.NET Security

JASON BOCK, PETE STROMQUIST,
TOM FISCHER, AND NATHAN SMITH

Apress™

.NET Security

ISBN (pbk): 1-59059-053-8
Printed and bound in the United States of America 12345678910

Technical Reviewers: Brock Allen, Robert Knudson, and Chris Sells
Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell, Jason Gilmore, Karen Watterson, John Zukowski
Managing Editor: Grace Wong
Copy Editor: Ami Knox
Compositor: Impressions Book and Journal Services, Inc.
Indexer: Valerie Haynes Perry
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski
Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010
and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.
Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone: 510-549-5930, Fax: 510-549-5939, Email: info@apress.com, Web site: http://www.apress.com

The source code for this book is available to readers at http://www.apress.com in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 5

Role Access Security

In this chapter, you'll investigate how to use Role Access Security (RAS) to protect resources. You'll study the definitions that are found in .NET and how they work. Finally, you'll learn about impersonation and how it works in .NET.

Let's start by looking at the .NET types that are used in RAS from a coding perspective.

Using .NET Class Definitions

All of the .NET class definitions types are in the `System.Security.Principal` namespace, and all can be found in the mscorlib assembly. Some classes define what identities and principals there are along with their relationship to each other. Other classes exist that help in defining how identities and principals are established for the current thread of execution. In this section, I'll cover the basics of these classes.

Let's start by looking at the definition of an identity in .NET.

IIdentity Interface

The `IIdentity` interface is used to define information about a user. Typically, that user is running the current code, although there's no requirement that a class that implements this interface has to relate to the underlying OS user. `IIdentity` defines three read-only properties:

- `Name`: A string that defines the logical name of the user. This can take on any form—a typical format is "domain\user" (for example, "apress\jbock").

- `AuthenticationType`: A string that defines the kind of authentication used to verify the user. Examples of common authentication schemes are Kerebos and Passport.

- `IsAuthenticated`: A Boolean that states whether the current user has been authenticated.

.NET supplies the four following implementations of `IIdentity`:

- `GenericIdentity`
- `WindowsIdentity`
- `FormsIdentity`
- `PassportIdentity`

Let's take a quick dive into these implementations.

GenericIdentity

As its name implies, `GenericIdentity` is not directly tied to any protocol to authenticate the identity in question. Creating a `GenericIdentity` object is pretty easy, as you can see here:

```
IIdentity genericNameOnly =
    new GenericIdentity("jbock");
IIdentity genericNoNameOnly =
    new GenericIdentity("");
IIdentity genericNameAndTypeK =
    new GenericIdentity("jbock", "Kerebos");
IIdentity genericNameAndTypeNT =
    new GenericIdentity("jbock", "NTLM");
```

You have two options for object construction. You can simply pass in a user name, or you can pass in a user name plus an authentication type.

Although it looks like the last two objects (`genericNameAndTypeK` and `genericNameAndTypeNT`) are using some kind of authentication protocol, `GenericIdentity` doesn't use that last argument value to authenticate the given user. In fact, if you ran this code in a console application and called `IsAuthenticated` on each object:

```
Console.WriteLine("Name only - is it authenticated?  " +
    genericNameOnly.IsAuthenticated.ToString());
Console.WriteLine("No name only - is it authenticated?  " +
    genericNoNameOnly.IsAuthenticated.ToString());
```

```
Console.WriteLine("Name + Kerebos - is it authenticated?  " +
    genericNameAndTypeK.IsAuthenticated.ToString());
Console.WriteLine("Name + NT - is it authenticated?  " +
    genericNameAndTypeNT.IsAuthenticated.ToString());
```

you'd get the following results:

```
Name only - is it authenticated?  True
No name only - is it authenticated?  False
Name + Kerebos - is it authenticated?  True
Name + NT - is it authenticated?  True
```

`GenericIdentity` only "authenticates" the identity if a user name is given.

WindowsIdentity

As the name suggests, `WindowsIdentity` is all about identifying a specific Windows user. The biggest difference between `WindowsIdentity` and `GenericIdentity` is in the construction of the object. `WindowsIdentity` has four constructors, and all of them have as its first arguments an `IntPtr` type, as you can see here:

```
IntPtr windowsToken /* =...*/;
IIdentity windowsUser =
    new WindowsIdentity(windowsToken, "NTLM",
        WindowsAccountType.Normal, true);
```

This example shows the constructor that takes the most arguments (the other constructors are just simplified versions of this one). The second argument is the authentication type. The third argument is a `WindowsAccountType` enumeration value that identifies the kind of account the identity is. It has the following four values:

- `Anonymous`: An anonymous account
- `Guest`: A guest account
- `Normal`: A regular Windows account
- `System`: A system account

The fourth parameter allows the client to specify if the user represented by the first argument is authenticated or not.

As the preceding code snippet suggests, you need to do some extra work to set the identity object to a user. (I'll show you later on in the section "Windows Impersonation" how to set the `windowsToken` value appropriately.) However, you can get the current Windows user's information by calling the static method `GetCurrent()`, like this:

```
IIdentity currentWindowsUser = WindowsIdentity.GetCurrent();
```

The Name value will be of the format "domain\user" and the authentication type will be "NTLM". The user should also be authenticated.

If you want to get an anonymous Windows user, you could use the static method called `GetAnonymous()` as follows:

```
IIdentity anonWindowsUser = WindowsIdentity.GetAnonymous();
```

In this case, there's no user name or authentication type, nor will the anonymous user be authenticated.

FormsIdentity and PassportIdentity

For the sake of completeness, I'll mention that there are two other `IIdentity` implementations in .NET: `FormsIdentity` and `PassportIdentity`. `FormsIdentity` is primarily used in ASP.NET applications, and `PassportIdentity` represents the user authenticated from a Passport. I won't delve into their workings here; I'll revisit these types in Chapters 7 and 8.

Now that you have a good understanding of how identities work in .NET, let's move on to the `IPrincipal` interface.

IPrincipal Interface

This interface is also pretty bare bones—it has two members:

- `Identity`: A read-only property to get the identity related to the principal
- `IsInRole`: A method that informs you if the principal is a member of a given role.

.NET provides two implementations of `IPrincipal`: `GenericPrincipal` and `WindowsPrincipal`, and I'll discuss these next.

GenericPrincipal

GenericPrincipal is a simplistic implementation of IPrincipal. To create a GenericPrincipal object, you have to give it an IIdentity-based object along with a list of roles that the identity belongs to. Here's an example:

```
string[] roles = {"administrators", "developers"};
IIdentity genericNameOnly = new GenericIdentity("jbock");
GenericPrincipal genericPrincipal =
    new GenericPrincipal(genericNameOnly, roles);

Console.WriteLine("Is the principal in the developer's role?  " +
    genericPrincipal.IsInRole("developers"));
Console.WriteLine("Is the principal in the accountant's role?  " +
    genericPrincipal.IsInRole("accountants"));
```

In this case, the console results would be as follows:

```
Is the principal in the developer's role?  True
Is the principal in the accountant's role  False
```

WindowsPrincipal

WindowsPrincipal is almost identical in looks to GenericPrincipal, except for its constructor. It will only take a WindowsIdentity instance as shown here:

```
WindowsPrincipal winPrincipal =
    new WindowsPrincipal(WindowsIdentity.GetCurrent());
Console.WriteLine("Is the principal in the administrator's role?  " +
    winPrincipal.IsInRole(@"weinstefaner\Administrators"));
```

Now, given that the box that this code is running on is called weinstefaner and I'm in the Adminstrators group, what do you think the console will show this time? Here's the answer:

```
Is the principal in the administrator's role?  False
```

This surprised me the first time I saw it. After doing some digging, I found out that you have to use the string `BUILTIN` in place of the machine name:

```
Console.WriteLine("Is the principal in the administrator's role?  " +
    winPrincipal.IsInRole(@"BUILTIN\Administrators"));
```

This will work, but this is also a little unintuitive, especially if you're not very familiar with how Windows security works.[1] Fortunately, `WindowsPrincipal` overloads `IsInRole()` to make this check easier to manage. One overload takes a `WindowsBuiltInRole` enumeration, which defines the following standard Windows roles:

- `AccountOperator`
- `Administrator`
- `BackupOperator`
- `Guest`
- `PowerUser`
- `PrintOperator`
- `Replicator`
- `SystemOperator`
- `User`

Therefore, you can change that last code snippet to the following, which is better from a locale-neutral perspective, as you identify the logical role with a physical number:

```
Console.WriteLine("Is the principal in the administrator's role?  " +
    winPrincipal.IsInRole(WindowsBuiltInRole.Administrator));
```

The other overload takes a relative identifier (RID) as an `int` as shown here:

```
Console.WriteLine("Is the principal in the administrator's role?  " +
    winPrincipal.IsInRole(0x00000220));
```

1. All `BUILTIN` represents is a domain that exists on every Windows box.

This value represents the Administrator role.[2]

Now that you know how principals are defined, let's see how you can retrieve the current principal for a given thread.

Principal Policy

As you saw in one of the code snippets in the section "WindowsIdentity," it's possible to determine the current Windows user via GetCurrent() on WindowsIdentity. There's no equivalent GetCurrent() method on WindowsPrincipal, but there is the following static property called CurrentPrincipal on the Thread object to retrieve the current principal:

```
using System.Threading;
IPrincipal currentPrincipal = Thread.CurrentPrincipal;
```

However, if you ran the following code:

```
IPrincipal currentPrincipal = Thread.CurrentPrincipal;
IIdentity currentIdentity = currentPrincipal.Identity;
Console.WriteLine("Current user - is it authenticated?  " +
    currentIdentity.IsAuthenticated.ToString());
Console.WriteLine("Current user - name?  " +
    currentIdentity.Name);
Console.WriteLine("Current user - authentication type?  " +
    currentIdentity.AuthenticationType);
```

you'd get these results:

```
Current user - is it authenticated?  False
Current user - name?
Current user - authentication type?
```

The reason nothing comes up when you do this is due to the AppDomain's default principal policy choice. An enumeration called PrincipalPolicy defines these three policy schemes:

2. If you want to find out what RIDs are available, look in the Winnt.h file. The values for the WindowsBuiltInRole members match these RIDs, but there are more RIDs in the header file than what the enumeration defines.

- NoPrincipal
- UnauthenticatedPrincipal
- WindowsPrincipal

UnauthenticatedPrinicipal is the default. Therefore, if you want the identity contained in the principal to be determined via Windows, you need to call SetPrincipalPolicy() on the current AppDomain and pass it the WindowsPrincipal value, like so:

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

Now, when you extract the current principal from Thread and examine the identity values, here's what you get:

```
Current user - is it authenticated?  True
Current user - name?  WEINSTEFANER\Administrator
Current user - authentication type?  NTLM
```

I've covered the essentials of identities and principals. Now, let's put that knowledge to work by securing code based on these values via permission attributes.

Understanding Identity Permission Attributes

So far, you've seen how to use IIdentity- and IPrincipal-based objects in .NET. However, you'll also want to use them for more than simply investigative purposes, like preventing users from executing a piece of code if they don't have the right credentials. .NET has a permission for just such a purpose, PrincipalPermission, with an attribute you can use for declarative security purposes, PrincipalPermissionAttribute.

PrincipalPermission has these three properties that you can set:

- Authenticated
- Name
- Role

For example, this code requires that the caller must be in the `Manager` role:

```
[PrincipalPermission(SecurityAction.Demand, Role="Manager")]
private static void RunManagerCode()
{
    Console.WriteLine("Manager work goes here...");
}
```

However, this code requires that the only one who can run this code is an authenticated manager named `Joe` as shown here:

```
[PrincipalPermission(SecurityAction.Demand,
    Authenticated=true, Role="Manager",
    Name="Joe")]
private static void RunAuthenticatedManagerJoesCode()
{
     Console.WriteLine("Authenticated manager Joe's work goes here...");
}
```

You can also combine these permission checks if you'd like:

```
[PrincipalPermission(SecurityAction.Demand, Role="Cook")]
[PrincipalPermission(SecurityAction.Demand, Role="Manager",
    Name="Joe")]
private static void RunCookOrManagerJoesCode()
{
    Console.WriteLine("Cook or manager Joe's work goes here...");
}
```

Of course, you're not limited to declarative checks with `PrincipalPermission`; you can make imperative checks as well:

```
PrincipalPermission prinPerm =
    new PrincipalPermission(null, "Manager");
prinPerm.Demand();
```

NOTE *This permission can only work at the method or type level. It cannot be used across an entire assembly.*

CROSS-REFERENCE *Declarative and imperative checks are discussed in Chapter 4 in the section "Permission Requests."*

Now take a look at Listing 5-1 to see what happens when you use these permission checks. Given that you have declared the three preceding `RunXXXCode()` methods, set up two principals and call these methods.

Listing 5-1. Protecting Code via Role Membership

```
string[] managerRoles = {"Manager", "Cook"};
string[] cookRoles = {"Cook"};
IIdentity managerIdentity = new GenericIdentity("Jane");
IPrincipal managerPrincipal = new GenericPrincipal(managerIdentity, managerRoles);
IIdentity cookIdentity = new GenericIdentity("Joe");
IPrincipal cookPrincipal = new GenericPrincipal(cookIdentity, cookRoles);
Thread.CurrentPrincipal = cookPrincipal;
try
{ RunManagerCode(); }
catch(Exception managerEx)
{
    Console.WriteLine("Could not run the managers's code:  " +
        managerEx.Message);
}

try
{ RunCookOrManagerJoesCode(); }
catch(Exception managerEx)
{
    Console.WriteLine("Could not run cook or manager Joe's code:  " +
        managerEx.Message);
}

try
{ RunAuthenticatedManagerJoesCode(); }
catch(Exception managerEx)
```

```
{
    Console.WriteLine("Could not run authenticated manager " +
        "Joe's code:  " +
        managerEx.Message);
}
```

With the current principal set to a cook, this is the output:

```
Could not run the managers's code:  Request for principal permission failed.
Cook or manager Joe's work goes here...
Could not run authenticated manager Joe's code:
    Request for principal permission failed.
```

Now, set the principal to a manager:

```
Thread.CurrentPrincipal = managerPrincipal;
```

This change gives the new results:

```
Manager work goes here...
Cook or manager Joe's work goes here...
Could not run authenticated manager Joe's code:
    Request for principal permission failed.
```

In both cases, the last method (`RunAuthenticatedManagerJoesCode()`) could not be called. With `cookPrincipal`, the identity is not in the "Manager" role—with `managerPrincipal`, the identity's name is not "Joe". The only way `RunAuthenticatedManagerJoesCode()` can be invoked is if the identity has the name "Joe" and is in the "Manager" role.

SOURCE CODE *The IdentityTest application in the Chapter5 subdirectory contains code that looks at how* `IIdentity`*- and* `IPrincipal`*-based objects work, as well as using* `PrincipalPermission` *to restrict access to code.*

Now that you know how to secure methods via identity and role membership, let's investigate how impersonation works.

Exploring Impersonation

As you may know, *impersonation* means taking on the identity of someone else to execute code with his or her credentials. Sometimes, this is a desired feature; other times, malicious code will try to act like another user with elevated permissions to get access to resources.

As you saw in the last section, you can use `PrincipalPermissionAttribute` to limit access to code. However, it's pretty easy to break through this barrier. For example, let's say there are two methods on a type called `SpecificRoleMembershipOnly()` and `SpecificWindowsUserOnly()` as shown here:

```
public class Privileged
{
    [PrincipalPermission(SecurityAction.Demand, Role="BigBucksClub")]
    public void SpecificRoleMembershipOnly()
    {
        Console.WriteLine("A privileged role member has arrived...");
    }

    [PrincipalPermission(SecurityAction.Demand,
        Name=@"WEINSTEFANER\Administrator")]
    public void SpecificWindowsUserOnly()
    {
        Console.WriteLine("A specific Windows user has arrived...");
    }
}
```

Now, if you attempted to call these methods from a .NET application without doing anything with principals and identities on the current thread, you wouldn't be successful, as `CurrentPrincipal` wouldn't contain any principal information. Of course, if you were the administrator on the `WEINSTEFANER` domain, you could alter the call to set the thread's current principal as follows:

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

But as the following code snippet demonstrates, you don't need to be an administrator to call `SpecificWindowsUserOnly()`:

```
IIdentity fakeWindowsIdentity = new
    GenericIdentity(@"WEINSTEFANER\Administrator");
IPrincipal fakeWindowsPrincipal = new
    GenericPrincipal(fakeWindowsIdentity, null);
Thread.CurrentPrincipal = fakeWindowsPrincipal;
p.SpecificWindowsUserOnly();
```

Don't be fooled by the format of the name! In this case, all `PrincipalPermission` enforces is that the name matches the current thread's identity name; it doesn't care how it got there.

You can run into the same case with the role-secured method as demonstrated here:

```
String[] fakeRole = {"BigBucksClub"};
IIdentity fakeRoleIdentity =
    new GenericIdentity(@"BigBucksWannabe");
IPrincipal fakeRolePrincipal =
    new GenericPrincipal(fakeRoleIdentity, fakeRole);
Thread.CurrentPrincipal = fakeRolePrincipal;
objRef.SpecificRoleMembershipOnly();
```

In all cases, the problem is that you can change the thread's current principal. To prevent this kind of mischievous activity, you could use the `SecurityPermission` attribute and deny yourself the ability to change the principal, like so:

```
[SecurityPermission(SecurityAction.Deny,
    Flags=SecurityPermissionFlag.ControlPrincipal)]
```

At first glance, this may seem kind of odd. After all, you're trying to circumvent all this role-access security junk anyway, so why would you rid yourself of the workaround?

The key is to refuse assemblies the ability to change the principal, which can be done by not granting the permission to assemblies via the .NET Configuration tool. Specifically, if you edit the permission settings for `SecurityPermission`, you need to leave the Allow principal control option unchecked as Figure 5-1 shows.

Figure 5-1. Denying principal control

In this case, I'm assuming that the permission is part of a permission set that's used in a code group and that the assembly matched the membership condition of that code group.

You can also prevent the principal from being changed when you create an `AppDomain` (an object that provides code isolation between unrelated applications):

```
AppDomain newAD = AppDomain.CreateDomain("ApressDomain");
newAD.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
SecurityPermission sp =
    new SecurityPermission(
    SecurityPermissionFlag.ControlPrincipal);
sp.Deny();
int ret =
    newAD.ExecuteAssembly(@"D:\assembly\PPTester.exe");
```

In this case, you create the AppDomain, and call SetPrincipalPolicy() to make sure the principal is based on the current user. You deny any downstream code the capability to change the principal, and then you run an EXE.

Caveats with Limiting Principal Modification

Now, it may seem like you've done your job to prevent any code in the PPTester assembly from changing the principal. However, you've actually been too aggressive. So, say the following code is the start of its entry point method:

```
static void Main(string[] args)
{
    Console.WriteLine("Main entered.");
    Console.WriteLine("Identity name:  " +
        Thread.CurrentPrincipal.Identity.Name);
    //  More code goes here...
}
```

If you run this code, you may be a bit surprised by the results in the console window:

```
Main entered.
System.Security.SecurityException: Request failed.
```

You're probably thinking, "Hey! All I'm doing is trying to see what the current identity's name is. Why am I getting the exception?" The reason is a little subtle. If you look at the documentation for SetPrincipalPolicy() in the SDK, here's what it says:

> *Specifies how principal and identity objects should be attached to a thread* ***if*** *the thread attempts to bind to a principal while executing in this application domain. [Emphasis mine.]*

This means if the principal is never needed, it's never created. Once it's needed, it's cached for future reference. So what you did is state that you wanted to use the current Windows user for the principal's identity. However, no entity tried to use the principal in any way before Deny() was called. Therefore, simply looking at the Name property internally sets the thread's current principal, which you can't do.

You might try to solve this by using the thread's principal object in some way before `Deny()` is called as shown here:

```
newAD.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
bool isAuth =
    Thread.CurrentPrincipal.Identity.IsAuthenticated;
SecurityPermission sp =
    new SecurityPermission(
    SecurityPermissionFlag.ControlPrincipal);
sp.Deny();
```

However, that doesn't work as expected either:

```
Main entered.
Identity name:
```

Again, the problem isn't obvious. You've set the new `AppDomain`'s principal policy to `WindowsPrincipal`, but the current `AppDomain`'s policy has never been touched. Therefore, when you call `CurrentPrincipal` on `Thread`, you're going to get the principal for the current `AppDomain`—in this case, the no-name user you saw earlier in this chapter, at the end of "Understanding Identity Permission Attributes."

You could set the current `AppDomain`'s principal as well, but I think the easiest way to set this up is by specifying the new `AppDomain`'s principal explicitly. Here's an example:

```
newAD.SetThreadPrincipal(
    new WindowsPrincipal(WindowsIdentity.GetCurrent()));
```

Now everything works as planned. Any code in the new `AppDomain` that tries to set the principal to impersonate another principal will get a `SecurityException`.

SOURCE CODE *The PrincipalPermTest folder in the Chapter5 subdirectory contains two projects: PPTester and PPLauncher. PPTester contains the* `Privileged` *type along with code that you can use to try and set the current principal. PPLauncher will load PPTester—you can change the code here to see what happens when you prevent PPTester from changing the principal. Remember to set the target directory correctly when* `ExecuteAssembly()` *is called!*

I've covered the essentials of impersonation. Next, I'll show you how Windows-specific impersonation works.

Windows Impersonation

Now it's time to talk about the `WindowsIdentity` constructor that takes an `IntPtr` as its only argument. If you've done security programming in Windows, you may have run across the `LogonUser()`[3] API call. The function in the SDK looks like this:

```
BOOL LogonUser(
  LPTSTR lpszUsername,    // user name
  LPTSTR lpszDomain,      // domain or server
  LPTSTR lpszPassword,    // password
  DWORD dwLogonType,      // type of logon operation
  DWORD dwLogonProvider,  // logon provider
  PHANDLE phToken         // receive tokens handle
);
```

To use this call requires an unmanaged method invocation (or P/Invoke) as shown here:

```
[DllImport("advapi32.dll")]
static extern int LogonUser(String UserName, String Domain,
    String Password, int LogonType,
    int LogonProvider, ref IntPtr WindowToken);
```

The first three parameters are pretty easy to figure out how to use. The `UserName` is set to the user you want to impersonate (for example, "BobK"). `Domain` specifies the name of the server or domain that should contain the given user information. (Note that you can set `Domain` to "." if you want to authenticate against the local database, or "null" if the user name will be in user principal name, or UPN, format—for example, "BobK@SomeDomain".) And `Password` is the associated password that only the user (or trusted identities that want to impersonate the user) should know.

Now take a look at the last three parameters in detail. `LogonType` specifies the kind of logon you want to perform. The SDK lists seven different types, but for these purposes, use `LOGON32_LOGON_NETWORK`.[4] `LogonProvider` allows you to specify the provider you wish to use to authenticate against. For these tests, you'll use

3. There's also `LogonUserEx()`, but I'm going with `LogonUser()` here because it's a bit easier to use.

4. You can find the values of the constants `LogonUser()` uses in WinBase.h.

the default provider, which is `LOGON32_PROVIDER_DEFAULT`. Finally, if the logon is successful, `WindowToken` will contain a valid handle to a user token. Therefore, you need to pass in an `IntPtr` by reference to get this value when the method is done.

If the method is successful, you'll get zero as the return value. Any other value denotes some kind of error. You can find out what the error code is by calling `GetLastError()` as shown here:

```
[DllImport("kernel32.dll")]
static extern int GetLastError();
```

One error code you may immediately run into when you try to call `LogonUser()` is `ERROR_LOGON_TYPE_NOT_GRANTED`.[5] To use `LogonUser()` requires that the account that the process is running under has the `SE_TCB_NAME` privilege. You can find out if your account has this by running the Local Security Settings tool, which is found under Administrative Tools. Figure 5-2 shows what the interface for this tool looks like when you run it.

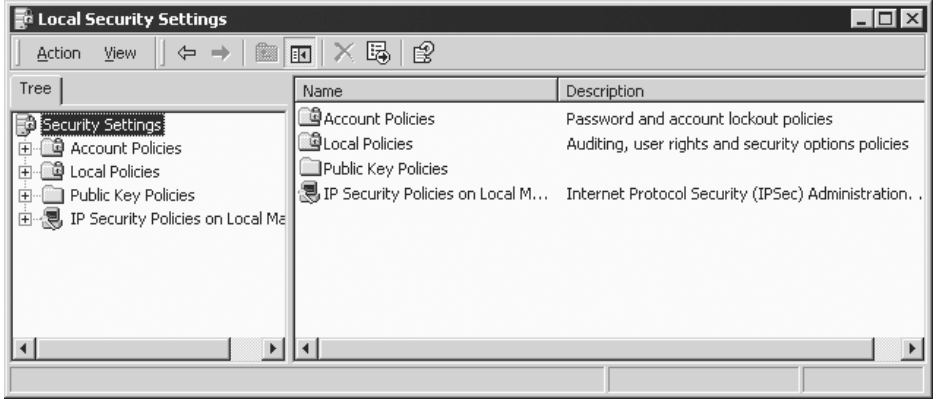

Figure 5-2. Local Security Settings tool

Now open Local Policies ➢ User Rights Assignment, and double-click the list view item Act as part of the operating system. Figure 5-3 shows what the resulting dialog box looks like.

5. To find out all of the return values for `LogonUser()`, search for the phrase "ERROR_LOGON" in WinError.h.

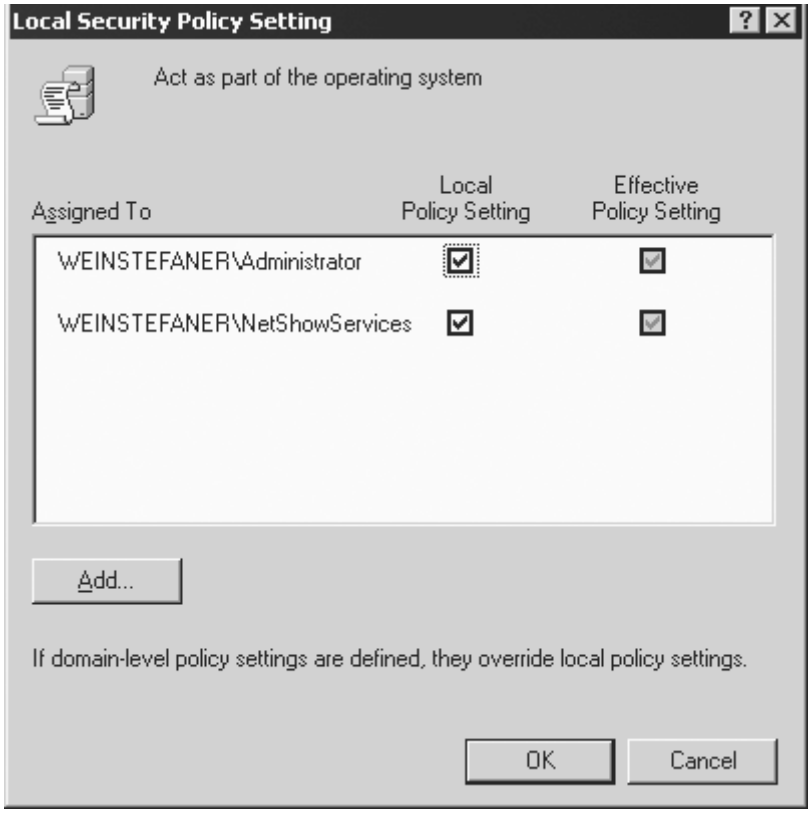

Figure 5-3. Privilege dialog box

To add a specific user, click the Add button at the lower-left corner of the screen, and select the user you want. When the user is added, the Effective Policy Setting option won't be checked, so if that user is currently logged on, he or she must log off to receive the `SE_TCB_NAME` privilege.

CAUTION *There are a lot of reasons why you do not want to grant this privilege to just any user. Essentially, this privilege gives the account access to anything in the operating system. Even the Administrator account does not receive this privilege by default. The only reason you have to do so in this case is for you to be able to call* `LogonUser()`.

Next, assume that you added this privilege to the account you run under. Now, try to get a user token in a .NET application. To illustrate how you might approach this, I created two accounts on my machine, JaneDoe and JohnSmith. JaneDoe is part of the Administrators and Users group, whereas JohnSmith is only in the Users group.[6] I then added the code you see in Listing 5-2 to a console application.

Listing 5-2. Using LogonUser() in .NET

```
class WIT
{
    private static String t_Domain = null;
    private static String t_Password = null;
    private static String t_UserName = null;
    private const int LOGON32_LOGON_NETWORK = 3;
    private const int LOGON32_PROVIDER_DEFAULT = 0;
    private const int FORMAT_MESSAGE_FROM_SYSTEM = 0x00001000;
    private const int FORMAT_MESSAGE_ALLOCATE_BUFFER = 0x00000100;

    [DllImport("advapi32.dll")]
    static extern int LogonUser(String UserName, String Domain,
        String Password, int LogonType, int LogonProvider,
        ref IntPtr WindowToken);

    [DllImport("kernel32.dll", CharSet=CharSet.Auto)]
    static extern int FormatMessage(int Flags,
        ref IntPtr MessageSource, int MessageID,
        int LanguageID, ref IntPtr Buffer,
        int BufferSize, int Arguments);

    [DllImport("kernel32.dll")]
    static extern bool CloseHandle(IntPtr Handle);

    [DllImport("kernel32.dll")]
    static extern int GetLastError();

    [STAThread]
    static void Main(string[] args)
    {
```

6. You can set these accounts on your local machine by traversing to the Local Users and Groups node in the Computer Management tool. I'll leave that as an exercise for you—it's pretty easy to do.

```
    try
    {
        Console.WriteLine("Main entered...");
        GetLogonInformation();

        Console.WriteLine("Attempting logon...");
        IntPtr windowsToken  = IntPtr.Zero;
        int logonRetVal = LogonUser(t_UserName, t_Domain, t_Password,
            LOGON32_LOGON_NETWORK, LOGON32_PROVIDER_DEFAULT,
            ref windowsToken);

        if(0 != logonRetVal)
        {
            Console.WriteLine("Logon successful.");
            WindowsIdentity newWI = new WindowsIdentity(windowsToken);
            Console.WriteLine("Logon name: " + newWI.Name);
            CloseHandle(windowsToken);
        }
        else
        {
            throw new Exception(
                "Error occurred with LogonUser().  " +
                "Error number:  " + GetLastError() + ", " +
                "Error message:  " + CreateLogonUserError(GetLastError()));
        }
    }
    catch(Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
}

private static void GetLogonInformation()
{
    Console.WriteLine("Please enter the user name.");
    t_UserName = Console.ReadLine();
    Console.WriteLine("Please enter the password.");
    t_Password = Console.ReadLine();
    Console.WriteLine("Please enter the domain.");
    t_Domain = Console.ReadLine();
}
```

```
    private static String CreateLogonUserError(int ErrorCode)
    {
        IntPtr bufferPtr = IntPtr.Zero;
        IntPtr messageSource = IntPtr.Zero;

        int retVal = FormatMessage(
            FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_ALLOCATE_BUFFER,
            ref messageSource, ErrorCode, 0, ref bufferPtr, 1, 0);

            string strRetval =
                Marshal.PtrToStringAuto(bufferPtr, retVal);

        return strRetval;
    }
}
```

There's a fair amount of code here, so I'll go through each section in detail.

First, you get the username/password/domain combination from the user via `GetLoginInformation()`. Once that's done, you call `LogonUser()`. Note that you initialize the token value to `IntPtr.Zero` before you make this call. If the call was successful, you create a new `WindowsIdentity` object with the token value. You also have to call `CloseHandle()` on the token when you're done with it.

If the call didn't work, you throw an exception. The string that is passed to the new `Exception` object is built with the error code returned from `GetLastError()`, and an error message that can be obtained with `FormatMessage()`. `FormatMessage()` can be a tricky API to call. However, all you need to do is get a message from the system, so the call declaration can be simplified to what I have in the code snippet. Note that by setting `Flags` to contain `FORMAT_MESSAGE_ALLOCATE_BUFFER`, you get a pointer to the string. This is why you're calling `PtrToStringAuto()` to get the string's contents. Figure 5-4 shows what happens when the credentials are correct, and Figure 5-5 displays the results of a bad authentication.

Figure 5-4. Successful LogonUser() call

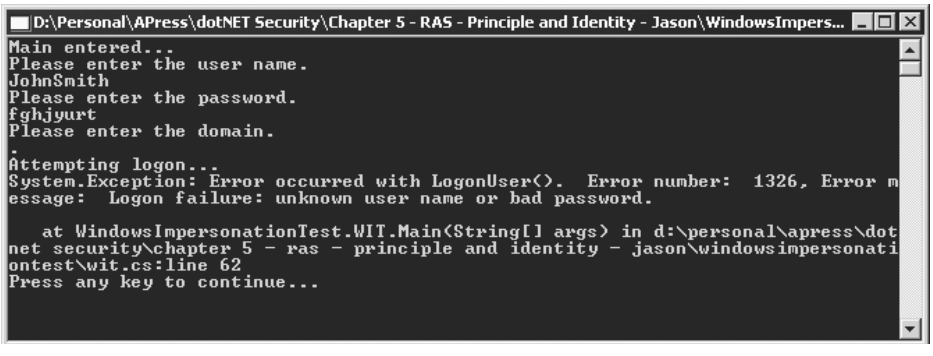

Figure 5-5. Unsuccessful LogonUser() call

Okay, now that you have all of this baseline code in place, you can finally perform Windows impersonation as shown in Listing 5-3. This is actually the easy part.

Listing 5-3. Windows Impersonation in .NET

```
WindowsImpersonationContext wic = newWI.Impersonate();
WindowsIdentity currentWI = WindowsIdentity.GetCurrent();
Console.WriteLine("Current Windows name after impersonation:  " +
    currentWI.Name);
wic.Undo();
currentWI = WindowsIdentity.GetCurrent();
Console.WriteLine("Current Windows name after Undo():  " +
    currentWI.Name);
```

You call `Impersonate()` on a `WindowsIdentity` object, and store the return value in a `WindowsImpersonationContext` object. This is basically a caching mechanism to store the current user's information, as a call to `Undo()` will revert the user to the current Windows user. Figure 5-6 shows what the console application does when you add this code.

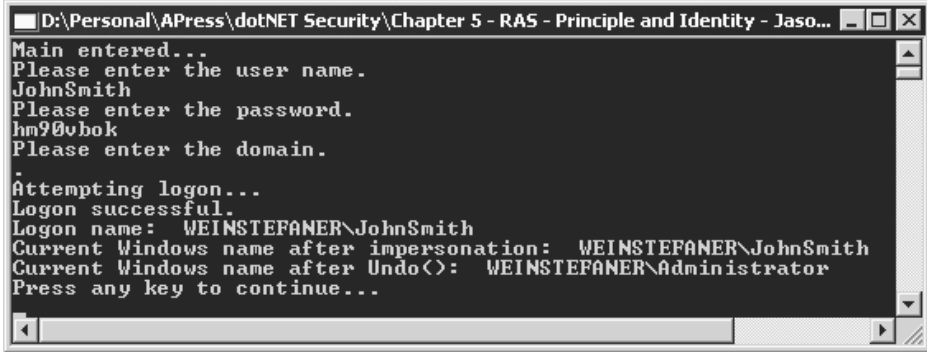

Figure 5-6. Impersonation results

Windows impersonation is still subject to being able to change the thread's user token. If this has been denied, you won't be able to do Windows impersonation. However, if you are able to do Windows impersonation, you need to grab the current Windows user from `WindowsIdentity` and *not* from the `Thread` class (via `CurrentThread.CurrentPrincipal`). Windows impersonation will not automatically set the current principal, as you can see here:

```
IIdentity curIdent = Thread.CurrentPrincipal.Identity;
Console.WriteLine("Current identity from the thread " +
    "before impersonation:  " +
    curIdent.Name);
WindowsImpersonationContext wic = newWI.Impersonate();
WindowsIdentity currentWI = WindowsIdentity.GetCurrent();
Console.WriteLine("Current Windows name after impersonation:  " +
    currentWI.Name);
curIdent = Thread.CurrentPrincipal.Identity;
Console.WriteLine("Current identity from the thread " +
    "after impersonation:  " +
    curIdent.Name);
```

If you run this code, you'll see that the name of the identity from `Thread` is the same before and after impersonation.

SOURCE CODE *The WindowsImpersonationTest application in the Chapter5 subdirectory contains all of the code that I went through in this section.*

Summary

In this chapter, I covered the following topics:

- Basics of role-based security in .NET
- Interfaces and the concrete types that .NET provides, and how you can use them to discover principal information
- Impersonation of an identity and how you can prevent this in .NET

In the next chapter, I'll start to move beyond the core .NET types and delve onto some real-world scenarios where security is essential to the problem at hand. Specifically, that problem is communicating with distributed objects, or what is known as remoting in .NET. That's what Chapter 6 is all about.