

**Microsoft®**

# **Designing Application-Managed Authorization**



patterns & practices

*Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.*

*Microsoft, Active Directory, ActiveX, MSDN, Visual Studio, Win32, and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.*

*© 2002 Microsoft Corporation. All rights reserved.*

*Version 1.0*

*The names of actual companies and products mentioned herein may be the trademarks of their respective owners.*

# Contents

## Designing Application-Managed Authorization

Summary . . . . .	1
Download . . . . .	1
Contents . . . . .	1
Introduction . . . . .	2
Understanding Authorization . . . . .	3
Mitigating Security Threats . . . . .	4
Selecting Authorization Mechanisms . . . . .	6
Checking at the Gate . . . . .	8
Using Roles for Authorization . . . . .	9
Designing Authentication for Authorization . . . . .	14
Performing Authorization Based on Windows Authentication . . . . .	15
Performing Authorization Based on Non-Windows Authentication . . . . .	17
Designing Identity Flow for Authorization . . . . .	19
Using Automatic Identity Flow . . . . .	19
Implementing Manual Identity Flow . . . . .	19
Performing Authorization in an Enterprise Application . . . . .	23
Performing Authorization in the User Interface Tier . . . . .	23
Performing Authorization in the Business Tier . . . . .	27
Performing Authorization in the Data Tier . . . . .	30
Creating Authorization Code with .NET Role-Based Security . . . . .	37
Performing Authorization Checks . . . . .	38
Separating Business Logic and Authorization Logic . . . . .	42
Handling Authorization Errors . . . . .	43
Performing Authorization with Multiple Threads . . . . .	45
Extending the Default Implementation . . . . .	46
Reusing Authorization Implementations . . . . .	48
Creating a Reusable Authorization Framework . . . . .	48
Securing the Implementation . . . . .	51
Improving the Performance of a Reusable Authorization Framework . . . . .	51
Appendix . . . . .	51
How to Enable Authorization in a .NET Remoting Component . . . . .	51
How to Perform Authorization in an XML Web Service . . . . .	55
How to Create an Authorization Custom Exception Type . . . . .	57
How to Change the Principal in an ASPNET Application . . . . .	60
How to Build a GenericPrincipal Using SQL Server . . . . .	61
How to Use System.EnterpriseServices COM+ Role-Based Security . . . . .	62
Feedback and Support . . . . .	65
Collaborators . . . . .	66
Additional Resources . . . . .	67



# Designing Application-Managed Authorization

## Summary

This guide provides guidelines for designing and coding application-managed authorization for single or multi-tier applications that are based on Microsoft® .NET. It focuses on common authorization tasks and scenarios, and it provides information that helps you choose the best approaches and techniques. This guide is intended for architects and developers.

This guide assumes that readers have a basic knowledge of topics such as Windows authentication and authorization, XML Web services, and .NET remoting. For more information about designing distributed .NET-based applications, see “Designing Applications and Services” in the MSDN® Library. For more information about designing security into distributed applications, see “Building Secure ASP.NET Applications” in the MSDN Library. For more general design guidance, see the .NET Architecture Center in Microsoft TechNet.

## Download

To download this guide in PDF format, go to <http://microsoft.com/downloads/details.aspx?FamilyId=40A58453-EC1B-4627-874B-F83437DBE00C&displaylang=en>

## Contents

This guide includes the following sections:

- Introduction
- Understanding Authorization
- Designing Authentication for Authorization
- Designing Identity Flow for Authorization
- Performing Authorization in an Enterprise Application
- Creating Authorization Code with .NET Role-Based Security
- Reusing Authorization Implementations
- Appendix

## Introduction

This guide describes how to perform authorization in .NET-based applications. It explains the term *authorization* and discusses several mechanisms for performing authorization. This guide also describes:

- Important concepts such as identities and principals.
- How to use role-based security to authorize a category of users who share the same security privileges.
- Significant differences between .NET and COM+ role-based security.

The specific authorization mechanism you adopt often depends on how you authenticate, or verify the identity of, users. This guide examines:

- The differences between Windows authentication and non-Windows authentication.
- How these authentication mechanisms affect authorization.
- How to flow identity information for authorization purposes to remote application tiers.

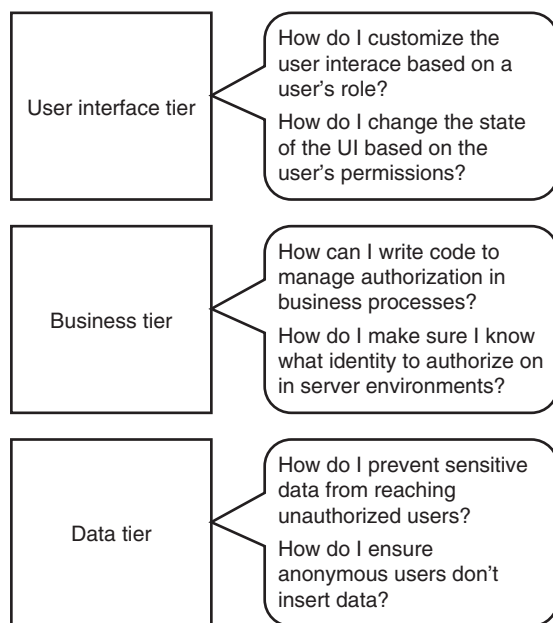
In a typical enterprise application, you need to perform different kinds of authorization at each tier in the application. To help you to identify the need for authorization in each tier and to choose appropriate authorization strategies in various scenarios, this guide describes typical authorization tasks in the user interface tier, the business tier, and the data tier. Figure 1 highlights some of the important authorization issues in each tier in an enterprise application.

The .NET Framework Class Library provides a variety of interfaces and classes that help you perform authorization with .NET role-based security. This guide describes:

- Several techniques for checking whether a user belongs to a particular role.
- How to handle authorization errors.
- Specific authorization issues that arise in multithreaded .NET applications.

Much of the effort you invest in defining an authorization framework can be reused across multiple applications. This guide concludes by describing:

- How to define a reusable authorization framework.
- Guidelines on how to maximize the security and performance of such a framework.



**Figure 1**

*Performing authorization in each tier in an enterprise application*

---

**Note:** This guide pertains to application managed authorization using features of the .NET Framework. The Authorization Manager API and Microsoft Management Console (MMC) snap-in available in the Microsoft Windows® .NET Server 2003 family of operating systems provide applications with a complete role-based access control framework. The Authorization Manager API, also known as AzMan, provides a simplified development model in which to manage flexible groups and business rules and to store authorization policies. For more information, see “Authorization Interfaces” and “Authorization Objects” in the MSDN Library.

---

## Understanding Authorization

*Authorization* is confirmation that an authenticated principal—a user, a computer, a network device, or an assembly—has permission to perform an operation. Protection allows only appointed users to perform certain actions, and it prevents malicious acts.

This section of the guide describes:

- The protection that authorization provides.
- Basic authorization.
- The authorization capabilities of the .NET Framework.

## Mitigating Security Threats

Authorization alone is not enough to secure an application; therefore, this guide briefly mentions the types of threats that face your application. Some of the common security threats are as follows; these threats are often referred to by the acronym *STRIDE*. They include:

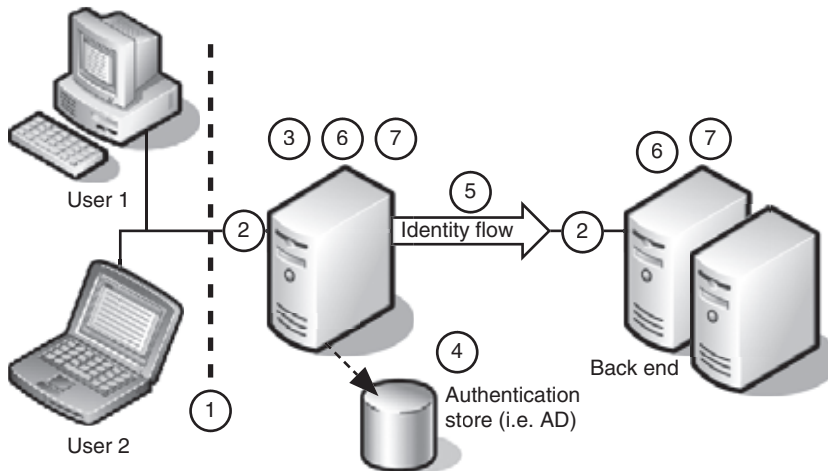
- **Spoofing identity**—An unauthorized user impersonating a valid user of the application
- **Tampering with data**—An attacker illegally changing or destroying data
- **Repudiability**—The ability of a user to deny that he or she performed an action
- **Information disclosure**—Sensitive data released to users or to locations that should not have access to it
- **Denial of service**—Acts of sabotage that make applications unavailable to users
- **Elevation of privilege**—A user illegally gaining an unacceptably high level of access to the application

You can use such techniques as the following to address STRIDE threats:

- **Authentication** — Strong authentication helps mitigate identity spoofing. When a user logs on to Windows or starts an application, he or she enters information in the form of *credentials*, such as a user name and password. Windows uses a protocol such as NTLM or Kerberos to validate the user's credentials and to log the user onto the system. Applications often use the product of a system logon or implement custom authentication as a basis for authorization. For more information about authentication, see "Building Secure ASP.NET Applications" in the MSDN Library.
- **Authorization**—Use the authorization techniques described in this guide to prevent data tampering and information disclosure threats.
- **Identity flow** — Applications deployed across multiple computers sometimes need to pass information representing the identity of the authenticated user among the computers of the system. Identity flow is typical when authentication occurs on the first computer and other application logic resides on a separate computer. For more information about identity flow, see "Designing Identity Flow for Authorization" later in this guide.
- **Auditing** — A record of both authorized and unauthorized operations reduces repudiability. This guide does not describe auditing in detail. For more information about auditing, see "Building Secure ASP.NET Applications" in the MSDN Library.

For more information about STRIDE, see “Designing for Securability” in the MSDN Library.

Figure 2 shows a model for how to mitigate STRIDE security threats in a multi-tiered application.



**Figure 2**

*Model of a secure multi-tiered application*

Figure 2 depicts a multiple physical tier deployment; however, many smaller applications are implemented on one physical tier, simplifying authentication, authorization, and identity flow. Figure 2 illustrates the following measures to mitigate security threats:

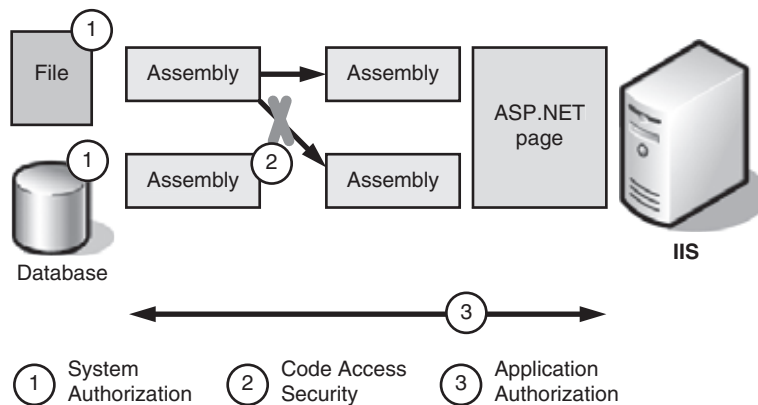
1. Bandwidth throttling reduces denial of service (DoS) attacks. This prevents applications from becoming swamped by continuous and undesired requests from malicious applications or users.
2. Encryption enables secure communication.
3. Authentication prevents identity spoofing.
4. Authentication verifies credentials against a data store.
5. Identity flows between the application tiers (optional).
6. Auditing enables repudiability.
7. Authorization prevents data tampering and exposure threats.

## Selecting Authorization Mechanisms

You can use various authorization mechanisms to control the functionality of your applications so that they behave as intended and cannot be misused either accidentally or deliberately. These authorization mechanisms fall into one of the following categories:

- **System Authorization**—Windows protects resources such as files and stored procedures with access control lists (ACLs). ACLs specify which users are allowed to access securable resources.
- **.NET Code Access Security Authorization**—Code access security authorizes code to perform actions based on the code's origin. For example, code access security determines which assemblies can access other assemblies in a .NET-based application, based on evidence criteria.
- **Application Authorization**—The application code itself authorizes user actions.

Use a combination of these approaches to create a secure application, as shown in Figure 3.



**Figure 3**  
*Selecting authorization mechanisms*

### System Authorization

*System authorization* is the process of setting resource permissions or ACLs for objects that the operating system controls, such as printers and files. System administrators maintain these settings. System authorization is a yes or no decision: A user is either authorized or not authorized to access the resource.

---

Examples of system authorization include:

- Authorization settings in Microsoft ASP.NET-based applications that limit access to URL files or paths specified in Web.config files.
- Permissions set in the Active Directory® directory service.
- NTFS file system access control entries.
- Message queuing permissions.
- Permissions granted within server products such as Microsoft SQL Server™. Such permissions might involve individual objects such as tables or views.

For more information about system level security and authorization, see “Building Secure ASP.NET Applications” in the MSDN Library.

System authorization applies constraints to individual objects; however, .NET Code Access Security Authorization is necessary to constrain the code.

### **.NET Code Access Security Authorization**

The .NET common language runtime uses code access security to constrain executable code. Code access security works by granting permissions (known as *permission sets*) to application code based on evidence. This evidence can include the code’s origin, its publisher; or other evidence, such as an assembly’s strong name.

Permission sets let you control what the application can perform, such as deleting files or accessing the Internet. For example, you can limit the application to use only isolated storage or to control access to printing.

Code access security takes the evidence into account regardless of the user’s identity. Even if a user with administrative privileges uses the application, the code access security permissions remain the same. For example, if the code comes from the Internet, limits—such as the capacity to delete files—apply regardless of who the user is.

Examples of code access security usage include:

- Preventing a downloaded component from performing dangerous actions by limiting its power and placing it in an isolated environment. Isolation helps prevent rogue code from compromising your system.
- Creating an isolated environment for hosted code running on a Web server or on an application server.
- Limiting the power of your components to prevent malicious code from misusing them.

---

**Note:** Use Caspol.exe or the Microsoft .NET Framework Configuration management console to configure code access security.

---

For more information about code access security, see the *.NET Framework Developer's Guide* article "Code Access Security" in the MSDN Library.

Code access security helps secure your system by checking permissions for the code, but depending on the application, you might also need to use application authorization to check permissions for the user.

## **Application Authorization**

Most applications implement different functionality or security permissions, depending on the user interacting with the system. Design *application authorization* to limit access to application resources or to implement business rules based on the user's role within the application.

The primary purpose of application authorization is to secure functionality and other intangible items, such as business logic. For this reason, application authorization is difficult to implement with current system-level technologies because these technologies tend to use settings, such as ACLs, on physical resources. For example, you might want to secure an operation that approves an expense claim from an employee, but there is no physical resource to secure; therefore, when you design application authorization, you should focus on high-level operations rather than on individual resources.

Application authorization provides an alternative approach to system authorization when system authorization mechanisms are too finely grained, or when they don't take into account application data or state. For example, system level security standards for XML Web services are still under development and are therefore still evolving. You do not need to wait until the standards are complete to add security to or create secure XML Web services. For XML Web services you build today, you can implement application authorization and use Secure Sockets Layer (SSL) or other combinations to secure calls to the service.

Examples of application authorization include:

- Checking whether a user has permissions to perform specific actions—for example, approving an expense claim—within the application.
- Checking whether a user has permission to access application resources—for example, retrieving sensitive columns from a database.

You will learn how to design and code these types of application authorization later in this guide.

## **Checking at the Gate**

To prevent an operation from needlessly continuing to the point of failure, always authorize a user's request as soon as you can. Each point of authorization is called a *gatekeeper*. Examples of gatekeeping mechanisms include file and URL authorization

in an ASP.NET gate. There may be several gatekeepers as the identity flows down through the tiers. Checking at the gate reduces the number of authorization checks necessary deeper in the system (past the entry point, or gate).

Objects that perform authorization checks deeper in the system have fewer requirements for logic that compensates for the authorization failure. The individual component is not responsible for dealing with authorization failure beyond raising an exception to inform the caller of the failure.

## Using Roles for Authorization

The .NET Framework provides a role-based application authorization capability. A *role* is a category or set of users who share the same security privileges.

Using roles rather than specific user identities provides the following benefits:

- You don't have to change your application when changes occur, such as users being added, being promoted, or leaving their jobs.
- Maintaining permissions for roles is easier than for individual users. For example, working with 10 roles is easier and less time-consuming than working with 120 users.
- A user can be a member of more than one role, allowing flexibility in how you assign and test permissions.

## Defining Roles Based on the Business Organization

Roles can represent the position the user holds in an organization; for example:

- Manager
- Employee
- ClaimApprovalDepartment

A benefit to this approach is that the information is generally retrieved from a store such as in Active Directory. Often these roles are useful in modeling actual business requirements.

## Achieving Independence from Organizational Changes

You can also use roles to indicate the types of operations that a user performs for his or her job. Such roles allow you to link application functions to individual users; for example:

- CanApproveClaim
- CanAccessLab
- CanViewBenefits

This second approach is flexible because you can design the roles around application functionality rather than around organizational structures. It might be harder to maintain because of a lack of infrastructure in which to store these roles. An application needs a combination of approaches in most cases.

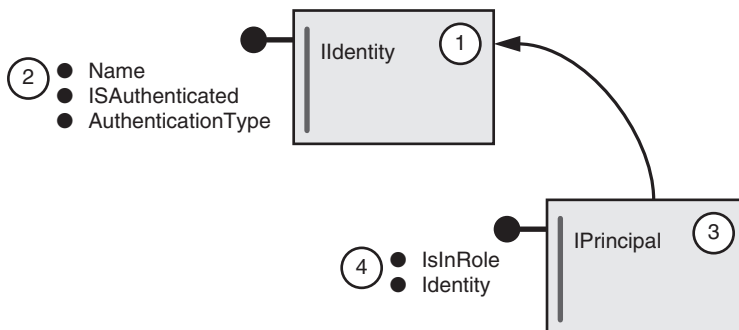
### Performing Authorization Without Using Roles

Sometimes you must base authorization on who the user is rather than on the roles he or she plays within the application. For example, you might allow only a direct manager to approve an employee's expense claim. You can achieve this finer level of authorization by comparing the current user to the manager of the employee who filed the claim.

### Using Role-Based Security in .NET-Based Applications

The .NET Framework provides a role-based security implementation in the `System.Security.Principal` namespace, which you use for application authorization. To work with application authorization in the .NET Framework, create `IIdentity` and `IPrincipal` objects to represent users. An `IIdentity` encapsulates an authenticated user. An `IPrincipal` is a combination of the identity of the user and any roles he or she has.

Figure 4 shows the relationship between `IIdentity` and `IPrincipal` objects.



**Figure 4**

*Relationship between `IIdentity` and `IPrincipal` objects*

Notice the following points in Figure 4:

1. An `IIdentity` object is an instance of a class that implements `IIdentity`. An `IIdentity` object represents a particular user.
2. The `IIdentity` interface has the properties `Name`, `IsAuthenticated`, and `AuthenticationType`. Classes that implement `IIdentity` often contain additional private members specific to their purpose. For example, the `WindowsIdentity` class encapsulates the account token for the user on whose behalf the code is running.

3. An **IPrincipal** object is an instance of a class that implements **IPrincipal**. The **IPrincipal** object is a combination of an **IIdentity** representing the user and any roles he or she has. This allows for a separation of authentication and authorization capabilities.
4. The **IPrincipal** object is used to perform authorization with the **IsInRole** method and provides access to the **IIdentity** object through the **Identity** property.

### Working with Identities

The .NET Framework provides four classes that implement the **IIdentity** interface:

- **WindowsIdentity**
- **GenericIdentity**
- **PassportIdentity**
- **FormsIdentity**

Each class enables you to work with different kinds of user identities. To access the current **WindowsIdentity** object for an application that uses Windows authentication, use the static **GetCurrent** method of the **WindowsIdentity** class, as the following code shows:

```
WindowsIdentity currentIdentity = WindowsIdentity.GetCurrent();
```

You can also create custom identity classes by implementing the **IIdentity** interface in your own custom class. For more information about creating custom identities, see “Extending the Default Implementation” later in this guide. For more information about how to work with the default **IIdentity** implementations, see “Designing Authentication for Authorization” later in this guide.

### Working with Principals

The .NET Framework provides the **IPrincipal** interface to link user roles and identities. All managed code that performs application authorization should use an object of a class that implements **IPrincipal**. For example, the **WindowsPrincipal** and **GenericPrincipal** classes provide inbuilt implementations of **IPrincipal**. Alternatively, you can create your own custom principal classes based on **IPrincipal**.

To make your coding more efficient, link the **IPrincipal** object to the current thread by using techniques described in the [Designing Authentication for Authorization](#) section later in this guide. Linking the **IPrincipal** object to the thread provides the current thread easy access to the **IPrincipal** object by using the static **CurrentPrincipal** property of the **Thread** object, as the following code shows:

```
WindowsPrincipal currentPrincipal = (WindowsPrincipal) Thread.CurrentPrincipal;
```

You can then perform an authorization check by testing whether the user is a member of a particular role. You do so by using the **IsInRole** method of the **IPrincipal** interface, as the following code shows:

```
bool roleMemberFlag = Thread.CurrentPrincipal.IsInRole( "CanApproveClaims" );
```

ASP.NET applications handle **IPrincipal** objects differently than do other .NET-based applications. ASP.NET creates the appearance of a session over the stateless HTTP protocol. As a part of this session, the **IPrincipal** object representing the user is available from the **User** property of the **HttpContext** object for all code executing the user's request. The common language runtime automatically updates **Thread.CurrentPrincipal** with the **HttpContext.User** value after the **OnAuthenticate** event of the **Global.asax** file.

ASP.NET applications often use the **User** property to perform authorization checks, as the following code shows:

```
bool roleMemberFlag = Context.User.IsInRole( "CanApproveClaims" );
```

---

**Note:** Manually changing **HttpContext.User** automatically updates the **Thread.CurrentPrincipal** for all threads executing within the same HTTP context. However, changing the **Thread.CurrentPrincipal** does not affect the **HttpContext.User** property. It affects only the chosen thread for the remainder of the request.

---

For more information about creating your own **IPrincipal** types, see “Extending the Default Implementation” later in this guide. For more information about how to work with the default **IPrincipal** implementations, see “Designing Authentication for Authorization” later in this guide.

### Granting Permissions to Work with **Identity** and **IPrincipal** Objects

The ability to work with **Identity** objects is a sensitive operation because user-related information is available. Allowing applications to change the current **IPrincipal** object should also be protected because application authorization capability is based on the current principal. The framework provides this protection by requiring that these operations have a code access security permission. Grant the **SecurityPermissionAttribute.ControlPrincipal** permission to applications that need to manipulate these objects by using **Caspol.exe** or the .NET Framework Configuration tool.

By default, all locally installed applications have this permission because they run under the Full Trust permission set.

Executing the following methods require the **ControlPrincipal** permission:

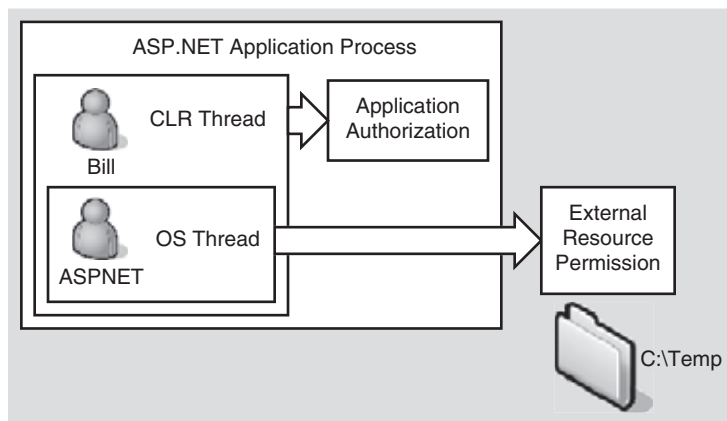
- **AppDomain.SetThreadPrincipalPolicy()**
- **WindowsIdentity.GetCurrent()**
- **WindowsIdentity.Impersonate()**
- **Thread.CurrentPrincipal()**

For more information about setting security permissions using CASPOL, see “Configuring Security Policy Using the Code Access Security Policy Tool (Caspol.exe)” in the MSDN Library.

### Managing Authorization Differences Between Windows and the Common Language Runtime

The common language runtime has a separate security infrastructure on top of the Windows security infrastructure. A Windows thread has a token for a Windows authenticated user, whereas the runtime thread has an **IPrincipal** object to represent the user.

When developing your code, you must therefore consider at least two security contexts that represent a user. For example, consider an ASP.NET-based application using forms authentication: The ASP.NET process runs under a Windows service account (a user account created specifically for an application) named *ASPNET* by default. Imagine a user named *Bill* logs on to the Web site. The **Thread.CurrentPrincipal** property represents the Forms authenticated user, *Bill*. The common language runtime sees the thread running as *Bill*, so any managed code sees *Bill* as the principal. If the managed code requests access to a system resource such as a file, the unmanaged code sees the *ASPNET* Windows service account regardless of the **Thread.CurrentPrincipal** value. Figure 5 represents these relationships.



**Figure 5**

*The authorization relationship between common language runtime and operating system threads*

Impersonation allows you to change which user account the operating system thread executes under in its interactions with Windows. You can impersonate a Windows identity by calling the **WindowsIdentity.Impersonate** method. This form of impersonation allows you to access local resources as a particular user. When you call the **Impersonate** method, you will be logged on as the user that the **WindowsIdentity** represents. You must have access to the user's credentials on the server and call the **LogonUser** API. However, creating the **WindowsIdentity** manually can be complex, so you should perform impersonation only when it is absolutely necessary.

The **WindowsImpersonationContext.Undo** method reverts a user's identity back to the original identity after the application code impersonates another user. These functions are generally called in a matched pair as the following code shows:

```
// Impersonate a Windows identity
WindowsImpersonationContext impersonationCtx =
    WindowsIdentity.Impersonate(userToken);

// Do something under the context of the impersonated user

// Revert to the user's real identity
impersonationCtx.Undo();
```

For more information about impersonating a Windows account, see “Impersonating and Reverting” in the MSDN Library.

---

**Note:** On Windows 2000, a process must have the SE\_TCB\_NAME privilege to call **LogonUser** API. For more information about validating user credentials, see article Q180548, “HOWTO: Validate User Credentials on Microsoft Operating Systems,” in the Microsoft Knowledge Base.

---

## Designing Authentication for Authorization

Authorization depends on authentication—that is, a user or process must be authenticated before it can be authorized to view or use protected resources. This section of the guide examines two authentication mechanisms and explains how each affects authorization:

- Performing authorization based on Windows authentication
- Performing authorization based on non-Windows authentication

Your choice of authentication mechanism is often influenced by factors unrelated to authorization, such as the availability of Windows user accounts and general environmental considerations, including the client's browser type. However, the authentication choice you make does affect the way you perform your authorization checks.

---

## Performing Authorization Based on Windows Authentication

All applications execute under a Windows user account known as the *Windows identity*. During Windows authentication, you use this Windows user account or the role to which the Windows user belongs to perform your authorization checks. Windows chooses the user account based on credentials supplied by one of the following techniques:

- Using information supplied by the user when he or she logged on to the computer
- Using a service account that you created especially for the application by using configuration tools such as the Component Services management utility

The overall process for using authorization with Windows authentication is:

1. Windows uses NTLM or Kerberos to validate the user's credentials. For more information about using Kerberos delegation, see "How To: Implement Kerberos Delegation for Windows 2000" in "Building Secure ASP.NET Applications."
2. The successful logon produces a Windows access token, which a **WindowsIdentity** encapsulates for .NET applications.
3. Usually, the runtime automatically sets the **Thread.CurrentPrincipal** property to a **WindowsPrincipal** object that contains the Windows notion of roles; however, in some situations you must set it yourself as described later in this section.
4. Authorization can now take place using **Thread.CurrentPrincipal**.
5. When you call application logic remotely, there are some situations in which you need to manually provide the identity to the callee. This issue is described in greater detail in the Implementing Manual Identity Flow section later in this guide.

---

**Note:** Link the **IPrincipal** object to the thread, and then retrieve the principal or identity information from the **Thread.CurrentPrincipal** property. This allows you to maintain the role collection in memory and therefore reduce how often you need to look up this information.

---

### Performing Authorization by Using Windows Roles

You can check to see whether a user is a member of specific Windows user groups (such as "<domain>\Users") by using a **WindowsPrincipal** object. The **WindowsPrincipal** object has an **Identity** property, which returns a **WindowsIdentity** object that describes the identity of the current user.

You can configure .NET-based applications hosted within ASP.NET to have automatic access to a **WindowsPrincipal** from the **Thread.CurrentPrincipal** property by enabling Windows authentication in Web.config. You can use this technique in ASP.NET Web applications, XML Web Services, and .NET remoting objects hosted within ASP.NET.

Other .NET-based applications, such as Windows services, console, and Windows Forms-based applications, require you to establish **Thread.CurrentPrincipal** using one of the following approaches:

- Calling **SetPrincipalPolicy**
- Setting **CurrentPrincipal**

---

**Note:** Code that works with **IIdentity** and **IPrincipal** objects requires the **ControlPrincipal** permission. Use code access security to grant this permission to the application. For more information, see “Using Role-Based Security in .NET-Based Applications” earlier in this guide.

---

### **Calling SetPrincipalPolicy**

The **SetPrincipalPolicy** method of the **AppDomain** object links specific types of **IPrincipal** objects to the application domain. Use the **WindowsPrincipal** value of the **PrincipalPolicy** enumeration to link the current **WindowsPrincipal** object to the application threads, as the following code shows:

```
AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
```

The principal is then available for authorization checks. Call **SetPrincipalPolicy** at the beginning of your application’s execution to ensure that the **IPrincipal** object is linked to the threads before you perform any authorization checks.

Do not use **SetPrincipalPolicy** within ASP.NET applications because ASP.NET authentication manages this value for you during the **Application\_AuthenticateRequest** event handler defined in **Global.asax**.

---

**Note:** **SetPrincipalPolicy** is effective only if no default principal exists for the application domain.

---

### **Setting CurrentPrincipal**

To set the **CurrentPrincipal** property, you must first create a new **WindowsPrincipal** object by passing a **WindowsIdentity** object to the **WindowsPrincipal** constructor. You can then link the new **WindowsPrincipal** to the **Thread.CurrentPrincipal** property as the following code shows:

```
Thread.CurrentPrincipal = new WindowsPrincipal(WindowsIdentity.GetCurrent());
```

The **IPrincipal** object is then available for authorization checks.

For information about how to perform authorization checks with a **WindowsPrincipal**, see “Performing Authorization Checks” later in this guide.

## Performing Authorization by Using Application-Defined Roles

To create authorization checks based on application-defined roles (such as `CanApproveClaims`) you must manually create a generic or custom **IPrincipal** object. You do not need to call the `SetPrincipalPolicy` method because its default setting is **NoPrincipal**, which is the correct setting for assigning your own principal.

Most of the time, create a **GenericIdentity** object based on the name of the authenticated user. (To get this name, use the `Name` property of the **IIdentity** interface.) This avoids any security issues involving the user's logon token—accessible using a **WindowsIdentity** object—when presenting the **IIdentity** object to other components. You then create a **GenericPrincipal** that links the **IIdentity** object to your list of application-defined roles.

The following code shows how to create **GenericIdentity** and **GenericPrincipal** objects when you use Windows authentication:

```
GenericIdentity identity = new GenericIdentity(WindowsIdentity.GetCurrent().Name);
Thread.CurrentPrincipal = new GenericPrincipal(identity,
    new string[] {"CanApproveClaims", "User"});
```

In most cases, retrieve the application-defined roles from a data store rather than hard-coding the roles. This allows you greater flexibility in assigning role membership. For an example of this approach, see “How to Build a **GenericPrincipal** Using SQL Server” in the Appendix.

For information about how to perform authorization checks with a **GenericPrincipal**, see “Performing Authorization Checks” later in this guide.

## Performing Authorization Based on Non-Windows Authentication

You might not want to or be able to use Windows authentication for application authorization. This could be because the users do not have Windows user accounts or because technical constraints keep you from using Windows authentication to validate the user's credentials.

Non-Windows authentication is the process of verifying the user's credentials using a technology other than Windows. This type of authentication includes those provided to you by the .NET Framework, such as ASP.NET Forms authentication and those you create yourself.

The overall process for using authorization with non-Windows authentication is:

1. The application gathers the user credentials.
2. The application verifies the credentials, typically against a custom data store such as a SQL Server database.

3. Initially, the **Thread.CurrentPrincipal** property is set to a **GenericPrincipal** object that does not contain any roles. You replace **Thread.CurrentPrincipal** with a new **GenericPrincipal** object, or with an object of a custom class that implements **IPrincipal** to provide application-defined roles.
4. Authorization can now take place using **Thread.CurrentPrincipal**.
5. When calling application logic remotely, you might need to manually provide the identity to the callee.

Common types of non-Windows authentication include:

- **ASP.NET Forms authentication**—ASP.NET Forms authentication automatically creates a **FormsIdentity** object that represents the validated identity. It contains a **FormsAuthenticationTicket**, which contains information about the user's authentication session.

The Forms authentication provider creates a **GenericPrincipal** based on the **FormsIdentity** object but with no role membership.

For information about how to use ASP.NET Forms authentication, see article Q301240, "HOW TO: Implement Forms-Based Authentication in Your ASP.NET Application by Using C# .NET," in the Microsoft Knowledge Base.

Forms authentication almost always uses credentials that are specific to your application. For an example of how to verify Windows credentials obtained using forms authentication, see article Q316748, "HOW TO: Authenticate Against the Active Directory by Using Forms Authentication and Visual C# .NET," in the Microsoft Knowledge Base.

- **Passport authentication**—ASP.NET applications can use .NET Passport to perform authentication. Passport authentication produces a **PassportIdentity** object to represent a user identity. The **PassportIdentity** object extends the basic **IIdentity** interface to include Passport profile information.

The Passport authentication provider creates a **GenericPrincipal** based on the **PassportIdentity** object but with no role membership.

For more information about Passport authentication, see "The Passport Authentication Provider" in the MSDN Library.

- **ISAPI authentication solutions**—An alternative approach is to implement a manual or custom authentication mechanism using HTTP headers such as the authentication mechanisms found in Microsoft Commerce Server. For more information about security in Microsoft Commerce Server, see "Commerce Server Security" in the MSDN Library.

The HTTP transport provides information (such as a cookie) for use in constructing a generic or custom **IIdentity** object and a generic or custom **IPrincipal** object.

---

For more information about the authentication techniques available in ASP.NET, see “Authentication in ASP.NET: .NET Security Guidance” in the MSDN Library.

All the previously listed types of non-Windows authentication involve **GenericPrincipal** objects or custom **IPrincipal** objects. For an example of how to link a **GenericPrincipal** to application defined roles, see “How to Build a GenericPrincipal Using SQL Server” in the Appendix.

## Designing Identity Flow for Authorization

Authentication creates **IIdentity** and **IPrincipal** objects for authorization purposes and determines how you can programmatically pass the identity information to application logic deployed remotely on other computers. The propagation of an authenticated identity is known as *identity flow*. There are two ways to achieve identity flow:

- Automatic identity flow
- Manual identity flow

### Using Automatic Identity Flow

The common language runtime automatically provides identity flow when all of the code that requires the identity executes in the same context. The caller and callee are in the same context when they share the same application domain. If the client code (referred to as the *caller*) and the component being called (referred to as the *callee*) are running within the same context, .NET automatically uses the same **Thread.CurrentPrincipal** object for both the caller and the callee. For cases in which the callee and caller execute on different threads, see “Performing Authorization with Multiple Threads” later in this chapter.

Examples of code executing within a single application domain include:

- An ASP.NET Web site that contains all of the required code within in-process components and has no remotely deployed logic.
- A standalone Windows Forms-based application with no remotely deployed logic.

### Implementing Manual Identity Flow

Implement manual identity flow when authorization checks occur in a remote tier that cannot perform authentication because of technical constraints. An example of this situation is calling a remote component using .NET remoting over the TCP channel. For more information about using channels in .NET Remoting applications, see “Microsoft .NET Remoting: A Technical Overview” in the MSDN Library.

If you are implementing your own identity flow, pass the data in a manner that differs from the way you call your code (known as *out-of-band*). If your component interface involves calling functions and passing data using parameters, plan to send identity information some other way.

For example, if you are calling an XML Web Service using SOAP, don't send the identity information as part of the message body; put it in a custom SOAP header. This is not so much a security consideration as a code design issue. A separation like this enables extensibility, reuse, and aggregation of your code interfaces. This also allows you to change the way you send the identity information as technological standards evolve without changing your interface contract. The Web Services Security (WS-Security) specification demonstrates this.

Note that identity flow is different from passing credentials (which is effectively a reauthentication on the server). Alternatively, you can pass the credentials within the message itself—as long as you encrypt the credentials. However, the encryption would involve using a shared secret key or a public key system, in which case passing the credentials in this manner isn't much of an advantage.

Following are some best practices for implementing identity flow:

- Use the strongest authentication available within the functional requirements and within technological constraints.
- Some identity flow techniques involve storing and/or passing secret information that is critical for the security of the identity flow mechanism. Encrypt the secret information or use a secure channel to keep the information safe. The classes in the **System.Security.Cryptography** namespace might be helpful. Also, the “How To: Create a DPAPI Library” sample in Building Secure ASP.NET Applications may be helpful.
- Store secret information securely using a technology such as the Data Protection API (DPAPI). For information about DPAPI, see “Windows Data Protection” in the MSDN Library.
- Ensure that your implementation allows you to use auditing to the depth required. For information on auditing, see “Monitoring and Auditing for End Systems” in Microsoft TechNet.

### **Passing Identity Information in a Trusted Environment**

In a trusted environment, the callee leaves the responsibility of authentication to the caller. Any information passed from the caller to the callee is considered to be safe and reliable—for example, when a Web tier is the caller and a component on an internal application farm is the callee.

---

**Note:** This can be an insecure solution if it is possible for a rogue user to invoke code passing in another (and potentially fake) identity. To protect against these types of spoofing attacks, use technologies provided by the system layer to secure the communication channels with Secure Sockets Layer (SSL) or IP Security (IPSec), and use code access security to ensure that only appropriate code calls your functions.

---

There are two main ways to pass identity information between your application tiers in a trusted environment:

- **Passing only the user name**—The caller can pass only the user name to the callee. The callee can then query the authorization store to create a generic or custom **IPrincipal** object and attach it to the thread, as described earlier in this guide.

This approach is also used when you want to flow the identity to the database for auditing. For more information, see “Performing Authorization in the Data Tier” later in this guide.

This approach provides flexibility, because you are not forcing callers to pass a particular type of object or extra information, such as the application roles.

- **Passing the roles (using an IPrincipal object)**— After authenticating the user, the caller can pass the roles to the callee by serializing the generic or custom **IPrincipal** object. The callee can then deserialize the object and attach it to the thread, as described earlier in this guide. This provides a similar effect to what happens automatically in a single application domain.

Passing a custom **IPrincipal** object allows you to pass the roles and any extra information that a custom **IPrincipal** object could contain (such as user profile information), as well as the identity.

However, you cannot pass **IPrincipal** objects that encapsulate certain types of information. For example, you cannot pass a **WindowsPrincipal** or any other principal based on a **WindowsIdentity** object because the identity maintains a token that makes no sense outside of the original environment.

For more information about serializing an **IPrincipal** object, see “How to Enable Authorization in a .NET Remoting Component” in the Appendix.

---

**Note:** Passing identity information is acceptable only if you are within a trusted environment using a secured communication channel, such as a corporate LAN protected from the outside by a firewall. Protect yourself from spoofs caused by intercepting or modifying network data on the wire.

---

## Passing Identity Information in a Non-Trusted Environment

A non-trusted environment is one in which the caller does not trust the callee. In a non-trusted environment, the callee must validate the caller before performing authorization—for example, a Web application validating each request by a browser.

There are two common ways to pass identity information between your application tiers in a non-trusted environment:

- **Passing signed data**—Passing cryptographically signed data using public key infrastructure can be an effective means of identity flow. You typically use this approach with asynchronous communication because there is no session for authentication.

Passing signed data works by including a digital signature with the message. The digital signature is computed using the private key of the sender and the contents of the message. To verify, the callee obtains the public key of the sender through a key management system and uses this key to determine that the sender used the corresponding private key when signing the message.

The .NET Framework provides the **SignedXml** class as an implementation of the XMLDSIG standard. This standard doesn't address issues such as key management or the trusting of keys. A receiver can make no assumptions about the trustworthiness of the keys used in this process. This is the responsibility of your application code. For details about the XMLDSIG standard, see "XML-Signature Syntax and Processing" on the W3C Web site.

For sample code showing how to use the **SignedXml** class, see the CAPICOM SDK; CAPICOM is a Microsoft ActiveX® control that provides a COM interface to Microsoft CryptoAPI. This sample code illustrates the full round trip of using a certificate authority, signing, and verifying an XMLDSIG signature. You can find the sample code in the Platform SDK Redistributable: CAPICOM 2.0 in the `samples\c_sharp\xmldsig` directory. For more information about CAPICOM, see the "CAPICOM Reference" in the MSDN library.

- **Passing a token**—Passing a token involves passing a piece of information that the callee verifies to determine who the user is.

Passing a token requires a shared authentication mechanism or a way to determine whether the token is valid. Passing a token might include passing a Kerberos ticket between components. Another example is a session cookie for an e-commerce Web site that an Internet Information Services (IIS) application creates upon authentication. The cookie passes back to the Web server each time the browser makes a call. This is typical of ISAPI solutions.

Now that you understand how authorization works with the various authentication mechanisms and how to pass authorization data across tiers, you need to see how authorization works in each of the three tiers in an enterprise application.

## Performing Authorization in an Enterprise Application

Most of today's applications benefit from a multi-tier design because multiple tiers provide scalability, flexibility, and performance enhancements. Although the purpose of authorization is essentially the same within each of the application tiers—that is, to control user access to system functionality and data—you design and implement authorization differently across the tiers. The three tiers in an enterprise application are the:

- **User interface tier**
- **Business tier**
- **Data tier**

### Performing Authorization in the User Interface Tier

This section describes how to perform authorization in the user interface tier. The following topics are included later in the guide: Performing Authorization in the Business Tier and Performing Authorization in the Data Tier.

Performing application authorization in the user interface tier helps to ensure that only authorized users can view or alter data or perform business functions that are restricted to specific jobs (for example, salary-related activities). Authorizing users in the user interface tier is the first opportunity you have to restrict access to processes that require authorization in other tiers within the system.

Create access checks in the user interface tier when you need to:

- Enable and disable or show and hide controls based on the user's role membership. For more information, see "Altering the Controls on a Form" later in this section.
- Change the flow from one form (or page) to the next, based on the user's role membership. For more information, see "Altering the Page Flow" later in this section.

Follow these best practices when you create authorization code in the user interface tier:

- Configure system authorization to control entry to the user interface whenever possible. For example, configure the Web.config file in an ASP.NET application to grant access to the entry page to only authorized users. For information about ASP.NET URL authorization, see "Building Secure ASP.NET Applications" in the MSDN Library.

- Defend against the user accessing the user interface elements at an incorrect stage in the process or task by authorizing the user when each form or page loads. For example, a user may need to fill out multiple Web pages in an ASP.NET application before an operation can be completed. Allowing the user to type a URL directly into the browser may allow the user to access pages he or she is not entitled to view at a certain stage. Therefore, you should perform application authorization checks when each page loads.
- Do not base the security of your system exclusively on application authorization within the user interface. Perform additional access checks at the business or data tier because you might call these components in several ways or from different applications.
- If the user interface limits the data the user can view, don't read unnecessary data from the data source; this prevents any potential leak of the restricted data as it travels around the network.

---

**Note:** All code samples in the following topics show Web Forms for the examples. You can modify the code to work in Windows Forms by changing **User.IsInRole** to **Thread.CurrentPrincipal.IsInRole**. For more information about the **IsInRole** method and performing authorization checks, see “Performing Authorization Checks” later in this guide.

---

### Altering the Controls on a Form

In both Windows forms and ASP.NET applications, you may need to alter the way that you display controls in the user interface based on authorization. These changes include changing the visibility of some controls or disabling them.

For example, in an employee expense claims application, members of the `CanApproveClaims` role need a button in the form for approving claims. All users not in this role do not see the **Approve** button. The following code demonstrates this approach:

```
if(User.IsInRole( "CanApproveClaims" ))
    cmdApprove.Visible = true;
else
    cmdApprove.Visible = false;
```

Such simplistic approaches can result in long “if-then-else” chains when you create authorization code for complicated user interfaces or for interfaces with many role customizations.

Use the following techniques to lessen the complexity of displaying controls based on authorization checks:

- **Consolidate the logic**—Place the logic for altering controls inside a private method that you call from within the form's **Load** event. Therefore, you have only one location to maintain the form authorization logic related to displaying controls.

The following code shows how you can apply this approach to a Web form:

```
private void Page_Load(object sender, System.EventArgs e)
{
    ConfigureControls();
}

private void ConfigureControls()
{
    // Configure the form controls
}
```

- **Use a repeater control where possible**—In a situation in which a control corresponds to multiple lines of data, use a repeater control so that you can modify all the controls settings at one time.

For example, if you create a Web form that uses a *DataGrid*, you can use a templated column in the form of a check box to allow authorized users to approve multiple claims at the same time.

The following code shows how you can use this technique to display an *Approve* column in a *DataGrid*.

```
private void ConfigureControls()
{
    // Modify the visibility of the templated check box column
    // based on the role of the user
    if (User.IsInRole( "CanApproveClaims" ))
        ClaimGrid.Columns[3].Visible = true;
    else
        ClaimGrid.Columns[3].Visible = false;
}
```

- **Use a Model View Controller pattern**— The Model View Controller pattern helps you separate the logic that controls the behavior of the user interface elements from the events fired by those controls. In the claims example, a form may have a list box of employees and a **Show Claims** button. You can enforce a clear separation of logic by putting all the code that should execute when *ShowClaims* is pressed in another function (called a controller function), or even in another class. The list box acts as a view on the employee data, and the **Show**

**Claims** button handles user actions (such as a *click*) and calls the related controller function. The controller function is a method on your Web or Windows form that contains the logic to change the state of the business data and user interface. Place specific portions of your authorization code that change control state after the form is loaded in these controller functions rather than in the event handler.

- **Use separate forms for each role**—Create separate forms for each role if your number of roles is low and each role requires a significantly different version of a form. Test for role membership in each form and redirect to the correct form. This may add significant test effort—using the Model View Controller pattern as described earlier may help you factor out and reuse your presentation code.

### **Altering the Page Flow**

Sometimes—such as in a typical wizard user interface—you want the user to complete multiple forms (or pages) to perform a business process, depending on his or her role membership. This approach can change the page order or *flow*.

In the claims example, a GeneralManager member can view and approve claims from any employee in any department; however, a StoreManager member can view claims from only employees in his or her department. To allow the GeneralManager member to choose the department, create an extra step to retrieve this information before displaying the employee selection form.

The following code shows one way to create this type of page flow. The code uses the ASP.NET **Server.Transfer** method to transfer control to a different page. The calls to **Server.Transfer** are wrapped within a private method named `DisplayNextPage`:

```
private void btnView_Click(object sender, System.EventArgs e)
{
    DisplayNextPage();
}

private void DisplayNextPage()
{
    if (User.IsInRole( "GeneralManager" ))
        Server.Transfer( "ChooseDepartment.aspx" );
    else if (User.IsInRole( "StoreManager" ))
        Server.Transfer( "ChooseEmployee.aspx" );
}
```

## Performing Authorization in the Business Tier

Business processes range from fairly simple business logic to larger, long-running workflows consisting of many trusted and nontrusted partners. Your business tier code must ensure that all client requests meet the authorization and logic rules that the organization prescribes.

Create access checks in the business tier when you need to:

- Authorize an operation that performs a business process.
- Authorize a call from an external, nontrusted source before calling an internal business component. To do so could involve establishing an **IPrincipal** object for use by your business process in authorization. Use a service interface to manipulate the identity as it flows into your process, authorize a call to an internal business component, or both.
- Authorize calls to other parts of your distributed system or external services that are not part of your application. To do so could involve manipulating the identity flow out from your component. Use a service agent to make sure the current user can perform the external call and/or manipulate the identity as it flows out of your process.

Follow these best practices when creating authorization code in the business tier:

- Factor authorization capabilities into framework and utility components such as custom **IPrincipal** objects. This allows you to create authorization capabilities independent of the business logic.
- Perform authorization checks at the beginning of high-level processes. This allows you to configure authorization in fewer places, thus easing maintenance. However, to produce a secure solution, perform authorization within each publicly accessible method because calls to these methods may not have passed an authorization check.

## Using Role-Based Security in .NET vs. COM+

COM+ is best known for providing your applications with enterprise features such as transactional components, asynchronous activation, and object pooling. If you require any of these features in your .NET enterprise applications, you need to use COM+ rather than .NET role-based security. The COM+ role-based security system and the .NET framework role-based security do not interoperate. Therefore, you must decide early in the design which components will reside in COM+ and which will be .NET types; it will be hard to change the authorization code later.

For an example of how to target both the .NET role-based security system and the COM+ role-based security system, see “Unify the Role-Based Security Models for Enterprise and Application Domains with .NET” in MSDN Magazine.

In addition to role-based security, COM+ has built-in capability to flow the Windows user context and information about previous callers.

---

**Note:** Mixing COM+ and .NET role-based security in your components is not a safe practice. Windows switches between managed and unmanaged code when executing managed COM+ components, which makes principal information unreliable.

---

For code samples of COM+ role-based security, see “How to Use System.EnterpriseServices COM+ Role-Based Security” in the Appendix. For more information about using COM+ within .NET-based applications, see “Understanding Enterprise Services (COM+) in .NET” in the MSDN Library.

### **Performing Authorization in a Business Logic Component**

Authorization checks are often required in the components that contain the business logic. You can call these components directly from the code in the user interface tier by using .NET remoting or by using service interfaces that encapsulate the business components, such as XML Web services.

The authorization logic that you create frequently blends in with and becomes part of your component business logic. For information about how to separate business logic and authorization logic, see “Separating Business Logic and Authorization Logic” later in this guide. For information about how to create your authorization checks, see “Creating Authorization Code with .NET Role-Based Security” later in this guide.

### **Performing Authorization in Entities**

An entity is a representation of data that may also include some form of behavior. You can use numerous kinds of entity such as DataSets, XML strings, XML Document Object Models (XML DOMs), and custom entity classes in your application, depending on the physical and logical design constraints of the application. The application both produces and consumes these entities.

Entities frequently travel between application tiers, including complete round trips in which the data is retrieved from the data source, sent to the user interface, and returned to the data source for updating.

You can create classes to represent entities that can perform authorization checks. For example, only users belonging to certain roles can create and initialize an entity object.

---

However, in most cases, you should not perform application authorization in entities because:

- Pure data implementations such as DataSets, XML strings, and XML DOMs provide no inbuilt way to perform application authorization checks.
- Entities—including entities provided by a third party—can be highly mobile. You cannot be sure that client applications will be able to use the entity authorization features.

### **Performing Authorization in a Service Interface**

Service interfaces are façades that expose internal business logic to the outside world. Service interfaces provide an entry point to a business process, and they give you the flexibility to change the method signature of the internal process without affecting the outside caller.

Service interfaces can provide entry for nontrusted parties or for parties that require authentication and authorization before they allow access to the internal business component.

In the expense claim example, a service interface allows one claims processing component to be accessible to nontrusted clients by using an XML Web service and for trusted clients by using either .NET remoting or Distributed COM (DCOM). Service interfaces may also be custom Message Queuing (MSMQ) listeners.

For sample code that demonstrates authorization in an XML Web Service, see “How to Perform Authorization in an XML Web Service” in the Appendix.

### **Performing Authorization in a Service Agent**

A service agent is a bridge between business logic and an external service. XML Web Services and MSMQ message queues are two examples of external service agents.

You can use service agents to control access to external services by authorizing attempts to call the services from within your business tier. Service agents act as a proxy to an external service to allow you to perform validations, authorization, data formatting, and other tasks when your code calls the external service. Using a service agent also allows you to change which external service is being called and its signature, without affecting your business logic. You can write code in authorization service agents to prevent unauthorized users from calling external services.

---

**Note:** This type of authorization is separate from authorization that occurs in the external service. If you maintain the external service, perform the authorization check within the callee anyway, to ensure that the system is secure.

---

In the following service agent code, a user cannot pass a claimXML message to the ApproveClaim XML Web service method if he or she does not belong to the CanApproveClaims role.

```
public ClaimResponse ApproveClaim(string claimXML)
{
    if (Thread.CurrentPrincipal.IsInRole( "CanApproveClaims" ))
    {
        WebService.Claim claimService = new WebService.Claim();
        return claimService.ApproveClaim(claimXML);
    }
    else
        throw new SecurityException(
            "You are not authorized to approve claims.");
}
```

For more information about dealing with exceptions in authorization code, see “Handling Authorization Errors” later in this guide.

If the external service requires a specific user identity for authorization, create code within the service agent to pass an acceptable **IIdentity** object. Doing so changes the identity flow out of the component. For more information about manipulating identity flow, see “Designing Identity Flow for Authorization” earlier in this guide.

## Performing Authorization in the Data Tier

The data tier is the last line of defense against illegitimate requests from the user interface, business components, or attackers attempting to access the data directly. From an authorization perspective, the data tier ensures that only approved users can access and modify data regardless of how they attempt to access it.

Create access checks in the data tier when you need to perform the following actions:

- Prevent sensitive data from propagating outside the data tier.
- Prevent unauthorized modification of existing data or insertion of new data.
- Limit access to data by using stored procedures or views rather than modifying direct access permissions to the data tables.

To perform authorization in the data tier, you can use several techniques and objects, depending on your security requirements. You can perform authorization in the following data tier components:

- **Data Access Logic Components**—Data Access Logic Components are responsible for retrieving data from the database, saving data back to the database, and processing any logic required to achieve these data operations. All application data travels to and from the database through the Data Access Logic Components to the rest of the application.

- **Stored Procedures**—Stored procedures can perform authorization logic within the database and retrieve and update the data.
- **Database Security Features**—Use security features built into SQL Server to secure database objects such as tables, columns, views, and stored procedures.

For more information about working with Data Access Logic Components and entities, see “Designing Data Tier Components and Passing Data Through Tiers” in the MSDN Library.

The following questions will help you choose your authorization strategy in the data tier:

- **How complex is the authorization logic?**
  - **Simple**—Perform the authorization check in a stored procedure.
  - **Complex**—Perform the authorization check in the Data Access Logic Components, and call different stored procedures.
- **How complex is the stored procedure?**
  - **Simple**—Perform the authorization check in the Data Access Logic Components and call different stored procedures if creating multiple stored procedures that perform similar actions does not make maintenance more difficult.
  - **Complex**—Perform the authorization check in a stored procedure that takes a parameter that supplies information for the authorization check.  
If creating multiple stored procedures that perform similar actions makes maintenance more difficult, pass a value to the stored procedure to minimize the number of stored procedures.
- **How many stored procedures involve authorization checks?**
  - **A small percentage**—Create multiple versions of stored procedures that involve authorization checks. Create naming standards that obviously link the stored procedures, such as for GetClaimsMGR and GetClaimsEMP. This approach can be detrimental to application design as your application grows bigger, so use it with caution.
  - **A large percentage**—Create individual stored procedures that retrieve different data depending on who the connected user is. Use service accounts that represent application roles to connect to the database, allowing the stored procedure to perform the authorization check. This approach reduces maintenance by minimizing the number of stored procedures.

Follow these best practices when creating authorization code in the data tier:

- Connect to the database with few service accounts to maximize the performance benefit of connection pooling. Connection pooling allows applications to reuse connections when the connection information (such as the user details) matches an existing connection. Therefore, do not create connections as individual users.

For more information about connection management, see “Managing Database Connections” in the “.NET Data Access Architecture Guide” in the MSDN Library.

- Protect database objects with **GRANT** and **DENY** permissions. Denying direct access to the data tables blocks users who connect to the database by using applications such as query analyzers, which do not enforce business logic on data updates or queries. Limit access to stored procedures and views to only allowed user/service accounts.
- Always use stored procedures when using application authorization techniques that involve WHERE clauses. Doing so helps prevent the SQL injection attack, in which attackers append additional (usually malicious) SQL statements to legitimate WHERE clauses. For more information about SQL injection attacks, see “Building Secure ASP.NET Applications” in the MSDN Library.

### Performing Authorization in Data Access Logic Components

Data Access Logic Components are the last components that expose functionality before accessing your database. As such, you can use them to ensure only authorized users can access or modify data.

Perform authorization in Data Access Logic Components when:

- Multiple applications use the same Data Access Logic Components.
- You need to protect access to powerful functions exposed by the Data Access Logic Components or the data store.
- You need to restrict user access to sensitive data.

Examples of using role-based security within Data Access Logic Components include filtering data before it propagates to other application tiers and controlling which stored procedures your code executes.

For more detailed information about how to perform authorization checks, see “Performing Authorization Checks” later in this guide.

### Filtering Data in the Data Access Logic Components

To ensure you return only suitable information to other application tiers, selectively remove sensitive data in your Data Access Logic Component code.

The following code sample removes the ConfidentialNotes column from the data when users do not belong to the Manager role.

```

public DataSet GetClaimData()
{
    DataSet claimData;
    // Database code to retrieve DataSet
    //...
    if (!Thread.CurrentPrincipal.IsInRole( "Manager" ))
        claimData.tables[0].Columns.Remove["ConfidentialNotes"];
    return claimData;
}

```

Retrieving the unused column from the database is a waste of resources, especially if multiple columns are unused. Instead use stored procedures to select only the necessary data to increase database engine performance.

### Calling Different Stored Procedures from the Data Access Logic Component

You can create multiple stored procedures that return the correct data for only one role. Your Data Access Logic Component code then performs an authorization check to determine which stored procedure to use based on the user's roles.

For example, you can create a version of a stored procedure for Managers and another for Employees that differ only by the data retrieved. The Data Access Logic Component code checks the user's role membership and calls the correct stored procedure.

The following code shows how the Data Access Logic Component method chooses which stored procedure to call based on the user's role membership. This code uses a helper class to avoid showing a lot of ADO.NET code. The Microsoft Application Blocks for .NET provides the `SqlHelper` class:

```

public DataSet GetClaimData(Int32 id)
{
    DataSet dsClaimData;
    if (Thread.CurrentPrincipal.IsInRole( "Manager" ))
        dsClaimData= SqlHelper.ExecuteDataset(CONN_STRING,
            CommandType.StoredProcedure, "GetClaimsMGR",
            new SqlParameter("@ID", id));
    else if (Thread.CurrentPrincipal.IsInRole( "Employee" ))
        dsClaimData= SqlHelper.ExecuteDataset(CONN_STRING,
            CommandType.StoredProcedure, "GetClaimsEMP",
            new SqlParameter("@ID", id));
    return dsClaimData;
}

```

### Performing Authorization in the Stored Procedures

You can perform authorization within stored procedures by using two main approaches:

- **Passing a role flag**—Perform the authorization check in the Data Access Logic Component and pass an authorization result flag to the stored procedure. Doing so allows the stored procedure to filter the data based on the authorization flag, as the following code shows.

```
CREATE PROC GetClaimDetails
(
    @ClaimID int,
    @IsUserManager bit
)
AS

IF @IsUserManager = 0
    SELECT ClaimDate,
           ClaimAmount,
           ClaimDescription
    FROM Claims
    WHERE
        ClaimID = @ClaimID
ELSE
    SELECT ClaimDate,
           ClaimAmount,
           ClaimDescription,
           ApprovedFlag,
           ConfidentialNotes
    FROM Claims
    WHERE
        ClaimID = @ClaimID
```

- **Passing the user identity or other role information**—The Data Access Logic Component passes the user identity to the stored procedure as a parameter. The stored procedure then uses the parameter to look up the user's role membership and to select the data, as the following code shows.

```
CREATE PROC GetClaimDetails
(
    @ClaimID int,
    @UserIdentity varchar(20)
)
AS

IF NOT EXISTS(SELECT UserId FROM UserRoles
              WHERE UserId = @UserIdentity AND Role = 'Manager')
    SELECT ClaimDate,
           ClaimAmount,
```

```
        ClaimDescription
FROM Claims
WHERE
    ClaimID = @ClaimID
ELSE
SELECT    ClaimDate,
          ClaimAmount,
          ClaimDescription,
          ApprovedFlag,
          ConfidentialNotes
FROM Claims
WHERE
    ClaimID = @ClaimID
```

## Performing Authorization in the Database

Use the database security features as the foundation for a secure solution. You can then design application authorization by using stored procedures on top of the database security features, as described earlier in this guide.

Perform authorization using database security features to:

- Control access to tables, columns, views, stored procedures, and other data objects using SQL Data Definition Language (DDL) security statements such as **GRANT**, **DENY**, and **REVOKE**.
- Restrict access to sensitive data at the lowest possible level (only the data). SQL Server does not natively support row-level authorization; however, you can simulate it by creating views and stored procedures, as described later in this topic.
- Tighten security as much as possible. Note that this high level of security may come at the cost of decreased scalability.

Before you can perform authorization in the database, you must choose how your application will connect to the database.

### Using SQL Roles to Simplify Administration

An important consideration in your design of your system is to simplify the maintenance of SQL Server by using SQL Server roles. SQL Server provides several types of roles, including:

- **User-defined database roles**—Roles created by the **sp\_addrole** stored procedure unique to the database. You can then add SQL users to these roles with the **sp\_addrolemember** stored procedure.
- **Application roles**—Roles created for applications not for Windows users or groups.
- **Fixed database roles**—Fixed roles linked to common database functionality, such as SQL administrator and database creator; **sysadmin**; and **dbcreator**.

For more information about SQL Server security, authorization, and authorization roles, see “Building Secure ASP.NET Applications” in the MSDN Library.

Consider linking your application roles to the SQL Server roles to make administration consistent for both your application and SQL Server. Connect to the database as a service account that maps directly to a SQL role to achieve this linkage. The following section describes this technique.

### **Connecting to the Database**

To control access to tables, columns, views, stored procedures, and other data objects, SQL Server must be able to identify the user that requires authorization, by using one of the following connection approaches:

- **Windows authentication**—If SQL Server is set up to use Windows Only authentication, the process that establishes the connection must be executing under the context of the user. SQL Server uses this Windows user account for all forms of authorization.

This approach provides the strongest form of connection security because credentials are not passed over the wire.

- **SQL Server authentication (mixed security)**—If SQL Server is set up to use SQL Server and Windows authentication, you can use Windows authentication or you can specify the user in the connection string, which requires a user name and a password.

Create several service accounts that represent application roles regardless of how you connect to the database. Then grant permissions on each service account based on the role’s requirements.

For example, create service accounts such as `USR_MANAGER` and `USR_APPROVER` to implement the principle of least privilege, which allows access to only a subset of the application functionality.

This approach can cause difficulties. When you connect to the database by using Windows authentication, you need to impersonate the service account from the Data Access Logic Components.

For more information about managing database connections, see the “.NET Data Access Architecture Guide” in the MSDN Library.

### **Controlling Access to Sensitive Columns**

You can control access to columns using **GRANT**, **DENY**, and **REVOKE** permissions at the column level by using either T-SQL statements or the SQL Server Enterprise Manager utility. Use this authorization technique sparingly because maintaining a long list of column permissions for a large database is too time consuming to be practical. If you employ this approach, use service accounts when you connect to the database, to make maintenance manageable.

### Controlling Access to Sensitive Rows

SQL Server does not implicitly let you control access to individual rows. However, you can create this functionality by adding a security column to the data table. The security column can hold either an individual user name or the minimum role that is required to access the data (assuming roles have some form of hierarchy). The security column allows you to restrict access to individual rows, but this technique is ineffective if users can directly access the data tables.

You can then control access to individual rows by combining the security column feature of the data table in a view or stored procedure with the **SUSER\_SNAME** function, which retrieves the connected user name—or, more likely, the service account name.

The following code shows how you can create a view that returns only the current user's rows.

```
CREATE VIEW SecureClaimsView
AS
SELECT ClaimDate, ClaimAmount, ClaimDescription
FROM Claims
WHERE [User] = SUSER_SNAME()
```

---

**Note:** This approach requires that you use the individual user identity when you connect to the database so that the **SUSER\_SNAME** function retrieves the correct information. In this case, your application cannot take advantage of connection pooling.

---

## Creating Authorization Code with .NET Role-Based Security

This section describes how to write code to perform authorization with .NET role-based security. Role-based security enables you to perform authorization for an entire category or set of users rather than performing authorization for particular users.

You must take the following into account:

- **Performing authorization checks**—There are several programming techniques for performing authorization checks using .NET role-based security. You must decide which technique to use, depending on the requirements of your application.
- **Separating business logic and authorization logic**—It is important to have a strategy for separating your business logic and your authorization logic.
- **Handling authorization errors**—You must decide how to handle authorization failures. Authorization failure affects not only your component code but also any code that calls your component.

- **Performing authorization with multiple threads** —If your application uses multithreading, you must decide how best to associate an **IPrincipal** object with each thread. There are several options, depending on how you want to handle role-based authorization in a multithreaded application.
- **Extending the default implementation** —There are certain situations in which it is beneficial to extend the default implementation provided by classes and interfaces in the .NET Framework class library.

## Performing Authorization Checks

The .NET Framework provides three ways to check whether a principal is a member of a role:

- Manual checks
- Imperative checks
- Declarative checks

This topic describes each of these techniques and offers guidance on when to use one technique instead of the others. Each technique offers similar performance because all of the techniques use the **IsInRole** method of the principal for the executing thread. Likewise, all three techniques assume that the principal automatically or manually links to the **Thread.CurrentPrincipal** property, as described in Designing Authentication for Authorization earlier in this guide.

However, some significant differences exist between the techniques for checking whether a principal is a member of a role. For example:

- Both the manual and the imperative styles allow complex checks involving variables, as described later. The choice between the manual and imperative styles is one of personal preference.
- The declarative style is appropriate when your check involves simply comparing a constant role value and does not involve any variables or complex logic.

Of the three styles, checking roles manually is the most common approach.

### Checking Roles Manually

You can manually check whether a user belongs to a role by calling the **IsInRole** method on the **IPrincipal** object. However, **IsInRole** behaves differently for **GenericPrincipals** and **WindowsPrincipals**.

#### Checking Roles for GenericPrincipals

When you check roles manually using **GenericPrincipals**, you must use a string to represent the role name. For example, the following code checks whether the user belongs to the Claims Approver role.

```
if (Thread.CurrentPrincipal.IsInRole( "Claims Approver" ))
```

## Checking Roles for WindowsPrincipals

When you check roles manually using **WindowsPrincipals**, there are two different ways to represent the role name:

- **Strings**—Use strings if you are checking custom Windows user groups that administrators created for you, such as `<domain>\Claims Department` or `<machinename>\Claims Department`. For example, the following code checks whether the user belongs to a custom Windows user group named `<domain>\Claims Department`:

```
if (Thread.CurrentPrincipal.IsInRole( @"<domain>\Claims Department" ))
```

For local machine level groups, use the **Environment.MachineName** property. You do not have to change the code when you deploy this property to a different computer. The following code shows how to use the

**Environment.MachineName** property.

```
if (Thread.CurrentPrincipal.IsInRole(
    Environment.MachineName + @"\Claims Department" ))
```

---

**Note:** You must specify the word *BUILTIN* (as in `BUILTIN\<Group>`) when using a string value to check for a built-in local Windows user group, such as Administrators or Users, to indicate that this is a special user group name.

---

Use the Whoami.exe tool to discover the correct syntax of the user's Windows user groups. The Whoami.exe tool is available at <http://www.microsoft.com/windows2000/techinfo/reskit/tools/existing/whoami-o.asp>. To display the complete list of user roles, specify the **/all** switch when you execute the tool.

---

**Note:** .NET role-based security does not support the Windows 2000 format `username@domain.com`.

---

- **WindowsBuiltInRole enumeration**—Use the **WindowsBuiltInRole** enumeration to check role membership against built-in Windows user groups such as Administrators or Users. For example, the following code checks whether the user belongs to the built-in Windows Users role.

```
WindowsPrincipal WP = (WindowsPrincipal) Thread.CurrentPrincipal;
if (WP.IsInRole(WindowsBuiltInRole.User))
```

This enumeration enables you to more easily localize code because the Windows installation culture does not affect the list of user groups.

### Implementing Complex Manual Checks

Whether you use a **GenericPrincipal** or a **WindowsPrincipal**, you can implement complex access checks with multipart **IF** statements using logical operators such as logical AND (**&&**) and logical OR (**||**) in C#. The following code shows such a check:

```
if ((Thread.CurrentPrincipal.IsInRole( "Claims Approver" ) && (amount < 100)) ||  
    (Thread.CurrentPrincipal.IsInRole( "Senior Management" ) && (amount < 500)))
```

You can perform complex checks by combining operators such as the less than symbol (**<**) and the greater than symbol (**>**).

Handle authorization failure by throwing an exception. For more information on propagating authorization check failures, see “Throwing Authorization Exceptions” later in this guide.

### Checking Roles Imperatively

You can use imperative checks to mandate that a user belongs to a particular role. You can imperatively check the user’s role membership by creating a **PrincipalPermission** object and specifying the required role. The **Demand** method then performs the access check. If the principal is not a member of the demanded role, the common language runtime generates a **SecurityException**.

The following code checks to see whether the active principal belongs to an application-specific role CEO.

```
PrincipalPermission CEOPermission = new PrincipalPermission(null,"CEO");  
CEOPermission.Demand();
```

---

**Note:** In all the examples shown in this topic, you pass a **null** to the **PrincipalPermission** constructor for the **User** attribute so that any user who belongs to the appropriate role passes the access check. If you specify a user name, the runtime checks for a particular user, it defeats the purpose of using roles in this situation.

---

You can check role membership imperatively by using the following techniques:

- **Implementing an OR scenario**— You can implement complex checking by using the **Union** method, which represents an **OR** scenario. The following code checks whether the user is a member of the CEO or the Senior Manager role.

```
PrincipalPermission CEOPermission = new PrincipalPermission(null,"CEO");  
PrincipalPermission MgtPermission = new PrincipalPermission(null,  
    "Senior Manager");  
(CEOPermission.Union(MgtPermission)).Demand();
```

- **Creating an AND scenario**—You can create an **AND** scenario by performing two checks sequentially, as the following code shows.

```
PrincipalPermission CEOPermission = new PrincipalPermission(null,"CEO");
PrincipalPermission MgtPermission = new PrincipalPermission(null,
    "Senior Manager");
CEOPermission.Demand();
MgtPermission.Demand();
```

- **Working with variables**—You can check role names dynamically by combining variables, domain names, and machine names. This is similar to the way you perform manual checking, described earlier in this topic. The following code checks whether the user is a member of the Claims Department role; the **Environment.UserName** property gets the network domain name associated with the current user.

```
PrincipalPermission ClaimsPermission = new PrincipalPermission(null,
    Environment.UserName + @"\Claims Department");
ClaimsPermission.Demand();
```

---

**Note:** The **PrincipalPermission** class also has an **Intersect** method, but this method does not serve any purpose for role checking because two users never intersect. This method exists because it is required to implement the **IPermission** interface.

---

You can handle the exception generated by failed calls to **Demand** by using standard exception handling techniques. For more information about propagating authorization check failures, see “Handling Authorization Errors” in this section.

## Checking Roles Declaratively

You can place the **PrincipalPermissionAttribute** on classes, methods, and properties to declaratively require role permissions, as the following code shows. Note that member-level attributes override any class-level attributes.

```
[PrincipalPermission(SecurityAction.Demand, Role="CEO"),
 PrincipalPermission(SecurityAction.Demand, Role="Senior Manager")]
public class Claim
{
    public void ViewClaims()
    {
        // Only 'CEO' and 'Senior Manager' members can access this method
    }

    [PrincipalPermission (SecurityAction.Demand, Role="Claims Approver")]
    public void ApproveClaims()
    {
        // Only 'Claims Approver' members can access this method
    }
}
```

The .NET compilers store the **PrincipalPermissionAttribute** as XML in the assembly metadata during compilation. During application execution, the runtime instantiates a **PrincipalPermission** object and performs the access check. For this reason, the capabilities of the imperative and declarative access checks are similar.

The runtime does not create proxies for the call and does not marshal any parameters until the authorization check passes; therefore, performance improves in cases in which the authorization check fails. If the check fails, the runtime automatically throws a **System.Security.SecurityException**, and no part of the protected method code can execute. Because no part of the method executes, you cannot perform more complex authorization checks based on business logic, such as comparing parameter values.

The runtime evaluates the attributes at compile time, so you must hard-code the roles within the metadata; therefore, you cannot implement a dynamic solution by using values such as **Environment.MachineName**. You also cannot use the **WindowsBuiltInRole** enumeration because declarative checks use the base implementation of **IPrincipal**.

If you specify more than one **PrincipalPermissionAttribute**, the attributes are combined to represent an **OR** scenario. Therefore, the authorization check succeeds if the principal is a member of any of the roles. The following code demands that the caller of this method belongs to either the CEO or the Senior Manager role:

```
[PrincipalPermission(SecurityAction.Demand, Role="CEO"),  
 PrincipalPermission(SecurityAction.Demand, Role="Senior Manager")]  
public void ViewClaimHistory() { }
```

---

**Note:** It is not possible to declaratively check whether the user is a member of all of a list of roles.

---

The declarative technique tends to separate the authorization code from the other business logic. Separating the business logic from your authorization requirements improves the maintainability of your code. This separation allows you to change your authorization logic without changing your business logic, thereby reducing the possibility of inadvertently introducing business logic errors in the code.

## Separating Business Logic and Authorization Logic

The designers of the .NET role-based security framework created simple definitions for the **IIdentity** and **IPrincipal** interfaces, with the intent that developers would extend them to meet their needs. You might want to extend this functionality when:

- You need to store some extra information about the user (more than simply the user name and roles) that the application requires.

For example, you may need to store information such as an organizational site code to indicate that the user is located at a particular site.

- You want to use your own source of role-based security information. The .NET Framework provides an implementation (using the **WindowsIdentity** and **WindowsPrincipal** classes) that retrieves the role information from Windows. You will often want to implement custom extensions if you are maintaining your own store of information.
- You have specialized requirements for authorization. For example, you may need to write code that is dependent upon its deployment location. This is an example of “external knowledge” required to implement an authorization check.

## Handling Authorization Errors

If an authorization check fails within a business process, the component code needs to inform the caller of the failure. The best approach for doing so is to throw an exception. If you are implementing an asynchronous solution, treat the exception the same way as you would treat any standard asynchronous exception handling approach, such as inserting the exception into an exception queue when using MSMQ.

When you design your components, assume that appropriate authorization checking has occurred at the trust boundary gatekeeper. If these assumptions are violated, your component should throw an exception. For example, you can assume that the user interface will never allow users not in the Administrator role to delete files. It is effectively checking at the gate when the application creates the user interface. The component that performs the logic to delete the file must affirm this assumption.

## Throwing Authorization Exceptions

When application authorization checks fail, you can throw any of the following types of exception:

- **System.ApplicationException**—You can throw an **ApplicationException** with a message indicating that an application authorization check failed. Use this exception type in your application if you consider authorization checks to be a subset of your business logic. For example:

```
if (!Thread.CurrentPrincipal.IsInRole( "Claims Approver" ))
{
    throw new System.ApplicationException( "User is not a Claims Approver" );
}
```

- **System.Security.SecurityException**—You can throw a **SecurityException** with a message indicating that an application authorization check failed. For example:

```
if (!Thread.CurrentPrincipal.IsInRole( "Claims Approver" ))  
{  
    throw new System.Security.SecurityException(  
        "User is not a Claims Approver" );  
}
```

The runtime automatically throws this exception when an imperative or a declarative authorization check fails.

Use this exception in your application if you consider authorization failures to be security violations. This exception type enables you to catch all security-related exceptions in a single **catch** handler.

Code access security and other system-level actions also throw this exception, which may make it difficult for the caller to filter the exceptions and react appropriately.

- **Custom exceptions**—As mentioned earlier, **SecurityException** is the default exception type when you use the declarative and imperative authorization styles. The runtime returns the standard message, *Request for principal permission failed*. You might need to add extra information to help the caller deal with the exception in a particular way, depending on the particular error. To address these needs, you can define your own custom exception classes to represent the different kinds of authorization errors that can occur in your application.

Creating your own custom exception classes benefits your application in the following ways:

- The calling code can define separate **catch** handlers for each type of exception to handle errors in a particular way. For example, the calling code can filter certain types of exception and log them for operations purposes.
- If you design your authorization exception classes correctly, you can reuse them in multiple applications that involve authorization checks.
- You can group common types of authorization failure together. Doing so allows process and components that aggregate others to perform corrective action.

For more information about creating your own exceptions and for a sample implementation, see “How to Create an Authorization Custom Exception Type” in the Appendix.

---

**Note:** Propagated authorization exceptions should not contain any sensitive information that could compromise the system. For more information about this and other exception management best practices, see “Exception Management in .NET” in the MSDN Library.

---

## Handling Authorization Exceptions

The calling code can define **catch** handlers for authorization exceptions in addition to your role-based authorization exceptions. You might receive several forms of **SecurityException** violations, depending on your deployment scenario. For example, if you launch your code from a semitrusted environment, the runtime might throw exceptions to indicate code access security permission failures, such as file access issues. Although these exceptions are not related to role-based authorization, they might result in a **SecurityException**.

You can write exception-handling code to differentiate between role-based authorization exceptions and other types of authorization exceptions. Use the **PermissionType** property of the **SecurityException** object to check whether the exception relates to role-based authorization, as the following code shows.

```
try
{
    // Code here that could create a role-based authorization exception,
    // such as an imperative check.
}
catch (SecurityException se)
{
    if (se.PermissionType == typeof(PrincipalPermission))
        // Do something here if exception is role related.
    else
        // Do something here if exception is not role related.
}
```

The result of the **PermissionType** property allows you to filter both code access security and role-based authorization exceptions. After you filter an exception, you can throw a custom exception or handle the exception internally in the callee.

## Performing Authorization with Multiple Threads

Most .NET-based applications use multiple threads to improve responsiveness or to perform work asynchronously. Multithreading introduces complexities when you work with principals because each thread must link to an **IPrincipal** object. The runtime copies the **IPrincipal** object to any new threads that the parent thread creates.

The **Thread.CurrentPrincipal** property returns a different **IPrincipal** object according to the following rules:

- If you explicitly associate an **IPrincipal** object with the thread, that **IPrincipal** object is returned.
- If you do not explicitly associate an **IPrincipal** object with the thread and you use the **SetThreadPrincipal** method, a default **IPrincipal** object for the application domain is returned.

Calling the `AppDomain.CurrentDomain.SetThreadPrincipal` method ensures that the runtime uses a specific `IPrincipal` object for current and new threads. This option is not recommended for ASP.NET because the runtime handles principals. Also, it can affect other threads that are running on behalf of other users.

---

**Note:** The runtime throws a `PolicyException` if you call this method more than once during the lifetime of an application domain.

---

- If no `IPrincipal` object or default application domain `IPrincipal` object is associated with the thread, an `IPrincipal` object of the type specified in the `SetPrincipalPolicy` method is returned, as described earlier in this guide.
- If you do not use any of the preceding methods, `null` (or `Nothing`) is returned.

## Extending the Default Implementation

Follow these guidelines when extending the .NET-based application authorization framework:

- **Implement the `IIdentity` and `IPrincipal` interfaces**—This will ensure that your extensions are compatible with other features that use the .NET role-based security model, such as the ASP.NET URL authorization feature.
- **Use lazy initialization where possible**—If the task of populating the role information to construct your `IPrincipal` object is time consuming, you should delay this activity until the first time the information is required.
- **Attach your `IPrincipal` object to the executing thread**—This allows you to use the declarative, imperative, and manual authorization checking styles mentioned previously. It also works well with other technologies, such as ASP.NET authorization.

## Deciding on an Extension Strategy

Decide what type of extension to create and where you should create it. Table 1 contains some common extensions and indicates where you should create the additional logic required.

**Table 1: Where to create your application authorization extensions**

Type of extension	Custom identity	Custom principal	Helper class
User-specific information	X		
Permission or role related		X	
External knowledge required			X

## Implementing Custom Identities

You should create your own class that implements **IIdentity** when you require more functionality than is provided by the **GenericIdentity** or **WindowsIdentity** object. Possible enhancements include:

- Managing tokens or user identifiers. This approach is generally required when you need to encapsulate an authentication process that involves tokens. The **WindowsIdentity**, **FormsIdentity**, and **PassportIdentity** classes manage a token that represents the user. The classes use a Microsoft Win32® token, forms authentication ticket, and the Passport Unique Identifier (PUID), respectively.  
For example, you could create your own custom identity when using an ISAPI Single Sign-On solution.
- Encapsulating your own code in the **IIdentity** object, which is then available to all the applications. For example, the **PassportIdentity** provides many extra properties and methods over the standard **IIdentity** interface, including profile information. This extra information can be added to your identity class using standard techniques such as public properties. Another example might be that you frequently base authorization checks on the user's manager; in this situation, you can provide the user's manager as a value on the identity.

## Implementing Custom Principals

You should create your own class that implements **IPrincipal** when you require more functionality than is provided by the **GenericPrincipal** or **WindowsPrincipal** object. Possible enhancements include:

- Implementing your own enumeration of roles in the same way that the **WindowsBuiltInRole** enumeration works for a **WindowsPrincipal**. This can be achieved by overloading the **IsInRole** method to enable calling code to choose whether to use your enumeration or a string.
- Checking whether the principal belongs to any role in a specified list. For example:

```
CustomPrincipal.IsInAnyRole( string[] roles );
```

- Accessing the list or enumerating the roles that the principal belongs to. For example:

```
string[] roles = CustomPrincipal.Roles;
```

- Enforcing some type of role hierarchy so that some roles are considered "more than" or "less than" other roles. For example, a "Senior Manager" is "more than" or above a "Junior Manager."

## Reusing Authorization Implementations

You can reuse your application authorization framework across multiple applications if you design the framework with reuse in mind. Reuse typically involves the code and infrastructure used to implement the authorization capabilities.

Benefits of reusing your application authorization framework include the following:

- Developers in your organization can target one application authorization framework.
- You can easily reuse components that adhere to the application authorization framework in other applications because they use a standard authorization approach.
- The authorization subsystem becomes a focal point for synchronizing with other systems or platforms. Reuse alleviates the need to synchronize authorization details from legacy applications to new applications that need the authorization data.
- Authorization management is consistent across applications and makes the experience consistent for administration and operations personnel.

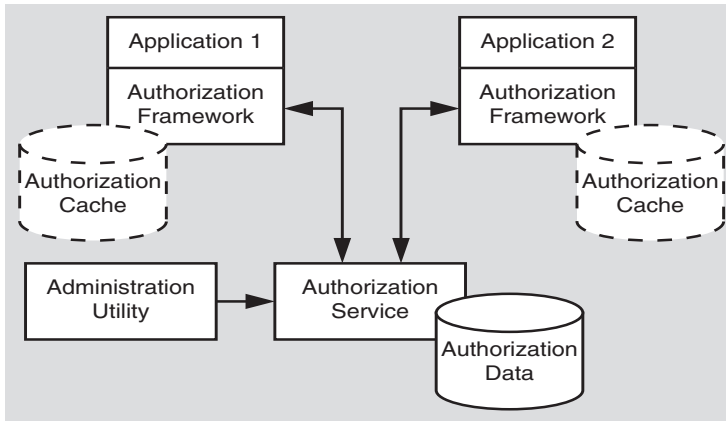
This section includes some best practices for reusing authorization implementations.

### Creating a Reusable Authorization Framework

You can choose how much of your application authorization framework to reuse across applications. Options include:

- Reusing one authorization database and one XML Web service that accesses the authorization data.
- Reusing the authorization framework but having separate database instances for each application.

Figure 6 shows an example of how you can implement application authorization reuse.



**Figure 6**

*Logical representation of application authorization reuse*

Note the following points in Figure 6:

- Different applications can reuse the same authorization framework.
- An authorization service provides secured access to authorization data.
- A centralized database holds the authorization data.
- A single administration utility provides all configuration.
- A local cache improves authorization performance.

Table 2 on the next page describes each of the components shown in Figure 6 and explains how to reuse these components in different projects and applications.

**Table 2: How to reuse authorization components**

Component	Description	How to reuse
Authorization framework	Custom code created to support authorization capabilities, such as custom principals, identities, and exceptions	Create an assembly that contains all the custom reusable code that will be useful to other projects. Optionally, strongly name the assembly and install it in the global assembly cache.
Authorization cache	An optional unit of functionality that keeps the authorization information physically close to where the framework works with it	Create a <b>Hashtable</b> object (or other such in-memory store) with an appropriate method for refreshing on demand. For efficiency, <b>IPrincipal</b> objects should use this cache for retrieving role membership instead of going to the database. Use encryption if you need to protect or hide the cached data; otherwise, sign the cached data to prevent tampering (this is particularly important if you store the data in a local file or in another vulnerable location at any point.)
Authorization service	An interface (or <i>gatekeeper</i> ) to the authorization data	Create an out-of-process component that runs under a security context that can access the authorization store. Using an out-of-process component allows you to authorize access to the store by checking to see whether the access occurs by the account specified. This allows you to prevent other users from accessing the store. Alternatively, you can build an XML Web service for central reuse. This provides the greatest interoperability. Remember to protect the communication channel between the authorization service and the authorization framework.
Administration utility	A user interface that application administrators use to maintain authorization data	Create a central Web site on which administrators can adjust the authorization settings. Alternatively, create a reusable graphical user interface that you deploy with each application.
Authorization data	The definition of the authorization data schema, stored procedures and Data Access Logic Components	Use one database for all authorization data. Alternatively, use a standard authorization schema for each application.

## Securing the Implementation

To secure your reusable authorization framework, follow these guidelines:

- Perform authentication checks on callers to the authorization data store. Doing so prevents unauthorized parties from querying the authorization information.
- Use a secure transport such as SSL or IPSec.
- Deploy an authorization service, as an XML Web service, in a separate directory for each application. Doing so allows you to configure the service to work only with the application in which you deploy it.

## Improving the Performance of a Reusable Authorization Framework

Improve the performance of your reusable application authorization framework by using the following techniques:

- Batch authorization queries whenever possible to avoid frequent out-of-process round trips. For example, retrieve roles for multiple users in a single request.
- Cache the authorization data close to where you will use it with an in-memory store, such as a **Hashtable**. The cache also reduces dependencies on the location and organization of the underlying store. You might also want a separate cache for each physical computer, for performance and increased security.
- Implement scheduled or on-demand refreshes of the cache information.
- Implement lazy initialization of the authorization cache to avoid retrieving authorization information when no access checks will occur.

## Appendix

This appendix includes the following topics:

- How to Enable Authorization in a .NET Remoting Component
- How to Perform Authorization in an XML Web Service
- How to Create an Authorization Custom Exception Type
- How to Change the Principal in an ASP.NET Application
- How to Build a GenericPrincipal Using SQL Server
- How to Use System.EnterpriseServices COM+ Role-Based Security

### How to Enable Authorization in a .NET Remoting Component

.NET remoting does not automatically flow the principal from one application domain to another, except when the domains are in the same process. You must manually pass this information between the caller and the callee and ensure that the information remains secure during passage.

For information about .NET remoting with Windows authentication, see “.NET Remoting Security Solution” in the MSDN Library.

When you are working with non-Windows authentication, you can provide identity flow by using the **CallContext** class and the **ILogicalThreadAffinative** interface.

The **CallContext** class lets you include additional information when a method is called on the remote object. This information can be anything you require for the individual method, such as user information.

Only types that implement the **ILogicalThreadAffinative** interface are sent to a server that is in a separate application domain. For example, you can create a simple type that implements **ILogicalThreadAffinative**; that stores an object based on **Identity** or **IPrincipal**. This object would be accessible in the remote object, and you could check role membership.

The following code shows how to create a class that implements the **ILogicalThreadAffinative** interface and persists an **IPrincipal** based object.

```
using System;
using System.Runtime.Remoting.Messaging;
using System.Security.Principal;

namespace RemotingPrincipalFlow
{
    /// <summary>
    /// The storage used to transport the principal between layers.
    /// </summary>
    [Serializable]
    public class PrincipalStorage : ILogicalThreadAffinative
    {
        public PrincipalStorage( IPrincipal principal )
        {
            // Check the parameter.
            if( principal == null )
                throw new ArgumentNullException(
                    "principal", "The argument cannot be null." );

            // Set the internal principal.
            _principal = principal;
        } IPrincipal _principal;

        public IPrincipal Principal
        {
            get
            {
                return _principal;
            }
        }
    }
}
```

The following code shows how the remote object can then use the **PrincipalStorage** class to retrieve the **IPrincipal** information by using the **CallContext.GetData** method.

```
using System;
using System.Threading;
using System.Runtime.Remoting.Messaging;

namespace RemotingPrincipalFlow
{
    /// <summary>
    /// The remoted class.
    /// </summary>
    public class RemObject : MarshalByRefObject
    {
        public RemObject(){ /* Empty constructor */ }

        public void ExampleMethodUsingCallContext()
        {
            // Retrieve the principal from the CallContext.
            PrincipalStorage ppal =
            CallContext.GetData( "Principal" ) as PrincipalStorage;

            // Set the current principal.
            if( ppal != null )
                Thread.CurrentPrincipal = ppal.Principal;

            // Show the principal information.
            Console.WriteLine( "Identity.Name: " +
                Thread.CurrentPrincipal.Identity.Name );
        }
    }
}
```

The following code shows how the client application sets the **IPrincipal** information before calling the remote object by using the **CallContext.SetData** method.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Messaging;
using System.Security.Principal;
using System.Threading;
using RemotingPrincipalFlow;

namespace RemotingPrincipalFlow
{
```

```
/// <summary>
/// The client application.
/// </summary>
class RemotingPrincipalFlowDemoClient
{
    [STAThread]
    static void Main(string[] args)
    {
        // Register remoting client channel.
        ChannelServices.RegisterChannel( new TcpChannel() );

        // Register the remoted class.
        RemotingConfiguration.RegisterWellKnownClientType(
            typeof(RemObject),
            "tcp://localhost:6000/RemObject" );

        // Set the current principal.
        Thread.CurrentPrincipal = new GenericPrincipal(
            new GenericIdentity( "Doug", "user" ),
            new string[] { "admin", "user" } );

        // Use the CallContext to set the current principal.
        CallContext.SetData( "principal",
            new PrincipalStorage( Thread.CurrentPrincipal ) );

        Console.WriteLine( "Calling remote object..." );

        // Use the remoted object.
        RemObject ro = new RemObject();
        ro.ExampleMethodUsingCallContext();

        Console.WriteLine( "Call complete, press any key..." );
        Console.ReadLine();
    }
}
}
```

The following code shows the server application that registers the remote object.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace RemotingPrincipalFlow
{
    /// <summary>
    /// The server host application.
    /// </summary>
```

```

class RemotingPrincipalFlowDemoServer
{
    [STAThread]
    static void Main(string[] args)
    {
        // Register the remoted class.
        RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(RemObject), "RemObject",
            WellKnownObjectMode.Singleton );

        // Register remoting server channel.
        ChannelServices.RegisterChannel( new TcpChannel(6000) );

        Console.WriteLine("Running server, press ENTER to quit");
        Console.ReadLine();
    }
}

```

## How to Perform Authorization in an XML Web Service

In the following code, the *ProcessClaim* XML Web service method performs an authorization check and throws an exception if the check fails. The check determines whether the sender of the expense claim is authorized to send it.

```

using System;
using System.Xml.Serialization;
using System.Web.Services;
using System.Security.Permissions;
using System.Security;

namespace AuthZInWS
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    [XmlInclude(typeof(EmployeeClaim))]
    [XmlInclude(typeof(ContractorClaim))]
    public class Claim
    {
        // Base class code for all claims.
        public void ProcessClaim()
        {
        }
    }

    public class EmployeeClaim : Claim
    {
        new public void ProcessClaim()
        {
        }
    }

    public class ContractorClaim : Claim
    {

```

```

        new public void ProcessClaim()
        {
        }
    }
    public class ClaimsServiceInterface
    {
        [WebMethod]
        [PrincipalPermission(SecurityAction.Demand, Authenticated=true)]
        public void ProcessClaim(Claim ClaimIn)
        {
            try
            {
                PrincipalPermission ClaimTypePrincipalPerm = null;
                if(ClaimIn is EmployeeClaim)
                    ClaimTypePrincipalPerm =
                        new PrincipalPermission(null,
                                                "Employee");
                else if(ClaimIn is ContractorClaim)
                    ClaimTypePrincipalPerm =
                        new PrincipalPermission(null,
                                                "Contractor");
                else
                    throw new ApplicationException(
                        "Claim Type is not allowed." );
                ClaimTypePrincipalPerm.Demand();
                ClaimIn.ProcessClaim();
            }
            catch
            {
                throw new SecurityException("Authorization failed");
            }
        }
    }
}

```

To ensure that you are dealing with an authenticated callee, specify the **Authenticated** property of the **PrincipalPermissionAttribute**. Doing so also ensures that the **Thread.CurrentPrincipal** automatically provides the correct **IPrincipal** object for the imperative authorization check.

The following code calls the `ProcessClaim` service. Only users in the `Employee` role can submit employee claims to the XML Web service.

```

static void Main(string[] args)
{
    localhost.Claim ClaimSubmission = new localhost.EmployeeClaim();
    localhost.ClaimsServiceInterface ClaimsService =
        new localhost.ClaimsServiceInterface();
    ClaimsService.ProcessClaim(ClaimSubmission);
}

```

This code assumes that the authenticated user belongs to application-defined roles of either Employee or Contractor. For information about how to load the **IPrincipal** object with roles, see “How to Change the Principal in an ASP.NET Application and How to Build a GenericPrincipal Using SQL Server” later in the Appendix.

## How to Create an Authorization Custom Exception Type

For troubleshooting or auditing, you might throw a custom exception that contains more information than does a typical authorization failure—for example, the user name or e-mail address of the person who caused the exception. The following code creates an exception class for handling an authorization failure. The exception class includes the user name from **Thread.CurrentPrincipal.Identity**.

```
using System;
using System.IO;
using System.Security;
using System.Security.Permissions;
using System.Threading;
using System.Security.Principal;

namespace AuthZExceptions
{
    public class ApplicationAuthorizationException : ApplicationException
    {
        // Default constructor.
        public ApplicationAuthorizationException()
        {
            SetUserName();
        }

        // The constructor accepts a single string message.
        public ApplicationAuthorizationException (string message) :
            base(message)
        {
            SetUserName();
        }

        // The constructor accepts a string message and an inner exception,
        // which is wrapped by this custom exception class.
        public ApplicationAuthorizationException(
            string message, Exception inner) : base(message, inner)
        {
            SetUserName();
        }

        // Why not add the whole principal?
        // The SystemException class is serializable, so you want to add
        // only information that can be serialized.
        // Identity objects often encapsulate a token (Windows) that cannot
        // be serialized, so use the string value of the identity instead.
        //
    }
}
```

```

// Other items you might want to add include:
// Authentication type
// Type of principal
private void SetUserName()
{
    if(Thread.CurrentPrincipal != null)
        m_strUserName =
            Thread.CurrentPrincipal.Identity.Name;
}
private string m_strUserName;
public string UserName
{
    get {return m_strUserName;}
}
}

```

The following code calls a component that performs the authorization check. When the check fails, the **catch** block or blocks test the exception and display the user name from the `ApplicationAuthorizationException`.

```

using System;
using System.Security;

namespace AuthZExceptions
{
    class CallerClass
    {
        /// <summary>
        /// The main entry point for the caller application.
        /// </summary>
        ///
        [STAThread]
        static void Main(string[] args)
        {
            try
            {
                CalleeClass.PerformAction();
            }
            catch(ApplicationAuthorizationException ex)
            {
                Console.WriteLine( se.Message + " " + ex.UserName );
            }
            catch (SecurityException ex)
            {
                Console.WriteLine( ex.Message );
            }
            finally
            {
                Console.ReadLine();
            }
        }
    }
}

```

The following code acts as the callee component that performs the authorization check. This code deliberately denies access to a particular file so that an exception not related to authorization can occur. The **catch** block uses the **PermissionType** property to filter the two possible exceptions.

```
using System;
using System.Security;
using System.Security.Permissions;
using System.Security.Principal;

namespace AuthZExceptions
{
    // The FileIOPermission is used only to demonstrate a SecurityException
    // in the ReadFile method.
    [FileIOPermission(SecurityAction.Deny, Read=@"C:\file.txt")]
    class CalleeClass
    {
        /// <summary>
        /// The class representing the callee.
        /// </summary>
        ///
        // Method that performs authorization check.
        [PrincipalPermission(SecurityAction.Demand, Role="CEO")]
        static void Authorize()
        {
            Console.WriteLine( "Authorization succeeded" );
        }

        public static void PerformAction()
        {
            // Link the WindowsPrincipal
            AppDomain.CurrentDomain.SetPrincipalPolicy(
                PrincipalPolicy.WindowsPrincipal);

            try
            {
                // Perform the authorization check.
                Authorize();

                // Perform action that creates a SecurityException.
                ReadFile();
            }
            catch (SecurityException se)
            {
                if(se.PermissionType == typeof(PrincipalPermission))
                    throw new ApplicationAuthorizationException
                        ( "Authorization Failed" , se );
                else
                    throw se;
            }
        }
    }
}
```

```
        // Dummy method to throw a SecurityException.
        [FileIOPermission(SecurityAction.Demand, Read=@"c:\file.txt")]
        static void ReadFile() {}
    }
}
```

## How to Change the Principal in an ASP.NET Application

You can change the default **IPrincipal** object within an ASP.NET application to include application-specific roles loaded from a data store after authentication takes place. Changing the **IPrincipal** object lets you rely on any type of authentication while still using application-defined roles for your authorization checks. The **Application\_AuthenticateRequest** event handler is the best location to change the **IPrincipal** object because the authenticated identity is available and the event handler executes before any Web page or XML Web service logic.

The following code replaces the **HttpContext.Current**'s **IPrincipal** object with a **GenericPrincipal** object based on the authenticated identity and adds the object to the **HttpContext.Cache** to improve performance:

```
protected void Application_AuthenticateRequest(Object sender, EventArgs e)
{
    // Check whether there is a current user and that
    // authentication has occurred.
    if (!(HttpContext.Current.User == null))
    {
        IIdentity CurrentUserIdentity = HttpContext.Current.User.Identity;

        // Check to see whether the Principal was cached.
        string CachedPrincipalKey = "CachedPrincipal" + id.Name;
        if (HttpContext.Current.Cache[CachedPrincipalKey] == null)
        {
            // Load the principal by calling the GetPrincipal method.
            HttpContext.Current.Cache.Add(
                CachedPrincipalKey,
                GetPrincipal(CurrentUserIdentity),
                null,
                DateTime.MaxValue,
                new TimeSpan(0,30,0),
                CacheItemPriority.Normal,
                null);
        }
        HttpContext.Current.User = (IPrincipal)
            HttpContext.Current.Cache[CachedPrincipalKey];
    }
}
```

The sample uses the cache to reduce the number of queries to the database. The cache was selected for the example because the cache lets you set a timeout value. However, using session state will require less testing, because ASP.NET guarantees that the user has access to only his or her session. Care needs to be taken with this sort of code to ensure that the thread running the user's request can never be issued with the wrong Principal. For a sample implementation of the **GetPrincipal** method, see the next topic.

## How to Build a GenericPrincipal Using SQL Server

The following example presents a simple database schema that you can use to populate a **GenericPrincipal** with roles. Run the following code to create the UserRoles table:

```
IF EXISTS (select * from dbo.sysobjects where id = object_id(N'[UserRoles]') and
OBJECTPROPERTY(id, N'IsUserTable') = 1)
DROP TABLE [UserRoles]
GO

CREATE TABLE [UserRoles] (
    [UserName] [varchar] (50) COLLATE SQL_Latin1_General_CP1_CI_AS NULL ,
    [Role] [varchar] (50) COLLATE SQL_Latin1_General_CP1_CI_AS NULL
) ON [PRIMARY]
GO

INSERT INTO [Claims].[dbo].[UserRoles]([UserName], [Role])
VALUES('Chris', 'Admin')

INSERT INTO [Claims].[dbo].[UserRoles]([UserName], [Role])
VALUES('Doug', 'Admin')

INSERT INTO [Claims].[dbo].[UserRoles]([UserName], [Role])
VALUES('Doug', 'Manager')
GO
```

The **GetPrincipal** method that follows takes an **IIdentity** as a parameter, which then provides the identity information to the **GenericPrincipal**. The code uses the name of the identity to look up the roles in the UserRoles table for populating the roles in the **IPrincipal** object.

Notice that the code passes the identity name as a string to the **GenericIdentity** constructor rather than using the **IIdentity** parameter. Passing the identity name as a string allows you to use the code in conjunction with any authentication type.

The code also passes a second string to the **GenericIdentity** constructor to indicate the type of authentication that was originally used.

```
private IPrincipal GetPrincipal(IIdentity user)
{
    //Get the roles from the table based on a user name only.
    string SQL =
        "SELECT Role FROM UserRoles WHERE UserName = '" + user.Name + "'";

    SqlConnection MyConnection = new SqlConnection(
        "data source=localhost;initial catalog=Claims;Integrated Security=SSPI");
    SqlCommand MyCommand = new SqlCommand(SQL, MyConnection);
    MyConnection.Open();
    SqlDataReader MyDataReader = MyCommand.ExecuteReader();

    ArrayList alRoles = new ArrayList();

    // Load the roles into an ArrayList.
    while (MyDataReader.Read())
        alRoles.Add(MyDataReader.GetString(0));

    MyDataReader.Close();
    MyCommand.Dispose();
    MyConnection.Close();
    MyConnection.Dispose();

    // Convert the roles to a string[], and load GenericPrincipal.
    string[] myRoles = (string[])al.ToArray(typeof(string));
    return new GenericPrincipal(
        new GenericIdentity(user.Name, user.GetType()),
        myRoles);
}
```

For sample code that calls the **GetPrincipal** method, see “How to Change the Principal in an ASP.NET Application” earlier in this Appendix.

## How to Use System.EnterpriseServices COM+ Role-Based Security

When a .NET-based application uses COM+ to provide role-based security, you can check role membership by using one of two approaches: manual or declarative.

### ► To perform a manual or a declarative access check

1. In your code, specify the **ApplicationAccessControlAttribute** at the assembly level to set the level of authorization using the **AccessChecksLevel** property. Use the **AccessChecksLevelOption.ApplicationComponent** value to perform checks at both the application and component levels.
2. Enable access checks for each component that requires role-based security by applying the **ComponentAccessControlAttribute** to the appropriate class.

3. Derive from **ServiceComponent** for any class requiring role-based checks.
4. Add any roles required by your application by using the **SecurityRoleAttribute** at the assembly, class, interface, or method level of your assembly.

---

**Note:** Steps 1, 3, and 4 don't actually have to be done in code. They can be done in the component services management console.

---

► **To complete configuration of the component services application**

1. Configure authentication to at least the call level; otherwise, the interceptor cannot access caller information.
2. Assign users to appropriate roles.

---

**Note:** To make your application work, you might need to add steps, such as installing the assembly in the global assembly cache. These steps are specific to your deployment and are not covered in this guide. For more information about installing a Component Services application, see "COM+ Integration: How .NET Enterprise Services Can Help You Build Distributed Applications" in MSDN Magazine.

---

## Performing Manual COM+ Role Checks

Manual role-based checks rely on the availability of the security call context object within your application code. This context object is available if you enable security for your component services application.

When checking COM+ role membership manually, use the following techniques:

- Check whether a user belongs to a particular role by calling the **IsCallerInRole** method on the **SecurityCallContext** object. As a precaution, check whether security is enabled before you check role membership. The result from calling **IsCallerInRole** is always **true** if security is disabled.

The following code shows how to test security and check for role membership.

```
if (ContextUtil.IsSecurityEnabled)
    return SecurityCallContext.CurrentCall.IsCallerInRole(
        "Claims Approver" );
```

- Create complex checks by using multipart **IF** statements using standard syntax such as logical AND (&&) and logical OR (||) checks. The following code shows such a test.

```
if (SecurityCallContext.CurrentCall.IsCallerInRole( "Claims Approver" ) ||
    SecurityCallContext.CurrentCall.IsCallerInRole( "Senior Management" ))
```

- Create constants in your application, or retrieve the user roles from an external source, such as a database.
- Use the **SecurityCallContext.CurrentCall.OriginalCaller** property to access information about the user. This property returns a **SecurityIdentity** object, as the following code shows.

```
Debug.WriteLine(SecurityCallContext.CurrentCall.OriginalCaller.AccountName);
```

## Performing Declarative COM+ Role Checks

Declarative checks enable users who belong to a specified role to activate the component and call any of its public methods. This type of role check is analogous to specifying the declarative .NET role-based security attributes at the class level. You specify the role that has access to the class by using the **SecurityRoleAttribute** at the class level, as the following example shows.

```
using System;
using System.EnterpriseServices;
using System.Reflection;

[assembly: ApplicationName("Expenses")]
[assembly: ApplicationActivation(ActivationOption.Server)]
[assembly: ApplicationAccessControl(true,
AccessChecksLevel=AccessChecksLevelOption.ApplicationComponent)]
[assembly: SecurityRole("Claims Approver")]
[assembly: SecurityRole("Senior Management")]

namespace Expenses
{
    [ComponentAccessControl()]
    [SecurityRole("Claims Approver")]
    public class Claim : ServicedComponent
    {
        public bool Approve() {return true;}
    }
}
```

## Performing Declarative Checks Down to the Method Level

Performing declarative checks on methods requires more steps than do checks at the class level:

1. Implement an interface.
2. Include the **SecureMethodAttribute** at the class or the method level. This attribute configures role security at the method level when you install the application into COM+. To allow all methods to be secured either declaratively within the code or manually using the administration utility, specify the attribute at the class level.

3. To link the methods to roles, use the **SecurityRoleAttribute** on the method implementation.
4. Add users manually in the Component Services management console to the Marshaler role. COM+ creates this role automatically when you install the application into COM+ so that both managed and unmanaged clients can call the secure methods.

The following code demonstrates these steps.

```
using System;
using System.EnterpriseServices;
using System.Reflection;

[assembly: ApplicationName("Expenses")]
[assembly: ApplicationActivation(ActivationOption.Server)]
[assembly: ApplicationAccessControl(true,
    AccessChecksLevel=AccessChecksLevelOption.ApplicationComponent)]
[assembly: SecurityRole("Claims Approver")]
[assembly: SecurityRole("Senior Management")]

namespace Expenses
{
    public interface IClaim
    {
        bool Approve();
    }
    [ComponentAccessControl()]
    [SecureMethod]
    public class Claim : ServedComponent, IClaim
    {
        [SecurityRole("Claims Approver")]
        public bool Approve() {}
    }
}
```

## Feedback and Support

Questions? Comments? Suggestions? For feedback on Designing Application-Managed Authorization, please email us at [devfdbck@microsoft.com](mailto:devfdbck@microsoft.com).

The Application Blocks for .NET components and guides are designed to jumpstart development of .NET distributed applications. The sample code and guidance is provided "as-is." While this deliverable has undergone testing and is considered a robust set of procedures and recommendations, it is not supported like a traditional Microsoft product.

A newsgroup has also been created to assist you with the Application Blocks for .NET components and guides. Use this newsgroup to consult with your counterparts, peers, and Microsoft Support Professionals in an online, open forum.

Everyone else benefits from your questions and comments, and our development team is monitoring the newsgroup on a daily basis:

Newsgroup: Web-Based Reader

*[http://msdn.microsoft.com/newsgroups/loadframes.asp?icp=msdn&slcid=us&newsgroup=microsoft.public.dotnet.distributed\\_apps](http://msdn.microsoft.com/newsgroups/loadframes.asp?icp=msdn&slcid=us&newsgroup=microsoft.public.dotnet.distributed_apps)*

Newsgroup: NNTP Reader

*[news://msnews.microsoft.com/microsoft.public.dotnet.distributed\\_apps](news://msnews.microsoft.com/microsoft.public.dotnet.distributed_apps)*

## **Collaborators**

Many thanks to the following contributors and reviewers: Erik Olson, Dave McPherson, Riyaz Pishori, Srinath Vasireddy, Abhishek Bhattacharya (Sapient), Dimitris Georgakopoulos (Sapient), Chris Brooks, Ross Cockburn, Kenny Jones, Angela Crocker, Andy Olson, Lance Hendrix, Mark White, Steve Busby, Unmesh Redkar, J.D. Meier, and Diego Gonzalez.

# Microsoft®

## patterns & practices



*Proven practices for predictable results*

Patterns & practices are Microsoft's recommendations for architects, software developers, and IT professionals responsible for delivering and managing enterprise systems on the Microsoft platform. Patterns & practices are available for both IT infrastructure and software development topics.

Patterns & practices are based on real-world experiences that go far beyond white papers to help enterprise IT pros and developers quickly deliver sound solutions. This technical guidance is reviewed and approved by Microsoft engineering teams, consultants, Product Support Services, and by partners and customers. Organizations around the world have used patterns & practices to:

### **Reduce project cost**

- Exploit Microsoft's engineering efforts to save time and money on projects
- Follow Microsoft's recommendations to lower project risks and achieve predictable outcomes

### **Increase confidence in solutions**

- Build solutions on Microsoft's proven recommendations for total confidence and predictable results
- Provide guidance that is thoroughly tested and supported by PSS, not just samples, but production quality recommendations and code

### **Deliver strategic IT advantage**

- Gain practical advice for solving business and IT problems today, while preparing companies to take full advantage of future Microsoft technologies.

**To learn more about *patterns & practices* visit: [msdn.microsoft.com/practices](http://msdn.microsoft.com/practices)**

**To purchase *patterns & practices* guides visit: [shop.microsoft.com/practices](http://shop.microsoft.com/practices)**

patterns & practices

*Proven practices for predictable results*

# patterns & practices



*Proven practices for predictable results*

Patterns & practices are available for both IT infrastructure and software development topics. There are four types of patterns & practices available:

## **Reference Architectures**

Reference Architectures are IT system-level architectures that address the business requirements, operational requirements, and technical constraints for commonly occurring scenarios. Reference Architectures focus on planning the architecture of IT systems and are most useful for architects.

## **Reference Building Blocks**

References Building Blocks are re-usable sub-systems designs that address common technical challenges across a wide range of scenarios. Many include tested reference implementations to accelerate development.

Reference Building Blocks focus on the design and implementation of sub-systems and are most useful for designers and implementors.

## **Operational Practices**

Operational Practices provide guidance for deploying and managing solutions in a production environment and are based on the Microsoft Operations Framework. Operational Practices focus on critical tasks and procedures and are most useful for production support personnel.

## **Patterns**

Patterns are documented proven practices that enable re-use of experience gained from solving similar problems in the past. Patterns are useful to anyone responsible for determining the approach to architecture, design, implementation, or operations problems.

**To learn more about *patterns & practices* visit: [msdn.microsoft.com/practices](http://msdn.microsoft.com/practices)**

**To purchase *patterns & practices* guides visit: [shop.microsoft.com/practices](http://shop.microsoft.com/practices)**

# patterns & practices current titles



## December 2002

### Reference Architectures

- Microsoft Systems Architecture—Enterprise Data Center 2007 pages
- Microsoft Systems Architecture—Internet Data Center 397 pages
- Application Architecture for .NET: Designing Applications and Services 127 pages
- Microsoft SQL Server 2000 High Availability Series: Volume 1: Planning 92 pages
- Microsoft SQL Server 2000 High Availability Series: Volume 2: Deployment 128 pages
- Enterprise Notification Reference Architecture for Exchange 2000 Server 224 pages
- Microsoft Content Integration Pack for Content Management Server 2001 and SharePoint Portal Server 2001 124 pages
- UNIX Application Migration Guide 694 pages
- Microsoft Active Directory Branch Office Guide: Volume 1: Planning 88 pages
- Microsoft Active Directory Branch Office Series Volume 2: Deployment and Operations 195 pages
- Microsoft Exchange 2000 Server Hosting Series Volume 1: Planning 227 pages
- Microsoft Exchange 2000 Server Hosting Series Volume 2: Deployment 135 pages
- Microsoft Exchange 2000 Server Upgrade Series Volume 1: Planning 306 pages
- Microsoft Exchange 2000 Server Upgrade Series Volume 2: Deployment 166 pages

### Reference Building Blocks

- Data Access Application Block for .NET 279 pages
- .NET Data Access Architecture Guide 60 pages
- Designing Data Tier Components and Passing Data Through Tiers 70 pages
- Exception Management Application Block for .NET 307 pages
- Exception Management in .NET 35 pages
- Monitoring in .NET Distributed Application Design 40 pages
- Microsoft .NET/COM Migration and Interoperability 35 pages
- Production Debugging for .NET-Connected Applications 176 pages
- Authentication in ASPNET: .NET Security Guidance 58 pages
- Building Secure ASPNET Applications: Authentication, Authorization, and Secure Communication 608 pages

### Operational Practices

- Security Operations Guide for Exchange 2000 Server 136 pages
- Security Operations for Microsoft Windows 2000 Server 188 pages
- Microsoft Exchange 2000 Server Operations Guide 113 pages
- Microsoft SQL Server 2000 Operations Guide 170 pages
- Deploying .NET Applications: Lifecycle Guide 142 pages
- Team Development with Visual Studio .NET and Visual SourceSafe 74 pages
- Backup and Restore for Internet Data Center 294 pages

**For current list of titles visit: [msdn.microsoft.com/practices](http://msdn.microsoft.com/practices)**

**To purchase *patterns & practices* guides visit: [shop.microsoft.com/practices](http://shop.microsoft.com/practices)**