown below; the included data is highlighted:

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod
      Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
    <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1" />
    <Reference URI="#book">
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
      <DigestValue>1UhFInEywZYY/3eLgCqg5w+IROI=</DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>DUuD4ZJd8YiDLIr7HimDWGmCXYQDpX1jv1xRxKLgccw/lTyh3XjB6Q==
</SignatureValue>
  <Object Id="book">
    <book xmlns="">
      <title>Programming .NET Security</title>
      <author>Adam Freeman</author>
      <author>Allen Jones</author>
    </book>
  </Object>
</Signature>
```

By including the data in this way, you create an XML document that contains the data that was signed, details of how the signature was created (hashing and signature algorithms), and the signature itself, which allows Alice to send a single XML document to Bob when exchanging signed messages.

16.3.4 Verifying an XML Signature

Begin verifying the signature by loading the XML Signature document (which you have saved as the file *xmlsig.xml*) into an instance of the `SignedXml` class:

```
# C#

// load the XML document
XmlDocument x_xml_doc = new XmlDocument( );
x_xml_doc.Load("xmlsig.xml");

// create the SignedXml instance
SignedXml x_signed_xml = new SignedXml( );

// get the node list from the XML sig doc
XmlNodeList x_node_list = x_xml_doc.GetElementsByTagName("Signature");
```

```

// load the XML signature document
x_signed_xml.LoadXml((XmlElement)x_node_list[0]);

# Visual Basic .NET

' load the XML document
Dim x_xml_doc As XmlDocument = New XmlDocument( )
x_xml_doc.Load("xmlsig.xml")

' create the SignedXml instance
Dim x_signed_xml As SignedXml = New SignedXml( )

' get the node list from the XML sig doc
Dim x_node_list As XmlNodeList = x_xml_doc.GetElementsByTagName("Signature")
' load the XML signature document
x_signed_xml.LoadXml(CType(x_node_list(0), XmlElement))

```

You need to create the asymmetric algorithm that will be used to verify the signature; you should configure the algorithm class with the public parameters from the key pair that was used to generate the signature.

Once you have created the algorithm, create an instance of the `KeyInfo` class, which accepts an instance of either the `DSAKeyValue` or `RSAKeyValue` class, depending on which algorithm has been used. Configure the `SignedXml` class to use the asymmetric algorithm by setting the value of the `KeyInfo` property:

```

# C#

// create a new instance of the DSA algorithm
DSA x_dsa = DSA.Create( );

// configure the signing key
// ...

// create the KeyInfo instance
KeyInfo x_info = new KeyInfo( );

// add the DSA instance to the KeyInfo
x_info.AddClause(new DSAKeyValue(x_dsa));

// configure the SignedXml class to use the
// DSA keys for verification
x_signed_xml.KeyInfo = x_info;

# Visual Basic .NET

' create a new instance of the DSA algorithm
Dim x_dsa As DSA = DSA.Create( )

' configure the signing key
' ...

```

```
' create the KeyInfo instance
Dim x_info As KeyInfo = New KeyInfo( )

' add the DSA instance to the KeyInfo
x_info.AddClause(New DSAKeyValue(x_dsa))

' configure the SignedXml class to use the
' DSA keys for verification
x_signed_xml.KeyInfo = x_info
```

Finally, verify the signature by calling the `CheckSignature` method of the `SignedXml` class; this method returns `true` if the signature is valid and `false` if it is not:

```
# C#

// verify the signature
bool x_signature_valid = x_signed_xml.CheckSignature( );

# Visual Basic .NET

' verify the signature
Dim x_signature_valid As Boolean = x_signed_xml.CheckSignature( )
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

16.4 Extending the .NET Framework

In [Chapter 15](#), we extended the .NET Framework by creating an implementation of the ElGamal algorithm that supported encryption. In this section, we extend our implementation to include support for creating and verifying digital signatures.

We have provided a C# implementation of the ElGamal algorithm only. Like almost all encryption algorithms, ElGamal relies on mathematical operations that are not possible in Visual Basic .NET without creating additional support functions to compensate for the limited numeric support the language provides.

16.4.1 The ElGamal Signature Functions Explained

The ElGamal algorithm supports encryption and digital signatures using the same key pair. Keys created by following the key generation protocol (discussed in [Chapter 15](#)) can be used to create and verify digital signatures using the functions that we explain in this section. As a reminder, the parameters p , g , and y form the public key, and the parameter x is the private key.

The basic protocol for creating an ElGamal signature is:

1. Choose a random number, k , that is relatively prime to $(p-1)$.
2. Compute the signature element, a , as follows:

$$a = g^k \text{ mod } p$$
3. Compute the signature element, b , as follows (m is the data to sign):
4. Concatenate a and b to form the signature.

To verify an ElGamal signature, we follow the following protocol:

1. Split the digital signature to form the signature elements, a and b .
2. Compute the first verification element, e^1 , as follows:

$$e_1 = y^a a^b \text{ mod } p$$
3. Compute the second verification element, e^2 , as follows:

$$e_2 = g^m \text{ mod } p$$
4. The signature is valid if e^1 and e^2 are of equal value and is not valid otherwise.

Notice that, unlike RSA, the ElGamal algorithm specifies different functions for data encryption and digital signatures.

16.4.2 Defining the Signature Function Class

Begin by defining the `ElGamalSignature` class, which implements the basic signature and verification functions that we described in the preceding section:

```
using System;
using System.Security.Cryptography;
```

```
public class ElGamalSignature {
```

Define the `mod` method to ensure that you always receive a positive result when using the modular arithmetic support included in the `BigInteger` class:

```
    public static BigInteger mod(BigInteger p_base, BigInteger p_val) {
        BigInteger x_result = p_base % p_val;
        if (x_result < 0) {
            x_result += p_val;
        }
        return x_result;
    }
}
```

The `CreateSignature` class implements the ElGamal signature function and creates a digital signature; the arguments to this method are the data to sign and the key to use:

```
    public static byte[] CreateSignature(byte[] p_data,
        ElGamalKeyStruct p_key_struct) {
```

Begin by generating a random number, k , that is relatively prime to $p - 1$. Refer to [Table 15-6](#) for a summary of the methods we use in the `BigInteger` class:

```
    // define P -1
    BigInteger x_pminusone = p_key_struct.P - 1;
    // create K, which is the random number
    BigInteger K;
    do {
        K = new BigInteger( );
        K.genRandomBits(p_key_struct.P.bitCount( ) -1, new Random( ));
    } while (K.gcd(x_pminusone) != 1);
```

The following statements compute the values of the signature elements, a and b . Although these statements may appear convoluted, they do compute the values that you require; their appearance is a result of the requirements of the `BigInteger` class and the fact that we have defined the `mod` method in this class:

```
    // compute the values A and B
    BigInteger A = p_key_struct.G.modPow(K, p_key_struct.P);
    BigInteger B = mod(mod(K.modInverse(x_pminusone)
        * (new BigInteger(p_data)
```

```
-(p_key_struct.X*(A)), (x_pminusone)), (x_pminusone));
```

Having completed the computations, now create a byte array to hold the signature data and fill it with the signature elements that you created previously to create the complete signature:

```
// copy the bytes from A and B into the result array
byte[] x_a_bytes = A.getBytes( );
byte[] x_b_bytes = B.getBytes( );
// define the result size
int x_result_size = ((p_key_struct.P.bitCount( ) + 7) / 8) * 2);
// create an array to contain the ciphertext
byte[] x_result = new byte[x_result_size];
// populate the arrays
Array.Copy(x_a_bytes, 0, x_result, x_result_size / 2
    - x_a_bytes.Length, x_a_bytes.Length);
Array.Copy(x_b_bytes, 0, x_result, x_result_size
    - x_b_bytes.Length, x_b_bytes.Length);

// return the result array
return x_result;
}
```

The `VerifySignature` method implements the ElGamal verification function, accepting arguments that represent the data that was signed, the signature to verify, and the key with which to perform the verification:

```
public static bool VerifySignature(byte[] p_data, byte[] p_sig,
    ElGamalKeyStruct p_key_struct) {
```

We begin by splitting the signature into two sections and creating instances of the `BigInteger` class to represent the signature elements `a` and `b`:

```
// define the result size
int x_result_size = p_sig.Length/2;

// extract the byte arrays that represent A and B
byte[] x_a_bytes = new byte[x_result_size];
Array.Copy(p_sig, 0, x_a_bytes, 0, x_a_bytes.Length);
byte[] x_b_bytes = new byte[x_result_size];
Array.Copy(p_sig, x_result_size, x_b_bytes, 0, x_b_bytes.Length);

// create big integers from the byte arrays
BigInteger A = new BigInteger(x_a_bytes);
BigInteger B = new BigInteger(x_b_bytes);
```

The remaining statements compute the two verification elements, and the method returns `2` if the elements have the same value (the signature is valid) and `false` otherwise (the signature is not valid):

```
// create the two results
BigInteger x_result1 = mod(p_key_struct.Y.modPow(A, p_key_struct.P)
    * A.modPow(B, p_key_struct.P), p_key_struct.P);
```



```

        BigInteger x_result2 = p_key_struct.G.modPow(new BigInteger(p_data),
            p_key_struct.P);

        // return true if the two results are the same

        return x_result1 == x_result2;
    }
}

```

16.4.3 Implementing the Managed Class Methods

We now define the implementations for the signature function methods in the managed algorithm class, which we left unfinished in [Chapter 15](#). The `Sign` and `VerifySignature` methods call the static `CreateSignature` and `VerifySignature` methods in the `ElGamalSignature` class, which we defined in the preceding section. Both methods check to see if the user has specified a key and will create a new pair as required:

```

public class ElGamalManaged : ElGamal {

    // ... other methods

    public override byte[] Sign(byte[] p_hashcode) {
        if (NeedToGenerateKey( )) {
            // we need to create a new key before we can export
            CreateKeyPair(KeySizeValue);
        }
        return ElGamalSignature.CreateSignature(p_hashcode, o_key_struct);
    }

    public override bool VerifySignature(byte[] p_hashcode, byte[] p_signature) {
        if (NeedToGenerateKey( )) {
            // we need to create a new key before we can export
            CreateKeyPair(KeySizeValue);
        }
        return ElGamalSignature.VerifySignature(p_hashcode,
            p_signature, o_key_struct);
    }

    // ... other methods
}

```

This implementation means that the `Sign` and `VerifySignature` methods depend directly on the "raw" signature functions, which are created without PKCS #1 formatting. We define separate formatting classes in the following sections.

16.4.4 Defining the PKCS #1 Helper Class

We have defined the `ElGamalSignatureFormatHelper` class to simplify our formatter and deformatter implementations. This class is responsible for formatting blocks of data in accordance with the PKCS #1 formatting technique:

```
using System;
using System.Security.Cryptography;

public class ElGamalSignatureFormatHelper {
```

We begin by defining the set of algorithm IDs that we listed in [Table 16-1](#). These IDs are included in the PKCS #1-formatted data; see [Section 16.2](#) for details of PKCS #1. We have defined the algorithm IDs as byte arrays to simplify the formatting process:

```
private static byte[] MD5_BYTES
    = new byte[] {0x30, 0x20, 0x30, 0x0C, 0x06, 0x08, 0x2A, 0x86,
                 0x48, 0x86, 0xF7, 0x0D, 0x02, 0x05, 0x05, 0x00,
                 0x04, 0x10};

private static byte[] SHA1_BYTES
    = new byte[] {0x30, 0x21, 0x30, 0x09, 0x06, 0x05, 0x2b, 0x0E, 0x03,
                 0x02, 0x1A, 0x05, 0x00, 0x04, 0x14};

private static byte[] SHA256_BYTES
    = new byte[] {0x30, 0x31, 0x30, 0x0d, 0x06, 0x09, 0x60, 0x86,
                 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01, 0x05,
                 0x00, 0x04, 0x20};

private static byte[] SHA384_BYTES
    = new byte[] {0x30, 0x41, 0x30, 0x0d, 0x06, 0x09, 0x60, 0x86,
                 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x02, 0x05,
                 0x00, 0x04, 0x30};

private static byte[] SHA512_BYTES
    = new byte[] {0x30, 0x51, 0x30, 0x0d, 0x06, 0x09, 0x60, 0x86,
                 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x03, 0x05,
                 0x00, 0x04, 0x40};
```

The `GetHashAlgorithmID` returns a byte array representing the ID for a hashing algorithm, which is specified as the method argument:

```
private static byte[] GetHashAlgorithmID(HashAlgorithm p_hash) {
    if (p_hash is MD5) {
        return MD5_BYTES;
    } else if (p_hash is SHA1) {
        return SHA1_BYTES;
    } else if (p_hash is SHA256) {
        return SHA256_BYTES;
    } else if (p_hash is SHA384) {
        return SHA384_BYTES;
    } else if (p_hash is SHA512) {
        return SHA512_BYTES;
    } else {
        throw new ArgumentException("Unknown hashing algorithm", "p_hash");
    }
}
```

`CreateEMSA_PKCS1_v1_5_ENCODE` formats a block of data in accordance with the PKCS #1 technique that we described in [Section 16.2](#) of this chapter:

```
public static byte[] CreateEMSA_PKCS1_v1_5_ENCODE(byte[] p_hashcode,
    HashAlgorithm p_hash_alg, int p_key_length) {

    // Concatenate the algorithm ID for the hash algorithm
    // and the hash code to form T
    byte[] x_algorithm_id = GetHashAlgorithmID(p_hash_alg);
    byte[] T = new byte[p_hashcode.Length + x_algorithm_id.Length];
    Array.Copy(x_algorithm_id, 0, T, 0, x_algorithm_id.Length);
    Array.Copy(p_hashcode, 0, T, x_algorithm_id.Length, p_hashcode.Length);

    // Generate an octet string PS consisting of p_key_length - T.Length -
    // 3 octets with hexadecimal value 0xff.
    // The length of PS will be at least 8 octets.
    int x_PS_length = p_key_length - T.Length - 3;
    byte[] PS = new byte[x_PS_length < 0 ? 8 : x_PS_length];
    for (int i = 0; i < PS.Length; i++) {
        PS[i] = 0xFF;
    }
    // Concatenate PS, the DER encoding T, and other padding to form the
    // encoded message EM as EM = 0x00 || 0x01 || PS || 0x00 || T .
    byte[] EM = new byte[3 + PS.Length + T.Length];
    EM[0] = 0x00;
    EM[1] = 0x01;
    Array.Copy(PS, 0, EM, 2, PS.Length);
    EM[PS.Length + 2] = 0x00;
    Array.Copy(T, 0, EM, PS.Length + 3, T.Length);
    // Output EM.
    return EM;
}
}
```

16.4.5 Defining the Signature Formatter Class

The `ElGamalPKCS1SignatureFormatter` class extends `AsymmetricSignatureFormatter`, and uses `ElGamalSignatureFormatHelper` to create a PKCS #1-formatted hash code, which is passed to the `Sign` method of the `ElGamalManaged` class:

```
using System;
using System.Security.Cryptography;

public class ElGamalPKCS1SignatureFormatter : AsymmetricSignatureFormatter {
    private string o_hash_name; // the hash algorithm to use
    private ElGamalManaged o_key; // the ElGamal algorithm

    public override void SetHashAlgorithm(string p_name) {
        o_hash_name = p_name;
    }
}
```

```

public override void SetKey(AsymmetricAlgorithm p_key) {
    if (p_key is ElGamalManaged) {
        o_key = p_key as ElGamalManaged;
    } else {
        throw new ArgumentException(
            "Key is not an instance of ElGamalManaged", "p_key");
    }
}

public override byte[] CreateSignature(byte[] p_data) {
    if (o_hash_name == null || o_key == null) {
        throw new
            CryptographicException("Key and Hash Algorithm must be set");
    } else {
        // create the hashing algorithm
        HashAlgorithm x_hash_alg = HashAlgorithm.Create(o_hash_name);
        // create a PKCS1 formatted block from the data
        byte[] x_pkcs
            = ElGamalSignatureFormatHelper.CreateEMSA_PKCS1_v1_5_ENCODE(p_data,
                x_hash_alg, o_key.KeyStruct.P.bitCount( ));
        // create and return the signature
        return o_key.Sign(x_pkcs);
    }
}
}
}

```

16.4.6 Defining the Signature Deformatter Class

The `ElGamalPKCS1SignatureDefomatter` class extends `AsymmetricSignatureDefomatter` and uses `ElGamalSignatureFormatHelper` to create a PKCS #1-formatted hash code to verify a signature, using the `VerifySignature` method of the `ElGamalManaged` class:

```

using System;
using System.Security.Cryptography;

public class ElGamalPKCS1SignatureDefomatter : AsymmetricSignatureDefomatter

    private string          o_hash_name;    // the hash algorithm to use
    private ElGamalManaged o_key;         // the ElGamal algorithm

    public override void SetHashAlgorithm(string p_name) {
        o_hash_name = p_name;
    }

    public override void SetKey(AsymmetricAlgorithm p_key) {
        if (p_key is ElGamalManaged) {
            o_key = p_key as ElGamalManaged;
        } else {
            throw new ArgumentException(
                "Key is not an instance of ElGamalManaged", "p_key");
        }
    }
}

```

```

    }

    public override bool VerifySignature(byte[] p_data, byte[] p_signature) {
        if (o_hash_name == null || o_key == null) {
            throw new
                CryptographicException("Key and Hash Algorithm must be set");
        } else {
            // create the hashing algorithm
            HashAlgorithm x_hash_alg = HashAlgorithm.Create(o_hash_name);
            // create a PKCS1 formatted block from the data
            byte[] x_pkcs
                = ElGamalSignatureFormatHelper.CreateEMSA_PKCS1_v1_5_ENCODE(
                    p_data, x_hash_alg, o_key.KeyStruct.P.bitCount( ));

            // create and return the signature
            return o_key.VerifySignature(x_pkcs, p_signature);
        }
    }
}

```

16.4.7 Testing the Algorithm

The following statements test the support for raw and PKCS #1-formatted ElGamal signatures:

```

using System;
using System.Text;
using System.Security.Cryptography;

public class Test {

    public static void Main( ) {
        // define the byte array that we will use
        // as plaintext
        byte[] x_plaintext
            = Encoding.Default.GetBytes("Programming .NET Security");

        // Create an instance of the algorithm and generate some keys
        ElGamal x_alg = new ElGamalManaged( );
        // set the key size - keep is small to speed up the tests
        x_alg.KeySize = 384;
        // extract and print the xml string (this will cause
        // a new key pair to be generated)
        string x_xml_string = x_alg.ToXmlString(true);
        Console.WriteLine("\n{0}\n", x_xml_string);

        // create a signature for the plaintext
        ElGamal x_sign_alg = new ElGamalManaged( );
        // set the keys - note that we export with the
        // private parameters since we are signing data
        x_sign_alg.FromXmlString(x_alg.ToXmlString(true));
        byte[] x_signature = x_sign_alg.Sign(x_plaintext);
    }
}

```

```
// verify the signature
ElGamal x_verify_alg = new ElGamalManaged( );
// set the keys - note that we export without the
// private parameters since we are verifying data
x_verify_alg.FromXmlString(x_alg.ToXmlString(false));
Console.WriteLine("BASIC SIGNATURE: {0}",
    x_verify_alg.VerifySignature(x_plaintext, x_signature));

// check the formatted signature support
// - create a hash code for the plaintext
HashAlgorithm x_hash_alg = HashAlgorithm.Create("SHA1");
byte[] x_hashcode = x_hash_alg.ComputeHash(x_plaintext);

// check the PKCS#1 signature formatting
ElGamalPKCS1SignatureFormatter x_sig_formatter
    = new ElGamalPKCS1SignatureFormatter( );
x_sig_formatter.SetHashAlgorithm("SHA1");
x_sig_formatter.SetKey(x_sign_alg);
x_signature = x_sig_formatter.CreateSignature(x_hashcode);

// verify the signature
ElGamalPKCS1SignatureDeformatter x_sig_deformatter
    = new ElGamalPKCS1SignatureDeformatter( );
x_sig_deformatter.SetHashAlgorithm("SHA1");
x_sig_deformatter.SetKey(x_verify_alg);

Console.WriteLine("PKCS#1 SIGNATURE: {0}",
    x_sig_deformatter.VerifySignature(x_hashcode, x_signature));
}

private static bool CompareArrays(byte[] p_arr1, byte[] p_arr2) {
    for (int i = 0; i < p_arr1.Length; i++) {
        if (p_arr1[i] != p_arr2[i]) {
            return false;
        }
    }
    return true;
}
}
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 17. Cryptographic Keys

In this chapter, we discuss how the .NET Framework supports the cryptographic keys that we have used in the previous chapters. We explain the importance of keys and the decisions you must make in deciding how you will create them. We provide examples of how to program the .NET support for cryptographic keys and finish our coverage of cryptography by extending our ElGamal implementation to support session key exchange formatting.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

17.1 Cryptographic Keys Explained

When you use cryptography, you simplify problems by relying on your ability to manage secret keys correctly; in essence, you exchange one problem (the need to communicate securely) for another (protecting the key), which you expect to be simpler. For example, when you use data encryption to keep messages confidential, you must protect your secret or private key, which should, in principle, be simpler than arranging to meet in person or using a secure courier service. You can simplify the problem of authentication by using digital signatures, but these are useful only as long as you are able to protect a private key; Bob can trust signatures from Alice for as long as it takes Eve to guess or otherwise acquire the private key.

The importance of managing your keys cannot be overstated; Alice and Bob must manage theirs carefully to protect encoded messages. Eve could undermine all of their efforts if she acquires the keys. We use the word "acquire," because as we discussed in [Chapter 12](#), Eve can do more than simply guessing the value of the key-Eve may try to steal the key, or attempt to deceive Alice or Bob into revealing their keys to her.

17.1.1 Understanding Key Complexity

You want to create keys that provide suitable protection against attack and yet are easy to use. Choosing a key that provides suitable protection means assessing the amount of time that the confidential data should remain secure and selecting a key length that would take at least this amount of time to find using brute force; we touched on how key lengths affect security in [Chapter 14](#) and [Chapter 15](#). In general, the longer the key, the longer it takes to check all of the possible key values and the more secure it is.

It might seem as though the best approach is for Alice and Bob to create keys that are as long as possible, to provide the greatest level of security. The problem is that they must remember their keys, and type them in when they want to use them; the security advantages of a 2048-bit RSA key are obvious, but it is almost impossible to remember. For example, the sequence below is the private key parameter **D** for a 2048-bit RSA key ([Chapter 15](#) contains full details of the RSA algorithm):

```
56 90 E2 5B F7 FA CB 85 0C CA F5 3B 0A 25 DC 07 F2 8D 32 4F 0D 4B EB 87 4C D1 E9
90 30 09 74 40 9B BE FA B3 CB C9 C1 5D C9 9D F9 B4 34 6F 27 CA 32 A0 0B FF 84 58
E9 E7 47 4C F9 E3 33 DD DA 3A CF AE 4D DA F8 DB C5 07 97 B6 42 26 84 E8 F9 07 3C
01 E3 D3 B7 C5 60 A9 8B 85 B2 1A 93 D1 74 27 64 4C 01 74 F7 5E 58 61 B9 C9 19 33
73 43 2D F3 0C 29 43 40 98 A4 85 15 FD 50 8C 8C 42 F1 19 01
```

Alice will be very frustrated if she has to remember this sequence (and the sequences for the other key parameters) and type it correctly before she can decrypt data or sign a document; the chances of her being able to correctly do this are slim. Alice will do what *all* users do when faced with this kind of memory problem-she will write down the password. Alice has now changed the nature of her security; the length of time it takes Eve to acquire the key is affected by how hard it is to steal Alice's notepad.

By asking Alice to do the impossible, we have changed the nature of her cryptographic security. As

soon as Alice writes down the key, we are dealing with a physical security problem. Even if Alice works in a "secure" building, Eve knows that it is easier to break into Alice's building and read her notes than it is to test every possible key value. We can make life easier for Alice and Bob by using very small key values, but while we reduce the chance of having the keys written down, we significantly increase the efficacy of brute force attacks.

We also need to consider the type of data that we use to create keys. The most secure way to create a key is to use a random data generator, which makes it harder for Eve to brute force the key value, because all key values are equally likely. The problem with random data is that it results in a sequence that is difficult to remember-people are not naturally able to remember an otherwise meaningless sequence of digits. Alice and Bob will start writing down their keys again if we ask them to use randomly generated keys.

If we ask Alice and Bob to select their own keys, we will find that they tend to pick words or phrases that have a personal significance. For example, Bob might use the name of his favorite sports team, the name of his cat, or his wife's birthday. The concern is that Eve will be able to acquire the key quickly by testing only key values derived from Bob's background and preferences. There are only a limited number of sports teams, and it does not take long to test them all.

The tension between the needs of the user and the needs to attain a certain level of security is one of the fundamental challenges you will face when designing systems that use cryptography. There is no simple answer to this problem, and the most common mistake we have seen is where software forces users to work with long and random keys, and the employer addresses this by making writing down keys a punishable offense. Users are incapable of remembering long data sequences, and outlawing the natural solution just forces users to write the keys down in secret.

One potential solution is to generate a sequence of words to create a key that is a nonsensical phrase on the basis that users will find this type of key easier to remember-for example, "Blue Jazz Fish Happy Green Walking Light Special." Though people *can* remember this kind of phrase, they can only do so for a short period and still end up writing down the phrase in the long term. Another problem is that we have made it easier to brute force the key by restricting the key to a set of English words; Eve can use a dictionary to test combinations of words and do so relatively quickly.

The most common technique is to rely on a shorter key to protect a long cryptographic key-for example, rely on a Windows log-on password to protect an RSA key. This approach has an obvious appeal, because the user has to remember only one thing, based on the assumption that it is harder to attack the user's computer than it is to guess the key, presumably because the computer, the network, and every other device connected to the network are completely secure. A quick glance through most IT security bulletins demonstrates why this is not a reasonable assumption to make.

17.1.2 Exchanging Symmetric Algorithm Keys

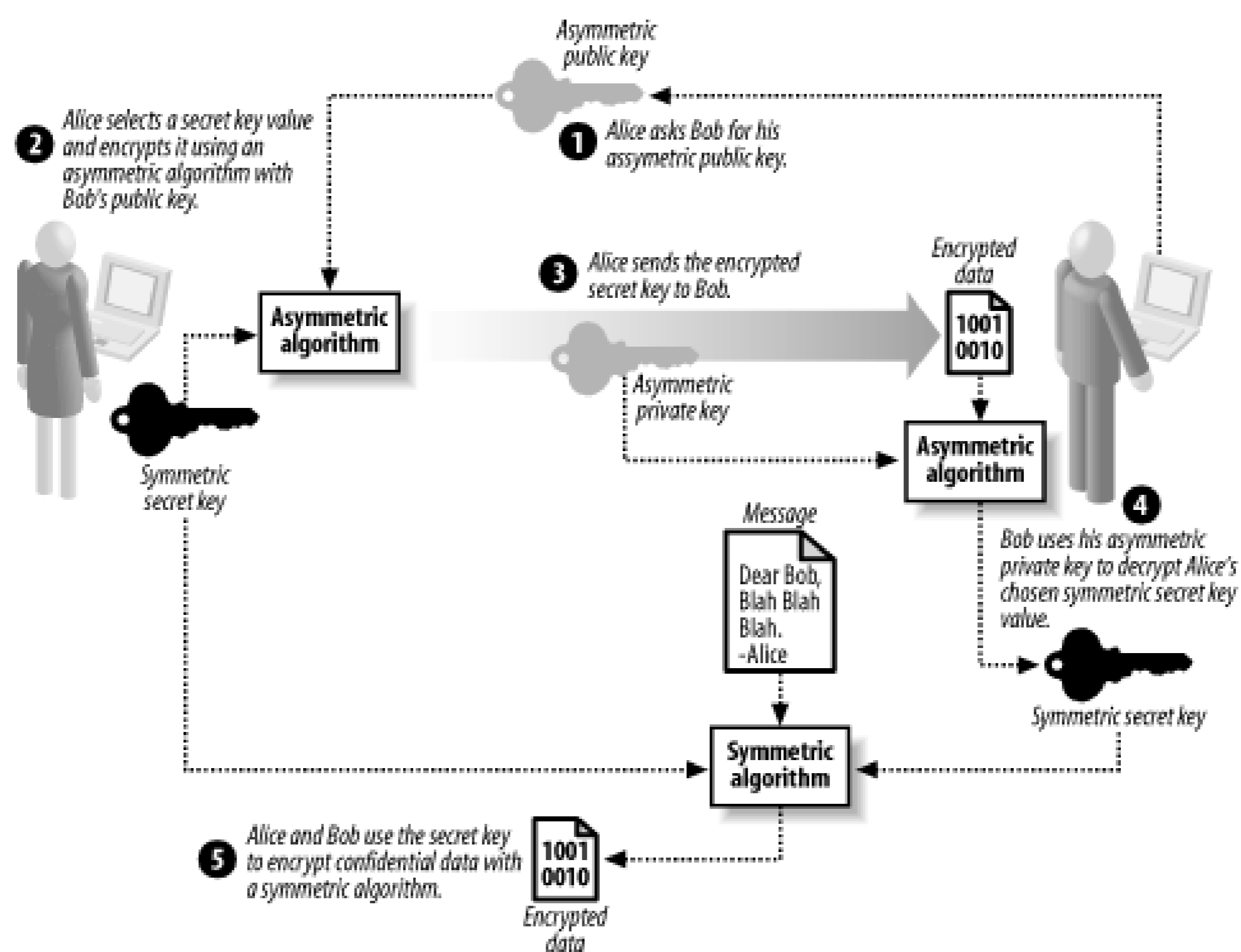
In [Chapter 14](#), we explained that agreeing on a key for a symmetrical algorithm was problematic-Alice and Bob need to meet in a secure location, where they can be sure that Eve cannot eavesdrop on their conversation. As an alternative, Alice and Bob can use a trusted and secure courier service to send keys to one another. Both of these approaches present Eve with an attractive line of attack-breaking the security of the "secure" location or bribing one of the "trusted" couriers could be easier than brute forcing the keys. Worse still, Alice and Bob should change their key regularly (as discussed in [Chapter 14](#)), presenting Eve with further opportunities.

In this section, we explain how Alice and Bob can use asymmetric algorithms to perform symmetric

key exchange, and exchange the problem of agreeing on symmetric keys for the simpler problem of using asymmetric keys. [Figure 17-1](#) illustrates the basic protocol for key exchange, which works as follows:

1. Alice asks Bob for his asymmetric public key.
2. Alice selects a symmetric secret key value.
3. Alice uses an asymmetric algorithm to encrypt the value with Bob's public key.
4. Alice sends the encrypted key value to Bob.
5. Bob uses his asymmetric private key to decrypt the key value.
6. Alice and Bob use a symmetric algorithm to encrypt data using the symmetric key value that Alice selected.

Figure 17-1. Exchanging symmetric keys using asymmetric encryption



At the end of this process, Alice and Bob have decided on a new symmetric secret key without the difficulty of arranging a secure meeting. Eve may intercept the key exchange messages between Alice and Bob, but she cannot decrypt the secret key value without Bob's asymmetric secret key. In essence, Alice and Bob have simplified their key agreement problem by relying on the security of Bob's asymmetric key pair, and so another aspect of cryptography becomes an issue of key security.

Using the described key exchange protocol, the exchange of symmetric keys becomes a simple and secure process. To improve security further, there is no reason why Alice and Bob should not create

new keys every time that they need to exchange confidential data; such keys are called *session keys*.

Alice and Bob could have encrypted all of their confidential data using an asymmetric encryption algorithm, but such algorithms process data very slowly compared to symmetric algorithms. This kind of key exchange makes the best use of both kinds of algorithm—symmetrical algorithms are fast and asymmetric algorithms are ideally suited to encrypting the small secret key values. Key exchange is also suitable for "impromptu" encryption, where someone who Bob has no prior relationship with wishes to send him confidential data. [Figure 17-2](#) illustrates this technique, which we summarize as follows:

1. Eugene, who has no prior relationship with Bob, selects a symmetric session key.
2. Eugene composes a confidential message and uses a symmetric algorithm to encrypt the message with the session key.
3. Eugene uses an asymmetric algorithm to encrypt the session key with Bob's public key.
4. Eugene sends the encrypted message data and the encrypted session key to Bob.

Figure 17-2. Anonymous encryption using key exchange techniques

Impromptu encryption allows Eugene to send confidential data to Bob that can be decrypted using Bob's secret key only, even though a symmetric algorithm has been used to encrypt the message data. Not only does this technique require no arrangement between Bob and Eugene, it also does not need Eugene to identify himself, allowing the impromptu encryption to be used anonymously.

Bob reads the message sent to him by Eugene by following this protocol:

1. Bob uses an asymmetric algorithm to decrypt the session key with his private key.
2. Bob uses a symmetric algorithm to decrypt the message with the session key.
3. Bob reads the decrypted message.

[\[Team LiB \]](#)

[Team LiB]

17.2 Programming Cryptographic Keys

In the following sections, we demonstrate how to use the .NET support for creating and managing keys. Some of these techniques are functionality equivalent, and when deciding between them for your projects, you must take into consideration the tension between the abilities of the user and the demands for data security; see Section 17.1.1 for details.

17.2.1 Creating Keys

In the following sections, we discuss three techniques for creating keys. Only one of these techniques presents the user with data that is easy to memorize. You must be pragmatic when deciding how to create new keys, and select a process that satisfies the security demands of your project *and* the practical demands of the users.

17.2.1.1 Using the algorithm classes

The simplest way to create keys is to use the functionality built into all of the .NET algorithm classes for both symmetric and asymmetric algorithms. The .NET classes create new keys as they are needed; if you attempt to perform any cryptographic operation and you have not explicitly specified the keys to use, then the .NET classes will create new keys automatically. The following statements demonstrate how to use this functionality to print out the key value for a symmetrical algorithm (for full details see Chapter 14):

```
# C#

// create an instance of the symmetric algorithm
SymmetricAlgorithm x_alg = SymmetricAlgorithm.Create("Rijndael");

// set the length of key that we want to create
x_alg.KeySize = 128;

// get the key value, which will cause the implementation
// class to create a new secret key
byte[] x_secret_key = x_alg.Key;

// print out the key
foreach (byte b in x_secret_key) {
    Console.WriteLine("{0:X2} ", b);
}

# Visual Basic .NET

' create an instance of the symmetric algorithm
Dim x_alg As SymmetricAlgorithm = SymmetricAlgorithm.Create("Rijndael")

' set the length of key that we want to create
```

```

x_alg.KeySize = 128

' get the key value, which will cause the implementation
' class to create a new secret key
Dim x_secret_key As Byte( ) = x_alg.Key

' print out the key
Dim b As Byte
For Each b In x_secret_key
    Console.WriteLine("{0:X2} ", b)
Next b

```

The following statements demonstrate how to create a new key and print out the value for an asymmetric algorithm; unlike most asymmetric functions, obtaining details of the key can be performed through the abstract `AsymmetricAlgorithm` class (for full details see Chapter 15):

```

# C#

// create an instance of the asymmetric algorithm
AsymmetricAlgorithm x_alg = AsymmetricAlgorithm.Create("RSA");

// set the length of key that we want to create
x_alg.KeySize = 1024;

// get the key value, which will cause the implementation
// class to create a new secret key
string x_key_pair = x_alg.ToXmlString(true);

// print out the key
Console.WriteLine(x_key_pair);

# Visual Basic .NET

' create an instance of the asymmetric algorithm
Dim x_alg As AsymmetricAlgorithm = AsymmetricAlgorithm.Create("RSA")

' set the length of key that we want to create
x_alg.KeySize = 1024

' get the key value, which will cause the implementation
' class to create a new secret key
Dim x_key_pair As String = x_alg.ToXmlString(True)

' print out the key
Console.WriteLine(x_key_pair)

```

The principal benefit of creating keys in this manner is that you do not have to have any prior knowledge about how the key should be created; the algorithm class is responsible for creating keys in accordance with the relevant specification. The principle drawback is that the .NET classes generate keys that users will find difficult to remember.



When using an algorithm class to create a new key, you must be confident that the implementation creates keys securely and there are no flaws in the implementation that might result in key values that are easily guessed.

17.2.1.2 Using a random number generator

The second approach to creating keys is to use a random number generator (RNG). The .NET Framework provides support classes for generating random data; Figure 17-3 illustrates the class hierarchy.

Figure 17-3. The .NET Framework class hierarchy for random number generators



You cannot generate random data to create a key pair for an asymmetric algorithm. You must rely on the implementation classes to create key pairs or follow the appropriate key generation protocol; see Chapter 15 for details. Asymmetric algorithms create keys using numbers that are mathematically related and cannot be created from random data.

The class hierarchy for random number generators follows the abstract class/implementation class model that is used for hashing algorithms and symmetrical algorithms. The .NET Framework includes one implementation class, which is a wrapper around the RNG functions of the native Windows Cryptography API. Table 17-1 details the public members of the abstract `RandomNumberGenerator` class.

Table 17-1. Members of the `RandomNumberGenerator` class

Member	Description
<code>Create</code>	Creates an instance of a <code>RandomNumberGenerator</code> implementation class by name or creates an instance of the default implementation class if no name is specified
<code>GetBytes</code>	Populates a byte array with a sequence of random values
<code>GetNonZeroBytes</code>	Populates a byte array with a sequence of random values, none of which will be 0

You must know how many bytes of random data to create to prepare a key for a given algorithm correctly. The following statements demonstrate how to use the random number generator to create a key for a symmetric algorithm:

```
# C#
```

```

// create an instance of the symmetric algorithm
SymmetricAlgorithm x_alg = SymmetricAlgorithm.Create("Rijndael");

// set the length of key that we want to create
x_alg.KeySize = 128;

// we need to create an array of 16 bytes (16 bytes is 128 bits)
byte[] x_key_data = new byte[16];

// create the RNG
RandomNumberGenerator x_rng = RandomNumberGenerator.Create( );

// use the RNG to populate the byte array with random data
x_rng.GetBytes(x_key_data);

// set the key value for the symmetrical algorithm
x_alg.Key = x_key_data;

# Visual Basic .NET

' create an instance of the symmetric algorithm
Dim x_alg As SymmetricAlgorithm = SymmetricAlgorithm.Create("Rijndael")

' set the length of key that we want to create
x_alg.KeySize = 128

' we need to create an array of 16 bytes (16 bytes is 128 bits)
Dim x_key_data(15) As Byte

' create the RNG
Dim x_rng As RandomNumberGenerator = RandomNumberGenerator.Create( )

' use the RNG to populate the byte array with random data
x_rng.GetBytes(x_key_data)

' set the key value for the symmetrical algorithm
x_alg.Key = x_key_data

```

17.2.1.3 Using a key-derivation protocol

The previous two approaches result in values that a user will find difficult to remember. Users tend to prefer key values that have some meaning, but keys made up of proper words do not make ideal cryptographic keys; they are more susceptible to brute force attacks, because not all key values are equally likely when we are restricted to alphanumeric values listed in a dictionary.

A key-derivation protocol is a compromise between the need to create keys that are difficult to guess and the need for users to remember the key values. This kind of protocol processes a password selected by the user to create a cryptographic key. The user remembers the password, and types it in (rather than a sequence of numeric values). The derivation protocol transforms the password into a cryptographic key. The .NET Framework supports one key-derivation protocol, which is based on the PKCS #5/PBKDF1 standard. We summarize the protocol, as follows:

1. Alice selects a password. The password can be made up of more than one word—for example, "Programming .NET Security" would be acceptable as a password; though the word "password" is used, "pass phrase" is more appropriate.
2. Alice generates eight bytes of random data, known as the *salt*. The salt value is not secret, and Alice can safely write it down. The benefit of adding random data to the password is that two users who select the same password will end up with different cryptographic keys.
3. Alice concatenates her password and the salt data together to form the data block.
4. Alice selects a hashing algorithm and creates a hash code for the data block.
5. For a specified number of iterations, Alice uses the hashing algorithm to create a new hash code using the previous hash code as the input. The first hash code is produced by hashing the data block. The second hash code is produced by hashing the first hash code. The third hash code is created by hashing the second hash code, and so on. It is common to perform 100 iterations.
6. Alice uses the first n bytes of the final hash code to create a cryptographic key of $8n$ bits. For example, if Alice selected the SHA-1 hashing algorithm, she will acquire a hash code that is 20 bytes long (because SHA-1 produces 160-bit hash codes). If Alice required a 128-bit key, she will use the first 16 bytes of the hash code value.

Key-derivation protocols are deterministic, meaning that they will always create the same cryptographic key when supplied with specific password and salt values. Keys derived from password: are suitable for symmetric algorithms, but asymmetric algorithms require you to follow the appropriate key-generation protocol to create new key pairs. Keys that are derived in this way are no as secure as those created from random data, and the password should be chosen so that it is difficult to guess.

Figure 17-4 illustrates the .NET Framework class hierarchy for derivation protocols. The `PasswordDerivedBytes` class implements the protocol we described; the members of this class are listed in Table 17-2.

Figure 17-4. The .NET Framework class hierarchy for byte derivation schemes

Table 17-2. PasswordDeriveBytes Members

Member	Description
Properties	
<code>HashName</code>	The name of the hashing algorithm to use; see Chapter 13 for details of the supported algorithms

Member	Description
<code>IterationCount</code>	The number of iterations to perform to create the final hash code
<code>Salt</code>	A byte array representing the salt data
Methods	
<code>GetBytes</code>	Creates the final hash code and returns a number of bytes, specified as an integer method argument

The following statements demonstrate how to use the `PasswordDeriveBytes` class to derive a key using "Programming .NET Security" as the password; the password and the salt value are specified in the class constructor:

```
# C#

// create the random salt value
byte[] x_salt = new byte[8];
RandomNumberGenerator x_rand = RandomNumberGenerator.Create( );
x_rand.GetBytes(x_salt);

// create the derivation protocol class
PasswordDeriveBytes x_pwd
    = new PasswordDeriveBytes("Programming .NET Security", x_salt);
// specify the number of iterations
x_pwd.IterationCount = 100;
// specify the hashing algorithm
x_pwd.HashName = "SHA1";

// create the key
byte[] x_key = x_pwd.GetBytes(16);

// write out the salt value
Console.Write("SALT: ");
foreach (byte b in x_salt) {
    Console.Write("{0:X2} ", b);
}
Console.WriteLine( );

// write out the key value
Console.Write("KEY: ");
foreach (byte b in x_key) {
    Console.Write("{0:X2} ", b);
}
Console.WriteLine( );

# Visual Basic .NET

' create the random salt value
Dim x_salt(7) As Byte
Dim x_rand As RandomNumberGenerator = RandomNumberGenerator.Create( )
x_rand.GetBytes(x_salt)
```

```

' create the derivation protocol class
Dim x_pwd As PasswordDeriveBytes _
= New PasswordDeriveBytes("Programming .NET Security", x_salt)

' specify the number of iterations
x_pwd.IterationCount = 100
' specify the hashing algorithm
x_pwd.HashName = "SHA1"

' create the key
Dim x_key( ) As Byte = x_pwd.GetBytes(16)

' write out the salt value
Dim b As Byte
Console.Write("SALT: ")
For Each b In x_salt
    Console.Write("{0:X2} ", b)
Next
Console.WriteLine( )

' write out the key value
Console.Write("KEY: ")
For Each b In x_key
    Console.Write("{0:X2} ", b)
Next
Console.WriteLine( )

```

17.2.2 Using Key Persistence

The implementation classes for the DSA and RSA algorithms are wrappers around native code contained in the Windows Crypto API. These classes expose a feature of this API that allows asymmetric key pairs to be stored persistently by the operating system; the user does not have to remember the key parameters, which are protected by the Windows account password.

Relying on the security of the Windows operating system to protect cryptographic key pairs is not suitable for all projects; projects that require high levels of security are likely to be affected by the relative ease with which a Windows password can be attacked. The full details of the Windows Crypto API are beyond the scope of this book, but in this section, we briefly demonstrate how to use the .NET classes to store key pairs persistently.

The `System.Security.Cryptography.CspParameters` class allows you to specify the details of how a key pair should be stored—there are various options available, but for your purposes, you require only a name to associate with the keys. The name relates to the *key store*, and you can use any name that suits your needs; the following statements create an instance of the `CspParameters` class and set the key store name to be *MyKeys*:

```

# C#

// create the parameters instance
CspParameters x_params = new CspParameters( );
// specify the container name

```

```
x_params.KeyContainerName = "MyKeys";
```

```
# Visual Basic .NET
```

```
' create the parameters instance
Dim x_params As CspParameters = New CspParameters( )
' specify the container name
x_params.KeyContainerName = "MyKeys"
```

Once you have created your `CspParameters` instance, you can use it as the argument to the constructor of the algorithm implementation class. The final step is to set the `PersistKeyInCsp` property to `true`, specifying that you want to store the algorithm's key pair persistently. The following statements demonstrate how to create an instance of the `RSACryptoServiceProvider` class using the `CspParameters` instance as an argument and set the `PersistKeyInCsp` property:

```
# C#
```

```
// create an instance of the crypto provider class with the csp parameters
RSACryptoServiceProvider x_rsa = new RSACryptoServiceProvider(x_params);
```

```
// enable key persistence
x_rsa.PersistKeyInCsp = true;
```

```
# Visual Basic .NET
```

```
' create an instance of the crypto provider class with the csp parameters
Dim x_rsa As RSACryptoServiceProvider = New RSACryptoServiceProvider(x_params)

' enable key persistence
x_rsa.PersistKeyInCsp = true
```

The asymmetric implementation classes will create a key pair automatically before performing any cryptographic operations or exporting the key parameters. When the key pair is created, it will be stored persistently and the user will not have to try and remember (or write down) the key details. This technique can be used only with classes that build on functionality of the Windows Crypto API; implementations from third-party companies may not provide this support.

17.2.3 Key Exchange Formatting

The .NET Framework includes classes specifically for exchanging session keys using asymmetric encryption. The process for encrypting a key value is as follows:

1. Create a new instance of the appropriate symmetric algorithm class and create a new session key.
2. Create a new instance of an asymmetric algorithm class and import the public key that you wish to use to encrypt the session key value.
3. Create a new instance of a key exchange formatter and specify the asymmetric class created in the previous step.

4. Encrypt the session key value using the key exchange formatter.

The formatter class is responsible for preparing the session key data prior to encryption with the asymmetric algorithm. Figure 17-5 illustrates the .NET class hierarchy for key exchange formatter classes. The .NET class library includes formatting classes for the RSA algorithm; the DSA algorithm is not suitable for session key exchange since it does not support data encryption.

Figure 17-5. The .NET Framework class hierarchy for key exchange formatting



Two formatting schemes are supported—the OAEP scheme is preferred. The second scheme, PKCS #1 v1.5, is included for compatibility with legacy systems. The formatting scheme is used to protect the session key value from specific attacks; we implement the OAEP scheme in the Section 17.3 of this chapter, but it is not essential that you understand the details of the formatting schemes available, only that you know how to use them. Table 17-3 summarizes the members of the abstract `AsymmetricExchangeFormatter` class.

Table 17-3. AsymmetricExchangeFormatter members

Member	Description
Properties	
<code>SetKey</code>	Sets the instance of the asymmetric algorithm implementation class to encrypt the session key
<code>CreateKeyExchange</code>	Encrypts a session key, expressed as an array of bytes
Methods	
<code>Parameters</code>	Returns the parameters of the key used to encrypt the session key

Example 17-1 statements demonstrate how to create and format a 128-bit Rijndael session key using the RSA algorithm and the OAEP formatting scheme:

Example 17-1. 128-bit Rijndael session key

```

# C#

// create an instance of the Rijndael algorithm and
// specify 128-bits as the key length
Rijndael x_rijndael = Rijndael.Create( );
x_rijndael.KeySize = 128;
  
```

```

// rely on the fact that the algorithm implementation
// class will create a new key to obtain the session value
byte[] x_session_key = x_rijndael.Key;

// create an instance of the RSA algorithm class
RSA x_rsa = new RSACryptoServiceProvider( );

//
// ... specify public key to encrypt with
//

// create a new instance of the RSA OAEP formatter
RSAAOEPKeyExchangeFormatter x_formatter
    = new RSAAOEPKeyExchangeFormatter( );

// specify the RSA instance we created as the one
// to use when encrypting the session key data
x_formatter.SetKey(x_rsa);

// encrypt the session key with the formatter
byte[] x_exchange_data = x_formatter.CreateKeyExchange(x_session_key);

// write out the encrypted data
foreach (byte b in x_exchange_data) {
    Console.WriteLine("{0:X2} ", b);
}

# Visual Basic .NET

' create an instance of the Rijndael algorithm and
' specify 128-bits as the key length
Dim x_rijndael As Rijndael = Rijndael.Create( )
x_rijndael.KeySize = 128

' rely on the fact that the algorithm implementation
' class will create a new key to obtain the session value
Dim x_session_key( ) As Byte = x_rijndael.Key

' create an instance of the RSA algorithm class
Dim x_rsa As RSA = New RSACryptoServiceProvider( )

'
' ... specify public key to encrypt with
'

' create a new instance of the RSA OAEP formatter
Dim x_formatter As RSAAOEPKeyExchangeFormatter = New RSAAOEPKeyExchangeFormatter( )
' specify the RSA instance we created as the one
' to use when encrypting the session key data
x_formatter.SetKey(x_rsa)

' encrypt the session key with the formatter

```

```

Dim x_exchange_data( ) As Byte = x_formatter.CreateKeyExchange(x_session_key)

' write out the encrypted data
Dim b As Byte
For Each b In x_exchange_data
    Console.WriteLine("{0:X2} ", b)
Next

```

The output below shows an example of the formatted exchange data; the data will change each time the statements are executed, because a new session key will have been created at random by the Rijndael class:

```

C5 3B 7F 1C 44 B9 DF 3C C6 7F 7E 18 3E E7 F9 15 2E 0C B3 2A EF EC 4C EF 45 1D 51
B8 CC D1 FB C8 7F 8B A8 4F 58 92 76 20 61 13 C1 A2 4E 06 50 9A E2 0E 97 34 8A C7
19 8C 21 59 67 30 3A 57 9A E3 B9 4C 5F 56 4F 10 54 1A 83 5B 45 1A 4F 39 A9 C4 64
C2 11 5C 82 6D E1 A9 F3 BD F3 79 87 EA 13 52 2B EF 5F 71 8B 82 08 F6 9D 57 88 43
42 AE 75 E4 DD B9 BD 52 0F DB CD 86 E7 D2 17 0F 2F DB BF A8

```

The output is sent to the person with whom you wish to exchange confidential messages. Notice that the encrypted session key is much longer than the 20 bytes that make up the 128-bit Rijndael session key; this is a result of the padding added to the session key during the formatting process; the encrypted data is always the same length to prevent revealing the length of the session key to eavesdroppers. Figure 17-6 illustrates the hierarchy for the classes that decrypt the exchange data and restore the session key value by removing the formatting.

Figure 17-6. The .NET Framework class hierarchy for key exchange deformation

The process for decrypting a key value is as follows:

1. Create a new instance of an asymmetric algorithm class and import the private key that you wish to use to decrypt the formatted exchange data.
2. Create a new instance of a key exchange deformatter and specify the asymmetric class created in the previous step.
3. Decrypt the formatted exchange data using the key exchange deformatter.
4. Create a new instance of the appropriate symmetric algorithm class and set the session key.

Table 17-4 summarizes the members of the abstract `AsymmetricExchangeDefformatter` class.

Table 17-4. AsymmetricExchangeDefformatter members

Member	Description
Properties	
<code>SetKey</code>	Sets the instance of the asymmetric algorithm implementation class to decrypt the exchange data
<code>DecryptKeyExchange</code>	Decrypts exchange data, expressed as an array of bytes
Methods	
<code>Parameters</code>	Returns the parameters of the key used to decrypt the exchange data

The following statements demonstrate how to decrypt the exchange data from the previous example to restore the session key:

```
# C#
```

```
byte[] x_exchange_data = new byte[] {0xC5, 0x3B, 0x7F, 0x1C, 0x44, 0xB9, 0xDF,
    0x3C, 0xC6, 0x7F, 0x7E, 0x18, 0x3E, 0xE7, 0xF9, 0x15, 0x2E, 0x0C, 0xB3,
    0x2A, 0xEF, 0xEC, 0x4C, 0xEF, 0x45, 0x1D, 0x51, 0xB8, 0xCC, 0xD1, 0xFB,
    0xC8, 0x7F, 0x8B, 0xA8, 0x4F, 0x58, 0x92, 0x76, 0x20, 0x61, 0x13, 0xC1,
    0xA2, 0x4E, 0x06, 0x50, 0x9A, 0xE2, 0x0E, 0x97, 0x34, 0x8A, 0xC7, 0x19,
    0x8C, 0x21, 0x59, 0x67, 0x30, 0x3A, 0x57, 0x9A, 0xE3, 0xB9, 0x4C, 0x5F,
    0x56, 0x4F, 0x10, 0x54, 0x1A, 0x83, 0x5B, 0x45, 0x1A, 0x4F, 0x39, 0xA9,
    0xC4, 0x64, 0xC2, 0x11, 0x5C, 0x82, 0x6D, 0xE1, 0xA9, 0xF3, 0xBD, 0xF3,
    0x79, 0x87, 0xEA, 0x13, 0x52, 0x2B, 0xEF, 0x5F, 0x71, 0x8B, 0x82, 0x08,
    0xF6, 0x9D, 0x57, 0x88, 0x43, 0x42, 0xAE, 0x75, 0xE4, 0xDD, 0xB9, 0xBD,
    0x52, 0x0F, 0xDB, 0xCD, 0x86, 0xE7, 0xD2, 0x17, 0x0F, 0x2F, 0xDB, 0xBF,
    0xA8};

// create an instance of the RSA algorithm class
RSA x_rsa = new RSACryptoServiceProvider( );

//
// ... specify private key to decrypt with
//

// create a new instance of the RSA OAEP deformatter
RSAOAEPKeyExchangeDeformatter x_deformatter
    = new RSAOAEPKeyExchangeDeformatter( );

// specify the RSA instance we created as the one
// to use when encrypting the session key data
x_deformatter.SetKey(x_rsa);

// decrypt the exchange data with the deformatter to
// obtain the Rijndael session key
byte[] x_session_key = x_deformatter.DecryptKeyExchange(x_exchange_data);

// create an instance of the Rijndael algorithm
Rijndael x_rijndael = Rijndael.Create( );
// set the session key
```

```
x_rijndael.Key = x_session_key;
```

```
# Visual Basic .NET
```

```
Dim x_exchange_data( ) As Byte = New Byte( ) {&HC5, &H3B, &H7F, &H1C, &H44, &HB9, _
&HDF, &H3C, &HC6, &H7F, &H7E, &H18, &H3E, &HE7, &HF9, &H15, &H2E, &HC, &HB3, _
&H2A, &HEF, &HEC, &H4C, &HEF, &H45, &H1D, &H51, &HB8, &HCC, &HD1, &HFB, _
&HC8, &H7F, &H8B, &HA8, &H4F, &H58, &H92, &H76, &H20, &H61, &H13, &HC1, _
&HA2, &H4E, &H6, &H50, &H9A, &HE2, &HE, &H97, &H34, &H8A, &HC7, &H19, _
&H8C, &H21, &H59, &H67, &H30, &H3A, &H57, &H9A, &HE3, &HB9, &H4C, &H5F, _
&H56, &H4F, &H10, &H54, &H1A, &H83, &H5B, &H45, &H1A, &H4F, &H39, &HA9, _
&HC4, &H64, &HC2, &H11, &H5C, &H82, &H6D, &HE1, &HA9, &HF3, &HBD, &HF3, _
&H79, &H87, &HEA, &H13, &H52, &H2B, &HEF, &H5F, &H71, &H8B, &H82, &H8, _
&HF6, &H9D, &H57, &H88, &H43, &H42, &HAE, &H75, &HE4, &HDD, &HB9, &HBD, _
&H52, &HF, &HDB, &HCD, &H86, &HE7, &HD2, &H17, &HF, &H2F, &HDB, &HBF, _
&HA8}
```

```
' create an instance of the RSA algorithm class
```

```
Dim x_rsa As RSA = New RSACryptoServiceProvider( )
```

```
'
```

```
' ... specify private key to decrypt with
```

```
'
```

```
' create a new instance of the RSA OAEP deformatter
```

```
Dim x_deformatter As RSAOAEPKeyExchangeDeformatter _
```

```
= New RSAOAEPKeyExchangeDeformatter( )
```

```
' specify the RSA instance we created as the one
```

```
' to use when encrypting the session key data
```

```
x_deformatter.SetKey(x_rsa)
```

```
' decrypt the exchange data with the deformatter to
```

```
' obtain the Rijndael session key
```

```
Dim x_session_key( ) As Byte = x_deformatter.DecryptKeyExchange(x_exchange_data)
```

```
' create an instance of the Rijndael algorithm
```

```
Dim x_rijndael As Rijndael = Rijndael.Create( )
```

```
' set the session key
```

```
x_rijndael.Key = x_session_key
```

In Chapter 15, we demonstrated that the RSA implementation class always prepares data by applying either OAEP or PKCS #1 v1.5 formatting, making its functionality equivalent to using the formatting classes discussed in this section. These classes define a *generic* approach to key exchange that can be applied to all implementation classes, including those provided by third parties. In the next section, we define the formatting classes for our ElGamal implementation, which does not format data during encryption automatically.

[Team LiB]

[\[Team LiB \]](#)

17.3 Extending the .NET Framework

In the following sections, we complete our implementation of the ElGamal algorithm by creating the key exchange formatter and deformatter classes. We have only provided a C# implementation of the ElGamal key exchange formatters. Like almost all encryption algorithms, ElGamal relies on mathematical operations that are not possible in Visual Basic .NET without creating additional support functions to compensate for the limited numeric support the language provides.

17.3.1 ElGamal Key Exchange Explained

Our implementation of the ElGamal encryption functions exposes the "raw" algorithm; that is, unlike the Microsoft RSA implementation, our `ElGamalManaged` class does not format data prior to encryption.

To provide support for session key exchange, we must provide classes that extend the `AsymmetricExchangeFormatter` and `AsymmetricExchangeDefformatter` classes that we discussed in the previous section. We have selected the OAEP formatting scheme for our key exchange; we have listed the complete code for our formatting classes, which naturally includes the details of the OAEP scheme. We do not discuss the details of OAEP, and you should refer to the PKCS #1 standard (which defines both the OAEP and PKCS #1 v1.5 formatting schemes) for details.

17.3.2 Defining the Key Exchange Formatter

We begin by defining `ElGamalOAEPKeyExchangeFormatter`, which is our formatter class, extending the `AsymmetricKeyExchangeFormatter` class that we discussed in the previous section:

```
using System;
using System.IO;
using System.Security.Cryptography;

public class ElGamalOAEPKeyExchangeFormatter : AsymmetricKeyExchangeFormatter {
    private ElGamalManaged          o_algorithm;
    private Random                   o_random;
    private PKCS1MaskGenerationMethod o_mask_generator;
}
```

The constructor creates an instance of the ElGamal implementation class, `ElGamalManaged`. As you see in the `SetKey` method, you can import the keys from any implementation class that extends the abstract `ElGamal` class using the common parameter import and export methods. The other instance variables are related to the OAEP formatting process:

```
public ElGamalOAEPKeyExchangeFormatter( ) {
    // create the instance of the algorithm
    o_algorithm = new ElGamalManaged( );
    // init the rnd
}
```

```

        o_random = new Random( );
        // init the mask generator
        o_mask_generator = new PKCS1MaskGenerationMethod( );
    }

```

The `Parameters` property returns the parameters of the public key that will be used to encrypt the session key; we create the result by using the `ToXmlString` method defined in the `AsymmetricAlgorithm` class (see [Chapter 15](#) for details):

```

    public override string Parameters {
        get {
            return o_algorithm.ToXmlString(false);
        }
    }

```

The `SetKey` method allows the user to specify an instance of the ElGamal algorithm that should be used for encryption; if the class is not an instance of our abstract `ElGamal` class, throw an exception; otherwise, import the public key parameters into the `ElGamal` instance created by the constructor:

```

    public override void SetKey(AsymmetricAlgorithm p_key) {
        // ensure that we are dealing with an ElGamal algorithm
        if (p_key is ElGamal) {
            // export the key and push it into the algorithm
            o_algorithm.ImportParameters(((ElGamal)p_key).ExportParameters(false));
        } else {
            // we can't continue because the algorithm
            // is the one for this class
            throw new ArgumentException("Key Algorithm is not ElGamal", "p_key");
        }
    }

```

The `CreateKeyExchange` method is responsible for formatting the session key and encrypting the result using the `ElGamal` implementation class; there are two overloaded versions of this method, but they are functionality equivalent, because the differentiating argument is not currently used by the .NET Framework:

```

    public override byte[] CreateKeyExchange(byte[] p_byte, Type p_type) {
        // the Type parameter is not currently supported by the
        // .NET Framework, so we just need to work with the byte array
        return CreateKeyExchange(p_byte);
    }

    public override byte[] CreateKeyExchange(byte[] p_byte) {
        // create the OAEP padded data
        byte[] x_padded_data = CreateOAEPpaddedData(p_byte);
        // create the ciphertext from the padded data
        byte[] x_ciphertext = o_algorithm.EncryptData(x_padded_data);
        // return the ciphertext
        return x_ciphertext;
    }

```

The remaining methods in this class are responsible for applying the OAEP formatting, which is outside the scope of this book:

```

// the lHash value which is the preamble to an OAEP block
private byte[] o_lhash
    = new BigInteger("da39a3ee5e6b4b0d3255bfef95601890afd80709", 16).getBytes( );

private byte[] CreateOAEPData(byte[] p_data) {
    // create a memory stream to hold the padded data
    MemoryStream x_stream = new MemoryStream( );

    // define K
    int x_K = o_algorithm.KeySize/8 -1;

    // determine the block size
    int x_max_bytes = x_K - (2 * o_lhash.Length) - 2;
    // determine how many complete blocks there are
    int x_complete_blocks = p_data.Length / x_max_bytes;

    // run through and process the complete blocks
    int i = 0;
    byte[] x_block;
    for (; i < x_complete_blocks; i++) {
        x_block = CreateSingleOAEPBlock(p_data, i * x_max_bytes,
            x_max_bytes, x_K);
        x_stream.Write(x_block, 0, x_block.Length);
    }

    // process any remaining data
    x_block = CreateSingleOAEPBlock(p_data, i * x_max_bytes,
        p_data.Length - (i * x_max_bytes), x_K);
    x_stream.Write(x_block, 0, x_block.Length);

    // return the padded data
    return x_stream.ToArray( );
}

private byte[] CreateSingleOAEPBlock(byte[] p_data, int p_offset,
    int p_count, int p_K) {

    // b.          Generate an octet string PS consisting of
    // k - mLen - 2hLen - 2 zero octets.
    // The length of PS may be zero.
    byte[] x_PS = new byte[p_K - p_count - (2*o_lhash.Length) -2];

    // c.          Concatenate lHash, PS, a single octet with
    // hexadecimal value 0x01, and the message M to form a data
    // block DB of length k - hLen - 1 octets as
    // DB = lHash || PS || 0x01 || M .
    byte[] x_DB = new byte[o_lhash.Length + x_PS.Length + 1 + p_count];
    Array.Copy(o_lhash, 0, x_DB, 0, o_lhash.Length);
    Array.Copy(x_PS, 0, x_DB, o_lhash.Length, x_PS.Length);
    x_DB[o_lhash.Length + x_PS.Length] = 0x01;
    Array.Copy(p_data, p_offset, x_DB,

```

```

        o_lhash.Length + x_PS.Length + 1, p_count);

// d.          Generate a random octet string seed of length hLen
BigInteger x_temp = new BigInteger( );
x_temp.genRandomBits(o_lhash.Length * 8, o_random);
byte[] x_seed = x_temp.getBytes( );

// e.          Let dbMask = MGF (seed, k - hLen - 1)
byte[] x_dbMask = o_mask_generator.GenerateMask(x_seed,
        p_K - o_lhash.Length - 1);

// f.          Let maskedDB = DB XOR dbMask.
byte[] x_maskedDB = new byte[x_DB.Length];
byte[] x_temp_arr =
        (new BigInteger(x_DB) ^ new BigInteger(x_dbMask)).getBytes( );
Array.Copy(x_temp_arr, 0, x_maskedDB,
        x_maskedDB.Length - x_temp_arr.Length, x_temp_arr.Length);

// g.          Let seedMask = MGF (maskedDB, hLen).
byte[] x_seedMask = o_mask_generator.GenerateMask(x_maskedDB,
        o_lhash.Length);

// h.          Let maskedSeed = seed XOR seedMask.
byte[] x_maskedSeed
        = (new BigInteger(x_seed) ^ new BigInteger(x_seedMask)).getBytes( );

// i. Concatenate a single octet with hexadecimal value 0x00, maskedSeed,
// and maskedDB to form an encoded message EM of length k octets as
// EM = 0x00 || maskedSeed || maskedDB
byte[] x_EM = new byte[1 + o_lhash.Length + x_DB.Length];
Array.Copy(x_maskedSeed, 0, x_EM, x_EM.Length - x_DB.Length
        - x_maskedSeed.Length, x_maskedSeed.Length);
Array.Copy(x_maskedDB, 0, x_EM, x_EM.Length - x_maskedDB.Length,
        x_maskedDB.Length);

// return the result
return x_EM;
    }
}

```

17.3.3 Defining the Key Exchange Deformatter

To complete the ElGamal implementation, define the `ElGamalOAEPKeyExchangeDefomatter` class, which decrypts the exchange data; this class extends the `AsymmetricKeyExchangeDefomatter` class:

```

using System.Security.Cryptography;
using System.IO;
using System;

public class ElGamalOAEPKeyExchangeDefomatter : AsymmetricKeyExchangeDefomatter {

```

```

private ElGamalManaged          o_algorithm;
private PKCS1MaskGenerationMethod o_mask_generator;

```

The constructor creates an instance of the ElGamal implementation class, `ElGamalManaged`. As you see in the `SetKey` method, you can import the keys from any implementation class that extends the abstract `ElGamal` class using the common parameter import and export methods. The other instance variables are related to the OAEP formatting process:

```

public ElGamalOAEPKeyExchangeDeformatter( ) {
    // create the instance of the algorithm
    o_algorithm = new ElGamalManaged( );
    // init the mask generator
    o_mask_generator = new PKCS1MaskGenerationMethod( );
}

public override void SetKey(AsymmetricAlgorithm p_key) {
    // ensure that we are dealing with an ElGamal algorithm
    if (p_key is ElGamal) {
        // export the key and push it into the algorithm
        o_algorithm.ImportParameters(((ElGamal)p_key).ExportParameters(true));
    } else {
        // we can't continue because the algorithm
        // is the one for this class
        throw new ArgumentException("Key Algorithm is not ElGamal", "p_key");
    }
}

```

The `Parameters` property returns the parameters of the private key that will be used to decrypt the exchange data; create the result by using the `ToXmlString` method defined in the `AsymmetricAlgorithm` class (see [Chapter 15](#) for details). Unlike the equivalent property in the formatter class, the deformatter implementation must also import the key to use through the `Parameters` property, which you do through the `FromXmlString` method:

```

public override string Parameters {
    get {
        return o_algorithm.ToXmlString(true);
    }
    set {
        o_algorithm.FromXmlString(value);
    }
}

public override byte[] DecryptKeyExchange(byte[] p_byte) {
    // decrypt the ciphertext
    byte[] x_padded = o_algorithm.DecryptData(p_byte);
    // remove the OAEP padding
    byte[] x_plaintext = RestoreOAEPpaddedData(x_padded);
    // return the ciphertext
    return x_plaintext;
}

```

The remaining methods in this class are responsible for removing the OAEP formatting, which is

outside the scope of this book:

```

// the lHash value which is the preamble to an OAEP block
private byte[] o_lhash
    = new BigInteger("da39a3ee5e6b4b0d3255bfef95601890afd80709", 16).getBytes( );

private byte[] RestoreOAEPData(byte[] p_data) {
    // create a memory stream to hold the padded data
    MemoryStream x_stream = new MemoryStream( );

    // define K
    int x_K = o_algorithm.KeySize/8 -1;

    // determine how many complete blocks there are
    int x_blocks = p_data.Length / x_K;

    // run through and process the blocks blocks
    byte[] x_block;
    for (int i = 0; i < x_blocks; i++) {
        x_block = RestoreSingleOAEPBlock(p_data, i * x_K, x_K, x_K);
        x_stream.Write(x_block, 0, x_block.Length);
    }

    // return the padded data
    return x_stream.ToArray( );
}

private byte[] RestoreSingleOAEPBlock(byte[] p_data, int p_offset,
    int p_count, int p_K) {

    // b. Separate the encoded message EM into a single octet Y,
    // an octet string maskedSeed of length hLen, and an octet
    // string maskedDB of length k - hLen - 1
    // as EM = Y || maskedSeed || maskedDB
    byte[] x_maskedSeed = new byte[o_lhash.Length];
    Array.Copy(p_data, p_offset + 1, x_maskedSeed, 0, o_lhash.Length);
    byte[] x_maskedDB = new byte[p_K - o_lhash.Length -1];
    Array.Copy(p_data, p_offset + 1 + o_lhash.Length, x_maskedDB,
        0, x_maskedDB.Length);

    // c.          Let seedMask = MGF (maskedDB, hLen).
    byte[] x_seedMask
        = o_mask_generator.GenerateMask(x_maskedDB, o_lhash.Length);

    // d.          Let seed = maskedSeed XOR seedMask
    byte[] x_seed = (new BigInteger(x_maskedSeed)
        ^ new BigInteger(x_seedMask)).getBytes( );

    // e.          Let dbMask = MGF (seed, k - hLen - 1).
    byte[] x_dbMask
        = o_mask_generator.GenerateMask(x_seed, p_K - o_lhash.Length -1);
}

```

```
// f.          Let DB = maskedDB XOR dbMask.
byte[] x_DB = (new BigInteger(x_maskedDB)
    ^ new BigInteger(x_dbMask)).getBytes( );

// g.          Separate DB into an octet string lHash' of length hLen, a
// (possibly empty) padding string PS consisting of octets with
// hexadecimal value 0x00, and a message M as
// DB = lHash' || PS || 0x01 || M .
// If there is no octet with hexadecimal value 0x01 to separate PS from M,
// if lHash does not equal lHash', or if Y is nonzero,
// output "decryption error" and stop.
for (int i = 0; i < o_lhash.Length; i++) {
    if (x_DB[i] != o_lhash[i]) {
        throw new CryptographicException("Decryption Error");
    }
}
if (p_data[0] != 0) {
    throw new CryptographicException("Decryption Error");
}
// find the index representing the start of M
int x_index = -1;
for (int i = o_lhash.Length; i < x_DB.Length; i++) {
    if (x_DB[i] == 0x01) {
        // the next index is the start of M
        x_index = i + 1;
        break;
    } else if (x_DB[i] != (byte)0x00) {
        throw new CryptographicException("Decryption Error");
    }
}
if (x_index == -1) {
    throw new CryptographicException("Decryption Error");
}

// extract and return M
byte[] x_message = new byte[x_DB.Length - x_index];
Array.Copy(x_DB, x_index, x_message, 0, x_message.Length);
return x_message;
}
}
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Part IV: .NET Application Frameworks

Discusses other aspects of .NET Framework security not specifically related to runtime security of cryptography. These include ASP.NET application security, integration with the security-related features of Enterprise Services (COM+), and the use of the Windows Event Log for recording security events:

[Chapter 18](#)

[Chapter 19](#)

[Chapter 20](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 18. ASP.NET Application Security

This chapter discusses the features you can use to increase the security of your ASP.NET applications. We discuss the overall issue of ASP.NET application security, and the mechanisms the .NET Framework provides to provide authentication, authorization, and impersonation services to ASP.NET applications. We then discuss the use of code-access security within ASP.NET hosted environments. We do not attempt to teach you about ASP.NET programming—a vast topic that is well beyond the scope of this book; we assume you already have experience developing with ASP.NET.

[\[Team LiB \]](#)

[[Team LiB](#)]

18.1 ASP.NET Security Explained

Web applications and XML web services developed using ASP.NET are the means through which many companies provide information and services to their customers and partners. The very nature of these services demands that they be accessible-usually located at the edge of the corporate network-and consequently reachable by anyone with an Internet connection. To be of value, these services must be available and reliable, meaning that authorized people can use them when they need to, without concern that the service will crash or behave unexpectedly. In addition, depending on the service and the type of data it processes, there may be a requirement (or possibly a legal obligation) for the service provider to ensure that the data produced by, or stored within, the service remains secure and secret.

Unfortunately, the reachability of ASP.NET applications, coupled with their propensity to represent useful, valuable, or at least mildly interesting functionality, means that ASP.NET applications are more likely to be the target of security attack than other applications that you develop. These attacks aim to bring down the service, change its functionality, or gain access to valuable data or services. In addition, given that ASP.NET applications often represent one of the few points of presence a company has on the Internet, attackers may try to use the ASP.NET application as a gateway through which to gain access to the corporate network behind, which usually contains even more valuable and useful information.

18.1.1 ASP.NET Security Overview

ASP.NET provides a framework that dramatically simplifies the development of web applications (including XML web services) and provides a rich set of commonly required services that programmers would otherwise have to implement themselves. These services include state management, caching, and device-independent UI generation. From a security perspective, ASP.NET provides services to support user authentication, authorization, and impersonation.

ASP.NET successfully abstracts the complex development issues associated with implementing many of its shared services, and exposes them through a mixture of simple object models, configuration settings, and declarative program statements. As such, implementing security in ASP.NET applications is more an issue of configuration than programming.

Fortunately, despite the efforts to provide a simplified development model, ASP.NET remains flexible. ASP.NET applications are .NET applications that run in the CLR. Programmers can still write ASP.NET applications in both C# and Visual Basic .NET and have access to the full capabilities of the .NET class library. This includes access to all of the security classes we discuss elsewhere in this book.

Because ASP.NET applications run in the CLR, they are constrained by the permissions granted to them by CAS, which we discussed in [Chapter 5](#) through [Chapter 9](#). Initially, it may seem odd to use CAS (usually aimed at controlling the action of mobile code) to control the actions of server-based applications. However, if you have a server that hosts ASP.NET applications produced by different companies, CAS will allow you to give each company the freedom to manage its own application, while ensuring that the code it uploads cannot affect the host server or the applications of other companies. In addition, should hackers manage to gain control of an application, their actions will be

constrained by the permissions assigned to the application by CAS, limiting the damage they can do.

18.1.2 ASP.NET Configuration Files

ASP.NET uses XML-based configuration files as a simple and convenient mechanism through which to configure elements of application operation. This includes important security settings, such as the user authentication mechanism to use and the code-access security policy to apply to an ASP.NET application. We discuss all of these security settings later in this chapter, but first it is important that you understand how ASP.NET configuration files work.

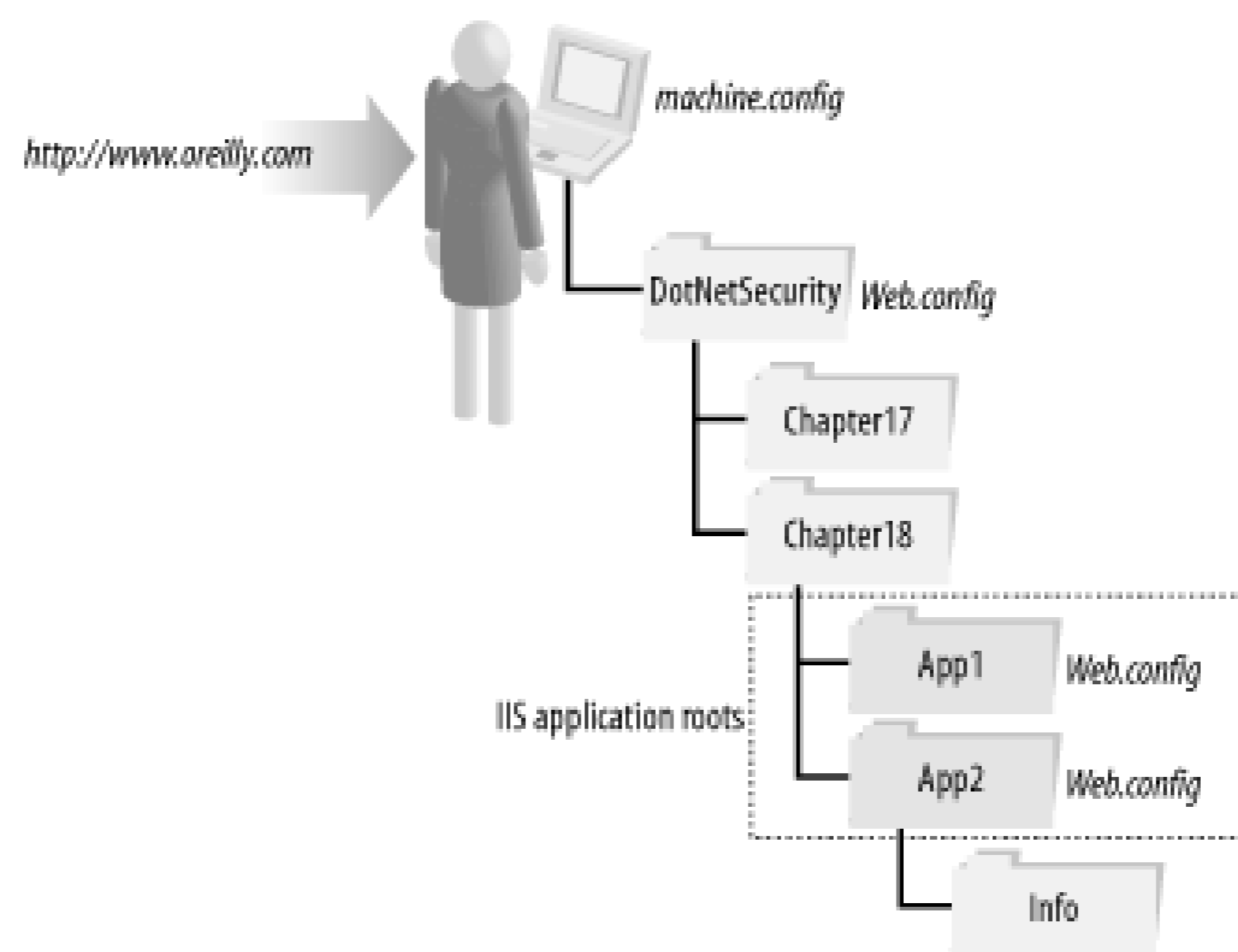
ASP.NET implements a hierarchically configured file structure. As with all .NET applications, the main source of configuration information is the *machine.config* file, which is located in the *CONFIG* folder of the .NET Framework installation directory. The *machine.config* file provides a central location in which you can implement configuration settings that apply to all ASP.NET applications running on the machine.

Each folder in an ASP.NET application's virtual path can contain a configuration file named *Web.config*, which provides configuration settings for the application resources located in that folder. Child folders inherit the configuration of their parent folder, meaning that configuration settings cascade down through an application's virtual folder hierarchy. However, child folders can also contain their own *Web.config*; the settings contained in this file override those configured higher up in the folder hierarchy.

The configuration file hierarchy is based on an application's virtual folder structure, not its physical folder structure.

To illustrate the behavior of configuration files, consider the folder hierarchy in [Figure 18-1](#), which shows a fictitious representation of the web site *http://www.oreilly.com*. The site contains a virtual folder named *DotNetSecurity* used to contain sample applications from a book. The samples are divided by chapter, and below that, each sample application is in its own folder (*App1*, *App2*, and so on); each application directory is configured as an IIS application root. Application roots play an important role in ASP.NET application configuration; though *Web.config* files can appear anywhere in the ASP.NET application folder hierarchy, there are many settings that cannot be applied below the level of an application root folder.

Figure 18-1. ASP.NET configuration file hierarchy



In [Figure 18-1](#), the *machine.config* file resides at the top of the configuration hierarchy, and its settings apply to all folders in the hierarchy below it unless they are overridden. The top-level *DotNetSecurity* folder contains a *Web.config* file, which will apply to all resources in all of the folders below it; this includes the *Chapter17* and *Chapter18* folders and all of their child folders. Both of the *App1* and *App2* folders have their own *Web.config* files. The *Info* folder contains no *Web.config* file and is subject to the configuration settings specified in the following configuration files:

- *machine.config*
- *DotNetSecurity/Web.config*
- *DotNetSecurity/Chapter18/App2/Web.config*

ASP.NET detects and implements changes made to *Web.config* files automatically; therefore there is no need to restart IIS or your computer in order for changes to take effect.

18.1.2.1 ASP.NET security-related configuration elements

ASP.NET configuration files contain settings that affect many aspects of the ASP.NET application. We list the most important security-related configuration elements in [Table 18-1](#) along with a brief description. Each of the configuration elements listed are children of the `<system.web>` element, except for the `<location>` element, which is a direct child of the root `<configuration>` element. The result is the structure shown in the following listing. For a complete list of all ASP.NET configuration elements, consult the .NET Framework SDK documentation.

```

<configuration>
  <location/>
  <system.web>
    <authentication/>
    <authorization/>
    <identity/>
    <machineKey/>
  
```

```

        <processModel/>
        <securityPolicy/>
        <trust/>
    </system.web>
</configuration>

```

Table 18-1. ASP.NET security-related configuration elements

Element	Description
<code><authentication></code>	Controls which mechanism ASP.NET uses to authenticate users who try to use the application. This element cannot be specified in <i>Web.config</i> files contained in folders below the level of application root. See Section 18.3 later in this chapter for details.
<code><authorization></code>	Controls the users and roles that can access application resources contained in the folder. This element can be configured at any level of the configuration hierarchy. See Section 18.4 later in this chapter for details.
<code><identity></code>	Controls whether an ASP.NET application process runs in the context of the ASP.NET worker process, an authenticated user, or a user specified in the configuration file. This element can be configured at any level of the configuration hierarchy. See Section 18.5 later in this chapter for details.
<code><location></code>	Allows a configuration file to target settings at a specific resource or folder and allows those settings to be locked so that they cannot be overridden by a <i>Web.config</i> file further down the folder hierarchy. This element can be configured at any level of the configuration hierarchy. We provide further details on the use of this element in Section 18.1.2.2 .
<code><machineKey></code>	Specifies the keys ASP.NET should use for the encryption and decryption of forms-authentication and view-state data. This element cannot be specified in <i>Web.config</i> files contained in folders below the level of an application root.
<code><processModel></code>	Controls many aspects of the behavior of the ASP.NET process, including the maximum number of work threads to use, thread-idle timeout, and memory limits. From a security perspective, this element allows you to configure the user account under which the ASP.NET worker process executes. This element can be configured only in the <i>machine.config</i> file. See Section 18.2 later in this chapter for details.
<code><securityPolicy></code>	Defines a mapping between named security levels (trust levels) and policy files. This element cannot be specified in <i>Web.config</i> files contained in folders below the level of an application root. See Section 18.6 later in this chapter for details.
<code><trust></code>	Configures the trust level assigned to the ASP.NET application. This element cannot be specified in <i>Web.config</i> files contained in folders below the level of an application root. See Section 18.6 later in this chapter for details.

18.1.2.2 Using `<location>` elements

Normally, the configuration settings contained in a *Web.config* file apply to all of the resources contained in the folder where the *Web.config* file is located, as well as all child folders. The `<location>` element allows you to target configuration settings at a specific folder or resource. Specify the target of the configuration settings in the `<location>` element's `path` attribute.

The `<location>` element is commonly used in conjunction with the `<authorization>` element to control access to individual resources (we discuss the specifics of the `<authorization>` element in [Section 18.4](#) later in this chapter). In the following *Web.config* located in the *DotNetSecurity* folder in [Figure 18-1](#), we specify that only members of the `Admins` role can access the *Chapter17* folder and only members of the `Developers` role can access the *Chapter18/App2/Main.aspx* file. Notice that a separate `<location>` element encapsulates each set of configuration settings.

The `allowOverride = "false"` attribute of the first `<location>` element means that the configuration settings cannot be overridden in the *Web.config* files of child folders. This is a critical feature in order to manage the security configuration of ASP.NET applications, especially in a hosted environment where different companies have the ability to upload their own ASP.NET applications and configuration files:

```
<configuration>
  <location path="Chapter17" allowOverride="false">
    <system.web>
      <authorization>
        <allow roles="Admins"/>
        <deny users="*" />
      </authorization>
    </system.web>
  </location>
  <location path="Chapter18/App2/Main.aspx">
    <system.web>
      <authorization>
        <allow roles="Developers"/>
        <deny users="*" />
      </authorization>
    </system.web>
  </location>
</configuration>
```

If you apply the `allowOverride = "false"` attribute setting to a `<location>` element that does not specify a `path`, it will completely disable the ability for *Web.config* files in child folders to override any configuration settings. For example:

```
<configuration>
  <location allowOverride="false" />
</configuration>
```

You can use the `<location>` element to great effect to centralize your application configuration information into a single file. For simple applications, this can result in a more understandable and manageable application configuration. However, we recommend doing this only if the number of different locations is few, and the configuration information is relatively straightforward. Otherwise, like all XML files, your configuration file can come become long and confusing, making it difficult to read and work with.

[\[Team LiB \]](#)

[[Team LiB](#)]

18.2 Configuring the ASP.NET Worker Process Identity

ASP.NET uses a pool of worker processes to handle ASP.NET application requests. By default, each worker process executes in the context of a special Windows account named ASPNET. The ASPNET account is created when the .NET Framework and ASP.NET are installed, and is assigned a limited set of Windows privileges and permissions. Running ASP.NET applications with reduced permissions ensures that the applications cannot access system resources that they should not and reduces the risks posed by an application whose security has been compromised. See the .NET Framework SDK documentation for specific details of the permissions and privileges assigned to the ASPNET account.



You can configure ASP.NET applications to run in the context of the authenticated user making the request. See [Section 18.5](#) later in this chapter for details.

You can change the account under which ASP.NET worker processes run by configuring the `userName` and `password` attributes of the `<processModel>` element, which is located in the `machine.config` file. This change affects all ASP.NET applications running on the machine; it is not possible to change the process identity for select applications.

Unlike changes to the `Web.config` files, ASP.NET does not detect and implement changes made to the `machine.config` file automatically. You must restart both the IIS Admin Service and the World Wide Web Publishing Service in order for changes to the `<processModel>` element to take effect. You can restart these services by running Control Panel Administration Tools Services.

ASP.NET provides two preconfigured values that you can assign to the `userName` attribute to control the worker process identity:

Machine

This is the default value and runs the ASP.NET application as the ASPNET account.

System

This value causes the ASP.NET worker process to run as the same account under which IIS is running; by default this will be the built-in Windows System account.

When using either of these settings, the value of the `password` attribute should be set to `AutoGenerate` to avoid the necessity of specifying the true password. For example, the following extract from the `machine.config` file configures the ASP.NET worker process to run as the same account used by IIS:

```
<configuration>
  <processModel userName="System" password="AutoGenerate" />
</configuration>
```

Alternately, you can specify the name of an account under which the worker process should run. The

`userName` attribute specifies the account name, and the `password` attribute contains the account's password in clear text. The following extract from the *machine.config* file configures the ASP.NET application process to run as the account "SomeUser," which has the password "secret":

```
<configuration>
  <processModel userName="SomeUser" password="secret" />
</configuration>
```

The obvious downside of this approach is the presence of the plaintext username and password in the configuration file. Although by default IIS and ASP.NET are configured not to serve *.config* files in response to client requests, it is never good practice to list such information in plaintext.

To avoid the need to store plaintext usernames and passwords in your configuration files, store encrypted versions of them in the Windows registry. In the configuration file, provide a reference to the registry keys where the encrypted information is located, as shown in the following example:

```
<configuration>
  <processModel userName="registry:HKLM\Software\OReilly\AspNet,Name"
    password="registry:HKLM\Software\OReilly\AspNet,Password" />
</configuration>
```

The `registry:` keyword signals to the configuration system that it should refer to the Windows registry to obtain the necessary username and password details. Following the colon is the registry key path, which must be part of the local machine (HKLM) hive. The final element (following the comma) gives the name of the registry value where the information is located.

Microsoft provides a utility named *Aspnet_setreg.exe* that encrypts the username and password values, and places them into the registry. Full details of how to obtain and use this tool are available from MSDN by searching for Microsoft Knowledge Base Article number 329290.

[\[Team LiB \]](#)

[[Team LiB](#)]

18.3 Authentication

ASP.NET supports four authentication modes: None, Windows, Forms, and Passport. Enable these authentication modes using a combination of IIS administration, configuration settings, and code. Configure different authentication modes for different ASP.NET applications using settings in your application's *Web.config* file; however, you cannot configure authentication modes below the level of an application root.

Understand that before a client request reaches your ASP.NET application, it must pass through IIS authentication. The result of IIS authentication provides the input that drives the .NET authentication mechanism. For the Windows authentication of ASP.NET, IIS authentication provides your application with the information about the authenticated Windows user. However, for the None, Forms, and Passport authentication modes, it is most likely that you will want to turn off IIS authentication by enabling anonymous access to IIS.

Unless you enable impersonation (discussed in [Section 18.5](#) later in this chapter), ASP.NET authentication has no effect on the Windows user context under which your ASP.NET application executes. Your application executes in the context of the ASP.NET worker process identity, which we discussed in the previous section. ASP.NET authentication controls the identity and principal assigned to your ASP.NET application. We introduced the concepts of identities and principals in [Chapter 10](#) and showed how to use them to enforce role-based security.

In the following sections, we discuss each of the ASP.NET authentication modes, but first we summarize the IIS authentication modes, which you must configure correctly to make ASP.NET authentication and impersonation function.

18.3.1 Configuring IIS Authentication Modes

IIS supports a variety of authentication modes, which are configured through Internet Services Manager. IIS allows you to configure authentication down to the individual file level, and enable multiple authentication mechanisms simultaneously.

Enabling anonymous authentication means that IIS will not force a client to authenticate regardless of whether other authentication mechanisms are enabled. This is often a source of confusion and unexpected behavior. For any other form of IIS authentication to work, you must disable anonymous authentication.

To configure the IIS authentication mode for a folder or file, you must select it in Internet Service Manager, open its properties window, and select the Directory Security tab. From the Anonymous access and authentication control group on this tab, click the Edit button to open the Authentication Methods window shown in [Figure 18-2](#). [Figure 18-2](#) shows the Authentication Methods window as it appears for IIS Version 5.1 running on Windows XP Professional.

Figure 18-2. Configuring IIS authentication modes



The Authentication Methods window allows you to configure four of the five authentication mechanisms supported by IIS. Except for anonymous access, each method depends on a client application providing credentials that IIS can map to a valid Windows user account. If the credentials do not match a valid Windows account, IIS rejects the client request. This dependency on Windows user accounts is the key factor you must consider when deciding to implement IIS authentication and ASP.NET Windows authentication (which we discuss later in this section). For large-scale, web-based, and cross-platform projects, depending on a Windows user accounts database may be inappropriate. We summarize each of the IIS authentication mechanisms here:

Anonymous access

Anonymous access is more an absence of authentication than an authentication mechanism. IIS accepts requests from any user and does not try to authenticate them. The User name and Password boxes allow you to specify the credentials of a Windows account under which the ASP.NET worker process runs if impersonation is enabled, which we discuss in [Section 18.5](#) later in this chapter.

Basic authentication

When a client requests a resource protected by Basic authentication, IIS challenges the client to provide a username and password. The client application responds by sending the username and password details across the network in clear text, meaning that Basic authentication is inappropriate for secure applications unless you use Secure Sockets Layer (SSL) or Transport Layer Security (TLS) to provide a secure connection between your server and the client.

Digest authentication

Digest authentication operates in the same way as Basic authentication, but the client passes a hash of the user's password across the network instead of the clear text password. This makes Digest authentication more secure than Basic authentication but requires the IIS Server to be a member of a Windows domain that uses Active Directory to contain its user accounts. Only

Active Directory allows IIS to access a user's password in such a way as to allow IIS to verify the hashed password provided by the client.

Integrated Windows authentication

When IIS challenges a client to provide user credentials, the client application does not prompt the user for account information; instead, it extracts the necessary details from the operating system based on the currently logged-on user. Credentials are communicated using Windows authentication protocols, which ensure that passwords are not sent across the network.

However, the dependence on client-side Windows account information and Windows authentication protocols limits the clients with which this type of authentication can be used.

Client Certificate authentication

Client Certificate authentication requires every client to have an X509 digital certificate. Each certificate maps to a specific Windows user account. When a client makes a request, it contains the client's certificate. IIS validates the certificate and determines which Windows user is making the request. Unlike the other authentication mechanisms listed here, you do not configure Client Certificate authentication using the interface shown in [Figure 18-2](#). Compared to the other authentication mechanisms, Client Certificate authentication requires significant configuration and management to implement correctly; consult the IIS documentation for details on how to configure Client Certificate authentication.

18.3.2 No Authentication

If you are not concerned about authenticating users, or intend to use some custom authentication mechanism implemented within your ASP.NET application, you can turn off ASP.NET authentication with the following configuration file setting:

```
<configuration>
  <system.web>
    <authentication mode="None">
  </system.web>
</configuration>
```

Even if you are not concerned about using ASP.NET authentication mechanisms, you can still enable IIS authentication so that you can allow only permitted Windows users to access your application. To allow truly unrestricted access to your application, enable anonymous access to IIS.

18.3.3 Windows Authentication

Windows authentication relies on IIS to authenticate any inbound requests using one of its built-in Basic, Digest, Integrated, or Client Certificate authentication mechanisms before the request reaches your ASP.NET application. As long as IIS anonymous authentication is not enabled, IIS will ensure that the requesting user has a valid Windows account and pass the authenticated user's information to the ASP.NET application.

With Windows authentication enabled, the ASP.NET application principal is set to an instance of `System.Security.Principal.WindowsPrincipal` configured to represent the authenticated Windows user. See [Chapter 10](#) for details on how you can use the `WindowsPrincipal` object to enforce role-based security within your ASP.NET application.

To enable Windows authentication, configure the `<authentication>` element as shown in the following example:

```
<configuration>
  <system.web>
    <authentication mode="Windows">
  </system.web>
</configuration>
```

18.3.4 Forms Authentication

Windows authentication requires users of your ASP.NET application to have Windows user accounts. As we have mentioned, for large web-based and cross-platform applications, this may not be feasible or desirable. Forms authentication allows you to implement a custom authentication mechanism where you can authenticate users against any type of authority.



To use Forms authentication, you should enable IIS anonymous access. If you do not have anonymous access enabled, IIS will force all clients to authenticate using whatever mechanism is enabled prior to running the Forms authentication mechanism; this is not normally the desired behavior.

Forms authentication relies on URL authorization to trigger the authentication process; we discuss URL authorization in [Section 18.4](#) later in this chapter. For now, it is enough to understand that authorization controls the application resources an authenticated user is permitted to access. The Forms authentication process works as follows:

1. The client requests a resource from your ASP.NET application for which you have configured authorization checks—for example, the *Main.aspx* file.
2. ASP.NET checks for an authentication cookie in the client request. If no cookie is found, the client is redirected to an HTML page where they can enter their credentials—usually a username and password.
3. The user submits her credentials to your ASP.NET application, and you authenticate them against whatever authority you have decided to use—commonly a database or directory server.
4. If the user's credentials are valid, ASP.NET creates an authentication cookie and sends it back to the client along with a redirection to the resource that she originally requested.
5. In future requests, the authentication cookie will exist, and so ASP.NET will not redirect the client to the authentication page, as long as the cookie has not expired.

Although ASP.NET provides much of the infrastructure and a number of utility classes to simplify the Forms authentication process, much of the implementation comes down to you. In the following sections, we implement a simple Forms authentication mechanism to demonstrate the steps required. The example consists of three files: *Web.config*, *Logon.aspx*, and *Main.aspx*.

To run the application, place each of these files in a folder accessible through IIS, and configure the folder as an application root using Internet Service Manager. In the two *.aspx* pages, there are only small sections of code, which we include inline instead of in code-behind files. In any but the most

trivial of ASP.NET application, we recommend that you use code-behind modules to improve the readability and maintainability of your application.



During the Forms authentication process, a user's credentials are sent over the network in clear text. You must encrypt the underlying network communications using SSL or TLS if you want to protect the user's credentials during authentication.

18.3.4.1 Configuring Forms authentication

The first step in implementing Forms authentication is to configure your ASP.NET application. The following *Web.config* file demonstrates each of the configuration elements that affect Forms authentication:

```
<configuration>
  <system.web>
    <authentication mode="Forms">
      <forms name="ProgDotNetSecurity" loginUrl="Logon.aspx" protection="All"
        timeout="180" path="/" requireSSL="false" slidingExpiration="true">
        <credentials passwordFormat="Clear">
          <user name="Alice" password="secret1"/>
          <user name="Bob" password="secret2"/>
          <user name="Eve" password="secret3"/>
        </credentials>
      </forms>
    </authentication>
    <authorization>
      <deny users="?"/>
    </authorization>
  </system.web>
</configuration>
```

Most essential is the `<authentication mode="Forms">` element, which is the main switch to tell ASP.NET that you want to use Forms authentication. The child `<forms>` element configures some aspects of the Forms authentication process and the structure of the authentication cookie that ASP.NET will create. The most important attribute is `loginUrl`, which specifies the URL to where ASP.NET will redirect an unauthenticated client request. We specify the *Logon.aspx* file, which will reside in the same folder as this *Web.config* file and the rest of our sample application.

The `<credentials>` element and its child `<user>` elements provide a static authority against which you can authenticate named users. In our example, we have defined three user accounts: Alice, Bob, and Eve. Passwords for each account can be stored in clear text, as we have done in the example, or hashed using either SHA-1 or MD5; the `passwordFormat` attribute of the `<credentials>` element specifies the password format. Use of the `<credentials>` element is optional; though it is easy to use and useful for testing purposes, it rarely provides a suitable solution for a secure production system.

[Table 18-2](#) lists the attributes of the `<forms>`, `<credentials>`, and `<user>` elements and describes their purpose.

Table 18-2. Attributes of the forms, credentials, and user elements

Attribute	Description
<forms> element attributes	
<code>name</code>	The name of the cookie your application uses for ASP.NET Forms authentication. Defaults to <code>.ASPXAUTH</code> .
<code>loginUrl</code>	If no authentication cookie is found in a client request, the client is redirected to the specified URL for authentication.
<code>protection</code>	Specifies the type of encryption to use for cookies. Possible values are <code>All</code> , <code>None</code> , <code>Encryption</code> , and <code>Validation</code> .
<code>timeout</code>	An integer value that specifies the number of minutes after creation at which the authentication cookie expires. If <code>slidingExpiration</code> is <code>true</code> , <code>timeout</code> specifies the number of minutes from when the cookie was last used before it expires. The <code>timeout</code> value affects only session cookies; persistent cookies do not time out.
<code>path</code>	The path for the cookie. It is best to leave this as its default value (<code>/</code>) to avoid the problem of browsers not correctly returning authentication cookies due to case-mismatch issues.
<code>requireSSL</code>	Specifies whether a secure connection is required to transmit the authentication cookie. Valid values are <code>true</code> and <code>false</code> .
<code>slidingExpiration</code>	Specifies whether sliding expiration is enabled, which resets an authentication cookie's expiration time each time it is used. Valid values are <code>true</code> and <code>false</code> .
<credentials> element attributes	
<code>passwordFormat</code>	Specifies the algorithm used to generate the hashed passwords listed for any contained <code><user></code> elements. Possible values are <code>Clear</code> , <code>MD5</code> , and <code>SHA1</code> . <code>Clear</code> means that passwords are listed in clear text; <code>MD5</code> and <code>SHA1</code> identify the specific hashing algorithm used to encrypt the password. For a complete description of hashing algorithms, see Chapter 13 .
<user> element attributes	
<code>name</code>	The user's name.
<code>password</code>	The user's password encoded using the mechanism specified in the <code>passwordFormat</code> attribute of the enclosing <code><credentials></code> element. It is important to understand that passwords encoded using the SHA-1 or MD5 hashing algorithms cannot be converted back to their original clear-text password. For a complete description of hashing algorithms, see Chapter 13 .

Also critical to the Forms authentication configuration is the `<authorization>` element, which lists

the application resources that require authorization to access. The need for authorization is what triggers ASP.NET to enforce Forms authentication. If a client tries to access an application resource (*SomeFile.aspx*), for example, that does not require authorization; the client will not be required to have an authentication cookie. We discuss the syntax of the `<authorization>` element in [Section 18.4](#) later in this chapter. For now, it is enough to know that the configuration shown here means that all anonymous users are denied access to all files; this requires all client request to have an authentication cookie.

18.3.4.2 Creating the logon page

Following the configuration of our ASP.NET application, the next step is to create a login page. This is the page where ASP.NET redirects all unauthenticated user requests. Remember that client requests containing a valid authentication cookie are routed directly to the requested application resource; only those without an authentication cookie are redirected to the login page.

In our *Web.config* file in the previous section, we used the `loginUrl` attribute of the `<forms>` elements to specify that the name of our login page is *Logon.aspx*, and that it is located in the application directory; *Logon.aspx* displays the interface shown in [Figure 18-3](#).

Figure 18-3. Forms authentication logon page

The following code implements the *Logon.aspx* page; most of the code is responsible for drawing the static interface elements. After the user enters his credentials and clicks on Logon, the `Logon_Click` method executes. We have highlighted the two lines that perform the most crucial elements of the Forms authentication:

```
# C#

<html>

    <script language="C#" runat="server">
        void Logon_Click(Object sender, EventArgs e) {

            // Perform authentication of the user credentials.
            // Use the static FormsAuthentication.Authenticate method
```



```

// to authenticate the credentials against those contained
// in the <credentials> element of Web.config.
if (FormsAuthentication.Authenticate(Name.Text,Password.Text)) {

    // Credentials OK, redirect to originally requested page.
    FormsAuthentication.RedirectFromLoginPage(Name.Text,true);
}
}
</script>

<body>
<h2>Forms Authentication Example - Logon Page</h2>
<form id="LogonForm" method="post" runat="server">
  <table>
    <tr align="right">
      <td>User Name : </td>
      <td><asp:TextBox id="Name" runat="server"/></td>
    </tr>
    <tr align="right">
      <td>Password : </td>
      <td><asp:TextBox id="Password" runat="server"/></td>
    </tr>
    <tr align="center">
      <td colspan="2">
        <asp:Button id="Logon" runat="server" Text="Logon"
          OnClick="Logon_Click"/>
      </td>
    </tr>
  </table>
</form>
</body>
</html>

# Visual Basic .NET

<html>

<script language="vb" runat="server">
  Sub Logon_Click(ByVal sender As Object, ByVal e As EventArgs)

    ' Perform authentication of the user credentials.
    ' Use the static FormsAuthentication.Authenticate method
    ' to authenticate the credentials against those contained
    ' in the <credentials> element of Web.config.
    If FormsAuthentication.Authenticate(Name.Text,Password.Text) Then

      ' Credentials OK, redirect to originally requested page.
      FormsAuthentication.RedirectFromLoginPage(Name.Text, True)
    End If
  End Sub
</script>

```

```

<body>
  <h2>Forms Authentication Example - Logon Page</h2>
  <form id="LogonForm" method="post" runat="server">
    <table>
      <tr align="right">
        <td>User Name : </td>
        <td><asp:TextBox id="Name" runat="server"/></td>
      </tr>
      <tr align="right">
        <td>Password : </td>
        <td><asp:TextBox id="Password" runat="server"/></td>
      </tr>
      <tr align="center">
        <td colspan="2">
          <asp:Button id="Logon" runat="server" Text="Logon"
            OnClick="Logon_Click"/>
        </td>
      </tr>
    </table>
  </form>
</body>
</html>

```

The first highlighted line calls the static `FormsAuthentication.Authenticate` method, which looks at the users you have defined in the `<credentials>` element of our *Web.config* file. If the supplied username and password match one of the listed user accounts, the `Authenticate` method returns `true`, and you should authenticate the user. Note that you could use any mechanism at this point to authenticate the user; we have used the *Web.config* `<credentials>` element for simplicity. It would be more common to reference a database or directory to authenticate a user in most production systems.

The `System.Web.Security.FormsAuthentication` class provides a set of static helper members for use during the processing of Forms logon requests and the management of authentication cookies. Refer to the .NET Framework documentation for a complete description of available functionality.

The second highlighted line calls the static `FormsAuthentication.RedirectFromLoginPage` method once you have authenticated the user's credentials successfully. The first argument to the `RedirectFromLoginPage` method is the name of the authenticated user. This is stored in the authentication cookie.

On future client requests, ASP.NET extracts the username from the authentication cookie and creates a `System.Web.Security.FormsIdentity` object from it; `FormsIdentity` implements the `System.Security.Principal.IIdentity` interface. ASP.NET places the `FormsIdentity` inside a `System.Security.Principal.GenericPrincipal` object, which it assigns to the ASP.NET process.

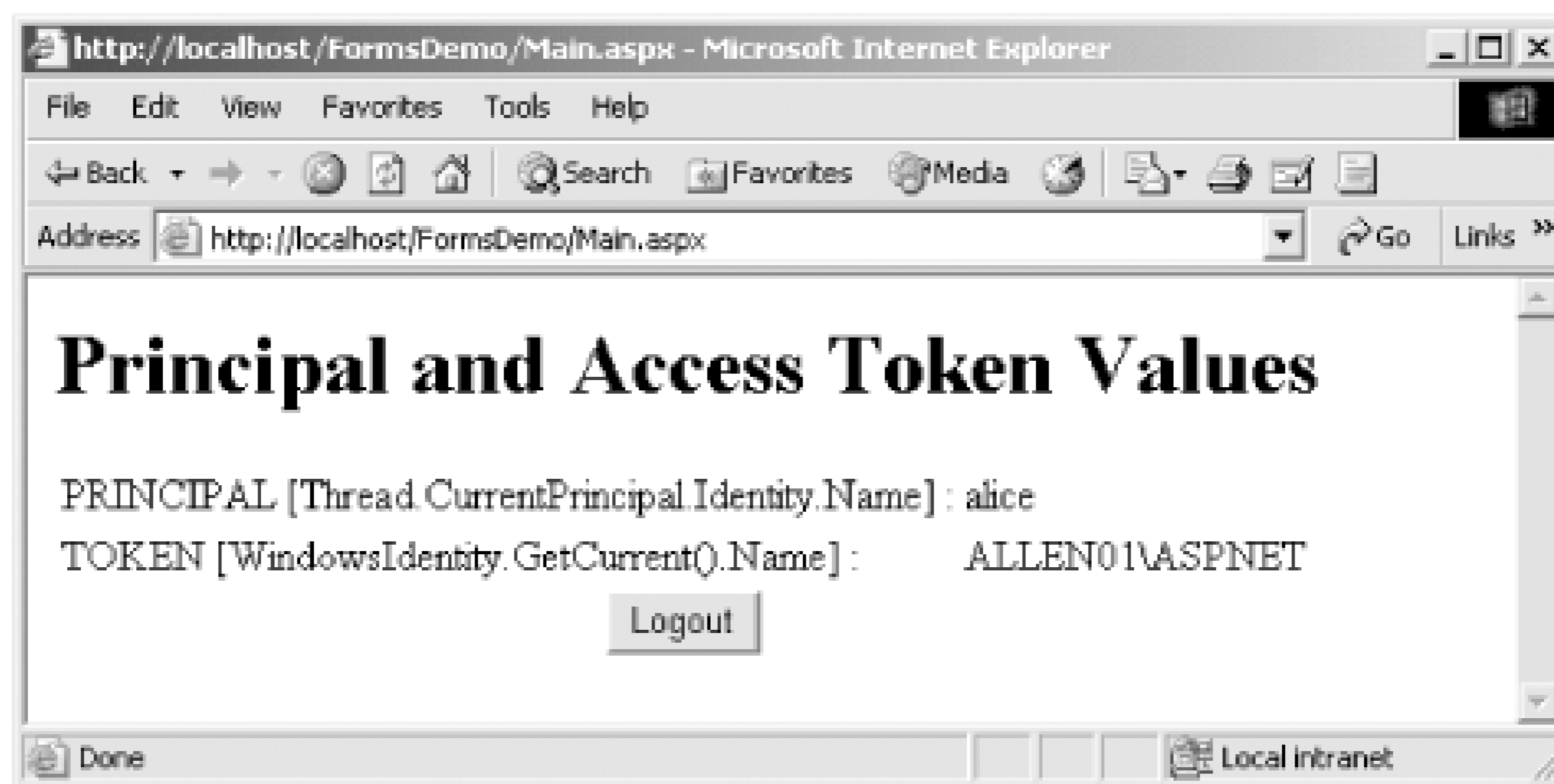
The second argument to the `RedirectFromLoginPage` method is a Boolean, which specifies whether the authentication cookie should be persisted or remain valid only for the session. In this instance, we have specified `true` in order to persist the cookie.

18.3.4.3 Creating the protected page

The *Web.config* and *Logon.aspx* files are all that is required to implement simple Forms authentication. With the configuration specified, any request for an ASP.NET application resource in the application must contain a valid authentication cookie, or ASP.NET will redirect the request to the *Logon.aspx* page.

To complete our example, we create a page named *Main.aspx*. The *Main.aspx* page presents the user with the interface shown in [Figure 18-4](#). It shows the identity represented by the current thread's principal (accessed using `System.Threading.Thread.CurrentPrincipal.Identity.Name`) and the Windows account under which the ASP.NET worker process is executing (accessed through `System.Security.Principal.WindowsIdentity.GetCurrent().Name`).

Figure 18-4. Forms authentication protected page



[Example 18-1](#) contains the code for *Main.aspx*. Most of the code implements the static user interface elements or displays the user identity details when the page is first loaded. However, we have included a Logout button to demonstrate how to terminate a client session:

Example 18-1. Main.aspx

```
# C#

<html>

  <script language="C#" runat="server">
    void Page_Load(Object sender, EventArgs e) {

      // Initialize the Label values to show the Identity represented
      // by the current thread's Principal and Windows access token.
      Principal.Text =
        System.Threading.Thread.CurrentPrincipal.Identity.Name;
      Token.Text =
        System.Security.Principal.WindowsIdentity.GetCurrent( ).Name;
    }
  }

```

```

void Logout_Click(Object sender, EventArgs e) {

    // Remove the cookie
    FormsAuthentication.SignOut( );

    // Redirect back to logon page
    Response.Redirect("Main.aspx");
}
</script>

<body>
<h1>Principal and Access Token Values</h1>
<form id="Form1" method="post" runat="server">
    <table>
        <tr>
            <td>PRINCIPAL [Thread.CurrentPrincipal.Identity.Name] :</td>
            <td><asp:Label id="Principal" runat="server"/></td>
        </tr>
        <tr>
            <td>TOKEN [WindowsIdentity.GetCurrent( ).Name] :</asp:Label></td>
            <td><asp:Label id="Token" runat="server"/></td>
        </tr>
        <tr align="center">
            <td colspan="2">
                <asp:Button id="Logout" runat="server" Text="Logout "
                    OnClick="Logout_Click"/>
            </td>
        </tr>
    </table>
</form>
</body>
</html>

```

```
# Visual Basic .NET
```

```
<html>
```

```

<script language="vb" runat="server">
    Sub Page_Load(ByVal sender As Object, ByVal e As EventArgs)

        ' Initialize the Label values to show the Identity represented
        ' by the current thread's Principal and Windows access token.
        Principal.Text = _
            System.Threading.Thread.CurrentPrincipal.Identity.Name
        Token.Text = _
            System.Security.Principal.WindowsIdentity.GetCurrent( ).Name
    End Sub

    Sub Logout_Click(ByVal sender As Object, ByVal e As EventArgs)

        ' Remove the cookie

```

```

        FormsAuthentication.SignOut( )

        ' Redirect back to logon page
        Response.Redirect("Main.aspx")
    End Sub
</script>

<body>
    <h1>Principal and Access Token Values</h1>
    <form id="Form1" method="post" runat="server">
        <table>
            <tr>
                <td>PRINCIPAL [Thread.CurrentPrincipal.Identity.Name] :</td>
                <td><asp:Label id="Principal" runat="server"/></td>
            </tr>
            <tr>
                <td>TOKEN [WindowsIdentity.GetCurrent( ).Name] :</asp:Label></td>
                <td><asp:Label id="Token" runat="server"/></td>
            </tr>
            <tr align="center">
                <td colspan="2">
                    <asp:Button id="Logout" runat="server" Text="Logout "
                        OnClick="Logout_Click"/>
                </td>
            </tr>
        </table>
    </form>
</body>
</html>

```

When the user clicks the Logout button, ASP.NET executes the `Logout_Click` method, which contains two highlighted lines of code that constitute our logout functionality.

The first line calls the `FormsAuthentication.SignOut` method, which forces the deletion of the client's authentication cookie. In the second line, we call the `Response.Redirect` method to redirect the client back to the `Main.aspx` page. This causes the client to reload the current page, but because the client no longer has a valid authentication cookie, ASP.NET redirects the request to the `Logon.aspx` page for authentication.

18.3.5 Passport Authentication

Microsoft Passport .NET provides a suite of web-based services designed to centralize and simplify Internet authentication and purchasing. ASP.NET Passport authentication uses the single-sign-in (SSI) capabilities of Passport .NET to authenticate users requesting access to your ASP.NET applications.

SSI allows a you to use a single username and password to access all services that participate in the Passport .NET service. This frees you from having to remember multiple usernames and passwords, or ensuring that they are kept the same on all sites. As more web sites adopt the use of Passport .NET, the value of SSI will grow, making your web experience simpler and more pleasurable; however, to date the uptake has been limited.

The requirements of implementing Passport authentication make a useable demonstration beyond the scope of this book. You must install the Passport .NET SDK on your development machine, and learn how to use it. Then you need to license the Passport .NET service to use it from your web site, and you must have a publicly accessible domain name that the Passport .NET service can use to communicate with your site. Finally, all of your users need to create Passport .NET accounts to access your system. For further details on Microsoft Passport .NET, visit its home page at <http://www.passport.net>.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

18.4 Authorization

Authorization controls whether an authenticated identity can access the resource it requested. There are two authorization mechanisms provided by ASP.NET: File authorization and URL authorization.

18.4.1 File Authorization

File authorization is enforced automatically when you enable ASP.NET Windows authentication, and cannot be disabled or configured. After IIS authenticates the user, but before ASP.NET hands the request to your application for processing, ASP.NET checks to see whether the authenticated user has the necessary NTFS permissions to access the requested resource. If she has permission, ASP.NET will hand the request to your application for processing; otherwise, it will refuse the request.

Understand that the NTFS checks performed during File authorization are in addition to the normal enforcement of NTFS file permissions performed during application execution. If your ASP.NET application accesses files during execution, they are accessed in the context of the ASP.NET worker process identity, or the identity of the authenticated Windows user if you have enabled impersonation. If the active identity does not have the necessary NTFS permissions to access the file an error will occur, which your application must handle. See [Section 18.5](#) later in this chapter for details of how to enable impersonation.

18.4.2 URL Authorization

URL authorization is a convenient mechanism for controlling access to resources when you are not using Windows authentication, although it can also be used to good effect at all times. While File authorization is based on the NTFS permissions of an authenticated Windows account, URL authorization is based on the principal (and identity) assigned to the application by the ASP.NET authentication process. See [Chapter 10](#) for details about identities and principals.

The URL authorization settings are contained in the configuration files of your ASP.NET application. You can configure URL authorization at any level including individual files, by using the `<location>` configuration element. URL authorization allows you to grant or deny access to resources based on the identity and roles of the current principal. You can also restrict access based on the type of HTTP action attempted, such as `Get` or `Post`.

To configure URL authorization, include an `<authorization>` element in your configuration file. The `<authorization>` element can contain two types of child elements: `<allow>` grants access to resources and `<deny>` denies access. Both the `<allow>` and `<deny>` elements require you to specify one or both of the users and roles, and optionally the HTTP command (or verb) to which the element applies. The following excerpt from a *Web.config* file provides examples of the `<authorization>` element's use:

```
<configuration>
  <system.web>
```

```
<authorization>
  <deny users="?" />
  <allow users="Alice,Bob" />
  <deny users="Eve" />
  <allow roles="Administrators, Managers, Developers" />
  <deny verbs="POST" roles="Users" />
  <allow verbs="GET" roles="Users" />
  <deny users="*" />
</authorization>
</system.web>
</configuration>
```

When authorizing access to a resource, ASP.NET will only search until it finds the first `<allow>` or `<deny>` element that applies to the current identity, and so the order of the elements is important. In addition, authorization restrictions created lower down in the application hierarchy take precedence over those defined higher up.

In the example, we use two special characters. The "?" character represents all unauthenticated users, and so the first thing our example does is refuse access to anyone that has not been authenticated. The second special character is "*", which represents all users; the last thing we do is deny access to all users that have not been granted specific or denied permission by one of the other settings.

By default, the *machine.config* file contains the following configuration, meaning that all users have access to all files:

```
<authorization>
  <allow users="*" />
</authorization>
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

18.5 Impersonation

As we discussed earlier in [Section 18.2](#), an ASP.NET application runs in the context of a specified user account. By default, this is a special account named ASPNET, which has very restricted access to system resources.

Impersonation allows an ASP.NET application to run in the context of the authenticated Windows user that made the web request. This means that all server-side resources accessed by the ASP.NET application occur in the context of the authenticated user. Although this complicates the task of security administration because you must configure access to individual resources for individual users or groups, it gives you fine-grained control over who accesses what resources.

To enable impersonation, add an `<identity>` element to the `<system.web>` element of your configuration file, as shown here:

```
<configuration>
  <system.web>
    <identity impersonate="true"/>
  </system.web>
</configuration>
```

You can configure impersonation at any level of the configuration hierarchy, and you can configure it differently for individual applications or application resources if desired.

If impersonation is enabled in conjunction with IIS anonymous authentication, the ASP.NET application will impersonate the user account configured in IIS for anonymous access, by default the IUSR_MACHINENAME user account. See [Section 18.3.1](#) earlier in this chapter for details on configuring anonymous authentication.

It is also possible to configure an ASP.NET application to impersonate a specific Windows user by adding `username` and `password` attributes to the `<identity>` element, as shown here. You should replace the "MACHINE" element of the username with the name of your computer, or the name of the Windows domain to which the user account belongs:

```
<configuration>
  <system.web>
    <identity impersonate="true" userName="MACHINE\bob" password="test"/>
  </system.web>
</configuration>
```

To avoid the inclusion of plaintext usernames and passwords in your configuration files, the `<identity>` element allows you to store encrypted credentials in the Windows registry and reference them from the `<identity>` element. This is the same as we described for the `<processModel>` element earlier in this chapter; refer to [Section 18.2](#) for details.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

18.6 ASP.NET and Code-Access Security

Although they present a simplified development model, ASP.NET applications are fully fledged .NET applications that run on the CLR. ASP.NET pages and associated code-behind modules are compiled into .NET assemblies prior to execution. As such, ASP.NET applications are subject to the runtime's code-access security controls, which we discussed in [Chapter 5](#) through [Chapter 9](#). However, ASP.NET applications represent a different application model than we focused on in those chapters.

With ASP.NET applications, you are not facing the problem of protecting your machines from highly mobile code downloaded from a variety of untrusted and potentially questionable sources. Usually, you are dealing with a smaller number of ASP.NET applications from known sources that are being uploaded to a shared hosting server. Your primary concerns are ensuring that the ASP.NET application code cannot tamper with the operation of the operating system, affect other hosted applications, or gain undesired levels of access to internal network-based resources.

Under the default code-access security policy (which we discussed in [Chapter 9](#)), all ASP.NET applications execute with full and unrestricted access, because they run from the local machine. Given the concerns we have just outlined, this is clearly not acceptable. ASP.NET implements code-access security in a way that allows the capabilities of each ASP.NET application to be constrained to a safe and secure level.

To configure the code-access permissions for an ASP.NET application, use the `<trust>` configuration element, which you can configure at the machine, site, or application level. The `level` attribute of the `<trust>` element specifies the name of a predefined trust level to assign to the application. The .NET Framework defines five standard trust levels: `Full`, `High`, `Medium`, `Low`, and `Minimal`.

Each named trust level maps to a policy level definition; we discussed policy levels in [Chapter 8](#). Each ASP.NET application runs in its own application domain, and the runtime applies the policy level associated with the trust level to the application domain in which the ASP.NET application is running. This has the effect of controlling the permissions granted to the ASP.NET application.

[Table 18-3](#) lists the names of the standard trust levels and identifies the file that contains its policy level definition. The files are all located in the `CONFIG` folder of the .NET Framework installation directory. For a complete description of the permissions granted by each trust level, consult the .NET Framework SDK documentation, or look at the contents of the policy file itself.

Table 18-3. ASP.NET trust levels and their policy file mappings

<code><trustLevel></code>	Policy file
<code>Full</code>	This trust level does not map to an external policy file, but represents the standard <code>FullTrust</code> named permission set described in Chapter 10 .
<code>High</code>	<code>web_hightrust.config</code> .

<trustLevel>	Policy file
Medium	<i>web_mediumtrust.config</i>
Low	<i>web_lowtrust.config</i>
Minimal	<i>web_minimaltrust.config</i>

The following *Web.config* extract from the *DotNetSecurity* folder shown in [Figure 18-1](#) configures trust levels for two folders. In the first instance, we use the `<location>` element to configure a **Low** trust level for the *Chapter17* folder. Because we also specify `allowOverride="false"`, all applications below the *Chapter17* folder have **Low** trust and cannot override the setting in their own *Web.config* files; this is a common approach when hosting multiple applications.

In the second instance, we specify a **Medium** trust level, which affects all folders and applications below the *DotNetSecurity* folder (except *Chapter 17*):

```
<configuration>
  <location path="Chapter17" allowOverride="false">
    <trust level="Low"/>
  </location>
  <system.web>
    <trust level="Medium"/>
  </system.web>
</configuration>
```

In addition to the standard trust levels defined in [Table 18-3](#), you can define your own using the `<securityPolicy>` and `<trustLevel>` configuration elements. The `<securityPolicy>` element can be specified at the machine, site, or application level, and contains a set of `<trustLevel>` elements, each defining a mapping between a trust level name and a security policy file.

The following *Web.config* example shows the definition of two new trust levels named **BasicService** and **PremiumService**. [Chapter 8](#) contains all the information you need to build a security policy hierarchy that you can store to disk and assign to a custom ASP.NET trust level:

```
<configuration>
  <system.web>
    <securityPolicy>
      <trustLevel name="BasicService" policyFile="basic_svc.config"/>
      <trustLevel name="PremiumService" policyFile="premium_svc.config"/>
    </securityPolicy>
  </system.web>
</configuration>
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 19. COM+ Security

COM+ is a set of component services that assist programmers in building scalable and robust enterprise applications. COM+ services include support for managing distributed transactions and controlling the life cycle of software components. In this chapter, we discuss the COM+ security services and explain how you can apply them to your COM+ components.

We assume that you have some experience using COM+, and are familiar with the basic design and implementation of its component services. Aside from a brief definition of the terms we use, we explain those aspects of COM+ that relate to security only. For more information, see *COM and .NET Component Services* by Juval Löwy (O'Reilly) as well as the .NET SDK.

[\[Team LiB \]](#)

[[Team LiB](#)]

19.1 COM+ Security Explained

A .NET class that uses COM+ services is called a *serviced component* or a *configured class*. A COM+ application consists of one or more serviced components, which are administered together; there is no requirement for the components within a COM+ application to interoperate, although typically a COM+ application comprises components that offer related functionality.

A serviced component makes functionality available to a client application by implementing one or more interfaces. The client application creates a new instance of the component class and consumes the functionality exposed by invoking the component members. The use of COM+ is not visible to the client, which uses a serviced component in the same way as any other .NET class.

There are three types of COM+ application, illustrated by [Figure 19-1](#), where the application type affects the way in which instances of serviced components are created. We summarize the application types as follows:

Library applications

This type of application creates new instances of serviced components within the process of the client application; the client and the component reside on the same computer.

Server applications

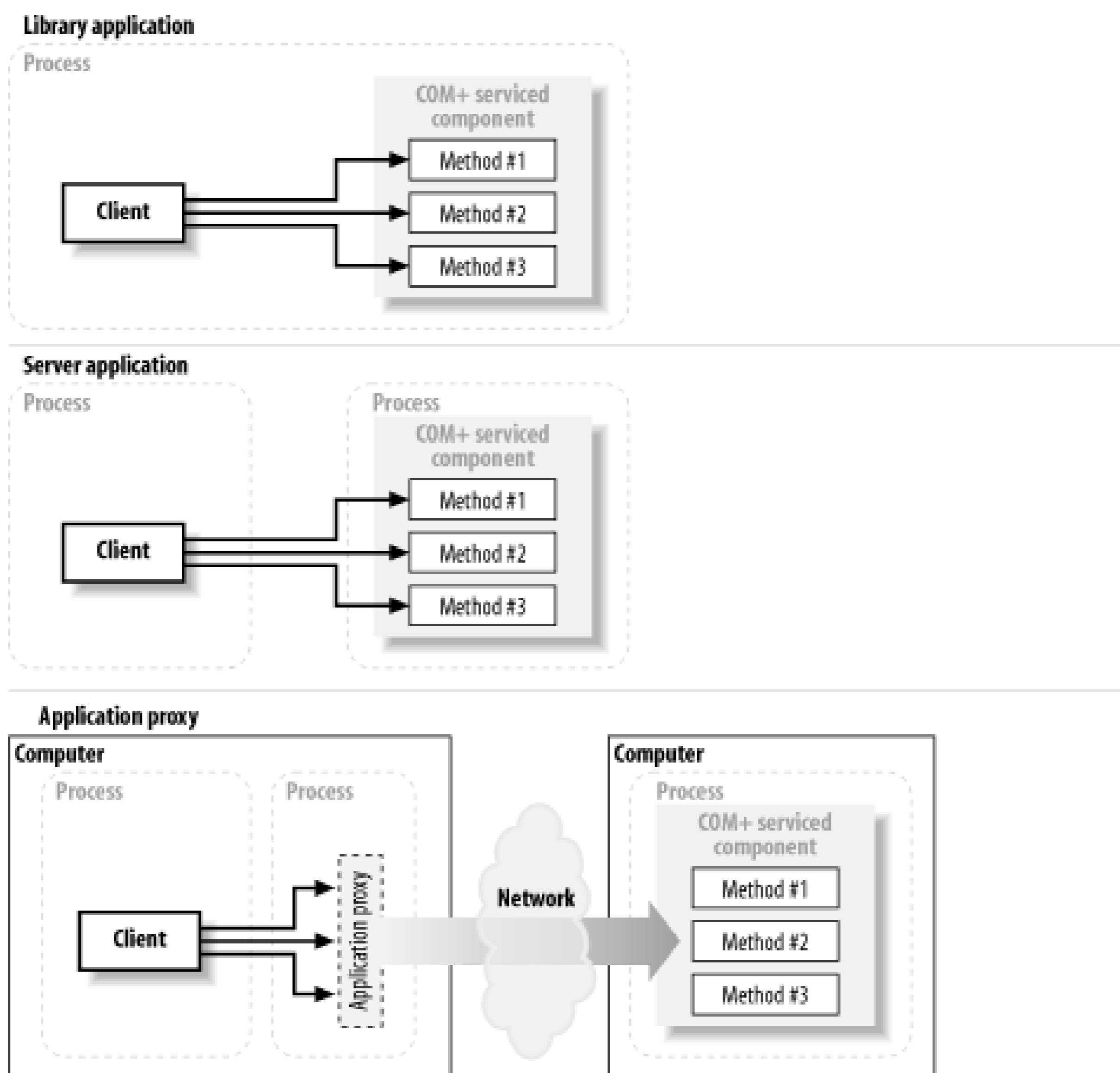
This type of application creates new instances of serviced components in a separate process than that of the client application but on the same computer. The client application invokes the methods of the serviced component using a Remote Procedure Call (RPC).

Application proxies

An application proxy allows a client application to invoke the methods of a serviced component on another computer. Method invocations are performed with a remote procedure call transmitted across a network, although the client application is unaware that the serviced component resides on another computer.

The type of application affects the COM+ security services that you can apply. COM+ security consists of *role-based security* (RBS) and *process-access security* (PAS). RBS controls access to components and their methods, and is applicable to all types of COM+ applications, irrespective of the number of processes or computers involved. PAS is responsible for coordinating security between different processes, and can be applied only to server applications and application proxies.

Figure 19-1. COM+ application types



19.1.1 COM+ Role-Based Security

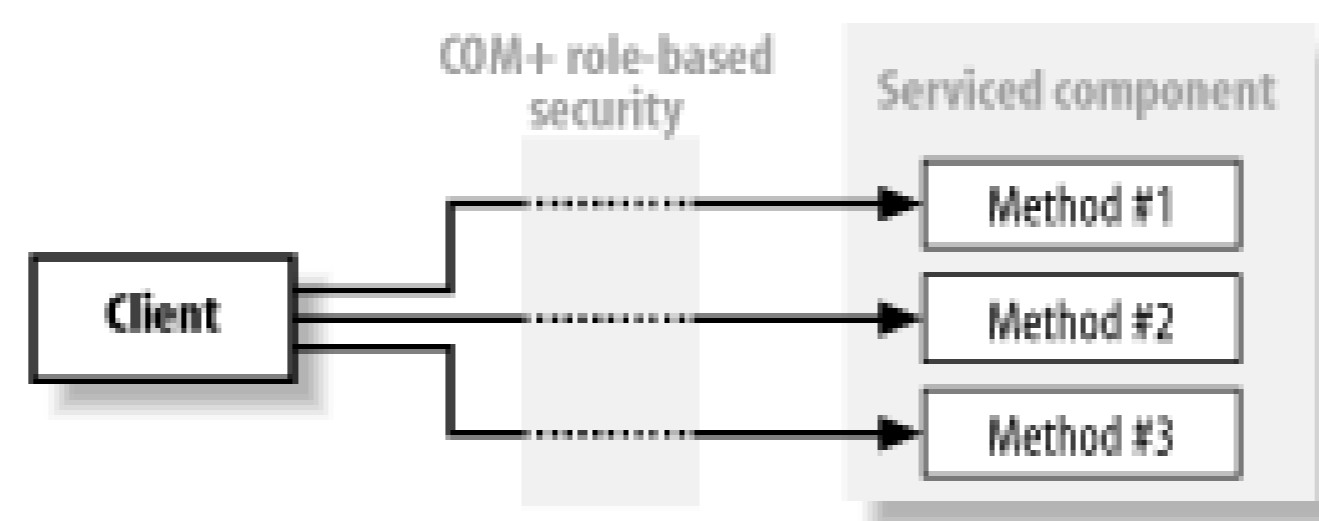
The most widely used aspect of COM+ security is RBS, which determines which clients may access the functionality provided by a serviced component.

Access to a serviced component is based on the identity of the user account that is executing the client application. COM+ applications define *roles*, which the system administrator assigns to Windows user accounts. Roles can grant a user access to the entire component or to individual methods within the component. Both the component developer and the system administrator create COM+ roles and associate them with components or component methods, as shown in [Figure 19-1](#). We demonstrate how to define roles programmatically in [Section 19.2](#) and how to use the COM+ administration tool in [Section 19.3](#).

The roles used by COM+ RBS are independent of the user groups supported by Windows. A system administrator must assign COM+ roles to user accounts before they can access components protected by COM+ RBS. If access to a serviced component requires a role, say Developers, then belonging to a Windows user group called Developers will not grant an account access to the component; the system administrator must explicitly assign membership of the COM+ Developers role.

The COM+ RBS system acts as a "gatekeeper" between client applications and serviced components, as illustrated in [Figure 19-2](#). When a client application invokes a serviced component method, COM+ determines the identity of the user account executing the application, assesses which roles have been assigned to the account, and permits or rejects the request based on the access granted by the assigned roles. For example, if the user account "Alice" is executing an application that invokes a method that requires the Developer role, COM+ RBS will grant access only if the system administrator has assigned the required COM+ role to Alice's user account.

Figure 19-2. COM+ role-based security acts as a gateway between the client and the serviced component



Although the programmer can define COM+ RBS roles, only the system administrator is able to assign those roles to Windows user accounts; see [Section 19.3](#) for more details.

COM+ RBS is independent of, and incompatible with, the .NET Framework role-based security that we introduced in [Chapter 10](#). You should use COM+ RBS when writing serviced components and when writing applications that interoperate with existing COM+ applications.

19.1.2 COM+ Process-Access Security

PAS applies to server applications and application proxies only; PAS is responsible for the trust relationship established between the processes that contain the client application and the serviced component. There are two aspects of PAS: authentication and impersonation.

19.1.2.1 Authentication

PAS authentication verifies the identity of clients trying to invoke serviced component methods; in the previous section, we explained that COM+ RBS uses the client identity to grant access to serviced components, and it is essential that we are able to identify a client accurately if we are to rely on COM+ RBS.

In COM+ server applications and application proxies, the client and component reside in separate processes; a malicious client could attempt to assume another identity in order to bypass role-based security. COM+ PAS provides a range of authentication levels that you can select for your project; we summarize these options below:

None

PAS does not verify the identity claims of the client when this option is used; this option offers the least protection.

Default

PAS uses the default machine configuration to select an authentication option; the *None* option will never be selected.

Connect

The client presents evidence to support its stated identity, which PAS validates. All subsequent communication is insecure.

Call

The headers of each call from the client are cryptographically signed, but no other data exchanged between the client and component is protected, allowing the communications to be tampered with.

Packet

This option is similar to *Call*, but the headers of each data packet are signed. This is the default PAS authentication option for .NET serviced components.

Packet Integrity

Each packet of data sent between the client and the component is signed to prevent tampering

Packet Privacy

Each packet of data sent between the client and the component is signed and encrypted to prevent tampering and to ensure that the entire communication between the client and the component remains confidential.

In general, the more security provided by an authentication option, the more system resources required to perform the authentication checks. It is important to select an authentication option that is appropriate for your project; always selecting the most secure option results in the needless consumption of computer resources and places an unwarranted burden on client applications.

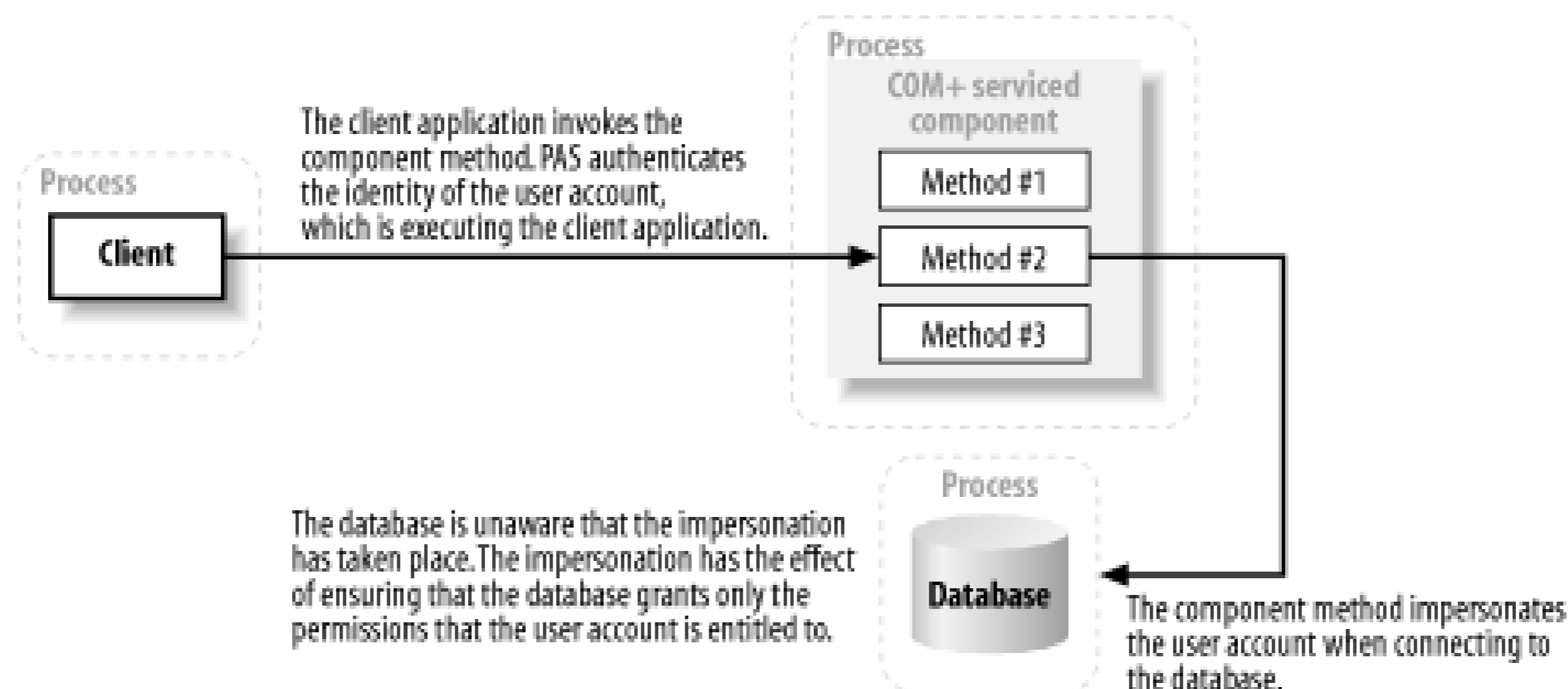
The authentication level used between the client and component processes depends on the configuration of the client as well as the COM+ application. The client and the serviced component declare their desired authentication options, and the most secure option is used; for example, if a component is configured to request Packet Integrity and the client is configured to request Packet Privacy, then Packet Privacy will be used.

In a COM+ library application, the issue of authentication does not arise; instances of a serviced component are created within the same process as the client application, and the identity of the caller is always known to be the owner of the client process.

19.1.2.2 Impersonation

It is common for serviced components to perform tasks on behalf of the account that executes the client application. For example, consider a serviced component that defines a method to write a record to a database. The client application calls the component method to write the record; the component may need to connect to the database and write the new record using the account identity to gain permission to write to the database and to create the audit trail correctly. When a component adopts the identity of a client, it is impersonating the account identity. [Figure 19-3](#) illustrates impersonation.

Figure 19-3. A serviced component can impersonate a client account in order to carry out tasks on behalf of the user



COM+ PAS defines several levels of impersonation, which we summarize in the following list:

Anonymous

The client is anonymous to the component, and no identity information is available (this level of impersonation is available only when the client process and the component process reside on the same computer).

Identify

Allows the server to identify the client and perform access checks using the client's access token.

Impersonate

The component can impersonate the client identity to access resources on the local computer.

Delegate

This is the highest level of impersonation and allows the component to impersonate the client identity to access network resources.

The level of impersonation is specified by the client application and is an expression of the level of trust that the client has in the integrity of the component. When a client specifies the **Anonymous** impersonation level, the component cannot access the client identity; when specifying the **Delegate** impersonation level, a client must fully trust the integrity of the component, because it can act on behalf of the user to perform any kind of task.

There is no direct .NET support for a client application to specify a COM+ PAS impersonation level; a native method must be used—consult the Windows Platform API for full details. In the next section, we demonstrate how a component can specify an impersonation level for when it communicates as a client.

For a serviced component, the value of impersonation becomes apparent when invoking the methods of another COM+ component, where the calling component becomes, in effect, a COM+ client application. Under these circumstances, the calling component specifies an impersonation level to define the level of trust that is granted to the called component. In the next section, we show you how to use the **ApplicationAccessControl** attribute to set the impersonation level your component will use when acting as a client of other COM+ components.

[\[Team LiB \]](#)

[Team LiB]

19.2 Programming COM+ Security

In this section, we explain how to configure COM+ security with .NET attributes; we do this by creating and deploying an example serviced component. We demonstrate how to use COM+ RBS and PAS to control access to a simple component that tracks software defects for an imaginary software product called *SecurityPro*. In Section 19.3, we demonstrate an alternative approach to configuring COM+ security.

We begin by defining the `IDefectTracker` interface, which contains the three methods we use to simulate defect tracking. We briefly describe the purpose of each interface method in the list below. We are developing this component to demonstrate COM+ security, so we do not define types to represent defects, or specify method arguments:

`ViewAllDefects`

Returns all of the pending defects for the product

`CreateNewDefect`

Creates a new record representing a pending defect

`CloseDefect`

Closes a defect and removes it from the pending list

The following statements define our simple `IDefectTracker` interface:

```
# C#

using System;

public interface IDefectTracker {

    void ViewAllDefects( );

    void CreateNewDefect( );

    void CloseDefect( );
}

# Visual Basic .NET

Imports System

Public Interface IDefectTracker

    Sub ViewAllDefects( )

    Sub CreateNewDefect( )
```

```
Sub CloseDefect( )
```

```
End Interface
```

In the following sections, we create a serviced component that implements the `IDefectTracker` interface, and apply .NET attributes to configure COM+ RBS and PAS.

19.2.1 Creating the Serviced Component

A serviced component must inherit from the `System.EnterpriseServices.ServicedComponent` class and must be a member of a strong-named assembly. The following statements list the `SecurityProTracker` class, which extends `ServicedComponent` and implements the `IDefectTracker` interface.

```
# C#
```

```
using System;
```

```
using System.EnterpriseServices;
```

```
// Specify the file containing the key for assembly signing
[assembly: System.Reflection.AssemblyKeyFile("mykey.key")]
```

```
public class SecurityProTracker: ServicedComponent, IDefectTracker {
```

```
    public void ViewAllDefects( ) {}
```

```
    public void CreateNewDefect( ) {}
```

```
    public void CloseDefect( ) {}
```

```
}
```

```
# Visual Basic .NET
```

```
Imports System
```

```
Imports System.EnterpriseServices
```

```
' Specify the file containing the key for assembly signing
<Assembly: System.Reflection.AssemblyKeyFile("mykey.key")>
```

```
Public Class SecurityProTracker
```

```
    Inherits ServicedComponent
```

```
    Implements IDefectTracker
```

```
    Public Sub ViewAllDefects( ) Implements IDefectTracker.ViewAllDefects
```

```
    End Sub
```

```
    Public Sub CreateNewDefect( ) Implements IDefectTracker.CreateNewDefect
```

```
    End Sub
```

```
    Public Sub CloseDefect( ) Implements IDefectTracker.CloseDefect
```

```
    End Sub
```

End Class



We do not discuss the members of the `ServicedComponent` class, which is outside the scope of this book; it is sufficient to understand that classes must extend `ServicedComponent` in order to be serviced components. Consult the .NET Framework for full details.

The emphasized statements specify the file containing the cryptographic key we use to create the strong name for our assembly; we create this file in Section 19.2.4. See Chapter 2 for details of assembly strong names. We have now satisfied the two requirements for COM+ serviced components and we can compile and install our serviced component without any further changes.

19.2.2 Specifying the COM+ Application Type

The `System.EnterpriseServices.ApplicationActivation` class is the first COM+ attribute we apply, allowing us to specify whether we want a COM+ server or library application. This attribute accepts a single parameter, which must be a value from the `System.EnterpriseServices.ActivationOption` enumeration, the values of which are detailed in Table 19-1. Application proxies are created with the Component Services tool; consult the Windows documentation for details.

Table 19-1. The enumeration values

Value	Description
<code>Library</code>	Specifies that serviced components are activated in the client application process
<code>Server</code>	Specifies that serviced components are activated in a process provided by COM+

The emphasized statements in the code fragment below demonstrate the use of the `ApplicationActivation` attribute to specify a COM+ server application for our example component:

```
# C#

using System;
using System.EnterpriseServices;

// Specify the file containing the key for assembly signing
[assembly: System.Reflection.AssemblyKeyFile("mykey.key")]
// Specify that we want a COM+ Server application
[assembly: ApplicationActivation(ActivationOption.Server)]

public class SecurityProTracker: ServicedComponent, IDefectTracker {

# Visual Basic .NET

Imports System
Imports System.EnterpriseServices
```

```
' Specify the file containing the key for assembly signing
<Assembly: System.Reflection.AssemblyKeyFile("mykey.key")>
' Specify that we want a COM+ Server application
<Assembly: ApplicationActivation(ActivationOption.Server)>
```

```
Public Class SecurityProTracker
    Inherits ServicedComponent
    Implements IDefectTracker
```

19.2.3 Applying the Security Attributes

In this section, we demonstrate how to use COM+ RBS and PAS, both of which are configured using .NET attributes from the `System.EnterpriseServices` namespace. COM+ security can be configured programmatically, as demonstrated in this part of the chapter, or administratively, as discussed in Section 19.3. When applying .NET attributes to configure RBS and PAS, you must bear in mind that the system administrator can change the security configuration for your components or even disable security completely.

19.2.3.1 The `ApplicationAccessControl` attribute

The first security attribute to apply is `ApplicationAccessControl`, which acts as a master switch for COM+ application security and allows you to specify your PAS authentication and impersonation levels. This attribute is always applied to an assembly, because it affects the entire COM+ application. The `ApplicationAccessControl` attribute defines four properties:

Value

This is the master switch for COM+ security and is applied at the application level. As we explained at the start of this chapter, a COM+ application is made up of one or more serviced components, and this value enables or disables RBS and PAS security for *every* component in the application. If this property is set to `false`, then no validation of client identity will be performed and no access control will be enforced.

AccessChecksLevel

This property accepts a value from the `AccessChecksLevelOption` enumeration, which defines two values, detailed below. If the `Value` property is set to `false`, then this property has no effect, since COM+ security is disabled.

Application

The `Application` value enables PAS only; client identity will be authenticated according to the value of the `ApplicationAccessControl` attribute's `Authentication` property (see below), but COM+ RBS is not enforced. This is the default used if no value is specified in the attribute declaration.

ApplicationComponent

The `ApplicationComponent` value specifies that PAS and RBS will be enabled.

Authentication

The `Authentication` property accepts a value from the `AuthenticationOption` enumeration, whose seven values refer to the authentication levels we discussed in Section 19.1.2.

ImpersonationLevel

The `ImpersonationLevel` property accepts a value from the `ImpersonationLevelOption` enumeration, whose values refer to the impersonation levels we discussed in Section 19.1.2

The emphasized statements below show the use of the `ApplicationAccessControl` attribute for our example component. We have enabled COM+ security, specified that we wish to use PAS and RBS, and accepted the default levels for authentication and impersonation:

```
# C#

using System;
using System.EnterpriseServices;

// Specify the file containing the key for assembly signing
[assembly: System.Reflection.AssemblyKeyFile("mykey.key")]
// Specify that we want a COM+ Server application
[assembly: ApplicationActivation(ActivationOption.Server)]

// Specify our application level security settings
[assembly: ApplicationAccessControl(
    // Enable COM+ Security
    Value=true,
    // Enable PAS and RBS
    AccessChecksLevel=AccessChecksLevelOption.ApplicationComponent,
    // Use the computer default values for authentication and impersonation
    Authentication=AuthenticationOption.Default,
    ImpersonationLevel=ImpersonationLevelOption.Identify)]

public class SecurityProTracker: ServicedComponent, IDefectTracker {

# Visual Basic .NET

Imports System
Imports System.EnterpriseServices

' Specify the file containing the key for assembly signing
<Assembly: System.Reflection.AssemblyKeyFile("mykey.key")>
' Specify that we want a COM+ Server application
<Assembly: ApplicationActivation(ActivationOption.Server)>

' Specify our application level security settings
<Assembly: ApplicationAccessControl( _
    Value:=True, _
    AccessChecksLevel:=AccessChecksLevelOption.ApplicationComponent, _
    Authentication:=AuthenticationOption.Default, _
    ImpersonationLevel:=ImpersonationLevelOption.Identify)>

Public Class SecurityProTracker
    Inherits ServicedComponent
    Implements IDefectTracker
```

19.2.3.2 The ComponentAccessControl Attribute

The `ComponentAccessControl` attribute enables or disables COM+ security for an individual serviced component. This attribute accepts a single argument, which is set to `true` (to enable COM+ security for a component) or `false` (to disable COM+ access control for a component). The emphasized statements below demonstrate how we apply this attribute to enable COM+ security for our `SecurityProTracker` component:

```
# C#

// Enable COM+ Security for this component
[ComponentAccessControl(true)]
public class SecurityProTracker: ServicedComponent, IDefectTracker {

# Visual Basic .NET

'Enable COM+ Security for this component
<ComponentAccessControl(True)> _
Public Class SecurityProTracker
    Inherits ServicedComponent
    Implements IDefectTracker
```

You must apply this attribute to every component that requires security services within a COM+ application. The `ComponentAccessControl` attribute has no effect if the `ApplicationAccessControl` attribute has disabled COM+ security at the application level.

19.2.3.3 The SecurityRole attribute

The `SecurityRole` attribute has a different affect depending on whether it is applied to an assembly, an interface, a class, or a method. In this section, we define some roles for our example component and use the `SecurityRole` attribute to apply COM+ RBS.

We define the following roles for our component. Figure 19-4 shows how we map these roles to the methods implemented by the `SecurityProTracker` component:

User

The User role can see the list of defects, but is unable to create new defects or close existing ones.

Tester

The Tester role is able to see the list of defects and to create new defects. Testers cannot close existing defects.

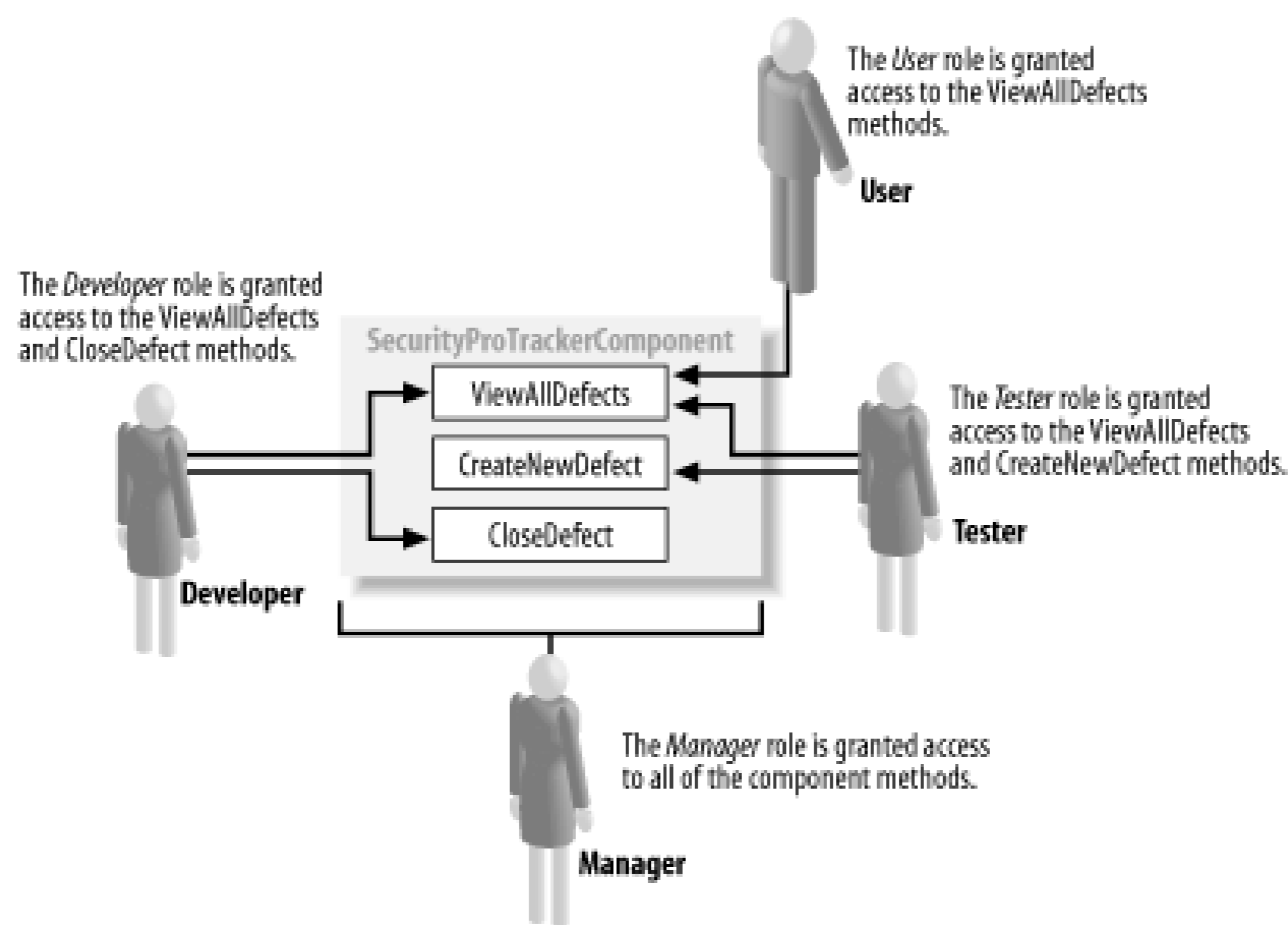
Developer

The Developer role is able to see the list of defects, and close defects. Developers are not able to create new defects.

Manager

The Manager role is able to view defects, create new defects and close defects.

Figure 19-4. Mapping roles to the `SecurityProTracker` component



When the `SecurityRole` attribute role is applied to an assembly, it has the effect of defining the role within the COM+ application; we shall see the effect of this in Section 19.3. When used to define roles in this way, the attribute defines the properties described by Table 19-2.

Table 19-2. SecurityRole attribute properties

Property	Description
<code>Role</code>	Specifies the name of the property to define
<code>Description</code>	Describes the role
<code>SetEveryoneAccess</code>	Adds the Windows <i>Everyone</i> group to the role

The `SetEveryoneAccess` property is worthy of special note; the `SecurityRole` can be applied to an assembly to define roles for a COM+ application, but the system administrator is responsible for assigning the COM+ roles to Windows user accounts. If the `SetEveryoneAccess` property is set to `true`, the Windows *Everyone* group will be assigned the role and all users will be able to access the methods protected by this role.

Use the `SetEveryoneAccess` property with care; it has the effect of assigning a role to all users, which may negate the benefits of COM+ RBS.

The following statement demonstrates how to define the *User* role by applying the `SecurityRole` attribute to an assembly, specifying a description and ensuring that the *Everyone* group is not assigned the role:

```
# C#

// Define the Users role
[assembly: SecurityRole("User",
```

```

        Description="Users of the Security Pro product",
        SetEveryoneAccess=false)]

```

```
# Visual Basic .NET
```

```

' Define the Users role
<Assembly: SecurityRole("User", _
    Description="Users of the Security Pro product", _
    SetEveryoneAccess:=False)>

```

Applying the `SecurityRole` attribute to an assembly only defines the role and allows you to provide a description; it does not grant the role access to the methods defined by an interface or component. The emphasized statements below demonstrate how we define the roles for our example component:

```
# C#
```

```

// Define the Users role
[assembly: SecurityRole("User",
    Description="Users of the Security Pro product",
    SetEveryoneAccess=false)]
// Define the Tester role
[assembly: SecurityRole("Tester",
    Description="Security Pro product testers",
    SetEveryoneAccess=false)]
// Define the Developer role
[assembly: SecurityRole("Developer",
    Description="Security Pro product developers",
    SetEveryoneAccess=false)]
// Define the Manager role
[assembly: SecurityRole("Manager",
    Description="Security Pro product managers",
    SetEveryoneAccess=false)]

// Enable COM+ Security for this component
[ComponentAccessControl(true)]
public class SecurityProTracker: ServicedComponent, IDefectTracker {

```

```
# Visual Basic .NET
```

```

' Define the Users role
<Assembly: SecurityRole("User", _
    Description="Users of the Security Pro product", _
    SetEveryoneAccess:=False)>
' Define the Tester role
<Assembly: SecurityRole("Tester", _
    Description="Security Pro product testers", _
    SetEveryoneAccess:=False)>
' Define the Developer role
<Assembly: SecurityRole("Developer", _
    Description="Security Pro product developers", _
    SetEveryoneAccess:=False)>
' Define the Manager role

```

```
<Assembly: SecurityRole("Manager", _
    Description:="Security Pro product managers", _
    SetEveryoneAccess:=False)>
```

```
'Enable COM+ Security for this component
<ComponentAccessControl(True)> _
Public Class SecurityProTracker
    Inherits ServicedComponent
    Implements IDefectTracker
```

When applied to a component, the `SecureMethod` attribute grants a role access to all of the methods defined by a component. We want to grant the *Manager* role access to all of the methods defined by our example component, and we will apply the `SecurityRole` attribute to the `SecurityProTracker` class, as shown by the emphasized statements below:

```
# C#

// Enable COM+ Security for this component
[ComponentAccessControl(true)]
// Grant the Manager role access to the component
[SecurityRole("Manager")]
public class SecurityProTracker: ServicedComponent, IDefectTracker {

# Visual Basic .NET

<ComponentAccessControl(True), _
SecurityRole("Manager")> _
Public Class SecurityProTracker
    Inherits ServicedComponent
    Implements IDefectTracker
```

We can grant roles permission to access individual methods by applying the `SecurityRole` attribute to individual methods in either the interface or the component class. If we apply the attribute to an interface method, the role will be able to access the method in any component that implements the interface. If we apply the `SecurityRole` attribute to a method in the component, the role is granted access to that component only, and not other components that implement the same interface. We will apply our access controls to the interface, as shown by the emphasized statements below:

```
# C#

public interface IDefectTracker {

    [SecurityRole("User")]
    [SecurityRole("Tester")]
    [SecurityRole("Developer")]
    void ViewAllDefects( );

    [SecurityRole("Tester")]
    void CreateNewDefect( );

    [SecurityRole("Developer")]
    void CloseDefect( );
```

```

}

# Visual Basic .NET

Public Interface IDefectTracker

    <SecurityRole("User"), _
    SecurityRole("Tester"), _
    SecurityRole("Developer")> _
    Sub ViewAllDefects( )

    <SecurityRole("Tester")> _
    Sub CreateNewDefect( )

    <SecurityRole("Developer")> _
    Sub CloseDefect( )

End Interface

```

When we apply multiple attributes, a user identity granted *any* of the specified roles will be granted access. We cannot require an identity to be assigned multiple roles in order to access a method or component.

19.2.3.4 The SecureMethod attribute

The final attribute to apply is `SecureMethod`, which enables per-method access control for a component. Without this attribute, COM+ RBS will be applied on a component level only, and the role protection we have assigned to the `IDefectTracker` interface will be ignored. The emphasized statement below demonstrates how we apply this attribute to our example class; this attribute defines no properties:

```

# C#

// Enable COM+ Security for this component
[ComponentAccessControl(true)]
[SecureMethod]
// Grant the Manager role access to the component
[SecurityRole("Manager")]
public class SecurityProTracker: ServicedComponent, IDefectTracker {

# Visual Basic .NET

<ComponentAccessControl(True), _
SecureMethod( ), _
SecurityRole("Manager")> _
Public Class SecurityProTracker
    Inherits ServicedComponent
    Implements IDefectTracker

```

19.2.4 Compiling and Installing the COM+ Application

In the previous section, we applied .NET attributes to configure COM+ RBS and PAS for a serviced component. In this section, we compile our example component and show you how to install a COM+ application. Our first task is to create a code file containing the statements we introduced in the previous section; we have listed the complete C# and Visual Basic .NET definitions of the example component below, and we have saved these statements into a file called *Tracker.cs* (C#) (see Example 19-1) or *Tracker.vb* (Visual Basic .NET) (see Example 19-2):

Example 19-1. Tracker.cs (C#)

```
using System;
using System.EnterpriseServices;

// Specify the file containing the key for assembly signing
[assembly: System.Reflection.AssemblyKeyFile("mykey.key")]
// Specify that we want a COM+ Server application
[assembly: ApplicationActivation(ActivationOption.Server)]
// Specify our application level security settings
[assembly: ApplicationAccessControl(
    // Enable COM+ Security
    Value=true,
    // Enable PAS and RBS
    AccessChecksLevel=AccessChecksLevelOption.ApplicationComponent,
    // Use the computer default values for authentication and impersonation
    Authentication=AuthenticationOption.Default,
    ImpersonationLevel=ImpersonationLevelOption.Identify)]

// Define the Users role
[assembly: SecurityRole("User",
    Description="Users of the Security Pro product",
    SetEveryoneAccess=false)]
// Define the Tester role
[assembly: SecurityRole("Tester",
    Description="Security Pro product testers",
    SetEveryoneAccess=false)]
// Define the Developer role
[assembly: SecurityRole("Developer",
    Description="Security Pro product developers",
    SetEveryoneAccess=false)]
// Define the Manager role
[assembly: SecurityRole("Manager",
    Description="Security Pro product managers",
    SetEveryoneAccess=false)]

// Enable COM+ Security for this component
[ComponentAccessControl(true)]
[SecureMethod]
// Grant the Manager role access to the component
[SecurityRole("Manager")]
public class SecurityProTracker: ServicedComponent, IDefectTracker {

    public void ViewAllDefects( ) {}
}
```

```

        public void CreateNewDefect( ) {}

        public void CloseDefect( ) {}
    }

public interface IDefectTracker {

    [SecurityRole("User")]
    [SecurityRole("Tester")]
    [SecurityRole("Developer")]
    void ViewAllDefects( );

    [SecurityRole("Tester")]
    void CreateNewDefect( );

    [SecurityRole("Developer")]
    void CloseDefect( );
}

```

Example 19-2. Tracker.vb (VB.NET)

```

Imports System
Imports System.EnterpriseServices

' Specify the file containing the key for assembly signing
<Assembly: System.Reflection.AssemblyKeyFile("mykey.key")>
' Specify that we want a COM+ Server application
<Assembly: ApplicationActivation(ActivationOption.Server)>

' Specify our application level security settings
<Assembly: ApplicationAccessControl( _
    Value:=True, _
    AccessChecksLevel:=AccessChecksLevelOption.ApplicationComponent, _
    Authentication:=AuthenticationOption.Default, _
    ImpersonationLevel:=ImpersonationLevelOption.Identify)>

' Define the Users role
<Assembly: SecurityRole("User", _
    Description:="Users of the Security Pro product", _
    SetEveryoneAccess:=False)>
' Define the Tester role
<Assembly: SecurityRole("Tester", _
    Description:="Security Pro product testers", _
    SetEveryoneAccess:=False)>
' Define the Developer role
<Assembly: SecurityRole("Developer", _
    Description:="Security Pro product developers", _
    SetEveryoneAccess:=False)>
' Define the Manager role
<Assembly: SecurityRole("Manager", _
    Description:="Security Pro product managers", _

```

```

        SetEveryoneAccess:=False)>

<ComponentAccessControl(True), _
SecureMethod( ), _
SecurityRole("Manager")> _
Public Class SecurityProTracker
    Inherits ServicedComponent
    Implements IDefectTracker

    Public Sub ViewAllDefects( ) Implements IDefectTracker.ViewAllDefects
    End Sub

    Public Sub CreateNewDefect( ) Implements IDefectTracker.CreateNewDefect
    End Sub

    Public Sub CloseDefect( ) Implements IDefectTracker.CloseDefect
    End Sub

End Class

Public Interface IDefectTracker

    <SecurityRole("User"), _
    SecurityRole("Tester"), _
    SecurityRole("Developer")> _
    Sub ViewAllDefects( )

    <SecurityRole("Tester")> _
    Sub CreateNewDefect( )

    <SecurityRole("Developer")> _
    Sub CloseDefect( )

End Interface

```

We need to create the cryptographic key file that we specified with the `AssemblyKeyFile` attribute. We will create a test key pair for our example using the Strong Name Tool (*sn.exe*); consult Chapter 2 for details of signing assemblies. We assume that your DOS path includes the tools provided with the .NET Framework SDK; consult the .NET documentation for instructions. The following statement creates the key file we specified:

```
sn -k mykey.key
```

The following statements show how to compile the code file to create *SecurityProTracker.dll*, which is a library assembly with a strong name:

```
# C#
```

```
csc /target:library /out:SecurityProTracker.dll DefectTracker.cs
```

```
# Visual Basic .NET
```

```
vbc /target:library /out:SecurityProTracker.dll /reference:System.EnterpriseServices.dll DefectTracker.vb
```

Now that we have created an assembly containing our component, we can use the .NET Services Installation tool (*regsvcs.exe*) to install the component on the local computer:

```
regsvcs SecurityProTracker.dll
```

The output from the .NET Services Installation tool is shown below:

```
Microsoft (R) .NET Framework Services Installation Utility Version 1.0.3705.288  
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
```

```
Installed Assembly:
```

```
    Assembly: C:\SecurityProTracker.dll  
    Application: SecurityProTracker  
    TypeLib: c:SecurityProTracker.tlb
```

[Team LiB]

[\[Team LiB \]](#)

19.3 Administering COM+ Security

In the previous section, we compiled and installed a COM+ application that contained our example component. We used .NET attributes to specify the COM+ security settings, but we need to complete the component configuration before the component can be used. In this section, we demonstrate how to assign roles to Windows user accounts and then continue to demonstrate how the COM+ administration tool can be used as an alternative to .NET attributes or as a means to alter the security configuration specified using attributes.

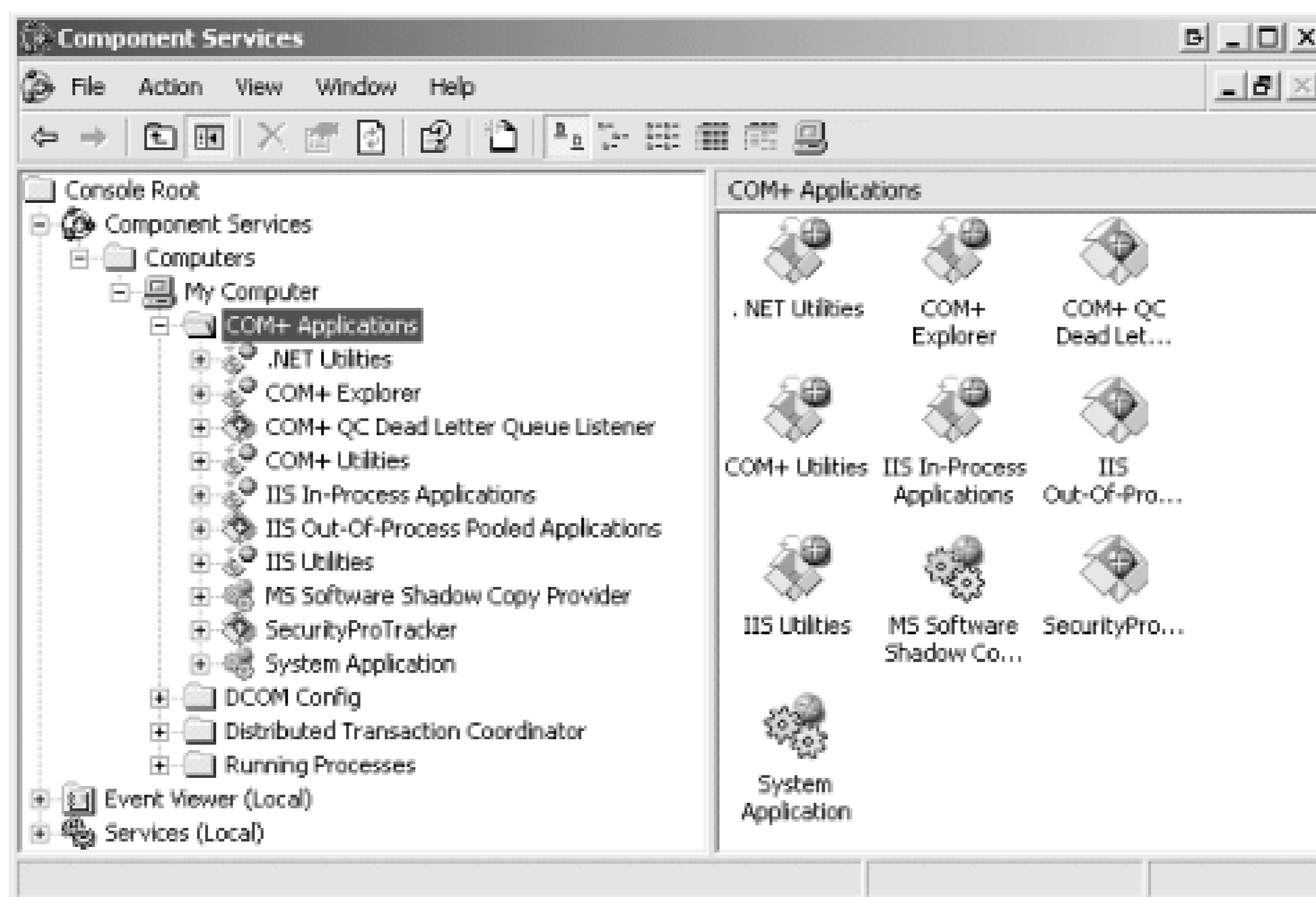
19.3.1 Viewing the COM+ Catalogue

The collection of COM+ applications installed on a computer is called the COM+ Catalogue. The catalogue is administered with the Component Services tool, which is included in the Administrative Tools section of the Windows Control Panels. [Figure 19-5](#) shows the Component Services tool as it appears when first started.

Figure 19-5. The Component Services tool

Expand the Component Services tree and drill down to the COM+ catalogue for the local computer; [Figure 19-6](#) shows the list of COM+ services for a standalone computer. The list of applications includes our example application (called *SecurityProTracker*, named after the assembly we created) as well as system applications that operate as part of Windows.

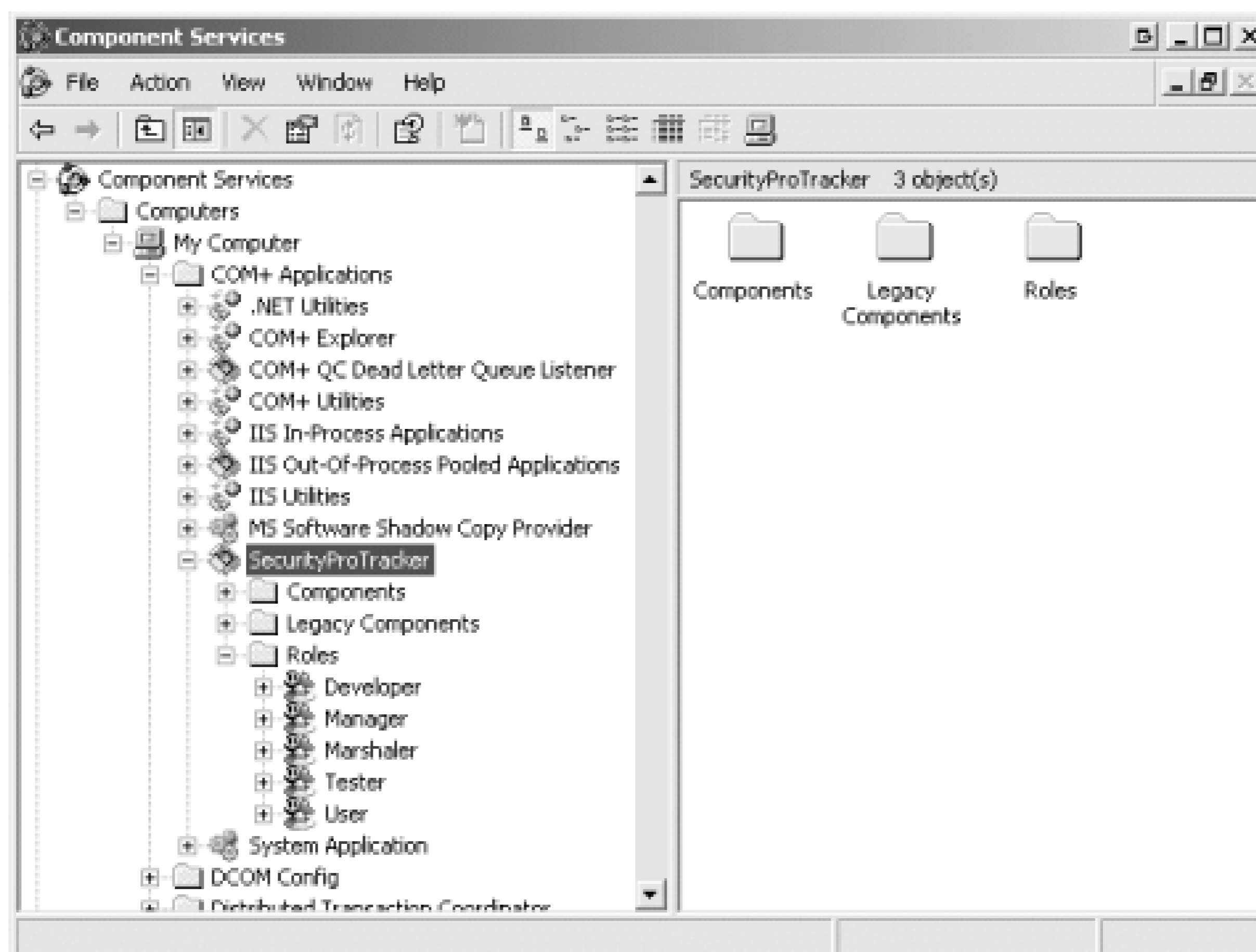
Figure 19-6. The COM+ catalogue



19.3.2 Populating COM+ Application Roles

Expand the *SecurityProTracker* item, and you will see details of the COM+ application that we created in the previous section. The Component Services explorer provides information grouped into Components, Legacy Components, and Roles. Expanding the Roles item lists the roles defined by the application, as shown in [Figure 19-7](#).

Figure 19-7. The roles associated with the SecurityProTracker COM+ application



You will see icons representing the roles that we specified with the `SecurityRole` attribute; these roles are Developer, Manager, Tester, and User. You will also see another role, named Marshaler—see the sidebar for details. The COM+ system processed the attributes from our assembly to create these roles, but they are not yet assigned to user accounts, and so clients cannot access our component. In this section, we demonstrate the simple process of assigning roles to user accounts in order to complete the configuration of our example component.

The COM+ Marshaler Role

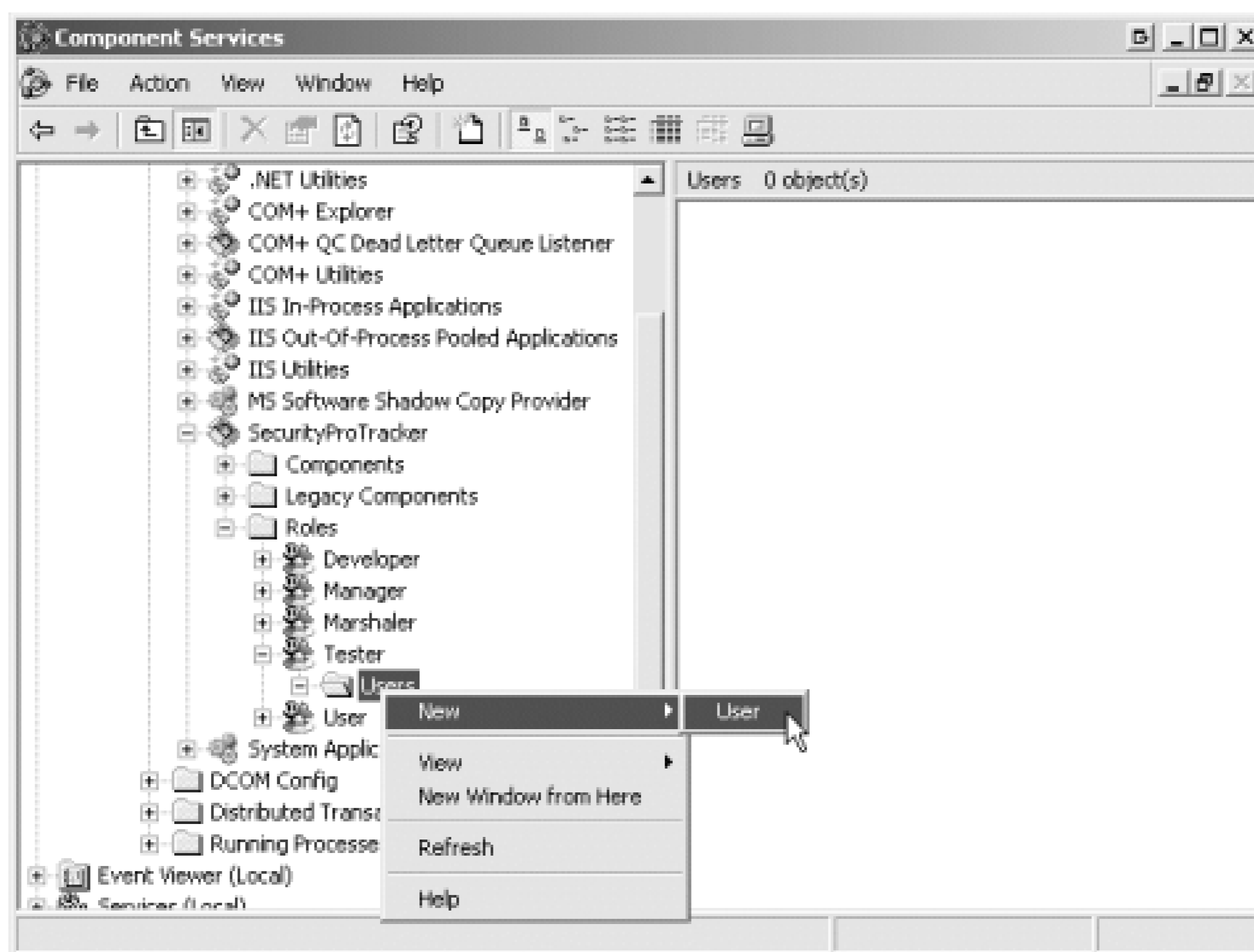
When you install a serviced component that uses the `SecureMethod` attribute, the COM+ system will automatically create the Marshaler role. The `SecureMethod` attribute indicates that you wish to apply RBS access control on individual methods defined by the component, so that, for example, users assigned the User role are able to access only the `ViewAllDefects` method.

The problem this creates is that users with this role have not been granted permission to create new instances of the component or to manage the disposal of component instances (because the constructor and the methods of the `IDisposable` interface are not exposed to the Developer role).

When assigning a role that has been applied to individual methods, you should also assign the Marshaler role. Assigning the Marshaler role to a user account grants that account permission to create new instances of a component, and to call the methods defined by the `System.IDisposable`, `System.IManagedObject`, and `System.EnterpriseServices.IServicedComponentInfo` interfaces; in essence, assigning this role confers the right to create and manage instances of a serviced component.

To assign a COM+ role to a user account, expand the role to reveal the Users item, and right-click to display the menu shown in [Figure 19-8](#).

Figure 19-8. Adding a new user to a COM+ role

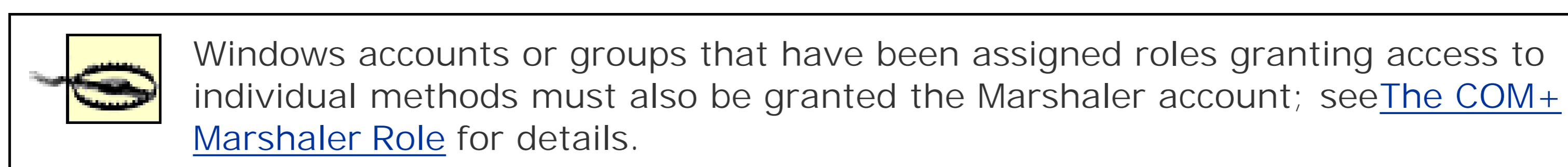


Select the User item from the New menu to display the dialog box illustrated by [Figure 19-9](#). Assign a role by typing the name of the user account into the text box—you can specify multiple accounts by separating the account names with a semi-colon. A useful feature is the ability to assign a role to a Windows group, which is equivalent to assigning the role to all of the individual accounts contained within the group. [Figure 19-9](#) illustrates adding the users Tim, Tom, Tammy, and the group "QA Group" to the COM+ Tester role.

Figure 19-9. The Select Users or Groups dialog

Pressing the OK button in the Select Users or Groups dialog assigns the roles to the specified accounts and groups; [Figure 19-10](#) shows the effect of adding Tim, Tom, Tammy, and the QA Group to the Tester role.

Figure 19-10. The accounts and groups assigned the COM+ Tester role



19.3.3 Assessing and Assigning Role Scope

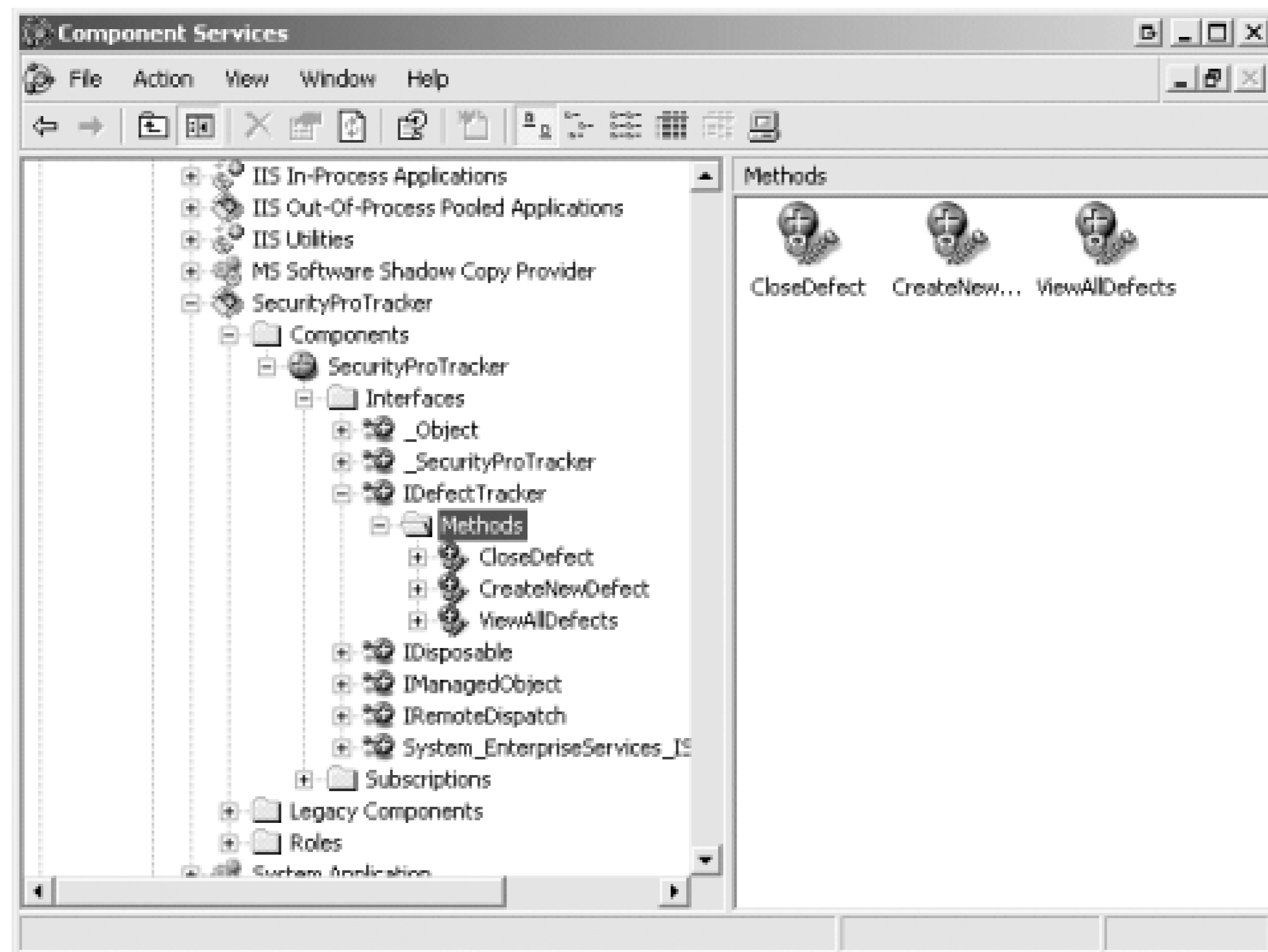
You can see what access has been granted to individual roles by expanding the Components section of the COM+ application display. Our example application contains the single component we created earlier in the chapter; expanding this item shows the Interfaces and Subscriptions items. You can see which roles have been granted access to the *SecurityProTracker* component by right-clicking on the component icon, picking the Properties menu item, and selecting the Security tab, illustrated by [Figure 19-11](#).

Figure 19-11. The Security tab of the SecurityProTracker Properties window

You will see that the Manager role is checked, meaning that user accounts assigned this role are

permitted to access all of the members defined by the `SecurityProTracker` component. If you expand the Interfaces item, you will see a list of the interfaces that the `SecurityProTracker` component implements; most of these are created by the COM+ system automatically. Expanding the `IDefectTracker` item lists the individual methods defined by our functional interface, shown by [Figure 19-12](#).

Figure 19-12. The list of methods defined by an interface implemented by the serviced component



You can see which roles are granted access to a specific method by right-clicking on a method icon, selecting the Properties menu item and choosing the Security tab from the properties window; [Figure 19-13](#) shows the roles granted access to the `ViewAllDefects` method.

Figure 19-13. Roles assigned to the ViewAllDefects method



You will see that the Developer, Tester, and User roles are checked, indicating that they are granted explicit access to this method, which is in accordance with our application of the `SecurityRole` attribute earlier in the chapter. The Manager role is listed in the Roles inherited by selected items(s) section, because the role was assigned to a parent object—in this case, to the component itself, which has the effect of implicitly granting access to all of the members defined by the interfaces it implements.

You can change the role configuration for a component by adding or removing check marks from the Security property page. For example, if you were to uncheck the Tester role for the `ViewAllDefects` method, the Tester role would have no access to any of the methods defined by the `SecurityProTracker` component, because the `ViewAllDefects` method was the only one to which you applied the Tester role.

COM+ allows both the component developer and the system administrator to manage the scope and application of RBS. In effect, this means that the configuration provided by the component developer provides an "out of the box" baseline configuration, which can be adapted to provide a customized configuration by the system administrator. It falls to the component developer to provide clear documentation explaining the baseline configuration and the effect of altering it. It falls to the system administrator to read this documentation and consider carefully the impact of making changes to the RBS configuration.

19.3.4 Managing COM+ Security

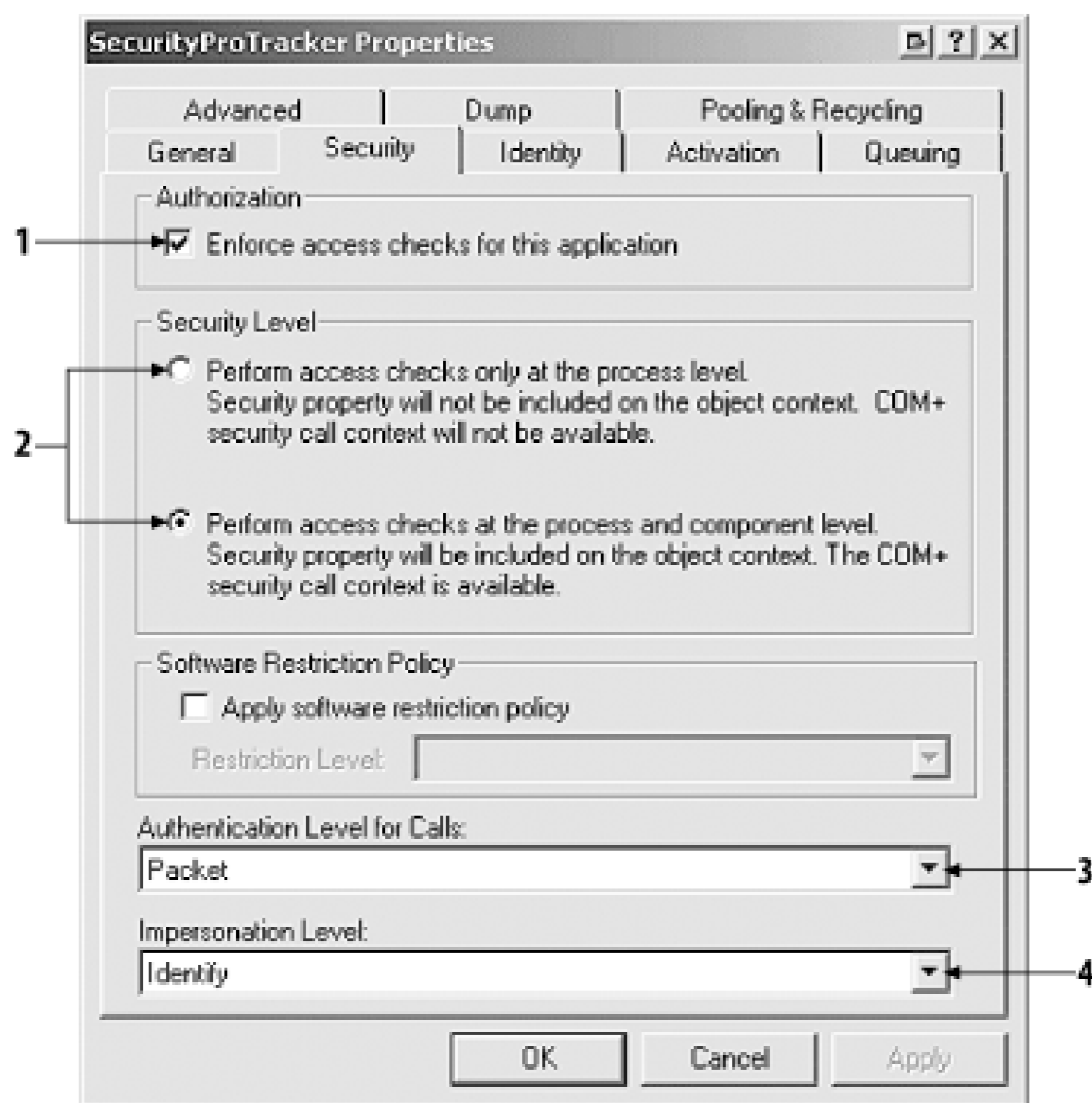
In the final section of this chapter, we will show you how the configuration options supported by the Component Services tool equate to the .NET attributes we used in [Section 19.2.3](#).

19.3.4.1 Managing the application

[Figure 19-14](#) illustrates the Security properties for our COM+ application example, which you can view by right-clicking on the application icon, choosing the Properties menu item, and clicking on the Security tab. These properties specify the settings for the security of the entire COM+ application,

which applies to all of the components in the application. The options displayed in this window are equivalent to the functionality provided by the `ApplicationAccessControl` attribute.

Figure 19-14. The Security Properties Page for the SecurityProTracker Application



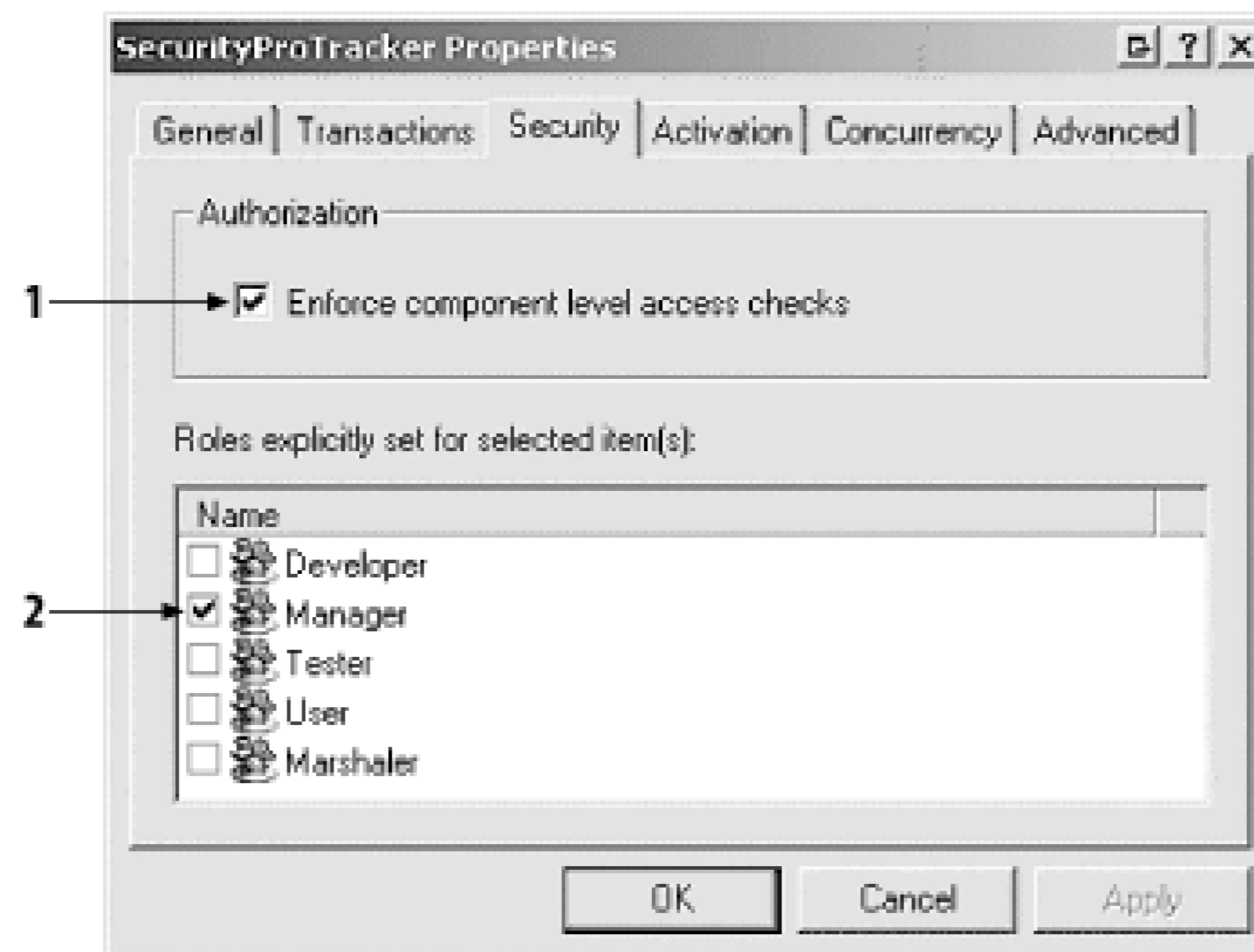
The Enforce access checks for this application check box (indicated by the number 1 in [Figure 19-14](#)), enabling and disabling RBS and PAS at the application level, equivalent to the `Value` property; bear in mind that access controls must also be enabled individually for the components within the application. The Security Level radio buttons (indicated by the number 2 in [Figure 19-14](#)) specify whether PAS or PAS and RBS will be enabled for the application; this is equivalent to the `AccessChecksLevel` property.

The Authentication Level for Calls and Impersonation Level combination boxes (indicated by the numbers 3 and 4 in [Figure 19-14](#)) are equivalent to the `Authentication` and `ImpersonationLevel` properties; the drop-down list contains the definitions represented by the `AuthenticationOption` and `ImpersonationLevelOption` enumerations.

19.3.4.2 Managing the component

[Figure 19-15](#) illustrates the security properties for our serviced component, which you can obtain by right-clicking on the component icon, choosing the Properties menu item, and clicking on the Security tab. These properties specify the security settings for the component, and each component within a COM+ application can be configured individually.

Figure 19-15. The Security Properties page for the component



The Enforce component level access checks (indicated by the number 1 in [Figure 19-15](#)) enables RBS for the selected component and is equivalent to the `ComponentAccessControl` attribute that we applied to the `SecurityProTracker` class. If this checkbox is not checked, then RBS will not be applied to this component, irrespective of the application-wide setting.

[[Team LiB](#)]

[\[Team LiB \]](#)

Chapter 20. The Event Log Service

The Event Log Service (ELS) is a component of the Windows operating system used to record and monitor significant events in a common and unified way. We begin this chapter by discussing the overall design of the ELS and then proceed to demonstrate how to use the ELS in your programs. The .NET Framework does not support the full functionality of the ELS, and while complete coverage of the ELS functionality is beyond the scope of this book, we do provide enough detail for you to understand the .NET ELS support; consult the Windows API documentation for complete coverage of the ELS.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

20.1 The Event Log Service Explained

In this section, we discuss the overall structure of the ELS and introduce three important elements of its design: event logs, event sources, and events. The principal security aspect of the ELS is as the means to audit Windows security events, for which .NET unfortunately does not provide good support. However, the ELS is an important tool that you should use within your own projects to record important application events.

20.1.1 Event Logs

The ELS defines a common format for storing events persistently in *log files*. The details of the file format and the location of the file on disk are not important to the programmer; the ELS acts as a broker between the source of an event (an application, device driver, etc.) and the log file in which the event is written. The ELS supports three default event logs, each of which has a specific purpose:

The *System* log

The System log records significant events that occur within components of the operating system (for example, a failure within a device driver).

The *Application* log

The Application log records events from applications (for example, an unexpected application failure).

The *Security* log

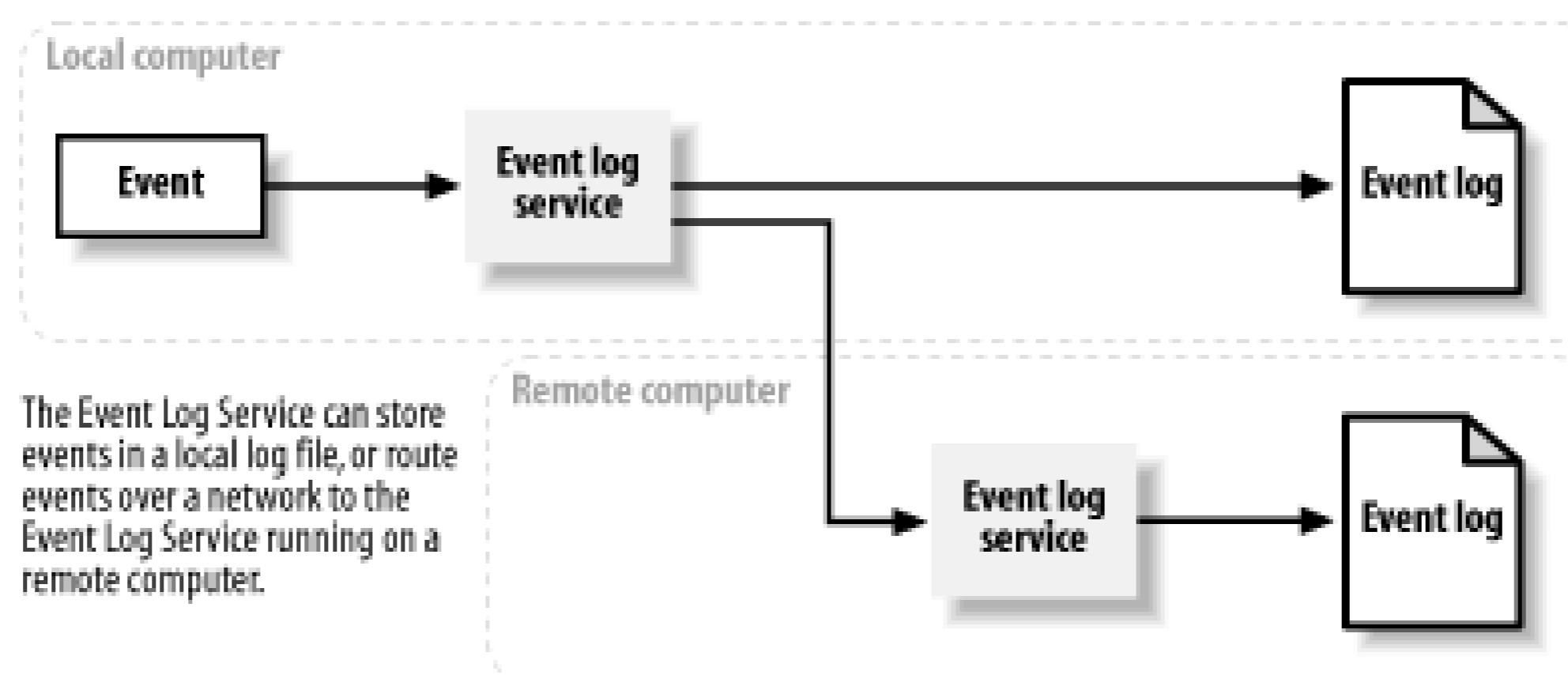
The Security log provides a record of audited security activity (for example, accessing a protected file).

Additional logs may supplement the three default logs, depending on the configuration of the Windows computer; for example, a computer configured as a Domain Name System (DNS) server will have a DNS Server log. Windows domain controllers will have two additional logs: Directory Service and File Replication Service. Programmers can also create custom logs that store events from only their applications or system components; we show you how to create custom logs in [Section 20.2.5](#) later in this chapter.

One of the most useful features of the ELS is the ability to route events over the network for storage by the ELS running on a remote computer, as illustrated in [Figure 20-1](#). This feature allows you to collate related events in a single location by designating a log server to store events on behalf of a group of computers, typically performing related tasks. This collation allows you to look for patterns of events that may not be visible if the events were stored locally on each computer. Event collation is especially useful for the Application log, where events that may not be significant on their own (for example, an application process terminates unexpectedly) may take on a greater meaning when seen alongside other events (perhaps the unexpected termination indicates a coordinated attack against a group of related computers).

Figure 20-1. Event sources can use the ELS to write events to logs that

are stored locally or on another computer



Access to the event logs is restricted based on the account under which an application is running. [Table 20-1](#) summarizes the restrictions for the Application and System logs. The `LocalSystem` account is used by Windows Services; consult the Windows documentation for more details.

Table 20-1. Account restrictions for the Application and System logs

Event Log	Account	Read events	Write events	Clear events
Application	<code>LocalSystem</code>			
	<code>Administrator</code>			
	Domain administrators			
	All other accounts			
System	<code>LocalSystem</code>			
	<code>Administrator</code>			
	Domain Administrators			
	All other accounts			

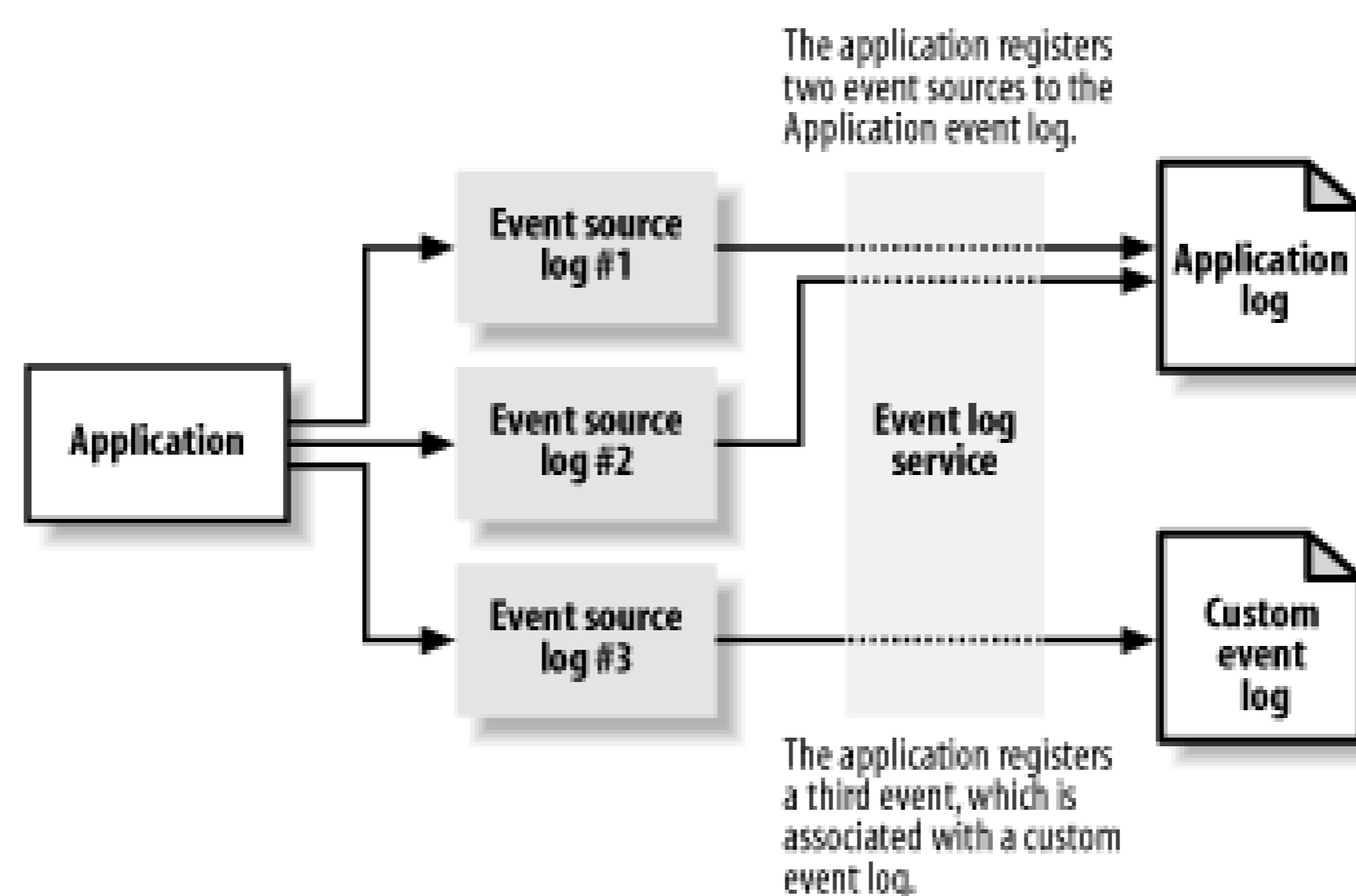
The Security log is a special case. Only operating system components can write events to this log based on a security audit policy. Applications can read and clear events from the Security log only when running under an account granted the "Manage Auditing and Security Log" user right; consult the Windows documentation for details of security and auditing configuration.

20.1.2 Event Sources

An application must register an *event source* before using the ELS to record events. The ELS writes the name of the event source to the event log as part of the event. Applications typically specify their name or the name of an application component as the event source value. We demonstrate how to create an event source in [Section 20.2.2](#).

Each event source is associated with a single event log, but applications can register and use more than one event source if they wish to write events to several logs; it is also possible to register multiple event sources that are associated with a single event log to create events with different event source values. [Figure 20-2](#) illustrates how an application can register event sources to be associated with several logs.

Figure 20-2. Applications can create several event sources in order to write events to more than one log file



Event sources are persistent, which means that once an application has registered an event source with the ELS, it is associated with the specified log file until the event source is deleted explicitly. We demonstrate how to create and delete event sources in [Section 20.2](#).

Registering an event source does not provide exclusive use, and because each event source is associated with a specific event log, you must ensure that the event sources you use are configured as you expect. [Figure 20-3](#) illustrates what can happen if two applications write events using the same event source. Application #1 has registered an event source and has specified that it should be associated with a custom log that is intended solely for that application's events. Application #2 uses the same event source name, and events intended for the Application event log are inadvertently routed to Application #1's custom log. See [Section 20.2.2](#) for details of how to check the configuration of an event source.

Figure 20-3. Registering an event source does not guarantee exclusive access to the registered name



Event sources provide functionality not supported in the .NET Framework; for the purposes of this book, an event source is simply a name that is associated with a specific log file. Consult the Windows API reference for full details of event source features.

20.1.3 Event Structure

The ELS uses a standardized structure to represent all events, irrespective of the log in which the event will be stored. [Figure 20-4](#) shows the part of the event structure exposed by the .NET Framework; we provide a brief explanation of each structural element in the following sections.

Figure 20-4. The structure of an event exposed by the .NET Framework

20.1.3.1 Event source name

This is the name of the event source used to log the event. As we explained in the previous section, applications can register multiple event sources.

20.1.3.2 Message

The message component is a human-readable description of the event, which may be of use when trying to determine the cause of a problem. We recommend that descriptions should be clearly phrased and to the point, and should not contain overly technical terms; consider using the binary data portion of an event to include detailed information about the reasons leading to the occurrence of an event.

20.1.3.3 Event type

The ELS defines a set of event types, which we summarize in [Table 20-2](#); the event type indicates the severity of the issue that the event represents.

Table 20-2. Event types

Event Type	Description
Error	Error events represent significant problems that the user or administrator should know about immediately. Errors usually indicate a loss of data or significant impairment of functionality.
Warning	A warning indicates a problem that is not significant in the short term, but which may result in future problems. For example, an application may indicate that a disk is low on space-the application can still write data to the disk, but if no action is taken, the disk will fill and the application will fail.
Information	Information events reflect infrequent operations that have completed successfully. For example, services may record that they have started successfully.
Success audit	Success audit events are security events that occur when an audited access attempt is successful. For example, a successful logon attempt is a success audit event.
Failure audit	Failure audit events are security events that occur when an audited access attempt fails. For example, a failed attempt to open a file is a failure audit event.

20.1.3.4 Event identifier and event category

The event identifier and category are application-specific numeric values. Applications can provide message files that provide human-readable explanations of the event, and the reasons leading to the event occurring; however, the .NET Framework does not support this functionality. Consult the Windows API for details of how to use message files.

20.1.3.5 Binary data

The event may contain binary data that is of use to someone trying to resolve the problem that caused this event to occur. The ELS does not interpret the data, and the content is dependent upon the application; there are no standardized formats or tools available to assist with the analysis of this data.

[\[Team LiB \]](#)

[[Team LiB](#)]

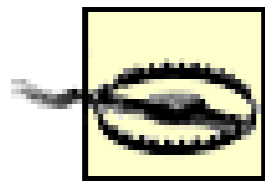
20.2 Programming the Event Log Service

The .NET Framework exposes ELS functionality through the `System.Diagnostics.EventLog` class; [Table 20-3](#) summarizes the public members of the `EventLog` class.

Table 20-3. Public members of the EventLog class

Member	Description
Properties	
<code>EnableRaisingEvents</code>	Gets or sets a value indicating whether the <code>EventLog</code> instance receives <code>EventWritten</code> event notifications. See Section 20.2.6 for details.
<code>Entries</code>	Returns the contents of the event log.
<code>Log</code>	Gets or sets the name of the event log to read from and write to.
<code>LogDisplayName</code>	Gets the user-friendly name of the event log.
<code>MachineName</code>	Gets or sets the name of the computer on which to read or write events.
<code>Source</code>	Gets or sets the name of the event source used to write events to the log.
Methods	
<code>Clear</code>	Removes all of the event entries from an event log.
<code>CreateEventSource</code>	Registers a new event source.
<code>Delete</code>	Deletes a specified event log from the ELS.
<code>DeleteEventSource</code>	Removes the registration for an event source.
<code>Exists</code>	Determines if a specified event log exists.
<code>GetEventLogs</code>	Obtains an <code>EventLog</code> array representing all of the event logs on the local or a remote computer. See Section 20.2.1 for details.
<code>LogNameFromSourceName</code>	Returns the name of the event log with which a specific event source is associated.
<code>SourceExists</code>	Determines if a specific event source has been registered with the ELS.
<code>WriteEntry</code>	Writes an event to the log. See Section 20.2.4 for details.
<code>EntryWritten</code>	An event that occurs when an event is written to an event log. See Section 20.2.6 for details.

In the following sections, we demonstrate how to use the `EventLog` class to program the ELS. The `EventLog` class does not expose all of the ELS functionality, and some advanced features are not available. You must use the unmanaged Windows API to gain access to the complete ELS feature set; see the Windows API documentation for details.



Your code must be granted the `EventLog` permission in order to access the Event Log Service; see [Chapter 7](#) for details of how to use permissions.

Many of the examples that follow demonstrate how to program the ELS of a remote computer. This functionality requires that a trust relationship is established; consult the Windows documentation for details of how to perform this task.

20.2.1 Querying the Event Log System

The `EventLog` class defines members that query the ELS for information about event logs. The static `GetEventLogs` method returns all of the event logs on a specified computer, represented as an array of `EventLog` instances. The following statements demonstrate how to use this method to obtain a list of event log names on the local computer:

```
# C#

// get the event logs installed on the local machine
EventLog[] x_logs = EventLog.GetEventLogs( );

// run through the array of event logs and print out the names
foreach (EventLog x_log in x_logs) {
    Console.WriteLine("Log Name: {0}", x_log.Log);
}

# Visual Basic .NET

' get the event logs installed on the local machine
Dim x_logs( ) As EventLog = EventLog.GetEventLogs( )

' run through the array of event logs and print out the names
Dim x_log As EventLog
For Each x_log In x_logs
    Console.WriteLine("Log Name: {0}", x_log.Log)
Next
```

The output these statements produce depends on the configuration of your computer; our results are below, showing that our computer has only the three default logs:

```
Log Name: Application
Log Name: Security
Log Name: System
```

We can enumerate the event logs on another computer by using the overloaded version of the `GetEventLog` method, which accepts the computer name as an argument, as shown below for the computer called `FILESERVER`.

```
# C#

// get the event logs installed on another computer
EventLog[] x_logs = EventLog.GetEventLogs("FILESERVER");
```

```
# Visual Basic .NET
```

```
' get the event logs installed on another computer
Dim x_logs( ) As EventLog = EventLog.GetEventLogs("FILESERVER")
```

We can determine if a specific log exists by using the static `Exists` method; the following statements demonstrate how to test for the existence of a log called *MyEventLog*-see the [Section 20.2.5](#) for details of how to create and delete event logs:

```
# C#
```

```
bool x_log_exists = EventLog.Exists("MyEventLog");
Console.WriteLine("Log Exists: {0}", x_log_exists);
```

```
# Visual Basic .NET
```

```
Dim x_log_exists As Boolean = EventLog.Exists("MyEventLog")
Console.WriteLine("Log Exists: {0}", x_log_exists)
```

The following statements determine the existence of *MyEventLog* on the computer called *FILESERVER*, using the overloaded form of the `Exists` method:

```
# C#
```

```
bool x_log_exists = EventLog.Exists("MyEventLog", "FILESERVER");
Console.WriteLine("Log Exists: {0}", x_log_exists);
```

```
# Visual Basic .NET
```

```
Dim x_log_exists As Boolean = EventLog.Exists("MyEventLog", "FILESERVER")
Console.WriteLine("Log Exists: {0}", x_log_exists)
```

20.2.2 Using Event Sources

We register an event source with the static `CreateEventSource` method; this is an overloaded method with forms that register an event source locally or on another computer. The following statements demonstrate how to register an event source named *MyEventSource*, associated with the Application event log on the local computer and a second event source named *MyOtherEventSource* associated with the System event log on the *FILESERVER* computer:

```
# C#
```

```
// create "MyEventSource" on the local computer
EventLog.CreateEventSource("MyEventSource", "Application");
```

```
// create "MyOtherEventSource" on the FILESERVER computer
```

```

EventLog.CreateEventSource("MyOtherEventSource", "System", " FILESERVER ");

# Visual Basic .NET

' create "MyEventSource" on the local computer
EventLog.CreateEventSource("MyEventSource", "Application")

' create "MyOtherEventSource" on the FILESERVER computer
EventLog.CreateEventSource("MyOtherEventSource", "System", "FILESERVER")

```

If you do not specify an event log name as an argument to the `CreateEventSource` method (by using the empty string ""), the event source will be associated with the Application log. Event sources are persistent, as we explained in [Section 20.1.2](#) of this chapter; attempting to register an event source that already exists throws an instance of the `System.ArgumentException` exception class. We can check to see if an event source has been registered using the static `SourceExists` method; the following statements demonstrate how to use this method to ensure that our two example event sources are registered and will create them if they are not:

```

# C#

// check to see if we need to create the local event source
if (!EventLog.SourceExists("MyEventSource")) {
    // create "MyEventSource" on the local computer
    EventLog.CreateEventSource("MyEventSource", "Application");
}

// check to see if we need to create the remote event source
if (!EventLog.SourceExists("MyOtherEventSource", "FILESERVER")) {
    // create "MyOtherEventSource" on the FILESERVER computer
    EventLog.CreateEventSource("MyOtherEventSource", "System", "FILESERVER");
}

# Visual Basic .NET

' check to see if we need to create the local event source
If Not EventLog.SourceExists("MyEventSource") Then
    ' create "MyEventSource" on the local computer
    EventLog.CreateEventSource("MyEventSource", "Application")
End If

' check to see if we need to create the remote event source
If Not EventLog.SourceExists("MyOtherEventSource", "FILESERVER") Then
    ' create "MyOtherEventSource" on the FILESERVER computer
    EventLog.CreateEventSource("MyOtherEventSource", "System", "FILESERVER")
End If

```

The static `LogNameFromSourceName` method determines the log with which an event source is associated, as shown by the following statements:

```

# C#

```

```

// obtain the name of the log associated with the local event source
string x_log_name = EventLog.LogNameFromSourceName("MyEventSource", ".");
// write out the log name
Console.WriteLine(x_log_name);

// obtain the name of the log associated with the remote event source
x_log_name = EventLog.LogNameFromSourceName("MyOtherEventSource", "FILESERVER");
// write out the log name
Console.WriteLine(x_log_name);

# Visual Basic .NET

' obtain the name of the log associated with the local event source
Dim x_log_name As String = EventLog.LogNameFromSourceName("MyEventSource", ".")
' write out the log name
Console.WriteLine(x_log_name)

' obtain the name of the log associated with the remote event source
x_log_name = EventLog.LogNameFromSourceName("MyOtherEventSource", "FILESERVER")
' write out the log name
Console.WriteLine(x_log_name)

```

Notice that unlike the other methods covered in this section, the `LogNameFromSourceName` method does not have separate overridden forms for dealing with local and remote computers; the local machine is specified by a period.

We can remove an event source registration by using the overloaded `DeleteEventSource` method. The following statements demonstrate how to remove the event sources we created earlier:

```

# C#

// delete the local event source
EventLog.DeleteEventSource("MyEventSource");

// delete the remote event source
EventLog.DeleteEventSource("MyOtherEventSource", "FILESERVER");

# Visual Basic .NET

' delete the local event source
EventLog.DeleteEventSource("MyEventSource")

' delete the remote event source
EventLog.DeleteEventSource("MyOtherEventSource", "FILESERVER")

```

Attempting to delete an event source that is not registered (or whose registration has already been removed) will throw an instance of `System.ArgumentException`.

20.2.3 Reading Event Logs

The first step towards reading the contents of an event log is to create a new instance of the

`EventLog` class, specifying the name of the log we want to read from as a constructor argument. The following statements demonstrate how to create new instances of the `EventLog` that refer to the local Application log and the System log on the *FILESERVER* computer:

```
# C#

// create a new instance that refers to the local Application log
EventLog x_local_log = new EventLog("Application");

// create a new instance which refers to the System log on FILESERVER
EventLog x_remote_log = new EventLog("System", "FILESERVER");

# Visual Basic .NET

' create a new instance that refers to the local Application log
Dim x_local_log As EventLog = New EventLog("Application")

' create a new instance which refers to the System log on FILESERVER
Dim x_remote_log As EventLog = New EventLog("System", "FILESERVER")
```

Once we have created an instance of the `EventLog` class, we can call the `Entries` property, which returns an instance of the `EventLogEntryCollection` class, containing the events in the log. [Table 20-4](#) summarizes the public methods of the `EventLogEntryCollection` class, which implements the `System.Collections.ICollection` and `System.Collections.IEnumerable` interfaces.

Table 20-4. Public members of the `EventLogEntryCollection` class

Member	Description
Properties	
<code>Count</code>	Returns the number of events in the log.
<code>Item</code>	Returns a specific event in the log, represented by the <code>EventLogEntry</code> class. For C#, this is the class indexer.
Methods	
<code>CopyTo</code>	Copies the events in the event log into an <code>EventLogEntry</code> array.

The `EventLogEntryCollection` class represents the collection of events stored in a log, each of which is represented by an instance of the `EventLogEntry` class. [Table 20-5](#) summarizes the public members of the `EventLogEntry` class; not all of the properties relate to ELS functionality supported by the .NET `EventLog` class. Consult the Windows API documentation for details of advanced ELS functionality.

Table 20-5. Public members of the `EventLogEntry` class

Property	Description
<code>Category</code>	Gets the category description associated with the event. Consult the Windows API documentation for details.
<code>CategoryNumber</code>	Gets the application-specific category value for the event.
<code>Data</code>	Gets the binary data associated with the event.
<code>EntryType</code>	Gets the event type, expressed as a value from the <code>EventLogEntryType</code> enumeration.
<code>EventID</code>	Gets the application-specific identifier for the event.
<code>Index</code>	Gets the position of this event in the event log.
<code>MachineName</code>	Gets the name of the computer on which the event was generated.
<code>Message</code>	Gets the human-readable message associated with this event.
<code>ReplacementStrings</code>	Gets the replacement strings associated with the event. Consult the Windows API documentation for details.
<code>Source</code>	Gets the name of the event source used to write the event.
<code>TimeGenerated</code>	Gets the time at which the event was created.
<code>TimeWritten</code>	Gets the time at which the event was written to the event log.
<code>UserName</code>	Gets the name of the user responsible for generating the event.

The following statements demonstrate how to use the `EventLog.Entries` property to obtain an `EventLogEntryCollection` for the Application log and print out details of the events using the `EventLogEntry` class:

```
# C#

// create a new instance that refers to the local Application log
EventLog x_local_log = new EventLog("Application");

// obtain a collection of the events through the Entries property
EventLogEntryCollection x_collection = x_local_log.Entries;

// write out the number of events in the log
Console.WriteLine("There are {0} entries in the event log", x_collection.Count);

// iterate through the entry collection and write out information
// about each event in turn
foreach (EventLogEntry x_entry in x_collection) {
    Console.WriteLine("Event Source: {0}, Event ID {1}, Event Message {2}",
        x_entry.Source, x_entry.EventID, x_entry.Message);
}

# Visual Basic .NET

' create a new instance that refers to the local Application log
```

```

Dim x_local_log As EventLog = New EventLog("Application")

' obtain a collection of the events through the Entries property
Dim x_collection As EventLogEntryCollection = x_local_log.Entries

' write out the number of events in the log
Console.WriteLine("There are {0} entries in the event log", x_collection.Count)

' iterate through the entry collection and write out information
' about each event in turn
Dim x_entry As EventLogEntry
For Each x_entry In x_collection
    Console.WriteLine("Event Source: {0}, Event ID {1}, Event Message {2}", _
        x_entry.Source, x_entry.EventID, x_entry.Message)
Next

```

20.2.4 Writing Events

The first step towards writing events is to create an instance of the `EventLog` class configured for the log and the computer that you wish to receive your event data; the following statements demonstrate how to use the overloaded `EventLog` constructor to create instances that refer to the Application log on the local and *FILESERVER* computers:

```

# C#

// create a new instance that refers to the local Application log
EventLog x_local_log = new EventLog("Application");

// create a new instance that refers to the remote Application Log
EventLog x_remote_log = new EventLog("Application", "FILESEVER");

# Visual Basic .NET

' create a new instance that refers to the local Application log
Dim x_local_log As EventLog = New EventLog("Application")

' create a new instance that refers to the remote Application Log
Dim x_remote_log As EventLog = New EventLog("Application", "FILESEVER")

```

We can achieve the same affect by using the default `EventLog` constructor in conjunction with the `MachineName` and `Log` properties, as illustrated by the following statements (had we wanted to refer to the local machine we would have set the `MachineName` property to be a period):

```

# C#

// create a new instance that refers to the local Application log
EventLog x_local_log = new EventLog( );
x_local_log.MachineName = "FILESERVER";
x_local_log.Log = "Application";

# Visual Basic .NET

```

```
' create a new instance that refers to the local Application log
Dim x_local_log As EventLog = New EventLog( )
x_local_log.MachineName = "FILESERVER"
x_local_log.Log = "Application"
```

Having created the `EventLog` instance, we must specify the event source that we want to use when writing an event; we do this with the `Source` property, as the following statements show:

```
# C#

// create a new instance that refers to the local Application log
EventLog x_local_log = new EventLog("Application");

// specify the event source that we'll use to record events
x_local_log.Source = "MyEventSource";

# Visual Basic .NET

' create a new instance that refers to the local Application log
Dim x_local_log As EventLog = New EventLog("Application")

' specify the event source that we'll use to record events
x_local_log.Source = "MyEventSource"
```

Once we have specified the computer, the event log, and the event source to use, we can use the `WriteEntry` method to write an event to the log; this method is overloaded with 10 different versions, which allows you to log an event with increasing levels of detail. The simplest version of the `WriteEntry` method accepts only the human-readable message element, whereas the most complex version accepts values for all of the event structure elements we described in [Section 20.1.3](#). In the following statements, we write an error message to log an unexpected process-termination event to the Application log on the local computer:

```
# C#

// create a new instance that refers to the local Application log
EventLog x_local_log = new EventLog( );
x_local_log.MachineName = ".";
x_local_log.Log = "Application";

// specify the event source that we'll use to record events
x_local_log.Source = "MyEventSource";

// define the binary data that will assist in debugging the problem
byte[] x_debugging_data = new byte[] {0xCA, 0xFE};

// write the event
x_local_log.WriteEntry(
    "MyApplication exited unexpectedly", // this is the event message
    EventLogEntryType.Error,           // specify an error message
    100,                                // this is the application-specific ID
    200,                                // this is the application-specific category
    x_debugging_data // this is the binary data
```



```

);

# Visual Basic .NET

' create a new instance that refers to the local Application log
Dim x_local_log As EventLog = New EventLog( )
x_local_log.MachineName = "."
x_local_log.Log = "Application"

' specify the event source that we'll use to record events
x_local_log.Source = "MyEventSource"

' define the binary data that will assist in debugging the problem
Dim x_debugging_data( ) As Byte = New Byte( ) {&HCA, &HFE}

' write the event
x_local_log.WriteEntry( _
    "MyApplication exited unexpectedly", _
    EventLogEntryType.Error, _
    100, _
    200, _
    x_debugging_data _
)

```

We specify the type of event we want to write by using a value from the `System.Diagnostics.EventLogEntryType` enumeration; [Table 20-6](#) lists the defined values, which correspond to the event types detailed in [Table 20-2](#).

Table 20-6. EventLogEntryType enumeration values

Value	Description
Error	Represents an error event
Warning	Represents a warning event
Information	Represents an informational event
SuccessAudit	Represents a successful attempt to access an audited resource
FailureAudit	Represents an unsuccessful attempt to access an audited resource

20.2.5 Using Custom Event Logs

You can create custom event logs by specifying the name you wish to use with the `Log` property of the `EventLog` class. When you write the first event to the log with the `WriteEntry` method, the ELS will create the custom log automatically. The event source specified by the `Source` property will be registered and associated with the new custom log. The following statements demonstrate how to create the custom log *MyCustomEventLog*; ELS does not create the custom log until the `WriteEntry` statement is executed:

```

# C#

// create a new instance that refers to the custom log
EventLog x_local_log = new EventLog( );
x_local_log.MachineName = ".";
x_local_log.Log = "MyCustomEventLog";

// specify the event source that we'll use to record events
x_local_log.Source = "MyCustomEventSource";

// write a log entry - the custom log will not be created until this point
x_local_log.WriteEntry("MyApplication exited unexpectedly");

# Visual Basic .NET

' create a new instance that refers to the custom log
Dim x_local_log As EventLog = New EventLog( )
x_local_log.MachineName = "."
x_local_log.Log = "MyCustomEventLog"

' specify the event source that we'll use to record events
x_local_log.Source = "MyCustomEventSource"

' write a log entry - the custom log will not be created until this point
x_local_log.WriteEntry("MyApplication exited unexpectedly")

```

Custom logs can be deleted with the `Delete` method defined by the `EventLog` class, as demonstrated by the following statements:

```

# C#

// delete the custom event log
EventLog.Delete("MyCustomEventLog");

# Visual Basic .NET

' delete the custom event log
EventLog.Delete("MyCustomEventLog")

```

Use the `Delete` method with caution; the event log will be removed and all of the entries and event sources will be lost permanently. An instance of the `System.InvalidOperationException` exception is thrown by the `Delete` method if you attempt to delete a log that does not exist.

The `EventLog.Delete` method will delete the Application and System default event logs without warning. Deleting these logs can cause applications to behave erratically.

20.2.6 Monitoring Event Logs

The `EventLog` class provides an event mechanism that allows us to monitor an event log. In this

section, we demonstrate how to use this feature, which relies on the .NET event delegate mechanism; consult the .NET documentation if you are unfamiliar with .NET delegates.

We can register for event notification by adding an instance of the `EntryWrittenEventHandler` delegate to the `EntryWritten` event defined by the `EventLog` class, as demonstrated by the statements below. The delegate method signature defines the `EntryWrittenEventArgs` class, which makes the event that has been written available as an instance of `EventLogEntry` accessible through the `Entry` property:

```
# C#

using System;
using System.Diagnostics;

public class EventLogMonitor {

    public static void Main( ) {

        // create a new instance of the EventLog class
        EventLog x_local_log = new EventLog("MyCustomEventLog");

        // create a delegate to process events from the ELS
        EntryWrittenEventHandler x_handler
            = new EntryWrittenEventHandler(MyOnEntryWrittenMethod);

        // add the delegate to the EventLog event
        x_local_log.EntryWritten += x_handler;

        // enable event processing
        x_local_log.EnableRaisingEvents = true;

        // wait to read a lone from the console - just to stop
        // the application from exiting
        Console.ReadLine( );
    }

    public static void MyOnEntryWrittenMethod(object p_source,
        EntryWrittenEventArgs p_args) {

        // extract the event object from the event arguments
        EventLogEntry x_entry = p_args.Entry;
        // write out the event details
        Console.WriteLine("Event Source: {0}, Event ID {1}, Event Message {2}",
            x_entry.Source, x_entry.EventID, x_entry.Message);
    }
}

# Visual Basic .NET

Imports System.Diagnostics
```

```

Class EventLogMonitor

    Shared Sub Main( )

        ' create a new instance of the EventLog class
        Dim x_local_log As EventLog = New EventLog("MyCustomEventLog")

        ' create a delegate to process events from the ELS
        AddHandler x_local_log.EntryWritten, AddressOf MyOnEntryWrittenMethod

        ' enable event processing
        x_local_log.EnableRaisingEvents = True

        ' wait to read a lone from the console - just to stop
        ' the application from exiting
        Console.ReadLine( )
    End Sub

    Public Shared Sub MyOnEntryWrittenMethod(ByVal p_source As Object, _
        ByVal p_args As EntryWrittenEventArgs)

        ' extract the event object from the event arguments
        Dim x_entry As EventLogEntry = p_args.Entry
        ' write out the event details
        Console.WriteLine("Event Source: {0}, Event ID {1}, Event Message {2}", _
            x_entry.Source, x_entry.EventID, x_entry.Message)

    End Sub
End Class

```

Events will not be raised until the value of the `EnableRaisingEvents` property is set to `true` (C#) or `True` (Visual Basic .NET). The `EventWritten` event is raised five seconds after an entry has been added to the monitored event log; if several events have been written during the five-second period, only the last event will be signaled via the delegate.

The `EventLog` class does not support monitoring event logs managed by other computers-only local logs can be monitored.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Part V: API Quick Reference

Provides a quick reference to all types defined in the security-related namespaces of the .NET Framework base class library:

[Chapter 23](#), `System.Security`

[Chapter 24](#), `System.Security.Cryptography`

[Chapter 25](#), `System.Security.Cryptography.X509Certificates`

[Chapter 26](#), `System.Security.Cryptography.Xml`

[Chapter 27](#), `System.Security.Permissions`

[Chapter 28](#), `System.Security.Policy`

[Chapter 29](#), `System.Security.Principal`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 21. How to Use This Quick Reference

The quick-reference section that follows packs a lot of information into a small space. This introduction explains how to get the most out of that information. It describes how the quick reference is organized and how to read the individual quick-reference entries.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

21.1 Finding a Quick-Reference Entry

The quick reference is organized into chapters, one per namespace. Each chapter begins with an overview of the namespace and includes a hierarchy diagram for the types (classes, interfaces, enumerations, delegates, and structs) in the namespace. Following the overview are quick-reference entries for all the types in the namespace.

[Figure 21-1](#) is a sample diagram showing the notation used to depict hierarchies and their relationships. Abstract classes are shown as slanted rectangles, and sealed classes as octagonal rectangles. Inheritance is shown as a solid line from the subtype, ending with a hollow triangle that points to the supertype. There are two notations that indicate interface implementation. The lollipop notation is used most of the time because it is easier to read. In some cases, especially where many types implement a given interface, the shaded box notation with a dashed line is used instead.

Figure 21-1. Class hierarchy notation

Important relationships between types (associations) are shown with a dashed line ending with an arrow. The figures don't show every possible association, but only those considered most important to developers.

Entries are organized alphabetically by type and namespace, so that related types are grouped near each other. Thus, in order to look up a quick-reference entry for a particular type, you must also know the name of the namespace that contains that type. Usually, the namespace is obvious from the context, and you should have no trouble looking up the quick-reference entry you want.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

21.2 Reading a Quick-Reference Entry

Each quick-reference entry contains quite a bit of information. The sections that follow describe the structure of a quick-reference entry, explaining what information is available, where it is found, and what it means. While reading the descriptions that follow, you will find it helpful to flip through the reference section itself to find examples of the features being described.

21.2.1 Type Name, Namespace, Assembly, Type Category, and Flags

Each quick-reference entry begins with a four-part title that specifies the name, namespace (followed by the assembly in parentheses), and type category of the type. It may also specify various additional flags that describe the type. The type name appears in bold at the upper left of the title. The namespace and assembly appear, in smaller print, in the lower left, below the type name.

In the upper-right corner of the title, you may find a list of flags that describe the type. The possible flags and their meanings are as follows:

ECMA

The type is part of the ECMA CLI specification.

Serializable

The type, or a supertype, implements `System.Runtime.Serialization.ISerializable` or has been flagged with the `System.Serializable` attribute.

Marshal by reference

This class, or a superclass, derives from `System.MarshalByRefObject`.

Context bound

This class, or a superclass, derives from `System.ContextBoundObject`.

Disposable

The type implements the `System.IDisposable` interface.

Flag

The enumeration is marked with the `System.FlagsAttribute`.

The lower-right portion of the title indicates the **type** category of the type (`class`, `delegate`, `enum`, `interface`, or `struct`). The class category may include modifiers, such as `sealed` or `abstract`.

21.2.2 Description

The title of each quick-reference entry is followed by a short description of the most important features of the type. This description may be anywhere from a couple of sentences to several paragraphs long.

21.2.3 Synopsis

The most important part of every quick-reference entry is the synopsis, which follows the title and description. The synopsis for a type looks a lot like its source code, except that the member bodies are omitted, and some additional annotations are added. If you know C# syntax, you can read the type synopsis.

The first line of the synopsis contains information about the type itself. It begins with a list of type modifiers, such as `abstract` and `sealed`. These modifiers are followed by the `class`, `delegate`, `enum`, `interface`, or `struct` keyword, and then by the name of the type. The type name may be followed by a colon (:) and a supertype or interfaces that the type implements.

The type-definition line is followed by a list of the members that the type defines. This list includes only those members that are explicitly declared in the type, are overridden from a base class, or are implementations of an interface member. Members that are simply inherited from a base class aren't shown; you will need to look up the base class definition to find those members. Once again, if you understand basic C# syntax, you should have no trouble making sense of these lines. The listing for each member includes the modifiers, type, and name of the member. For methods, the synopsis also includes the type and name of each method parameter. The member names are in boldface, so it is easy to scan the list of members looking for the one you want. The names of method parameters are in italics to indicate that they aren't to be used literally. The member listings are printed on alternating gray and white backgrounds to keep them visually separate.

21.2.3.1 Member availability and flags

Each member listing is a single line that defines the API for that member. These listings use C# syntax, so their meaning is immediately clear to any C# programmer. There is some auxiliary information associated with each member synopsis, however, that requires explanation.

The area to the right of the member synopsis displays a variety of flags that provide additional information about the member. Some of these flags indicate additional specification details that don't appear in the member API itself.

The following flags may be displayed to the right of a member synopsis:

Overrides

Indicates that a method overrides a method in one of its supertypes. The flag is followed by the name of the supertype that the method overrides.

Implements

Indicates that a method implements a method in an interface. The flag is followed by the name of the interface that is implemented.

=

For enumeration fields and constant fields, this flag is followed by the constant value of the field. Only constants of primitive and `String` types and constants with the value `null` are displayed. Some constant values are specification details, while others are implementation details. The reason symbolic constants are defined, however, is so you can write code that

doesn't directly rely on the constant value. Use this flag to help you understand the type, but don't rely on the constant values in your own programs.

21.2.3.2 Functional grouping of members

Within a type synopsis, the members aren't listed in strict alphabetical order. Instead, they are broken into functional groups and listed alphabetically within each group. Constructors, events, fields, methods, and properties are all listed separately. Instance methods are kept separate from static (class) methods. Public members are listed separately from protected members. Grouping members by category breaks a type down into smaller, more comprehensible segments, making the type easier to understand. This grouping also makes it easier for you to find a desired member.

Functional groups are separated from one another in a type synopsis with C# comments, such as `// Public Constructors`, `// Protected Instance Properties`, and `// Events`. The various functional categories are as follows (in the order in which they appear in a type synopsis):

Constructors

Displays the constructors for the type. Public constructors and protected constructors are displayed separately in subgroupings. If a type defines no constructor at all, the C# compiler adds a default no-argument constructor that is displayed here. If a type defines only private constructors, it can't be instantiated, so no constructor appears. Constructors are listed first, because the first thing you do with most types is instantiate them by calling a constructor.

Fields

Displays all fields defined by the type, including constants. Public and protected fields are displayed in separate subgroupings. Fields are listed here, near the top of the synopsis, because constant values are often used throughout the type as legal values for method parameters and return values.

Properties

Lists all the properties of the type, breaking them down into subgroupings for public and protected static properties and public and protected instance properties. After the property name, its accessors (`get` or `set`) are shown.

Static Methods

Lists the static methods (class methods) of the type, broken into subgroupings for public static methods and protected static methods.

Public Instance Methods

Contains all public instance methods.

Protected Instance Methods

Contains all protected instance methods.

21.2.4 Class Hierarchy

For any type that has a nontrivial inheritance hierarchy, the synopsis is followed by a Hierarchy section. This section lists all the supertypes of the type, as well as any interfaces implemented by those supertypes. It also lists any interfaces implemented by an interface. In the hierarchy listing,

arrows indicate supertype-to-subtype relationships, while the interfaces implemented by a type follow the type name in parentheses. For example, the following hierarchy indicates that `SomeClass` implements `IDisposable` and extends `MarshalByRefObject`, which itself extends `Object`:

```
System.Object → System.MarshalByRefObject      SomeClass(System.IDisposable)
```

If a type has subtypes, the Hierarchy section is followed by a Subtypes section that lists those subtypes. If an interface has implementations, the Hierarchy section is followed by an Implementations section that lists those implementations. While the Hierarchy section shows ancestors of the type, the Subtypes or Implementations section shows descendants.

21.2.5 Cross-References

The Hierarchy section of a quick-reference entry is followed by a number of optional cross-reference sections that indicate related types and methods that may be of interest. These sections are the following:

Passed To

This section lists all the members (from other types) that are passed an object of this type as an argument, including properties whose values can be set to this type. This is useful when you have an object of a given type and want to know where it can be used.

Returned By

This section lists all the members that return an object of this type, including properties whose values can take on this type. This is useful when you want to work with an object of this type but don't know how to obtain one.

Valid On

For attributes, this lists the attribute targets the attribute can be applied to.

Associated Events

For delegates, this lists the events it can handle.

21.2.6 A Note About Type Names

Throughout the quick reference, you'll notice that types are sometimes referred to by type name alone and at other times referred to by type name and namespace. If namespaces were always used, the type synopses would become long and hard to read. On the other hand, if namespaces were never used, it would sometimes be difficult to know what type was being referred to. The rules for including or omitting the namespace name are complex. They can be summarized approximately however, as follows:

- If the type name alone is ambiguous, the namespace name is always used.
- If the type is part of the `System` namespace or is a commonly used type such as `System.Collection.ICollection`, the namespace is omitted.
- If the type being referred to is part of the current namespace (and has a quick-reference entry

in the current chapter), the namespace is omitted. The namespace is also omitted if the type being referred to is part of a namespace that contains the current namespace.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Chapter 22. Converting from C# to VB Syntax

Although API Quick Reference information on all types and their members is shown using C# syntax, it is easy to mentally convert these entries to Visual Basic syntax. This chapter will provide the information you need to convert the C# documentation for each type into the syntax used by Visual Basic .NET programmers.



This chapter doesn't try to completely cover the syntax for each language element it discusses. Instead, it focuses on direct translation of the syntax of the types used in programming from C# to VB.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

22.1 General Considerations

The most evident difference between C# and VB syntax is that C# uses the semicolon (;) as a statement terminator, whereas VB uses a line break. As a result, a statement in C# can occupy multiple lines as long as it is terminated with a semicolon; a VB statement must occupy only a single line. Multiline statements in VB must appear with the VB line-continuation character (a space followed by an underscore) on all but the last line.

A second, and not quite so evident, difference is that C# is case-sensitive, whereas VB isn't. (Uniform casing for VB code is enforced by the Visual Studio environment, but it is by no means required.)

Finally, all types and their members have access modifiers that determine the type or member's accessibility. The keywords for these access modifiers are nearly identical in VB and C#, as [Table 22-1](#) shows.

Table 22-1. Access modifiers in C# and VB

C# keyword	VB keyword
<code>public</code>	<code>Public</code>
<code>private</code>	<code>Private</code>
<code>protected</code>	<code>Protected</code>
<code>internal</code>	<code>Friend</code>
<code>protected internal</code>	<code>Protected Friend</code>

[\[Team LiB \]](#)

[\[Team LiB \]](#)

22.2 Classes

C# uses the `class` statement along with opening and closing braces to indicate the beginning and end of a class definition. For example:

```
public class DataException : SystemException {
    // member definitions
}
```

In VB, a class definition is indicated by the `Class...End Class` construct:

```
Public Class DataException
    Inherits SystemException
    ' member definitions
End Class
```

In addition, C# classes can be marked as `abstract` or `sealed`; these correspond to the VB `MustInherit` and `NonInheritable` keywords, as shown in [Table 22-2](#).

Table 22-2. C# and equivalent VB class modifiers

C# keyword	VB keyword
<code>abstract</code>	<code>MustInherit</code>
<code>sealed</code>	<code>NonInheritable</code>

C# uses the colon to indicate either inheritance or interface implementation. Both the base class and the implemented interfaces are part of the `class` statement. For example:

```
public class DataSet : MarshalByValueComponent, IListSource,
    ISupportInitialize, ISerializable
```

In VB, a base class and any implemented interfaces are specified on separate lines immediately following the `class` statement. A class's base class is indicated by preceding its name with the `Inherits` keyword; any implemented interfaces are indicated by the `Implements` keyword. Hence, the previous definition of the `DataSet` class in C# would appear as follows in VB:

```
Public Class DataSet
    Inherits MarshalByValueComponent
    Implements IListSource, ISupportInitalize, ISerializable
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

22.3 Structures

C# uses the `struct` statement along with opening and closing braces to indicate the beginning and end of a structure definition. For example, `System.Windows.Forms.DataGridCell` is defined in C# as follows:

```
public struct DataGridCell {  
    // member definitions  
}
```

In VB, a structure definition is indicated by the `Structure...End Interface` construct:

```
Public Structure DataGridCell  
    ' member definitions  
End Structure
```

C# uses the colon with structures to indicate interface implementation. Any implemented interfaces are part of the `class` statement. In VB, any implemented interfaces are specified by an `Implements` statement on the line immediately following the `Structure` statement. However, none of the structures documented in the reference section of this book use interface inheritance.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

22.4 Interfaces

C# uses the `interface` statement along with opening and closing braces to indicate the beginning and end of an interface definition. For example:

```
public interface IDataAdapter {  
    // member definitions  
}
```

In VB, an interface definition is indicated by the `Interface...End Structure` construct:

```
Public Interface IDataAdapter  
    ' member definitions  
End Interface
```

C# uses the colon with interfaces to specify any implemented interfaces. For example:

```
public interface IDataReader : IDisposable, IDataRecord
```

In VB, any implemented interfaces are specified by an `Implements` statement on the line immediately following the `Interface` statement. The previous definition of `IDataReader` in C# would appear as follows in VB:

```
Public Interface IDataReader  
    Implements IDisposable, IDataRecord
```

[\[Team LiB \]](#)

[[Team LiB](#)]

22.5 Class, Structure, and Interface Members

Classes, structures, and interfaces can contain one or more fields, methods, properties, and events. This section discusses converting the C# syntax for each of these constructs to VB.

Note that .NET supports both static (or shared) members (which apply to the type as a whole, and typically don't require that an object of that type be instantiated) and instance members (which apply only to an instance of that type). Shared or static members are indicated by using the `static` keyword in C#. For example:

```
public static string ToString(long value);
```

The corresponding VB keyword is `Shared`, so the `FromResource` method, when converted to VB, has the following syntax:

```
Public Shared Function ToString(value As Long) As String
```

22.5.1 Fields

A field is a constant or a variable that is exposed as a publicly accessible member of a type. In C#, for example, the `Value` field of the `System.DBNull` class has the syntax:

```
public static readonly DBNull Value;
```

Note that C# indicates the data type of a field before the name of the field. (For C# data types and their VB equivalents, see [Table 22-3](#).) Also note that fields are frequently read-only. Constant fields, in fact, are always read-only. As a result, the use of the C# `readonly` keyword and the VB `ReadOnly` keyword with fields is quite common.

The syntax for the `Value` field in Visual Basic then becomes:

```
Public Shared ReadOnly Value As DBNull
```

22.5.2 Methods

In C#, all methods have a return value, which appears before the name of the function; in contrast, VB differentiates between function and subprocedures. C# functions without an explicit return value of `void`. For example, one of the overloads of the `DataSet` class's `AcceptChanges` method has the following syntax in C#:

```
public void AcceptChanges( );
```

C# methods that return `void` are expressed as subprocedures in VB. Here's the corresponding syntax of the `AcceptChanges` method:

```
Public Sub AcceptChanges( )
```

All C# methods other than those returning `void` are functions in VB. The function's return value appears in an `As` clause at the end of the function declaration. C# data types and their VB equivalents are shown in [Table 22-3](#). Methods that return arrays are indicated by adding braces (`[]`) to the return data type in C# and parentheses (`()`) to the return data type in VB.

For example, the `Copy` method of the `DataSet` class has the C# syntax:

```
public bool Copy( );
```

The VB equivalent is:

```
Public Function Copy( ) As Boolean
```

Table 22-3. C# data types and their VB equivalents

C# data type	VB data type
<code>bool</code>	<code>Boolean</code>
<code>byte</code>	<code>Byte</code>
<code>char</code>	<code>Char</code>
<code>decimal</code>	<code>Decimal</code>
<code>double</code>	<code>Double</code>
<code>float</code>	<code>Single</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>object</code>	<code>Object</code>
<code>sbyte</code>	<code>System.SByte</code>
<code>short</code>	<code>Short</code>
<code>string</code>	<code>String</code>
<code>System.Currency</code>	<code>Currency</code>
<code>System.DateTime</code>	<code>Date</code>
<code>uint</code>	<code>System.UInt32</code>
<code>ulong</code>	<code>System.UInt64</code>
<code>ushort</code>	<code>System.UInt16</code>
<code><class_name></code>	<code><class_name></code>
<code><delegate_name></code>	<code><delegate_name></code>
<code><interface_name></code>	<code><interface_name></code>
<code><structure_name></code>	<code><structure_name></code>

Method parameters in C# take the general form:

```
<data_type> <parameter_name>
```

In VB, method parameters take the form:

```
<parameter_name> As <data_type>
```

where *<data_type>* is any of the data types listed in [Table 22-3](#). If a parameter is an array, its data type is followed by braces in C# (e.g., `string[] Name`), while the parameter name is followed by parentheses in VB (e.g., `Name() As String`).

For example, one of the versions of the `DataTable` class's `Select` method has the following syntax in C#:

```
public DataRow[] Select(string filterExpression, string sort,
    DataRowState recordStates);
```

Its VB equivalent is:

```
Overloads Public Function Select(ByVal filterExpression As String, _
    ByVal sort As String, ByVal recordStates As DataRowState _
    ) As DataRow( )
```

VB allows methods to be called using either named or positional parameters. If named parameters are used, the parameter name must correspond to that shown in the documentation. For instance, `DataTable.Select` can be called as follows using named parameters:

```
dr = DataTable.Select(filterexpression:=flt, _
    sort:=sd, _
    recordstates:=DataRowState.CurrentRows)
```

C# also uses a number of object-oriented qualifiers with methods. These, and their VB equivalents, are shown in [Table 22-4](#).

Table 22-4. C# keywords used with methods and their VB equivalents

C# keyword	VB keyword
<code>abstract</code>	<code>MustOverride</code>
<code>override</code>	<code>Overrides</code>
<code>sealed</code>	<code>NotOverridable</code>
<code>virtual</code>	<code>Overridable</code>

In both C# and VB, constructors have a special syntax. In C#, constructors have the same name as the classes whose objects they instantiate and don't indicate a return value. For example, the default

constructor for the `SqlCommand` class is:

```
public SqlCommand( );
```

In VB, the constructor is represented by a call to a class's `New` subprocedure. The equivalent call to the `SqlCommand` class constructor in VB is:

```
Public Sub New( )
```

22.5.3 Properties

The `SqlCommand.CommandText` property provides a more or less typical example of a property definition using C# syntax:

```
public string CommandText {get; set;}
```

Like all C# type definitions, the property's data type precedes the property name. The `get;` and `set;` property accessors indicate that this is a read-write property. Read-only properties are indicated with a `get;` only, while write-only properties are indicated with a `set;` only.

The equivalent VB property definition is:

```
Public Property CommandText As String
```

Note that read/write properties aren't decorated with additional keywords in VB. Read-only properties, on the other hand, are indicated with the `ReadOnly` keyword in front of the `Property` keyword, while write-only properties have the `WriteOnly` keyword before the `Property` keyword.

Note that properties, like methods, can use the object-oriented modifiers listed in [Table 22-4](#).

22.5.4 Events

Events are declared in C# using the `event` keyword, which is followed by the delegate type returned by the event and the name of the event. For example, the `RowUpdated` event of the `SqlDataAdapter` class has the following syntax:

```
public event SqlRowUpdatedEventHandler RowUpdated;
```

The equivalent VB syntax is:

```
Public Event RowUpdated As SqlRowUpdatedEventHandler
```

In addition, the C# `event` and the VB `Event` keywords can be preceded by the object modifiers listed in [Table 22-4](#).

[\[Team LiB \]](#)

[\[Team LiB \]](#)

22.6 Delegates

The syntax for a delegate in C# closely follows the syntax for a method. The `delegate` statement is followed by the delegate's return type (or `void`, if there is none) and the delegate name. This in turn is followed by the delegate's parameter list, in which each parameter takes the form:

```
<parameter_type> <parameter_name>
```

For example:

```
public delegate void StateChangeEventHandler(object sender,  
    StateChangeEventArgs e);
```

In a VB `Delegate` statement, the `Delegate` keyword is followed by the `Sub` keyword (if the delegate returns a `void` in C#) or the `Function` keyword (if the delegate returns some other value). For example, in VB, the `StateChangeEventHandler` delegate has the following syntax:

```
Public Delegate Sub StateChangeEventHandler( _  
    ByVal sender As Object, ByVal e As StateChangeEventArgs)
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

22.7 Enumerations

C# uses the `enum` statement along with opening and closing braces to indicate the beginning and end of an enumeration definition. For example:

```
public enum CommandType {  
    // enumeration members  
}
```

In VB, an enumeration is defined by the `Enum...End Enum` construct. For example, the VB version of the `CommandType` enumeration declaration is:

```
Public Enum CommandType  
    ' enumeration members  
End Enum
```

In both C# and VB, the member listing consists of the name of the enumerated member and its value. These are identical in C# and VB, except that C# adds a comma to separate one member of the enumeration from another, whereas VB requires that they be on separate lines. For example, the full declaration of the `CommandType` enumeration in C# is:

```
public enum CommandType {  
    Text = 1,  
    StoredProcedure = 4,  
    TableDirect = 512  
}
```

The VB equivalent is:

```
Public Enum CommandType  
    Text = 1  
    StoredProcedure = 4  
    TableDirect = 512  
End Enum
```

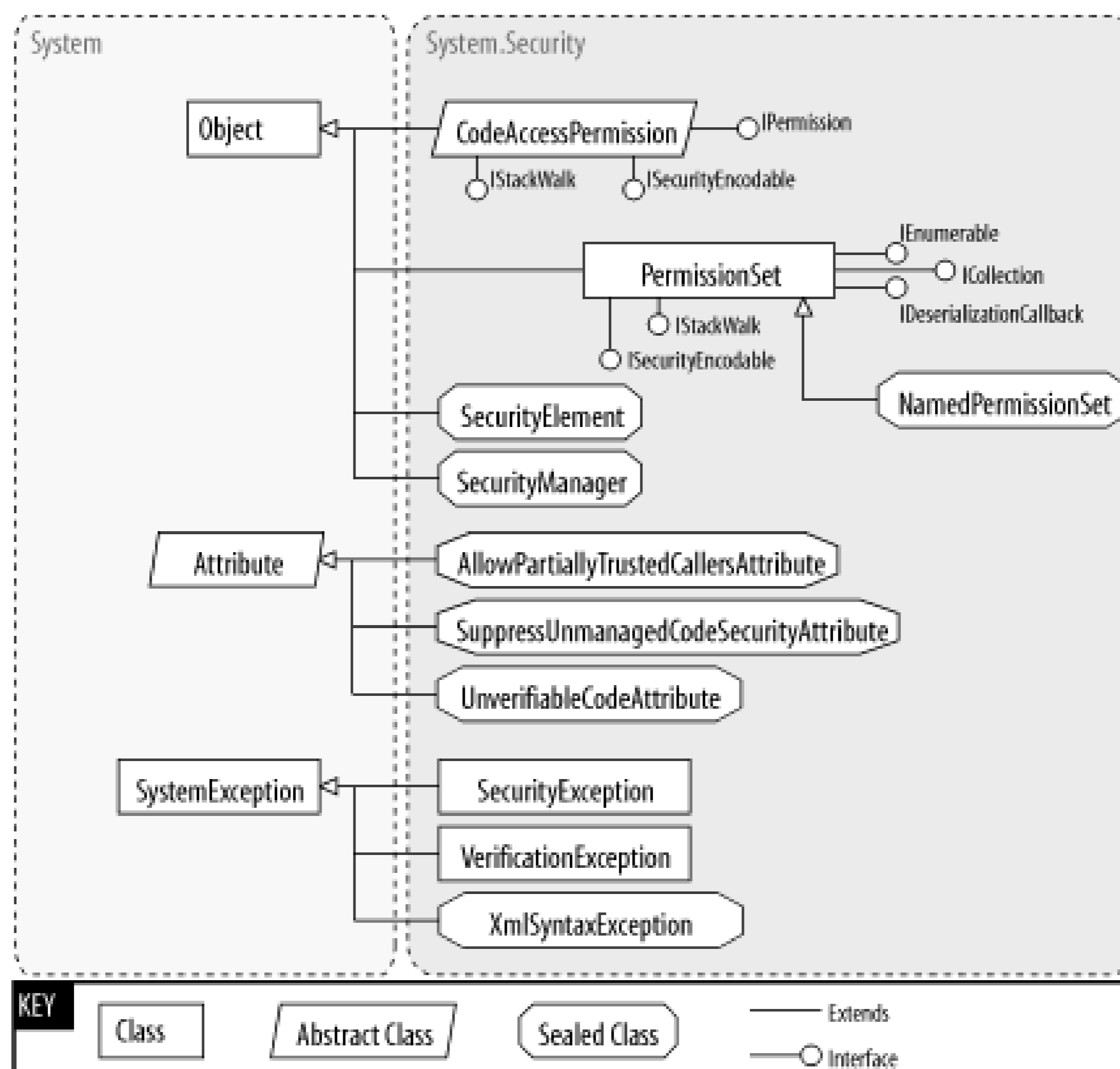
[\[Team LiB \]](#)

[Team LiB]

Chapter 23. The System.Security Namespace

The `System.Security` namespace contains the fundamental building blocks of .NET's code-access security framework. The classes contained in the `System.Security.Policy`, `System.Security.Permissions`, and `System.Security.Principal` namespaces extend, build on, and use the classes contained in the `System.Security` namespace to implement the security framework. [Figure 23-1](#) shows the types in this namespace.

Figure 23-1. The System.Security namespace



[Team LiB]

[\[Team LiB \]](#)

AllowPartiallyTrustedCallersAttribute

System.Security (mscorlib.dll) *sealed class*

```
public sealed class AllowPartiallyTrustedCallersAttribute : Attribute {  
    // Public Constructors  
    public AllowPartiallyTrustedCallersAttribute( );  
}
```

Applying the `AllowPartiallyTrustedCallersAttribute` attribute class at the assembly level of a strong named assembly allows partially trusted code to call the assembly's public and protected members. Without specifying the `AllowPartiallyTrustedCallersAttribute`, all public and protected members of a strong-named assembly are only accessible to fully trusted code (i.e., code granted the `FullTrust` permission set during policy resolution).

The restriction is implemented through a `LinkDemand` for the `FullTrust` permission set that is applied automatically by the compiler to every `public` and `protected` member of every `public` class in an assembly.

Hierarchy

`System.Object` `System.Attribute` `AllowPartiallyTrustedCallersAttribute`

Valid On

Assembly

[\[Team LiB \]](#)

[Team LiB]

CodeAccessPermission ECMA 1.0, serializable

System.Security (mscorlib.dll) *abstract class*

```

public abstract class CodeAccessPermission : IPermission, ISecurityEncodable, IStackWalk
// Protected Constructors
    protected CodeAccessPermission( );
// Public Static Methods
    public static void RevertAll( );
    public static void RevertAssert( );
    public static void RevertDeny( );
    public static void RevertPermitOnly( );
// Public Instance Methods
    public void Assert( );
// implements IStackWalk
    public abstract IPermission Copy( );
// implements IPermission
    public void Demand( );
// implements IStackWalk
    public void Deny( );
// implements IStackWalk
    public abstract void FromXml(SecurityElement elem);
// implements ISecurityEncodable
    public abstract IPermission Intersect(IPermission target);
// implements IPermission
    public abstract bool IsSubsetOf(IPermission target);
// implements IPermission
    public void PermitOnly( );
// implements IStackWalk
    public override string ToString( );
// overrides object
    public abstract SecurityElement ToXml( );
// implements ISecurityEncodable
    public virtual IPermission Union(IPermission other);
// implements IPermission
}

```

This abstract class provides the base class from which all code-access and identity permission classes are derived. It implements the `IPermission`, `IStackWalk`, and `ISecurityEncodable` interfaces, but only provides concrete implementations of the `IStackWalk` methods (`Demand()`, `Assert()`, `Deny()`, and `PermitOnly()`), leaving implementation of the other interface methods to the derived classes. The `staticRevertAll()`, `RevertAssert()`, `RevertDeny()`, and `RevertPermitOnly()` methods remove existing stack walk overrides of the appropriate type from the caller's stack frame.

Subclasses

Multiple types

[Team LiB]

[\[Team LiB \]](#)

I EvidenceFactory

System.Security (mscorlib.dll) *interface*

```
public interface IEvidenceFactory {  
    // Public Instance Properties  
    public Evidence Evidence {get; }  
}
```

The `System.Security.Policy.Evidence` collection available through the `Evidence` property of this interface contains the evidence assigned to an assembly or application domain by the runtime. This evidence is used by the runtime during policy resolution to determine the assembly or application domain's grant set.

Implemented By

`System.AppDomain`, `System.Reflection.Assembly`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

I Permission

ECMA 1.0

System.Security (mscorlib.dll)

interface

```
public interface IPermission : ISecurityEncodable {  
    // Public Instance Methods  
    public IPermission Copy( );  
    public void Demand( );  
    public IPermission Intersect(IPermission target);  
    public bool IsSubsetOf(IPermission target);  
    public IPermission Union(IPermission target);  
}
```

This `IPermission` interface defines the minimum set of methods that all permission classes must implement if they are to integrate with the code-access security framework of the .NET runtime.

`IPermission` defines the `Demand()` method, which is the most fundamental mechanism used to enforce code-access security. `Demand()` invokes a security demand for the implementing permission object. This ensures that calling code has the permission; otherwise, a `SecurityException` is thrown. All of the standard code-access and identity permissions implement the `IStackWalk` interface, which also defines a `Demand()` method. The `IStackWalk.Demand()` method initiates a stack-walk-based security demand, whereas `IPermission.Demand()` is intended for use by permission classes that do not use stack walks—for example, the `System.Security.Permissions.PrincipalPermission` class.

The exact behavior of the `Copy()`, `Intersect()`, `IsSubsetOf()`, and `Union()` methods vary based on the nature of the implementing permission class. `Copy()` returns a deep copy of the permission. `Intersect()` returns a new permission that represents the logical intersection of the current and a specified `IPermission`, whereas `Union()` returns the logical union of the two `IPermission` objects. `IsSubsetOf()` indicates whether the current `IPermission` represents a subset of the specified `IPermission`.

Implemented By

`CodeAccessPermission, System.Security.Permissions.PrincipalPermission`

Returned By

Multiple types

Passed To

```
CodeAccessPermission.{Intersect( ), IsSubsetOf( ), Union( )},  
System.Security.Permissions.PrincipalPermission.{Intersect( ), IsSubsetOf( ), Union( )},  
PermissionSet.{AddPermission( ), SetPermission( )}, SecurityManager.IsGranted( )
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

I SecurityEncodable

System.Security (mscorlib.dll)

interface

```
public interface ISecurityEncodable {  
    // Public Instance Methods  
    public void FromXml(SecurityElement e);  
    public SecurityElement ToXml( );  
}
```

The `ISecurityEncodable` interface defines methods to support converting security object state to and from XML representations.

Implemented By

Multiple types

[\[Team LiB \]](#)

[\[Team LiB \]](#)

I SecurityPolicyEncodable

System.Security (mscorlib.dll) *interface*

```
public interface ISecurityPolicyEncodable {  
    // Public Instance Methods  
    public void FromXml(SecurityElement e, System.Security.Policy.PolicyLevel level);  
    public SecurityElement ToXml(System.Security.Policy.PolicyLevel level);  
}
```

This interface, like `ISecurityEncodable`, defines methods to support converting security object state to and from XML representations. However, the `ToXml()` and `FromXml()` methods defined in `ISecurityPolicyEncodable` include a `System.Security.Policy.PolicyLevel` argument, allowing the implementing class to base the XML representation of its state on the policy level to which it belongs.

Implemented By

```
System.Security.Policy.{AllMembershipCondition,  
ApplicationDirectoryMembershipCondition, HashMembershipCondition,  
IMembershipCondition, PolicyStatement, PublisherMembershipCondition,  
SiteMembershipCondition, StrongNameMembershipCondition, UrlMembershipCondition,  
ZoneMembershipCondition}
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

I StackWalk

System.Security (mscorlib.dll) *interface*

```
public interface IStackWalk {
    // Public Instance Methods
    public void Assert( );
    public void Demand( );
    public void Deny( );
    public void PermitOnly( );
}
```

Permission classes that use stack walks to resolve security demands implement the `IStackWalk` interface. These include the `PermissionSet` class, as well as the `CodeAccessPermission` class, from which all standard code-access and identity permissions inherit.

`IStackWalk` defines the `Demand()` method, which duplicates the method of the same name defined in the `IPermission` interface. However, the `IStackWalk.Demand()` method initiates a stack-walk-based security demand, whereas `IPermission.Demand()` is not intended to use the call stack to resolve security demands. The `Assert()`, `Deny()`, and `PermitOnly()` methods annotate the call stack frame of the code from which they are executed, and affect the stack walk process initiated by a `Demand()` call.

`Assert()` allows code to vouch for the permissions of the callers above it on the call stack. During a stack walk, if a demanded permission matches, or is a subset of, the asserted permission, the stack walk terminates successfully without evaluating further stack frames. If the asserted permissions provide only a partial match of those demanded, the stack walk continues but only for the remaining unasserted permissions. Code must have been granted a permission in order to assert it, and the code must have the permission to assert defined by the `System.Security.Permissions.SecurityPermission` class.

`Deny()` has the opposite effect of `Assert()`. If the demanded permission matches, or is a subset of the denied permission, the stack walk is terminated, and the runtime throws a `SecurityException` to the code that initiated the `Demand()`.

`PermitOnly()` functions as an inverted `Deny()`. Instead of specifying the permissions that should terminate a stack walk, `PermitOnly()` specifies those permissions that permit the stack walk to continue unaffected. `PermitOnly()` provides a more concise way of denying large sets of permissions without the need to specify them all in a `Deny()` override.

Implemented By

`CodeAccessPermission`, `PermissionSet`

[\[Team LiB \]](#)

[Team LiB]

NamedPermissionSet

serializable

System.Security (mscorlib.dll)

sealed class

```

public sealed class NamedPermissionSet : PermissionSet {
// Public Constructors
    public NamedPermissionSet(NamedPermissionSet permSet);
    public NamedPermissionSet(string name);
    public NamedPermissionSet(string name, PermissionSet permSet);
    public NamedPermissionSet(string name, System.Security.Permissions.PermissionState s
// Public Instance Properties
    public string Description{set; get; }
    public string Name{set; get; }
// Public Instance Methods
    public NamedPermissionSet Copy(string name);
    public override PermissionSet Copy( );
// overrides PermissionSet
    public override void FromXml(SecurityElement et);
// overrides PermissionSet
    public override SecurityElement ToXml( );
// overrides PermissionSet
}

```

This subclass of `PermissionSet` allows the permission set to have a name and description assigned to it. `NamedPermissionSet` is used predominantly when working programmatically with security policy.

Hierarchy

```

System.Object      PermissionSet(ISecurityEncodable, System.Collections.ICollection,
System.Collections.IEnumerable, IStackWalk,
System.Runtime.Serialization.IDeserializationCallback)      NamedPermissionSet

```

Returned By

```

System.Security.Policy.PolicyLevel.{ChangeNamedPermissionSet( ), GetNamedPermissionSet(
RemoveNamedPermissionSet( )}

```

Passed To

```

System.Security.Policy.PolicyLevel.{AddNamedPermissionSet( ), RemoveNamedPermissionSet(

```

[Team LiB]

[Team LiB]

PermissionSet

ECMA 1.0, serializable

System.Security (mscorlib.dll)

class

```

public class PermissionSet : ISecurityEncodable, ICollection, IEnumerable, IStackWalk,
    System.Runtime.Serialization.IDeserializationCallback {
// Public Constructors
    public PermissionSet(PermissionSet permSet);
    public PermissionSet(System.Security.Permissions.PermissionState state);
// Public Instance Properties
    public virtual int Count{get;
// implements ICollection
    public virtual bool IsReadOnly{get; }
    public virtual bool IsSynchronized{get;
// implements ICollection
    public virtual object SyncRoot{get;
// implements ICollection
// Public Static Methods
    public static byte[ ] ConvertPermissionSet(string inFormat, byte[ ] inData, string
// Public Instance Methods
    public virtual IPermission AddPermission(IPermission perm);
    public virtual void Assert( );
// implements IStackWalk
    public bool ContainsNonCodeAccessPermissions( );
    public virtual PermissionSet Copy( );
    public virtual void CopyTo(Array array, int index);
// implements ICollection
    public virtual void Demand( );
// implements IStackWalk
    public virtual void Deny( );
// implements IStackWalk
    public virtual void FromXml(SecurityElement et);
// implements ISecurityEncodable
    public virtual IEnumerator GetEnumerator( );
// implements IEnumerable
    public virtual IPermission GetPermission(Type permClass);
    public virtual PermissionSet Intersect(PermissionSet other);
    public virtual bool IsEmpty( );
    public virtual bool IsSubsetOf(PermissionSet target);
    public virtual bool IsUnrestricted( );
    public virtual void PermitOnly( );
// implements IStackWalk
    public virtual IPermission RemovePermission(Type permClass);
    public virtual IPermission SetPermission(IPermission perm);
    public override string ToString( );

```

```
// overrides object
public virtual SecurityElement ToXml( );
// implements ISecurityEncodable
public virtual PermissionSet Union(PermissionSet other);
}
```

The `PermissionSet` class is a container for groups of `IPermission` objects. It implements the `IStackWal` and independently implements the `Copy()`, `Demand()`, `Intersect()`, `IsSubsetOf()`, and `Union()` which mirror those defined by the `IPermission` interface. This functionality allows a `PermissionSet` object to be treated as an individual `IPermission` object for imperative security operations, but the security operation is performed on all contained `IPermission` objects at once.

The contents of a `PermissionSet` are configured using the `AddPermission()`, `RemovePermission()`, `SetPermission()` methods. Only one of each permission type can be present in a `PermissionSet` at a time. If you call `AddPermission()`, if a permission of the same type already exists in the `PermissionSet`, the existing `IPermission` is replaced by a new `IPermission` whose value is calculated as the union of the existing and the new `IPermission` objects. Conversely, `SetPermission()` replaces the existing permission with the specified `IPermission`. The `GetPermission()` method returns a copy of the `IPermission` of the specified `System.Type` if one exists in the `PermissionSet`.

Subclasses

`NamedPermissionSet`

Returned By

`NamedPermissionSet.Copy()`, `System.Security.Permissions.PermissionSetAttribute.CreatePermissions()`, `System.Security.Policy.PermissionRequestEvidence.DeniedPermissions`, `OptionalPermissions`, `RequestedPermissions`, `System.Security.Policy.PolicyStatement.PermissionSet`, `SecurityManager.ResolvePolicy()`

Passed To

`System.AppDomain.DefineDynamicAssembly()`, `System.IO.IsolatedStorage.IsolatedStorage.GetPermissions()`, `System.Reflection.Emit.ConstructorBuilder.AddDeclarativeSecurity()`, `System.Reflection.Emit.MethodBuilder.AddDeclarativeSecurity()`, `System.Reflection.Emit.TypeBuilder.AddDeclarativeSecurity()`, `NamedPermissionSet.NamedPermissionSet()`, `System.Security.Policy.PermissionRequestEvidence.PermissionRequestEvidence()`, `System.Security.Policy.PolicyLevel.ChangeNamedPermissionSet()`, `System.Security.Policy.PolicyStatement.PolicyStatement()`, `SecurityManager.ResolvePolicy()`, `System.Xml.XmlSecureResolver.XmlSecureResolver()`

[Team LiB]

[\[Team LiB \]](#)

PolicyLevelType

serializable

System.Security (mscorlib.dll)

enum

```
public enum PolicyLevelType {  
    User = 0, Machine = 1, Enterprise = 2, AppDomain = 3  
}
```

This enumeration defines constants that represent the different levels of security policy configuration. `PolicyLevelType` is used when working with members of the `SecurityManager` class.

Hierarchy

```
System.Object → System.ValueType      System.Enum(System.IComparable,  
System.IFormattable, System.IConvertible)      PolicyLevelType
```

Passed To

```
SecurityManager.{LoadPolicyLevelFromFile( ), LoadPolicyLevelFromString( )}
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

SecurityElement

ECMA 1.0, serializable

System.Security (mscorlib.dll) *sealed class*

```

public sealed class SecurityElement {
// Public Constructors
    public SecurityElement(string tag);
    public SecurityElement(string tag, string text);
// Public Instance Properties
    public Hashtable Attributes{set; get; }
    public ArrayList Children{set; get; }
    public string Tag{set; get; }
    public string Text{set; get; }
// Public Static Methods
    public static string Escape(string str);
    public static bool IsValidAttributeName(string name);
    public static bool IsValidAttributeValue(string value);
    public static bool IsValidTag(string tag);
    public static bool IsValidText(string text);
// Public Instance Methods
    public void AddAttribute(string name, string value);
    public void AddChild(SecurityElement child);
    public string Attribute(string name);
    public bool Equal(SecurityElement other);
    public SecurityElement SearchForChildByTag(string tag);
    public string SearchForTextOfTag(string tag);
    public override string ToString( );
// overrides object
}

```

This class implements a simple, lightweight XML object model for encoding .NET security objects. A `SecurityElement` object represents a single XML element that can contain a tag, attributes, and text. To build a hierarchy of XML objects, add `SecurityElement` objects representing the child nodes to an existing `SecurityElement` object using its `Children` property or `AddChild()` method. The `ToString()` method returns a string containing the XML representation of the `SecurityElement` and all of its children.

Returned By

Multiple types

Passed To

Multiple types

[\[Team LiB \]](#)

[Team LiB]

SecurityException

CF 1.0, ECMA 1.0,
serializable

System.Security (mscorlib.dll)

class

```
public class SecurityException : SystemException {
    // Public Constructors
    public SecurityException( );
    public SecurityException(string message);
    public SecurityException(string message, Exception inner);
    public SecurityException(string message, Type type);
    public SecurityException(string message, Type type, string state);
    // Protected Constructors
    protected SecurityException(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context);
    // Public Instance Properties
    public string GrantedSet{get; }
    public string PermissionState{get; }
    public Type PermissionType{get; }
    public string RefusedSet{get; }
    // Public Instance Methods
    public override void GetObjectData(// overrides Exception
        System.Runtime.Serialization.SerializationInfo info, System.Runtime.Serialization.
    public override string ToString( );
    // overrides Exception
}
```

`SecurityException` is thrown when code tries to perform an action that it does not have permission to

Hierarchy

```
System.Object      System.Exception(System.Runtime.Serialization.ISerializable)      System
SecurityException
```

[Team LiB]

[Team LiB]

SecurityManager

System.Security (mscorlib.dll) *sealed class*

```
public sealed class SecurityManager {
    // Public Static Properties
    public static bool CheckExecutionRights {set; get; }
    public static bool SecurityEnabled {set; get; }
    // Public Static Methods
    public static bool IsGranted(IPermission perm);
    public static PolicyLevel LoadPolicyLevelFromFile(string path, PolicyLevelType type)
    public static PolicyLevel LoadPolicyLevelFromString(string str, PolicyLevelType type)
    public static IEnumerable PolicyHierarchy( );
    public static PermissionSet ResolvePolicy(System.Security.Policy.Evidence evidence);
    public static PermissionSet ResolvePolicy(System.Security.Policy.Evidence evidence,
PermissionSet reqdPset, PermissionSet optPset,
PermissionSet denyPset, out PermissionSet denied);
    public static IEnumerable ResolvePolicyGroups(System.Security.Policy.Evidence eviden
    public static void SavePolicy( );
    public static void SavePolicyLevel(System.Security.Policy.PolicyLevel level);
}
```

This class contains a set of static members that provide access to critical security system functionality a data. It is the principal access point for code needing to manipulate security policy programmatically, providing access to the active policy levels and allowing code to load and save policy levels.

`SecurityEnabled` is a master switch for all code-access security; setting it to false turns off all code-access and identity permission checks, but does not affect role-based security. `CheckExecutionRights` controls whether the runtime checks to ensure code has the `System.Security.Permissions.SecurityPermissionFlag.Execution` permission from the `System.Security.Permissions.SecurityPermission` class.

Though it is not possible to drive the runtime's policy resolution process programmatically, the `ResolvePolicy()` and `ResolvePolicyGroups()` methods simulate the policy resolution process to resolve the permissions and code groups for a specified `System.Security.Policy.Evidence` collection. The `IsGranted()` method is a lightweight method for testing to see if the current code has been granted a specified permission, but is not a suitable substitute for the `IPermission.Demand()` or `IStackWalk.Demand()` methods for the general enforcement of code-access security.

[Team LiB]

[\[Team LiB \]](#)

SecurityZone

serializable

System.Security (mscorlib.dll)

enum

```
public enum SecurityZone {
    MyComputer = 0, Intranet = 1, Trusted = 2, Internet = 3, Untrusted = 4, NoZone = -1
}
```

This enumeration defines constants that represent the different Internet Explorer security zones. `System.Security.Policy.Zone` evidence uses these values to identify the source zone of code. The values of `SecurityZone` are used when configuring a code group's membership condition as well as when using imperative and declarative security statements based on `System.Security.Permissions.ZoneIdentityPermission` objects.

Hierarchy

```
System.Object      System.ValueType      System.Enum(System.IComparable,
System.IFormattable, System.IConvertible)      SecurityZone
```

Returned By

```
System.Security.Permissions.ZoneIdentityPermission.SecurityZone,
System.Security.Permissions.ZoneIdentityPermissionAttribute.Zone,
System.Security.Policy.Zone.SecurityZone,
System.Security.Policy.ZoneMembershipCondition.SecurityZone
```

Passed To

```
System.Security.Permissions.ZoneIdentityPermission.{SecurityZone,
ZoneIdentityPermission( )},
System.Security.Permissions.ZoneIdentityPermissionAttribute.Zone,
System.Security.Policy.Zone.Zone( ),
System.Security.Policy.ZoneMembershipCondition.{SecurityZone,
ZoneMembershipCondition( )}
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

SuppressUnmanagedCodeSecurityAttribute

System.Security (mscorlib.dll) *sealed class*

```
public sealed class SuppressUnmanagedCodeSecurityAttribute : Attribute {  
    // Public Constructors  
    public SuppressUnmanagedCodeSecurityAttribute( );  
}
```

This attribute allows managed code to call unmanaged (native) code through PInvoke or COM interop without causing a stack walk to determine if all callers on the call stack have the `System.Security.Permissions.SecurityPermissionFlag.UnmanagedCode` permission from the `System.Security.Permissions.SecurityPermission` class. It is effective only when applied to PInvoke methods, classes containing PInvoke methods, or interface definitions through which COM interop calls are made, and will be ignored in all other instances.

When applied to frequently called methods that provide access to unmanaged code, `SuppressUnmanagedCodeSecurityAttribute` offers significant timesavings. However, careless use can easily open security holes that malicious code can use to access system resources.

Hierarchy

`System.Object` `System.Attribute` `SuppressUnmanagedCodeSecurityAttribute`

Valid On

Class, Method, Interface

[\[Team LiB \]](#)

[\[Team LiB \]](#)

UnverifiableCodeAttribute

System.Security (mscorlib.dll) *sealed class*

```
public sealed class UnverifiableCodeAttribute : Attribute {  
    // Public Constructors  
    public UnverifiableCodeAttribute( );  
}
```

This attribute class identifies a module as containing unverifiable code. You should not use this attribute directly, as it is applied where necessary by the compiler.

Hierarchy

System.Object → System.Attribute UnverifiableCodeAttribute

Valid On

Module

[\[Team LiB \]](#)

[Team LiB]

VerificationException

CF 1.0, ECMA 1.0,
serializable

System.Security (mscorlib.dll) *class*

```
public class VerificationException : SystemException {  
    // Public Constructors  
    public VerificationException( );  
    public VerificationException(string message);  
    public VerificationException(string message, Exception innerException);  
    // Protected Constructors  
    protected VerificationException(System.Runtime.Serialization.SerializationInfo info,  
        System.Runtime.Serialization.StreamingContext context);  
}
```

This exception is thrown when code that does not have permission to skip verification fails the verification process and cannot be verified as being type-safe.

Hierarchy

```
System.Object      System.Exception(System.Runtime.Serialization.ISerializable)  
System.SystemException  VerificationException
```

[Team LiB]

[\[Team LiB \]](#)

XmlSyntaxException

serializable

System.Security (mscorlib.dll)

sealed class

```
public sealed class XmlSyntaxException : SystemException {  
    // Public Constructors  
    public XmlSyntaxException( );  
    public XmlSyntaxException(int lineNumber);  
    public XmlSyntaxException(int lineNumber, string message);  
    public XmlSyntaxException(string message);  
    public XmlSyntaxException(string message, Exception inner);  
}
```

This exception is thrown when there are syntax errors when parsing XML.

Hierarchy

```
System.Object      System.Exception(System.Runtime.Serialization.ISerializable)  
System.SystemException      XmlSyntaxException
```

[\[Team LiB \]](#)

[Team LiB]

Chapter 24. The System.Security.Cryptography Namespace

The `System.Security.Cryptography` namespace provides support for managing cryptographic hash codes, data encryption, and digital signatures. The classes form a framework that can be extended easily to incorporate new algorithms. [Figure 24-1](#) and [Figure 24-2](#) show the types in this namespace.

Figure 24-1. Algorithms represented in System.Security.Cryptography

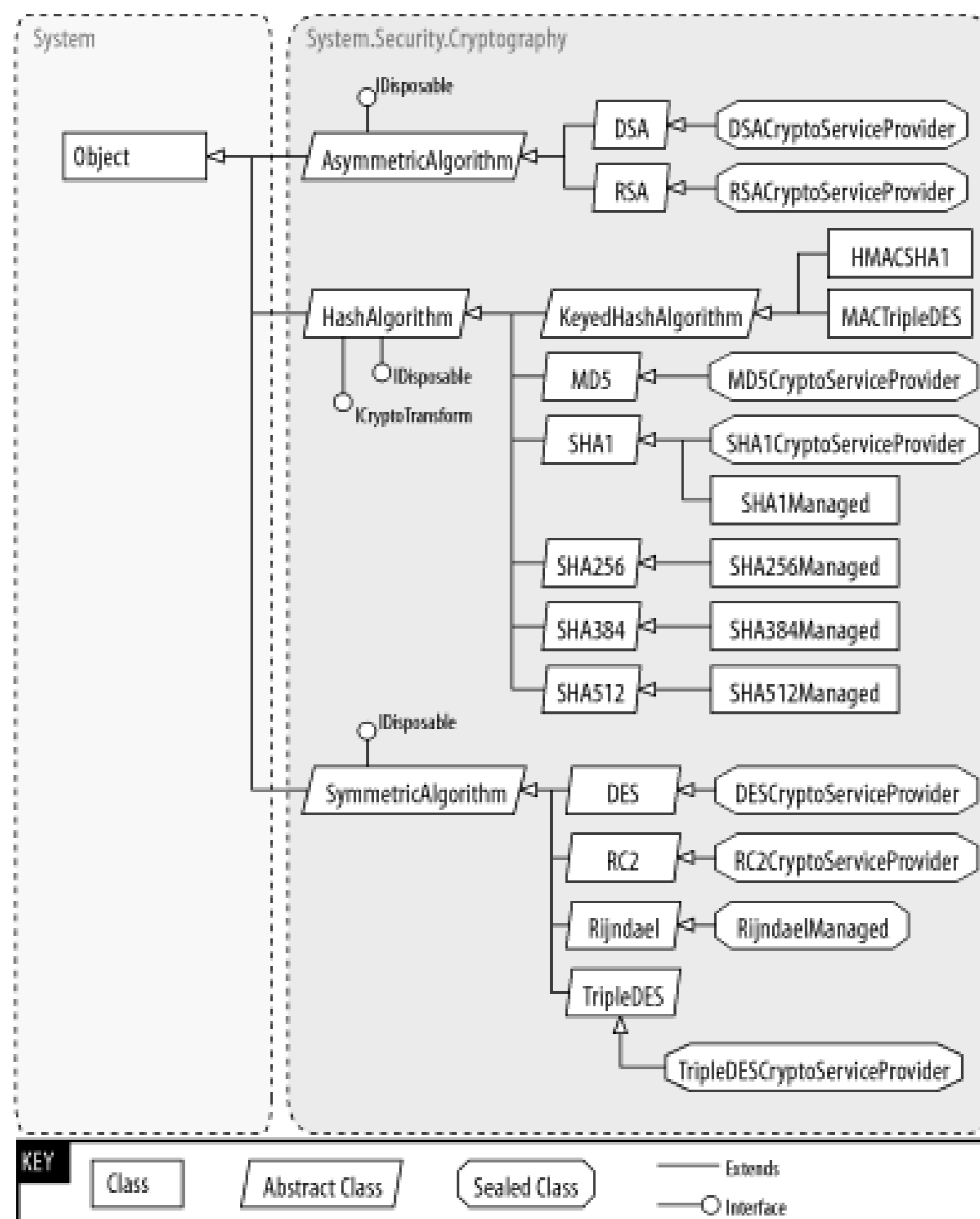
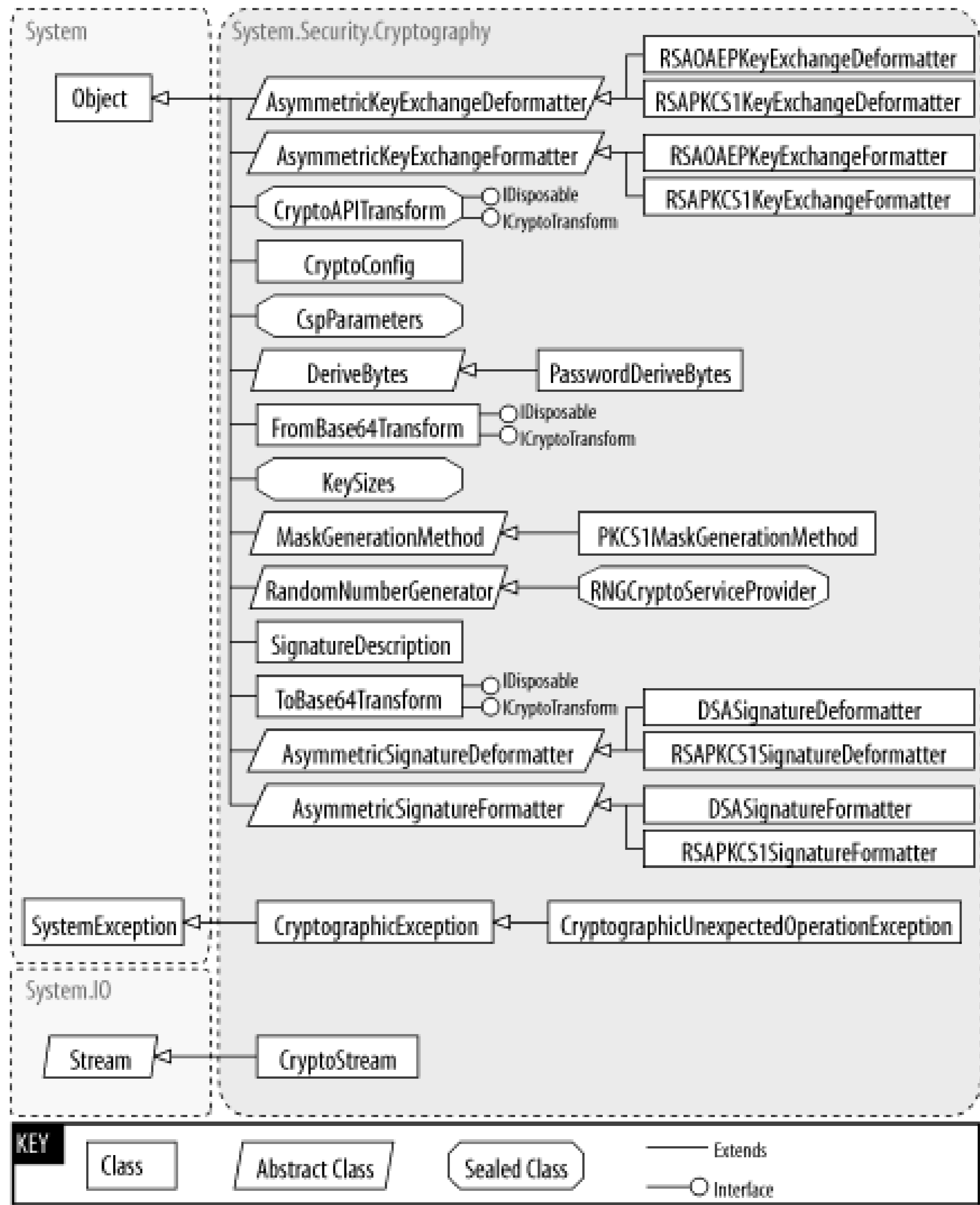


Figure 24-2. More types from System.Security.Cryptography



[Team LiB]

[\[Team LiB \]](#)

AsymmetricAlgorithm

disposable

System.Security.Cryptography
(mscorlib.dll)

*abstract
class*

```

public abstract class AsymmetricAlgorithm : IDisposable {
// Protected Constructors
    protected AsymmetricAlgorithm( );
// Protected Instance Fields
    protected int KeySizeValue;
    protected KeySizes[ ] LegalKeySizesValue;
// Public Instance Properties
    public abstract string KeyExchangeAlgorithm{get; }
    public virtual int KeySize{set; get; }
    public virtual KeySizes[ ] LegalKeySizes{get; }
    public abstract string SignatureAlgorithm{get; }
// Public Static Methods
    public static AsymmetricAlgorithm Create( );
    public static AsymmetricAlgorithm Create(string algName);
// Public Instance Methods
    public void Clear( );
    public abstract void FromXmlString(string xmlString);
    public abstract string ToXmlString(bool includePrivateParameters);
// Protected Instance Methods
    protected abstract void Dispose(bool disposing);
}

```

An asymmetric algorithm is one where the sender and the recipient do not share the same knowledge (i.e. only one party knows the secret key). All asymmetric algorithm implementation classes inherit from this abstract class. Implementations are instantiated using the `Create()` method, which accepts the name of an implementation as a `String` argument; the default implementation class is created if no name is specified; the system administrator determines the default class.

The `LegalKeySizes` property returns an array of the `KeySizes` class, specifying the set of key lengths supported by an implementation class. The `KeySize` property gets or sets the key length. The `FromXmlString()` and `ToXmlString()` methods import and export key parameters as XML strings.

Differences in the design of asymmetric algorithms mean that there is little common functionality embodied in this type; individual algorithm classes, such as `RSA` and `DSA`, define members for data encryption and digital signatures.

Subclasses

DSA, RSA

Returned By

```
System.Security.Cryptography.Xml.SignedXml.{GetPublicKey( ), SigningKey}
```

Passed To

Multiple types

[\[Team LiB \]](#)

[\[Team LiB \]](#)

AsymmetricKeyExchangeDeformatter

System.Security.Cryptography (mscorlib.dll) *abstract class*

```
public abstract class AsymmetricKeyExchangeDeformatter {
// Public Constructors
    public AsymmetricKeyExchangeDeformatter( );
// Public Instance Properties
    public abstract string Parameters{set; get; }
// Public Instance Methods
    public abstract byte[ ] DecryptKeyExchange(byte[ ] rgb);
    public abstract void SetKey(AsymmetricAlgorithm key);
}
```

This is the counterpart class to `AsymmetricKeyExchangeFormatter` and is the abstract base class which all key exchange deformatters extend. Implementations of this class decrypt formatted key exchange data.

The `DecryptKeyExchange()` method uses a private key to decrypt data encrypted with an implementation of the `AsymmetricKeyExchangeFormatter` class, using the corresponding public key. The `SetKey()` method specifies the `AsymmetricAlgorithm` instance that contains the private key. The `Parameters` property returns an XML string that describes the key exchange algorithm represented by the implementation class.

The .NET Framework class library includes the `RSOAEPKeyExchangeDeformatter` and `RSAPKCS1KeyExchangeDeformatter` classes, which decrypt data that has been encrypted using the RSA algorithm after being formatted with the Optimal Asymmetric Encryption Padding (OAEP) or PKCS #1 schemes.

Subclasses

`RSOAEPKeyExchangeDeformatter`, `RSAPKCS1KeyExchangeDeformatter`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

AsymmetricKeyExchangeFormatter

System.Security.Cryptography (mscorlib.dll) *abstract class*

```
public abstract class AsymmetricKeyExchangeFormatter {
    // Public Constructors
    public AsymmetricKeyExchangeFormatter( );
    // Public Instance Properties
    public abstract string Parameters{get; }
    // Public Instance Methods
    public abstract byte[ ] CreateKeyExchange(byte[ ] data);
    public abstract byte[ ] CreateKeyExchange(byte[ ] data, Type symAlgType);
    public abstract void SetKey(AsymmetricAlgorithm key);
}
```

Key exchange formatting allows a sender to encrypt a symmetric secret key value with an asymmetric algorithm and send the result to a recipient, who uses an asymmetric private key to decrypt the data. This process allows for the secure distribution of session keys, which are used with a symmetric algorithm to encrypt a single message. The secret key value is formatted prior to encryption to protect against cryptographic attack.

The abstract `AsymmetricKeyExchangeFormatter` class is the parent for all key exchange formatter implementations. The `CreateKeyExchange()` method uses an asymmetric public key to encrypt a symmetric secret key, expressed as a `System.Byte` array. The `SetKey()` method specifies the `AsymmetricAlgorithm` instance that contains the public key. The `Parameters` property returns an XML string that describes the key exchange algorithm represented by the implementation class.

The .NET Framework class library includes the `RSAOAEPKeyExchangeFormatter` and `RSAPKCS1KeyExchangeFormatter` classes, which format a secret key value using either the OAEP or PKCS #1 formatting schemes, prior to encryption with the RSA algorithm. The OAEP scheme has superseded PKCS #1, which should be used only for compatibility with legacy systems.

Subclasses

`RSAOAEPKeyExchangeFormatter`, `RSAPKCS1KeyExchangeFormatter`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

AsymmetricSignatureDeformatter

System.Security.Cryptography (mscorlib.dll) *abstract class*

```
public abstract class AsymmetricSignatureDeformatter {  
    // Public Constructors  
    public AsymmetricSignatureDeformatter( );  
    // Public Instance Methods  
    public abstract void SetHashAlgorithm(string strName);  
    public abstract void SetKey(AsymmetricAlgorithm key);  
    public abstract bool VerifySignature(byte[ ] rgbHash, byte[ ] rgbSignature);  
    public virtual bool VerifySignature(HashAlgorithm hash, byte[ ] rgbSignature);  
}
```

Implementations of this abstract class verify digital signatures created by the `AsymmetricSignatureFormatter` class.

The `VerifySignature()` method uses an asymmetric public key to verify a digital signature, expressed as a `System.Byte` array. The public key value is set by the `SetKey()` method, which accepts an instance of the `AsymmetricAlgorithm` class configured with the key parameters. The `SetHashAlgorithm()` method sets the name of the hash algorithm used to create the digital signature; this value is used in the verification process.

The .NET Framework class library includes the `DSASignatureDeformatter` and `RSAPKCS1SignatureDeformatter` classes, which verify digital signatures created with the `DSASignatureFormatter` and `RSAPKCS1SignatureFormatter` classes.

Subclasses

`DSASignatureDeformatter`, `RSAPKCS1SignatureDeformatter`

Returned By

`SignatureDescription.CreateDeformatter()`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

AsymmetricSignatureFormatter

System.Security.Cryptography
(mscorlib.dll)

*abstract
class*

```

public abstract class AsymmetricSignatureFormatter {
    // Public Constructors
    public AsymmetricSignatureFormatter( );
    // Public Instance Methods
    public abstract byte[ ] CreateSignature(byte[ ] rgbHash);
    public virtual byte[ ] CreateSignature(HashAlgorithm hash);
    public abstract void SetHashAlgorithm(string strName);
    public abstract void SetKey(AsymmetricAlgorithm key);
}

```

This is the abstract parent for all signature formatter implementation classes; such classes create digital signatures by formatting a cryptographic hash code and applying an asymmetric signature function to the formatted data.

The `CreateSignature()` method uses an asymmetric private key to create a digital signature, expressed as a `System.Byte` array. The private key value is specified by the `SetKey()` method, which accepts an instance of the `AsymmetricAlgorithm` class configured with the key parameters. The `SetHashAlgorithm()` method sets the name of the hash algorithm used to create the data to be signed; this value is used in the data-formatting process.

The .NET Framework class library includes the `DSASignatureFormatter` and `RSAPKCS1SignatureFormatter` classes, which create digital signatures using the `DSA` and `RSA` algorithm classes.

Subclasses

`DSASignatureFormatter`, `RSAPKCS1SignatureFormatter`

Returned By

`SignatureDescription.CreateFormatter()`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

CipherMode

serializable

System.Security.Cryptography
(mscorlib.dll)*enum*

```
public enum CipherMode {
    CBC = 1, ECB = 2, OFB = 3, CFB = 4, CTS = 5
}
```

The `SymmetricAlgorithm.Mode` property gets and sets a value from the `CipherMode` enumeration; the enumeration values represent different cipher modes, which affect the way in which data is processed during encryption.

The most commonly used `CipherMode` values are `CBC` (Cipher Block Chaining), which is suitable for most encryption tasks, and `CFB` (Cipher Feedback), which is suitable for encrypting data that is processed in small chunks.

Hierarchy

```
System.Object      System.ValueType      System.Enum(System.IComparable,
System.IFormattable, System.IConvertible)      CipherMode
```

Returned By

`SymmetricAlgorithm.Mode`

Passed To

`SymmetricAlgorithm.Mode`[\[Team LiB \]](#)

[\[Team LiB \]](#)

CryptoAPI Transform

disposable

System.Security.Cryptography
(mscorlib.dll)*sealed
class*

```
public sealed class CryptoAPITransform : ICryptoTransform, IDisposable {  
    // Public Instance Properties  
    public bool CanReuseTransform{get;}  
    // implements ICryptoTransform  
    public bool CanTransformMultipleBlocks{get;}  
    // implements ICryptoTransform  
    public int InputBlockSize{get;}  
    // implements ICryptoTransform  
    public IntPtr KeyHandle{get;}  
    public int OutputBlockSize{get;}  
    // implements ICryptoTransform  
    // Public Instance Methods  
    public void Clear( );  
    public int TransformBlock(byte[ ] inputBuffer, int inputOffset, int inputCount,  
    // implements ICryptoTransform  
        byte[ ] outputBuffer, int outputOffset);  
    public byte[ ] TransformFinalBlock(byte[ ] inputBuffer,  
    // implements ICryptoTransform  
        int inputOffset, int inputCount);  
    // Protected Instance Methods  
    protected override void Finalize( );  
    // overrides object  
}
```

This class implements the `ICryptoTransform` interface and is a wrapper around the Windows Crypto API. This is an internal class, and is not instantiated directly by the programmer.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

CryptoConfig

System.Security.Cryptography
(mscorlib.dll)

class

```
public class CryptoConfig {  
    // Public Constructors  
    public CryptoConfig( );  
    // Public Static Methods  
    public static object CreateFromName(string name);  
    public static object CreateFromName(string name, object[ ] args);  
    public static byte[ ] EncodeOID(string str);  
    public static string MapNameToOID(string name);  
}
```

This class defines the mapping between algorithm implementation classes and the `String` values used to create them; the `Create()` methods of the `AsymmetricAlgorithm` and `SymmetricAlgorithm` classes instantiate algorithms by name with the static `CreateFromName()` method of this class.

For example, the `String` values `SHA`, `SHA1` and `System.Security.Cryptography.SHA1` all map to the `SHA1CryptoServiceProvider` class detailed in this chapter. The mappings can be changed by editing the `Machine.config` file.

[\[Team LiB \]](#)

[Team LiB]

CryptographicException

serializable

System.Security.Cryptography
(mscorlib.dll)

class

```
public class CryptographicException : SystemException {  
    // Public Constructors  
    public CryptographicException( );  
    public CryptographicException(int hr);  
    public CryptographicException(string message);  
    public CryptographicException(string message, Exception inner);  
    public CryptographicException(string format, string insert);  
    // Protected Constructors  
    protected CryptographicException(System.Runtime.Serialization.SerializationInfo info  
        System.Runtime.Serialization.StreamingContext context);  
}
```

This exception represents a general error thrown when a cryptographic operation fails.

Hierarchy

```
System.Object      System.Exception(System.Runtime.Serialization.ISerializable)  
System.SystemException  CryptographicException
```

Subclasses

```
CryptographicUnexpectedOperationException
```

[Team LiB]

[Team LiB]

CryptographicUnexpectedOperationException serializable

System.Security.Cryptography
(mscorlib.dll)

class

```
public class CryptographicUnexpectedOperationException : CryptographicException {  
    // Public Constructors  
    public CryptographicUnexpectedOperationException( );  
    public CryptographicUnexpectedOperationException(string message);  
    public CryptographicUnexpectedOperationException(string message, Exception inner);  
    public CryptographicUnexpectedOperationException(string format, string insert);  
    // Protected Constructors  
    protected CryptographicUnexpectedOperationException(System.Runtime.Serialization.SerializationInfo info,  
        System.Runtime.Serialization.StreamingContext context);  
}
```

This exception represents a general error thrown when a cryptographic operation fails unexpectedly.

Hierarchy

```
System.Object      System.Exception(System.Runtime.Serialization.ISerializable)      System  
CryptographicException      CryptographicUnexpectedOperationException
```

[Team LiB]

[Team LiB]

CryptoStream

marshal by reference,
disposableSystem.Security.Cryptography
(mscorlib.dll)*class*

```
public class CryptoStream : System.IO.Stream {  
    // Public Constructors  
    public CryptoStream(System.IO.Stream stream, ICryptoTransform transform, CryptoStreamMode mode)  
    // Public Instance Properties  
    public override bool CanRead{get;}  
    // overrides System.IO.Stream  
    public override bool CanSeek{get;}  
    // overrides System.IO.Stream  
    public override bool CanWrite{get;}  
    // overrides System.IO.Stream  
    public override long Length{get;}  
    // overrides System.IO.Stream  
    public override long Position{set; get;}  
    // overrides System.IO.Stream  
    // Public Instance Methods  
    public void Clear( );  
    public override void Close( );  
    // overrides System.IO.Stream  
    public override void Flush( );  
    // overrides System.IO.Stream  
    public void FlushFinalBlock( );  
    public override int Read(in byte[] buffer, int offset, int count);  
    // overrides System.IO.Stream  
    public override long Seek(long offset, System.IO.SeekOrigin origin);  
    // overrides System.IO.Stream  
    public override void SetLength(long value);  
    // overrides System.IO.Stream  
    public override void Write(byte[] buffer, int offset, int count);  
    // overrides System.IO.Stream  
    // Protected Instance Methods  
    protected virtual void Dispose(bool disposing);  
    protected override void Finalize( );  
    // overrides object  
}
```

The `CryptoStream` class provides a stream-based model for encrypting and decrypting data using symmetric algorithms. The class constructor accepts an instance of `System.IO.Stream`, an implementation of the `ICryptoTransform` interface, and a value from the `CryptoStreamMode` enumeration.

If the `CryptoStreamMode` value is `Read()`, then data is read from `System.IO.Stream` and passed to the `ICryptoTransform` for processing. If the `CryptoStreamMode` value is `Write()`, then data is transformed by the `ICryptoTransform` implementation before being written to the `System.IO.Stream`. Implementations of `ICryptoTransform` can be created using the `SymmetricAlgorithm.CreateEncryptor()` and `SymmetricAlgorithm.CreateDecryptor()` methods.

This class extends `System.IO.Stream`, and therefore can be used as an argument in constructing a new `CryptoStream` itself, allowing transformations to be chained together.

Hierarchy

`System.Object` → `System.MarshalByRefObject` `System.IO.Stream(System.IDisposable)` (

[Team LiB]

[\[Team LiB \]](#)

CryptoStreamMode

serializable

System.Security.Cryptography
(mscorlib.dll)

enum

```
public enum CryptoStreamMode {  
    Read = 0, Write = 1  
}
```

This enumeration is used by the `CryptoStream` constructor to specify whether data will be read from, or written to, the `System.IO.Stream`.

Hierarchy

```
System.Object → System.ValueType      System.Enum(System.IComparable,  
System.IFormattable, System.IConvertible)      CryptoStreamMode
```

Passed To

```
CryptoStream.CryptoStream( )
```

[\[Team LiB \]](#)

[Team LiB]

CspParameters

System.Security.Cryptography *sealed*
(mscorlib.dll) *class*

```
public sealed class CspParameters {
    // Public Constructors
    public CspParameters( );
    public CspParameters(int dwTypeIn);
    public CspParameters(int dwTypeIn, string strProviderNameIn);
    public CspParameters(int dwTypeIn, string strProviderNameIn, string strContainerName
    // Public Instance Fields
    public string KeyContainerName;
    public int KeyNumber;
    public string ProviderName;
    public int ProviderType;
    // Public Instance Properties
    public CspProviderFlags Flags {set; get; }
}
```

The asymmetric algorithm implementation classes that are wrappers around the Windows Crypto API (`RSACryptoServiceProvider` and `DSACryptoServiceProvider`) are able to read and write key parameters from the persistent Windows key stores. This class can be used as an argument to the implementation class constructors in order to specify the details of which key store should be used.

Passed To

```
DSACryptoServiceProvider.DSACryptoServiceProvider( ),
PasswordDeriveBytes.PasswordDeriveBytes( ),
RNGCryptoServiceProvider.RNGCryptoServiceProvider( ),
RSACryptoServiceProvider.RSACryptoServiceProvider( )
```

[Team LiB]

[\[Team LiB \]](#)

CspProviderFlags

serializable, flags

System.Security.Cryptography
(mscorlib.dll)*enum*

```
public enum CspProviderFlags {  
    UseMachineKeyStore = 0x00000001, UseDefaultKeyContainer = 0x00000002  
}
```

The values of this enumeration specify the location where persistent asymmetric key values are stored; the constructor for the `CspParameters` class accepts a value from this enumeration.

Hierarchy

```
System.Object → System.ValueType      System.Enum(System.IComparable,  
System.IFormattable, System.IConvertible)      CspProviderFlags
```

Returned By

`CspParameters.Flags`

Passed To

`CspParameters.Flags`[\[Team LiB \]](#)

[\[Team LiB \]](#)

DeriveBytes

System.Security.Cryptography
(mscorlib.dll)

*abstract
class*

```
public abstract class DeriveBytes {  
    // Protected Constructors  
    protected DeriveBytes( );  
    // Public Instance Methods  
    public abstract byte[ ] GetBytes(int cb);  
    public abstract void Reset( );  
}
```

This abstract class is the basis for implementations that derive byte sequences from input data. For example, the .NET Framework class library includes the `PasswordDeriveBytes` class, which derives a cryptographic key from an easily remembered password.

The `GetBytes()` method derives a number of bytes from the input, where the number of bytes to derive is specified as the methods argument.

Subclasses

`PasswordDeriveBytes`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

DES

disposable

 System.Security.Cryptography
 (mscorlib.dll)
*abstract
class*

```

public abstract class DES : SymmetricAlgorithm {
// Public Constructors
    public DES( );
// Public Instance Properties
    public override byte[ ] Key {set; get;
// overrides SymmetricAlgorithm
// Public Static Methods
    public static DES Create( );
    public static DES Create(string algName);
    public static bool IsSemiWeakKey(byte[ ] rgbKey);
    public static bool IsWeakKey(byte[ ] rgbKey);
}

```

This class extends `SymmetricAlgorithm` and is the abstract representation of the Digital Encryption Standard (DES) algorithm. Individual implementations of DES extend this class. The .NET Framework class library includes the `DESCryptoServiceProvider` class, which is the default implementation.

In addition to the members defined by the parent class, `DES` implements the `IsSemiWeakKey()` and `IsWeakKey()` methods, which determine if the selected cryptographic key results in a cipher that is easy to break.

Hierarchy

System.Object SymmetricAlgorithm(System.IDisposable) DES

Subclasses

DESCryptoServiceProvider

[\[Team LiB \]](#)

[Team LiB]

DESCryptoServiceProvider

disposable

System.Security.Cryptography
(mscorlib.dll)*sealed
class*

```

public sealed class DESCryptoServiceProvider : DES {
// Public Constructors
    public DESCryptoServiceProvider( );
// Public Instance Methods
    public override ICryptoTransform CreateDecryptor(byte[] rgbKey, byte[] rgbIV);
// overrides SymmetricAlgorithm
    public override ICryptoTransform CreateEncryptor(byte[] rgbKey, byte[] rgbIV);
// overrides SymmetricAlgorithm
    public override void GenerateIV( );
// overrides SymmetricAlgorithm
    public override void GenerateKey( );
// overrides SymmetricAlgorithm
}

```

This subclass of `DES` is the default implementation of the DES algorithm and is a wrapper around the Windows Crypto API.

Hierarchy

```

System.Object      SymmetricAlgorithm(System.IDisposable)      DES
DESCryptoServiceProvider

```

[Team LiB]

[\[Team LiB \]](#)

DSA

disposable

System.Security.Cryptography
(mscorlib.dll)*abstract
class*

```
public abstract class DSA : AsymmetricAlgorithm {
// Public Static Methods
    public static DSA Create( );
    public static DSA Create(string algName);
// Public Instance Methods
    public abstract byte[ ] CreateSignature(byte[ ] rgbHash);
    public abstract DSAParameters ExportParameters(bool includePrivateParameters);
    public override void FromXmlString(string xmlString);
// overrides AsymmetricAlgorithm
    public abstract void ImportParameters(DSAParameters parameters);
    public override string ToXmlString( bool includePrivateParameters);
// overrides AsymmetricAlgorithm
    public abstract bool VerifySignature(byte[ ] rgbHash, byte[ ] rgbSignature);
}
```

This class extends `AsymmetricAlgorithm` and is the abstract representation of the Digital Signature Algorithm (DSA). Individual implementations of DSA will extend this class. The .NET Framework class library includes the `DSACryptoServiceProvider` class, which is the default implementation of the algorithm.

DSA key pairs are represented by the `DSAParameters` structure, and can be imported and exported from the `DSA` class with the `ImportParameters()` and `ExportParameters()` methods. The `CreateSignature()` method produces a digital signature for a cryptographic hash code, which can be verified using the `VerifySignature()` method.

The DSA algorithm does not support data encryption.

Hierarchy

```
System.Object    AsymmetricAlgorithm(System.IDisposable)    DSA
```

Subclasses

```
DSACryptoServiceProvider
```

Returned By

```
System.Security.Cryptography.Xml.DSAKeyValue.Key
```

Passed To

`System.Security.Cryptography.Xml.DSAKeyValue.{DSAKeyValue(), Key}`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

DSACryptoServiceProvider

disposable

System.Security.Cryptography
(mscorlib.dll)*sealed
class*

```
public sealed class DSACryptoServiceProvider : DSA {
// Public Constructors
    public DSACryptoServiceProvider( );
    public DSACryptoServiceProvider(CspParameters parameters);
    public DSACryptoServiceProvider(int dwKeySize);
    public DSACryptoServiceProvider(int dwKeySize, CspParameters parameters);
// Public Static Properties
    public static bool UseMachineKeyStore{set; get; }
// Public Instance Properties
    public override string KeyExchangeAlgorithm{get;
// overrides AsymmetricAlgorithm
    public override int KeySize{get;
// overrides AsymmetricAlgorithm
    public override KeySizes[] LegalKeySizes{get;
// overrides AsymmetricAlgorithm
    public bool PersistKeyInCsp{set; get; }
    public override string SignatureAlgorithm{get;
// overrides AsymmetricAlgorithm
// Public Instance Methods
    public override byte[] CreateSignature(byte[] rgbHash);
// overrides DSA
    public override DSAParameters ExportParameters(bool includePrivateParameters);
// overrides DSA
    public override void ImportParameters( DSAParameters parameters);
// overrides DSA
    public byte[] SignData(byte[] buffer);
    public byte[] SignData(byte[] buffer, int offset, int count);
    public byte[] SignData(System.IO.Stream inputStream);
    public byte[] SignHash(byte[] rgbHash, string str);
    public bool VerifyData(byte[] rgbData, byte[] rgbSignature);
    public bool VerifyHash(byte[] rgbHash, string str, byte[] rgbSignature);
    public override bool VerifySignature(byte[] rgbHash, byte[] rgbSignature);
// overrides DSA
// Protected Instance Methods
    protected override void Dispose(bool disposing);
// overrides AsymmetricAlgorithm
    protected override void Finalize( );
// overrides object
}
```

This subclass of `DSA` is the default implementation of the DSA and is a wrapper around the Windows Crypto API.

This class defines several new members; the `SignData()` method creates a digital signature by reading data from a stream. The `VerifyData()` method verifies a digital signature from the signed data (as opposed to the cryptographic hash code). The `SignHash()` and `VerifyHash()` methods are functionally equivalent to the `CreateSignature()` and `VerifySignature()` methods of the parent class.

Hierarchy

`System.Object` → `AsymmetricAlgorithm(System.IDisposable)` `DSA`
`DSACryptoServiceProvider`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

DSAParameters

serializable

System.Security.Cryptography
(mscorlib.dll)*struct*

```
public struct DSAParameters {  
    // Public Instance Fields  
    public int Counter;  
    public byte[] G;  
    public byte[] J;  
    public byte[] P;  
    public byte[] Q;  
    public byte[] Seed;  
    public byte[] X;  
    public byte[] Y;  
}
```

This structure represents the parameters of a DSA key, expressed as a set of `Byte` arrays. Key parameters are exported from the `DSA` class with the `ExportParameters()` method. Keys parameters are imported into the `DSA` class with the `ImportParameters()` method.

Hierarchy

```
System.Object      System.ValueType      DSAParameters
```

Returned By

```
DSA.ExportParameters( )
```

Passed To

```
DSA.ImportParameters( )
```

[\[Team LiB \]](#)

[Team LiB]

DSASignatureDeformatter

System.Security.Cryptography
(mscorlib.dll)

class

```
public class DSASignatureDeformatter : AsymmetricSignatureDeformatter {  
    // Public Constructors  
    public DSASignatureDeformatter( );  
    public DSASignatureDeformatter(AsymmetricAlgorithm key);  
    // Public Instance Methods  
    public override void SetHashAlgorithm(string strName);  
    // overrides AsymmetricSignatureDeformatter  
    public override void SetKey(AsymmetricAlgorithm key);  
    // overrides AsymmetricSignatureDeformatter  
    public override bool VerifySignature(byte[ ] rgbHash, byte[ ] rgbSignature) // over  
}
```

This child of the `AsymmetricSignatureDeformatter` class verifies Digital Signature Algorithm (DSA) signature. `DSASignatureDeformatter` class.

Hierarchy

System.Object AsymmetricSignatureDeformatter DSASignatureDeformatter

[Team LiB]

[\[Team LiB \]](#)

DSASignatureFormatter

System.Security.Cryptography
(mscorlib.dll)

class

```
public class DSASignatureFormatter : AsymmetricSignatureFormatter {  
    // Public Constructors  
    public DSASignatureFormatter( );  
    public DSASignatureFormatter(AsymmetricAlgorithm key);  
    // Public Instance Methods  
    public override byte[ ] CreateSignature(byte[ ] rgbHash);  
    // overrides AsymmetricSignatureFormatter  
    public override void SetHashAlgorithm(string strName);  
    // overrides AsymmetricSignatureFormatter  
    public override void SetKey(AsymmetricAlgorithm key);  
    // overrides AsymmetricSignatureFormatter  
}
```

This class extends the `AsymmetricSignatureFormatter` class and creates digital signatures using the DSA. Signatures created by `DSASignatureFormatter` can be verified with the `DSASignatureDeformatter` class.

Hierarchy

System.Object AsymmetricSignatureFormatter DSASignatureFormatter

[\[Team LiB \]](#)

[Team LiB]

FromBase64Transform

disposable

System.Security.Cryptography
(mscorlib.dll)*class*

```

public class FromBase64Transform : ICryptoTransform, IDisposable {
    // Public Constructors
    public FromBase64Transform( );
    public FromBase64Transform(FromBase64TransformMode whitespaces);
    // Public Instance Properties
    public virtual bool CanReuseTransform{get;}
    // implements ICryptoTransform
    public bool CanTransformMultipleBlocks{get;}
    // implements ICryptoTransform
    public int InputBlockSize{get;}
    // implements ICryptoTransform
    public int OutputBlockSize{get;}
    // implements ICryptoTransform
    // Public Instance Methods
    public void Clear( );
    public int TransformBlock(byte[ ] inputBuffer, int inputOffset, int inputCount,
        byte[ ] outputBuffer, int outputOffset);
    // implements ICryptoTransform
    public byte[ ] TransformFinalBlock(byte[ ] inputBuffer, int inputOffset, int inputCount);
    // implements ICryptoTransform
    // Protected Instance Methods
    protected virtual void Dispose(bool disposing);
    protected override void Finalize( );
    // overrides object
}

```

This implementation of the `ICryptoTransform` interface decodes Base64 data. This class is used in conjunction with the `CryptoStream` class.

[Team LiB]

[\[Team LiB \]](#)

FromBase64TransformMode serializable

System.Security.Cryptography (mscorlib.dll) *enum*

```
public enum FromBase64TransformMode {  
    IgnoreWhiteSpaces = 0, DoNotIgnoreWhiteSpaces = 1  
}
```

This enumeration is used as an argument to the `FromBase64Transform` constructor to specify whether whitespace in the Base64-encoded data should be ignored.

Hierarchy

```
System.Object → System.ValueType System.Enum(System.IComparable,  
System.IFormattable, System.IConvertible) FromBase64TransformMode
```

Passed To

```
FromBase64Transform.FromBase64Transform( )
```

[\[Team LiB \]](#)

[Team LiB]

HashAlgorithm

disposable

System.Security.Cryptography
(mscorlib.dll)*abstract
class*

```

public abstract class HashAlgorithm : ICryptoTransform, IDisposable {
    // Protected Constructors
    protected HashAlgorithm( );
    // Protected Instance Fields
    protected int HashSizeValue;
    protected byte[ ] HashValue;
    protected int State;
    // Public Instance Properties
    public virtual bool CanReuseTransform {get;}
    // implements ICryptoTransform
    public virtual bool CanTransformMultipleBlocks {get;}
    // implements ICryptoTransform
    public virtual byte[ ] Hash {get;}
    public virtual int HashSize {get;}
    public virtual int InputBlockSize {get;}
    // implements ICryptoTransform
    public virtual int OutputBlockSize {get;}
    // implements ICryptoTransform
    // Public Static Methods
    public static HashAlgorithm Create( );
    public static HashAlgorithm Create(string hashName);
    // Public Instance Methods
    public void Clear( );
    public byte[ ] ComputeHash(byte[ ] buffer);
    public byte[ ] ComputeHash(byte[ ] buffer, int offset, int count);
    public byte[ ] ComputeHash(System.IO.Stream inputStream);
    public abstract void Initialize( );
    public int TransformBlock(byte[ ] inputBuffer, int inputOffset,
        int inputCount, byte[ ] outputBuffer, int outputOffset);
    // implements ICryptoTransform
    public byte[ ] TransformFinalBlock(byte[ ] inputBuffer, int inputOffset, int inputCount);
    // implements ICryptoTransform
    // Protected Instance Methods
    protected virtual void Dispose(bool disposing);
    protected abstract void HashCore(byte[ ] array, int ibStart, int cbSize);
    protected abstract byte[ ] HashFinal( );
}

```

A hash algorithm processes data to create a fixed-length string, known as a hash code. Hash algorithms are useful because it is difficult to find two sets of data that result in the same hash code; hash algorithms are

important part of the process to create digital signatures and are used to ensure data integrity.

All hash algorithm implementations inherit from this abstract class. Implementation classes are instantiated by the `Create()` method, which accepts the name of an implementation as a `String` argument. The default implementation is created if no implementation name is specified; the system administrator can configure the algorithm.

The `ComputeHash()` method generates a hash code by either processing a `Byte` array or by reading data from a `Stream`.

Subclasses

`KeyedHashAlgorithm`, `MD5`, `SHA1`, `SHA256`, `SHA384`, `SHA512`

Returned By

`SignatureDescription.CreateDigest()`,
`System.Security.Policy.HashMembershipCondition.HashAlgorithm`

Passed To

`AsymmetricSignatureDeformatter.VerifySignature()`,
`AsymmetricSignatureFormatter.CreateSignature()`, `System.Security.Policy.Hash.GenerateHash()`,
`System.Security.Policy.HashMembershipCondition.{HashAlgorithm, HashMembershipCondition}`

[Team LiB]

[\[Team LiB \]](#)

HMACSHA1

disposable

 System.Security.Cryptography
 (mscorlib.dll)
class

```

public class HMACSHA1 : KeyedHashAlgorithm {
    // Public Constructors
    public HMACSHA1( );
    public HMACSHA1(byte[ ] rgbKey);
    // Public Instance Properties
    public string HashName{set; get; }
    public override byte[ ] Key{set; get; }
    // overrides KeyedHashAlgorithm
    // Public Instance Methods
    public override void Initialize( );
    // overrides HashAlgorithm
    // Protected Instance Methods
    protected override void Dispose(bool disposing);
    // overrides KeyedHashAlgorithm
    protected override void Finalize( );
    // overrides KeyedHashAlgorithm
    protected override void HashCore(byte[ ] rgb, int ib, int cb);
    // overrides HashAlgorithm
    protected override byte[ ] HashFinal( );
    // overrides HashAlgorithm
}

```

This class extends the `KeyedHashAlgorithm` class and implements the HMAC-SHA-1 algorithm, which uses the SHA-1 algorithm as the basis for creating keyed hash codes.

Hierarchy

```

System.Object      HashAlgorithm(ICryptoTransform, System.IDisposable)
KeyedHashAlgorithm  HMACSHA1

```

[\[Team LiB \]](#)

[Team LiB]

ICryptoTransform

disposable

System.Security.Cryptography
(mscorlib.dll)*interface*

```
public interface ICryptoTransform : IDisposable {
    // Public Instance Properties
    public bool CanReuseTransform {get; }
    public bool CanTransformMultipleBlocks {get; }
    public int InputBlockSize {get; }
    public int OutputBlockSize {get; }
    // Public Instance Methods
    public int TransformBlock(byte[] inputBuffer, int inputOffset, int inputCount,
        byte[] outputBuffer, int outputOffset);
    public byte[] TransformFinalBlock(byte[] inputBuffer, int inputOffset, int inputCount);
}
```

The `ICryptoTransform` interface defines the basic operations of a symmetric block cipher, and is used in conjunction with the `CryptoStream` class to transform data. Implementations of this interface are most commonly obtained through the `SymmetricAlgorithm.CreateEncryptor()` and `SymmetricAlgorithm.CreateDecryptor()` methods.

Implemented By

```
CryptoAPITransform, FromBase64Transform, HashAlgorithm, ToBase64Transform
```

Returned By

```
SymmetricAlgorithm.{CreateDecryptor( ), CreateEncryptor( )}
```

Passed To

```
CryptoStream.CryptoStream( )
```

[Team LiB]

[\[Team LiB \]](#)

KeyedHashAlgorithm

disposable

System.Security.Cryptography
(mscorlib.dll)*abstract
class*

```
public abstract class KeyedHashAlgorithm : HashAlgorithm {
// Protected Constructors
    protected KeyedHashAlgorithm( );
// Protected Instance Fields
    protected byte[ ] KeyValue;
// Public Instance Properties
    public virtual byte[ ] Key{set; get; }
// Public Static Methods
    public static KeyedHashAlgorithm Create( );
    public static KeyedHashAlgorithm Create(string algName);
// Protected Instance Methods
    protected override void Dispose(bool disposing);
// overrides HashAlgorithm
    protected override void Finalize( );
// overrides object
}
```

Keyed hash algorithms provide authenticated hash codes by combining a secret key with the data to process. The recipient can validate the hash code only if he knows the secret key.

All keyed hash algorithm implementations inherit from this abstract class, which extends the `HashAlgorithm` class. Implementation classes are instantiated using the `Create()` method, which accepts the name of an implementation as a `System.String` argument. The default implementation class is created if no name is specified; the system administrator determines the default class.

In addition to the members defined by the parent class, `KeyedHashAlgorithm` implements the `Key` property, which gets and sets the value of the secret key, expressed as a `System.Byte` array.

Hierarchy

```
System.Object      HashAlgorithm(ICryptoTransform, System.IDisposable)
KeyedHashAlgorithm
```

Subclasses

```
HMACSHA1, MACTripleDES
```

Passed To

```
System.Security.Cryptography.Xml.SignedXml.{CheckSignature( ), ComputeSignature( )}
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

KeySizes

System.Security.Cryptography
(mscorlib.dll)

*sealed
class*

```
public sealed class KeySizes {  
    // Public Constructors  
    public KeySizes(int minSize, int maxSize, int skipSize);  
    // Public Instance Properties  
    public int MaxSize{get; }  
    public int MinSize{get; }  
    public int SkipSize{get; }  
}
```

The `KeySizes` class expresses a range of cryptographic key lengths; instances of this class are returned from the `SymmetricAlgorithm.LegalKeySizes()` and `AsymmetricAlgorithm.LegalKeySizes()` methods.

The three class properties express the range of key lengths; `MinSize` specifies the shortest supported key length, `MaxSize` specifies the longest supported length, and `SkipSize` specifies the interval between supported lengths. For example, if `MinSize` is 16, `MaxSize` is 32, and `SkipSize` is 8, then the `KeySize` would represent key lengths of 16, 24, and 32 bits.

Returned By

`AsymmetricAlgorithm.LegalKeySizes`, `SymmetricAlgorithm.LegalBlockSizes`, `LegalKeySizes`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

MACTripleDES

disposable

System.Security.Cryptography
(mscorlib.dll)*class*

```
public class MACTripleDES : KeyedHashAlgorithm {
// Public Constructors
    public MACTripleDES( );
    public MACTripleDES(byte[ ] rgbKey);
    public MACTripleDES(string strTripleDES, byte[ ] rgbKey);
// Public Instance Methods
    public override void Initialize( );
// overrides HashAlgorithm
// Protected Instance Methods
    protected override void Dispose(bool disposing);
// overrides KeyedHashAlgorithm
    protected override void Finalize( );
// overrides KeyedHashAlgorithm
    protected override void HashCore(byte[ ] rgbData, int ibStart, int cbSize);
// overrides HashAlgorithm
    protected override byte[ ] HashFinal( );
// overrides HashAlgorithm
}
```

This class extends the `KeyedHashAlgorithm` class and implements the MAC-Triple-DES algorithm, which uses the Triple-DES encryption algorithm to create a keyed hash code.

Hierarchy

```
System.Object      HashAlgorithm(ICryptoTransform, System.IDisposable)
KeyedHashAlgorithm  MACTripleDES
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

MaskGenerationMethod

System.Security.Cryptography
(mscorlib.dll)

*abstract
class*

```
public abstract class MaskGenerationMethod {  
    // Protected Constructors  
    protected MaskGenerationMethod( );  
    // Public Instance Methods  
    public abstract byte[ ] GenerateMask(byte[ ] rgbSeed, int cbReturn);  
}
```

This class creates masks used in key exchange algorithms. This is an internal class and is not instantiated directly by the programmer.

Subclasses

PKCS1MaskGenerationMethod

[\[Team LiB \]](#)

[\[Team LiB \]](#)

MD5

disposable

System.Security.Cryptography
(mscorlib.dll)

*abstract
class*

```
public abstract class MD5 : HashAlgorithm {  
    // Protected Constructors  
    protected MD5( );  
    // Public Static Methods  
    public static MD5 Create( );  
    public static MD5 Create(string algName);  
}
```

This class extends `HashAlgorithm` and is the abstract representation of the MD5 algorithm. Individual implementations of MD5 will extend this class. The .NET Framework class library includes the `MD5CryptoServiceProvider` class, which is the default implementation.

Hierarchy

System.Object HashAlgorithm(ICryptoTransform, System.IDisposable) MD5

Subclasses

MD5CryptoServiceProvider

[\[Team LiB \]](#)

[\[Team LiB \]](#)

MD5CryptoServiceProvider

disposable

System.Security.Cryptography
(mscorlib.dll)*sealed
class*

```

public sealed class MD5CryptoServiceProvider : MD5 {
// Public Constructors
    public MD5CryptoServiceProvider( );
// Public Instance Methods
    public override void Initialize( );
// overrides HashAlgorithm
// Protected Instance Methods
    protected override void Dispose(bool disposing);
// overrides HashAlgorithm
    protected override void Finalize( );
// overrides object
    protected override void HashCore(byte[ ] rgb, int ibStart, int cbSize);
// overrides HashAlgorithm
    protected override byte[ ] HashFinal( );
// overrides HashAlgorithm
}

```

This subclass of `MD5` is the default implementation of the MD5 algorithm and is a wrapper around the Windows Crypto API.

Hierarchy

```

System.Object      HashAlgorithm(ICryptoTransform, System.IDisposable)      MD5
MD5CryptoServiceProvider

```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

PaddingMode

serializable

System.Security.Cryptography
(mscorlib.dll)*enum*

```
public enum PaddingMode {  
    None = 1, PKCS7 = 2, Zeros = 3  
}
```

The `PaddingMode` enumeration is used with the `SymmetricAlgorithm.Padding` property to specify a padding scheme for data encryption. The `PKCS7` value specifies PKCS #7 padding, where the value of the padding bytes is set to be the number of padding bytes (for example, if seven bytes of padding are required, then the value of these bytes will be 7). The `Zeros` value specifies that all padding bytes should be set to have a value of 0.

Hierarchy

```
System.Object      System.ValueType      System.Enum(System.IComparable,  
System.IFormattable, System.IConvertible)      PaddingMode
```

Returned By

`SymmetricAlgorithm.Padding`

Passed To

`SymmetricAlgorithm.Padding`[\[Team LiB \]](#)

[Team LiB]

PasswordDeriveBytes

System.Security.Cryptography
(mscorlib.dll)

class

```
public class PasswordDeriveBytes : DeriveBytes {
// Public Constructors
    public PasswordDeriveBytes(string strPassword, byte[] rgbSalt);
    public PasswordDeriveBytes(string strPassword, byte[] rgbSalt, CspParameters cspPa.
    public PasswordDeriveBytes(string strPassword, byte[] rgbSalt, string strHashName,
        int iterations);
    public PasswordDeriveBytes(string strPassword, byte[] rgbSalt, string strHashName,
        int iterations, CspParameters cspParams);
// Public Instance Properties
    public string HashName{set; get; }
    public int IterationCount{set; get; }
    public byte[] Salt{set; get; }
// Public Instance Methods
    public byte[] CryptDeriveKey(string alname, string alhashname, int keySize, byte
    public override byte[] GetBytes(int cb);
// overrides DeriveBytes
    public override void Reset( );
// overrides DeriveBytes
// Protected Instance Methods
    protected override void Finalize( );
// overrides object
}
```

This class extends the abstract `DeriveBytes` class and generates symmetric key values by applying a ha a password string. The `HashName` property specifies the hash algorithm to use and the `IterationCount` p indicates how many times the hash algorithm will be applied in order to create the symmetric key value

Hierarchy

System.Object DeriveBytes PasswordDeriveBytes

[Team LiB]

[\[Team LiB \]](#)

PKCS1MaskGenerationMethod

System.Security.Cryptography
(mscorlib.dll) *class*

```
public class PKCS1MaskGenerationMethod : MaskGenerationMethod {  
    // Public Constructors  
    public PKCS1MaskGenerationMethod( );  
    // Public Instance Properties  
    public string HashName{set; get; }  
    // Public Instance Methods  
    public override byte[ ] GenerateMask(byte[ ] rgbSeed, int cbReturn);  
    // overrides MaskGenerationMethod  
}
```

This class creates masks used as part of OAEP key exchange formatting. This is an internal class, and is not instantiated directly by the programmer.

Hierarchy

System.Object MaskGenerationMethod PKCS1MaskGenerationMethod

[\[Team LiB \]](#)

[\[Team LiB \]](#)

RandomNumberGenerator

System.Security.Cryptography
(mscorlib.dll)

*abstract
class*

```
public abstract class RandomNumberGenerator {
// Public Constructors
    public RandomNumberGenerator( );
// Public Static Methods
    public static RandomNumberGenerator Create( );
    public static RandomNumberGenerator Create(string rngName);
// Public Instance Methods
    public abstract void GetBytes(byte[] data);
    public abstract void GetNonZeroBytes(byte[] data);
}
```

The `RandomNumberGenerator` class is the abstract base from which random number generator implementations inherit. Implementation classes are instantiated with the `Create()` method, which accepts the name of an implementation as a `String` argument. An instance of the default algorithm is created if no name is specified; the system administrator can configure the default algorithm.

The `GetBytes()` method populates a `System.Byte` array with random values; the array to fill is the sole argument to the method. The `GetNonZeroBytes()` method populates a `System.Byte` array with random values, none of which will be 0.

The .NET Framework class library includes the `RNGCryptoServiceProvider` class, which is the default implementation.

Subclasses

`RNGCryptoServiceProvider`

Returned By

`RSOAEPKeyExchangeFormatter.Rng`, `RSAPKCS1KeyExchangeDeformatter.RNG`,
`RSAPKCS1KeyExchangeFormatter.Rng`

Passed To

`RSOAEPKeyExchangeFormatter.Rng`, `RSAPKCS1KeyExchangeDeformatter.RNG`,
`RSAPKCS1KeyExchangeFormatter.Rng`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

RC2

disposable

 System.Security.Cryptography
 (mscorlib.dll)
*abstract
class*

```

public abstract class RC2 : SymmetricAlgorithm {
// Public Constructors
    public RC2( );
// Protected Instance Fields
    protected int EffectiveKeySizeValue;
// Public Instance Properties
    public virtual int EffectiveKeySize {set; get; }
    public override int KeySize {set; get; }
// overrides SymmetricAlgorithm
// Public Static Methods
    public static RC2 Create( );
    public static RC2 Create(string AlgName);
}

```

This class extends `SymmetricAlgorithm` and is the abstract representation of the RC2 algorithm. Individual implementations of RC2 will extend this class. The .NET Framework class library includes the `RC2CryptoServiceProvider` class, which is the default implementation.

Hierarchy

```

System.Object      SymmetricAlgorithm(System.IDisposable)      RC2

```

Subclasses

```

RC2CryptoServiceProvider

```

[\[Team LiB \]](#)

[Team LiB]

RC2CryptoServiceProvider

disposable

System.Security.Cryptography
(mscorlib.dll)*sealed
class*

```

public sealed class RC2CryptoServiceProvider : RC2 {
    // Public Constructors
    public RC2CryptoServiceProvider( );
    // Public Instance Properties
    public override int EffectiveKeySize{set; get;
    // overrides RC2
    // Public Instance Methods
    public override ICryptoTransform CreateDecryptor(byte[ ] rgbKey, byte[ ] rgbIV);
    // overrides SymmetricAlgorithm
    public override ICryptoTransform CreateEncryptor(byte[ ] rgbKey, byte[ ] rgbIV);
    // overrides SymmetricAlgorithm
    public override void GenerateIV( );
    // overrides SymmetricAlgorithm
    public override void GenerateKey( );
    // overrides SymmetricAlgorithm
}

```

This subclass of `RC2` is the default implementation of the RC2 algorithm and is a wrapper around the Windows Crypto API.

Hierarchy

```

System.Object    SymmetricAlgorithm(System.IDisposable)    RC2
RC2CryptoServiceProvider

```

[Team LiB]

[\[Team LiB \]](#)

Rijndael

disposable

System.Security.Cryptography
(mscorlib.dll)

*abstract
class*

```
public abstract class Rijndael : SymmetricAlgorithm {  
    // Public Constructors  
    public Rijndael( );  
    // Public Static Methods  
    public static Rijndael Create( );  
    public static Rijndael Create(string algName);  
}
```

This class extends `SymmetricAlgorithm` and is the abstract representation of the Rijndael algorithm. Individual implementations of Rijndael will extend this class. The .NET Framework class library includes the `Rijndael` class, which is the default implementation.

The Rijndael algorithm is also known as the Advanced Encryption Standard (AES).

Hierarchy

System.Object SymmetricAlgorithm(System.IDisposable) Rijndael

Subclasses

RijndaelManaged

[\[Team LiB \]](#)

[Team LiB]

RijndaelManaged

disposable

System.Security.Cryptography
(mscorlib.dll)*sealed
class*

```
public sealed class RijndaelManaged : Rijndael {
// Public Constructors
    public RijndaelManaged( );
// Public Instance Methods
    public override ICryptoTransform CreateDecryptor(byte[] rgbKey, byte[] rgbIV);
// overrides SymmetricAlgorithm
    public override ICryptoTransform CreateEncryptor(byte[] rgbKey, byte[] rgbIV);
// overrides SymmetricAlgorithm
    public override void GenerateIV( );
// overrides SymmetricAlgorithm
    public override void GenerateKey( );
// overrides SymmetricAlgorithm
}
```

This subclass of `Rijndael` is the default implementation of the Rijndael/AES algorithm. `RijndaelManaged` is implemented in managed code, rather than acting as a wrapper around the Windows Crypto API.

Hierarchy

```
System.Object      SymmetricAlgorithm(System.IDisposable)      Rijndael
RijndaelManaged
```

[Team LiB]

[\[Team LiB \]](#)

RNGCryptoServiceProvider

System.Security.Cryptography
(mscorlib.dll)

sealed
class

```
public sealed class RNGCryptoServiceProvider : RandomNumberGenerator {  
    // Public Constructors  
    public RNGCryptoServiceProvider( );  
    public RNGCryptoServiceProvider(byte[] rgb);  
    public RNGCryptoServiceProvider(CspParameters cspParams);  
    public RNGCryptoServiceProvider(string str);  
    // Public Instance Methods  
    public override void GetBytes(byte[] data);  
    // overrides RandomNumberGenerator  
    public override void GetNonZeroBytes(byte[] data);  
    // overrides RandomNumberGenerator  
    // Protected Instance Methods  
    protected override void Finalize( );  
    // overrides object  
}
```

This subclass of `RandomNumberGenerator` is the default random number generator implementation, and is a wrapper around the Windows Crypto API.

Hierarchy

System.Object RandomNumberGenerator RNGCryptoServiceProvider

[\[Team LiB \]](#)

[\[Team LiB \]](#)

RSA

disposable

System.Security.Cryptography
(mscorlib.dll)

*abstract
class*

```
public abstract class RSA : AsymmetricAlgorithm {
// Public Constructors
    public RSA( );
// Public Static Methods
    public static RSA Create( );
    public static RSA Create(string algName);
// Public Instance Methods
    public abstract byte[ ] DecryptValue(byte[ ] rgb);
    public abstract byte[ ] EncryptValue(byte[ ] rgb);
    public abstract RSAParameters ExportParameters(bool includePrivateParameters);
    public override void FromXmlString(string xmlString);
// overrides AsymmetricAlgorithm
    public abstract void ImportParameters(RSAParameters parameters);
    public override string ToXmlString(bool includePrivateParameters);
// overrides AsymmetricAlgorithm
}
```

This class extends `AsymmetricAlgorithm` and is the abstract representation of the RSA algorithm. Individual implementations of RSA will extend this class. RSA key pairs are represented by the `RSAParameters` structure, and can be imported and exported from the `RSA` class with the `ImportParameters()` and `ExportParameters()` methods.

The `EncryptValue()` and `DecryptValue()` methods defined by this class are not implemented by the `RSACryptoServiceProvider` class, which is the only implementation class included in the .NET Framework class library; data encryption must be performed by using methods defined in the implementation class.

This class does not define any members to support digital signatures; see the `RSACryptoServiceProvider` or the `RSAPKCS1SignatureFormatter` classes in this chapter for details of RSA digital signature support.

Hierarchy

```
System.Object      AsymmetricAlgorithm(System.IDisposable)      RSA
```

Subclasses

```
RSACryptoServiceProvider
```

Returned By

`System.Security.Cryptography.Xml.RSAKeyValue.Key`

Passed To

`System.Security.Cryptography.Xml.RSAKeyValue.{Key, RSAKeyValue()}`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

RSACryptoServiceProvider

disposable

System.Security.Cryptography
(mscorlib.dll)*sealed
class*

```

public sealed class RSACryptoServiceProvider : RSA {
    // Public Constructors
    public RSACryptoServiceProvider( );
    public RSACryptoServiceProvider(CspParameters parameters);
    public RSACryptoServiceProvider(int dwKeySize);
    public RSACryptoServiceProvider(int dwKeySize, CspParameters parameters);
    // Public Static Properties
    public static bool UseMachineKeyStore{set; get; }
    // Public Instance Properties
    public override string KeyExchangeAlgorithm{get; }
    // overrides AsymmetricAlgorithm
    public override int KeySize{get; }
    // overrides AsymmetricAlgorithm
    public bool PersistKeyInCsp{set; get; }
    public override string SignatureAlgorithm{get; }
    // overrides AsymmetricAlgorithm
    // Public Instance Methods
    public byte[ ] Decrypt(byte[ ] rgb, bool fOAEP);
    public override byte[ ] DecryptValue(byte[ ] rgb);
    // overrides RSA
    public byte[ ] Encrypt(byte[ ] rgb, bool fOAEP);
    public override byte[ ] EncryptValue(byte[ ] rgb);
    // overrides RSA
    public override RSAParameters ExportParameters( bool includePrivateParameters);
    // overrides RSA
    public override void ImportParameters( RSAParameters parameters);
    // overrides RSA
    public byte[ ] SignData(byte[ ] buffer, int offset, int count, object halg);
    public byte[ ] SignData(byte[ ] buffer, object halg);
    public byte[ ] SignData(System.IO.Stream inputStream, object halg);
    public byte[ ] SignHash(byte[ ] rgbHash, string str);
    public bool VerifyData(byte[ ] buffer, object halg, byte[ ] signature);
    public bool VerifyHash(byte[ ] rgbHash, string str, byte[ ] rgbSignature);
    // Protected Instance Methods
    protected override void Dispose(bool disposing);
    // overrides AsymmetricAlgorithm
    protected override void Finalize( );
    // overrides object
}

```

This subclass of `RSA` is the default implementation of the RSA algorithm and is a wrapper around the Windows Crypto API.

This class defines several new members in addition to those of the `RSA` class; the `Encrypt()` and `Decrypt()` methods encrypt and decrypt `System.Byte` arrays with the RSA algorithm. The `SignHash()` and `VerifyHash()` methods create and verify digital signatures for cryptographic hash codes. The `SignData()` method creates a cryptographic hash code for data read from a `System.IO.Stream` or a `System.Byte` array and creates a signature for the result.

`RSACryptoServiceProvider` does not implement the `EncryptValue()` and `DecryptValue()` methods defined in the `RSA` class.

Hierarchy

`System.Object` → `AsymmetricAlgorithm(System.IDisposable)` `RSA`
`RSACryptoServiceProvider`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

RSAOAEPKeyExchangeDeformatter

System.Security.Cryptography
(mscorlib.dll)

class

```
public class RSAOAEPKeyExchangeDeformatter : AsymmetricKeyExchangeDeformatter {  
    // Public Constructors  
    public RSAOAEPKeyExchangeDeformatter( );  
    public RSAOAEPKeyExchangeDeformatter(AsymmetricAlgorithm key);  
    // Public Instance Properties  
    public override string Parameters{set; get;}  
    // overrides AsymmetricKeyExchangeDeformatter  
    // Public Instance Methods  
    public override byte[ ] DecryptKeyExchange(byte[ ] rgbData);  
    // overrides AsymmetricKeyExchangeDeformatter  
    public override void SetKey(AsymmetricAlgorithm key);  
    // overrides AsymmetricKeyExchangeDeformatter  
}
```

This is an implementation of the `AsymmetricKeyExchangeDeformatter` class, which decrypts symmetric secret key values encrypted using the `RSAOAEPKeyExchangeFormatter` class.

Hierarchy

System.Object AsymmetricKeyExchangeDeformatter RSAOAEPKeyExchangeDeformatter

[\[Team LiB \]](#)

[\[Team LiB \]](#)

RSAOAEPKeyExchangeFormatter

System.Security.Cryptography
(mscorlib.dll)

class

```
public class RSAOAEPKeyExchangeFormatter : AsymmetricKeyExchangeFormatter {  
    // Public Constructors  
    public RSAOAEPKeyExchangeFormatter( );  
    public RSAOAEPKeyExchangeFormatter(AsymmetricAlgorithm key);  
    // Public Instance Properties  
    public byte[ ] Parameter{set; get; }  
    public override string Parameters{get; }  
    // overrides AsymmetricKeyExchangeFormatter  
    public RandomNumberGenerator Rng{set; get; }  
    // Public Instance Methods  
    public override byte[ ] CreateKeyExchange(byte[ ] rgbData);  
    // overrides AsymmetricKeyExchangeFormatter  
    public override byte[ ] CreateKeyExchange(byte[ ] rgbData, Type symAlgType);  
    // overrides AsymmetricKeyExchangeFormatter  
    public override void SetKey(AsymmetricAlgorithm key);  
    // overrides AsymmetricKeyExchangeFormatter  
}
```

This is an implementation of the `AsymmetricKeyExchangeFormatter` class, and formats a symmetric secret key value using the OAEP scheme. The formatted data is encrypted with the RSA algorithm.

In addition to the members defined by the parent class, `RSAOAEPKeyExchangeFormatter` defines the `Rng` property, which gets or sets the instance of `RandomNumberGenerator` used in the creation of the OAEP data.

Hierarchy

System.Object AsymmetricKeyExchangeFormatter RSAOAEPKeyExchangeFormatter

[\[Team LiB \]](#)

[\[Team LiB \]](#)

RSAParameters

serializable

System.Security.Cryptography
(mscorlib.dll)*struct*

```
public struct RSAParameters {  
    // Public Instance Fields  
    public byte[] D;  
    public byte[] DP;  
    public byte[] DQ;  
    public byte[] Exponent;  
    public byte[] InverseQ;  
    public byte[] Modulus;  
    public byte[] P;  
    public byte[] Q;  
}
```

This structure represents the parameters of an RSA key, expressed as a set of `Byte` arrays. Key parameters are exported from the `RSA` class with the `RSA.ExportParameters()` method. Key parameters are imported into the `RSA` class with the `RSA.ImportParameters()` method.

Hierarchy

```
System.Object      System.ValueType      RSAParameters
```

Returned By

```
RSA.ExportParameters( )
```

Passed To

```
RSA.ImportParameters( )
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

RSAPKCS1KeyExchangeDeformatter

System.Security.Cryptography
(mscorlib.dll)

class

```
public class RSAPKCS1KeyExchangeDeformatter : AsymmetricKeyExchangeDeformatter {  
    // Public Constructors  
    public RSAPKCS1KeyExchangeDeformatter( );  
    public RSAPKCS1KeyExchangeDeformatter(AsymmetricAlgorithm key);  
    // Public Instance Properties  
    public override string Parameters{set; get;}  
    // overrides AsymmetricKeyExchangeDeformatter  
    public RandomNumberGenerator RNG{set; get; }  
    // Public Instance Methods  
    public override byte[] DecryptKeyExchange(byte[] rgbIn);  
    // overrides AsymmetricKeyExchangeDeformatter  
    public override void SetKey(AsymmetricAlgorithm key);  
    // overrides AsymmetricKeyExchangeDeformatter  
}
```

This is an implementation of the `AsymmetricKeyExchangeDeformatter` class, which decrypts symmetric secret key values encrypted using the `RSAPKCS1KeyExchangeFormatter` class.

Hierarchy

System.Object AsymmetricKeyExchangeDeformatter RSAPKCS1KeyExchangeDeformatter

[\[Team LiB \]](#)

[\[Team LiB \]](#)

RSAPKCS1KeyExchangeFormatter

System.Security.Cryptography
(mscorlib.dll)

class

```
public class RSAPKCS1KeyExchangeFormatter : AsymmetricKeyExchangeFormatter {  
    // Public Constructors  
    public RSAPKCS1KeyExchangeFormatter( );  
    public RSAPKCS1KeyExchangeFormatter(AsymmetricAlgorithm key);  
    // Public Instance Properties  
    public override string Parameters{get;  
    // overrides AsymmetricKeyExchangeFormatter  
    public RandomNumberGenerator Rng{set; get; }  
    // Public Instance Methods  
    public override byte[ ] CreateKeyExchange(byte[ ] rgbData);  
    // overrides AsymmetricKeyExchangeFormatter  
    public override byte[ ] CreateKeyExchange(byte[ ] rgbData, Type symAlgType);  
    // overrides AsymmetricKeyExchangeFormatter  
    public override void SetKey(AsymmetricAlgorithm key);  
    // overrides AsymmetricKeyExchangeFormatter  
}
```

This is an implementation of the `AsymmetricKeyExchangeFormatter` class that formats a symmetric secret key value using the PKCS #1 scheme. The formatted data is encrypted using the RSA algorithm.

Hierarchy

System.Object AsymmetricKeyExchangeFormatter RSAPKCS1KeyExchangeFormatter

[\[Team LiB \]](#)

[\[Team LiB \]](#)

RSAPKCS1SignatureDeformatter

System.Security.Cryptography
(mscorlib.dll)

class

```
public class RSAPKCS1SignatureDeformatter : AsymmetricSignatureDeformatter {  
    // Public Constructors  
    public RSAPKCS1SignatureDeformatter( );  
    public RSAPKCS1SignatureDeformatter(AsymmetricAlgorithm key);  
    // Public Instance Methods  
    public override void SetHashAlgorithm(string strName);  
    // overrides AsymmetricSignatureDeformatter  
    public override void SetKey(AsymmetricAlgorithm key);  
    // overrides AsymmetricSignatureDeformatter  
    public override bool VerifySignature(byte[ ] rgbHash, byte[ ] rgbSignature);  
    // overrides AsymmetricSignatureDeformatter  
}
```

This child of the `AsymmetricSignatureDeformatter` class verifies RSA signatures created with the `RSAPKCS1SignatureFormatter` class.

Hierarchy

System.Object AsymmetricSignatureDeformatter RSAPKCS1SignatureDeformatter

[\[Team LiB \]](#)

[\[Team LiB \]](#)

RSAPKCS1SignatureFormatter

System.Security.Cryptography
(mscorlib.dll)

class

```
public class RSAPKCS1SignatureFormatter : AsymmetricSignatureFormatter {  
    // Public Constructors  
    public RSAPKCS1SignatureFormatter( );  
    public RSAPKCS1SignatureFormatter(AsymmetricAlgorithm key);  
    // Public Instance Methods  
    public override byte[ ] CreateSignature(byte[ ] rgbHash);  
    // overrides AsymmetricSignatureFormatter  
    public override void SetHashAlgorithm(string strName);  
    // overrides AsymmetricSignatureFormatter  
    public override void SetKey(AsymmetricAlgorithm key);  
    // overrides AsymmetricSignatureFormatter  
}
```

This class extends the `AsymmetricSignatureFormatter` class and creates digital signatures using the RSA algorithm. Signatures created by `RSAPKCS1SignatureFormatter` can be verified with the `RSAPKCS1SignatureDeformatter` class.

Hierarchy

System.Object AsymmetricSignatureFormatter RSAPKCS1SignatureFormatter

[\[Team LiB \]](#)

[\[Team LiB \]](#)

SHA1

disposable

System.Security.Cryptography
(mscorlib.dll)

*abstract
class*

```
public abstract class SHA1 : HashAlgorithm {  
    // Protected Constructors  
    protected SHA1( );  
    // Public Static Methods  
    public static SHA1 Create( );  
    public static SHA1 Create(string hashName);  
}
```

This class extends `HashAlgorithm` and is the abstract representation of the SHA-1 algorithm, which creates a 160-bit hash code. Individual implementations of SHA-1 extend this class. The .NET Framework class library includes the `SHA1CryptoServiceProvider` class, which is the default implementation.

Hierarchy

```
System.Object      HashAlgorithm(ICryptoTransform, System.IDisposable)      SHA1
```

Subclasses

```
SHA1CryptoServiceProvider, SHA1Managed
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

SHA1CryptoServiceProvider

disposable

System.Security.Cryptography
(mscorlib.dll)*sealed
class*

```

public sealed class SHA1CryptoServiceProvider : SHA1 {
// Public Constructors
    public SHA1CryptoServiceProvider( );
// Public Instance Methods
    public override void Initialize( );
    // overrides HashAlgorithm
// Protected Instance Methods
    protected override void Dispose(bool disposing);
    // overrides HashAlgorithm
    protected override void Finalize( );
    // overrides object
    protected override void HashCore(byte[ ] rgb, int ibStart, int cbSize);
    // overrides HashAlgorithm
    protected override byte[ ] HashFinal( );
    // overrides HashAlgorithm
}

```

This subclass of **SHA1** is the default implementation of the SHA-1 algorithm and is a wrapper around the Windows Crypto API.

Hierarchy

```

System.Object    HashAlgorithm(ICryptoTransform, System.IDisposable)    SHA1
SHA1CryptoServiceProvider

```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

SHA1Managed

disposable

System.Security.Cryptography
(mscorlib.dll)*class*

```

public class SHA1Managed : SHA1 {
// Public Constructors
    public SHA1Managed( );
// Public Instance Methods
    public override void Initialize( );
// overrides HashAlgorithm
// Protected Instance Methods
    protected override void HashCore(byte[ ] rgb, int ibStart, int cbSize);
// overrides HashAlgorithm
    protected override byte[ ] HashFinal( );
// overrides HashAlgorithm
}

```

This subclass of `SHA1` is a managed-code implementation of the SHA-1 algorithm.

Hierarchy

```

System.Object      HashAlgorithm(ICryptoTransform, System.IDisposable)      SHA1
SHA1Managed

```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

SHA256

disposable

System.Security.Cryptography
(mscorlib.dll)*abstract*
class

```
public abstract class SHA256 : HashAlgorithm {  
    // Public Constructors  
    public SHA256( );  
    // Public Static Methods  
    public static SHA256 Create( );  
    public static SHA256 Create(string hashName);  
}
```

This class extends `HashAlgorithm` and is the abstract representation of the SHA-256 algorithm, which creates a 256-bit hash code. Individual implementations of SHA-256 extend this class. The .NET Framework class library includes the `SHA256Managed` class, which is the default implementation.

Hierarchy

System.Object HashAlgorithm(ICryptoTransform, System.IDisposable) SHA256

Subclasses

SHA256Managed

[\[Team LiB \]](#)

[\[Team LiB \]](#)

SHA256Managed

disposable

System.Security.Cryptography
(mscorlib.dll)*class*

```
public class SHA256Managed : SHA256 {  
    // Public Constructors  
    public SHA256Managed( );  
    // Public Instance Methods  
    public override void Initialize( );  
    // overrides HashAlgorithm  
    // Protected Instance Methods  
    protected override void HashCore(byte[ ] rgb, int ibStart, int cbSize);  
    // overrides HashAlgorithm  
    protected override byte[ ] HashFinal( );  
    // overrides HashAlgorithm  
}
```

This subclass of `SHA256` is the default implementation of the SHA-256 algorithm. `SHA256Managed` is implemented in managed code, rather than acting as a wrapper around the Windows Crypto API.

Hierarchy

```
System.Object      HashAlgorithm(ICryptoTransform, System.IDisposable)      SHA256  
SHA256Managed
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

SHA384

disposable

System.Security.Cryptography
(mscorlib.dll)

*abstract
class*

```
public abstract class SHA384 : HashAlgorithm {  
    // Public Constructors  
    public SHA384( );  
    // Public Static Methods  
    public static SHA384 Create( );  
    public static SHA384 Create(string hashName);  
}
```

This class extends `HashAlgorithm` and is the abstract representation of the SHA-384 algorithm, which creates a 384-bit hash code. Individual implementations of SHA-384 extend this class. The .NET Framework class library includes the `SHA384Managed` class, which is the default implementation.

Hierarchy

System.Object HashAlgorithm(ICryptoTransform, System.IDisposable) SHA384

Subclasses

SHA384Managed

[\[Team LiB \]](#)

[\[Team LiB \]](#)

SHA384Managed

disposable

System.Security.Cryptography
(mscorlib.dll)*class*

```

public class SHA384Managed : SHA384 {
    // Public Constructors
    public SHA384Managed( );
    // Public Instance Methods
    public override void Initialize( );
    // overrides HashAlgorithm
    // Protected Instance Methods
    protected override void HashCore(byte[ ] rgb, int ibStart, int cbSize);
    // overrides HashAlgorithm
    protected override byte[ ] HashFinal( );
    // overrides HashAlgorithm
}

```

This subclass of `SHA384` is the default implementation of the SHA-384 algorithm. `SHA384Managed` is implemented in managed code, rather than acting as a wrapper around the Windows Crypto API.

Hierarchy

```

System.Object      HashAlgorithm(ICryptoTransform, System.IDisposable)      SHA384
SHA384Managed

```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

SHA512

disposable

System.Security.Cryptography
(mscorlib.dll)*abstract*
class

```
public abstract class SHA512 : HashAlgorithm {  
    // Public Constructors  
    public SHA512( );  
    // Public Static Methods  
    public static SHA512 Create( );  
    public static SHA512 Create(string hashName);  
}
```

This class extends `HashAlgorithm` and is the abstract representation of the SHA-512 algorithm, which creates a 512-bit hash code. Individual implementations of SHA-512 extend this class. The .NET Framework class library includes the `SHA512Managed` class, which is the default implementation.

Hierarchy

System.Object HashAlgorithm(ICryptoTransform, System.IDisposable) SHA512

Subclasses

SHA512Managed

[\[Team LiB \]](#)

[\[Team LiB \]](#)

SHA512Managed

disposable

System.Security.Cryptography
(mscorlib.dll)*class*

```
public class SHA512Managed : SHA512 {
// Public Constructors
    public SHA512Managed( );
// Public Instance Methods
    public override void Initialize( );
    // overrides HashAlgorithm
// Protected Instance Methods
    protected override void HashCore(byte[ ] rgb, int ibStart, int cbSize);
    // overrides HashAlgorithm
    protected override byte[ ] HashFinal( );
    // overrides HashAlgorithm
}
```

This subclass of `SHA512` is the default implementation of the SHA-512 algorithm. `SHA512Managed` is implemented in managed code, rather than acting as a wrapper around the Windows Crypto API.

Hierarchy

System.Object	HashAlgorithm(ICryptoTransform, System.IDisposable)	SHA512
SHA512Managed		

[\[Team LiB \]](#)

[Team LiB]

SignatureDescription

System.Security.Cryptography
(mscorlib.dll)

class

```
public class SignatureDescription {  
    // Public Constructors  
    public SignatureDescription( );  
    public SignatureDescription(System.Security.SecurityElement el);  
    // Public Instance Properties  
    public string DeformatterAlgorithm{set; get; }  
    public string DigestAlgorithm{set; get; }  
    public string FormatterAlgorithm{set; get; }  
    public string KeyAlgorithm{set; get; }  
    // Public Instance Methods  
    public virtual AsymmetricSignatureDeformatter CreateDeformatter(AsymmetricAlgorithm key);  
    public virtual HashAlgorithm CreateDigest( );  
    public virtual AsymmetricSignatureFormatter CreateFormatter(AsymmetricAlgorithm key)  
}
```

This class creates instances of the `AsymmetricSignatureFormatter` and `AsymmetricSignatureDeformatter` implementation classes based on the values of the class properties. The `FormatterAlgorithm` and `DeformatterAlgorithm` properties specify the formatting scheme; the `DigestAlgorithm` and the `KeyAlgorithm` properties specify the algorithms to use.

Instances of the formatter and deformatter implementation classes are created with the `CreateFormatter` and `CreateDeformatter()` methods.

[Team LiB]

[Team LiB]

SymmetricAlgorithm

disposable

System.Security.Cryptography
(mscorlib.dll)*abstract
class*

```

public abstract class SymmetricAlgorithm : IDisposable {
// Public Constructors
    public SymmetricAlgorithm( );
// Protected Instance Fields
    protected int BlockSizeValue;
    protected int FeedbackSizeValue;
    protected byte[] IVValue;
    protected int KeySizeValue;
    protected byte[] KeyValue;
    protected KeySizes[] LegalBlockSizesValue;
    protected KeySizes[] LegalKeySizesValue;
    protected CipherMode ModeValue;
    protected PaddingMode PaddingValue;
// Public Instance Properties
    public virtual int BlockSize {set; get; }
    public virtual int FeedbackSize {set; get; }
    public virtual byte[] IV {set; get; }
    public virtual byte[] Key {set; get; }
    public virtual int KeySize {set; get; }
    public virtual KeySizes[] LegalBlockSizes {get; }
    public virtual KeySizes[] LegalKeySizes {get; }
    public virtual CipherMode Mode {set; get; }
    public virtual PaddingMode Padding {set; get; }
// Public Static Methods
    public static SymmetricAlgorithm Create( );
    public static SymmetricAlgorithm Create(string algName);
// Public Instance Methods
    public void Clear( );
    public virtual ICryptoTransform CreateDecryptor( );
    public abstract ICryptoTransform CreateDecryptor(byte[] rgbKey, byte[] rgbIV);
    public virtual ICryptoTransform CreateEncryptor( );
    public abstract ICryptoTransform CreateEncryptor(byte[] rgbKey, byte[] rgbIV);
    public abstract void GenerateIV( );
    public abstract void GenerateKey( );
    public bool ValidKeySize(int bitLength);
// Protected Instance Methods
    protected virtual void Dispose(bool disposing);
    protected override void Finalize( );
// overrides object
}

```

A symmetric algorithm is one where the sender and the recipient share the same knowledge (i.e. both parties know the secret key). All symmetric algorithm implementations inherit from this abstract class. Instances of implementation classes are created using the `Create()` method, which accepts the name of an implementation as a `System.String` argument. An instance of the default algorithm is created if no name is specified; the system administrator can configure the default algorithm.

The `LegalKeySizes` and `LegalBlockSizes` properties return arrays of the `KeySizes` class, which specify the set of key lengths and cipher block sizes supported by an implementation class. The `KeySize` and `BlockSize` properties get or set the current key and block lengths.

The `IV` and `Key` properties get or set the initialization vector (IV) and secret key to use for encryption or decryption. New values for these properties are created with the `GenerateIV()` and `GenerateKey()` methods; implementation classes will automatically create a new IV and secret key.

Subclasses of the `SymmetricAlgorithm` class do not perform encryption and decryption directly; the `CreateEncryptor()` and `CreateDecryptor()` methods return instances of the `ICryptoTransform` type to which these tasks are delegated.

Subclasses

`DES` , `RC2` , `Rijndael` , `TripleDES`

[Team LiB]

[Team LiB]

ToBase64Transform

disposable

System.Security.Cryptography
(mscorlib.dll)*class*

```
public class ToBase64Transform : ICryptoTransform, IDisposable {  
    // Public Constructors  
    public ToBase64Transform( );  
    // Public Instance Properties  
    public virtual bool CanReuseTransform{get;}  
    // implements ICryptoTransform  
    public bool CanTransformMultipleBlocks{get;}  
    // implements ICryptoTransform  
    public int InputBlockSize{get;}  
    // implements ICryptoTransform  
    public int OutputBlockSize{get;}  
    // implements ICryptoTransform  
    // Public Instance Methods  
    public void Clear( );  
    public int TransformBlock(byte[ ] inputBuffer, int inputOffset, int inputCount,  
        byte[ ] outputBuffer, int outputOffset);  
    // implements ICryptoTransform  
    public byte[ ] TransformFinalBlock(byte[ ] inputBuffer, int inputOffset, int inputCount);  
    // implements ICryptoTransform  
    // Protected Instance Methods  
    protected virtual void Dispose(bool disposing);  
    protected override void Finalize( );  
    // overrides object  
}
```

This implementation of the `ICryptoTransform` interface encodes data using Base64. This class is used in conjunction with the `CryptoStream` class.

[Team LiB]

[\[Team LiB \]](#)

TripleDES

disposable

System.Security.Cryptography
(mscorlib.dll)*abstract
class*

```

public abstract class TripleDES : SymmetricAlgorithm {
// Public Constructors
    public TripleDES( );
// Public Instance Properties
    public override byte[ ] Key{set; get;
// overrides SymmetricAlgorithm
// Public Static Methods
    public static TripleDES Create( );
    public static TripleDES Create(string str);
    public static bool IsWeakKey(byte[ ] rgbKey);
}

```

This class extends `SymmetricAlgorithm` and is the abstract representation of the Triple-DES algorithm, which is based on the DES. Individual implementations of Triple-DES will extend this class. The .NET Framework class library includes the `TriplesDESCryptoServiceProvider` class, which is the default implementation.

In addition to the members defined by the parent class, `TriplesDES` implements the `IsWeakKey()` method, which determines if the selected cryptographic key results in a cipher that is easy to break.

Hierarchy

```

System.Object      SymmetricAlgorithm(System.IDisposable)      TripleDES

```

Subclasses

```

TriplesDESCryptoServiceProvider

```

[\[Team LiB \]](#)

[Team LiB]

TripleDESCryptoServiceProvider disposable

System.Security.Cryptography *sealed*
 (mscorlib.dll) *class*

```
public sealed class TripleDESCryptoServiceProvider : TripleDES {
// Public Constructors
    public TripleDESCryptoServiceProvider( );
// Public Instance Methods
    public override ICryptoTransform CreateDecryptor(byte[] rgbKey, byte[] rgbIV);
// overrides SymmetricAlgorithm
    public override ICryptoTransform CreateEncryptor(byte[] rgbKey, byte[] rgbIV);
// overrides SymmetricAlgorithm
    public override void GenerateIV( );
// overrides SymmetricAlgorithm
    public override void GenerateKey( );
// overrides SymmetricAlgorithm
}
```

This subclass of `TripleDES` is the default implementation of the Triple-DES algorithm and is a wrapper around the Windows Crypto API.

Hierarchy

```
System.Object      SymmetricAlgorithm(System.IDisposable)      TripleDES
TripleDESCryptoServiceProvider
```

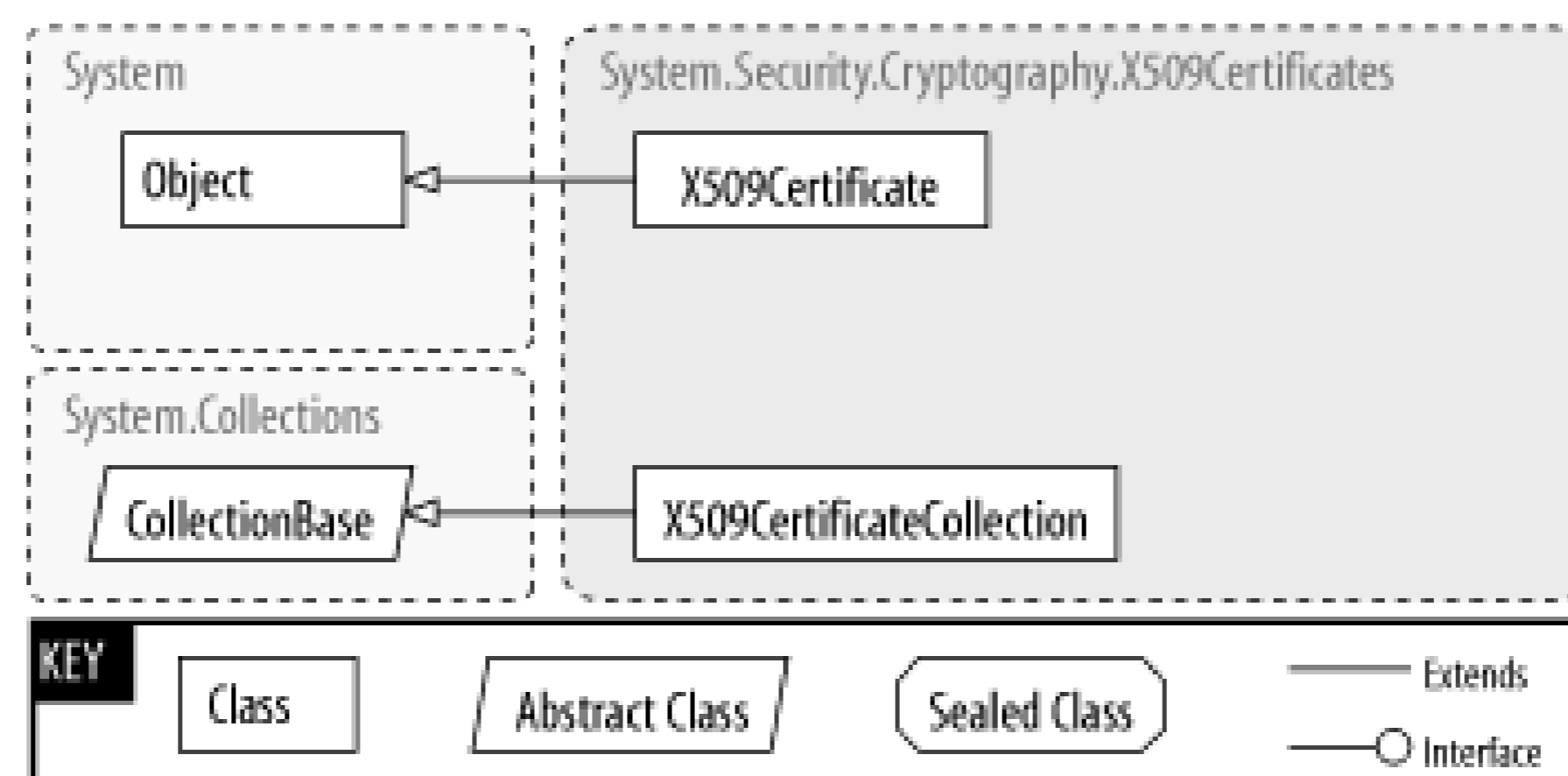
[Team LiB]

[Team LiB]

Chapter 25. The System.Security.Cryptography.X509Certificates Namespace

The `System.Security.Cryptography.X509Certificates` namespace provides support for Authenticode/X.509 v3 digital certificates. Figure 25-1 shows the types in this namespace.

Figure 25-1. The System.Security.Cryptography.X509Certificates namespace



[Team LiB]

[\[Team LiB \]](#)

X509Certificate

CF 1.0, serializable

 System.Security.Cryptography.X509Certificates *class*
 (mscorlib.dll)

```

public class X509Certificate {
    // Public Constructors
    public X509Certificate(byte[] data);
    public X509Certificate(IntPtr handle);
    public X509Certificate(X509Certificate cert);
    // Public Static Methods
    public static X509Certificate CreateFromCertFile(string filename);
    public static X509Certificate CreateFromSignedFile(string filename);
    // Public Instance Methods
    public virtual bool Equals(X509Certificate other);
    public virtual byte[] GetCertHash( );
    public virtual string GetCertHashString( );
    public virtual string GetEffectiveDateString( );
    public virtual string GetExpirationDateString( );
    public virtual string GetFormat( );
    public override int GetHashCode( );
    // overrides object
    public virtual string GetIssuerName( );
    public virtual string GetKeyAlgorithm( );
    public virtual byte[] GetKeyAlgorithmParameters( );
    public virtual string GetKeyAlgorithmParametersString( );
    public virtual string GetName( );
    public virtual byte[] GetPublicKey( );
    public virtual string GetPublicKeyString( );
    public virtual byte[] GetRawCertData( );
    public virtual string GetRawCertDataString( );
    public virtual byte[] GetSerialNumber( );
    public virtual string GetSerialNumberString( );
    public override string ToString( );
    // overrides object
    public virtual string ToString(bool fVerbose);
}

```

The `X509Certificate` class represents an X509 v3 digital certificate. Instances of this class are created using the static `CreateFromCertFile()` and `CreateFromSignedFile()` methods, which obtain the certificate details from an X509 v3 certificate file or a signed file, respectively. The class defines 17 methods that provide information about the certificate; for example, the `GetName()` method returns the name of the principle to which the certificate was issued and the `GetIssuerName()` method returns the name of the certification authority. Some methods are paired together, returning the same data formatted in different ways; for example, the `GetPublicKey()` method returns the certificate public key as an array of bytes, while the `GetPublicKeyString()`

method return the key as a hexadecimal string.

Returned By

```
System.Net.ServicePoint.{Certificate, ClientCertificate},  
System.Reflection.Module.GetSignerCertificate( ), X509CertificateCollection.this,  
X509CertificateEnumerator.Current,  
System.Security.Permissions.PublisherIdentityPermission.Certificate,  
System.Security.Policy.Publisher.Certificate,  
System.Security.Policy.PublisherMembershipCondition.Certificate
```

Passed To

Multiple types

[\[Team LiB \]](#)

[\[Team LiB \]](#)

X509CertificateCollection

serializable

System.Security.Cryptography.X509Certificates
(system.dll) *class*

```
public class X509CertificateCollection : CollectionBase {
// Public Constructors
    public X509CertificateCollection( );
    public X509CertificateCollection(X509Certificate[] value);
    public X509CertificateCollection(X509CertificateCollection value);
// Public Instance Properties
    public X509Certificate this{set; get; }
// Public Instance Methods
    public int Add(X509Certificate value);
    public void AddRange(X509Certificate[] value);
    public void AddRange(X509CertificateCollection value);
    public bool Contains(X509Certificate value);
    public void CopyTo(X509Certificate[] array, int index);
    public X509CertificateEnumerator GetEnumerator( );
    public override int GetHashCode( ); // overrides object
    public int IndexOf(X509Certificate value);
    public void Insert(int index, X509Certificate value);
    public void Remove(X509Certificate value);
}
```

This class defines a collection of `X509Certificate` objects, defining methods to add, remove, and enumerate certificates.

Hierarchy

```
System.Object      System.Collections.CollectionBase(System.Collections.IList,
System.Collections ICollection, System.Collections.IEnumerable)
X509CertificateCollection
```

Returned By

```
System.Net.HttpWebRequest.ClientCertificates
```

Passed To

```
X509CertificateEnumerator.X509CertificateEnumerator( )
```

[\[Team LiB \]](#)

[Team LiB]

X509CertificateCollection.X509CertificateEnumerator

System.Security.Cryptography.X509Certificates *class*
(system.dll)

```
public class X509CertificateCollection.X509CertificateEnumerator : IEnumerator {  
    // Public Constructors  
    public X509CertificateCollection.X509CertificateEnumerator (X509CertificateCollection  
    // Public Instance Properties  
    public X509Certificate Current {get; }  
    // Public Instance Methods  
    public bool MoveNext( );  
    // implements IEnumerator  
    public void Reset( );  
    // implements IEnumerator  
}
```

This class implements the `System.Collections.IEnumerator` interface to enumerate the `X509CertificateCollection` contained by an `X509CertificateCollection`. Instances of this class are typically obtained through the `X509CertificateCollection.GetEnumerator()` method.

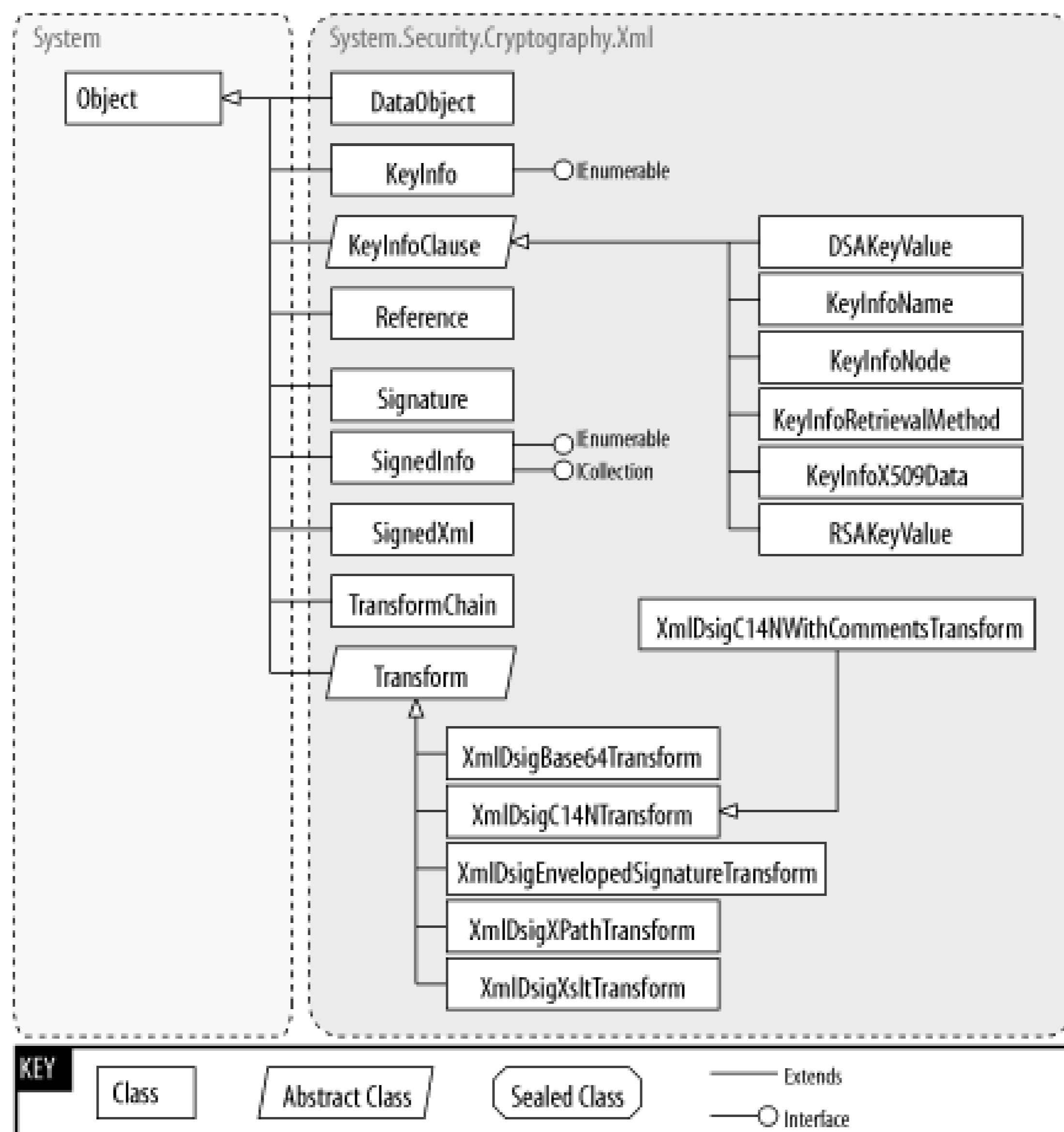
[Team LiB]

[Team LiB]

Chapter 26. The System.Security.Cryptography.Xml Namespace

The `System.Security.Cryptography.Xml` namespace provides support for the XML Signature standard, which specifies a mechanism for representing cryptographic signatures as XML documents. The full specification for the XML Signature standard can be found at <http://www.w3.org/TR/xmlsig-core/>. Figure 26-1 shows the types in this namespace.

Figure 26-1. The System.Security.Cryptography.Xml namespace



[Team LiB]

[Team LiB]

DataObject

System.Security.Cryptography.Xml
(system.security.dll)

class

```
public class DataObject {  
    // Public Constructors  
    public DataObject( );  
    public DataObject(string id, string mimeType, string encoding, System.Xml.XmlElement  
// Public Instance Properties  
    public XmlNodeList Data {set; get; }  
    public string Encoding {set; get; }  
    public string Id {set; get; }  
    public string MimeType {set; get; }  
// Public Instance Methods  
    public XmlElement GetXml( );  
    public void LoadXml(System.Xml.XmlElement value);  
}
```

The `DataObject` class specifies data embedded within the XML signature document; the data is signed by creating an instance of `Reference` that specifies local data (where the value of the `Reference.Id` property is with "#"). Instances of this class are associated with a signature using the `SignedXml.AddObject()` method.

Passed To

`Signature.AddObject()`, `SignedXml.AddObject()`

[Team LiB]

[\[Team LiB \]](#)

DSAKeyValue

System.Security.Cryptography.Xml
(system.security.dll)

class

```
public class DSAKeyValue : KeyInfoClause {  
    // Public Constructors  
    public DSAKeyValue( );  
    public DSAKeyValue(System.Security.Cryptography.DSA key);  
    // Public Instance Properties  
    public DSA Key{set; get; }  
    // Public Instance Methods  
    public override XmlElement GetXml( );  
    // overrides KeyInfoClause  
    public override void LoadXml(System.Xml.XmlElement value);  
    // overrides KeyInfoClause  
}
```

This class extends `KeyInfoClause` and represents a DSA key, specified by the `Key` property. When an instance of this class is associated with the `KeyInfo` class, the resulting XML Signature document will include the DSA public key.

Hierarchy

System.Object KeyInfoClause DSAKeyValue

[\[Team LiB \]](#)

[\[Team LiB \]](#)

KeyInfo

System.Security.Cryptography.Xml
(system.security.dll)

class

```
public class KeyInfo : IEnumerable {  
    // Public Constructors  
    public KeyInfo( );  
    // Public Instance Properties  
    public int Count{get; }  
    public string Id{set; get; }  
    // Public Instance Methods  
    public void AddClause(KeyInfoClause clause);  
    public IEnumerator GetEnumerator( );  
    // implements IEnumerable  
    public IEnumerator GetEnumerator(Type requestedObjectType);  
    public XmlElement GetXml( );  
    public void LoadXml(System.Xml.XmlElement value);  
}
```

The **KeyInfo** class is used to include information in an XML Signature that assists the recipient in identifying the cryptographic key required for verification. Individual pieces of information are represented by implementations of the **KeyInfoClause** class, instances of which are associated with this class through the **AddClause**() method. The **SignedXml.KeyInfo** property gets or sets the **KeyInfo** instance included in an XML Signature document.

Returned By

Signature.KeyInfo, **SignedXml.KeyInfo**

Passed To

Signature.KeyInfo, **SignedXml.KeyInfo**

[\[Team LiB \]](#)

[\[Team LiB \]](#)

KeyInfoClause

System.Security.Cryptography.Xml (system.security.dll) *abstract class*

```
public abstract class KeyInfoClause {  
    // Public Constructors  
    public KeyInfoClause( );  
    // Public Instance Methods  
    public abstract XmlElement GetXml( );  
    public abstract void LoadXml(System.Xml.XmlElement element);  
}
```

This is the abstract parent for all classes representing the identity of a cryptographic key. Instances of this class are associated with an XML Signature document through the `KeyInfo` class.

Subclasses

`DSAKeyValue`, `KeyInfoName`, `KeyInfoNode`, `KeyInfoRetrievalMethod`, `KeyInfoX509Data`, `RSAKeyValue`

Passed To

`KeyInfo.AddClause()`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

KeyInfoName

System.Security.Cryptography.Xml
(system.security.dll)

class

```
public class KeyInfoName : KeyInfoClause {  
    // Public Constructors  
    public KeyInfoName( );  
    // Public Instance Properties  
    public string Value{set; get; }  
    // Public Instance Methods  
    public override XmlElement GetXml( );  
    // overrides KeyInfoClause  
    public override void LoadXml(System.Xml.XmlElement value);  
    // overrides KeyInfoClause  
}
```

This class represents an identifier that may be used in identifying the key required to validate an XML Signature. The **Value** property gets and sets the identifier value, which is dependent on the format of the key used to create the signature.

Hierarchy

System.Object KeyInfoClause KeyInfoName

[\[Team LiB \]](#)

[\[Team LiB \]](#)

KeyInfoNode

System.Security.Cryptography.Xml
(system.security.dll)

class

```
public class KeyInfoNode : KeyInfoClause {  
    // Public Constructors  
    public KeyInfoNode( );  
    public KeyInfoNode(System.Xml.XmlElement node);  
    // Public Instance Properties  
    public XmlElement Value{set; get; }  
    // Public Instance Methods  
    public override XmlElement GetXml( );  
    // overrides KeyInfoClause  
    public override void LoadXml(System.Xml.XmlElement value);  
    // overrides KeyInfoClause  
}
```

This class allows the .NET support for XML Signatures to include information about cryptographic key that are not supported by the default cryptographic algorithms. The **Value** property gets or sets the XML content of a **KeyInfoNode** instance.

Hierarchy

System.Object KeyInfoClause KeyInfoNode

[\[Team LiB \]](#)

[\[Team LiB \]](#)

KeyInfoRetrievalMethod

System.Security.Cryptography.Xml
(system.security.dll)

class

```
public class KeyInfoRetrievalMethod : KeyInfoClause {  
    // Public Constructors  
    public KeyInfoRetrievalMethod( );  
    public KeyInfoRetrievalMethod(string strUri);  
    // Public Instance Properties  
    public string Uri{set; get; }  
    // Public Instance Methods  
    public override XmlElement GetXml( );  
    // overrides KeyInfoClause  
    public override void LoadXml(System.Xml.XmlElement value);  
    // overrides KeyInfoClause  
}
```

The `KeyInfoRetrievalMethod` class allows an XML Signature document to refer to information outside of the signature document that identifies the key required to verify the signature. The `Value` property gets or sets the URL representing the location of the information.

Hierarchy

System.Object KeyInfoClause KeyInfoRetrievalMethod

[\[Team LiB \]](#)

[Team LiB]

KeyInfoX509Data

System.Security.Cryptography.Xml
(system.security.dll)

class

```
public class KeyInfoX509Data : KeyInfoClause {
// Public Constructors
    public KeyInfoX509Data( );
    public KeyInfoX509Data(byte[ ] rgbCert);
    public KeyInfoX509Data(System.Security.Cryptography.X509Certificates.X509Certificate
// Public Instance Properties
    public ArrayList Certificates{get; }
    public byte[ ] CRL{set; get; }
    public ArrayList IssuerSerials{get; }
    public ArrayList SubjectKeyIds{get; }
    public ArrayList SubjectNames{get; }
// Public Instance Methods
    public void AddCertificate(System.Security.Cryptography.X509Certificates.X509Certificate
    public void AddIssuerSerial(string issuerName, string serialNumber);
    public void AddSubjectKeyId(byte[ ] subjectKeyId);
    public void AddSubjectName(string subjectName);
    public override XmlElement GetXml( );
// overrides KeyInfoClause
    public override void LoadXml(System.Xml.XmlElement element);
// overrides KeyInfoClause
}
```

The `KeyInfoX509Data` class extends the `KeyInfoClause` class and specifies an X.509 v3 certificate that required to validate an XML Signature. The `AddCertificate()` method includes a certificate in the info represented by the `System.Security.Cryptography.X509Certificates.X509Certificate` class.

Hierarchy

System.Object KeyInfoClause KeyInfoX509Data

[Team LiB]

[\[Team LiB \]](#)

Reference

System.Security.Cryptography.Xml
(system.security.dll)

class

```

public class Reference {
// Public Constructors
    public Reference( );
    public Reference(System.IO.Stream stream);
    public Reference(string uri);
// Public Instance Properties
    public string DigestMethod{set; get; }
    public byte[ ] DigestValue{set; get; }
    public string Id{set; get; }
    public TransformChain TransformChain{get; }
    public string Type{set; get; }
    public string Uri{set; get; }
// Public Instance Methods
    public void AddTransform(Transform transform);
    public XmlElement GetXml( );
    public void LoadXml(System.Xml.XmlElement value);
}

```

Instances of this class act as a reference to the signed data, expressed as a URI (using the `Uri` property) or as a local reference to an instance of the `DataObject` class (using the `Id` property). References are associated with a signature through the `SignedXml.AddReference()` method.

Implementations of the `Transform` class can be added to a reference with the `AddTransform()` method; the transformations will be applied in order before the data is signed. The algorithm used to create the cryptographic hash code is set with the `DigestMethod` property-the default value is <http://www.w3.org/2000/09/xmlsig#sha1>, which specifies the SHA-1 hash algorithm.

Passed To

`SignedInfo.AddReference()`, `SignedXml.AddReference()`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

RSAKeyValue

System.Security.Cryptography.Xml
(system.security.dll) *class*

```
public class RSAKeyValue : KeyInfoClause {  
    // Public Constructors  
    public RSAKeyValue( );  
    public RSAKeyValue(System.Security.Cryptography.RSA key);  
    // Public Instance Properties  
    public RSA Key{set; get; }  
    // Public Instance Methods  
    public override XmlElement GetXml( );  
    // overrides KeyInfoClause  
    public override void LoadXml(System.Xml.XmlElement value);  
    // overrides KeyInfoClause  
}
```

This class extends `KeyInfoClause` and represents an RSA key, specified by the `Key` property. When an instance of this class is associated with the `KeyInfo` class, the resulting XML Signature document will include the RSA public key.

Hierarchy

System.Object KeyInfoClause RSAKeyValue

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Signature

System.Security.Cryptography.Xml
(system.security.dll)

class

```
public class Signature {  
    // Public Constructors  
    public Signature( );  
    // Public Instance Properties  
    public string Id{set; get; }  
    public KeyInfo KeyInfo{set; get; }  
    public IList ObjectList{set; get; }  
    public byte[ ] SignatureValue{set; get; }  
    public SignedInfo SignedInfo{set; get; }  
    // Public Instance Methods  
    public void AddObject(DataObject dataObject);  
    public XmlElement GetXml( );  
    public void LoadXml(System.Xml.XmlElement value);  
}
```

This class represents an XML Signature; instances of this class are obtained through the `SignedXml.Signature` property. The `KeyInfo`, `ObjectList`, and `SignedInfo` properties provide information about the structure of the XML Signature document.

Returned By

`SignedXml.Signature`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

SignedInfo

System.Security.Cryptography.Xml
(system.security.dll)

class

```
public class SignedInfo : ICollection, IEnumerable {  
    // Public Constructors  
    public SignedInfo( );  
    // Public Instance Properties  
    public string CanonicalizationMethod{set; get; }  
    public int Count{get; }  
    // implements ICollection  
    public string Id{set; get; }  
    public bool IsReadOnly{get; }  
    public bool IsSynchronized{get; }  
    // implements ICollection  
    public ArrayList References{get; }  
    public string SignatureLength{set; get; }  
    public string SignatureMethod{set; get; }  
    public object SyncRoot{get; }  
    // implements ICollection  
    // Public Instance Methods  
    public void AddReference(Reference reference);  
    public void CopyTo(Array array, int index);  
    // implements ICollection  
    public IEnumerator GetEnumerator( );  
    // implements IEnumerable  
    public XmlElement GetXml( );  
    public void LoadXml(System.Xml.XmlElement value);  
}
```

The `SignedInfo` class provides information about an XML Signature; instances of this class are obtained through the `SignedXml.SignedInfo` property. The `SignatureMethod` property describes the signature and hash algorithms that have been used to create a signature. The `References` property provides access to the set of references contained by the signature.

Returned By

`Signature.SignedInfo`, `SignedXml.SignedInfo`

Passed To

`Signature.SignedInfo`

[\[Team LiB \]](#)

[Team LiB]

SignedXml

System.Security.Cryptography.Xml
(system.security.dll)

class

```

public class SignedXml {
    // Public Constructors
    public SignedXml( );
    public SignedXml(System.Xml.XmlDocument document);
    public SignedXml(System.Xml.XmlElement elem);
    // Public Static Fields
    public const string XmlDsigCanonicalizationUrl;
    // =http://www.w3.org/TR/2001/REC-xml-c14n-20010315
    public const string XmlDsigCanonicalizationWithCommentsUrl;
    // =http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments
    public const string XmlDsigDSAUrl;
    // =http://www.w3.org/2000/09/xmlldsig#dsa-sha1
    public const string XmlDsigHMACSHA1Url;
    // =http://www.w3.org/2000/09/xmlldsig#hmac-sha1
    public const string XmlDsigMinimalCanonicalizationUrl;
    // =http://www.w3.org/2000/09/xmlldsig#minimal
    public const string XmlDsigNamespaceUrl;
    // =http://www.w3.org/2000/09/xmlldsig#
    public const string XmlDsigRSASHA1Url;
    // =http://www.w3.org/2000/09/xmlldsig#rsa-sha1
    public const string XmlDsigSHA1Url;
    // =http://www.w3.org/2000/09/xmlldsig#sha1
    // Protected Instance Fields
    protected Signature m_signature;
    protected string m_strSigningKeyName;
    // Public Instance Properties
    public KeyInfo KeyInfo{set; get; }
    public XmlResolver Resolver{set; }
    public Signature Signature{get; }
    public string SignatureLength{get; }
    public string SignatureMethod{get; }
    public byte[] SignatureValue{get; }
    public SignedInfo SignedInfo{get; }
    public AsymmetricAlgorithm SigningKey{set; get; }
    public string SigningKeyName{set; get; }
    // Public Instance Methods
    public void AddObject(DataObject dataObject);
    public void AddReference(Reference reference);
    public bool CheckSignature( );
    public bool CheckSignature(System.Security.Cryptography.AsymmetricAlgorithm key);

```

```
public bool CheckSignature(System.Security.Cryptography.KeyedHashAlgorithm macAlg);  
public bool CheckSignatureReturningKey(out System.Security.Cryptography.AsymmetricAlgo  
public void ComputeSignature( );  
public void ComputeSignature(System.Security.Cryptography.KeyedHashAlgorithm macAlg)  
public virtual XmlElement GetIdElement(System.Xml.XmlDocument document, string idVal  
public XmlElement GetXml( );  
public void LoadXml(System.Xml.XmlElement value);  
// Protected Instance Methods  
protected virtual AsymmetricAlgorithm GetPublicKey( );  
}
```

The `SignedXml` class is the principle class used to create and verify XML signatures. References to data : signature document through the `AddReference()` method, while data objects are associated through th method. The `ComputeSignature()` method performs a signature operation, while the `CheckSignature(existing XML Signature.`

[Team LiB]

[\[Team LiB \]](#)

Transform

System.Security.Cryptography.Xml *abstract*
(system.security.dll) *class*

```
public abstract class Transform {
// Public Constructors
    public Transform( );
// Public Instance Properties
    public string Algorithm{set; get; }
    public abstract Type[] InputTypes{get; }
    public abstract Type[] OutputTypes{get; }
    public XmlResolver Resolver{set; }
// Public Instance Methods
    public abstract object GetOutput( );
    public abstract object GetOutput(Type type);
    public XmlElement GetXml( );
    public abstract void LoadInnerXml(System.Xml.XmlNodeList nodeList);
    public abstract void LoadInput(object obj);
// Protected Instance Methods
    protected abstract XmlNodeList GetInnerXml( );
}
```

This is the abstract parent for all transformation-implementation classes. Transformations are applied to data prior to the generation of the cryptographic hash code used in the digital signature. Transformations are associated with data via the `Reference.AddTransform()` method; multiple transformations can be associated with a `Reference`, and each will be applied to the data in sequence.

Subclasses

`XmlDsigBase64Transform`, `XmlDsigC14NTransform`, `XmlDsigEnvelopedSignatureTransform`,
`XmlDsigXPathTransform`, `XmlDsigXsltTransform`

Returned By

`TransformChain.this`

Passed To

`Reference.AddTransform()`, `TransformChain.Add()`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

TransformChain

System.Security.Cryptography.Xml
(system.security.dll)

class

```
public class TransformChain {  
    // Public Constructors  
    public TransformChain( );  
    // Public Instance Properties  
    public int Count {get; }  
    public Transform this {get; }  
    // Public Instance Methods  
    public void Add(Transform transform);  
    public IEnumerator GetEnumerator( );  
}
```

This class represents the sequence of transformations applied to data prior to creating a digital signature; each transformation is represented by an implementation of the `Transform` class. The number of transformations in the chain can be determined with the `Count` property, while individual transformations are accessible through the `Item` property (which is the C# class indexer). Instances of this class are obtained through the `Reference.TransformChain` property.

Returned By

`Reference.TransformChain`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

XmlDsigBase64Transform

System.Security.Cryptography.Xml
(system.security.dll)

class

```
public class XmlDsigBase64Transform : Transform {
// Public Constructors
    public XmlDsigBase64Transform( );
// Public Instance Properties
    public override Type[] InputTypes {get; }
// overrides Transform
    public override Type[] OutputTypes {get; }
// overrides Transform
// Public Instance Methods
    public override object GetOutput( );
// overrides Transform
    public override object GetOutput(Type type);
// overrides Transform
    public override void LoadInnerXml(System.Xml.XmlNodeList nodeList);
// overrides Transform
    public override void LoadInput(object obj);
// overrides Transform
// Protected Instance Methods
    protected override XmlNodeList GetInnerXml( );
// overrides Transform
}
```

This implementation of the abstract `Transform` class decodes data encoded with the Base 64 scheme.

Hierarchy

System.Object Transform XmlDsigBase64Transform

[\[Team LiB \]](#)

[\[Team LiB \]](#)

XmlDsigC14NTransform

System.Security.Cryptography.Xml
(system.security.dll)

class

```
public class XmlDsigC14NTransform : Transform {
    // Public Constructors
    public XmlDsigC14NTransform( );
    public XmlDsigC14NTransform(bool includeComments);
    // Public Instance Properties
    public override Type[] InputTypes {get; }
    // overrides Transform
    public override Type[] OutputTypes {get; }
    // overrides Transform
    // Public Instance Methods
    public override object GetOutput( );
    // overrides Transform
    public override object GetOutput(Type type);
    // overrides Transform
    public override void LoadInnerXml(System.Xml.XmlNodeList nodeList);
    // overrides Transform
    public override void LoadInput(object obj);
    // overrides Transform
    // Protected Instance Methods
    protected override XmlNodeList GetInnerXml( );
    // overrides Transform
}
```

This implementation of the abstract `Transform` class implements the C14N transformation, as defined by the XMLDSIG specification. C14N refers to the word "canonicalization," which is written as the letter C, followed by 14 other letters, followed by the letter N.

The XML Signature canonicalization transformation ensures that XML documents that have differing element ordering (but still contain the same data) are treated as representing the same data.

Hierarchy

System.Object Transform XmlDsigC14NTransform

Subclasses

XmlDsigC14NWithCommentsTransform

[\[Team LiB \]](#)

[\[Team LiB \]](#)

XmlDsigC14NWithCommentsTransform

System.Security.Cryptography.Xml (system.security.dll) *class*

```
public class XmlDsigC14NWithCommentsTransform : XmlDsigC14NTransform {  
    // Public Constructors  
    public XmlDsigC14NWithCommentsTransform( );  
}
```

This class extends `XmlDsigC14NTransform` in order to provide a C14N transformation that includes comments in the canonical output.

Hierarchy

```
System.Object    Transform    XmlDsigC14NTransform  
XmlDsigC14NWithCommentsTransform
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

XmlDsigEnvelopedSignatureTransform

System.Security.Cryptography.Xml
(system.security.dll) *class*

```
public class XmlDsigEnvelopedSignatureTransform : Transform {
    // Public Constructors
    public XmlDsigEnvelopedSignatureTransform( );
    public XmlDsigEnvelopedSignatureTransform(bool includeComments);
    // Public Instance Properties
    public override Type[] InputTypes {get; }
    // overrides Transform
    public override Type[] OutputTypes {get; }
    // overrides Transform
    // Public Instance Methods
    public override object GetOutput( );
    // overrides Transform
    public override object GetOutput(Type type);
    // overrides Transform
    public override void LoadInnerXml(System.Xml.XmlNodeList nodeList);
    // overrides Transform
    public override void LoadInput(object obj);
    // overrides Transform
    // Protected Instance Methods
    protected override XmlNodeList GetInnerXml( );
    // overrides Transform
}
```

This implementation of the abstract `Transform` class removes the Signature element that is present when the XML Signature is within in the XML document that is being signed.

Hierarchy

System.Object Transform XmlDsigEnvelopedSignatureTransform

[\[Team LiB \]](#)

[\[Team LiB \]](#)

XmlDsigXPathTransform

System.Security.Cryptography.Xml
(system.security.dll)

class

```
public class XmlDsigXPathTransform : Transform {
    // Public Constructors
    public XmlDsigXPathTransform( );
    // Public Instance Properties
    public override Type[] InputTypes {get; }
    // overrides Transform
    public override Type[] OutputTypes {get; }
    // overrides Transform
    // Public Instance Methods
    public override object GetOutput( );
    // overrides Transform
    public override object GetOutput(Type type);
    // overrides Transform
    public override void LoadInnerXml(System.Xml.XmlNodeList nodeList);
    // overrides Transform
    public override void LoadInput(object obj);
    // overrides Transform
    // Protected Instance Methods
    protected override XmlNodeList GetInnerXml( );
    // overrides Transform
}
```

This implementation of the abstract `Transform` class applies an XPATH expression to the data to be signed; the expression is specified with the `LoadInput()` method.

Hierarchy

System.Object Transform XmlDsigXPathTransform

[\[Team LiB \]](#)

[\[Team LiB \]](#)

XmlDsigXsltTransform

System.Security.Cryptography.Xml
(system.security.dll)

class

```
public class XmlDsigXsltTransform : Transform {
    // Public Constructors
    public XmlDsigXsltTransform( );
    public XmlDsigXsltTransform(bool includeComments);
    // Public Instance Properties
    public override Type[] InputTypes {get; }
    // overrides Transform
    public override Type[] OutputTypes {get; }
    // overrides Transform
    // Public Instance Methods
    public override object GetOutput( );
    // overrides Transform
    public override object GetOutput(Type type);
    // overrides Transform
    public override void LoadInnerXml(System.Xml.XmlNodeList nodeList);
    // overrides Transform
    public override void LoadInput(object obj);
    // overrides Transform
    // Protected Instance Methods
    protected override XmlNodeList GetInnerXml( );
    // overrides Transform
}
```

This implementation of the abstract `Transform` class applies an XSLT expression to the data to be signed; the expression is specified with the `LoadInput()` method.

Hierarchy

System.Object Transform XmlDsigXsltTransform

[\[Team LiB \]](#)

[[Team LiB](#)]

Chapter 27. The System.Security.Permissions Namespace

The `System.Security.Permissions` namespace contains permission classes and their attribute counterparts. Permissions are the mechanism through which the .NET runtime enforces code-access security. Most commonly, code uses permissions to protect access to its important functionality and resources by making security demands. A security demand raises a `System.Security.SecurityException` if the grant set of the calling code (and all previous callers or the call stack) does not include a permission equivalent to the one demanded. Other security operations include link demands, inheritance demands, stack walk overrides, and permission requests.

There are two ways to express security statements in code: imperative syntax and declarative syntax. Imperative syntax uses the methods of permission objects whereas declarative syntax involves the application of permission attributes to code elements, such as assemblies, classes, and methods. Permission attributes result in the instantiation and use of permission objects, but this is handled by the runtime and is transparent to both the programmer and user.

All permission classes implement the `System.Security.IPermission` interface, which defines the basic functionality all permissions must implement to integrate with the runtime's code-access security mechanisms. However, there are three distinct categories of permissions defined in the `System.Security.Permissions` namespace: code-access permissions, identity permissions, and role-based permissions. The code-access permission classes represent actions defined in, and resources accessible through, the .NET Framework class library that are subject to code-access security control. The identity permission classes represent the value of *host* evidence that an assembly or application domain presents to the runtime; only evidence types that implement the `System.Security.Policy.IIdentityPermissionFactory` have corresponding identity permission classes. The only role-based permission (`PrincipalPermission`) represents the identity and roles of a user on whose behalf code is running.

All of the code-access and identity permission classes extend the abstract `System.Security.CodeAccessPermission` base class, which implements both the `System.Security.IPermission` and `System.Security.IStackWalk` interfaces. Executing an imperative security demand for any of these permission classes involves calling their `System.Security.IStackWalk.Demand()` method, which results in a stack walk. A stack walk uses the call stack to ensure that not only the immediate caller, but also all previous caller have the demanded permission. Both the `System.Security.IStackWalk` interface and the `System.Security.CodeAccessPermission` base class define methods that allow code to manipulate the stack walk process using imperative security syntax.

Each permission class has an attribute counterpart that derives from the abstract `CodeAccessSecurityAttribute` class. These attribute classes allow their corresponding permission to be applied to code using declarative security syntax. Declarative security syntax provides access to a broader range of security operations than imperative syntax, including link demands, inheritance demands, and permission requests; the values of the `SecurityAction` enumeration represent the

full range of declarative security statements. Because declarative security statements are stored as assembly metadata, the runtime and security administrators can evaluate the security requirements of an assembly without the need to load or execute the assembly. However, the content of declarative security statements are fixed at code time, and cannot be based on runtime state, or use input, unlike imperative statements. In addition, declarative security statements cannot use program constructs, such as loops and conditionals, to evaluate complex security requirements.

Figure 27-1 and Figure 27-1 show the types in this namespace.

Figure 27-1. CodeAccessPermissions from this namespace

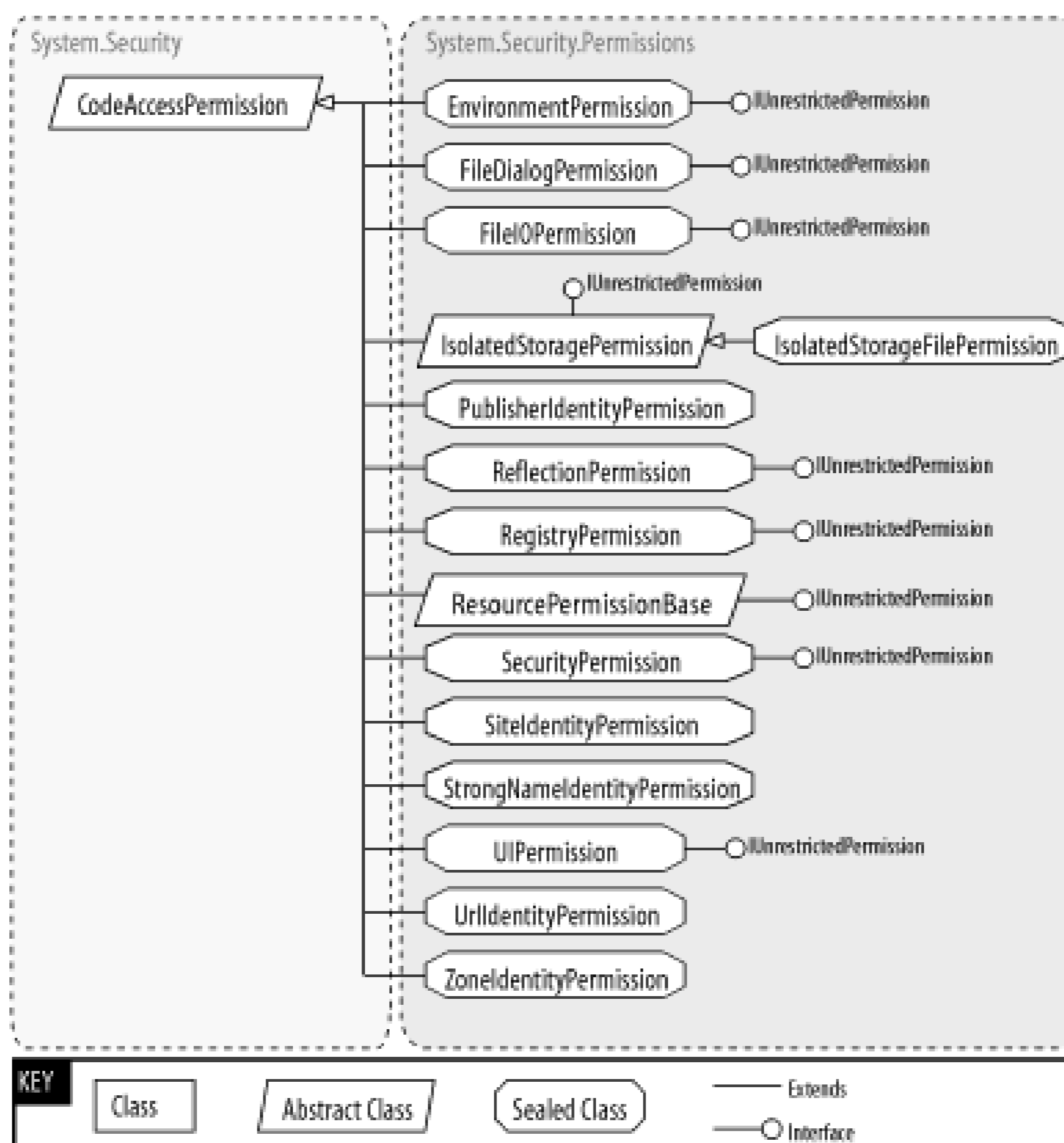
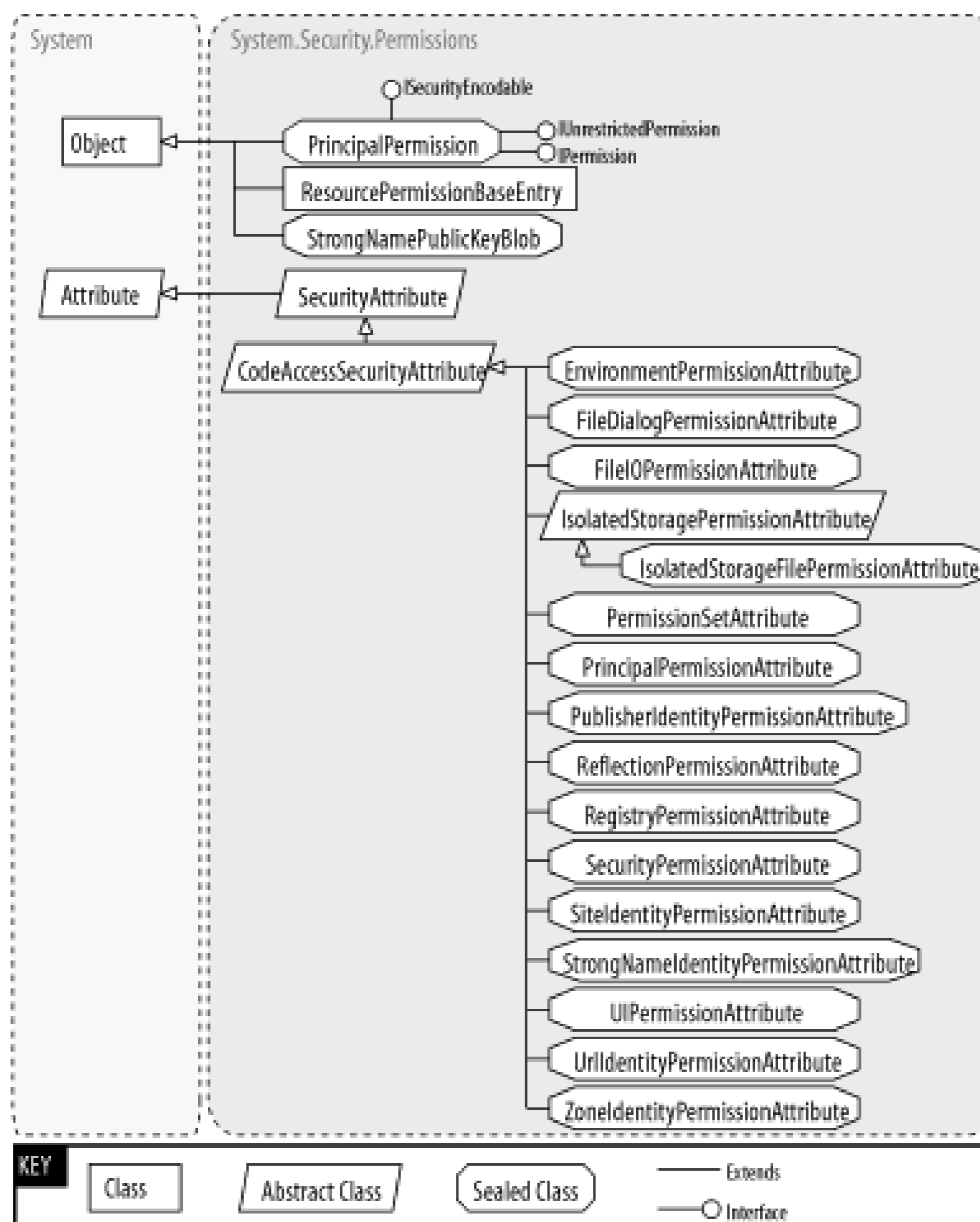


Figure 27-2. More types from this namespace



[Team LiB]

[\[Team LiB \]](#)

CodeAccessSecurityAttribute

ECMA 1.0,
serializableSystem.Security.Permissions
(mscorlib.dll)*abstract
class*

```
public abstract class CodeAccessSecurityAttribute : SecurityAttribute {  
    // Public Constructors  
    public CodeAccessSecurityAttribute(SecurityAction action);  
}
```

This abstract class, which extends the `SecurityAttribute` class, provides the base class from which all permission attributes derive. Permission attributes provide the mechanism through which to express declarative security statements for various program elements. By convention, all permission attributes that derive from `CodeAccessSecurityAttribute` implement a constructor that takes a value of the `SecurityAction` enumeration. The value of the `SecurityAction` argument specifies the declarative security statement represented by the permission attribute instance.

Hierarchy

```
System.Object      System.Attribute      SecurityAttribute  
CodeAccessSecurityAttribute
```

Subclasses

Multiple types

Valid On

Assembly, Class, Struct, Constructor, Method

[\[Team LiB \]](#)

[Team LiB]

EnvironmentPermission

ECMA 1.0,
serializableSystem.Security.Permissions
(mscorlib.dll)*sealed*
class

```

public sealed class EnvironmentPermission : System.Security.CodeAccessPermission : IUnre
    IBuiltInPermission {
// Public Constructors
    public EnvironmentPermission(EnvironmentPermissionAccess flag, string pathList);
    public EnvironmentPermission(PermissionState state);
// Public Instance Methods
    public void AddPathList(EnvironmentPermissionAccess flag, string pathList);
    public override IPermission Copy( );
// overrides CodeAccessPermission
    public override void FromXml(System.Security.SecurityElement esd);
// overrides CodeAccessPermission
    public string GetPathList(EnvironmentPermissionAccess flag);
    public override IPermission Intersect(System.Security.IPermission target);
// overrides CodeAccessPermission
    public override bool IsSubsetOf(System.Security.IPermission target);
// overrides CodeAccessPermission
    public bool IsUnrestricted( );
// implements IUnrestrictedPermission
    public void SetPathList(EnvironmentPermissionAccess flag, string pathList);
    public override SecurityElement ToXml( );
// overrides CodeAccessPermission
    public override IPermission Union(System.Security.IPermission other);
// overrides CodeAccessPermission
}

```

This code-access permission controls access to read and write environment variables, as well as the ability to create and delete environment variables. The members of the `System.Environment` class. To create an `EnvironmentPermission` object representing access to environment variables, pass a semicolon-separated list of environment variable names to the object constructor and a value from the `EnvironmentPermissionAccess` enumeration. The `EnvironmentPermissionAccess` enumeration values that represent the type of access granted to an environment variable, such as permission to read or write environment variables.

The `EnvironmentPermission` class maintains separate lists of environment variables with read access (`EnvironmentPermissionAccess.Read`), and those with write access (`EnvironmentPermissionAccess.Write`). The `EnvironmentPermission` class does not treat the `EnvironmentPermissionAccess.AllAccess` as a separate access level; it assigns independent read and write access to environment variables assigned this access level.

The lists of accessible environment variables are managed using the `SetPathList()` and `AddPathList()` methods. Both methods take values of the `EnvironmentPermissionAccess` enumeration and a semicolon-separated list of environment variable names as arguments. `SetPathList()` replaces the existing list of variables with the new set; its `EnvironmentPermissionAccess` argument

argument specifies whether to replace the read or write list. `AddPathList()` grants access to the specified path to the existing list; its `EnvironmentPermissionAccess` argument specifies which variable list to modify.

The `GetPathList()` method returns a semi-colon separated list of environment variables that has access to the specified path. `GetPathList()` accepts only the `EnvironmentPermissionAccess.Read` or `EnvironmentPermissionAccess.Write` values. If any other value is specified, it will throw a `System.ArgumentException`.

Hierarchy

```
System.Object → System.Security.CodeAccessPermission(System.Security.IPermission,
System.Security.ISecurityEncodable, System.Security.IStackWalk)
EnvironmentPermission(IUnrestrictedPermission, IBuiltInPermission)
```

[Team LiB]

[\[Team LiB \]](#)

EnvironmentPermissionAccess

ECMA 1.0,
serializable,
flagSystem.Security.Permissions
(mscorlib.dll)*enum*

```
public enum EnvironmentPermissionAccess {  
    NoAccess = 0x00000000, Read = 0x00000001, Write = 0x00000002, AllAccess = 0x00000003  
}
```

Values of the `EnvironmentPermissionAccess` enumeration are used with the `EnvironmentPermission` class to specify the type of access granted to a set of environment variables by an `EnvironmentPermission` object. `Read` represents access to read an environment variable, and `Write` represents access to create, delete, and change an environment variable; `Write` does not encompass the permission granted by `Read`. `AllAccess` represents the combined permissions of both `Read` and `Write`.

Hierarchy

```
System.Object      System.ValueType      System.Enum(System.IComparable,  
System.IFormattable, System.IConvertible)      EnvironmentPermissionAccess
```

Passed To

```
EnvironmentPermission.{AddPathList( ), EnvironmentPermission( ), GetPathList( ),  
SetPathList( )}
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

EnvironmentPermissionAttribute ECMA 1.0, serializable

System.Security.Permissions
(mscorlib.dll) *sealed
class*

```
public sealed class EnvironmentPermissionAttribute : CodeAccessSecurityAttribute {
// Public Constructors
    public EnvironmentPermissionAttribute(SecurityAction action);
// Public Instance Properties
    public string All{set; get; }
    public string Read{set; get; }
    public string Write{set; get; }
// Public Instance Methods
    public override IPermission CreatePermission( );
// overrides SecurityAttribute
}
```

This attribute class provides declarative syntax support for `EnvironmentPermission`. The `All`, `Read`, and `Write` properties take semicolon-separated lists of environment variable names to which the `EnvironmentPermissionAttribute` represents access. `Read` specifies the environment variables to which read access is granted, and `Write` specifies the environment variables to which write access is granted. The `All` property contains the variables to which both read and write access is granted. Setting `Unrestricted` to true represents full access to all environment variables.

Hierarchy

```
System.Object      System.Attribute      SecurityAttribute
CodeAccessSecurityAttribute  EnvironmentPermissionAttribute
```

Valid On

Assembly, Class, Struct, Constructor, Method

[\[Team LiB \]](#)

[Team LiB]

FileDialogPermission

serializable

System.Security.Permissions
(mscorlib.dll)*sealed
class*

```

public sealed class FileDialogPermission : System.Security.CodeAccessPermission : IUnres
    IBuiltInPermission {
// Public Constructors
    public FileDialogPermission(FileDialogPermissionAccess access);
    public FileDialogPermission(PermissionState state);
// Public Instance Properties
    public FileDialogPermissionAccess Access{set; get; }
// Public Instance Methods
    public override IPermission Copy( );
// overrides CodeAccessPermission
    public override void FromXml(System.Security.SecurityElement esd);
// overrides CodeAccessPermission
    public override IPermission Intersect(System.Security.IPermission target);
// overrides CodeAccessPermission
    public override bool IsSubsetOf(System.Security.IPermission target);
// overrides CodeAccessPermission
    public bool IsUnrestricted( );
// implements IUnrestrictedPermission
    public override string ToString( );
// overrides CodeAccessPermission
    public override SecurityElement ToXml( );
// overrides CodeAccessPermission
    public override IPermission Union(System.Security.IPermission target);
// overrides CodeAccessPermission
}

```

This code-access permission controls the ability for code to allow users to access files and folders through `System.Windows.Forms.OpenFileDialog` and the `System.Windows.Forms.SaveFileDialog` classes. This is useful for providing restricted access to files and folders, without the need to grant permissions with `FileDialogPermission` type of file dialog to which access is granted by a `FileDialogPermission` object is specified at construction the `FileDialogPermissionAccess` enumeration. To retrieve or modify the file dialog access represented by a `FileDialogPermission` object, use its `Access` property.

Hierarchy

```

System.Object      System.Security.CodeAccessPermission(System.Security.IPermission,
System.Security.ISecurityEncodable, System.Security.IStackWalk)
FileDialogPermission(IUnrestrictedPermission, IBuiltInPermission)

```

[Team LiB]

[\[Team LiB \]](#)

FileDialogPermissionAccess serializable, flag

System.Security.Permissions
(mscorlib.dll) *enum*

```
public enum FileDialogPermissionAccess {  
    None = 0x00000000, Open = 0x00000001, Save = 0x00000002, OpenSave = 0x00000003  
}
```

This enumeration is used in conjunction with the `FileDialogPermission` class to specify the type of file dialog access a `FileDialogPermission` object represents. `Open` represents the ability to call the `System.Windows.Forms.OpenFileDialog.OpenFile()` method, and `Save` represents the ability to call the `System.Windows.Forms.SaveFileDialog.OpenFile()` method. The `OpenSave` value represents the combined permissions of both the `Open` and `Save` values; `None` represents no access.

Hierarchy

```
System.Object      System.ValueType      System.Enum(System.IComparable,  
System.IFormattable, System.IConvertible)      FileDialogPermissionAccess
```

Returned By

```
FileDialogPermission.Access
```

Passed To

```
FileDialogPermission.{Access, FileDialogPermission( )}
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

FileDialogPermissionAttribute serializable

System.Security.Permissions
(mscorlib.dll) *sealed class*

```
public sealed class FileDialogPermissionAttribute : CodeAccessSecurityAttribute {
// Public Constructors
    public FileDialogPermissionAttribute(SecurityAction action);
// Public Instance Properties
    public bool Open{set; get; }
    public bool Save{set; get; }
// Public Instance Methods
    public override IPermission CreatePermission( );
// overrides SecurityAttribute
}
```

This permission attribute provides declarative syntax support for `FileDialogPermission`. The `Open` and `Save` properties take Boolean values that identify the type of access granted to the file dialog UI components. Setting `Open` to true represents permission to execute the `System.Windows.Forms.OpenFileDialog.OpenFile()` method, and setting `Save` to true represents permission to execute the `System.Windows.Forms.SaveFileDialog.OpenFile()` method. Setting `Unrestricted` to true represents the ability to call both methods.

Hierarchy

```
System.Object      System.Attribute      SecurityAttribute
CodeAccessSecurityAttribute      FileDialogPermissionAttribute
```

Valid On

Assembly, Class, Struct, Constructor, Method

[\[Team LiB \]](#)

[Team LiB]

FileIOPermission

ECMA 1.0, serializable

System.Security.Permissions
(mscorlib.dll)*sealed*
class

```

public sealed class FileIOPermission : System.Security.CodeAccessPermission : IUnrestricted
    IBuiltInPermission {
// Public Constructors
    public FileIOPermission(FileIOPermissionAccess access, string path);
    public FileIOPermission(FileIOPermissionAccess access, string[] pathList);
    public FileIOPermission(PermissionState state);
// Public Instance Properties
    public FileIOPermissionAccess AllFiles {set; get; }
    public FileIOPermissionAccess AllLocalFiles {set; get; }
// Public Instance Methods
    public void AddPathList(FileIOPermissionAccess access, string path);
    public void AddPathList(FileIOPermissionAccess access, string[] pathList);
    public override IPermission Copy( );
// overrides CodeAccessPermission
    public override void FromXml(System.Security.SecurityElement esd);
// overrides CodeAccessPermission
    public string[] GetPathList(FileIOPermissionAccess access);
    public override IPermission Intersect(System.Security.IPermission target);
// overrides CodeAccessPermission
    public override bool IsSubsetOf(System.Security.IPermission target);
// overrides CodeAccessPermission
    public bool IsUnrestricted( );
// implements IUnrestrictedPermission
    public void SetPathList(FileIOPermissionAccess access, string path);
    public void SetPathList(FileIOPermissionAccess access, string[] pathList);
    public override SecurityElement ToXml( );
// overrides CodeAccessPermission
    public override IPermission Union(System.Security.IPermission other);
// overrides CodeAccessPermission
}

```

The `FileIOPermission` class represents the ability for code to access files and folders. The `FileIOPermission` maintains separate lists of files and folders to which different levels of access have been granted, and the `FileIOPermission` object can represent different types of access to an arbitrary number of files and folders. Simple configurations are possible using the `FileIOPermission` constructors; to create more complex `FileIOPermission` configurations requires use of the `AddPathList()` and `SetPathList()` methods.

To create a `FileIOPermission` object representing access to a single file or folder, pass a `System.String` file or folder's full path to the `FileIOPermission` constructor along with a value from the `FileIOPermissionAccess` enumeration. The `FileIOPermissionAccess` enumeration contains values that represent the type of access.

file or folder, such as permission to read, write, or append. A second constructor takes a `System.String` the names of multiple files and folders, but takes only a single `FileIOPermissionAccess` value, applying level to all of the files and folders.

The lists of files and folders to which each level of access is granted are managed using the `SetPathList` and `AddPathList()` methods. Both methods take either a single or an array of `System.String` values specifying names of the files and folders to grant the specified level of access. The `AddPathList()` method adds a specified type for the set of files and folders to its current list. The `SetPathList()` method replaces its contents and folders of the specified access level. The `AllFiles` property provides a simple mechanism to specify all files and folders regardless of location, and the `AllLocalFiles` property does the same, but only for files on the local hard drive.

The `GetPathList()` method returns a `System.String` array containing the full path name of files and folders for which access of the specified level is granted. The `FileIOPermissionAccess.AllAccess` and `FileIOPermissionAccess.NoAccess` values are invalid in calls to `GetPathList()`, and will throw a `System.ArgumentException`.

For performance reasons, the .NET file and folder classes check file access permissions only when they create files and so care must be taken when passing file and folder references.

Hierarchy

```
System.Object      System.Security.CodeAccessPermission(System.Security.IPermission,
System.Security.ISecurityEncodable , System.Security.IStackWalk)
FileIOPermission(IUnrestrictedPermission, IBuiltInPermission)
```

[Team LiB]

[\[Team LiB \]](#)

FileIOPermissionAccess

ECMA 1.0,
serializable, flag

System.Security.Permissions (mscorlib.dll)

enum

```
public enum FileIOPermissionAccess {
    NoAccess = 0x00000000, Read = 0x00000001, Write = 0x00000002, Append = 0x00000004,
    PathDiscovery = 0x00000008, AllAccess = 0x0000000F
}
```

The `FileIOPermissionAccess` enumeration contains values that are used with the `FileIOPermission` class to specify the type of access represented by a `FileIOPermission` object. `Read` represents permission to read from a file or folder, and `Write` represents permission to write to, delete, or overwrite files and folders; `Write` does not encompass the ability to `Read`. `Append` represents the ability to append material to a file or folder, as well as to create new folders. The `PathDiscovery` value represents the ability to access information from a file or folder's path, which could provide username and folder structure information that would be useful to malicious code. `AllAccess` includes the combined permissions of the `Read`, `Write`, `Append`, and `PathDiscovery` values.

Hierarchy

```
System.Object      System.ValueType      System.Enum(System.IComparable,
System.IFormattable, System.IConvertible)      FileIOPermissionAccess
```

Returned By

```
FileIOPermission.{AllFiles, AllLocalFiles}
```

Passed To

```
FileIOPermission.{AddPathList( ), AllFiles, AllLocalFiles, FileIOPermission( ),
GetPathList( ), SetPathList( )}, System.Security.Policy.FileCodeGroup.FileCodeGroup( )
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

FileIOPermissionAttribute

ECMA 1.0,
serializableSystem.Security.Permissions
(mscorlib.dll)*sealed*
class

```

public sealed class FileIOPermissionAttribute : CodeAccessSecurityAttribute {
// Public Constructors
    public FileIOPermissionAttribute(SecurityAction action);
// Public Instance Properties
    public string All{set; get; }
    public string Append{set; get; }
    public string PathDiscovery{set; get; }
    public string Read{set; get; }
    public string Write{set; get; }
// Public Instance Methods
    public override IPermission CreatePermission( );
// overrides SecurityAttribute
}

```

`FileIOPermissionAttribute` provides declarative syntax support for the `FileIOPermission` class. The `All`, `Append`, `PathDiscovery`, `Read`, and `Write` properties take the full pathname of the file or folder to which access of the appropriate level is granted. Each property represents access as defined by the `FileIOPermissionAccess` enumeration member of the same name. Unlike `FileIOPermission`, a `FileIOPermissionAttribute` can only represent access of each type to a single file or folder; apply multiple attributes to a program element in order to specify access to more than one. Setting `Unrestricted` to true represents full access to all files and folders.

Hierarchy

```

System.Object      System.Attribute      SecurityAttribute
CodeAccessSecurityAttribute  FileIOPermissionAttribute

```

Valid On

Assembly, Class, Struct, Constructor, Method

[\[Team LiB \]](#)

[\[Team LiB \]](#)

IsolatedStorageContainment serializable

System.Security.Permissions
(mscorlib.dll) *enum*

```
public enum IsolatedStorageContainment {
    None = 0, DomainIsolationByUser = 16, AssemblyIsolationByUser = 32,
    DomainIsolationByRoamingUser = 80, AssemblyIsolationByRoamingUser = 96,
    AdministerIsolatedStorageByUser = 112, UnrestrictedIsolatedStorage = 240
}
```

This enumeration contains values used with the `IsolatedStorageFilePermission` and `IsolatedStorageFilePermissionAttribute` classes to represent the level of access granted to isolated storage. `AssemblyIsolationByUser` represents the ability to obtain a store isolated by assembly and user, while `DomainIsolationByUser` represents the ability to obtain a store isolated by assembly, user, and application domain. `AssemblyIsolationByRoamingUser` and `DomainIsolationByRoamingUser` represent access to equivalent levels of isolation, but include the ability to obtain roaming stores. `AdministerIsolatedStorageByUser` represents administrative access to all stores for the current user, and `UnrestrictedIsolatedStorage` provides full and unrestricted access to all of the current user's stores.

Hierarchy

```
System.Object      System.ValueType      System.Enum(System.IComparable,
System.IFormattable, System.IConvertible)      IsolatedStorageContainment
```

Returned By

```
IsolatedStoragePermission.UsageAllowed,
IsolatedStoragePermissionAttribute.UsageAllowed
```

Passed To

```
IsolatedStoragePermission.UsageAllowed,
IsolatedStoragePermissionAttribute.UsageAllowed
```

[\[Team LiB \]](#)

[Team LiB]

IsolatedStorageFilePermission serializable

System.Security.Permissions *sealed*
(mscorlib.dll) *class*

```
public sealed class IsolatedStorageFilePermission : IsolatedStoragePermission : IBuiltInPermission
// Public Constructors
    public IsolatedStorageFilePermission(PermissionState state);
// Public Instance Methods
    public override IPermission Copy( );
// overrides CodeAccessPermission
    public override IPermission Intersect(System.Security.IPermission target);
// overrides CodeAccessPermission
    public override bool IsSubsetOf(System.Security.IPermission target);
// overrides CodeAccessPermission
    public override IPermission Union(System.Security.IPermission target);
// overrides CodeAccessPermission
}
```

The `IsolatedStorageFilePermission` code-access permission extends the `IsolatedStoragePermission` access to the methods of the `System.IO.IsolatedStorage.IsolatedStorageFile` class.

Hierarchy

```
System.Object      System.Security.CodeAccessPermission(System.Security.IPermission,
System.Security.ISecurityEncodable , System.Security.IStackWalk)
IsolatedStoragePermission(IUnrestrictedPermission)
IsolatedStorageFilePermission( IBuiltInPermission)
```

[Team LiB]

[Team LiB]

IsolatedStorageFilePermissionAttribute serializable

System.Security.Permissions
(mscorlib.dll)

sealed
class

```
public sealed class IsolatedStorageFilePermissionAttribute : IsolatedStoragePermissionA
// Public Constructors
    public IsolatedStorageFilePermissionAttribute(SecurityAction action);
// Public Instance Methods
    public override IPermission CreatePermission( );
// overrides SecurityAttribute
}
```

This attribute extends the `IsolatedStoragePermissionAttribute` and provides declarative security support for `IsolatedStorageFilePermission` code-access permission.

Hierarchy

```
System.Object      System.Attribute      SecurityAttribute      CodeAccessSecurityAttribute
IsolatedStoragePermissionAttribute      IsolatedStorageFilePermissionAttribute
```

Valid On

Assembly, Class, Struct, Constructor, Method

[Team LiB]

[Team LiB]

IsolatedStoragePermission

serializable

System.Security.Permissions
(mscorlib.dll)

*abstract
class*

```

public abstract class IsolatedStoragePermission : System.Security.CodeAccessPermission
// Public Constructors
    public IsolatedStoragePermission(PermissionState state);
// Public Instance Properties
    public IsolatedStorageContainment UsageAllowed{set; get; }
    public long UserQuota{set; get; }
// Public Instance Methods
    public override void FromXml(System.Security.SecurityElement esd);
// overrides CodeAccessPermission
    public bool IsUnrestricted( );
// implements IUnrestrictedPermission
    public override SecurityElement ToXml( );
// overrides CodeAccessPermission
}

```

This abstract class provides the base class from which all types of isolated storage permission classes derive. The unique feature of the `IsolatedStoragePermission` class, which all derived classes inherit, are the `UsageAllowed` and `UserQuota` properties. The `UsageAllowed` property represents the scope of store code can obtain, and `UserQuota` specifies the store

Hierarchy

```

System.Object      System.Security.CodeAccessPermission(System.Security.IPermission,
System.Security.ISecurityEncodable , System.Security.IStackWalk)
IsolatedStoragePermission(IUnrestrictedPermission)

```

Subclasses

```

IsolatedStorageFilePermission

```

Returned By

```

System.IO.IsolatedStorage.IsolatedStorage.GetPermission( )

```

[Team LiB]

[Team LiB]

IsolatedStoragePermissionAttribute serializable

System.Security.Permissions
(mscorlib.dll) *abstract class*

```
public abstract class IsolatedStoragePermissionAttribute : CodeAccessSecurityAttribute
// Public Constructors
    public IsolatedStoragePermissionAttribute(SecurityAction action);
// Public Instance Properties
    public IsolatedStorageContainment UsageAllowed{set; get; }
    public long UserQuota{set; get; }
}
```

`IsolatedStoragePermissionAttribute` is the abstract base class from which all types of isolated storage permission attribute classes derive. It provides declarative support for the abstract `IsolatedStoragePermission` class. Currently, the only isolated storage permission attribute implementation is the `IsolatedStorageFilePermissionAttribute`. The `UsageAllowed` property accepts a member of the `IsolatedStorageContainment` enumeration and represents the permitted level of isolation. The `UserQuota` property configures the maximum size that a store can attain.

Hierarchy

```
System.Object      System.Attribute      SecurityAttribute      CodeAccessSecurityAttribute
    IsolatedStoragePermissionAttribute
```

Subclasses

```
IsolatedStorageFilePermissionAttribute
```

Valid On

Assembly, Class, Struct, Constructor, Method

[Team LiB]

[\[Team LiB \]](#)

I UnrestrictedPermission

System.Security.Permissions
(mscorlib.dll)

interface

```
public interface IUnrestrictedPermission {  
    // Public Instance Methods  
    public bool IsUnrestricted( );  
}
```

Code-access permission classes that implement the `IUnrestrictedPermission` interface can represent an unrestricted or completely restricted state. These two states represent the extremes of a permission class's permission range. For example, the `FileIOPermission` class can represent complete access to all files and folders, or no file and folder access.

By convention, a permission class that implements the `IUnrestrictedPermission` interface must declare a constructor that takes a `PermissionState` argument. An unrestricted permission is created by passing the value `PermissionState.Unrestricted` to the permission constructor, while a completely restricted permission is created using the value `PermissionState.None`. The `IsUnrestricted()` method indicates whether a permission object represents its unrestricted state.

Implemented By

Multiple types

[\[Team LiB \]](#)

[\[Team LiB \]](#)

PermissionSetAttribute

serializable

System.Security.Permissions
(mscorlib.dll)*sealed
class*

```

public sealed class PermissionSetAttribute : CodeAccessSecurityAttribute {
// Public Constructors
    public PermissionSetAttribute(SecurityAction action);
// Public Instance Properties
    public string File{set; get; }
    public string Name{set; get; }
    public bool UnicodeEncoded{set; get; }
    public string XML{set; get; }
// Public Instance Methods
    public override IPermission CreatePermission( );
// overrides SecurityAttribute
    public PermissionSet CreatePermissionSet( );
}

```

The `PermissionSetAttribute` class supports the use of permission sets in declarative security statements. To specify the contents of the permission set, use one of mutually exclusive `File`, `Name`, or `XML` properties. `File` takes the name of a file that contains the permission set's XML definition. `Name` specifies one of the standard immutable named permission sets defined by the .NET Framework (`FullTrust`, `SkipVerification`, `Execution`, `Nothing`, `LocalIntranet`, or `Internet`). `XML` takes a `System.String` containing the XML definition of the permission set. To create a permission set that represents unrestricted access, set the `Unrestricted` property to true.

When required, the runtime calls the `CreatePermissionSet()` method to create a correctly configured `System.Security.PermissionSet` from the `PermissionSetAttribute`.

Hierarchy

```

System.Object      System.Attribute      SecurityAttribute
CodeAccessSecurityAttribute  PermissionSetAttribute

```

Valid On

Assembly, Class, Struct, Constructor, Method

[\[Team LiB \]](#)

[\[Team LiB \]](#)

PermissionState

ECMA 1.0, serializable

System.Security.Permissions
(mscorlib.dll)*enum*

```
public enum PermissionState {  
    None = 0, Unrestricted = 1  
}
```

This enumeration is used when instantiating code-access permission classes that implement the `IUnrestrictedPermission` interface. A code-access permission object created by passing the `Unrestricted` value to its constructor represents unrestricted access to the resources and actions controlled by the permission. A permission object created using the `None` value represents no access.

Hierarchy

```
System.Object    System.ValueType    System.Enum(System.IComparable,  
System.IFormattable, System.IConvertible)    PermissionState
```

Passed To

Multiple types

[\[Team LiB \]](#)

[Team LiB]

PrincipalPermission

serializable

System.Security.Permissions
(mscorlib.dll)*sealed
class*

```

public sealed class PrincipalPermission : System.Security.IPermission,
    System.Security.ISecurityEncodable, IUnrestrictedPermission, IBuiltInPermission {
// Public Constructors
    public PrincipalPermission(PermissionState state);
    public PrincipalPermission(string name, string role);
    public PrincipalPermission(string name, string role, bool isAuthenticated);
// Public Instance Methods
    public IPermission Copy( );
// implements System.Security.IPermission
    public void Demand( );
// implements System.Security.IPermission
    public void FromXml(System.Security.SecurityElement elem);
// implements System.Security.ISecurityEncodable
    public IPermission Intersect(System.Security.IPermission target);
// implements System.Security.IPermission
    public bool IsSubsetOf(System.Security.IPermission target);
// implements System.Security.IPermission
    public bool IsUnrestricted( );
// implements IUnrestrictedPermission
    public override string ToString( );
// overrides object
    public SecurityElement ToXml( );
// implements System.Security.ISecurityEncodable
    public IPermission Union(System.Security.IPermission other);
// implements System.Security.IPermission
}

```

The `PrincipalPermission` class is the only role-based permission provided in the .NET Framework class library. `PrincipalPermission` allows code to make imperative security demands to ensure that the current thread's `System.Security.Principal.IPrincipal` object represents a specific identity or contains specified role names. Unlike the code-access and identity permission classes, `PrincipalPermission` does not inherit from `System.Security.CodeAccessPermission`. In addition, though `PrincipalPermission` implements `System.Security.IPermission`, it does not implement `System.Security.IStackWalk`, which means that security demands against `PrincipalPermission` objects do not result in stack walks.

A `PrincipalPermission` object must be configured during construction, as no properties are provided to manage its state. The constructor `name` argument specifies the identity name, and the `role` argument specifies the name of a single role of which the identity must be a member. The `isAuthenticated` argument specifies whether the principal must have been authenticated.

[Team LiB]

[\[Team LiB \]](#)

PrincipalPermissionAttribute serializable

System.Security.Permissions *sealed*
 (mscorlib.dll) *class*

```
public sealed class PrincipalPermissionAttribute : CodeAccessSecurityAttribute {
// Public Constructors
    public PrincipalPermissionAttribute(SecurityAction action);
// Public Instance Properties
    public bool Authenticated{set; get; }
    public string Name{set; get; }
    public string Role{set; get; }
// Public Instance Methods
    public override IPermission CreatePermission( );
// overrides SecurityAttribute
}
```

The `PrincipalPermissionAttribute` class provides limited declarative syntax support for the `PrincipalPermission` class. The `PrincipalPermissionAttribute` class only supports declarative security demands and does not allow code to make declarative stack walk overrides or permission requests. The `Name`, `Role`, and `Authenticated` properties provide the mechanism through which to specify the conditions that the current thread's principal must meet to pass a security demand.

Hierarchy

```
System.Object      System.Attribute      SecurityAttribute
CodeAccessSecurityAttribute  PrincipalPermissionAttribute
```

Valid On

Class, Method

[\[Team LiB \]](#)

[Team LiB]

PublisherIdentityPermission serializable

System.Security.Permissions *sealed*
(mscorlib.dll) *class*

```
public sealed class PublisherIdentityPermission : System.Security.CodeAccessPermission
// Public Constructors
    public PublisherIdentityPermission(PermissionState state);
    public PublisherIdentityPermission(System.Security.Cryptography.X509Certificates.X509Certificate certificate);
// Public Instance Properties
    public X509Certificate Certificate{set; get; }
// Public Instance Methods
    public override IPermission Copy( );
// overrides CodeAccessPermission
    public override void FromXml(System.Security.SecurityElement esd);
// overrides CodeAccessPermission
    public override IPermission Intersect(System.Security.IPermission target);
// overrides CodeAccessPermission
    public override bool IsSubsetOf(System.Security.IPermission target);
// overrides CodeAccessPermission
    public override SecurityElement ToXml( );
// overrides CodeAccessPermission
    public override IPermission Union(System.Security.IPermission target);
// overrides CodeAccessPermission
}
```

The `PublisherIdentityPermission` class represents the `System.Security.Policy.Publisher` evidence application domain presented to the runtime. The `PublisherIdentityPermission` class enables code to make statements based on the publisher certificate used to sign an assembly. For example, code can use the `PublisherIdentityPermission` class to ensure that only code signed by their company or a trusted third party can call the methods of the assembly.

The `PublisherIdentityPermission` class provides a constructor that takes a `System.Security.Cryptography.X509Certificates.X509Certificate` object containing an X509 certificate of the publisher that signed an assembly. The `X509Certificate` object can be manipulated after construction through its `Certificate` property.

Hierarchy

```
System.Object      System.Security.CodeAccessPermission(System.Security.IPermission,
System.Security.ISecurityEncodable, System.Security.IStackWalk)
PublisherIdentityPermission(IBuiltInPermission)
```

[Team LiB]

[Team LiB]

PublisherIdentityPermissionAttribute serializable

System.Security.Permissions
(mscorlib.dll)

sealed
class

```
public sealed class PublisherIdentityPermissionAttribute : CodeAccessSecurityAttribute
// Public Constructors
    public PublisherIdentityPermissionAttribute(SecurityAction action);
// Public Instance Properties
    public string CertFile{set; get; }
    public string SignedFile{set; get; }
    public string X509Certificate{set; get; }
// Public Instance Methods
    public override IPermission CreatePermission( );
// overrides SecurityAttribute
}
```

`PublisherIdentityPermissionAttribute` provides declarative syntax support for the `PublisherIdentityPermission` class. The publisher certificate that represents the publisher of an assembly can be specified using three different properties. The `CertFile` property takes the name of a certificate file containing the publisher's X.509 certificate. The `SignedFile` property takes the name of a file that is already signed with the publisher's certificate (which will be extracted). The `X509Certificate` property takes a `System.String` containing a hexadecimal representation of the publisher's certificate.

Hierarchy

```
System.Object      System.Attribute      SecurityAttribute      CodeAccessSecurityAttribute
    PublisherIdentityPermissionAttribute
```

Valid On

Assembly, Class, Struct, Constructor, Method

[Team LiB]

[Team LiB]

ReflectionPermission ECMA 1.0, serializable

System.Security.Permissions *sealed*
 (mscorlib.dll) *class*

```
public sealed class ReflectionPermission : System.Security.CodeAccessPermission : IUnres
    IBuiltInPermission {
// Public Constructors
    public ReflectionPermission(PermissionState state);
    public ReflectionPermission(ReflectionPermissionFlag flag);
// Public Instance Properties
    public ReflectionPermissionFlag Flags{set; get; }
// Public Instance Methods
    public override IPermission Copy( );
// overrides CodeAccessPermission
    public override void FromXml(System.Security.SecurityElement esd);
// overrides CodeAccessPermission
    public override IPermission Intersect(System.Security.IPermission target);
// overrides CodeAccessPermission
    public override bool IsSubsetOf(System.Security.IPermission target);
// overrides CodeAccessPermission
    public bool IsUnrestricted( );
// implements IUnrestrictedPermission
    public override SecurityElement ToXml( );
// overrides CodeAccessPermission
    public override IPermission Union(System.Security.IPermission other);
// overrides CodeAccessPermission
}
```

The `ReflectionPermission` code-access permission class controls the ability for code to access nonvisib through .NET's reflection capabilities. This includes the ability of code to access protected and private ty to create assemblies dynamically in memory. The functionality represented by a `ReflectionPermission` construction using members of the `ReflectionPermissionFlag` enumeration. To retrieve or modify the `ReflectionPermission` object, use its `Flags` property.

Hierarchy

```
System.Object      System.Security.CodeAccessPermission(System.Security.IPermission,
System.Security.ISecurityEncodable , System.Security.IStackWalk)
ReflectionPermission(IUnrestrictedPermission, IBuiltInPermission)
```

[Team LiB]

[\[Team LiB \]](#)

ReflectionPermissionAttribute

ECMA 1.0,
serializableSystem.Security.Permissions
(mscorlib.dll)*sealed
class*

```
public sealed class ReflectionPermissionAttribute : CodeAccessSecurityAttribute {
// Public Constructors
    public ReflectionPermissionAttribute(SecurityAction action);
// Public Instance Properties
    public ReflectionPermissionFlag Flags{set; get; }
    public bool MemberAccess{set; get; }
    public bool ReflectionEmit{set; get; }
    public bool TypeInformation{set; get; }
// Public Instance Methods
    public override IPermission CreatePermission( );
// overrides SecurityAttribute
}
```

ReflectionPermissionAttribute provides declarative syntax support for the ReflectionPermission class. To configure the level of reflection functionality access represented by a ReflectionPermissionAttribute, assign values of the ReflectionPermissionFlag enumeration to the Flags property. Alternatively, the MemberAccess, ReflectionEmit, and TypeInformation properties take Boolean values to configure the type of access represented by the ReflectionPermissionAttribute; these properties map to the member values defined in the ReflectionPermissionFlag enumeration. Setting Unrestricted to true creates a ReflectionPermissionAttribute that represents unrestricted access to reflection functionality and is equivalent to the value ReflectionPermissionFlag.AllFlags.

Hierarchy

```
System.Object      System.Attribute      SecurityAttribute
CodeAccessSecurityAttribute  ReflectionPermissionAttribute
```

Valid On

Assembly, Class, Struct, Constructor, Method

[\[Team LiB \]](#)

[Team LiB]

ReflectionPermissionFlag

ECMA 1.0,
serializable, flagSystem.Security.Permissions
(mscorlib.dll)*enum*

```
public enum ReflectionPermissionFlag {
    NoFlags = 0x00000000, TypeInformation = 0x00000001, MemberAccess = 0x00000002, Reflecti
    AllFlags = 0x00000007
}
```

The values of the `ReflectionPermissionFlag` enumeration are used to specify the level of access to .NET capabilities that a `ReflectionPermission` object represents. `NoFlags` represents default reflection access to public members and type information. The `MemberAccess` and `TypeInformation` values represent reflection on nonpublic members and types, respectively. `ReflectionEmit` allows code to use the classes in the `System.Reflection.Emit` namespace to create assemblies in memory. The `AllFlags` value represents all the above values. Any level of access can be represented using logical OR operations to combine the member values.

Hierarchy

```
System.Object      System.ValueType      System.Enum(System.IComparable, System.IFormattable)
System.IConvertible      ReflectionPermissionFlag
```

Returned By

```
ReflectionPermission.Flags, ReflectionPermissionAttribute.Flags
```

Passed To

```
ReflectionPermission.{Flags, ReflectionPermission( )}, ReflectionPermissionAttribute.Flags
```

[Team LiB]

[Team LiB]

RegistryPermission

serializable

System.Security.Permissions
(mscorlib.dll)*sealed
class*

```

public sealed class RegistryPermission : System.Security.CodeAccessPermission : IUnrestrictedPermission
{
    // Public Constructors
    public RegistryPermission(PermissionState state);
    public RegistryPermission(RegistryPermissionAccess access, string pathList);
    // Public Instance Methods
    public void AddPathList(RegistryPermissionAccess access, string pathList);
    public override IPermission Copy( );
    // overrides CodeAccessPermission
    public override void FromXml(System.Security.SecurityElement esd);
    // overrides CodeAccessPermission
    public string GetPathList(RegistryPermissionAccess access);
    public override IPermission Intersect(System.Security.IPermission target);
    // overrides CodeAccessPermission
    public override bool IsSubsetOf(System.Security.IPermission target);
    // overrides CodeAccessPermission
    public bool IsUnrestricted( );
    // implements IUnrestrictedPermission
    public void SetPathList(RegistryPermissionAccess access, string pathList);
    public override SecurityElement ToXml( );
    // overrides CodeAccessPermission
    public override IPermission Union(System.Security.IPermission other);
    // overrides CodeAccessPermission
}

```

The `RegistryPermission` class represents the ability of code to access and manipulate the contents of the registry. More specifically, `RegistryPermission` controls access to secured members of the `Microsoft.Win32.RegistryPermission`. `RegistryPermission` maintains separate lists of registry key names to which different levels of access have been granted, and therefore a single `RegistryPermission` object can represent different types of access to an arbitrary set of registry keys.

To create a `RegistryPermission` object, pass a `System.String` containing a semicolon-separated list of registry key names along with a value from the `RegistryPermissionAccess` enumeration to the `RegistryPermission` constructor. The `RegistryPermissionAccess` enumeration contains values that represent the type of access granted to a registry key: `RegistryPermissionAccess.Read` (permission to read), `RegistryPermissionAccess.Write` (permission to write), or `RegistryPermissionAccess.Create` (permission to create).

To create a `RegistryPermission` object that represents a more complex permission set, use the `AddPathList` and `SetPathList` methods. Both methods take a `System.String` containing a semicolon-separated list of registry key names and a value from the `RegistryPermissionAccess` enumeration, which specifies the level of access to be granted to the registry keys. The `AddPathList` method adds the newly specified key names to the list of already granted keys, while the `SetPathList` method replaces its existing list of keys with the newly specified set.

The `GetPathList()` method returns a `System.String` containing a semicolon-separated list of registry of the specified level is granted. The `RegistryPermissionAccess.AllAccess` and `RegistryPermission` values are invalid in calls to `GetPathList()`, and will cause a `System.ArgumentException` to be throw

Hierarchy

```
System.Object → System.Security.CodeAccessPermission(System.Security.IPermission,  
System.Security.ISecurityEncodable , System.Security.IStackWalk)  
RegistryPermission(IUnrestrictedPermission, IBuiltInPermission)
```

[Team LiB]

[\[Team LiB \]](#)

RegistryPermissionAccess serializable, flag

System.Security.Permissions
(mscorlib.dll) *enum*

```
public enum RegistryPermissionAccess {  
    NoAccess = 0x00000000, Read = 0x00000001, Write = 0x00000002, Create = 0x00000004,  
    AllAccess = 0x00000007  
}
```

This enumeration is used to configure the type of registry access represented by a `RegistryPermission` object. The `Create`, `Read`, and `Write` values represent the ability to create, read, and write registry values, respectively. Each value is independent and does not include the permissions of the others. For example, `Write` does not encompass the permissions of the `Read` value. The `AllAccess` value represents the combination of `Read`, `Write`, and `Create`. If other combinations are needed, the values can be combined using a logical OR operation.

Hierarchy

```
System.Object      System.ValueType      System.Enum(System.IComparable,  
System.IFormattable, System.IConvertible)      RegistryPermissionAccess
```

Passed To

```
RegistryPermission.{AddPathList( ), GetPathList( ), RegistryPermission( ), SetPathList( )}
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

RegistryPermissionAttribute serializable

System.Security.Permissions
(mscorlib.dll) *sealed
class*

```
public sealed class RegistryPermissionAttribute : CodeAccessSecurityAttribute {
// Public Constructors
    public RegistryPermissionAttribute(SecurityAction action);
// Public Instance Properties
    public string All{set; get; }
    public string Create{set; get; }
    public string Read{set; get; }
    public string Write{set; get; }
// Public Instance Methods
    public override IPermission CreatePermission( );
// overrides SecurityAttribute
}
```

The `RegistryPermissionAttribute` attribute class provides declarative syntax support for the `RegistryPermission` class. The `All`, `Create`, `Read`, and `Write` properties take semicolon-separated lists of registry key names that the `RegistryPermissionAttribute` represents as the appropriate access level. Setting `Unrestricted` to true represents full access to all registry keys.

Hierarchy

```
System.Object      System.Attribute      SecurityAttribute
CodeAccessSecurityAttribute      RegistryPermissionAttribute
```

Valid On

Assembly, Class, Struct, Constructor, Method

[\[Team LiB \]](#)

[Team LiB]

ResourcePermissionBase

serializable

System.Security.Permissions
(system.dll)*abstract
class*

```

public abstract class ResourcePermissionBase : System.Security.CodeAccessPermission : IUnrestrictedPermission
// Protected Constructors
    protected ResourcePermissionBase( );
    protected ResourcePermissionBase(PermissionState state);
// Public Static Fields
    public const string Any;
// =*
    public const string Local;
// =.
// Protected Instance Properties
    protected Type PermissionAccessType {set; get; }
    protected string[] TagNames {set; get; }
// Public Instance Methods
    public override IPermission Copy( );
// overrides CodeAccessPermission
    public override void FromXml(System.Security.SecurityElement securityElement);
// overrides CodeAccessPermission
    public override IPermission Intersect(System.Security.IPermission target);
// overrides CodeAccessPermission
    public override bool IsSubsetOf(System.Security.IPermission target);
// overrides CodeAccessPermission
    public bool IsUnrestricted( );
// implements IUnrestrictedPermission
    public override SecurityElement ToXml( );
// overrides CodeAccessPermission
    public override IPermission Union(System.Security.IPermission target);
// overrides CodeAccessPermission
// Protected Instance Methods
    protected void AddPermissionAccess(ResourcePermissionBaseEntry entry);
    protected void Clear( );
    protected ResourcePermissionBaseEntry[] GetPermissionEntries( );
    protected void RemovePermissionAccess(ResourcePermissionBaseEntry entry);
}

```

The abstract ResourcePermissionBase class provides common functionality for Windows resource-oriented classes including the System.Diagnostics.EventLogPermission, System.Diagnostics.PerformanceCounterPermission, System.DirectoryServices.DirectoryServicesPermission, and System.ServiceProcess.ServiceProcessPermission classes.

Hierarchy

```
System.Object → System.Security.CodeAccessPermission(System.Security.IPermission,  
System.Security.ISecurityEncodable, System.Security.IStackWalk)  
ResourcePermissionBase(IUnrestrictedPermission)
```

Subclasses

```
System.Diagnostics.{EventLogPermission, PerformanceCounterPermission}
```

[Team LiB]

[Team LiB]

ResourcePermissionBaseEntry serializable

System.Security.Permissions
(system.dll) *class*

```
public class ResourcePermissionBaseEntry {  
    // Public Constructors  
    public ResourcePermissionBaseEntry( );  
    public ResourcePermissionBaseEntry(int permissionAccess, string[] permissionAccess.  
    // Public Instance Properties  
    public int PermissionAccess {get; }  
    public string[] PermissionAccessPath {get; }  
}
```

`ResourcePermissionBaseEntry` is used in conjunction with classes that extend the `ResourcePermissionBaseEntry`. It represents an administrable unit of code-access security.

Returned By

```
ResourcePermissionBase.GetPermissionEntries( )
```

Passed To

```
ResourcePermissionBase.{AddPermissionAccess( ), RemovePermissionAccess( )}
```

[Team LiB]

[Team LiB]

SecurityAction

ECMA 1.0, serializable

System.Security.Permissions
(mscorlib.dll)*enum*

```
public enum SecurityAction {
    Demand = 2, Assert = 3, Deny = 4, PermitOnly = 5, LinkDemand = 6, InheritanceDemand = 7,
    RequestMinimum = 8, RequestOptional = 9, RequestRefuse = 10
}
```

The `SecurityAction` enumeration contains values that represent the different declarative security statements. Every permission attribute class derived from the `CodeAccessSecurityAttribute` class defines a constructor that takes a value of the `SecurityAction` enumeration as an argument. The value of the `SecurityAction` argument identifies the action that the declarative security statement performs, and determines the program elements that are valid targets for the security attribute. The nine values of the `SecurityAction` enumeration fall into three categories: security demands, permission requests, and stack overrides. Declarative security demands and stack overrides can be applied to classes and functional members (i.e., methods, properties, and events), whereas permission requests are valid only at the assembly level.

The security demands include the values `Demand`, `InheritanceDemand`, and `LinkDemand`. `Demand` initiates a security demand, which will result in a stack walk when used with an attribute whose permission counterpart implements the `System.Security.IStackWalk` interfaces. This includes all code-access and identity permissions, but not `PrincipalPermission`, which only implements the `System.Security.IPermission` interface. The `InheritanceDemand` and `LinkDemand` statements have no imperative syntax equivalent. `InheritanceDemand` ensures that code trying to inherit a class or override a member has the demanded permission. `LinkDemand` ensures that the immediate calling code has the demanded permission, without invoking a stack walk.

The permission request statements also have no imperative statement equivalent, and include the values `RequestMinimum`, `RequestOptional`, and `RequestRefuse`. Permission requests provide a mechanism for assemblies to communicate details to the runtime and security administrators about the permissions they do and do not need. `RequestMinimum` specifies a set of permissions that the runtime must grant to an assembly in order for it to function correctly. After the runtime resolves the permissions for an assembly, if the grant set does not contain all of the requested permissions, then the runtime will throw a `System.Security.Policy.PolicyException` and will not load the assembly. The `RequestOptional` statement defines the maximum set of permissions that the runtime should grant to an assembly regardless of security policy. This is often used to specify a set of nonessential permissions that can be used if granted. The `RequestRefuse` statement specifies the set of permissions that the runtime should never grant to the assembly. `RequestRefuse` ensures code can never be granted permissions that it does not need, and so cannot be used by malicious code as a gateway to access secured actions or resources.

The stack override values -`Assert`, `Deny`, and `PermitOnly` - have the same effect as the methods of the same name defined in the `System.Security.IStackWalk` interface.

Hierarchy

`System.Object` → `System.ValueType` `System.Enum(System.IComparable, System.IFormattable, System.IConvertible)` `SecurityAction`

Returned By

`SecurityAttribute.Action`

Passed To

Multiple types

[Team LiB]

[\[Team LiB \]](#)

SecurityAttribute

ECMA 1.0, serializable

System.Security.Permissions
(mscorlib.dll) *abstract class*

```
public abstract class SecurityAttribute : Attribute {
// Public Constructors
    public SecurityAttribute(SecurityAction action);
// Public Instance Properties
    public SecurityAction Action{set; get; }
    public bool Unrestricted{set; get; }
// Public Instance Methods
    public abstract IPermission CreatePermission( );
}
```

The abstract `SecurityAttribute` class provides a base class from which the `CodeAccessSecurityAttribute` is derived. The `Unrestricted` property allows creation of security attributes that represent permissions in an unrestricted state. Although all derived permission attributes inherit `Unrestricted`, it is of use only when dealing with the attribute counterparts of permissions that implement the `IUnrestrictedPermission` interface. When required, the runtime calls the `CreatePermission()` method to create configured permission objects correctly from the configuration defined in a security attribute.

Hierarchy

System.Object System.Attribute SecurityAttribute

Subclasses

CodeAccessSecurityAttribute

Valid On

Assembly, Class, Struct, Constructor, Method

[\[Team LiB \]](#)

[Team LiB]

SecurityPermission ECMA 1.0, serializable

System.Security.Permissions *sealed*
 (mscorlib.dll) *class*

```
public sealed class SecurityPermission : System.Security.CodeAccessPermission : IUnrestrictedPermission, IBuiltInPermission {
    // Public Constructors
    public SecurityPermission(PermissionState state);
    public SecurityPermission(SecurityPermissionFlag flag);
    // Public Instance Properties
    public SecurityPermissionFlag Flags {set; get; }
    // Public Instance Methods
    public override IPermission Copy( );
    // overrides CodeAccessPermission
    public override void FromXml(System.Security.SecurityElement esd);
    // overrides CodeAccessPermission
    public override IPermission Intersect(System.Security.IPermission target);
    // overrides CodeAccessPermission
    public override bool IsSubsetOf(System.Security.IPermission target);
    // overrides CodeAccessPermission
    public bool IsUnrestricted( );
    // implements IUnrestrictedPermission
    public override SecurityElement ToXml( );
    // overrides CodeAccessPermission
    public override IPermission Union(System.Security.IPermission target);
    // overrides CodeAccessPermission
}
```

This code-access permission controls access to a set of 14 discrete operations that are critical to the secure environment. These include highly trusted operations, such as the ability of code to control evidence, create application domains, and skip verification. The specific operations represented by a `SecurityPermission` are specified at construction using members of the `SecurityPermissionFlag` enumeration. To retrieve or modify the flags represented by a `SecurityPermission` object, use the `Flags` property.

Hierarchy

```
System.Object
  System.Security.CodeAccessPermission(System.Security.IPermission,
  System.Security.ISecurityEncodable, System.Security.IStackWalk)
  SecurityPermission(IUnrestrictedPermission, IBuiltInPermission)
```

[Team LiB]

[\[Team LiB \]](#)

SecurityPermissionAttribute

serializable

System.Security.Permissions
(mscorlib.dll)*sealed
class*

```

public sealed class SecurityPermissionAttribute : CodeAccessSecurityAttribute {
// Public Constructors
    public SecurityPermissionAttribute(SecurityAction action);
// Public Instance Properties
    public bool Assertion{set; get; }
    public bool ControlAppDomain{set; get; }
    public bool ControlDomainPolicy{set; get; }
    public bool ControlEvidence{set; get; }
    public bool ControlPolicy{set; get; }
    public bool ControlPrincipal{set; get; }
    public bool ControlThread{set; get; }
    public bool Execution{set; get; }
    public SecurityPermissionFlag Flags{set; get; }
    public bool Infrastructure{set; get; }
    public bool RemotingConfiguration{set; get; }
    public bool SerializationFormatter{set; get; }
    public bool SkipVerification{set; get; }
    public bool UnmanagedCode{set; get; }
// Public Instance Methods
    public override IPermission CreatePermission( );
// overrides SecurityAttribute
}

```

This permission attribute provides declarative security support for the `SecurityPermission` code-access permission. To configure the set of security operations represented by a `SecurityPermissionAttribute`, assign values from the `SecurityPermissionFlag` enumeration to the `Flags` property. Alternately, `SecurityPermissionAttribute` provides properties that map to the individual values defined in the `SecurityPermissionFlag` enumeration. Setting these properties to true allows the configuration of a `SecurityPermissionAttribute` to represent any set of security operations. Setting `Unrestricted` to true creates a `SecurityPermissionAttribute` that represents access to all security operations and is equivalent to the `SecurityPermissionFlag.AllFlags` value.

Hierarchy

```

System.Object      System.Attribute      SecurityAttribute
CodeAccessSecurityAttribute  SecurityPermissionAttribute

```

Valid On

Assembly, Class, Struct, Constructor, Method

[\[Team LiB \]](#)

[Team LiB]

SecurityPermissionFlag

ECMA 1.0,
serializable, flagSystem.Security.Permissions
(mscorlib.dll)*enum*

```
public enum SecurityPermissionFlag {
    NoFlags = 0x00000000, Assertion = 0x00000001,
    UnmanagedCode = 0x00000002, SkipVerification = 0x00000004, Execution = 0x00000008,
    ControlThread = 0x00000010, ControlEvidence = 0x00000020, ControlPolicy = 0x00000040,
    SerializationFormatter = 0x00000080, ControlDomainPolicy = 0x00000100, ControlPrincipal = 0x00000200,
    ControlAppDomain = 0x00000400, RemotingConfiguration = 0x00000800, Infrastructure = 0x00001000,
    BindingRedirects = 0x00002000, AllFlags = 0x00003FFF
}
```

The values of the `SecurityPermissionFlag` enumeration specify the security actions that a `SecurityPermissionAttribute` object represents. Of the values we have discussed in this book, the most important are `Execution`, without which code cannot run in the .NET Framework. `SkipVerification` gives unverifiable code the ability to run, and `UnmanagedCode` gives code the ability to run unmanaged (native) code through `PInvoke` or `COMInterop`. `Assertion` allows code to assert any permission that it has been granted-see the `System.Security.ISecurityPermission` details.

`ControlAppDomain` represents the ability to create and manipulate application domains, `ControlDomainPolicy` allows code to assign security policy to an application domain, and `ControlPrincipal` controls the ability of code to manage an application domain (used in role-based security). `ControlEvidence` allows code to manipulate the evidence of an application domain as it is created or an assembly as it is loaded, and `ControlPolicy` means that code can view and modify security policy.

The `AllFlags` value represents a combination of all values, and `NoFlags` represents no permission. Any combination of flags can be represented using logical OR operations to combine the member values.

Hierarchy

```
System.Object      System.ValueType      System.Enum(System.IComparable, System.IFormattable)
System.IConvertible SecurityPermissionFlag
```

Returned By

```
SecurityPermission.Flags, SecurityPermissionAttribute.Flags
```

Passed To

```
SecurityPermission.{Flags, SecurityPermission( )}, SecurityPermissionAttribute.Flags
```

[Team LiB]

[Team LiB]

SiteIdentityPermission

serializable

System.Security.Permissions
(mscorlib.dll)*sealed
class*

```

public sealed class SiteIdentityPermission : System.Security.CodeAccessPermission : IBu:
// Public Constructors
    public SiteIdentityPermission(PermissionState state);
    public SiteIdentityPermission(string site);
// Public Instance Properties
    public string Site{set; get; }
// Public Instance Methods
    public override IPermission Copy( );
// overrides CodeAccessPermission
    public override void FromXml(System.Security.SecurityElement esd);
// overrides CodeAccessPermission
    public override IPermission Intersect(System.Security.IPermission target);
// overrides CodeAccessPermission
    public override bool IsSubsetOf(System.Security.IPermission target);
// overrides CodeAccessPermission
    public override SecurityElement ToXml( );
// overrides CodeAccessPermission
    public override IPermission Union(System.Security.IPermission target);
// overrides CodeAccessPermission
}

```

The `SiteIdentityPermission` class represents the `System.Security.Policy.Site` evidence of an asse domain, allowing code to invoke imperative security statements based on the web site from which code example, code could ensure that only assemblies run from the <https://www.oreilly.com> site have access functionality.

To create a `SiteIdentityPermission` object representing a specific source web site, pass a `System.Str` site address to the `SiteIdentityPermission` constructor; the site can include the "*" wildcard character to modify the web site represented by a `SiteIdentityPermission`, use the `Site` property.

Hierarchy

```

System.Object      System.Security.CodeAccessPermission(System.Security.IPermission,
System.Security.ISecurityEncodable, System.Security.IStackWalk)
SiteIdentityPermission( IBuiltInPermission)

```

[Team LiB]

[\[Team LiB \]](#)

SiteIdentityPermissionAttribute serializable

System.Security.Permissions *sealed*
 (mscorlib.dll) *class*

```
public sealed class SiteIdentityPermissionAttribute : CodeAccessSecurityAttribute {
// Public Constructors
    public SiteIdentityPermissionAttribute(SecurityAction action);
// Public Instance Properties
    public string Site{set; get; }
// Public Instance Methods
    public override IPermission CreatePermission( );
// overrides SecurityAttribute
}
```

The `SiteIdentityPermissionAttribute` class provides declarative syntax support for the `SiteIdentityPermission`. Specify the web site represented by the `SiteIdentityPermissionAttribute` by assigning a `System.String` containing the web site name to the `Site` property.

Hierarchy

```
System.Object      System.Attribute      SecurityAttribute
CodeAccessSecurityAttribute      SiteIdentityPermissionAttribute
```

Valid On

Assembly, Class, Struct, Constructor, Method

[\[Team LiB \]](#)

[Team LiB]

StrongNameIdentityPermission serializable

System.Security.Permissions *sealed*
(mscorlib.dll) *class*

```
public sealed class StrongNameIdentityPermission : System.Security.CodeAccessPermission
// Public Constructors
    public StrongNameIdentityPermission(PermissionState state);
    public StrongNameIdentityPermission(StrongNamePublicKeyBlob blob, string name, Version version);
// Public Instance Properties
    public string Name {set; get; }
    public StrongNamePublicKeyBlob PublicKey {set; get; }
    public Version Version {set; get; }
// Public Instance Methods
    public override IPermission Copy( );
// overrides CodeAccessPermission
    public override void FromXml(System.Security.SecurityElement e);
// overrides CodeAccessPermission
    public override IPermission Intersect(System.Security.IPermission target);
// overrides CodeAccessPermission
    public override bool IsSubsetOf(System.Security.IPermission target);
// overrides CodeAccessPermission
    public override SecurityElement ToXml( );
// overrides CodeAccessPermission
    public override IPermission Union(System.Security.IPermission target);
// overrides CodeAccessPermission
}
```

This identity permission represents the `System.Security.Policy.StrongName` evidence that an assembly was presented to the runtime when it was loaded or created. The `StrongNameIdentityPermission` class enables runtime decisions based on the strong name of an assembly, allowing code to ensure that only specific assemblies can access functionality.

The `StrongNameIdentityPermission` class provides a constructor that takes the name, version, and public key data of an existing `StrongNameIdentityPermission` object. The `Name`, `Version`, and `PublicKey` properties of a `StrongNameIdentityPermission` object provide the name, version, and public key data of an existing `StrongNameIdentityPermission` object. `Name` is a `System.String` containing the strong name of the assembly. `Version` uses a `System.Version` object to represent the four elements of a strong-named assembly's version number: these are major, minor, build, and revision. The major and minor elements are mandatory, but build and revision are optional. The `PublicKey` property uses a `StrongNamePublicKeyBlob` object to represent the public key data of the strong name.

Hierarchy

```
System.Object      System.Security.CodeAccessPermission(System.Security.IPermission,
```

```
System.Security.ISecurityEncodable , System.Security.IStackWalk)  
StrongNameIdentityPermission( IBuiltInPermission)
```

[Team LiB]

[Team LiB]

StrongNameIdentityPermissionAttribute serializable

System.Security.Permissions
(mscorlib.dll)

sealed
class

```
public sealed class StrongNameIdentityPermissionAttribute : CodeAccessSecurityAttribute
// Public Constructors
    public StrongNameIdentityPermissionAttribute(SecurityAction action);
// Public Instance Properties
    public string Name{set; get; }
    public string PublicKey{set; get; }
    public string Version{set; get; }
// Public Instance Methods
    public override IPermission CreatePermission( );
// overrides SecurityAttribute
}
```

This attribute class provides declarative syntax support for the `StrongNameIdentityPermission` class. The `Name`, `PublicKey`, and `Version` attributes all take `System.String` objects containing the information that defines the strong name; the `StrongNameIdentityPermissionAttribute` converts these string values to the appropriate types.

Hierarchy

```
System.Object      System.Attribute      SecurityAttribute      CodeAccessSecurityAttribute
      StrongNameIdentityPermissionAttribute
```

Valid On

Assembly, Class, Struct, Constructor, Method

[Team LiB]

[\[Team LiB \]](#)

StrongNamePublicKeyBlob

serializable

System.Security.Permissions
(mscorlib.dll)*sealed
class*

```
public sealed class StrongNamePublicKeyBlob {
// Public Constructors
    public StrongNamePublicKeyBlob(byte[] publicKey);
// Public Instance Methods
    public override bool Equals(object obj);
// overrides object
    public override int GetHashCode( );
// overrides object
    public override string ToString( );
// overrides object
}
```

The `StrongNamePublicKeyBlob` class is used in conjunction with the constructor and `PublicKey` properties of the `StrongNameIdentityPermission` class. It provides a convenient container for passing byte arrays that represent the public key component of a strong name. A byte array containing the raw public key data must be passed to the `StrongNamePublicKeyBlob` constructor because no properties or methods are provided to manipulate the data.

Returned By

```
StrongNameIdentityPermission.PublicKey, System.Security.Policy.StrongName.PublicKey,
System.Security.Policy.StrongNameMembershipCondition.PublicKey
```

Passed To

```
StrongNameIdentityPermission.{PublicKey, StrongNameIdentityPermission( )},
System.Security.Policy.StrongName.StrongName( ),
System.Security.Policy.StrongNameMembershipCondition.{PublicKey,
StrongNameMembershipCondition( )}
```

[\[Team LiB \]](#)

[Team LiB]

UIPermission

serializable

System.Security.Permissions
(mscorlib.dll)*sealed
class*

```

public sealed class UIPermission : System.Security.CodeAccessPermission : IUnrestrictedI
    IBuiltInPermission {
// Public Constructors
    public UIPermission(PermissionState state);
    public UIPermission(UIPermissionClipboard clipboardFlag);
    public UIPermission(UIPermissionWindow windowFlag);
    public UIPermission(UIPermissionWindow windowFlag, UIPermissionClipboard clipboardFl.
// Public Instance Properties
    public UIPermissionClipboard Clipboard{set; get; }
    public UIPermissionWindow Window{set; get; }
// Public Instance Methods
    public override IPermission Copy( );
// overrides CodeAccessPermission
    public override void FromXml(System.Security.SecurityElement esd);
// overrides CodeAccessPermission
    public override IPermission Intersect(System.Security.IPermission target);
// overrides CodeAccessPermission
    public override bool IsSubsetOf(System.Security.IPermission target);
// overrides CodeAccessPermission
    public bool IsUnrestricted( );
// implements IUnrestrictedPermission
    public override SecurityElement ToXml( );
// overrides CodeAccessPermission
    public override IPermission Union(System.Security.IPermission target);
// overrides CodeAccessPermission
}

```

The `UIPermission` class controls the ability of code to access clipboard functionality as well as various w elements of the user interface. Values from the `UIPermissionClipboard` and `UIPermissionWindow` enur used in the `UIPermission` constructor to configure the level of access represented. The `Clipboard` and `W` properties allow code to access and modify the configuration after construction.

Hierarchy

```

System.Object      System.Security.CodeAccessPermission(System.Security.IPermission,
System.Security.ISecurityEncodable , System.Security.IStackWalk)
UIPermission(IUnrestrictedPermission, IBuiltInPermission)

```

[Team LiB]

[\[Team LiB \]](#)

UIPermissionAttribute

serializable

System.Security.Permissions
(mscorlib.dll)*sealed
class*

```
public sealed class UIPermissionAttribute : CodeAccessSecurityAttribute {
// Public Constructors
    public UIPermissionAttribute(SecurityAction action);
// Public Instance Properties
    public UIPermissionClipboard Clipboard{set; get; }
    public UIPermissionWindow Window{set; get; }
// Public Instance Methods
    public override IPermission CreatePermission( );
// overrides SecurityAttribute
}
```

The `UIPermissionAttribute` class provides declarative syntax support for the `UIPermission` class. The `Clipboard` property takes a value from the `UIPermissionClipboard` enumeration to configure the access granted to the clipboard, and the `Window` property takes a value from the `UIPermissionWindow` enumeration to configure the access granted to various window elements. Setting `Unrestricted` to true creates a `UIPermissionAttribute` that represents unrestricted access to all window and clipboard functionality

Hierarchy

```
System.Object      System.Attribute      SecurityAttribute
CodeAccessSecurityAttribute  UIPermissionAttribute
```

Valid On

Assembly, Class, Struct, Constructor, Method

[\[Team LiB \]](#)

[\[Team LiB \]](#)

UIPermissionClipboard

serializable

System.Security.Permissions
(mscorlib.dll)*enum*

```
public enum UIPermissionClipboard {  
    NoClipboard = 0, OwnClipboard = 1, AllClipboard = 2  
}
```

Values of the `UIPermissionClipboard` enumeration are used to specify the type of clipboard access represented by a `UIPermission` object. `OwnClipboard` means that code can freely place data into the clipboard but cannot programmatically read the clipboard contents. `AllClipboard` represents unrestricted clipboard access.

Hierarchy

```
System.Object      System.ValueType      System.Enum(System.IComparable,  
System.IFormattable, System.IConvertible)      UIPermissionClipboard
```

Returned By

```
UIPermission.Clipboard, UIPermissionAttribute.Clipboard
```

Passed To

```
UIPermission.Clipboard, UIPermissionAttribute.Clipboard
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

UIPermissionWindow

serializable

System.Security.Permissions
(mscorlib.dll)*enum*

```
public enum UIPermissionWindow {  
    NoWindows = 0, SafeSubWindows = 1, SafeTopLevelWindows = 2, AllWindows = 3  
}
```

The `UIPermissionWindow` enumeration contains values that the `UIPermission` class uses to represent the type of window elements code is permitted to use. `SafeSubWindows` represents the ability to use safe subwindows such as message boxes and common dialog controls. `SafeTopLevelWindows` represents permission to use top-level windows that are considered safe because they are clearly labeled to avoid spoofing attacks. `AllWindows` represents the ability to access all windows elements.

Hierarchy

```
System.Object      System.ValueType      System.Enum(System.IComparable,  
System.IFormattable, System.IConvertible)      UIPermissionWindow
```

Returned By

```
UIPermission.Window, UIPermissionAttribute.Window
```

Passed To

```
UIPermission.{UIPermission( ), Window}, UIPermissionAttribute.Window
```

[\[Team LiB \]](#)

[Team LiB]

UrlIdentityPermission

serializable

System.Security.Permissions
(mscorlib.dll)*sealed
class*

```
public sealed class UrlIdentityPermission : System.Security.CodeAccessPermission : IBuiltInPermission
// Public Constructors
    public UrlIdentityPermission(PermissionState state);
    public UrlIdentityPermission(string site);
// Public Instance Properties
    public string Url{set; get; }
// Public Instance Methods
    public override IPermission Copy( );
// overrides CodeAccessPermission
    public override void FromXml(System.Security.SecurityElement esd);
// overrides CodeAccessPermission
    public override IPermission Intersect(System.Security.IPermission target);
// overrides CodeAccessPermission
    public override bool IsSubsetOf(System.Security.IPermission target);
// overrides CodeAccessPermission
    public override SecurityElement ToXml( );
// overrides CodeAccessPermission
    public override IPermission Union(System.Security.IPermission target);
// overrides CodeAccessPermission
}
```

The `UrlIdentityPermission` class represents the `System.Security.Policy.Url` evidence of an assembly domain. The `UrlIdentityPermission` class allows code to make security decisions based on the URL from which the code was loaded using imperative security statements. `UrlIdentityPermission` is useful for ensuring that only the code from particular locations can execute important actions and resources. For example, only code loaded from the local network can run methods that manage user accounts.

To create a `UrlIdentityPermission` object representing a specific source URL, pass a `System.String` to the `UrlIdentityPermission` constructor. The URL can include the "*" wildcard character as its last character. The URL represented by a `UrlIdentityPermission` can also be configured using its `Url` property.

Hierarchy

```
System.Object      System.Security.CodeAccessPermission(System.Security.IPermission,
System.Security.ISecurityEncodable, System.Security.IStackWalk)
UrlIdentityPermission(IBuiltInPermission)
```

[Team LiB]

[\[Team LiB \]](#)

UrlIdentityPermissionAttribute serializable

System.Security.Permissions *sealed*
(mscorlib.dll) *class*

```
public sealed class UrlIdentityPermissionAttribute : CodeAccessSecurityAttribute {  
    // Public Constructors  
    public UrlIdentityPermissionAttribute(SecurityAction action);  
    // Public Instance Properties  
    public string Url{set; get; }  
    // Public Instance Methods  
    public override IPermission CreatePermission( );  
    // overrides SecurityAttribute  
}
```

The `UrlIdentityPermissionAttribute` attribute class provides declarative syntax support for the `UrlIdentityPermission` class. The URL is specified by assigning a `System.String` containing the URL to the `Url` property.

Hierarchy

```
System.Object      System.Attribute      SecurityAttribute  
CodeAccessSecurityAttribute      UrlIdentityPermissionAttribute
```

Valid On

Assembly, Class, Struct, Constructor, Method

[\[Team LiB \]](#)

[Team LiB]

ZoneIdentityPermission

serializable

System.Security.Permissions
(mscorlib.dll)*sealed*
class

```

public sealed class ZoneIdentityPermission : System.Security.CodeAccessPermission : IBu:
// Public Constructors
    public ZoneIdentityPermission(PermissionState state);
    public ZoneIdentityPermission(System.Security.SecurityZone zone);
// Public Instance Properties
    public SecurityZone SecurityZone{set; get; }
// Public Instance Methods
    public override IPermission Copy( );
// overrides CodeAccessPermission
    public override void FromXml(System.Security.SecurityElement esd);
// overrides CodeAccessPermission
    public override IPermission Intersect(System.Security.IPermission target);
// overrides CodeAccessPermission
    public override bool IsSubsetOf(System.Security.IPermission target);
// overrides CodeAccessPermission
    public override SecurityElement ToXml( );
// overrides CodeAccessPermission
    public override IPermission Union(System.Security.IPermission target);
// overrides CodeAccessPermission
}

```

This identity permission class represents the `System.Security.Policy.Zone` evidence presented by an application domain to the runtime. The `ZoneIdentityPermission` class allows code to invoke imperative based on the zone from which code originated. For example, code could ensure that only assemblies running intranet have access to methods that manipulate data in an important database.

To create a `ZoneIdentityPermission` object representing a specific zone, pass a member of the `System.Security.SecurityZone` enumeration to object constructor. The `System.Security.SecurityZone` contains values that represent the standard Internet Explorer security zones. To get or set the security zone of a `ZoneIdentityPermission` object, use its `SecurityZone` property.

Hierarchy

```

System.Object      System.Security.CodeAccessPermission(System.Security.IPermission,
System.Security.ISecurityEncodable, System.Security.IStackWalk)
ZoneIdentityPermission(IBuiltInPermission)

```

[Team LiB]

[\[Team LiB \]](#)

ZoneIdentityPermissionAttribute serializable

System.Security.Permissions
(mscorlib.dll) *sealed class*

```
public sealed class ZoneIdentityPermissionAttribute : CodeAccessSecurityAttribute {  
    // Public Constructors  
    public ZoneIdentityPermissionAttribute(SecurityAction action);  
    // Public Instance Properties  
    public SecurityZone Zone{set; get; }  
    // Public Instance Methods  
    public override IPermission CreatePermission( );  
    // overrides SecurityAttribute  
}
```

This attribute class provides declarative syntax support for the `ZoneIdentityPermission` class. The security zone is specified by assigning a value of the `System.Security.SecurityZone` enumeration to the `Zone` property.

Hierarchy

```
System.Object      System.Attribute      SecurityAttribute  
CodeAccessSecurityAttribute      ZoneIdentityPermissionAttribute
```

Valid On

Assembly, Class, Struct, Constructor, Method

[\[Team LiB \]](#)

[[Team LiB](#)]

Chapter 28. The System.Security.Policy Namespace

The `System.Security.Policy` namespace contains classes that represent evidence and the key elements of security policy, including membership conditions, code groups, and policy levels. Security policy is the set of configurable rules that provide a mapping between evidence and permissions. Specifically, the runtime uses security policy to determine which code-access permissions to grant to an assembly or application domain based on the set of evidence that the assembly or application domain presents—a process known as policy resolution.

.NET divides security policy into four policy levels: enterprise, machine, user, and application domain. Each policy level consists of a set of code groups organized into a tree structure. There are different types of code groups, but most contain a membership condition (based on the values of evidence objects) and a permission set that the code group grants to assemblies and application domains that qualify for membership. Some code groups contain a set of child code groups, which member assemblies and application domains are also compared to for membership, thus creating a tree structure.

During policy resolution, the runtime traverses the tree of code groups in each policy level and compares the evidence presented by the assembly or application domain with the membership condition of each code group. If the evidence meets the code group's membership condition, then the runtime grants the assembly or application domain the permissions contained in the code group's permission set. To calculate the permissions granted by a policy level, the runtime takes the union of all the permission sets granted by the code groups to which an assembly or application domain qualifies for membership. The final set of permissions is calculated by intersecting the permission set granted by each policy level. Different types of code groups and code group attributes can alter the way that the runtime combines code group permission sets.

[Figure 28-1](#) and [Figure 28-2](#) show the types in this namespace.

Figure 28-1. Policy objects from this namespace

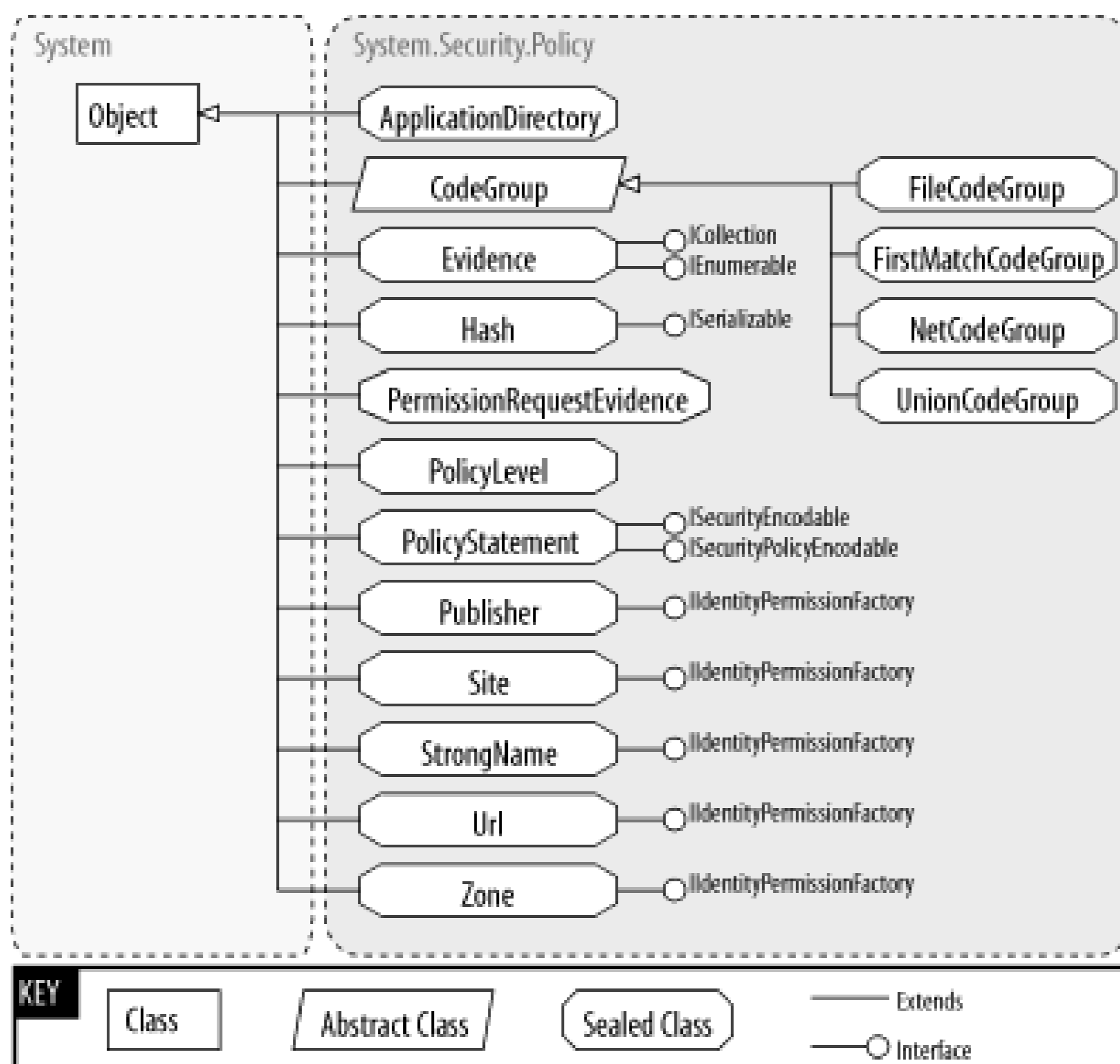
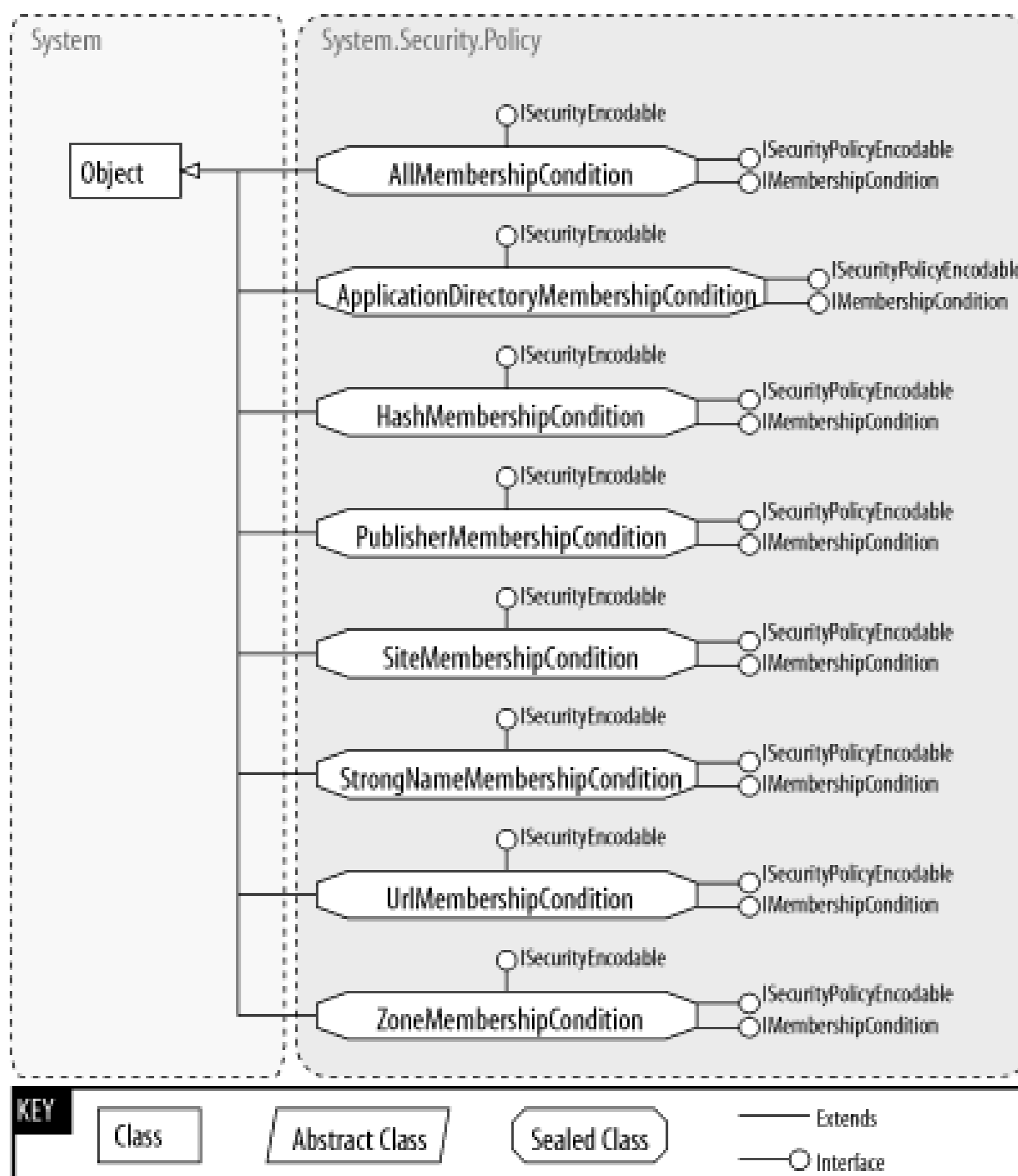


Figure 28-2. Membership conditions from this namespace



[Team LiB]

[Team LiB]

AllMembershipCondition

serializable

System.Security.Policy
(mscorlib.dll)*sealed
class*

```

public sealed class AllMembershipCondition : IMembershipCondition, System.Security.ISecurityPolicyEncodable, IConstantMembershipCondition {
    // Public Constructors
    public AllMembershipCondition( );
    // Public Instance Methods
    public bool Check(Evidence evidence);
    // implements IMembershipCondition
    public IMembershipCondition Copy( );
    // implements IMembershipCondition
    public override bool Equals(object o);
    // overrides object
    public void FromXml(System.Security.SecurityElement e);
    // implements System.Security.ISecurityEncodable
    public void FromXml(System.Security.SecurityElement e,
    // implements System.Security.ISecurityPolicyEncodable
        PolicyLevel level);
    public override int GetHashCode( );
    // overrides object
    public override string ToString( );
    // overrides object
    public SecurityElement ToXml( );
    // implements System.Security.ISecurityEncodable
    public SecurityElement ToXml(PolicyLevel level);
    // implements System.Security.ISecurityPolicyEncodable
}

```

This membership condition matches all code. The `Check()` method always returns true, which means that all application domains, regardless of their evidence, qualify for membership of a `CodeGroup` for which a `AllMembershipCondition` object is the membership condition.

[Team LiB]

[\[Team LiB \]](#)

ApplicationDirectory

serializable

System.Security.Policy
(mscorlib.dll)*sealed
class*

```
public sealed class ApplicationDirectory : IBuiltInEvidence {  
    // Public Constructors  
    public ApplicationDirectory(string name);  
    // Public Instance Properties  
    public string Directory {get; }  
    // Public Instance Methods  
    public object Copy( );  
    public override bool Equals(object o);  
    // overrides object  
    public override int GetHashCode( );  
    // overrides object  
    public override string ToString( );  
    // overrides object  
}
```

The `ApplicationDirectory` evidence class represents the root installation directory of an application. A URL representing the application installation path is passed as a `System.String` to the `ApplicationDirectory` constructor, and is accessible through the `Directory` property.

The `ApplicationDirectoryMembershipCondition` class uses `ApplicationDirectory` evidence during policy resolution to evaluate whether an assembly or application domain qualifies for membership of a `CodeGroup`.

[\[Team LiB \]](#)

[Team LiB]

ApplicationDirectoryMembershipCondition serializable

System.Security.Policy
(mscorlib.dll)

sealed
class

```
public sealed class ApplicationDirectoryMembershipCondition : IMembershipCondition, System.Security.ISecurityPolicyEncodable, IConstantMembershipCondition {
    // Public Constructors
    public ApplicationDirectoryMembershipCondition( );
    // Public Instance Methods
    public bool Check(Evidence evidence);
    // implements IMembershipCondition
    public IMembershipCondition Copy( );
    // implements IMembershipCondition
    public override bool Equals(object o);
    // overrides object
    public void FromXml(System.Security.SecurityElement e);
    // implements System.Security.ISecurityEncodable
    public void FromXml(System.Security.SecurityElement e,
    // implements System.Security.ISecurityPolicyEncodable
        PolicyLevel level);
    public override int GetHashCode( );
    // overrides object
    public override string ToString( );
    // overrides object
    public SecurityElement ToXml( );
    // implements System.Security.ISecurityEncodable
    public SecurityElement ToXml(PolicyLevel level);
    // implements System.Security.ISecurityPolicyEncodable
}
```

The `ApplicationDirectoryMembershipCondition` membership condition class uses the value of both `AssemblyEvidence` and `ApplicationDirectoryEvidence` to determine if an assembly or application domain qualifies for membership of a `CodeGroup`. The specified `Evidence` collection contains both `Url` and `ApplicationDirectory` evidence, and the `Url.Value` is a child of the `ApplicationDirectory.Directory` property.

[Team LiB]

[Team LiB]

CodeGroup

serializable

System.Security.Policy
(mscorlib.dll)*abstract
class*

```

public abstract class CodeGroup {
// Public Constructors
    public CodeGroup(IMembershipCondition membershipCondition, PolicyStatement policy);
// Public Instance Properties
    public virtual string AttributeString {get; }
    public IList Children {set; get; }
    public string Description {set; get; }
    public IMembershipCondition MembershipCondition {set; get; }
    public abstract string MergeLogic {get; }
    public string Name {set; get; }
    public virtual string PermissionSetName {get; }
    public PolicyStatement PolicyStatement {set; get; }
// Public Instance Methods
    public void AddChild(CodeGroup group);
    public abstract CodeGroup Copy( );
    public bool Equals(CodeGroup cg, bool compareChildren);
    public override bool Equals(object o);
// overrides object
    public void FromXml(System.Security.SecurityElement e);
    public void FromXml(System.Security.SecurityElement e, PolicyLevel level);
    public override int GetHashCode( );
// overrides object
    public void RemoveChild(CodeGroup group);
    public abstract PolicyStatement Resolve(Evidence evidence);
    public abstract CodeGroup ResolveMatchingCodeGroups(Evidence evidence);
    public SecurityElement ToXml( );
    public SecurityElement ToXml(PolicyLevel level);
// Protected Instance Methods
    protected virtual void CreateXml(System.Security.SecurityElement element, PolicyLevel level);
    protected virtual void ParseXml(System.Security.SecurityElement e, PolicyLevel level);
}

```

The abstract `CodeGroup` class provides the base class from which all concrete code group classes derive. Code groups are the fundamental building blocks of security policy. Each of the enterprise, machine, user, and domain policy levels consists of a hierarchy of code groups that descend from a single root `CodeGroup` object. The root `CodeGroup` for a policy level is accessible through the `PolicyLevel.RootCodeGroup` property.

Although implementation varies with each concrete subclass, the most important elements of a `CodeGroup` are its membership condition, policy statement, and children. The membership condition and policy statement are set at creation as arguments to the `CodeGroup` constructor, or later through the `MembershipCondition` and `PolicyStatement` properties.

The `MembershipCondition` property takes an object that implements the `IMembershipCondition` interface and provides the logic that determines whether an assembly or application domain qualifies for membership in a `CodeGroup`. The `PolicyStatement` property takes a `PolicyStatement` object that defines the effect the `CodeGroup` has on its members, including the permissions it grants and any attributes that the `CodeGroup` has.

Each `CodeGroup` contains a set of child `CodeGroup` objects each child can have its own children, and so on, establishing the code group tree structure used during policy resolution. The role of children in the policy resolution process varies with each subclass, and in some cases, the order of children is important. The `Children` property returns an ordered `System.Collections.IList` of child `CodeGroup` objects. To add and remove children use the `AddChild()` and `RemoveChild()` methods.

The most important method of the `CodeGroup` class is `Resolve()`, which takes an `Evidence` collection as an argument. Policy resolution of an assembly or application domain consists of the runtime calling the `Resolve` method on the root `CodeGroup` in each policy level and passing it the assembly or application domain's `Evidence` collection. Although it is not possible for a program to invoke policy resolution for an assembly or application domain, `Resolve()` and `ResolveMatchingCodeGroups()` are useful for testing the permissions and code group membership an assembly or application domain would be granted by the `CodeGroup` object. `Resolve()` returns a `PolicyStatement` that represents the net effect of all code groups in the tree to which the assembly or application domain qualified for membership.

Subclasses

`FileCodeGroup`, `FirstMatchCodeGroup`, `NetCodeGroup`, `UnionCodeGroup`

Returned By

`FileCodeGroup`.{`Copy()`, `ResolveMatchingCodeGroups()`}, `PolicyLevel`.{`ResolveMatchingCodeGroups()`, `RootCodeGroup`}

Passed To

`PolicyLevel`.`RootCodeGroup`

[Team LiB]

[\[Team LiB \]](#)

Evidence

CF 1.0, serializable

System.Security.Policy
(mscorlib.dll)*sealed
class*

```

public sealed class Evidence : ICollection, IEnumerable {
// Public Constructors
    public Evidence( );
    public Evidence(Evidence evidence);
    public Evidence(object[] hostEvidence, object[] assemblyEvidence);
// Public Instance Properties
    public int Count {get; }
// implements ICollection
    public bool IsReadOnly {get; }
    public bool IsSynchronized {get; }
// implements ICollection
    public bool Locked {set; get; }
    public object SyncRoot {get; }
// implements ICollection
// Public Instance Methods
    public void AddAssembly(object id);
    public void AddHost(object id);
    public void CopyTo(Array array, int index);
// implements ICollection
    public IEnumerator GetAssemblyEnumerator( );
    public IEnumerator GetEnumerator( );
// implements IEnumerable
    public IEnumerator GetHostEnumerator( );
    public void Merge(Evidence evidence);
}

```

The `Evidence` class is a specialized collection used to contain evidence objects. The `Evidence` class actually contains two sets of evidence objects: a collection of host evidence and a collection of assembly evidence. Host evidence is the primary type of evidence used as input by the runtime to drive security policy resolution. The runtime, or possibly a trusted host, specifies the host evidence. Host evidence is added to the `Evidence` collection using the `AddHost()` method and accessed through the `GetHostEnumerator()` method.

Assembly evidence is normally specified at build time by the assembly developer and embedded in the assembly file as a serialized resource. As the runtime loads the assembly, it deserializes any assembly evidence and places it inside the assembly's `Evidence` collection. A trusted host can also provide assembly evidence when it loads an assembly using the `AddAssembly()` method, although this is uncommon. The `GetAssemblyEnumerator()` method provides access to the assembly evidence contained in an `Evidence` collection.

Despite containing two separate collections, `Evidence` also provides members to manipulate the

combined contents of both collections at once. The `Count` property returns the total number of evidence objects, and the `GetEnumerator()` method returns a `System.Collections.IEnumerator` that enumerates across all objects in the `Evidence` collection.

During policy resolution, the `Evidence` collection of an assembly or application domain is passed to each `CodeGroup` through the `CodeGroup.Resolve()` method. The `CodeGroup` then passes the `Evidence` collection to the `IMembershipCondition.Check()` method of its membership condition object. The membership condition evaluates the `Evidence` collection to determine if it contains the appropriate evidence values to qualify for `CodeGroup` membership.

Returned By

```
System.AppDomain.Evidence, System.Reflection.Assembly.Evidence,  
System.Security.IEvidenceFactory.Evidence,  
System.Xml.XmlSecureResolver.CreateEvidenceForUrl( )
```

Passed To

Multiple types

[\[Team LiB \]](#)

[Team LiB]

FileCodeGroup

serializable

System.Security.Policy
(mscorlib.dll)*sealed
class*

```

public sealed class FileCodeGroup : CodeGroup {
    // Public Constructors
    public FileCodeGroup(IMembershipCondition membershipCondition,
        System.Security.Permissions.FileIOPermissionAccess access);
    // Public Instance Properties
    public override string AttributeString {get; }
    // overrides CodeGroup
    public override string MergeLogic {get; }
    // overrides CodeGroup
    public override string PermissionSetName {get; }
    // overrides CodeGroup
    // Public Instance Methods
    public override CodeGroup Copy( );
    // overrides CodeGroup
    public override bool Equals(object o);
    // overrides CodeGroup
    public override int GetHashCode( );
    // overrides CodeGroup
    public override PolicyStatement Resolve(Evidence evidence);
    // overrides CodeGroup
    public override CodeGroup ResolveMatchingCodeGroups(Evidence evidence);
    // overrides CodeGroup
    // Protected Instance Methods
    protected override void CreateXml(System.Security.SecurityElement element, PolicyLevel level);
    // overrides CodeGroup
    protected override void ParseXml(System.Security.SecurityElement e, PolicyLevel level);
    // overrides CodeGroup
}

```

This class is a subclass of the `CodeGroup` class. `FileCodeGroup` does not support attributes and does not have a statically defined permission set, and so there is no need to assign it a `PolicyStatement`. With each call to `Resolve()`, the `FileCodeGroup` object determines if the provided `Evidence` collection contains `Url` evidence that represents a URL with the "file://" protocol. If so, the `FileCodeGroup` dynamically generates a permission set that contains a `System.Security.Permissions.FileIOPermission` that grants access to the directory specified by the `Url` evidence. The type of file access to grant (read, write, and so on) is specified in the `FileCodeGroup` constructor using a member of the `System.Security.Permissions.FileIOPermissionAccess` enumeration.

Members of a `FileCodeGroup` are tested for membership against all of its child code groups. The final permission granted by the `FileCodeGroup` is the union of its permission set and the permission sets of the children. An assembly or application domain also qualifies for membership.

Hierarchy

`System.Object` → `CodeGroup` → `FileCodeGroup`

[Team LiB]

[Team LiB]

FirstMatchCodeGroup

serializable

System.Security.Policy
(mscorlib.dll)*sealed*
class

```

public sealed class FirstMatchCodeGroup : CodeGroup {
// Public Constructors
    public FirstMatchCodeGroup(IMembershipCondition membershipCondition, PolicyStatement
// Public Instance Properties
    public override string MergeLogic{get; }
// overrides CodeGroup
// Public Instance Methods
    public override CodeGroup Copy( );
// overrides CodeGroup
    public override PolicyStatement Resolve(Evidence evidence);
// overrides CodeGroup
    public override CodeGroup ResolveMatchingCodeGroups(Evidence evidence);
// overrides CodeGroup
}

```

The `FirstMatchCodeGroup` class is a concrete subclass of the `CodeGroup` class. Members of a `FirstMatchCodeGroup` are tested for membership against its child code groups in order. After qualifying membership to a child group, the comparison stops, which means that the order of the children is important. The final permission set granted by the `FirstMatchCodeGroup` is the union of its permission set and at most children.

Hierarchy

System.Object CodeGroup FirstMatchCodeGroup

[Team LiB]

[Team LiB]

Hash

serializable

System.Security.Policy
(mscorlib.dll)

sealed
class

```
public sealed class Hash : System.Runtime.Serialization.ISerializable, IBuiltInEvidence
// Public Constructors
    public Hash(System.Reflection.Assembly assembly);
// Public Instance Properties
    public byte[] MD5{get; }
    public byte[] SHA1{get; }
// Public Instance Methods
    public byte[] GenerateHash(System.Security.Cryptography.HashAlgorithm hashAlg);
    public void GetObjectData(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context);
// implements ISerializable
    public override string ToString( );
// overrides object
}
```

The **Hash** evidence class represents the hash value of an assembly's content. Unlike the other standard evidence classes, **Hash** provides multiple evidence values (hash values) from a single set of source data. The evidence data source is a **System.Reflection.Assembly** object that is passed to the **Hash** object constructor. The **MD5** and **SHA1** properties calculate and return the MD5 and SHA1 hash values for the content of the **Assembly**. The **GenerateHash()** method allows any hashing implementation that derive from **System.Security.Cryptography.HashAlgorithm** to be used to generate a hash value of the **Assembly** content.

During policy resolution, the **HashMembershipCondition** class uses **Hash** evidence to evaluate whether code qualifies for membership of a **CodeGroup**.

[Team LiB]

[Team LiB]

HashMembershipCondition

serializable

System.Security.Policy
(mscorlib.dll)*sealed
class*

```

public sealed class HashMembershipCondition : IMembershipCondition,
    System.Security.ISecurityEncodable, System.Security.ISecurityPolicyEncodable {
// Public Constructors
    public HashMembershipCondition(System.Security.Cryptography.HashAlgorithm hashAlg, byte[]
// Public Instance Properties
    public HashAlgorithm HashAlgorithm {set; get; }
    public byte[] HashValue {set; get; }
// Public Instance Methods
    public bool Check(Evidence evidence);
// implements IMembershipCondition
    public IMembershipCondition Copy( );
// implements IMembershipCondition
    public override bool Equals(object o);
// overrides object
    public void FromXml(System.Security.SecurityElement e);
// implements System.Security.ISecurityEncodable
    public void FromXml(System.Security.SecurityElement e, PolicyLevel level);
// implements System.Security.ISecurityPolicyEncodable
    public override int GetHashCode( );
// overrides object
    public override string ToString( );
// overrides object
    public SecurityElement ToXml( );
// implements System.Security.ISecurityEncodable
    public SecurityElement ToXml(PolicyLevel level);
// implements System.Security.ISecurityPolicyEncodable
}

```

This membership condition class uses the value of `Hash` evidence to determine if an assembly or application qualifies for membership of a `CodeGroup`. The `HashValue` property contains the hash value required for the `HashAlgorithm` property contains a `System.Security.Cryptography.HashAlgorithm` object that in hash algorithm used to calculate the hash value.

When policy resolution calls the `Check()` method, if the provided `Evidence` collection contains a `Hash` of `HashMembershipCondition` object calls the `Hash.GenerateHash()` method and passes the `System.Security.Cryptography.HashAlgorithm` object from its `HashAlgorithm` property. If the return matches its `HashValue` property, then `CodeGroup` membership is granted.

[Team LiB]

[\[Team LiB \]](#)

IIdentityPermissionFactory

System.Security.Policy
(mscorlib.dll)

interface

```
public interface IIdentityPermissionFactory {  
    // Public Instance Methods  
    public IPermission CreateIdentityPermission(Evidence evidence);  
}
```

During security policy resolution, host evidence objects that implement the `IIdentityPermissionFactory` result in an identity permission of the appropriate type being included in the grant set of the assembly or application domain that presents the evidence. The runtime calls the `CreateIdentityPermission()` method of the evidence object, which returns a `System.Security.IPermission` object of the appropriate type. The evidence objects that implement `IIdentityPermissionFactory` are `Publisher`, `Site`, `StrongName`, `Url`, and `Zone`.

Implemented By

`Publisher`, `Site`, `StrongName`, `Url`, `Zone`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

I MembershipCondition

System.Security.Policy
(mscorlib.dll)

interface

```
public interface IMembershipCondition : System.Security.ISecurityEncodable,
    System.Security.ISecurityPolicyEncodable {
    // Public Instance Methods
    public bool Check(Evidence evidence);
    public IMembershipCondition Copy( );
    public bool Equals(object obj);
    public string ToString( );
}
```

The `IMembershipCondition` interface defines the functionality that all membership condition classes must implement. Membership condition classes define the tests that assemblies and application domains must pass in order to qualify for membership of a `CodeGroup`. The `Check()` method takes an `Evidence` collection argument and returns true if it contains evidence objects that meet the set of criteria defined within the implementing membership condition class.

Implemented By

```
AllMembershipCondition, ApplicationDirectoryMembershipCondition,
HashMembershipCondition, PublisherMembershipCondition, SiteMembershipCondition,
StrongNameMembershipCondition, UrlMembershipCondition, ZoneMembershipCondition
```

Returned By

```
AllMembershipCondition.Copy( ), ApplicationDirectoryMembershipCondition.Copy( ),
CodeGroup.MembershipCondition, HashMembershipCondition.Copy( ),
PublisherMembershipCondition.Copy( ), SiteMembershipCondition.Copy( ),
StrongNameMembershipCondition.Copy( ), UrlMembershipCondition.Copy( ),
ZoneMembershipCondition.Copy( )
```

Passed To

```
CodeGroup.{CodeGroup( ), MembershipCondition}, FileCodeGroup.FileCodeGroup( ),
FirstMatchCodeGroup.FirstMatchCodeGroup( ), NetCodeGroup.NetCodeGroup( ),
UnionCodeGroup.UnionCodeGroup( )
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

NetCodeGroup

serializable

System.Security.Policy
(mscorlib.dll)*sealed
class*

```

public sealed class NetCodeGroup : CodeGroup {
// Public Constructors
    public NetCodeGroup(IMembershipCondition membershipCondition);
// Public Instance Properties
    public override string AttributeString{get; }
// overrides CodeGroup
    public override string MergeLogic{get; }
// overrides CodeGroup
    public override string PermissionSetName{get; }
// overrides CodeGroup
// Public Instance Methods
    public override CodeGroup Copy( );
// overrides CodeGroup
    public override PolicyStatement Resolve(Evidence evidence);
// overrides CodeGroup
    public override CodeGroup ResolveMatchingCodeGroups(Evidence evidence);
// overrides CodeGroup
}

```

The `NetCodeGroup` class is a concrete subclass of the `CodeGroup` class. `NetCodeGroup` does not support attributes and does not contain a statically defined permission set, and so does not require a `PolicyStatement`. When the `Resolve()` method is called, the `NetCodeGroup` object determines if the specified `Evidence` collection contains `Url` evidence that represents a URL with the "http://" or "https://" protocols. If so, the `NetCodeGroup` dynamically generates a permission set that contains a `System.Net.WebPermission` that grants access to the web site specified in the `Url` evidence.

Members of a `NetCodeGroup` are tested for membership against all of its child code groups. The final permission set granted by the `NetCodeGroup` is the union of its permission set and the permission sets of the children to which the assembly or application domain also qualifies for membership.

Hierarchy

```

System.Object    CodeGroup    NetCodeGroup

```

[\[Team LiB \]](#)

[Team LiB]

PermissionRequestEvidence

serializable

System.Security.Policy
(mscorlib.dll)*sealed
class*

```

public sealed class PermissionRequestEvidence : IBuiltInEvidence {
// Public Constructors
    public PermissionRequestEvidence(System.Security.PermissionSet request,
        System.Security.PermissionSet optional, System.Security.PermissionSet denied);
// Public Instance Properties
    public PermissionSet DeniedPermissions{get; }
    public PermissionSet OptionalPermissions{get; }
    public PermissionSet RequestedPermissions{get; }
// Public Instance Methods
    public PermissionRequestEvidence Copy( );
    public override string ToString( );
// overrides object
}

```

The `PermissionRequestEvidence` class is a special evidence class used by the runtime during policy resolution to represent the declarative permission requests contained within an assembly. The `RequestedPermissions`, `OptionalPermissions`, and `DeniedPermissions` properties each return a `System.Security.PermissionSet` containing the minimal, optional, and refused permission requests made by the assembly. An assembly's `PermissionRequestEvidence` is contained in the assembly's `Evidence` collection during policy resolution but does not remain as part of the assembly's `Evidence` collection after policy resolution is complete.

[Team LiB]

[\[Team LiB \]](#)

PolicyException

serializable

System.Security.Policy (mscorlib.dll) *class*

```
public class PolicyException : SystemException {  
    // Public Constructors  
    public PolicyException( );  
    public PolicyException(string message);  
    public PolicyException(string message, Exception exception);  
    // Protected Constructors  
    protected PolicyException(System.Runtime.Serialization.SerializationInfo info,  
        System.Runtime.Serialization.StreamingContext context);  
}
```

This exception is thrown when the runtime encounters problems processing or applying security policy—for example, when code attempts to set an application domain's policy that has already been set or when the minimum permission requests of an assembly cannot be fulfilled by the runtime.

Hierarchy

```
System.Object      System.Exception(System.Runtime.Serialization.ISerializable)  
System.SystemException  PolicyException
```

[\[Team LiB \]](#)

[Team LiB]

PolicyLevel

serializable

System.Security.Policy
(mscorlib.dll)*sealed
class*

```

public sealed class PolicyLevel {
// Public Instance Properties
    public IList FullTrustAssemblies {get; }
    public string Label {get; }
    public IList NamedPermissionSets {get; }
    public CodeGroup RootCodeGroup {set; get; }
    public string StoreLocation {get; }
// Public Static Methods
    public static PolicyLevel CreateAppDomainLevel( );
// Public Instance Methods
    public void AddFullTrustAssembly(StrongName sn);
    public void AddFullTrustAssembly(StrongNameMembershipCondition snMC);
    public void AddNamedPermissionSet(System.Security.NamedPermissionSet permSet);
    public NamedPermissionSet ChangeNamedPermissionSet(string name, System.Security.Perm:
    public void FromXml(System.Security.SecurityElement e);
    public NamedPermissionSet GetNamedPermissionSet(string name);
    public void Recover( );
    public void RemoveFullTrustAssembly(StrongName sn);
    public void RemoveFullTrustAssembly(StrongNameMembershipCondition snMC);
    public NamedPermissionSet RemoveNamedPermissionSet(System.Security.NamedPermissionSet
    public NamedPermissionSet RemoveNamedPermissionSet(string name);
    public void Reset( );
    public PolicyStatement Resolve(Evidence evidence);
    public CodeGroup ResolveMatchingCodeGroups(Evidence evidence);
    public SecurityElement ToXml( );
}

```

This class represents a security policy level and provides the ability to access and manipulate the object: three key elements: fully trusted assemblies, named permission sets, and code group hierarchy.

The `PolicyLevel` class does not implement any constructors. `PolicyLevel` objects representing the current machine, and user policy levels are obtained through the `System.Security.SecurityManager.PolicyHierarchy` method. The `System.Security.SecurityManager.PolicyHierarchy()` method returns a `System.Collections.IEnumerator` containing `PolicyLevel` objects for each of the three policy levels. To obtain a specific `PolicyLevel`, step through the enumerator and test the value of `Label` property against the strings "Machine," and "User."

It is not possible to obtain a `PolicyLevel` representing an existing application domain's policy level, but `CreateAppDomainLevel()` method creates a new `PolicyLevel` suitable for assigning to an application domain. The `System.AppDomain.SetAppDomainPolicy()` method. The `System.AppDomain.SetAppDomainPolicy()` method can be called once; on subsequent calls a `PolicyException` is thrown.

The read-only `FullTrustAssemblies` property returns a `System.Collections.IList` containing a set of `StrongNameMembershipCondition` objects that represent the policy level's set of fully trusted assemblies. `AddFullTrustAssembly()` and `RemoveFullTrustAssembly()` methods allow for the administration of assemblies list.

The read-only `NamedPermissionSets` property also returns a `System.Collections.IList`. This list contains `System.Security.NamedPermissionSet` objects that represent the named permission sets defined in the `AddNamedPermissionSet()` and `RemoveNamedPermissionSet()` methods allow for the administration of permission set list. The `ChangeNamedPermissionSet()` method allows the contents of a specified name to be easily replaced with the contents of a `System.Security.PermissionSet` without the need to remove and re-add the set.

The `RootCodeGroup` property gets or sets the root `CodeGroup` of the policy level's code group tree. Further modification of the code group tree is possible only by using the methods of `CodeGroup` to traverse the tree. Assigning a new value to the `RootCodeGroup` property causes the old root `CodeGroup` to be replaced along with the existing tree. When the `RootCodeGroup` property is set, the `RootCodeGroup` property makes a *copy* of the new root `CodeGroup`, which means that any modifications to the code group tree through the original `CodeGroup` object are not reflected in the policy level's copy.

The `Reset()` method returns a policy level to its default "out-of-box" configuration, and the `Recover()` method returns the policy level to its previously saved state. Although it is not possible to invoke the runtime's policy reconfiguration programmatically, the `Resolve()` and `ResolveMatchingCodeGroups()` methods are useful for evaluating what a `PolicyLevel` would grant to an assembly or application domain.

Returned By

`System.Security.SecurityManager.{LoadPolicyLevelFromFile(), LoadPolicyLevelFromString()}`

Passed To

Multiple types

[Team LIB]

[Team LiB]

PolicyStatement

serializable

System.Security.Policy
(mscorlib.dll)*sealed
class*

```

public sealed class PolicyStatement : System.Security.ISecurityEncodable,
    System.Security.ISecurityPolicyEncodable {
// Public Constructors
    public PolicyStatement(System.Security.PermissionSet permSet);
    public PolicyStatement(System.Security.PermissionSet permSet, PolicyStatementAttribute
// Public Instance Properties
    public PolicyStatementAttribute Attributes{set; get; }
    public string AttributeString{get; }
    public PermissionSet PermissionSet{set; get; }
// Public Instance Methods
    public PolicyStatement Copy( );
    public void FromXml(System.Security.SecurityElement et);
// implements System.Security.ISecurityEncodable
    public void FromXml(System.Security.SecurityElement et, PolicyLevel level);
// implements System.Security.ISecurityPolicyEncodable
    public SecurityElement ToXml( );
// implements System.Security.ISecurityEncodable
    public SecurityElement ToXml(PolicyLevel level);
// implements System.Security.ISecurityPolicyEncodable
}

```

The `PolicyStatement` class defines the set of permissions that a code group grants to its members, as well as the attributes of the code group. Of the four concrete `CodeGroup` implementations, only the `UnionCodeGroup` and `FirstMatchCodeGroup` classes require a `PolicyStatement` object. The `FileCodeGroup` and `NetCodeGroup` generate their permission sets dynamically, and do not support attributes. A `PolicyStatement` is assigned to a `UnionCodeGroup` or `FirstMatchCodeGroup` object as an argument of the code group's constructor, or through the `CodeGroup.PolicyStatement` property.

A `PolicyStatement` object contains a `System.Security.PermissionSet` object that specifies the permissions granted to the code group's members and a value from the `PolicyStatementAttribute` enumeration that specifies the attribute assigned to the code group. Both values can be set as arguments of the `PolicyStatement` constructor or through construction through the `PermissionSet` and `Attributes` properties. The `AttributeString` property returns a readable representation of the value held in the `Attributes` property.

Returned By

```
CodeGroup.{PolicyStatement, Resolve( )}, PolicyLevel.Resolve( )
```

Passed To

```
CodeGroup.{CodeGroup( ), PolicyStatement} , FirstMatchCodeGroup.FirstMatchCodeGroup( ) ,  
UnionCodeGroup.UnionCodeGroup( )
```

[Team LiB]

[Team LiB]

PolicyStatementAttribute serializable, flag

System.Security.Policy (mscorlib.dll) *enum*

```
public enum PolicyStatementAttribute {
    Nothing = 0x00000000, Exclusive = 0x00000001, LevelFinal = 0x00000002, All = 0x00000003
}
```

The `PolicyStatementAttribute` enumeration is used in conjunction with the `PolicyStatement` class. `PolicyStatementAttribute` contains values that are assigned to the `PolicyStatement.Attributes` property to define a code group's attributes. Code group attributes affect the policy resolution process by controlling how the permissions granted by one code group are merged with the permissions granted by other code groups.

The `Exclusive` value specifies that the permissions granted by the code group are the maximum set of permissions that a group member can receive from the current policy level. Because the permissions granted by each policy level intersect to determine the final grant set, a code group with the `Exclusive` attribute actually defines the maximum set of permissions in the final grant set. An assembly or application domain can be a member of at most one `Exclusive` code group per policy level, or the runtime throws a `PolicyException`.

Policy levels are resolved in the order enterprise, machine, user, and finally application domain. The `LevelFinal` value specifies that no policy levels below the current level will be evaluated. If resolving the permissions for an assembly, the runtime always evaluates the application domain policy regardless of whether the assembly is the member of a `LevelFinal` code group.

`PolicyStatementAttribute` values can be combined using a logical OR operator, but the enumeration also provides the `All` value, which represents both the `Exclusive` and `LevelFinal` attributes. `Nothing` means that no attributes are set and is the default state for a `PolicyStatement`.

Hierarchy

```
System.Object      System.ValueType      System.Enum(System.IComparable,
System.IFormattable, System.IConvertible)      PolicyStatementAttribute
```

Returned By

```
PolicyStatement.Attributes
```

Passed To

```
PolicyStatement.{Attributes, PolicyStatement( )}
```

[Team LiB]

[Team LiB]

Publisher

serializable

System.Security.Policy
(mscorlib.dll)*sealed
class*

```

public sealed class Publisher : IIdentityPermissionFactory, IBuiltInEvidence {
// Public Constructors
    public Publisher(System.Security.Cryptography.X509Certificates.X509Certificate cert)
// Public Instance Properties
    public X509Certificate Certificate{get; }
// Public Instance Methods
    public object Copy( );
    public IPermission CreateIdentityPermission(Evidence evidence);
// implements IIdentityPermissionFactory
    public override bool Equals(object o);
// overrides object
    public override int GetHashCode( );
// overrides object
    public override string ToString( );
// overrides object
}

```

This evidence class represents the Authenticode X.509v3 digital signature used to sign an assembly. The digital signature is passed as a `System.Security.Cryptography.X509Certificates.X509Certificate` argument to the `Publisher` constructor and is accessible after creation through the `Certificate` property.

The `PublisherMembershipCondition` class uses `Publisher` evidence during policy resolution to evaluate whether an assembly or application domain qualifies for membership of a `CodeGroup`.

Because the `Publisher` class implements the `IIdentityPermissionFactory` interface, an assembly or application domain with `Publisher` evidence will contain an appropriately configured `System.Security.Permissions.PublisherIdentityPermission` object in its grant set.

[Team LiB]

[Team LiB]

PublisherMembershipCondition serializable

System.Security.Policy
(mscorlib.dll)

sealed
class

```
public sealed class PublisherMembershipCondition : IMembershipCondition, System.Security.
    System.Security.ISecurityPolicyEncodable, IConstantMembershipCondition {
// Public Constructors
    public PublisherMembershipCondition(System.Security.Cryptography.X509Certificates.X509Certificate certificate);
// Public Instance Properties
    public X509Certificate Certificate{set; get; }
// Public Instance Methods
    public bool Check(Evidence evidence);
// implements IMembershipCondition
    public IMembershipCondition Copy( );
// implements IMembershipCondition
    public override bool Equals(object o);
// overrides object
    public void FromXml(System.Security.SecurityElement e);
// implements System.Security.ISecurityEncodable
    public void FromXml(System.Security.SecurityElement e, PolicyLevel level);
// implements System.Security.ISecurityPolicyEncodable
    public override int GetHashCode( );
// overrides object
    public override string ToString( );
// overrides object
    public SecurityElement ToXml( );
// implements System.Security.ISecurityEncodable
    public SecurityElement ToXml(PolicyLevel level);
// implements System.Security.ISecurityPolicyEncodable
}
```

The `PublisherMembershipCondition` class uses the X.509v3 certificate contained in `Publisher` evidence application domain qualifies for membership to a `CodeGroup`. The `Certificate` property contains the `System.Security.Cryptography.X509Certificates.X509Certificate` object against which the `Publisher` object tests.

When policy resolution calls the `Check()` method, the `PublisherMembershipCondition` tests the `Evidence` to ensure that it contains a `Publisher` object. If so, the `PublisherMembershipCondition` object compares the `Publisher.Certificate` property to determine if it grants `CodeGroup` membership.

[Team LiB]

[\[Team LiB \]](#)

Site

serializable

System.Security.Policy
(mscorlib.dll)

sealed
class

```
public sealed class Site : IIdentityPermissionFactory, IBuiltInEvidence {
// Public Constructors
    public Site(string name);
// Public Instance Properties
    public string Name {get; }
// Public Static Methods
    public static Site CreateFromUrl(string url);
// Public Instance Methods
    public object Copy( );
    public IPermission CreateIdentityPermission(Evidence evidence);
// implements IIdentityPermissionFactory
    public override bool Equals(object o);
// overrides object
    public override int GetHashCode( );
// overrides object
    public override string ToString( );
// overrides object
}
```

The `Site` evidence class specifies the fully qualified host name from where an assembly was loaded-for example, *www.oreilly.com*. The site name is passed as a `System.String` to the `Site` constructor, and is accessible through the `Name` property. The static `CreateFromUrl()` method enables easy creation of `Site` objects from URL strings.

During policy resolution, the `SiteMembershipCondition` class uses `Site` evidence to test whether an assembly or application domain qualifies for membership of a `CodeGroup`.

Because the `Site` class implements the `IIdentityPermissionFactory` interface, an assembly or application domain with `Site` evidence will contain an appropriately configured `System.Security.Permissions.SiteIdentityPermission` object in its grant set.

[\[Team LiB \]](#)

[Team LiB]

SiteMembershipCondition

serializable

System.Security.Policy
(mscorlib.dll)*sealed
class*

```

public sealed class SiteMembershipCondition : IMembershipCondition,
    System.Security.ISecurityEncodable, System.Security.ISecurityPolicyEncodable, ICor
// Public Constructors
    public SiteMembershipCondition(string site);
// Public Instance Properties
    public string Site{set; get; }
// Public Instance Methods
    public bool Check(Evidence evidence);
// implements IMembershipCondition
    public IMembershipCondition Copy( );
// implements IMembershipCondition
    public override bool Equals(object o);
// overrides object
    public void FromXml(System.Security.SecurityElement e);
// implements System.Security.ISecurityEncodable
    public void FromXml(System.Security.SecurityElement e, PolicyLevel level);
// implements System.Security.ISecurityPolicyEncodable
    public override int GetHashCode( );
// overrides object
    public override string ToString( );
// overrides object
    public SecurityElement ToXml( );
// implements System.Security.ISecurityEncodable
    public SecurityElement ToXml(PolicyLevel level);
// implements System.Security.ISecurityPolicyEncodable
}

```

This membership condition class relies on the value of `Site` evidence to determine if an assembly or app membership to a `CodeGroup`. The `Site` property contains a `System.String` that identifies the site name is based. The site name can include wildcards-for example, `*.oreilly.com`, which will match any site in t

When policy resolution calls the `Check()` method, the `SiteMembershipCondition` object tests the speci ensure it contains `Site` evidence. If so, `SiteMembershipCondition` determines `CodeGroup` membership contained in its `Site` property with the site name contained in the `Site.Name` property.

[Team LiB]

[\[Team LiB \]](#)

StrongName

serializable

System.Security.Policy
(mscorlib.dll)*sealed
class*

```
public sealed class StrongName : IIdentityPermissionFactory, IBuiltInEvidence {
// Public Constructors
    public StrongName(System.Security.Permissions.StrongNamePublicKeyBlob blob,
        string name, Version version);
// Public Instance Properties
    public string Name {get; }
    public StrongNamePublicKeyBlob PublicKey {get; }
    public Version Version {get; }
// Public Instance Methods
    public object Copy( );
    public IPermission CreateIdentityPermission(Evidence evidence);
// implements IIdentityPermissionFactory
    public override bool Equals(object o);
// overrides object
    public override int GetHashCode( );
// overrides object
    public override string ToString( );
// overrides object
}
```

The `StrongName` evidence class represents an assembly's strong name. The `PublicKey` property contains a `System.Security.Permissions.StrongNamePublicKeyBlob` object that contains the public key of the assembly's strong name. The `Name` property holds the simple name of the assembly, and the `Version` property contains a `System.Version` object that represents the assembly's version.

The `StrongNameMembershipCondition` class uses `StrongName` evidence during policy resolution to evaluate whether an assembly or application domain qualifies for membership of a `CodeGroup`.

Because the `StrongName` class implements the `IIdentityPermissionFactory` interface, an assembly or application domain with `StrongName` evidence will contain an appropriately configured `System.Security.Permissions.StrongNameIdentityPermission` object in its grant set.

Passed To

```
PolicyLevel.{AddFullTrustAssembly( ), RemoveFullTrustAssembly( )}
```

[\[Team LiB \]](#)

[Team LiB]

StrongNameMembershipCondition serializable

System.Security.Policy
(mscorlib.dll)

sealed
class

```
public sealed class StrongNameMembershipCondition : IMembershipCondition, System.Security.Policy.ISecurityPolicyEncodable, IConstantMembershipCondition {
    // Public Constructors
    public StrongNameMembershipCondition(System.Security.Permissions.StrongNamePublicKeyBlob publicKey, string name, Version version);
    // Public Instance Properties
    public string Name {set; get; }
    public StrongNamePublicKeyBlob PublicKey {set; get; }
    public Version Version {set; get; }
    // Public Instance Methods
    public bool Check(Evidence evidence);
    // implements IMembershipCondition
    public IMembershipCondition Copy( );
    // implements IMembershipCondition
    public override bool Equals(object o);
    // overrides object
    public void FromXml(System.Security.SecurityElement e);
    // implements System.Security.ISecurityEncodable
    public void FromXml(System.Security.SecurityElement e, PolicyLevel level);
    // implements System.Security.ISecurityPolicyEncodable
    public override int GetHashCode( );
    // overrides object
    public override string ToString( );
    // overrides object
    public SecurityElement ToXml( );
    // implements System.Security.ISecurityEncodable
    public SecurityElement ToXml(PolicyLevel level);
    // implements System.Security.ISecurityPolicyEncodable
}
```

This membership condition class uses `StrongName` evidence to determine if an assembly or application demands membership of a `CodeGroup`. The `PublicKey` property contains a `System.Security.Permissions.StrongNamePublicKeyBlob` object that represents the public key information to base the membership test on. The `Name` and `Version` properties allow the `StrongNameMembershipCondition` to perform a more specific and accurate test for membership.

When policy resolution calls the `Check()` method, if the provided `Evidence` collection contains a `StrongNameMembershipCondition`, it compares its `PublicKey`, `Name`, and `Version` properties with the equivalent `StrongName` object. If the values match, then membership of the `CodeGroup` is granted.

`StrongNameMembershipCondition` objects are also used in the `PolicyLevel` class to identify a policy level for assemblies.

Passed To

```
PolicyLevel.{AddFullTrustAssembly( ), RemoveFullTrustAssembly( )}
```

[Team LiB]

[Team LiB]

UnionCodeGroup

serializable

System.Security.Policy
(mscorlib.dll)*sealed*
class

```

public sealed class UnionCodeGroup : CodeGroup {
// Public Constructors
    public UnionCodeGroup(IMembershipCondition membershipCondition, PolicyStatement polic
// Public Instance Properties
    public override string MergeLogic{get; }
// overrides CodeGroup
// Public Instance Methods
    public override CodeGroup Copy( );
// overrides CodeGroup
    public override PolicyStatement Resolve(Evidence evidence);
// overrides CodeGroup
    public override CodeGroup ResolveMatchingCodeGroups(Evidence evidence);
// overrides CodeGroup
}

```

The `UnionCodeGroup` class is a subclass of `CodeGroup` and is the most commonly used type of code group. Assemblies and application domains that qualify for membership are compared for membership against children of the `UnionCodeGroup`, and so on down through the code group tree. The final permission set granted by the `UnionCodeGroup` is the union of its permission set, and the permissions granted by all children code groups of which the assembly or application domain is also a member.

Hierarchy

System.Object CodeGroup UnionCodeGroup

[Team LiB]

[\[Team LiB \]](#)

Url

serializable

System.Security.Policy
(mscorlib.dll)*sealed
class*

```

public sealed class Url : IIdentityPermissionFactory, IBuiltInEvidence {
// Public Constructors
    public Url(string name);
// Public Instance Properties
    public string Value{get; }
// Public Instance Methods
    public object Copy( );
    public IPermission CreateIdentityPermission(Evidence evidence);
// implements IIdentityPermissionFactory
    public override bool Equals(object o);
// overrides object
    public override int GetHashCode( );
// overrides object
    public override string ToString( );
// overrides object
}

```

The `Url` evidence class identifies the URL from where an assembly was loaded—for example, `https://www.oreilly.com:8080/SomeFile.exe`. The source URL is passed as a `System.String` argument to the `Url` constructor and is accessible later through the `Value` property.

During policy resolution, the `UrlMembershipCondition` class uses `Url` evidence to determine if code qualifies for membership of a code group. The `ApplicationDirectoryMembershipCondition` class also uses `Url` evidence in conjunction with `ApplicationDirectory` evidence to determine code group membership.

Because the `Url` class implements the `IIdentityPermissionFactory` interface, an assembly or application domain with `Url` evidence will contain an appropriately configured `System.Security.Permissions.UrlIdentityPermission` object in its grant set.

[\[Team LiB \]](#)

[Team LiB]

UrlMembershipCondition

serializable

System.Security.Policy
(mscorlib.dll)*sealed
class*

```

public sealed class UrlMembershipCondition : IMembershipCondition, System.Security.ISecurityPolicyEncodable, IConstantMembershipCondition {
    // Public Constructors
    public UrlMembershipCondition(string url);
    // Public Instance Properties
    public string Url{set; get; }
    // Public Instance Methods
    public bool Check(Evidence evidence);
    // implements IMembershipCondition
    public IMembershipCondition Copy( );
    // implements IMembershipCondition
    public override bool Equals(object o);
    // overrides object
    public void FromXml(System.Security.SecurityElement e);
    // implements System.Security.ISecurityEncodable
    public void FromXml(System.Security.SecurityElement e, PolicyLevel level);
    // implements System.Security.ISecurityPolicyEncodable
    public override int GetHashCode( );
    // overrides object
    public override string ToString( );
    // overrides object
    public SecurityElement ToXml( );
    // implements System.Security.ISecurityEncodable
    public SecurityElement ToXml(PolicyLevel level);
    // implements System.Security.ISecurityPolicyEncodable
}

```

The `UrlMembershipCondition` class uses `Url` evidence to determine if an assembly or application domain membership of a `CodeGroup`. The `Url` property contains the URL required to qualify for membership and wildcard as the final element-for example, `http://www.oreilly.com/apps/*`. The value of the `Url` property is passed to the `UrlMembershipCondition` constructor.

During policy resolution, the `Check()` method enumerates the provided `Evidence` collection and determines if there is any `Url` evidence object. If so, `UrlMembershipCondition` tests for `CodeGroup` membership by comparing the `Url` property with the URL contained in the `Url.Value` property.

[Team LiB]

[\[Team LiB \]](#)

Zone

serializable

System.Security.Policy
(mscorlib.dll)*sealed
class*

```

public sealed class Zone : IIdentityPermissionFactory, IBuiltInEvidence {
// Public Constructors
    public Zone(System.Security.SecurityZone zone);
// Public Instance Properties
    public SecurityZone SecurityZone {get; }
// Public Static Methods
    public static Zone CreateFromUrl(string url);
// Public Instance Methods
    public object Copy( );
    public IPermission CreateIdentityPermission(Evidence evidence);
// implements IIdentityPermissionFactory
    public override bool Equals(object o);
// overrides object
    public override int GetHashCode( );
// overrides object
    public override string ToString( );
// overrides object
}

```

This evidence class identifies the Internet Explorer security zone from where an assembly was loaded. The runtime assigns `Zone` evidence to an assembly by comparing the URL from where the assembly was loaded, with the security zone configuration in Internet Explorer. The type of `Zone` evidence is specified by passing a value of the `System.Security.SecurityZone` enumeration to the `Zone` constructor and can be accessed after construction through the `SecurityZone` property.

During policy resolution, the `ZoneMembershipCondition` class uses `Zone` evidence to evaluate whether code qualifies for membership of a `CodeGroup`.

Because the `Zone` class implements the `IIdentityPermissionFactory` interface, an assembly or application domain with `Zone` evidence will contain an appropriately configured `System.Security.Permissions.ZoneIdentityPermission` object in its grant set.

[\[Team LiB \]](#)

[Team LiB]

ZoneMembershipCondition

serializable

System.Security.Policy
(mscorlib.dll)*sealed*
class

```

public sealed class ZoneMembershipCondition : IMembershipCondition, System.Security.ISec
    System.Security.ISecurityPolicyEncodable, IConstantMembershipCondition {
// Public Constructors
    public ZoneMembershipCondition(System.Security.SecurityZone zone);
// Public Instance Properties
    public SecurityZone SecurityZone{set; get; }
// Public Instance Methods
    public bool Check(Evidence evidence);
// implements IMembershipCondition
    public IMembershipCondition Copy( );
// implements IMembershipCondition
    public override bool Equals(object o);
// overrides object
    public void FromXml(System.Security.SecurityElement e);
// implements System.Security.ISecurityEncodable
    public void FromXml(System.Security.SecurityElement e, PolicyLevel level);
// implements System.Security.ISecurityPolicyEncodable
    public override int GetHashCode( );
// overrides object
    public override string ToString( );
// overrides object
    public SecurityElement ToXml( );
// implements System.Security.ISecurityEncodable
    public SecurityElement ToXml(PolicyLevel level);
// implements System.Security.ISecurityPolicyEncodable
}

```

The `ZoneMembershipCondition` class uses the value of `Zone` evidence to determine if an assembly or application qualifies for membership of a `CodeGroup`. The `SecurityZone` property contains a member of the `System.Security.SecurityZone` enumeration that identifies the security zone required for membership. `SecurityZone` is set in the `ZoneMembershipCondition` constructor.

When policy resolution calls the `Check()` method, the `ZoneMembershipCondition` enumerates the provided collection to determine if it contains `Zone` evidence. If so, the `ZoneMembershipCondition` compares the value of the `SecurityZone` property with the value of the `Zone.SecurityZone` property. If the two values match, the application domain is granted membership of the `CodeGroup`.

[Team LiB]

[[Team LiB](#)]

Chapter 29. The System.Security.Principal Namespace

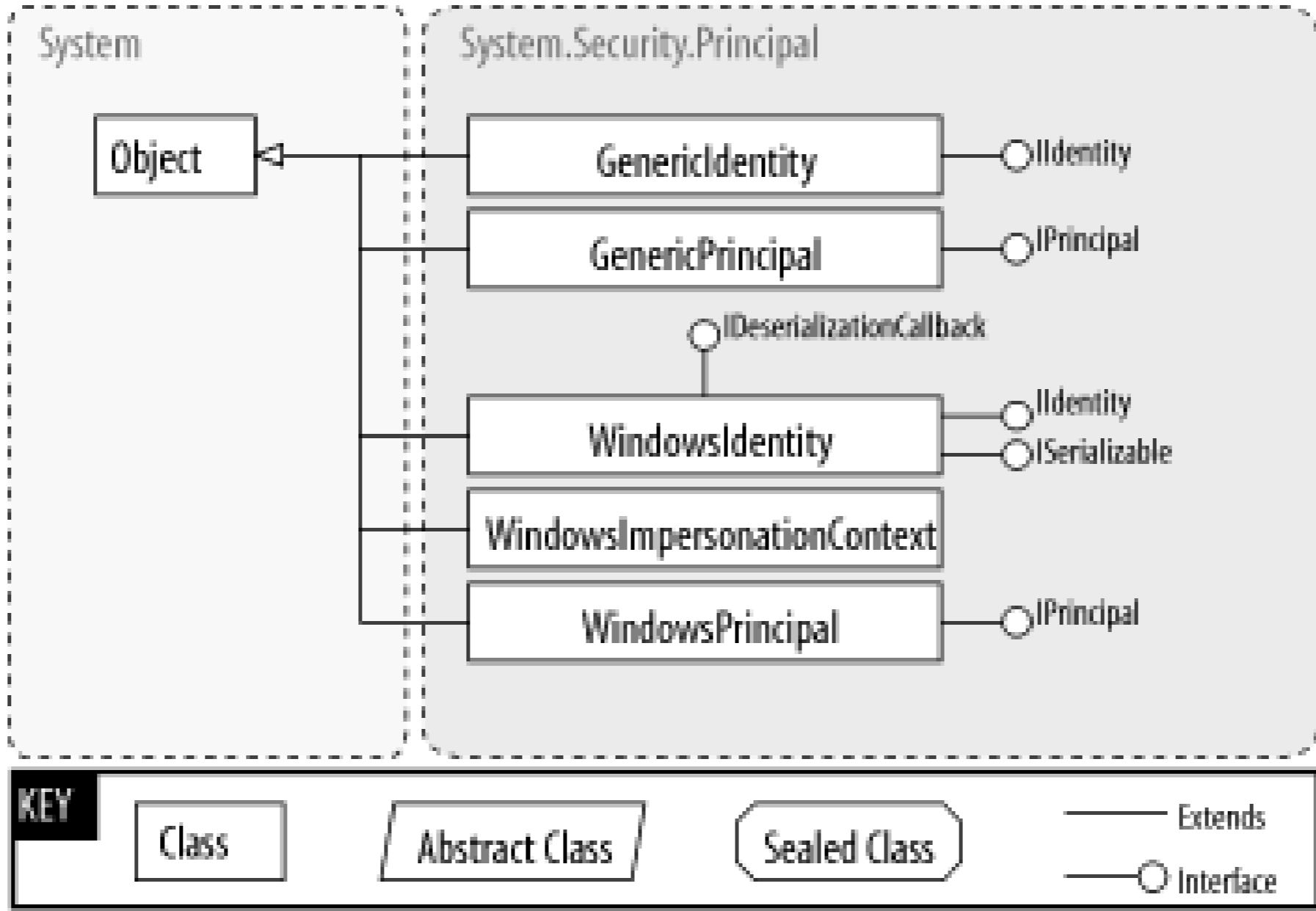
The .NET runtime provides a simple role-based security mechanism that enables code to make security decisions based on the user that is running the code, and the roles to which the user belongs. .NET's role-based security model is independent of any underlying authentication and authorization mechanism, and relies on two key abstractions that represent the user and their roles: identities and principals. An identity represents an authenticated user, and the principal is a container that holds both the identity and the set of roles to which the identity belongs. Principals are assigned to threads and provide the information necessary for the runtime to authorize and control the actions of the current user.

The `System.Security.Principal` namespace contains the interfaces that define the functionality of identities and principals, and includes two concrete role-based security implementations. The first implementation consists of the classes named with the prefix "Generic." The generic role-based security implementation is simple and requires direct manipulation to configure identities and principals, but can be used in conjunction with any user authentication and authorization mechanism. The second implementation consists of the classes named with the prefix "Windows." The Windows role-based security implementation integrates with the Windows user accounts mechanism and allows code to base security decisions on Windows user accounts, and the user groups to which they belong.

Role-based security is enforced using security demands similar to those used to enforce code-access security. The `System.Security.Permissions.PrincipalPermission` class provides support for imperative role-based security demands, and the `System.Security.Permissions.PrincipalPermissionAttribute` class provides support for declarative demands. When code invokes a role-based security demand, the runtime evaluates the principal of the current thread to ensure that it has the demanded identity and roles. If not, the demand raises a `System.Security.SecurityException`. Role-based security demands do not cause stack walks.

[Figure 29-1](#) shows the types in this namespace.

Figure 29-1. The System.Security.Principal namespace



[Team LiB]

[\[Team LiB \]](#)

GenericIdentity

serializable

System.Security.Principal (mscorlib.dll) class

```
public class GenericIdentity : IIdentity {  
    // Public Constructors  
    public GenericIdentity(string name);  
    public GenericIdentity(string name, string type);  
    // Public Instance Properties  
    public virtual string AuthenticationType{get; }  
    // implements IIdentity  
    public virtual bool IsAuthenticated{get; }  
    // implements IIdentity  
    public virtual string Name{get; }  
    // implements IIdentity  
}
```

This implementation of the `IIdentity` interface provides a generic representation of a user. `GenericIdentity` objects are created using standard constructors that take either the user's name or the user's name and a string representing the authentication mechanism used to authenticate the user. The ability to create `GenericIdentity` objects with any username means that the `GenericIdentity` class can be used to represent users authenticated against any authority.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

GenericPrincipal

serializable

System.Security.Principal (mscorlib.dll) *class*

```
public class GenericPrincipal : IPrincipal {  
    // Public Constructors  
    public GenericPrincipal(IIdentity identity, string[ ] roles);  
    // Public Instance Properties  
    public virtual IIdentity Identity{get; }  
    // implements IPrincipal  
    // Public Instance Methods  
    public virtual bool IsInRole(string role);  
    // implements IPrincipal  
}
```

The `GenericPrincipal` class provides a generic implementation of the `IPrincipal` interface. The `GenericPrincipal` constructor takes an `IIdentity` object representing the principal's user and a `System.String` array containing the names of the roles to which the user belongs. The specified `IIdentity` is normally an instance of `GenericIdentity`, but this is not mandatory. Used in conjunction with the `GenericIdentity` class, `GenericPrincipal` provides a flexible principal implementation that can be used with any authentication and authorization mechanism.

The `IsInRole()` method takes the name of a role and returns true if it was included in the array of role names passed to the `GenericPrincipal` constructor. The role name comparison is case-insensitive.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

IIdentity

System.Security.Principal
(mscorlib.dll)

interface

```
public interface IIdentity {  
    // Public Instance Properties  
    public string AuthenticationType {get; }  
    public bool IsAuthenticated {get; }  
    public string Name {get; }  
}
```

The `IIdentity` interface defines the functionality that all identity classes must implement. An identity object represents the user on whose behalf code is running. `IIdentity` defines three read-only properties. `Name` is the most commonly used property and returns the name of the user. `IsAuthenticated` returns a Boolean value indicating whether the current identity has been authenticated. `AuthenticationType` returns a `System.String` that identifies the authentication mechanism used to authenticate the user.

Implemented By

`GenericIdentity`, `WindowsIdentity`

Returned By

`GenericPrincipal.Identity`, `IPrincipal.Identity`, `WindowsPrincipal.Identity`

Passed To

`GenericPrincipal.GenericPrincipal()`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

I Principal

System.Security.Principal
(mscorlib.dll)

interface

```
public interface IPrincipal {  
    // Public Instance Properties  
    public IIdentity Identity { get; }  
    // Public Instance Methods  
    public bool IsInRole(string role);  
}
```

This interface defines the functionality that all principal classes must implement. An `IPrincipal` object is associated with a thread running in the .NET runtime to represent the user on whose behalf the managed code is running. An `IPrincipal` object is a container for an `IIdentity`, which represents the active user, and a set of role names of which the user is a member.

The read-only `Identity` property returns the `IIdentity` object contained in the `IPrincipal`. The full set of roles for the contained identity is not accessible. It is only possible to test the set of roles to see if it contains a specific value using the `IsInRole()` method.

Implemented By

`GenericPrincipal`, `WindowsPrincipal`

Returned By

`System.Threading.Thread.CurrentPrincipal`

Passed To

`System.AppDomain.SetThreadPrincipal()`, `System.Threading.Thread.CurrentPrincipal`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

PrincipalPolicy

serializable

System.Security.Principal (mscorlib.dll) *enum*

```
public enum PrincipalPolicy {
    UnauthenticatedPrincipal = 0, NoPrincipal = 1, WindowsPrincipal = 2
}
```

The values of this enumeration are used in the `System.AppDomain.SetPrincipalPolicy()` method. `SetPrincipalPolicy()` specifies how default `IIdentity` and `IPrincipal` objects are created for threads running in the application domain if an `IPrincipal` has not been assigned to the thread by the program or host.

The value `NoPrincipal` means that the runtime does not create default identity or principal objects for a thread. `UnauthenticatedPrincipal` means that the runtime will assign threads a `GenericPrincipal` containing an empty role set and a `GenericIdentity` that represents an unnamed user. The `WindowsPrincipal` value causes the runtime to create a `WindowsPrincipal` object and assign it to the thread. The `WindowsPrincipal` object contains a `WindowsIdentity` that represents the currently active Windows user account and a set of roles that represent the Windows user groups to which the user belongs. `UnauthenticatedPrincipal` is the default value.

Hierarchy

```
System.Object      System.ValueType      System.Enum(System.IComparable,
System.IFormattable, System.IConvertible)      PrincipalPolicy
```

Passed To

```
System.AppDomain.SetPrincipalPolicy( )
```

[\[Team LiB \]](#)

[\[Team LiB \]](#)

WindowsAccountType

serializable

System.Security.Principal (mscorlib.dll) *enum*

```
public enum WindowsAccountType {  
    Normal = 0, Guest = 1, System = 2, Anonymous = 3  
}
```

The values of the `WindowsAccountType` enumeration are used in `WindowsIdentity` constructors to specify the type of Windows account represented by the new `WindowsIdentity` object.

Hierarchy

`System.Object` → `System.ValueType` `System.Enum(System.IComparable,
System.IFormattable, System.IConvertible)` `WindowsAccountType`

Passed To

`WindowsIdentity.WindowsIdentity()`

[\[Team LiB \]](#)

[Team LiB]

WindowsBuiltInRole

serializable

System.Security.Principal (mscorlib.dll) *enum*

```
public enum WindowsBuiltInRole {
    Administrator = 544, User = 545, Guest = 546, PowerUser = 547, AccountOperator = 548,
    SystemOperator = 549, PrintOperator = 550, BackupOperator = 551, Replicator = 552
}
```

The `WindowsBuiltInRole` enumeration contains values used in one of the overloaded `WindowsPrincipal.IsInRole()` methods. Its values represent commonly used Windows user groups that are present in most Windows NT, Windows 2000, and Windows XP installations, and avoid the need to specify string values for the group names. It is apparent from the names of the member values which Windows group they represent, with the exception of `SystemOperator`, which represents the Windows Server Operators group.

Hierarchy

```
System.Object      System.ValueType      System.Enum(System.IComparable,
System.IFormattable, System.IConvertible)      WindowsBuiltInRole
```

Passed To

```
WindowsPrincipal.IsInRole( )
```

[Team LiB]

[Team LiB]

WindowsIdentity

serializable

System.Security.Principal (mscorlib.dll) *class*

```

public class WindowsIdentity : IIdentity, System.Runtime.Serialization.ISerializable,
    System.Runtime.Serialization.IDeserializationCallback {
// Public Constructors
    public WindowsIdentity(IntPtr userToken);
    public WindowsIdentity(IntPtr userToken, string type);
    public WindowsIdentity(IntPtr userToken, string type, WindowsAccountType acctType);
    public WindowsIdentity(IntPtr userToken, string type, WindowsAccountType acctType, b
    public WindowsIdentity(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context);
    public WindowsIdentity(string sUserPrincipalName);
    public WindowsIdentity(string sUserPrincipalName, string type);
// Public Instance Properties
    public virtual string AuthenticationType{get; }
// implements IIdentity
    public virtual bool IsAnonymous{get; }
    public virtual bool IsAuthenticated{get; }
// implements IIdentity
    public virtual bool IsGuest{get; }
    public virtual bool IsSystem{get; }
    public virtual string Name{get; }
// implements IIdentity
    public virtual IntPtr Token{get; }
// Public Static Methods
    public static WindowsIdentity GetAnonymous( );
    public static WindowsIdentity GetCurrent( );
    public static WindowsImpersonationContext Impersonate(IntPtr userToken);
// Public Instance Methods
    public virtual WindowsImpersonationContext Impersonate( );
// Protected Instance Methods
    protected override void Finalize( );
// overrides object
}

```

The `WindowsIdentity` class provides an `IIdentity` implementation that represents a Windows user. Because `WindowsIdentity` is Windows-specific, it implements members useful for working with Windows user accounts in addition to the functionality defined by `IIdentity`.

The static `GetCurrent()` method returns a `WindowsIdentity` object representing the currently logged-on user. The static `GetAnonymous()` method returns a `WindowsIdentity` object representing an anonymous Windows user. `WindowsIdentity` constructors support creation of `WindowsIdentity` objects that represent users other than the current user. Each constructor requires a Windows access token representing the desired user. A handle to the Windows access token is passed to the constructor wrapped in a `System.IntPtr` object. The Windows access token is usually obtained th

unmanaged code, such as the `LogonUser()` method of the `advapi32.dll/Win32` library. The access token `WindowsIdentity` is available through its `Token` property. Starting with .NET 1.1, however, it is possible to create a `WindowsIdentity` using a string argument containing the user Principal Name of the user, such as the `v` to the `LogonUser()` API. However, this functionality is only available on Windows 2003 Server or later platform, it will throw an `ArgumentException` .

Calling the `Impersonate()` method changes the Windows access token of the current thread to that of the `WindowsIdentity` object. By creating a `WindowsIdentity` that represents a user other than the active user, `Impersonate()` allows code to perform operating system-level impersonation. The `Impersonate()` method returns a `WindowsImpersonationContext` object representing the Windows access token prior to impersonation; in order to revert to the original access token once impersonation is no longer required.

The `IsAnonymous` , `IsGuest` , and `IsSystem` properties provide an easy-to-use mechanism for determining if a `WindowsIdentity` object represents an anonymous, guest, or system Windows user account. Determining if a `WindowsIdentity` object represents a process of elimination; there is no `IsNormal` property.

The `Name` property of a `WindowsIdentity` object will return a name in the form `DOMAINNAME\USERNAME` where `DOMAINNAME` specifies the authority used to validate the user; for example, `COMPANY_X\Gary` or `MY_MACHINE\Peter`.

Passed To

```
WindowsPrincipal.WindowsPrincipal( )
```

[Team LiB]

[\[Team LiB \]](#)

WindowsImpersonationContext

System.Security.Principal (mscorlib.dll) *class*

```
public class WindowsImpersonationContext {  
    // Public Instance Methods  
    public void Undo( );  
    // Protected Instance Methods  
    protected override void Finalize( );  
    // overrides object  
}
```

The `WindowsImpersonationContext` class represents the active Windows user prior to code impersonating another Windows user through a call to the `WindowsIdentity.Impersonate()` method. The `WindowsIdentity.Impersonate()` method returns a `WindowsImpersonationContext` object. By calling the `Undo()` method of this object, the thread access token reverts to the one contained in the `WindowsImpersonationContext` object.

Returned By

`WindowsIdentity.Impersonate()`

[\[Team LiB \]](#)

[\[Team LiB \]](#)

WindowsPrincipal

serializable

System.Security.Principal (mscorlib.dll) class

```
public class WindowsPrincipal : IPrincipal {
    // Public Constructors
    public WindowsPrincipal(WindowsIdentity ntIdentity);
    // Public Instance Properties
    public virtual IIdentity Identity{get; }
    // implements IPrincipal
    // Public Instance Methods
    public virtual bool IsInRole(int rid);
    public virtual bool IsInRole(string role);
    // implements IPrincipal
    public virtual bool IsInRole(WindowsBuiltInRole role);
}
```

The `WindowsPrincipal` class provides a Windows-specific implementation of the `IPrincipal` interface that contains a `WindowsIdentity` object representing a Windows user. The `WindowsPrincipal` constructor takes only a `WindowsIdentity` object; the roles to which the user belongs are determined from the user's Windows access token.

`WindowsPrincipal` allows code to check the Windows groups to which the represented user belongs through its implementation of the `IPrincipal.IsInRole()` method, which performs a case-insensitive comparison of the user's Windows groups with the specified role name. `WindowsPrincipal` also includes two additional overloads of the `IsInRole()` method. The first takes an integer specifying the Windows RID for the role. The second overload takes a member of the `WindowsBuiltInRole` enumeration.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of *Programming .NET Security* is a Smoothhound shark. The Smoothhound shark is a harmless bottom-dwelling reef shark, which is also known as a mud shark or dogfish. It is a moderately sized shark with two high dorsal fins and short blunt teeth. It also has small grinding plates, which are used to crush shellfish. Besides shellfish, the Smoothhound shark forages reefs for worms, octopuses, and small fish.

The Smoothhound shark can be found in both inshore and offshore habitats, often over tidal flats and in shallow bays. Also, the Smoothhound shark is viviparous, which means it gives birth to live young; inside the female body, pups hatch from eggs and remain there until they are fully developed.

Sarah Sherman was the production editor and copyeditor, and Matt Hutchinson was the proofreader for *Programming .NET Security*. Claire Cloutier provided quality control. Johnna VanHoose Dinse and Tom Dinse wrote the index.

Ellie Volckhausen designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout. This book was converted by Andrew Savikas to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. The tip and warning icons were drawn by Christopher Bing. This colophon was written by Sarah Sherman.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Madeleine Newell) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, and Jeff Liggett.

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[-Rc argument](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[abstract cipher function classes, ElGamal algorithm](#)

[abstract classes](#)

[ElGamal algorithm](#)

[hashing algorithms](#)

[XTEA](#)

[access modifiers, C# and Visual Basic](#)

[access tokens, Windows security](#)

[activation records](#)

[AddFullTrustAssembly method](#)

[AddNamedPermissionSet method](#)

[Adler32 algorithm](#)

[administration, shared assemblies and](#)

[al.exe](#)

[algorithm classes, cryptographic key creation](#)

[algorithms](#) [See also cipher functions]

[asymmetric](#)

[security of](#)

[asymmetric encryption](#)

[ElGamal](#)

[abstract cipher function class definition](#)

[abstract class definition](#)

[BigInteger class](#)

[decryption class definition](#)

[encryption and decryption functions](#)

[encryption class definition](#)

[implementation class definition](#)

[key generation protocol](#)

[ElGamal, testing](#)

[encryption](#)

[.NET Framework, symmetric encryption](#)

[RSA](#)

[encrypted data creation](#)

[key length](#)

[signing algorithm, XMLDSIG](#)

[symmetric](#)

[cipher modes](#)

[instantiating](#)

[Key and IV properties](#)

[KeySize and BlockSize properties](#)

[symmetric algorithms, cryptographic keys](#)

[XTEA \(Extended Tiny Encryption Algorithm\)](#) [2nd](#)

[abstract class definition](#)

[ICryptoTransform interface](#)

[implementation class definition](#)

[XTEAManagedDecryptor class](#)

[XTEAManagedEncryptor class](#)

[AllowPartiallyTrustedCallers attribute](#)

[Anakrino/Exemplar decompiler](#)
[anonymous access, IIS authentication](#)
[APIs \(application programming interfaces\), Windows Crypto API](#)
[AppDomain class, evidence](#)
[application domain policy level](#)
application domains
 [application isolation and](#)
 [assembly loading, evidence](#)
 [assembly management and](#)
 [CAS and](#)
 [CLR and](#)
 [configuration](#)
 [evidence](#)
 [evidence assigning](#)
 [identity and](#)
 [isolated storage](#)
 [isolated storage and](#)
 [overview](#)
 [Publisher evidence](#)
 [restricted resources, trust levels and](#)
 [runtime security and](#)
 [security policy and](#)
application isolation
 [applications domains and](#)
 [.NET Framework](#)
[Application log \(ELS\)](#)
[application proxies, COM+](#)
[application resources \(assemblies\)](#)
[application roles \(COM+\), populating](#)
[application types \(COM+\), specifying](#)
[ApplicationAccessControl attribute \(COM+\)](#)
[ApplicationDirectory class](#)
applications
 COM+
 [installing](#)
 [management](#)
 [COM+ types](#)
 [deployment](#)
 [execution](#)
 [monitoring](#)
 [secure applications](#) [See secure applications]
 [security testing](#)
[architect \(roles\)](#)
ASP.NET
 [authentication](#)
 [Forms authentication](#)
 [IIS](#)
 [impersonation and](#)
 [Passport authentication](#)
 [protected page](#)
 [turning off](#)
 [Windows](#)

authorization

[file authorization](#)

[URL authorization](#)

[CAS and](#)

[CLR and](#)

configuration

[files](#)

[<location\\> element](#)

[logon page](#)

[security-related elements](#)

[impersonation](#)

[security, overview](#)

[Web.config file](#)

[worker processes](#)

[<processModel\\> element](#)

assemblies

[application domain and](#)

[application domain policies, programming considerations](#)

[Author assembly, trust](#)

[AuthorMembershipCondition example](#)

[compiling](#)

[GAC and](#)

[creating](#)

[multi-file](#)

[single file](#)

[declarative security statements, Permissions View tool](#)

[decompiling](#)

[delayed signing support](#)

evidence

[assigning](#)

[assigning to](#)

[embedding in assemblies](#)

[permissions](#)

[viewing](#)

[fully trusted](#)

[deleting from list](#)

[.NET Framework Configuration tool and](#)

[programming considerations](#)

[fully trusted assemblies](#)

[GAC \(Global Assembly Cache\)](#)

[installing into](#)

[uninstalling from](#)

[identity, evidence and](#)

[installing](#)

[isolated storage and](#)

[loading, evidence](#)

[minimum permission requests](#)

[multi-file, distributing](#)

[optional permission requests](#)

[overview](#)

[permissions](#)

[policy resolution](#)

- [private compared to shared](#)
- [publisher certificates](#)
- [refusing requests](#)
- [scope attributes](#)
- [shared](#)
- [strong named, serviced components \(COM+\) and strong names](#)
 - [creating](#)
 - [specifying key pairs](#)
 - [verifying](#)
 - [version and culture metadata](#)
- [type-safe code](#)
- [Assembly class, evidence](#)
- [assembly evidence 2nd](#)
- [Assembly Linker tool](#)
- [assembly manifest](#)
- [AssemblyCulture attribute](#)
- [AssemblyDelaySign attribute](#)
- [assemblyEvidence argument, Evidence class](#)
- [AssemblyKeyFile attribute 2nd](#)
- [AssemblyKeyName attribute](#)
- [AssemblyVersion attribute](#)
- [Assert operations, security demands and assigning](#)
- [asymmetric algorithms](#)
 - [signature function](#)
 - [verification function](#)
- [asymmetric encryption 2nd](#)
 - [algorithm security](#)
 - [decrypting data](#)
 - [digital signatures](#)
 - [ElGamal algorithm](#)
 - [encrypted data creation](#)
 - [data padding](#)
 - [keys, creating](#)
- [AsymmetricAlgorithm class 2nd](#)
 - [digital signatures](#)
 - [key-related properties](#)
 - [properties and methods](#)
- [AsymmetricExchangeDeformatter class](#)
- [AsymmetricExchangeFormatter class](#)
- [AsymmetricSignatureDeformatter class](#)
- [AsymmetricSignatureFormatter class](#)
- [attribute classes](#)
 - [defining](#)
- [attributes](#)
 - [ApplicationAccessControl \(COM+\)](#)
 - [assembly scope](#)
 - [code group, policy resolution](#)
 - [ComponentAccessControl \(COM+\)](#)
 - [declarative security statements](#)
 - [SecureMethod \(COM+\)](#)

[security policy code groups](#)

[SecurityRole \(COM+\)](#)

[AttributeUsage attribute](#)

[authentication](#)

[ASP.NET](#)

[Forms authentication](#)

[impersonation and](#)

[Passport authentication](#)

[turning off](#)

[Windows](#)

[COM+ process-access security \(PAS\)](#)

[cryptography](#)

[IIS](#)

[anonymous access](#)

[basic authentication](#)

[client certificate authentication](#)

[digest authentication](#)

[impersonation and](#)

[integrated Windows authentication](#)

[mode configuration](#)

[RBS and](#)

[Authentication Methods window, Internet Service Manager](#)

[Author assembly, trust](#)

[Author evidence class 2nd](#)

[authority, RBS](#)

[authorization](#)

[ASP.NET](#)

[file authorization](#)

[URL authorization](#)

[RBS and](#)

[<authorization\> element](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[basic authentication, IIS](#)

[batch files, Code-Access Security Policy tool](#)

[BigInteger class](#)

[birthday attacks](#)

[block cypher hash codes \(keyed hashing algorithms\)](#)

[block padding](#)

[BlockSize property, symmetric algorithms](#)

[brute force attacks](#)

[cryptography](#)

[probability and](#)

[business sponsor](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[C# syntax, converting to Visual Basic syntax](#)

[call stack 2nd](#) [See also [stack walk](#)]

[CanTransformMultipleBlocks property](#)

CAS (code-access security)

[application domains and](#)

[ASP.NET and](#)

[classes](#)

[code-access permissions](#)

[Code-Access Security Policy tool](#)

[enforcing](#)

[evidence](#)

[extensions, assembly trust and](#)

[isolated storage and](#)

[.NET Framework Configuration tool](#)

[overview](#)

[permissions](#)

[assembly scope attributes](#)

[attribute class definitions](#)

[creating custom](#)

[dll creation](#)

[enforcing security](#)

[example custom class](#)

[example custom class definition](#)

[example custom class enumeration](#)

[IPermission interface](#)

[ISecurityEncodable interface](#)

[IUnrestrictedPermission interface](#)

[overview](#)

[Permissions View tool](#)

[security policy](#)

[statements](#)

[imperative security](#)

[overview](#)

[turning on/off](#)

[user policy level](#)

[Windows security and](#)

[case sensitivity, C#](#)

[Caspol.exe 2nd](#)

[CAS, turning on/off](#)

[execution permission checks, turning on/off](#)

[forcing securing changes](#)

[isolated storage security](#)

[policy evaluation](#)

[policy level administration](#)

[code groups](#)

[fully trusted assemblies](#)

[named permission sets](#)

- [target code group](#)
- [resetting policies](#)
- [catalogue \(COM+\), viewing](#)
- [CBC \(Cipher Block Chaining\) cipher mode](#)
- [Certificate Creation tool](#)
- certificates
 - [publisher](#)
 - [X509, creating](#)
- [CFB \(Cipher Feedback\) cipher mode](#)
- [Check method](#)
- [IMembershipCondition interface](#)
- [CheckExecutionRights property](#)
- [child code groups \(security policies\)](#)
- [Children property, CodeGroup objects](#)
- [Cipher Block Chaining \(CBC\) cipher mode](#)
- [Cipher Feedback \(CFB\) cipher mode](#)
- [cipher functions 2nd \[See also algorithms\]](#)
 - [block padding](#)
 - [CFB \(Cipher Feedback\)](#)
 - [cipher block chaining](#)
 - [ECB](#)
 - [XTEA](#)
- [cipher modes](#)
 - [CFB \(Cipher Feedback\)](#)
 - [Cipher Block Chaining \(CBC\)](#)
 - [ECB \(Electronic Codebook\)](#)
 - [symmetric algorithms, configuring](#)
- [ciphertext](#)
- [class hierarchies \(quick-reference\)](#)
- [class libraries, trust levels](#)
- classes
 - [AsymmetricAlgorithm](#)
 - [AsymmetricExchangeDeformatter](#)
 - [AsymmetricExchangeFormatter](#)
 - [AsymmetricSignatureDeformatter](#)
 - [AsymmetricSignatureFormatter](#)
 - [attribute, defining](#)
 - [AuthorMembershipCondition example](#)
 - [C#/Visual Basic syntax](#)
 - [code-access permission](#)
 - [custom membership condition](#)
 - [defining](#)
 - [EIGamalOAEPKeyExchangeFormatter](#)
 - [EIGamalOAEPKeyExchangeDeformatter](#)
 - [EIGamalPKCS1SignatureFormatter](#)
 - [EIGamalSignature](#)
 - [Evidence](#)
 - evidence
 - [implementation](#)
 - [standard evidence classes](#)
 - [Hash class](#)
 - [identity permission](#)

[IMembershipCondition interface and namespace quick reference and .NET Framework, COM+ and permission attribute](#)
[permission statements](#)
[permissions, programming considerations](#)
[RandomNumberGenerator](#)
[SecurityElement](#)
[signature formatter classes, digital signatures](#)
[SignedXML](#)
[Site class](#)
[StrongName](#)
[Url](#)
[XMLDSIG](#)
[Zone](#)
[classesPublisher](#)
[client certificate authentication, IIS](#)
[CloseDefect method](#)
CLR (Common Language Runtime)
[application domains 2nd](#)
[configuration](#)
[evidence](#)
[identity](#)
[security policy and application execution and](#)
ASP.NET
[applications](#)
[runtime](#)
evidence
[application domains](#)
[assemblies](#)
[identities, application domains](#)
[Internet Explorer and non-type-safe code](#)
[operating system, shell command execution](#)
[security policy, application domains and type-safe code](#)
[verification, application deployment and](#)
[Code Access Security Policy tool](#)
[code groups](#)
[attributes, policy resolution](#)
[Caspol.exe and Code-Access Security Policy tool and creating 2nd 3rd](#)
[creating manually](#)
[importing, from file selects](#)
[membership conditions, programming](#)
[named permission sets and policy levels](#)
[policy statements, programming](#)
[code-access security \[See CAS\]](#)
[Code-Access Security Policy tool 2nd](#)

- [code-access security, turning on/off](#)
- [execution permission checks, turning on/off](#)
- [forcing security changes](#) 2nd
- [policy level administration](#)
 - [code groups](#)
 - [fully trusted assemblies](#)
 - [named permission sets](#)
 - [target code group](#)
- [resetting policies](#)
- CodeAccessPermission class
 - [interfaces](#)
 - [methods](#)
 - [removing stack walk overrides](#)
- [CodeAccessSecurityAttribute class](#)
- [CodeGroup class](#)
 - [methods and properties](#)
- COM+
 - [application management](#)
 - [application roles, populating](#)
 - [application types](#)
 - [specifying](#)
 - [applications, installing](#)
 - attributes
 - [ApplicationAccessControl](#)
 - [ComponentAccessControl](#)
 - [SecureMethod](#)
 - [SecurityRole](#)
 - [catalogue, viewing](#)
 - [component management](#)
 - [configuration](#)
 - [Maeshaller role](#)
- PAS (process-access security)
 - [authentication](#)
 - [impersonation](#)
- [RBS \(role-based security\)](#)
 - [compatibility with .NET role-based security](#)
- [roles, scope](#)
- [serviced components, creating](#)
- [command line, Code-Access Security Policy tool](#)
- compiling
 - [assemblies](#)
 - [creating](#)
 - [GAC and](#) 2nd
 - [decompiling](#)
 - [assemblies](#)
 - [protecting against](#)
- [Component Services explorer \(COM+\)](#)
- [ComponentAccessControl attribute \(COM+\)](#)
- [ComputeHash method](#)
 - [streamed data](#)
- [ComputeSignature method](#)
- confidentiality

- [cryptography](#)
- [symmetric encryption](#)
- configuration
 - [application domains](#)
 - ASP.NET
 - [files](#)
 - [<location\\> element](#)
 - [logon page](#)
 - [security-related configuration elements](#)
 - COM+
 - [forms authentication](#)
 - [IIS authentication modes](#)
 - [security policy 2nd](#)
 - [SecurityPermissionAttribute class, properties](#)
 - [shared assemblies and](#)
 - symmetric algorithms
 - [cipher modes](#)
 - [Key and IV properties](#)
 - [KeySize and BlockSize properties](#)
 - [WindowsPrincipal class](#)
- [configured classes \(COM+\)](#)
- [ControlDomainPolicy permission](#)
- [ControlEvidence permission](#)
 - [System.Security.SecurityException](#)
- [ControlPolicy permission](#)
- [Copy method](#)
 - [IPermission interface](#)
- [cracker \(roles\)](#)
- Create method
 - [hashing algorithms](#)
 - [SymmetricAlgorithm class](#)
- [CreateDecryptor method](#)
- [CreateDirectory method](#)
- [CreateEncryptor method 2nd](#)
- [CreateEventSource method, registering event sources](#)
- [CreateFromUrl factory method](#)
- [CreateKeyExchange method](#)
- [CreateKeyPair method](#)
- [CreateNewDefect method](#)
- [CreatePermission method](#)
- [CreateSignature method](#)
- [credentials, RBS and](#)
- [cryptanalysis](#)
- [Crypto API \(Windows\)](#)
- [cryptographic hash code](#)
- [cryptographic key pairs](#)
- cryptographic keys
 - [complexity](#)
 - [CreateKeyExchange method](#)
- creating
 - [algorithm classes and](#)
 - [key-derivation protocol](#)

- [random number generator and](#)
- [ElGamal encryption functions](#)
- [ElGamalOAEPKeyExchangeFormatter class](#)
- [key exchanges, encryption](#)
- [overview](#)
- [persistence](#)
- [session keys](#)
 - [encryption](#)
- [symmetric algorithms](#)
- [symmetric key exchange](#)
- [Cryptographic Service Provider \(CSP\)](#)
- [cryptography 2nd](#)
 - [attacks on, types of](#)
 - [authentication](#)
 - [birthday attacks](#)
 - [brute force attacks](#)
 - [confidentiality](#)
 - [hash functions](#)
 - [hashing algorithms](#)
 - [creating](#)
 - [hash code lengths](#)
 - [message size limits](#)
 - [integrity](#)
 - [key management](#)
 - [message collisions](#)
 - [software weaknesses](#)
- [Cryptography.DSA class](#)
 - [CreateSignature method](#)
 - [VerifySignature method](#)
- [CryptoStream class](#)
 - [XTEA \(Extended Tiny Encryption Algorithm\)](#)
- [CSP \(Cryptographic Service Provider\)](#)
- [CurrentPrincipal property, IPrincipal interface](#)
- [custom event logs, creating](#)
- [custom evidence](#)
 - [Author assembly](#)
 - [constructors](#)
 - [creating](#)
 - [identity permissions](#)
 - [immutability](#)
 - [Object.ToString method](#)
 - [serializing](#)
 - [uses](#)
- [custom runtime hosts](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[Data Encryption Standard \(DES\) algorithm](#)

[data sharing, isolated storage and](#)

[data types](#)

[assemblies and](#)

[C#/Visual Basic equivalents](#)

[declarative demands](#)

[declarative security statements](#)

[Assert operations](#)

[custom permission classes](#)

[Deny operations](#)

[inspecting](#)

[Permissions View tool and](#)

[PermitOnly operations](#)

[role-based](#)

[decompiling](#)

[assemblies](#)

[protecting against](#)

[reasons not to](#)

[Decrypt method](#)

[DecryptData method](#)

[decryption](#)

[ElGamal algorithm](#)

[ElGamalOAEPKeyExchangeDeformatter class](#)

[RSACryptoServiceProvider class](#)

[symmetric algorithms](#)

[DecryptValue method](#)

[delayed signing support for assemblies](#)

[delegates](#)

[C#/Visual Basic delegates syntax](#)

[namespace quick reference and](#)

[Delete method, custom event logs](#)

[DeleteDirectory method, isolated storage](#)

[DeleteEventSource method](#)

[DeleteFile method, isolated storage](#)

[Demand method, PrincipalPermission object](#)

[Demand statement](#) [2nd](#) [See also security demands]

[Deny operations, security demands and](#)

[deployment, applications](#)

[DES \(Data Encryption Standard\) algorithm](#)

[developer role \(COM+\)](#)

[digest authentication, IIS](#)

[digital signatures](#)

[algorithms](#)

[managed algorithm class](#)

[assembly strong names](#)

[AsymmetricAlgorithm class](#)

[AsymmetricSignatureDeformatter class](#)

[AsymmetricSignatureFormatter class](#)
[byte arrays, verification](#)
[ComputeSignature method](#)
[CreateSignature method](#)
[creating](#)
[defomatter class](#)
[DSACryptoServiceProvider class](#)
[ElGamal signature functions](#)
[ElGamalPKCS1SignatureDeformatter class](#)
[ElGamalPKCS1SignatureFormatter class](#)
[ElGamalSignature class](#)
[forging](#)
[key pairs, creating](#)
[managed class methods](#)
[overview](#)
[PKCS #1 Helper class](#)
[PKCS #1 protocol](#)
[programming](#)
[publisher certificates](#)
[reusing](#)
[RSACryptoServiceProvider class](#)
[security objectives](#)
[signature formatter classes](#)
[signature function](#)
[SignData method](#)
[signed documents, altering](#)
[SignHash method](#)
[verification](#)
[verification function](#)
[VerifyData method](#)
[VerifyHash method](#)
[XMLDSIG](#)
 [embedding objects](#)
 [overview](#)
 [signature creation 2nd](#)
 [SignedXML class](#)
 [signing algorithm](#)
 [verification](#)

directories

[deleting, from isolated storage](#)
 [searches, isolated storage](#)
[Directory Service log](#)
[Dispose method](#)
[distribution, multi-file assemblies](#)
[dlls, creating](#)
[DNS Server log](#)
[DSA \(Digital Signature Algorithm\)](#)
[DSACryptoServiceProvider class](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[ECB \(Electronic Codebook\) cipher mode](#)

[editing security policies, manually](#)

[Electronic Codebook \(ECB\) cipher mode](#)

[ElGamal algorithm](#)

[abstract cipher function class definition](#)

[abstract class definition](#)

[BigInteger class](#)

[decryption class definition](#)

[encryption and decryption functions](#)

[encryption class definition](#)

[encryption functions, session key exchanges](#)

[implementation class definition](#)

[key generation protocol](#)

[signature functions](#)

[testing](#)

[ElGamalAbstractCipher class](#)

[ElGamalEncryptor class](#)

[ElGamalKeyStruct structure](#) [2nd](#)

[ElGamalManaged class](#)

[ElGamalOAEPKeyExchangeDeformatter class](#)

[ElGamalOAEPKeyExchangeFormatter class](#)

[ElGamalParameters structure](#)

[ElGamalPKCS1SignatureFormatter class](#)

[ElGamalSignature class](#)

[ElGamaPKCS1SignatureDeformatter class](#)

[ELS \(Event Log Service\)](#)

[binary data in](#)

[custom event logs](#)

[event identifier and category](#)

[event source name](#)

[event source values](#)

[event structure](#)

[event types](#)

[logs](#)

[message component](#)

[monitoring event logs](#)

[querying](#)

[reading event logs](#)

[registering event sources](#)

[writing events](#)

[email, resource restrictions](#)

[embedding](#)

[evidence in assemblies](#)

[objects, in digital signatures](#)

[Encrypt method](#)

[EncryptData method](#)

[encryption](#) [\[See also cryptography\]](#) [2nd](#) [3rd](#)

- [algorithms](#)
- [asymmetric](#)
 - [algorithm security](#)
 - [creating keys](#)
 - [ElGamal algorithm](#)
 - [encrypted data creation](#)
- [ElGamal algorithm](#)
- [ElGamal encryption functions](#)
- [functions, asymmetric](#)
- [RSACryptoServiceProvider class](#)
- [session keys](#)
- [symmetric](#)
 - [algorithms](#)
 - [block padding](#)
 - [cipher modes](#)
 - [creating encrypted data](#)
 - [encrypting and decrypting data](#)
 - [key lengths](#)
 - [.NET Framework algorithms](#)
- [EncryptValue method](#)
- [end-to-end security](#)
- [enterprise level policy](#)
- [enterprise policy level 2nd](#)
- enumerations
 - [C#/Visual Basic syntax](#)
 - [namespace quick reference and](#)
- [EnvironmentPermission object](#)
- [Event Log Service \[See ELS\]](#)
- [event logs \[See ELS\]](#)
- event sources (ELS)
 - [name](#)
 - [registering](#)
 - [values](#)
- [event structure \(ELS\)](#)
 - [binary data and](#)
 - [event identifier and category](#)
 - [event types](#)
 - [message component](#)
- EventLog class
 - [properties and methods](#)
 - [querying ELS](#)
- [EventLog.Entries property](#)
- EventLogEntryCollection class
 - [methods](#)
 - [properties](#)
- [events, C#/Visual Basic syntax](#)
- evidence
 - [AppDomain class](#)
 - [application domains](#)
 - [assigning evidence to](#)
- assemblies
 - [assigning evidence to 2nd](#)

- [embedding evidence in permissions viewing](#)
- [Assembly class](#)
- [assembly evidence 2nd](#)
- [CAS and custom evidence](#)
 - [constructors](#)
 - [creating](#)
 - [immutability](#)
 - [Object.ToString method](#)
 - [serializing](#)
 - [uses](#)
- [host evidence](#)
 - [ControlEvidence permission](#)
- [identity and inheritance](#)
- [.NET Framework, extending overview](#)
- [programming](#)
 - [assigning to assembly](#)
 - [Evidence collection overview](#)
- [security policies and standard evidence classes](#)
- [standard types](#)
- [Evidence class](#)
 - [assemblyEvidence argument](#)
 - [constructors](#)
 - [ControlEvidence permission](#)
 - [hostEvidence argument](#)
 - [ICollection implementation](#)
 - [members](#)
- [evidence classes](#)
 - [ApplicationDirectory class](#)
 - [Author class 2nd](#)
 - [content](#)
 - [custom, implementation](#)
 - [Hash class](#)
 - [Publisher class](#)
 - [serializable](#)
 - [Site class](#)
 - [StrongName class](#)
 - [Url class](#)
 - [Zone class](#)
- [Evidence collection](#)
 - [adding types](#)
 - [evidence objects](#)
 - [objects](#)
 - [copies](#)
 - [shallow copies](#)
- [Evidence property](#)

Exclusive attribute

[code group policies](#)

[security policy code groups](#)

[executable assemblies](#)

[execution permission checks, turning on/off](#)

[execution, secure applications](#)

[ExecutionEngineException exception](#)

[ExportParameters method 2nd 3rd](#)

[Extended Tiny Encryption Algorithm \[See XTEA\]](#)

[external resources, application domains](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[factory methods, CreateFromUrl](#)
[failure strategy development](#)
[FieldCodeGroup class, Resolve method](#)
[fields, C#/Visual Basic syntax](#)
[file authorization, ASP.NET](#)
[File Replication Service log](#)
[File.Delete method](#)
[FileCodeGroup class](#)
 [constructor](#)
[FileIOPermission object 2nd](#)
files
 [deleting, from isolated storage](#)
 [IIS authentication mode](#)
 [isolated storage](#)
 [searches](#)
 [policy files](#)
 [resource restrictions](#)
[FirstMatchCodeGroup class](#)
 [Resolve method](#)
[flags, quick-reference entries](#)
[folders, authentication modes, IIS](#)
[Forms authentication, ASP.NET](#)
 [configuration](#)
 [logon page](#)
 [protected page creation](#)
[FormsAuthentication.RedirectFromLoginPage method](#)
[FormsAuthentication.Authenticate method](#)
[FromXml method](#)
 [ISecurityEncodable interface](#)
[FromXmlString method 2nd](#)
[fully trusted assemblies 2nd](#)
 [.NET Framework Configuration tool and](#)
 [Caspol.exe and](#)
 [deleting from list](#)
 [programming considerations](#)
[fully trusted assemblies, list, adding to](#)
[functional resources, application domains](#)
[functions, ElGamal algorithm](#)
 [signature functions](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[GAC \(Global Assembly Cache\)](#)

[assemblies](#)

[installing](#)

[uninstalling](#)

[listing contents](#)

[reasons to use](#)

[gacutil.exe](#)

[GenerateHash method](#)

[GenericPrincipal class, generic RBS implementation](#)

[GenericPrincipal object](#)

[GetDirectoryNames method, isolated storage](#)

[GetFileNames method, isolated storage](#)

[GetHashCode method, overriding](#)

[GetNamedPermissionSet method](#)

[GetUserStorageForDomain](#)

[GetUserStoreForAssembly method](#)

[GetUserStoreForDomain method](#)

[Global Assembly Cache](#) [\[See GAC\]](#)

[Global Assembly Cache tool](#)

[grant set, permissions](#)

[granting permissions](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[hacker, roles](#)

[hardware, resource restrictions](#)

[Hash class](#)

[hash codes](#) [\[See also hashing algorithms\]](#)

[cryptography](#)

[digital signatures](#)

[generating](#)

[from data in memory](#)

[from streamed data](#)

[keys](#)

[parsing](#)

[validating](#)

[hash functions, cryptography](#)

[HashAlgorithm class](#)

[methods and properties](#)

[HashCore method, implementing](#)

[HashFinal method, implementing](#)

[hashing algorithms](#)

[abstract classes](#)

[extending .NET Framework](#)

[generating](#)

[from data in memory](#)

[from streamed data](#)

[hash codes](#)

[creating](#)

[lengths of](#)

[implementation classes](#)

[testing](#)

[Initialize method](#)

[instantiating](#)

[keyed](#)

[block cypher hash codes](#)

[creating](#)

[HMAC standard](#)

[.NET Framework](#)

[KeyedHashAlgorithm class](#)

[creating keyed hash codes](#)

[instantiating](#)

[message size limits](#)

[.NET Framework](#)

[programming considerations](#)

[HashSize property, implementing](#)

[HMAC standard \(keyed hashing algorithms\)](#)

[HMACSHA1 algorithm](#)

[host evidence](#)

[ControlEvidence permission](#)

[hostEvidence argument, Evidence class](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[ICollection collection, implementation](#)

[ICryptoTransform interface](#)

[XTEA](#)

[IDefectTracker interface](#)

[identification, trust and](#)

[identities](#)

[.NET RBS](#)

[application domains](#)

[assemblies, evidence](#)

[evidence](#)

[<identity\\> element, impersonation](#)

[identity permission](#)

[classes](#)

[identity permissions](#)

[custom evidence and](#)

[IIdentity interface](#)

[.NET RBS and](#)

[WindowsIdentity class](#)

[IIdentityPermissionFactory interface](#)

[IIS \(Internet Information Service\)](#)

[ASP.NET and](#)

[authentication](#)

[anonymous access](#)

[basic authentication](#)

[client certificate authentication](#)

[digest authentication](#)

[impersonation and](#)

[integrated Windows authentication](#)

[mode configuration](#)

[IMembershipCondition interface 2nd](#)

[custom membership conditions](#)

[member implementation](#)

[imperative demands](#)

[imperative RBS demands](#)

[imperative security statements](#)

[Assert operations](#)

[Deny operations](#)

[executing with SecurityPermission objects](#)

[PermitOnly operations](#)

[Impersonate method, WindowsIdentity object](#)

[impersonation](#)

[ASP.NET](#)

[COM+ process-access security \(PAS\)](#)

[implementation classes](#)

[ElGamal algorithm](#)

[hashing algorithms](#)

[testing](#)

[XTEA](#)

importing

[code groups, from file selects](#)

[named permission sets](#)

[ImportParameters method](#) [2nd](#) [3rd](#)

[inheritance, evidence](#)

[InheritanceDemand statement](#)

[initialization vector \(IV\)](#)

[symmetric algorithms, configuring](#)

[Initialize method, implementing](#)

installation

[assemblies](#)

[into GAC](#)

[COM+ applications](#)

[key pairs, into CSP container](#)

[integrated Windows authentication, IIS](#)

[integrity, cryptography](#)

[intellectual property, theft, decompiling and](#)

interfaces

[C#/Visual Basic syntax](#)

[CodeAccessPermission class](#)

[identity permissions](#)

[IPermission, imperative RBS demands](#)

[namespace quick reference and](#)

[Internet access, resource restrictions](#)

[Internet Explorer, CLR and](#)

[Intersect method, IPermission interface](#)

[intersection permission sets](#)

[IPermission interface](#)

[imperative RBS demands](#)

[implementing](#)

[methods](#)

IPrincipal interface

[CurrentPrincipal property](#)

[.NET RBS](#)

[IPrincipal object](#)

[PrincipalPolicy enumeration](#)

[SetThreadPrincipal method](#)

[WindowsPrincipal class](#)

ISecurityEncodable interface

[implementing](#)

[methods](#)

[isf argument, IsolatedStorageFile class](#)

[IsGranted method](#)

isolated storage

[administration](#)

[application domains and](#)

[as primary storage](#)

[by application domain](#)

[by assembly](#)

[by user](#)

[CAS and](#)

[DeleteDirectory method](#)

[DeleteFile method](#)

directories

[deleting](#)

[GetDirectoryNames method](#)

[searches](#)

files

[deleting](#)

[GetFileNames method](#)

[searches](#)

[GetUserStoreForAssembly method](#)

[GetUserStoreForDomain method](#)

[IsolatedStorageFile class](#)

[IsolatedStorageFileStream class](#)

[levels of isolation](#)

[limitations](#)

[overview 2nd](#)

[Remove method](#)

security

[Caspol.exe](#)

[Mscorcfg.msc](#)

[security policy configuration](#)

stores 2nd

[deleting](#)

[directories](#)

[user-level isolation](#)

[Isolated Storage tool](#)

[IsolatedStorageContainment enumeration](#)

[IsolatedStorageException](#)

[IsolatedStorageFile class, isf argument](#)

[IsolatedStorageFile instances, passing](#)

[IsolatedStorageFile.GetStore method](#)

[IsolatedStorageFileStream class, Handle property](#)

[IsSubsetOf method, IPermission interface](#)

[IStackWalk interface](#)

[PermissionSet class](#)

[IsUnrestricted method](#)

[IUnrestrictedPermission interface, methods](#)

[IUnrestrictedPermission interface](#)

[implementing](#)

[IV \(initialization vector\)](#)

[IV property, symmetric algorithm](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[key pairs](#)

[creating](#)

[installing in CSP container](#)

[Key property, symmetric algorithms](#)

[key-derivation protocol, cryptographic key creation](#)

[keyed hashing algorithms](#)

[block cypher hash codes](#)

[creating](#)

[creating keyed hash codes](#)

[HMAC standard](#)

[instatiating](#)

[.NET Framework](#)

[KeyedHashAlgorithm class](#)

[creating keyed hash codes](#)

[instantiating](#)

[KeyExchangeAlgorithm property](#)

[keys](#)

[asymmetric encryption](#)

[creating](#)

[length of](#)

[cryptography, management](#)

[encryption](#)

[hash codes](#)

[length, symmetric encryption](#)

[RSA algorithm, implementing](#)

[KeySize property](#)

[symmetric algorithm](#)

[KeyStruct property](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[Label property, PolicyLevel class](#)

[legal issues](#) [See also intellectual property]

[decompiling assemblies](#)

[LegalKeySizes property](#)

[LevelFinal attribute](#)

[code group policies](#)

[security policy code groups](#)

[libraries, DLLs, creating](#)

[library applications, COM+](#)

[library assemblies](#)

[creating](#)

[strong names and](#)

[line breaks \(Visual Basic syntax\)](#)

[LinkDemand statement](#)

[LoadAndList property, evidence viewing](#)

[<location\> element, ASP.NET configuration](#)

[Log property](#)

[LogNameFromSourceName method](#)

[logs, ELS \(Event Log Service\)](#)

[binary data](#)

[custom event logs](#)

[event identifier and category](#)

[event source name](#)

[event source values](#)

[event structure](#)

[event types](#)

[message component](#)

[monitoring event logs](#)

[querying](#)

[reading](#)

[registering event sources](#)

[writing events](#)

[luring attacks](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[machine level policy 2nd](#)

[machine policy code groups](#)

[MachineName property](#)

[makecert.exe](#)

[managed algorithm class, digital signatures and](#)

[managed class methods, digital signatures](#)

[managed code, isolated storage and](#)

[manager role \(COM+\)](#)

[Marshaller role \(COM+\)](#)

[MD5 hashing algorithm](#)

[member synopsis \(quick-reference entries\)](#)

[membership conditions](#)

[AuthorMembershipCondition example assembly](#)

[AuthorMembershipCondition example class](#)

[implementing](#)

[code groups](#)

[programming](#)

[programming considerations](#)

[custom](#)

[requirements](#)

[testing](#)

[policy resolution and](#)

[security policy code groups](#)

[XML representation, creating](#)

[MembershipCondition property](#)

[message collisions \(cryptography\)](#)

[MessageUtil class, Permissions View tool](#)

[methods](#)

[activation records and](#)

[AsymmetricAlgorithm class](#)

[C#/Visual Basic syntax](#)

[CodeAccessPermission class](#)

[CodeGroup class](#)

[ComputeSignature](#)

[CreateKeyExchange](#)

[CreateSignature](#)

[custom CAS permissions, creating](#)

[EventLog class](#)

[EventLogEntryCollection class](#)

[factory methods, CreateFromUrl](#)

[FormsAuthentication.RedirectFromLoginPage](#)

[FormsAuthentication.Authenticate](#)

[GenerateHash](#)

[GetUserStoreForAssembly](#)

[HashAlgorithm class](#)

[ICryptoTransform interface](#)

[IPermission interface](#)

[IsolatedStorageFile.GetStore](#)

[RSA class](#)

[RSACryptoServiceProvider class](#)

[SecurityManager class](#)

[SecurityRole attribute](#)

[SetThreadPrincipal](#)

[SignData](#)

[SignHash method](#)

[SymmetricAlgorithm class](#)

[VerifyData](#)

[VerifySignature](#)

[WindowsIdentity.GetCurrent](#)

[WindowsPrincipal.IsInRole](#)

[Microsoft Intermediate Language \(MSIL\)](#)

[MMC \(Microsoft Management Console\), .NET Framework Configuration tool](#)

[modules, assemblies](#)

[monitoring applications](#)

[Mscorcfg.msc](#)

[isolated storage security](#)

[MSIL \(Microsoft Intermediate Language\)](#)

[decompiling and](#)

[MSIL code section \(assemblies\)](#)

[multi-file assemblies](#)

[creating](#)

[delayed signing and](#)

[distributing](#)

[strong names, creating](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[named permission sets](#)

[.NET Framework Configuration tool and](#)

[Caspol.exe and](#)

[code groups](#)

[Code-Access Security Policy tool and](#)

[creating](#)

[creating manually](#)

[importing](#)

[permission objects](#)

[pre-defined](#)

[programming considerations](#)

[security policies](#)

[NamedPermissionSet class](#)

[namespaces](#)

[quick reference to](#)

[type names and](#)

[native code, runtime security and](#)

[native compilation \(decompiling protection\)](#)

[NeedToGenerateKey method](#)

[.NET Framework](#)

[algorithms, symmetric encryption](#)

[application isolation](#)

[classes, COM+ and](#)

[EventLog class, properties and methods](#)

[extending, evidence and](#)

[hashing algorithms](#)

[adding to](#)

[keyed hashing algorithms](#)

[large integer processing](#)

[.NET Framework Configuration tool 2nd](#)

[CAS](#)

[fully trusted assemblies and](#)

[adding to list](#)

[named permission sets and](#)

[named permission sets, creating](#)

[starting](#)

[NetCodeGroup class](#)

[constructor](#)

[Resolve method](#)

[networks, ELS logs](#)

[non-type-safe code](#)

[nonmanaged code, isolated storage and](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[OAEP padding mode](#)

[obfuscation \(decompiling protection\)](#)

[Object.ToString method, overriding](#)

[objects, embedding, in digital signatures](#)

[operating systems, application isolation](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[ParseHashCodeString method](#)

[parsing, hash codes](#)

[PAS \(process-access security\), COM+](#)

[authentication](#)

[impersonation](#)

[Passport authentication](#)

[permission attribute classes](#)

[permission classes](#)

[CAS statements](#)

[programming](#)

[class functionality](#)

[SecurityPermission class](#)

[serialization](#)

[permission demands](#)

[Demand statement](#)

[InheritanceDemand statement](#)

[LinkDemand statement](#)

[permission objects, named permission sets](#)

[permission requests](#)

[minimum](#)

[optional](#)

[refusing](#)

[types](#)

[permission sets](#)

[creating](#)

[named](#)

[programming considerations](#)

[security policies](#)

[security policy code groups](#)

[Permission Settings dialog box](#)

[permissions](#)

[application domain policies, programming considerations](#)

[CAS and](#)

[ControlEvidence](#)

[creating custom](#)

[assembly scope attributes](#)

[attribute class definitions](#)

[dll creation](#)

[enforcing security](#)

[example](#)

[example class definition](#)

[example enumeration](#)

[IPermission interface](#)

[ISecurityEncodable interface](#)

[IUnrestrictedPermission interface](#)

[grant set](#)

[granting](#)

[identity permissions, custom evidence](#)
[isolated storage](#)
[named permission sets](#)
[.NET Framework Configuration tool and
creating manually](#)
[PrincipalPermission class, RBS demands
requesting](#)
[security policies and
types](#)
[code-access](#)
[identity](#)
[overview](#)
[Permissions View tool, CAS](#)
[Permissions View tool, declarative security statements and
PermissionSet class](#)
[PermissionSetAttribute class](#)
[PermissionState enumeration 2nd](#)
[PermitOnly operations, security demands and
Permview.exe](#)
[persistence, cryptographic keys](#)
[PKCS #1 signature-formatting protocol](#)
[PKCS #1 Helper class](#)
[PKCS #7 padding](#)
[PKCS \(Public Key Cryptography Standard\)
plaintext, encrypting](#)
[policies](#)
[application domain, programming considerations](#)
[assemblies, fully trusted](#)
[AuthorMembershipCondition assembly](#)
[AuthorMembershipCondition example class
implementing](#)
[code groups](#)
[configuration 2nd](#)
[custom membership condition classes](#)
[levels](#)
[assembly trust and
manually editing](#)
[named permission sets](#)
[policy administration](#)
[policy evaluation, Caspol.exe and
policy files](#)
[enterprise policy level](#)
[fully trusted assemblies](#)
[machine policy code groups](#)
[named permission sets](#)
[policy levels](#)
[Caspol.exe and
code groups](#)
[Code-Access Security Policy tool and
named permission sets and
targeting](#)
[policy resolution 2nd 3rd](#)

- [code group attributes](#)
 - [granting permissions](#)
- policy statements
 - [code groups, programming considerations](#)
 - [programming](#)
- [PolicyHierarchy method](#)
- PolicyLevel class
 - [members](#)
 - [programming considerations](#)
 - [code group tree management](#)
 - [fully trusted assemblies](#)
 - [named permission sets](#)
- [PolicyStatementAttribute enumeration](#)
- [prime numbers, key generation and](#)
- [principal policy, IPrincipal interface](#)
- PrincipalPermission class
 - [constructors](#)
 - [imperative RBS demands](#)
 - [RBS demands and](#)
- [PrincipalPolicy enumeration](#)
- [principals, .NET RBS](#)
- [private assemblies](#)
- [private keys](#)
 - [importance of](#)
- [ProcessData method](#)
- [ProcessDataBlock method](#)
- [ProcessFinalDataBlock method](#)
- [<processModel\> element, ASP.NET worker processes](#)
 - [userName attribute](#)
- [programmer \(roles\)](#)
- programming
 - [application types \(COM+\), specifying](#)
 - [AuthorMembershipCondition example assembly](#)
 - [AuthorMembershipCondition example class](#)
 - [implementing](#)
- CAS permissions
 - [assembly scope attributes](#)
 - [attribute class definitions](#)
 - [custom](#)
 - [dll creation](#)
 - [enforcing security](#)
 - [example custom class](#)
 - [example custom class definition](#)
 - [example custom class enumeration](#)
 - [IPermission interface](#)
 - [ISecurityEncodable interface](#)
 - [IUnrestrictedPermission interface](#)
- [code groups 2nd](#)
 - [membership conditions](#)
 - [policy statements](#)
- COM+
 - [ApplicationAccessControl attribute](#)

- [ComponentAccessControl attribute](#)
- [SecureMethod attribute](#)
- [SecurityRole attribute](#)
- [custom membership conditions](#)
- [DefectTracker interface](#)
- [digital signatures](#)
- EIGamal algorithm
 - [abstract cipher function class definition](#)
 - [abstract class definition](#)
 - [decryption class definition](#)
 - [encryption class definition](#)
 - [implementation class definition](#)
- ELS
 - [custom event logs](#)
 - [monitoring event logs](#)
 - [querying](#)
 - [reading event logs](#)
 - [registering event sources](#)
 - [writing events](#)
- hash codes
 - [generating](#)
 - [generating from streamed data](#)
- [hashing algorithms, instantiating](#)
- [implementation classes, hashing algorithms](#)
- [keyed hashing algorithms](#)
 - [block cypher hash codes](#)
 - [creating keyed hash codes](#)
 - [HMAC standard](#)
 - [instantiating](#)
 - [.NET Framework](#)
- [multi-file assemblies, writing in different languages](#)
- [permission classes](#)
 - [class functionality](#)
 - [SecurityPermission class](#)
- [permission demands](#)
 - [Demand statement](#)
 - [InheritanceDemand statement](#)
 - [LinkDemand statement](#)
- [permission sets](#)
- [PolicyLevel class](#)
 - [code group tree management](#)
 - [fully trusted assemblies](#)
 - [named permission sets](#)
- [SecurityManager class](#)
- [serviced components \(COM+\), creating](#)
- [store directories](#)
- [SymmetricAlgorithm class, instantiating](#)

XTEA

- [abstract classes](#)
- [ICryptoTransform interface](#)
- [implementation classes](#)
- [XTEAManagedDecryptor class](#)

[XTEAManagedEncryptor class](#)

[project architect \(roles\)](#)

properties

[ApplicationAccessControl attribute \(COM+\)](#)

[AsymmetricAlgorithm class](#)

[C#/Visual Basic syntax](#)

[CodeGroup class](#)

[EventLog class](#)

[EventLogEntryCollection class](#)

[Evidence property](#)

[HashAlgorithm class](#)

[ICryptoTransform interface](#)

[LoadAndList](#)

[RSA class](#)

[RSACryptoServiceProvider class](#)

[SecurityManager class](#)

[SecurityPermissionAttribute class](#)

[SecurityRole attribute](#)

symmetric algorithms

[Key and IV](#)

[KeySize and BlockSize](#)

[SymmetricAlgorithm class](#)

[protected page, ASP.NET applications](#)

protocols

[ElGamal key generation](#)

key generation

[asymmetric encryption](#)

[RSA algorithm](#)

[public key cryptography](#)

[public key encryption](#) [See asymmetric encryption]

[public key token](#)

[public keys](#)

[assembly strong names](#)

[asymmetric encryption](#)

[delayed signing and](#)

[viewing](#)

[publisher certificates](#)

[Publisher class](#)

[Publisher evidence, application domains, assigning to](#)

[[Team LiB](#)]

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[queries, ELS \(Event Log System\)](#)

[quick reference to namespaces](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[random number generator, cryptographic key creation](#)

[RandomNumberGenerator class](#)

[RBS \(role-based security\)](#)

[authentication](#)

[authority](#)

[authorization](#)

[COM+](#)

[credentials](#)

[demands](#)

[declarative statements](#)

[imperative demands](#)

[PrincipalPermission class](#)

[generic implementation](#)

[identities](#)

[IIdentity interface 2nd](#)

[IPrincipal interface 2nd](#)

[CurrentPrincipal property](#)

[IPrincipal object](#)

[PrincipalPolicy enumeration](#)

[overview](#)

[roles](#)

[Windows implementation](#)

[impersonating Windows users](#)

[WindowsIdentity class](#)

[WindowsPrincipal class](#)

[Windows security comparison](#)

[RC2 algorithm](#)

[reading files, isolated storage](#)

[references, XML documents, XMLDSIG signatures and](#)

[RegistryPermission object](#)

[Remove method, isolated storage](#)

[RemoveFullTrustAssembly method](#)

[RemoveNamedPermissionSet method](#)

[reports, analysis](#)

[RequestMinimum statement](#)

[RequestOptional statement](#)

[RequestRefuse statement](#)

[requests permission, types](#)

[requiring trust](#)

[Resolve method, CodeGroup class](#)

[ResolveMatchingCodeGroups method](#)

[ResolvePolicy method](#)

[ResolvePolicyGroup method](#)

[restricted resources](#)

[trust](#)

[restricted resources, application domains](#)

[external resources](#)

- [functional resources](#)
- [subversion resources](#)
- [trust levels and](#)
- [Rijndael algorithm](#)
- [roaming-profiles, isolated storage and](#)
- [role-based permissions](#)
- [role-based security 2nd](#) [See RBS]
 - [overview](#)
 - [trust levels and](#)
- roles
 - [architect](#)
 - [business sponsor](#)
 - [cracker](#)
 - [programmer](#)
 - [RBS and](#)
 - [security tester](#)
 - [system administrator](#)
 - [user](#)
- [roles \(COM+\), scope](#)
- [RSA](#)
- [RSA algorithm](#)
 - [decrypting data](#)
 - [encrypted data creation](#)
 - [encrypted data creation, data padding](#)
 - [key generation protocol](#)
 - [key length](#)
 - [keys, implementing](#)
 - [signature function](#)
- [RSA class](#)
 - [properties and methods](#)
- [RSACryptoServiceProvider class](#)
 - [encrypting and decrypting data](#)
 - [instantiating](#)
 - [keys, implementing](#)
 - [properties and methods](#)
- [RSAParameters structure](#)
- [RSAParameters structure members](#)
- [runtime](#)
 - [custom runtime hosts](#)
 - [security, application domains and](#)
- [runtime security](#)
 - [native code](#)
 - [overview](#)
 - [unverifiable code](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[SavePolicy method](#)

[scope, COM+ roles](#)

[secret data, isolated storage and](#)

[secret key encryption](#) [See symmetric encryption]

[secrets](#)

[application design](#)

secure applications

[definition](#)

[deployment](#)

[design](#)

[failure](#)

[development](#)

[execution](#)

[monitoring](#)

[restricted resources](#)

[secrets](#)

[security testing](#)

[trust levels](#) 2nd

[Secure Hash Algorithm \(SHA\)](#)

[SecureMethod attribute](#)

[SecureMethod attribute \(COM+\)](#)

security

[cryptographic key pairs](#)

[end-to-end](#)

[need for](#)

[runtime security, application domains and](#)

[shared assemblies](#)

security demands

[CAS and](#)

[imperative security statements](#)

[principles of operation](#)

[stack walks](#)

[Assert operations](#)

[Deny operations](#)

[overriding](#)

[PermitOnly operations](#)

[reverting overrides](#)

[Security log \(ELS\)](#)

[security policy](#) [See policies]

[application domains and](#)

[CAS and](#)

[CLR, application domains](#)

[levels](#)

[policy levels](#)

[policy resolution and](#)

[security tester](#)

[security testing, applications](#)

- [SecurityAction enumeration 2nd](#)
- [SecurityElement class](#)
- [SecurityEnabled property](#)
- [SecurityException, Assert operations and](#)
- [SecurityManager class](#)
 - [programming considerations](#)
- [SecurityPermission class](#)
 - [implementing](#)
- [SecurityPermissionAttribute class](#)
 - [configuration properties](#)
- [SecurityPermissionFlag enumeration](#)
- [SecurityProTracker \(COM+\), populating application roles](#)
- [SecurityProTracker class](#)
- [SecurityRole attribute \(COM+\)](#)
- [SecurityRole attribute, properties](#)
- [seed data, symmetric encryption](#)
- [seed values, hashing algorithms](#)
- [Serializable attribute](#)
- serialization
 - [custom membership condition classes](#)
 - [evidence](#)
 - [permission classes](#)
- [server applications, COM+](#)
- [serviced components](#)
 - [creating](#)
- [ServicedComponent class](#)
- [services, resource restrictions](#)
- [session keys, cryptography](#)
- [session keys, encryption](#)
- [SetAppDomainPolicy method](#)
- [SetEveryoneAccess property](#)
- [SetThreadPrincipal method](#)
- [SHA \(Secure Hash Algorithm\)](#)
- [SHA1CryptoServiceProvider algorithm](#)
- [shallow copies, collection objects](#)
- [shared assemblies](#)
- [sharing data, isolated storage and](#)
- [side-channel attacks](#)
- [Sign method](#)
- [signature formatter classes](#)
- signature function
 - [digital signatures and](#)
 - [RSA protocol](#)
- [SignatureAlgorithm property](#)
- [Signcodex scheme](#)
- [SignData method](#)
- [signed documents, altering](#)
- [SignedXML class, XMLDSIG](#)
- [SignHash method](#)
- [signing authorities, delayed signing support for assemblies](#)
- [single-file assemblies](#)
 - [creating](#)

- [delayed signing and](#)
- [Site class](#)
- [sn.exe](#)
- [Software Publisher's Certificate \(SPC\)](#)
- [software, resource restrictions](#)
- [SPC \(Software Publisher's Certificate\)](#)
- [stack frames](#)
- [stack walks](#)
 - [imperative security statements](#)
 - [overriding 2nd](#)
 - [Assert operations](#)
 - [Deny operations](#)
 - [PermitOnly operations](#)
 - [reverting](#)
- [standard evidence classes](#)
- [statement terminator \(C#\)](#)
- [statements, CAS](#)
 - [declarative security](#)
 - [imperative security](#)
 - [overview](#)
- [storage, isolated storage](#)
- [stores](#)
 - [deleting](#)
 - [directories, creating](#)
 - [isolated storage](#)
- [Strong Name tool](#)
- [strong names](#)
 - [assemblies](#)
 - [creating](#)
 - [specifying key pair](#)
 - [version and culture data](#)
 - [delayed signing and](#)
 - [multi-file assemblies, creating](#)
 - [verifying](#)
- [strong-named assemblies, serviced components \(COM+\)](#)
- [StrongName class](#)
 - [constructors](#)
- [StrongNameMembershipCondition objects](#)
- [structs, namespace quick reference and](#)
- [structures, C#/Visual Basic syntax](#)
- [subversion resources, application domains](#)
- [symmetric algorithms](#)
 - [configuring](#)
 - [cipher modes](#)
 - [Key and IV properties](#)
 - [KeySize and BlockSize properties](#)
 - [cryptographic keys and](#)
 - [instantiating](#)
- [symmetric encryption 2nd](#)
 - [block padding](#)
 - [cipher modes](#)
 - [encrypted data, creating](#)

[initialization vector](#)

[key lengths](#)

[.NET Framework algorithms](#)

[symmetric key exchange, cryptographic keys](#)

[SymmetricAlgorithm class](#)

[encrypting and decrypting data](#)

[methods and properties](#)

[system administrator \(roles\)](#)

[system administrator, application deployment and](#)

[System log \(ELS\)](#)

[System.Security.Policy namespace, evidence classes](#)

[System.String argument, Site class constructor](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[test certificate](#)

[tester role \(COM+\)](#)

[testing](#)

[applications, security applications](#)

[certificates](#)

[custom membership conditions](#)

[ElGamal algorithm](#)

[implementation classes, hashing algorithms](#)

[third-party software, benefits and risks](#)

[threads, stack walks and](#)

[ToXml method](#)

[ToXmlString method](#)

[TransformBlock method](#)

[TransformFinalBlock method](#)

[Triple-DES algorithm](#)

[trust levels](#)

[Author assembly](#)

[class libraries](#)

[requiring](#)

[restricted resources and](#)

[role-based security and](#)

[secure application design](#)

[trust relationships](#)

[trust, restricted resources and](#)

[trusted assemblies, fully trusted assemblies list](#)

[type-safe code](#)

[application isolation and](#)

[non-type-safe code](#)

[types](#) [See also classes]

[names for](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[Union method, IPermission interface](#)

[union permission sets, creating](#)

[UnionCodeGroup class](#)

[Resolve method](#)

[unverifiable code, runtime security and](#)

[URL authorization, ASP.NET authorization](#)

[Url class](#)

[URLs \(Uniform Resource Locators\)](#)

[ApplicationDirectory class](#)

[authorization, forms authorization and](#)

[user \(roles\)](#)

[user groups, COM+ roles and](#)

[user level policy](#)

[user policy level](#)

[CAS](#)

[user role \(COM+\)](#)

[user-level isolation, storage](#)

[userName attribute, <processModel\> element](#)

[users, isolated storage and](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[ValidateHashCode method](#)

[validation, hash codes](#)

[verification function, digital signatures](#)

[verification, digital signatures, XMLDSIG](#)

[VerifyData method](#)

[VerifySignature method](#) [2nd](#)

[ViewAllDefects method](#)

[Visual Basic syntax, converting from C# syntax](#)

[Visual Studio .NET, multi-file assemblies and](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

web sites

[Anakrino/Exemplar decompiler](#)

[Web.config file, ASP.NET](#)

Windows

[authentication, ASP.NET](#)

[Crypto API](#)

[security, .NET RBS comparison](#)

[Windows .NET Server, .NET Framework Configuration tool](#)

[Windows 2000 Professional, .NET Framework Configuration tool execution](#)

[Windows 2000 Server, .NET Framework Configuration tool execution](#)

[Windows Crypto API](#)

[Windows executable assemblies](#)

[Windows security, CAS and](#)

[Windows XP Professional, .NET Framework Configuration Tool](#)

WindowsIdentity class

[impersonation and](#)

[members](#)

[RBS and](#)

[WindowsIdentity object, Impersonate method](#)

[WindowsIdentity.GetCurrent method](#)

WindowsPrincipal class

[configuration](#)

[RBS and](#)

[WindowsPrincipal.IsInRole method](#)

[overloads](#)

[worker processes, ASP.NET](#)

[<processModel\> element](#)

[WriteEntry method](#)

[writing files, isolated storage](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[X509 certificates, creating](#)

XML

- [ISecurityEncodable interface and membership condition representation, creating](#)
- [security policies, configuration](#)

XML documents, XMLDSIG

- [signature creation](#)
- [SignedXML class](#)

XMLDSIG

- [objects, embedding in signature](#)
- [overview](#)
- [signature creation](#)
- [signature verification](#)
- [SignedXML class](#)
- [signing algorithm](#)
- [XML documents, reference creation](#)

[XTEA \(Extended Tiny Encryption Algorithm\) 2nd](#)

- [abstract class definition](#)
- [ICryptoTransform interface](#)
- [implementation class definition](#)
- [XTEAManagedDecryptor class](#)
- [XTEAManagedEncryptor class](#)

[XTEADecrypt method](#)

[XTEAEncrypt method](#)

[XTEAManaged.CreateDecryptor method](#)

[\[Team LiB \]](#)

[\[Team LiB \]](#)

[\[SYMBOL\]](#) [\[A\]](#) [\[B\]](#) [\[C\]](#) [\[D\]](#) [\[E\]](#) [\[F\]](#) [\[G\]](#) [\[H\]](#) [\[I\]](#) [\[K\]](#) [\[L\]](#) [\[M\]](#) [\[N\]](#) [\[O\]](#) [\[P\]](#) [\[Q\]](#) [\[R\]](#) [\[S\]](#) [\[T\]](#) [\[U\]](#) [\[V\]](#) [\[W\]](#) [\[X\]](#) [\[Z\]](#)

[Zone class](#)

[\[Team LiB \]](#)