Key Security Concepts

Article • 05/26/2022 • 6 minutes to read

> **Note**
>
> This article applies to Windows.
>
> For information about ASP.NET Core, see **Overview of ASP.NET Core Security**.

.NET 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.

> **Note**
>
> Code Access Security (CAS) has been deprecated across all versions of .NET Framework and .NET. Recent versions of .NET do not honor CAS annotations and produce errors if CAS-related APIs are used. Developers should seek alternative means of accomplishing security tasks.

Principal

A principal represents the identity and role of a user and acts on the user's behalf. Role-based security in .NET 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 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 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.

C#

```
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.

C#

```
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 role-based security to implement authorization.

See also

- ASP.NET Core Security