Security in the .NET Framework

.NET Framework (current version)

The common language runtime and the .NET Framework provide many useful classes and services that enable developers to easily write secure code and enable system administrators to customize the permissions granted to code so that it can access protected resources. In addition, the runtime and the .NET Framework provide useful classes and services that facilitate the use of cryptography and role-based security.

In This Section

Security Changes in the .NET Framework
: Describes important changes to the .NET Framework security system.

Key Security Concepts
: Provides an overview of common language runtime security features. This section is of interest to developers and system administrators.

Role-Based Security
: Describes how to interact with role-based security in your code. This section is of interest to developers.

.NET Framework Cryptography Model
: Provides an overview of cryptographic services provided by the .NET Framework. This section is of interest to developers.

Secure Coding Guidelines
: Describes some of the best practices for creating reliable .NET Framework applications. This section is of interest to developers.

Secure Coding Guidelines for Unmanaged Code
: Describes some of the best practices and security concerns when calling unmanaged code.

Windows Identity Foundation
: Describes how you can implement claims-based identity in your applications.

Related Sections

.NET Framework Development Guide
: Provides a guide to all key technology areas and tasks for application development, including creating, configuring, debugging, securing, and deploying your application, and information about dynamic programming, interoperability, extensibility, memory management, and threading.

Code Access Security Basics

.NET Framework (current version)

Caution

Code Access Security and Partially Trusted Code

The .NET Framework provides a mechanism for the enforcement of varying levels of trust on different code running in the same application called Code Access Security (CAS). Code Access Security in .NET Framework should not be used as a mechanism for enforcing security boundaries based on code origination or other identity aspects. We are updating our guidance to reflect that Code Access Security and Security-Transparent Code will not be supported as a security boundary with partially trusted code, especially code of unknown origin. We advise against loading and executing code of unknown origins without putting alternative security measures in place.

This policy applies to all versions of .NET Framework, but does not apply to the .NET Framework included in Silverlight.

Every application that targets the common language runtime (that is, every managed application) must interact with the runtime's security system. When a managed application is loaded, its host automatically grants it a set of permissions. These permissions are determined by the host's local security settings or by the sandbox the application is in. Depending on these permissions, the application either runs properly or generates a security exception.

The default host for desktop applications allows code to run in full trust. Therefore, if your application targets the desktop, it has an unrestricted permission set. Other hosts or sandboxes provide a limited permission set for applications. Because the permission set can change from host to host, you must design your application to use only the permissions that your target host allows.

You must be familiar with the following code access security concepts in order to write effective applications that target the common language runtime:

- **Type-safe code**: Type-safe code is code that accesses types only in well-defined, allowable ways. For example, given a valid object reference, type-safe code can access memory at fixed offsets that correspond to actual field members. If the code accesses memory at arbitrary offsets outside the range of memory that belongs to that object's publicly exposed fields, it is not type-safe. To enable code to benefit from code access security, you must use a compiler that generates verifiably type-safe code. For more information, see the Writing Verifiably Type-Safe Code section later in this topic.

- **Imperative and declarative syntax**: Code that targets the common language runtime can interact with the security system by requesting permissions, demanding that callers have specified permissions, and overriding certain security settings (given enough privileges). You use two forms of syntax to programmatically interact with the .NET Framework security system: declarative syntax and imperative syntax. Declarative calls are performed using attributes; imperative calls are performed using new instances of classes within your code. Some calls can be performed only imperatively, others can be performed only declaratively, and some calls can be performed in either manner.

- **Secure class libraries**: A secure class library uses security demands to ensure that the library's callers have permission to access the resources that the library exposes. For example, a secure class library might have a method

for creating files that would demand that its callers have permissions to create files. The .NET Framework consists of secure class libraries. You should be aware of the permissions required to access any library that your code uses. For more information, see the Using Secure Class Libraries section later in this topic.

- **Transparent code**: Starting with the .NET Framework 4, in addition to identifying specific permissions, you must also determine whether your code should run as security-transparent. Security-transparent code cannot call types or members that are identified as security-critical. This rule applies to full-trust applications as well as partially trusted applications. For more information, see Security-Transparent Code.

Writing Verifiably Type-Safe Code

Just-in-time (JIT) compilation performs a verification process that examines code and tries to determine whether the code is type-safe. Code that is proven during verification to be type-safe is called *verifiably type-safe code*. Code can be type-safe, yet might not be verifiably type-safe because of the limitations of the verification process or of the compiler. Not all languages are type-safe, and some language compilers, such as Microsoft Visual C++, cannot generate verifiably type-safe managed code. To determine whether the language compiler you use generates verifiably type-safe code, consult the compiler's documentation. If you use a language compiler that generates verifiably type-safe code only when you avoid certain language constructs, you might want to use the PEVerify tool to determine whether your code is verifiably type-safe.

Code that is not verifiably type-safe can attempt to execute if security policy allows the code to bypass verification. However, because type safety is an essential part of the runtime's mechanism for isolating assemblies, security cannot be reliably enforced if code violates the rules of type safety. By default, code that is not type-safe is allowed to run only if it originates from the local computer. Therefore, mobile code should be type-safe.

Using Secure Class Libraries

If your code requests and is granted the permissions required by the class library, it will be allowed to access the library and the resources that the library exposes will be protected from unauthorized access. If your code does not have the appropriate permissions, it will not be allowed to access the class library, and malicious code will not be able to use your code to indirectly access protected resources. Other code that calls your code must also have permission to access the library. If it does not, your code will be restricted from running as well.

Code access security does not eliminate the possibility of human error in writing code. However, if your application uses secure class libraries to access protected resources, the security risk for application code is decreased, because class libraries are closely scrutinized for potential security problems.

Declarative Security

Declarative security syntax uses attributes to place security information into the metadata of your code. Attributes can be placed at the assembly, class, or member level, to indicate the type of request, demand, or override you want to use. Requests are used in applications that target the common language runtime to inform the runtime security system about the permissions that your application needs or does not want. Demands and overrides are used in libraries to help protect resources from callers or to override default security behavior.

Note

In the .NET Framework 4, there have been important changes to the .NET Framework security model and terminology. For more information about these changes, see Security Changes in the .NET Framework.

In order to use declarative security calls, you must initialize the state data of the permission object so that it represents the particular form of permission you need. Every built-in permission has an attribute that is passed a SecurityAction enumeration to describe the type of security operation you want to perform. However, permissions also accept their own parameters that are exclusive to them.

The following code fragment shows declarative syntax for requesting that your code's callers have a custom permission called `MyPermission`. This permission is a hypothetical custom permission and does not exist in the .NET Framework. In this example, the declarative call is placed directly before the class definition, specifying that this permission be applied to the class level. The attribute is passed a **SecurityAction.Demand** structure to specify that callers must have this permission in order to run.

VB

```
<MyPermission(SecurityAction.Demand, Unrestricted = True)> Public Class MyClass1

   Public Sub New()
      'The constructor is protected by the security call.
   End Sub


   Public Sub MyMethod()
      'This method is protected by the security call.
   End Sub


   Public Sub YourMethod()
      'This method is protected by the security call.
   End Sub
End Class
```

Imperative Security

Imperative security syntax issues a security call by creating a new instance of the permission object you want to invoke. You can use imperative syntax to perform demands and overrides, but not requests.

Before you make the security call, you must initialize the state data of the permission object so that it represents the particular form of the permission you need. For example, when creating a FileIOPermission object, you can use the constructor to initialize the **FileIOPermission** object so that it represents either unrestricted access to all files or no access to files. Or, you can use a different **FileIOPermission** object, passing parameters that indicate the type of access you want the object to represent (that is, read, append, or write) and what files you want the object to protect.

In addition to using imperative security syntax to invoke a single security object, you can use it to initialize a group of permissions in a permission set. For example, this technique is the only way to reliably perform assert calls on multiple permissions in one method. Use the PermissionSet and NamedPermissionSet classes to create a group of permissions and then call the appropriate method to invoke the desired security call.

You can use imperative syntax to perform demands and overrides, but not requests. You might use imperative syntax for demands and overrides instead of declarative syntax when information that you need in order to initialize the permission state becomes known only at run time. For example, if you want to ensure that callers have permission to read a certain file, but you do not know the name of that file until run time, use an imperative demand. You might also choose to use imperative checks instead of declarative checks when you need to determine at run time whether a condition holds and, based on the result of the test, make a security demand (or not).

The following code fragment shows imperative syntax for requesting that your code's callers have a custom permission called `MyPermission`. This permission is a hypothetical custom permission and does not exist in the .NET Framework. A new instance of `MyPermision` is created in `MyMethod`, guarding only this method with the security call.

VB

```
Public Class MyClass1

   Public Sub New()

   End Sub

   Public Sub MyMethod()
      'MyPermission is demanded using imperative syntax.
      Dim Perm As New MyPermission()
      Perm.Demand()
      'This method is protected by the security call.
   End Sub

   Public Sub YourMethod()
      'YourMethod 'This method is not protected by the security call.
   End Sub
End Class
```

Using Managed Wrapper Classes

Most applications and components (except secure libraries) should not directly call unmanaged code. There are several reasons for this. If code calls unmanaged code directly, it will not be allowed to run in many circumstances because code must be granted a high level of trust to call native code. If policy is modified to allow such an application to run, it can significantly weaken the security of the system, leaving the application free to perform almost any operation.

Additionally, code that has permission to access unmanaged code can probably perform almost any operation by calling into an unmanaged API. For example, code that has permission to call unmanaged code does not need FileIOPermission to access a file; it can just call an unmanaged (Win32) file API directly, bypassing the managed file API that requires **FileIOPermission**. If managed code has permission to call into unmanaged code and does call directly into unmanaged code, the security system will be unable to reliably enforce security restrictions, since the runtime cannot enforce such restrictions on unmanaged code.

If you want your application to perform an operation that requires accessing unmanaged code, it should do so through a trusted managed class that wraps the required functionality (if such a class exists). Do not create a wrapper class yourself if one already exists in a secure class library. The wrapper class, which must be granted a high degree of trust to be allowed to make the call into unmanaged code, is responsible for demanding that its callers have the appropriate permissions. If

you use the wrapper class, your code only needs to request and be granted the permissions that the wrapper class demands.

See Also

PermissionSet
FileIOPermission
NamedPermissionSet
SecurityAction
Using the Assert Method
Code Access Security
Code Access Security Basics
Extending Metadata Using Attributes
Metadata and Self-Describing Components

Role-Based Security

.NET Framework (current version)

Roles are often used in financial or business applications to enforce policy. For example, an application might impose limits on the size of the transaction being processed depending on whether the user making the request is a member of a specified role. Clerks might have authorization to process transactions that are less than a specified threshold, supervisors might have a higher limit, and vice-presidents might have a still higher limit (or no limit at all). Role-based security can also be used when an application requires multiple approvals to complete an action. Such a case might be a purchasing system in which any employee can generate a purchase request, but only a purchasing agent can convert that request into a purchase order that can be sent to a supplier.

.NET Framework role-based security supports authorization by making information about the principal, which is constructed from an associated identity, available to the current thread. The identity (and the principal it helps to define) can be either based on a Windows account or be a custom identity unrelated to a Windows account. .NET Framework applications can make authorization decisions based on the principal's identity or role membership, or both. A role is a named set of principals that have the same privileges with respect to security (such as a teller or a manager). A principal can be a member of one or more roles. Therefore, applications can use role membership to determine whether a principal is authorized to perform a requested action.

To provide ease of use and consistency with code access security, .NET Framework role-based security provides System.Security.Permissions.PrincipalPermission objects that enable the common language runtime to perform authorization in a way that is similar to code access security checks. The PrincipalPermission class represents the identity or role that the principal must match and is compatible with both declarative and imperative security checks. You can also access a principal's identity information directly and perform role and identity checks in your code when needed.

The .NET Framework provides role-based security support that is flexible and extensible enough to meet the needs of a wide spectrum of applications. You can choose to interoperate with existing authentication infrastructures, such as COM+ 1.0 Services, or to create a custom authentication system. Role-based security is particularly well-suited for use in ASP.NET Web applications, which are processed primarily on the server. However, .NET Framework role-based security can be used on either the client or the server.

Before reading this section, make sure that you understand the material presented in Key Security Concepts.

Related Topics

Title	Description
Principal and Identity Objects	Explains how to set up and manage both Windows and generic identities and principals.
Key Security Concepts	Introduces fundamental concepts you must understand before using .NET Framework security.

Reference

System.Security.Permissions.PrincipalPermission

Key Security Concepts

.NET Framework (current version)

The Microsoft .NET Framework offers role-based security to help address security concerns about mobile code and to provide support that enables components to determine what users are authorized to do.

Type safety and security

Type-safe code accesses only the memory locations it is authorized to access. (For this discussion, type safety specifically refers to memory type safety and should not be confused with type safety in a broader respect.) For example, type-safe code cannot read values from another object's private fields. It accesses types only in well-defined, allowable ways.

During just-in-time (JIT) compilation, an optional verification process examines the metadata and Microsoft intermediate language (MSIL) of a method to be JIT-compiled into native machine code to verify that they are type safe. This process is skipped if the code has permission to bypass verification. For more information about verification, see Managed Execution Process.

Although verification of type safety is not mandatory to run managed code, type safety plays a crucial role in assembly isolation and security enforcement. When code is type safe, the common language runtime can completely isolate assemblies from each other. This isolation helps ensure that assemblies cannot adversely affect each other and it increases application reliability. Type-safe components can execute safely in the same process even if they are trusted at different levels. When code is not type safe, unwanted side effects can occur. For example, the runtime cannot prevent managed code from calling into native (unmanaged) code and performing malicious operations. When code is type safe, the runtime's security enforcement mechanism ensures that it does not access native code unless it has permission to do so. All code that is not type safe must have been granted SecurityPermission with the passed enum member SkipVerification to run.

For more information, see Code Access Security Basics.

Principal

A principal represents the identity and role of a user and acts on the user's behalf. Role-based security in the .NET Framework supports three kinds of principals:

- Generic principals represent users and roles that exist independent of Windows users and roles.
- Windows principals represent Windows users and their roles (or their Windows groups). A Windows principal can impersonate another user, which means that the principal can access a resource on a user's behalf while presenting the identity that belongs to that user.
- Custom principals can be defined by an application in any way that is needed for that particular application. They can extend the basic notion of the principal's identity and roles.

For more information, see Principal and Identity Objects.

Authentication

Authentication is the process of discovering and verifying the identity of a principal by examining the user's credentials and validating those credentials against some authority. The information obtained during authentication is directly usable by your code. You can also use .NET Framework role-based security to authenticate the current user and to determine whether to allow that principal to access your code. See the overloads of the WindowsPrincipal.IsInRole method for examples of how to authenticate the principal for specific roles. For example, you can use the WindowsPrincipal.IsInRole(String) overload to determine if the current user is a member of the Administrators group.

A variety of authentication mechanisms are used today, many of which can be used with .NET Framework role-based security. Some of the most commonly used mechanisms are basic, digest, Passport, operating system (such as NTLM or Kerberos), or application-defined mechanisms.

Example

The following example requires that the active principal be an administrator. The *name* parameter is **null**, which allows any user who is an administrator to pass the demand.

Note

In Windows Vista, User Account Control (UAC) determines the privileges of a user. If you are a member of the Built-in Administrators group, you are assigned two run-time access tokens: a standard user access token and an administrator access token. By default, you are in the standard user role. To execute the code that requires you to be an administrator, you must first elevate your privileges from standard user to administrator. You can do this when you start an application by right-clicking the application icon and indicating that you want to run as an administrator.

VB

```
Imports System
Imports System.Threading
Imports System.Security.Permissions
Imports System.Security.Principal
```

```
Class SecurityPrincipalDemo


    Public Shared Sub Main()
        AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal)
        Dim principalPerm As New PrincipalPermission(Nothing, "Administrators")
        principalPerm.Demand()
        Console.WriteLine("Demand succeeded.")

    End Sub 'Main
End Class 'SecurityPrincipalDemo
```

The following example demonstrates how to determine the identity of the principal and the roles available to the principal. An application of this example might be to confirm that the current user is in a role you allow for using your application.

VB

```
Imports System
Imports System.Threading
Imports System.Security.Permissions
Imports System.Security.Principal

Class SecurityPrincipalDemo

    Public Shared Sub DemonstrateWindowsBuiltInRoleEnum()
        Dim myDomain As AppDomain = Thread.GetDomain()

        myDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal)
        Dim myPrincipal As WindowsPrincipal = CType(Thread.CurrentPrincipal,
WindowsPrincipal)
        Console.WriteLine("{0} belongs to: ", myPrincipal.Identity.Name.ToString())
        Dim wbirFields As Array = [Enum].GetValues(GetType(WindowsBuiltInRole))
        Dim roleName As Object
        For Each roleName In wbirFields
            Try
                ' Cast the role name to a RID represented by the WindowsBuildInRole
value.
                Console.WriteLine("{0}? {1}.", roleName,
myPrincipal.IsInRole(CType(roleName, WindowsBuiltInRole)))
                Console.WriteLine("The RID for this role is: " +
Fix(roleName).ToString())

            Catch
                Console.WriteLine("{0}: Could not obtain role for this RID.",
roleName)
            End Try
        Next roleName
        ' Get the role using the string value of the role.
        Console.WriteLine("{0}? {1}.", "Administrators",
```

```
myPrincipal.IsInRole("BUILTIN\" + "Administrators"))
        Console.WriteLine("{0}? {1}.", "Users", myPrincipal.IsInRole("BUILTIN\" +
"Users"))
        ' Get the role using the WindowsBuiltInRole enumeration value.
        Console.WriteLine("{0}? {1}.", WindowsBuiltInRole.Administrator,
myPrincipal.IsInRole(WindowsBuiltInRole.Administrator))
        ' Get the role using the WellKnownSidType.
        Dim sid As New SecurityIdentifier(WellKnownSidType.BuiltinAdministratorsSid,
Nothing)
        Console.WriteLine("WellKnownSidType BuiltinAdministratorsSid  {0}? {1}.",
sid.Value, myPrincipal.IsInRole(sid))

    End Sub 'DemonstrateWindowsBuiltInRoleEnum

    Public Shared Sub Main()
        DemonstrateWindowsBuiltInRoleEnum()

    End Sub 'Main
End Class 'SecurityPrincipalDemo
```

Authorization

Authorization is the process of determining whether a principal is allowed to perform a requested action. Authorization occurs after authentication and uses information about the principal's identity and roles to determine what resources the principal can access. You can use .NET Framework role-based security to implement authorization.

Principal and Identity Objects

.NET Framework (current version)

Managed code can discover the identity or the role of a principal through a Principal object, which contains a reference to an Identity object. It might be helpful to compare identity and principal objects to familiar concepts like user and group accounts. In most network environments, user accounts represent people or programs, while group accounts represent certain categories of users and the rights they possess. Similarly, .NET Framework identity objects represent users, while roles represent memberships and security contexts. In the .NET Framework, the principal object encapsulates both an identity object and a role. .NET Framework applications grant rights to the principal based on its identity or, more commonly, its role membership.

Identity Objects

The identity object encapsulates information about the user or entity being validated. At their most basic level, identity objects contain a name and an authentication type. The name can either be a user's name or the name of a Windows account, while the authentication type can be either a supported logon protocol, such as Kerberos V5, or a custom value. The .NET Framework defines a GenericIdentity object that can be used for most custom logon scenarios and a more specialized WindowsIdentity object that can be used when you want your application to rely on Windows authentication. Additionally, you can define your own identity class that encapsulates custom user information.

The IIdentity interface defines properties for accessing a name and an authentication type, such as Kerberos V5 or NTLM. All **Identity** classes implement the **IIdentity** interface. There is no required relationship between an **Identity** object and the Windows NT process token under which a thread is currently executing. However, if the **Identity** object is a **WindowsIdentity** object, the identity is assumed to represent a Windows NT security token.

Principal Objects

The principal object represents the security context under which code is running. Applications that implement role-based security grant rights based on the role associated with a principal object. Similar to identity objects, the .NET Framework provides a GenericPrincipal object and a WindowsPrincipal object. You can also define your own custom principal classes.

The IPrincipal interface defines a property for accessing an associated **Identity** object as well as a method for determining whether the user identified by the **Principal** object is a member of a given role. All **Principal** classes implement the **IPrincipal** interface as well as any additional properties and methods that are necessary. For example, the common language runtime provides the **WindowsPrincipal** class, which implements additional functionality for mapping Windows NT or Windows 2000 group membership to roles.

A **Principal** object is bound to a call context (CallContext) object within an application domain (AppDomain). A default call context is always created with each new **AppDomain**, so there is always a call context available to accept the **Principal** object. When a new thread is created, a **CallContext** object is also created for the thread. The **Principal** object reference is automatically copied from the creating thread to the new thread's **CallContext**. If the runtime cannot determine which **Principal** object belongs to the creator of the thread, it follows the default policy for **Principal** and **Identity** object creation.

A configurable application domain-specific policy defines the rules for deciding what type of **Principal** object to

associate with a new application domain. Where security policy permits, the runtime can create **Principal** and **Identity** objects that reflect the operating system token associated with the current thread of execution. By default, the runtime uses **Principal** and **Identity** objects that represent unauthenticated users. The runtime does not create these default **Principal** and **Identity** objects until the code attempts to access them.

Trusted code that creates an application domain can set the application domain policy that controls construction of the default **Principal** and **Identity** objects. This application domain-specific policy applies to all execution threads in that application domain. An unmanaged, trusted host inherently has the ability to set this policy, but managed code that sets this policy must have the System.Security.Permissions.SecurityPermission for controlling domain policy.

When transmitting a **Principal** object across application domains but within the same process (and therefore on the same computer), the remoting infrastructure copies a reference to the **Principal** object associated with the caller's context to the callee's context.

See Also

How to: Create a WindowsPrincipal Object
How to: Create GenericPrincipal and GenericIdentity Objects
Replacing a Principal Object
Impersonating and Reverting
Role-Based Security
Key Security Concepts