.NET Coding Standards (C#)

Coding conventions serve the following purposes:

* They create a consistent look to the code, so that readers can focus on content, not layout.
* They enable readers to understand the code more quickly by making assumptions based on previous experience.
* They facilitate copying, changing, and maintaining the code.
* They demonstrate C# best practices.

The guidelines in this topic are used by Microsoft to develop samples and documentation.

**Naming Conventions**

* In short examples that do not include [using directives](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-directive), use namespace qualifications. If you know that a namespace is imported by default in a project, you do not have to fully qualify the names from that namespace. Qualified names can be broken after a dot (.) if they are too long for a single line, as shown in the following example.

var currentPerformanceCounterCategory = new System.Diagnostics.

PerformanceCounterCategory();

* You do not have to change the names of objects that were created by using the Visual Studio designer tools to make them fit other guidelines.

**Layout Conventions**

Good layout uses formatting to emphasize the structure of your code and to make the code easier to read. Microsoft examples and samples conform to the following conventions:

* Use the default Code Editor settings (smart indenting, four-character indents, tabs saved as spaces). For more information, see [Options, Text Editor, C#, Formatting](https://docs.microsoft.com/en-us/visualstudio/ide/reference/options-text-editor-csharp-formatting).
* Write only one statement per line.
* Write only one declaration per line.
* If continuation lines are not indented automatically, indent them one tab stop (four spaces).
* Add at least one blank line between method definitions and property definitions.
* Use parentheses to make clauses in an expression apparent, as shown in the following code.

if ((val1 > val2) && (val1 > val3))

{

// Take appropriate action.

}

**Commenting Conventions**

* Place the comment on a separate line, not at the end of a line of code.
* Begin comment text with an uppercase letter.
* End comment text with a period.
* Insert one space between the comment delimiter (//) and the comment text, as shown in the following example.

// The following declaration creates a query. It does not run

// the query.

* Do not create formatted blocks of asterisks around comments.

**Language Guidelines**

The following sections describe practices that the C# team follows to prepare code examples and samples.

**String Data Type**

* Use [string interpolation](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/tokens/interpolated) to concatenate short strings, as shown in the following code.

string displayName = $"{nameList[n].LastName}, {nameList[n].FirstName}";

* To append strings in loops, especially when you are working with large amounts of text, use a [StringBuilder](https://docs.microsoft.com/en-us/dotnet/api/system.text.stringbuilder) object.

var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";

var manyPhrases = new StringBuilder();

for (var i = 0; i < 10000; i++)

{

manyPhrases.Append(phrase);

}

//Console.WriteLine("tra" + manyPhrases);

**Implicitly Typed Local Variables**

* Use [implicit typing](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/implicitly-typed-local-variables) for local variables when the type of the variable is obvious from the right side of the assignment, or when the precise type is not important.

// When the type of a variable is clear from the context, use var

// in the declaration.

var var1 = "This is clearly a string.";

var var2 = 27;

var var3 = Convert.ToInt32(Console.ReadLine());

* Do not use [var](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/var) when the type is not apparent from the right side of the assignment.

// When the type of a variable is not clear from the context, use an

// explicit type.

int var4 = ExampleClass.ResultSoFar();

* Do not rely on the variable name to specify the type of the variable. It might not be correct.

// Naming the following variable inputInt is misleading.

// It is a string.

var inputInt = Console.ReadLine();

Console.WriteLine(inputInt);

* Avoid the use of var in place of [dynamic](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/dynamic).
* Use implicit typing to determine the type of the loop variable in [for](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/for) and [foreach](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/foreach-in) loops.

The following example uses implicit typing in a for statement.

var syllable = "ha";

var laugh = "";

for (var i = 0; i < 10; i++)

{

laugh += syllable;

Console.WriteLine(laugh);

}

The following example uses implicit typing in a foreach statement.

foreach (var ch in laugh)

{

if (ch == 'h')

Console.Write("H");

else

Console.Write(ch);

}

Console.WriteLine();

**Unsigned Data Type**

* In general, use int rather than unsigned types. The use of int is common throughout C#, and it is easier to interact with other libraries when you use int.

**Arrays**

* Use the concise syntax when you initialize arrays on the declaration line.

// Preferred syntax. Note that you cannot use var here instead of string[].

string[] vowels1 = { "a", "e", "i", "o", "u" };

// If you use explicit instantiation, you can use var.

var vowels2 = new string[] { "a", "e", "i", "o", "u" };

// If you specify an array size, you must initialize the elements one at a time.

var vowels3 = new string[5];

vowels3[0] = "a";

vowels3[1] = "e";

// And so on.

**Delegates**

* Use the concise syntax to create instances of a delegate type.

// First, in class Program, define the delegate type and a method that

// has a matching signature.

// Define the type.

public delegate void Del(string message);

// Define a method that has a matching signature.

public static void DelMethod(string str)

{

Console.WriteLine("DelMethod argument: {0}", str);

}

// In the Main method, create an instance of Del.

// Preferred: Create an instance of Del by using condensed syntax.

Del exampleDel2 = DelMethod;

// The following declaration uses the full syntax.

Del exampleDel1 = new Del(DelMethod);

**try-catch and using Statements in Exception Handling**

* Use a [try-catch](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch) statement for most exception handling.

static string GetValueFromArray(string[] array, int index)

{

try

{

return array[index];

}

catch (System.IndexOutOfRangeException ex)

{

Console.WriteLine("Index is out of range: {0}", index);

throw;

}

}

* Simplify your code by using the C# [using statement](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-statement). If you have a [try-finally](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-finally) statement in which the only code in the finally block is a call to the [Dispose](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable.dispose) method, use a using statement instead.

// This try-finally statement only calls Dispose in the finally block.

Font font1 = new Font("Arial", 10.0f);

try

{

byte charset = font1.GdiCharSet;

}

finally

{

if (font1 != null)

{

((IDisposable)font1).Dispose();

}

}

// You can do the same thing with a using statement.

using (Font font2 = new Font("Arial", 10.0f))

{

byte charset = font2.GdiCharSet;

}

**&& and || Operators**

* To avoid exceptions and increase performance by skipping unnecessary comparisons, use [&&](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/conditional-and-operator) instead of [&](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/and-operator) and [||](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/conditional-or-operator) instead of [|](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/or-operator) when you perform comparisons, as shown in the following example.

Console.Write("Enter a dividend: ");

var dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");

var divisor = Convert.ToInt32(Console.ReadLine());

// If the divisor is 0, the second clause in the following condition

// causes a run-time error. The && operator short circuits when the

// first expression is false. That is, it does not evaluate the

// second expression. The & operator evaluates both, and causes

// a run-time error when divisor is 0.

if ((divisor != 0) && (dividend / divisor > 0))

{

Console.WriteLine("Quotient: {0}", dividend / divisor);

}

else

{

Console.WriteLine("Attempted division by 0 ends up here.");

}

**New Operator**

* Use the concise form of object instantiation, with implicit typing, as shown in the following declaration.

var instance1 = new ExampleClass();

The previous line is equivalent to the following declaration.

ExampleClass instance2 = new ExampleClass();

* Use object initializers to simplify object creation.

// Object initializer.

var instance3 = new ExampleClass { Name = "Desktop", ID = 37414,

Location = "Redmond", Age = 2.3 };

// Default constructor and assignment statements.

var instance4 = new ExampleClass();

instance4.Name = "Desktop";

instance4.ID = 37414;

instance4.Location = "Redmond";

instance4.Age = 2.3;

**Event Handling**

* If you are defining an event handler that you do not need to remove later, use a lambda expression.

public Form2()

{

// You can use a lambda expression to define an event handler.

this.Click += (s, e) =>

{

MessageBox.Show(

((MouseEventArgs)e).Location.ToString());

};

}

// Using a lambda expression shortens the following traditional definition.

public Form1()

{

this.Click += new EventHandler(Form1\_Click);

}

void Form1\_Click(object sender, EventArgs e)

{

MessageBox.Show(((MouseEventArgs)e).Location.ToString());

}

**Static Members**

* Call [static](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static) members by using the class name: *ClassName.StaticMember*. This practice makes code more readable by making static access clear. Do not qualify a static member defined in a base class with the name of a derived class. While that code compiles, the code readability is misleading, and the code may break in the future if you add a static member with the same name to the derived class.

**LINQ Queries**

* Use meaningful names for query variables. The following example uses seattleCustomers for customers who are located in Seattle.

var seattleCustomers = from cust in customers

where cust.City == "Seattle"

select cust.Name;

* Use aliases to make sure that property names of anonymous types are correctly capitalized, using Pascal casing.

var localDistributors =

from customer in customers

join distributor in distributors on customer.City equals distributor.City

select new { Customer = customer, Distributor = distributor };

* Rename properties when the property names in the result would be ambiguous. For example, if your query returns a customer name and a distributor ID, instead of leaving them as Name and ID in the result, rename them to clarify that Name is the name of a customer, and ID is the ID of a distributor.

var localDistributors2 =

from cust in customers

join dist in distributors on cust.City equals dist.City

select new { CustomerName = cust.Name, DistributorID = dist.ID };

* Use implicit typing in the declaration of query variables and range variables.

var seattleCustomers = from cust in customers

where cust.City == "Seattle"

select cust.Name;

* Align query clauses under the [from](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/from-clause) clause, as shown in the previous examples.
* Use [where](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/where-clause) clauses before other query clauses to ensure that later query clauses operate on the reduced, filtered set of data.

var seattleCustomers2 = from cust in customers

where cust.City == "Seattle"

orderby cust.Name

select cust;

* Use multiple from clauses instead of a [join](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/join-clause) clause to access inner collections. For example, a collection of Student objects might each contain a collection of test scores. When the following query is executed, it returns each score that is over 90, along with the last name of the student who received the score.

// Use a compound from to access the inner sequence within each element.

var scoreQuery = from student in students

from score in student.Scores

where score > 90

select new { Last = student.LastName, score };

**Security**

Evidence-based security and code access security provide very powerful, explicit mechanisms to implement security. Most application code can simply use the infrastructure implemented by .NET. In some cases, additional application-specific security is required, built either by extending the security system or by using new ad hoc methods.

Using .NET enforced permissions and other enforcement in your code, you should erect barriers to prevent malicious code from accessing information that you don't want it to have or performing other undesirable actions. Additionally, you must strike a balance between security and usability in all the expected scenarios using trusted code.

This overview describes the different ways code can be designed to work with the security system.

**Securing resource access**

When designing and writing your code, you need to protect and limit the access that code has to resources, especially when using or invoking code of unknown origin. So, keep in mind the following techniques to ensure your code is secure:

* Do not use Code Access Security (CAS).
* Do not use partial trusted code.
* Do not use the [AllowPartiallyTrustedCaller](https://docs.microsoft.com/en-us/dotnet/api/system.security.allowpartiallytrustedcallersattribute) attribute (APTCA).
* Do not use .NET Remoting.
* Do not use Distributed Component Object Model (DCOM).
* Do not use binary formatters.

Code Access Security and Security-Transparent Code are not supported as a security boundary with partially trusted code. We advise against loading and executing code of unknown origins without putting alternative security measures in place. The alternative security measures are:

* Virtualization
* AppContainers
* Operating system (OS) users and permissions
* Hyper-V containers

**Security-neutral code**

Security-neutral code does nothing explicit with the security system. It runs with whatever permissions it receives. Although applications that fail to catch security exceptions associated with protected operations (such as using files, networking, and so on) can result in an unhandled exception, security-neutral code still takes advantage of the security technologies in .NET.

A security-neutral library has special characteristics that you should understand. Suppose your library provides API elements that use files or call unmanaged code. If your code doesn't have the corresponding permission, it won't run as described. However, even if the code has the permission, any application code that calls it must have the same permission in order to work. If the calling code doesn't have the right permission, a [SecurityException](https://docs.microsoft.com/en-us/dotnet/api/system.security.securityexception) appears as a result of the code access security stack walk.

**Application code that isn't a reusable component**

If your code is part of an application that won't be called by other code, security is simple and special coding might not be required. However, remember that malicious code can call your code. While code access security might stop malicious code from accessing resources, such code could still read values of your fields or properties that might contain sensitive information.

Additionally, if your code accepts user input from the Internet or other unreliable sources, you must be careful about malicious input.

**Managed wrapper to native code implementation**

Typically in this scenario, some useful functionality is implemented in native code that you want to make available to managed code. Managed wrappers are easy to write using either platform invoke or COM interop. However, if you do this, callers of your wrappers must have unmanaged code rights in order to succeed. Under default policy, this means that code downloaded from an intranet or the Internet won't work with the wrappers.

Instead of giving unmanaged code rights to all applications that use these wrappers, it's better to give these rights only to the wrapper code. If the underlying functionality exposes no resources and the implementation is likewise safe, the wrapper only needs to assert its rights, which enables any code to call through it. When resources are involved, security coding should be the same as the library code case described in the next section. Because the wrapper is potentially exposing callers to these resources, careful verification of the safety of the native code is necessary and is the wrapper's responsibility.

**Library code that exposes protected resources**

The following approach is the most powerful and hence potentially dangerous (if done incorrectly) for security coding: your library serves as an interface for other code to access certain resources that aren't otherwise available, just as the .NET classes enforce permissions for the resources they use. Wherever you expose a resource, your code must first demand the permission appropriate to the resource (that is, it must perform a security check) and then typically assert its rights to perform the actual operation.

Securing State Data

Applications that handle sensitive data or make any kind of security decisions need to keep that data under their own control and cannot allow other potentially malicious code to access the data directly. The best way to protect data in memory is to declare the data as private or internal (with scope limited to the same assembly) variables. However, even this data is subject to access you should be aware of:

* Using reflection mechanisms, highly trusted code that can reference your object can get and set private members.
* Using serialization, highly trusted code can effectively get and set private members if it can access the corresponding data in the serialized form of the object.
* Under debugging, this data can be read.

Make sure none of your own methods or properties exposes these values unintentionally.

Securing user Input

User data, which is any kind of input (data from a Web request or URL, input to controls of a Microsoft Windows Forms application, and so on), can adversely influence code because often that data is used directly as parameters to call other code. This situation is analogous to malicious code calling your code with strange parameters, and the same precautions should be taken. User input is actually harder to make safe because there is no stack frame to trace the presence of the potentially untrusted data.

These are among the subtlest and hardest security bugs to find because, although they can exist in code that is seemingly unrelated to security, they are a gateway to pass bad data through to other code. To look for these bugs, follow any kind of input data, imagine what the range of possible values might be, and consider whether the code seeing this data can handle all those cases. You can fix these bugs through range checking and rejecting any input the code cannot handle.

Some important considerations involving user data include the following:

* Any user data in a server response runs in the context of the server's site on the client. If your Web server takes user data and inserts it into the returned Web page, it might, for example, include a **<script>** tag and run as if from the server.
* Remember that the client can request any URL.
* Consider tricky or invalid paths:
  + ..\ , extremely long paths.
  + Use of wild card characters (\*).
  + Token expansion (%token%).
  + Strange forms of paths with special meaning.
  + Alternate file system stream names such as filename::$DATA.
  + Short versions of file names such as longfi~1 for longfilename.
* Remember that Eval(userdata) can do anything.
* Be wary of late binding to a name that includes some user data.
* If you are dealing with Web data, consider the various forms of escapes that are permissible, including:
  + Hexadecimal escapes (%nn).
  + Unicode escapes (%nnn).
  + Overlong UTF-8 escapes (%nn%nn).
  + Double escapes (%nn becomes %mmnn, where %mm is the escape for '%').
* Be wary of user names that might have more than one canonical format. For example, you can often use either the MYDOMAIN\*username* form or the *username*@mydomain.example.com form.

Security & Race Conditions

Another area of concern is the potential for security holes exploited by race conditions. There are several ways in which this might happen. The subtopics that follow outline some of the major pitfalls that the developer must avoid.

## Race Conditions in the Dispose Method

If a class's **Dispose** method (for more information, see [Garbage Collection](https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/index)) is not synchronized, it is possible that cleanup code inside **Dispose** can be run more than once, as shown in the following example.

void Dispose()

{

if (myObj != null)

{

Cleanup(myObj);

myObj = null;

}

}

Because this **Dispose** implementation is not synchronized, it is possible for Cleanup to be called by first one thread and then a second thread before \_myObj is set to **null**. Whether this is a security concern depends on what happens when the Cleanup code runs. A major issue with unsynchronized **Dispose** implementations involves the use of resource handles such as files. Improper disposal can cause the wrong handle to be used, which often leads to security vulnerabilities.

## Race Conditions in Constructors

In some applications, it might be possible for other threads to access class members before their class constructors have completely run. You should review all class constructors to make sure that there are no security issues if this should happen, or synchronize threads if necessary.

## Race Conditions with Cached Objects

Code that caches security information or uses the code access security [Assert](https://docs.microsoft.com/en-us/dotnet/framework/misc/using-the-assert-method) operation might also be vulnerable to race conditions if other parts of the class are not appropriately synchronized, as shown in the following example.

void SomeSecureFunction()

{

if (SomeDemandPasses())

{

fCallersOk = true;

DoOtherWork();

fCallersOk = false;

}

}

void DoOtherWork()

{

if (fCallersOK)

{

DoSomethingTrusted();

}

else

{

DemandSomething();

DoSomethingTrusted();

}

}

If there are other paths to DoOtherWork that can be called from another thread with the same object, an untrusted caller can slip past a demand.

If your code caches security information, make sure that you review it for this vulnerability.

## Race Conditions in Finalizers

Race conditions can also occur in an object that references a static or unmanaged resource that it then frees in its finalizer. If multiple objects share a resource that is manipulated in a class's finalizer, the objects must synchronize all access to that resource.

Security and on-the-fly Race Conditions

Some libraries operate by generating code and running it to perform some operation for the caller. The basic problem is generating code on behalf of lesser-trust code and running it at a higher trust. The problem worsens when the caller can influence code generation, so you must ensure that only code you consider safe is generated.

You need to know exactly what code you are generating at all times. This means that you must have strict controls on any values that you get from a user, be they quote-enclosed strings (which should be escaped so they cannot include unexpected code elements), identifiers (which should be checked to verify that they are valid identifiers), or anything else. Identifiers can be dangerous because a compiled assembly can be modified so that its identifiers contain strange characters, which will probably break it (although this is rarely a security vulnerability).

It is recommended that you generate code with reflection emit, which often helps you avoid many of these problems.

When you compile the code, consider whether there is some way a malicious program could modify it. Is there a small window of time during which malicious code can change source code on disk before the compiler reads it or before your code loads the .dll file? If so, you must protect the directory containing these files, using an Access Control List in the file system, as appropriate.

Role Based Security

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](https://docs.microsoft.com/en-us/dotnet/api/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](https://docs.microsoft.com/en-us/dotnet/api/system.security.permissions.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](https://docs.microsoft.com/en-us/dotnet/standard/security/key-security-concepts).

Key Security Concepts

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](https://docs.microsoft.com/en-us/dotnet/standard/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](https://docs.microsoft.com/en-us/dotnet/api/system.security.permissions.securitypermission) with the passed enum member [SkipVerification](https://docs.microsoft.com/en-us/dotnet/api/system.security.permissions.securitypermissionattribute.skipverification) to run.

For more information, see [Code Access Security Basics](https://docs.microsoft.com/en-us/dotnet/framework/misc/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](https://docs.microsoft.com/en-us/dotnet/standard/security/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](https://docs.microsoft.com/en-us/dotnet/api/system.security.principal.windowsprincipal.isinrole) method for examples of how to authenticate the principal for specific roles. For example, you can use the [WindowsPrincipal.IsInRole(String)](https://docs.microsoft.com/en-us/dotnet/api/system.security.principal.windowsprincipal.isinrole" \l "System_Security_Principal_WindowsPrincipal_IsInRole_System_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 nameparameter 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.

using System;

using System.Threading;

using System.Security.Permissions;

using System.Security.Principal;

class SecurityPrincipalDemo

{

public static void Main()

{

AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);

PrincipalPermission principalPerm = new PrincipalPermission(null, "Administrators");

principalPerm.Demand();

Console.WriteLine("Demand succeeded.");

}

}

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.

using System;

using System.Threading;

using System.Security.Permissions;

using System.Security.Principal;

class SecurityPrincipalDemo

{

public static void DemonstrateWindowsBuiltInRoleEnum()

{

AppDomain myDomain = Thread.GetDomain();

myDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);

WindowsPrincipal myPrincipal = (WindowsPrincipal)Thread.CurrentPrincipal;

Console.WriteLine("{0} belongs to: ", myPrincipal.Identity.Name.ToString());

Array wbirFields = Enum.GetValues(typeof(WindowsBuiltInRole));

foreach (object roleName in wbirFields)

{

try

{

// Cast the role name to a RID represented by the WindowsBuildInRole value.

Console.WriteLine("{0}? {1}.", roleName,

myPrincipal.IsInRole((WindowsBuiltInRole)roleName));

Console.WriteLine("The RID for this role is: " + ((int)roleName).ToString());

}

catch (Exception)

{

Console.WriteLine("{0}: Could not obtain role for this RID.",

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.

SecurityIdentifier sid = new SecurityIdentifier(WellKnownSidType.BuiltinAdministratorsSid, null);

Console.WriteLine("WellKnownSidType BuiltinAdministratorsSid {0}? {1}.", sid.Value, myPrincipal.IsInRole(sid));

}

public static void Main()

{

DemonstrateWindowsBuiltInRoleEnum();

}

}

**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.