CHAPTER 5

Policies and Code Groups in Detail

- Using the Code Access Security Policy Tool
- Working with Code Groups
- Creating Custom Permissions
- Using Policy Objects
Policies are the mechanism that the Common Language Runtime (CLR) uses to examine evidence presented by an object and create a permission that the object uses to access the requested resource. Code groups are the mechanism that grants code a level of trust based on a number of criteria, such as location. All Code must have membership in at least one code group to execute. Chapter 4 started the discussion of policies and code groups. However, that discussion only covered the basics.

This chapter presents detailed information about both policies and code groups. You’ll discover additional methods for creating, editing, and deleting policies and code groups from any system. In addition, the chapter covers more techniques for working with code groups and policies in code—including programmatic techniques for managing these elements. Finally, you’ll learn how to create a default security policy for your system (an individual workstation or server). This is an important step because the .NET Framework only installs a default policy that may not meet your specific needs.

Using the Code Access Security Policy Tool

The Code Access Security Policy (CASPol) tool is a command line utility you can use to add and remove policies from a system, as well as make other adjustments. Normally, you want to use the .NET Framework Configuration tool demonstrated in the “Using the .NET Framework Configuration Tool” section of Chapter 4 to make changes, but this tool can come in quite handy for a number of tasks. Generally, you won’t use it directly at the command prompt, but will call on the CASPol tool from a batch file or a program to perform specific types of tasks automatically.

CASPol provides an extensive number of command line switches. There are so many, in fact, that you’ll quickly find yourself getting lost in switches if you try to use them all in your first program. You can find a complete list of these switches at http://msdn.microsoft.com/library/en-us/cptools/html/cpgrfcodeaccesssecuritypolicyutilitycaspolexe.asp. This section demonstrates some of the more common switches. Here’s the command line syntax for CASPol:

```command
CASPol [Option] [Arguments]
```

When you type CASPol by itself or with the ? switch, you receive a list of options. The list is relatively long, so knowing what you want to do is always best. An option tells CASPol what type of information you want to see or defines how you want CASPol to perform. For example, the -machine option tells CASPol to make all commands act at just the machine
level, while the `–addgroup` adds a new group. Only use one of each option type in any command. Typing `CASPol –machine –addgroup` would add a group to the machine level. You use the arguments to provide additional information. In general, use only the arguments that you actually need to perform a task.

CASPol is a task-oriented tool. You use it in different ways depending on the task you need to perform. Microsoft provides an overview of a large number of CASPol tasks at http://msdn.microsoft.com/library/en-us/cpguide/html/cpconusingcodeaccesssecurity-policytoolcaspolexe.asp. You can also find a list of CASPol command line options at http://msdn.microsoft.com/library/en-us/cptools/html/cpgrfcodeaccesssecuritypolicy-utilitycaspolexe.asp. This chapter only reviews a limited number of common tasks, but covers those tasks in a relatively detailed manner. CASPol is a powerful tool that you can use to automate a variety of tasks.

### Listing the Permissions and Code Groups

Even a default .NET Framework setup includes a number of policies and code groups, so listing them at the command prompt is often useful. For that reason, it’s usually better to list them to a file using the `CASPol –list` command, where `Filename` is the name of the file you want to use. (You can find a sample listing in the `Chapter 05\FullListing.TXT` file of the Sybex Web site.) The default setup includes all policies and code groups for all levels. CASPol formats the file so that you can potentially parse it using an external program.

You can reduce the amount of information by specifying a particular level. For example, typing `CASPol –user –list` displays just the policies and code groups at the user level. Of course, now you can’t compare differences between levels, but you do gain access to all of the information at one level.

Another way to list policies and code groups is by type. For example, you can type `CASPol –enterprise –listgroups` to list just the enterprise groups. Typing `CASPol –machine –listdescription` displays a description of the various groups, as shown in Figure 5.1. You also have options for listing all of the permissions (the `–listpset` switch) and all of the assemblies that have full trust (the `–listfulltrust` switch). The combination of all these switches provides relatively full access to all of the security information for the .NET Framework. Unfortunately, you have to issue the commands one at a time—you can’t combine them to create custom listings in a single pass.
Many of these switches also come in a shortened version that you can use to reduce your typing. For example, you could type the command in this example as `CASPo1 -m -ld`. The chapter uses the longer form to make the meaning of the command line entries clearer.

The `CASPo1 -listgroups` command is particularly handy when you want to see the hierarchical structure of the zones. Notice that there are actually subzones in some cases. For example, the `My_Computer_Zone` entry includes `Microsoft_Strong_Name` and `ECMA_Strong_Name` entries.

Of course, the question remains as to why you’d want to use this approach to list the various security elements when you can use the .NET Framework Configuration Tool. As you work with the various outputs, you’ll notice that they’re structured, which means you can process the output information in a variety of ways. The simple text makes it easy to manipulate the entries for storage in a permanent form in either a database or a written report.

Using this utility is also the only way to obtain information when you use a remote connection that doesn’t have the bandwidth required to use the .NET Framework Configuration Tool. A text interface is faster and uses fewer resources. These reasons keep many developers using text-based utilities, even when GUI alternatives exist.
Making Group Modifications

The .NET Framework security features work like a hierarchical database in some respects. Consequently, you can add and delete entries as needed to create new policies and setups. You can also modify existing policies. For example, you can add a new permission set to an existing group. This is one area where CASPol really shines. Imagine having to set up a new server using the .NET Framework Configuration Tool. Yes, you can eventually do it, but it’s going to take considerable time (assuming you can keep the network administrator awake). A simple batch file lets CASPol perform the required configuration for you using a single command.

The reason that I mentioned the hierarchical database approach is that you can visualize how a change will affect the system better when you use this perspective. Look again at the hierarchy in Figure 5.1. You can add a new code group at any point along that hierarchy. For example, if you wanted to add a super secret strong naming convention to the Microsoft_Strong_Name entry, you could type the following command: `CASPol -addgroup 1.1.1. -zone MyComputer Full-Trust -name MyStrongGroup -description "This is a special group."`. CASPol will display the security policy message shown in Figure 5.2. Type `yes` (lowercase and the full word) and CASPol will display the success message shown in Figure 5.2.

**FIGURE 5.2:**
When adding a new group, CASPol asks whether you want to change the security policy.

---

**TIP**
You can turn the policy change message shown in Figure 5.2 off using the `CASPol -polchgprompt off` command. This feature works well for batch processing because you don’t want the processing to stop to wait for input. However, it’s dangerous to turn policy change messages off when working at the command line because you won’t receive any warnings. It’s always better to receive the policy change message so that you can validate any changes before you make them. The best method is to turn the policy change messages off at the beginning of a batch file and turn them back on using the `CASPol -polchgprompt on` command at the end of the batch file.
When you execute the `–addgroup` command, CASPol adds a new group to the Microsoft_Strong_Name entry, as shown in Figure 5.3. (Figure 5.3 uses the .NET Framework Configuration Tool to make the addition clear, but you could also use the listing commands in the “Listing the Permissions and Code Groups” section.) The `–addgroup` command has several components. First, notice that the `–addgroup` entry specifies the location in the hierarchy as 1.1.1. You could also use a group name. Second, this new entry belongs to the MyComputer zone and it has the FullTrust permission. Figure 5.3 shows both of these elements. Third, the new group has a name of MyStrongName and a description of “This is a special group”. The examples in the Microsoft help file make it look like you can create a name with spaces, but this feature doesn’t appear to work without error. However, you can create a description with spaces by enclosing the description in double quotes.

Removing a group is actually easier than adding one. All you need is either the hierarchical level or the code group name. Type `CASPol -remgroup MyStrongGroup` to remove the MyStrongGroup added earlier. CASPol will ask you whether you’re sure about making the change as it did before. After you type `yes`, CASPol will remove the group and display the success message shown in Figure 5.4.

---

**FIGURE 5.3:**
Defining a new group adds the entry to the specified location in the group hierarchy.
WARNING

The `remgroup` command is a one-way process. Removing a group is permanent and there’s no undo command. Make sure you actually want to remove a group before you execute this command.

Making Permission Modifications

Working with permissions is similar to working with groups. To add a permission, you use the `addpset` command. For example, you might want to add a new permission set to the machine level by typing: `CASPol -machine -addpset MyPermission.XML`. Figure 5.5 shows the results of this example command using the .NET Framework Configuration Tool.

FIGURE 5.5:
Adding a permission lets you create custom setups for a code group.
Notice that the -addpset command requires an XML file that includes the permission information. You can create a custom permission or you can use a combination of named permissions. The example uses a combination of named permissions, as shown in Listing 5.1. (This listing isn’t complete. You can find the complete XML file in the \Chapter 05\ MyPermission.XML file of the source code located on the Sybex Web site.)

**Listing 5.1 Permission XML File Example**

```
<PermissionSet class="System.Security.NamedPermissionSet" version="1"
    Name="SpecialPermission"
    Description="Allows access to some resources.">
    <IPermission
        class="System.Security.Permissions.EnvironmentPermission,
        mscorlib,
        Version=1.0.5000.0,
        Culture=neutral,
        PublicKeyToken=b77a5c561934e089"
        version="1"
        Unrestricted="true"/>
    ... Other IPermission Entries ...
    <IPermission class="System.Diagnostics.PerformanceCounterPermission,
        System,
        Version=1.0.5000.0,
        Culture=neutral,
        PublicKeyToken=b77a5c561934e089"
        version="1"
        Unrestricted="true"/>
</PermissionSet>
```

**TIP**

When you want to work with named permissions, start with one of the existing permission sets such as Everything. Modify the permissions in the XML file to meet your specific needs. You can obtain a file with the existing named permissions using the CASPol -listpset > Filename command, where Filename is the name of the file you want to use.

Creating a permission set always entails the same features. You define the permission set itself using the `<PermissionSet>` tag, and then add any number of `<IPermission>` tags to define the permissions in the permission set.

The `<PermissionSet>` tag always includes a reference to the `System.Security.NamedPermissionSet` class as shown. The current version of the .NET Framework and all previous versions use a version number of 1 for this class, so you set the Version argument to 1. You must also
provide a Name argument that includes the name of the permission set. The Description argument is optional, but you should consider including it.

**TIP** You can undo most current policy change by using the CASPol –recover command. Make sure you specify a level when you want to undo just the change at a specific level. Otherwise, CASPol will undo all of your changes and you’ll need to start again. Note that CASPol doesn’t have an undo cache, so you can only undo the last change that you made.

Each of the `<IPermission>` tags includes the name of the class, the name of the file that contains the class, a version number, the culture, and the public token key for that permission class. All of these entries identify the class. You must also provide a version number for the `<IPermission>` tag and the level of access the permission provides. When you include `Unrestricted="true"`, the permission grants all of the access the class provides. All of the permission classes provide at least one property that accepts access levels. Look at the `SecurityPermission` entry and you’ll notice that it uses the `Flags` property to control the number of permissions this entry provides. Figure 5.6 shows the effect of using these flag values. Likewise, the `FileDialogPermission` entry uses the `Access` property. In this case, the permissions are the same as providing unrestricted access, so CASPol actually sets this entry to `Unrestricted="true"`.

**Figure 5.6:**
Setting access permissions using a property changes the way CASPol makes the policy change.
Defining the `<Permission>` tags means creating pointers to code that contains the permission handling functionality. The Everything permission contains all of the permissions that the .NET Framework supports, so you can use it as a template for your own permission set. Giving access to everything is usually too much; however, some of the supplied permissions are too restrictive. The reason you want to create custom named permissions is to ensure that the code group has enough permission to perform a given task, but not so much permission that it becomes a security risk. For example, the code in Listing 5.1 lacks the `IsolatedStorageFilePermission` because there's little reason for most code to access isolated file storage. The example also lacks the `DirectoryServicesPermission` and other permissions that you would want to grant only as needed.

Removing a permission set requires use of the `-rempset` command. For example, to remove the previous example, you'd type `CASPol -rempset SpecialPermission`. As with most policy changes you make, the system will ask if you really want to remove the permission.

**Adding an Assembly**

Adding an assembly to the security setup lets you use it for permissions, which in turn affects code groups. Microsoft assumes that you want to place this custom permission in a class. Consequently, you need to add the resulting assembly to the Global Assembly Cache (GAC), which means giving it a strong name.

Experimentation shows that you can place any assembly with a strong name in the policy assembly list even if it doesn't appear in the GAC. This means you could create a very specific permission for an application that would reside in the application assembly. The security is self-contained within the application it supports.

As an experiment, you can use the application found in the Signed Client example for Chapter 4. (You can find the full listing for this example in the `\Chapter 04\C#\SignedClient or \Chapter 04\VB\SignedClient` folder of the source code located on the Sybex Web site.) Type `CASPol -machine -addfulltrust SignedClient.EXE` to add the assembly to the list. Figure 5.7 shows the results of this command.

![Figure 5.7](image)
Obviously, this assembly doesn’t really do anything to demonstrate permissions, but it does show how to use the command. The “Creating a Permission Assembly” section of the chapter shows how to create a permission class, design a permission around that class, and then use the custom permission as the basis for a code group. The resulting applications won’t work without the custom permission object in place on the machine, which means that the application is useless without the proper support.

Interestingly enough, the help file doesn’t say how to remove an assembly from the Policy Assemblies list. Type `CASPol -machine -remfulltrust SignedClient.EXE` to remove the assembly from the list. This command will fail if the assembly is currently in use, but will succeed otherwise. The command must be able to find the assembly for removal, just as it does when you add it.

**Resolving Security Errors in Assemblies**

You can use CASPol to locate security problems in an assembly. An assembly may not run if it lacks permissions to execute or to perform specific tasks. For example, let’s say you want to check the Declarative example in Chapter 4 for errors. (You can find the full listing for this example in the `\Chapter 04\C#\Declarative` or `\Chapter 04\VB\Declarative` folder of the source code located on the Sybex Web site.) Type `CASPol -machine -resolvegroup Declarative.EXE` to determine the groups that this assembly belongs to at the machine level. Figure 5.8 shows typical output.

![Figure 5.8: Check for security errors in an assembly by using one of the resolve commands.](image)

The example belongs to the All_Code group, but that group doesn’t provide any access. However, the MyComputer zone that appears below the All_Code group does grant full access to system resources.
Working with permissions is similar to checking groups. For this example, you’d type `CASPol -machine -resolveperm Declarative.EXE` to determine the permissions of the Declarative example. In this case, the example has full access through the `System.Security_PERMISSIONSET` class. The example also has permissions from both the `System.Security.PERMISSIONS.Url-IdentityPermission` and `System.Security.PERMISSIONS.ZoneIdentityPermission` classes.

**Using the .NET Wizards**

You have another choice in addition to using CASPol or the .NET Framework Configuration Tool for modifying security. Using the .NET Wizards is an alternative when the security changes you need to make are extremely simple and straightforward. Figure 5.9 shows these three tools.

![Figure 5.9:](image)

As you can see, the three wizards help you perform simple security adjustments, trust an assembly, or fix an application setup. All you need to do to use these three wizards is double-click the appropriate icon and follow the prompts. It’s important to remember that these wizards can help in some situations, so you shouldn’t forget that they exist. However, most security needs are more complex than these wizards can handle.

**WARNING** Unfortunately, it’s not a good idea to use the wizards as a starting point and then make modifications using some other tool. Using the same tool from start to finish for a given task is less error prone because you don’t have to remember where you stopped making security changes when switching from one tool to another.

**Using Code Groups**

An essential part of maintaining a secure environment is defining code groups that actually provide some level of protection while allowing membership of code that does require specific resources. The permissions that you give the code group define how much security the code group provides as contrasted to the access it grants. These two concepts contrast with each other. Granting a code group greater access means reducing the amount of security that it provides.
The following sections describe code groups in detail. First, the benefits and limitations of the default code group are discussed. Once you understand the default code groups better, this section will demonstrate how to create custom groups of your own.

**Understanding the Default Groups**

The default .NET Framework setup doesn’t assume anything about the individual user or the enterprise—it focuses on the requirements of an individual machine. Consequently, the Code Groups folder for the User and Enterprise levels only contain the All_Code group. The Machine level contains all of the default groups that the .NET Framework supplies, as shown in Figure 5.10.

![Figure 5.10](image)

As shown in Figure 5.10, the default groups start with zone level security. You can see that each group defines a different zone, such as the Internet zone. Below each of these code groups are groups that define levels of access. At the My_Computer_Zone level, the code gains full access using either the Microsoft or European Computer Manufacturer’s Association (ECMA) strong naming conventions.

Notice that some of the entries in this list (Figure 5.10) have white diamonds in place of the more common purple diamonds. You can’t edit these custom code groups directly. In most cases, these custom code groups provide specific features that the originator doesn’t want changed in any way. Viewing the custom code group properties does show the XML used to create it, however, which is an important way to learn more about how to construct
custom code groups yourself. Figure 5.11 shows a typical custom code group representation. (Display this dialog box by clicking the View Custom Code Group Properties link and selecting the Custom Code Group tab.)

**Figure 5.11:**
Define a custom code group for specific situations where you don't want any changes.

Custom code groups are one situation where it becomes important to import the information from an XML file, rather than to define the group directly. Figure 5.11 shows that defining a custom code group is similar to defining a custom permission set as we did in Listing 5.1.

**TIP**
You can add custom code groups using the -custom option of the CASPol utility. For example, you could type `CASPol -addgroup 1.1.1 -custom MyMembershipCond.XML FullTrust -name MyStrongGroup -description 'This is a special group.'` to add a new group based on the MyMembershipCond.XML file content. Note that no matter how you add an XML-based membership condition, it’s always going to appear as a custom code group.

**Working with Code Groups**
Code groups can help you perform real work with applications by ensuring that a particular application actually has the required access. However, it's not just about ensuring correct security. A code group can reduce your coding burden and add consistency to the verification process.

You’ve already seen pieces of this concept. The “Making Group Modifications” section of this chapter emphasizes the hierarchical structure of the security setup. The “Writing the StrongNameIdentityPermission Class Code” section of Chapter 4 discusses how to use strong names in your code. Finally, the “Defining Membership and Evidence” section

Copyright © 2003 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved.
of Chapter 4 shows how to verify membership in a particular code group. The example in this section combines all of these features and shows how to access them using a code group that you've defined.

**WARNING**

If you suddenly notice that all code access security requests succeed, even from locations such as the Internet, someone may have turned code access security off. To turn code access security back on, type `CASPol --security on`. Obviously, if this solution fixes the problem, you need to investigate further. A cracker may have gained valuable access to your system.

### Defining the Special Code Group

A special code group lets you check for a particular membership. However, before you can create a special code group, you need a component that will check for this membership as part of allowing access to the features it provides. Listing 5.2 shows one way to approach the problem. This is the technique that many developers will use because it's easy and fast. (You can find the full listing for this example in the \Chapter 05\C#\CodeGroupComponent or \Chapter 05\VB\CodeGroupComponent folder of the source code located on the Sybex Web site.)

#### Listing 5.2 Specialized Code Group Component

```csharp
// Use attributes to define the security requirements.
[ZoneIdentityPermission(SecurityAction.LinkDemand,
    Zone=SecurityZone.MyComputer),
StrongNameIdentityPermissionAttribute(SecurityAction.LinkDemand,
    PublicKey='00240000048000009400000006020000002400000525341310004000001' +
    '000100530C096892537C141556FB86EC8EA1DFCE8E2135F8A4F261E820A' +
    '045CC9C36F2CAC150D767700D2A042045586F15DED4882989C446' +
    'B36C0EFEFEDC44E7E4810007C057A79E4ABE730DFAE26A3FE32CF3' +
    '705289BF736CF12F559E8804B0E1B780944FACE97182BA84674DA3BE3' +
    '\IC553DFAA1467C13F8810AD506DF')]

class SayHello2
{
    public String DoSayHello2() {
        return "Hello from DoSayHello2!";
    }
}
```

Each of these attributes checks a specific security attribute. To use this component, the caller must appear on the local computer and be signed with the correct key. You can place these two attributes in various locations. Placing them at the class level secures all of the
methods, events, and properties in the class as well. Securing individual elements means placing the two attributes immediately before those elements.

**Considering a Code Group Component Alternative**

The .NET Framework normally provides multiple ways to perform any given task, but you have to consider the tradeoffs of each option. This example could also use membership checks to achieve essentially the same effect. The advantages of using membership checks are that you can perform custom processing and you don’t have to provide a specific public key. The disadvantages of this approach are that you can’t place the check at multiple levels (at least not easily) and you have to write a lot more code. Listing 5.3 shows the code for the alternate version of the CodeGroupComponent. (You can find the full listing for this example in the \Chapter 05\C#\CodeGroupComponent or \Chapter 05\VB\CodeGroupComponent folder of the source code located on the Sybex Web site.)

**Listing 5.3 Alternative Code Group Component**

```csharp
public SayHello()
{
    // Get the calling assembly.
    Assembly Asm;
    Asm = Assembly.GetCallingAssembly();

    // Get the evidence from the assembly.
    Evidence EV;
    EV = Asm.Evidence;

    // Create a zone membership condition check.
    ZoneMembershipCondition ZoneMember;
    ZoneMember = new ZoneMembershipCondition(SecurityZone.MyComputer);

    // Check for application directory membership.
    if (!ZoneMember.Check(EV))
    {
        DialogResult Result;  // Return value.

        // Ask whether the user wants to continue.
        Result = MessageBox.Show("Using this component from an " +
                                "intranet connection can cause " +
                                "a security " breach.
                                "Potential Security Error",
                                MessageBoxButtons.YesNo,
                                MessageBoxIcon.Warning);

        if (Result == DialogResult.No)
            // Throw an exception.
            throw(new
```
PolicyException('Calling assembly zone incorrect.');
}

// Examine the evidence for this component.
IEnumerator Enum =
    Assembly.GetExecutingAssembly().Evidence.GetHostEnumerator();

// Get the strong name for this component.
StrongName SN = null;
while (Enum.MoveNext())
    if (Enum.Current.GetType() == typeof(StrongName))
        SN = (StrongName)Enum.Current;

// Create the strong name membership condition.
StrongNameMembershipCondition StrongMember =
    new StrongNameMembershipCondition(SN.PublicKey, null, null);

// Check the strong name membership.
if (!StrongMember.Check(EV))
    // Throw an exception.
    throw (new PolicyException('Calling assembly isn't a " +
        'Demonstration_Strong_Name member."'));
}

The code begins by getting the calling assembly—the client that's calling on the compo-

tent for service. It uses the Assembly object to obtain evidence. The first check determines

whether the caller is in the correct zone. It does this by creating the ZoneMember object

and using the Check() method against the caller's evidence.

When the caller is in the wrong zone, the code asks about the potential for a security

breach. If the user answers no, the code throws an exception. This is an example of how the

second technique is superior to the first because you've added flexibility in handling the sit-

uation. The example uses a PolicyException with a custom message. You can also use a

SecurityException. However, the PolicyException is more precise and specific.

The second check determines whether the caller has the correct strong name. Of course, this

means constructing a StrongName object. The code obtains the strong name using an enumera-

tion that checks the executing assembly, rather than the calling assembly. The executing assem-

bly is the component and you want to verify that both the caller and the component have the

same strong name. The code uses an enumeration to find the executing assembly strong name.

The strong name handling demonstrates another way in which the second method is superior

to the first. You don't have to hard code the public key, so changing keys is a simple change and

recompile.
WARNING  It's easy to confuse the calling and executing assemblies, especially when you need to work with multiple client levels. Always verify the use of the GetExecutingAssembly() method for the host application and the GetCallingAssembly() method for the calling application. Otherwise, you could perform comparisons that end up checking the caller against itself.

After the code locates the strong name information for the executing assembly, it creates a StrongNameMembershipCondition object using just the SN.PublicKey value. Notice that the code uses a null value for both the name and version arguments. The StrongMember object will match any caller signed with the same public key as the component. If you also added the name and version number, the call would fail unless the caller is the same as the current component. As with the zone check, the code throws a policy exception if the public keys don’t match.

Installing the CodeGroupComponent
You must sign this component. Add the example component to the GAC by typing GAC-Util /i CodeGroupComponent.DLL at the command prompt. (You can remove the component later by typing GACUtil /u CodeGroupComponent.DLL.) However, there’s no requirement that you always add your component to the GAC. This step does make working with the component easier. The following steps tell you how to create the specialized strong name code group based on this component using the .NET Framework Configuration Tool:

1. Highlight the My_Computer_Zone entry within the Machine\Code Groups folder.
2. Click the Add a Child Code Group link. The .NET Framework Configuration Tool displays an Identify the New Code Group dialog box.
3. Type Demonstration_Strong_Name in the Name field and A code group used for demonstration purposes. in the Description field. Click Next. The .NET Framework Configuration Tool displays a Choose a Condition Type dialog box.
4. Select Strong Name from the condition list. The Choose a Condition Type dialog box changes to accept strong name information.
5. Check the Name and Version entries to enable these security entries.
6. Click Import. You’ll see an Import Strong Name from Assembly dialog box.
7. Locate the CodeGroupComponent.DLL file and highlight it. Click Open. The Choose a Condition Type dialog box should look similar to the one shown in Figure 5.12.

8. Click Next. The .NET Framework Configuration Tool displays an Assign a Permission Set to the Code Group dialog box.

9. Select FullTrust from the permission set list, and then click Next.

10. Click Finish to complete the process.

The current setup lets anyone using the current machine also access the component. If the user isn’t part of the current machine or tries to access the machine from another location, then the call will fail. You could easily change this to a user specific check by moving the code group to the User list.

Writing the Code Group Access Code

The example component file actually contains two components. The first uses the attribute technique, while the second uses the membership technique. The code in Listing 5.4 shows how to test both components. Notice that this code isn’t anything special—it could easily represent nonsecure code. (You can find the full listing for this example in the \Chapter 05\C#\CodeGroup or \Chapter 05\VB\CodeGroup folder of the source code located on the Sybex Web site.)
Listing 5.4  Code Group Component Testing Example

```csharp
private void btnTest_Click(object sender, System.EventArgs e)
{
    SayHello MyHello; // The test component.

    try
    {
        // Create the component.
        MyHello = new SayHello();

        // Test the component.
        MessageBox.Show(MyHello.DoSayHello());
    }
    catch (PolicyException PE)
    {
        // Display an error message.
        MessageBox.Show(PE.Message);
    }
}

private void btnTest2_Click(object sender, System.EventArgs e)
{
    SayHello2 MyHello; // The test component.

    try
    {
        // Create the component.
        MyHello = new SayHello2();

        // Test the component.
        MessageBox.Show(MyHello.DoSayHello2());
    }
    catch (SecurityException SE)
    {
        // Display an error message.
        MessageBox.Show(SE.Message);
    }
}
```

In both cases, the code creates the component and then calls on the one method it contains to display a string on screen. The only difference between this application and any application you might have created in the past is that this application requires a key—the same key you used for the component. Once you sign the test program, it becomes part of the Demonstration_Strong_Name code group created earlier. Run the **CASPol -machine -resolvegroup CodeGroup.EXE** command. You’ll see output similar to that shown in Figure 15.13. This output demonstrates that the application is part of the code group.
Testing the Code Group

Both versions of the test program always run on the local machine as long as you sign the calling application and the component with the same key. Try compiling the test application without a key and you’ll notice that it fails. One interesting test is to compile both the test application and the component using MyKey2 supplied with the Signed Client example in Chapter 4. The more flexible membership testing technique shown in Listing 5.3 still works, but the simpler technique fails because the attribute hard codes the key.

The real test occurs when you place the test application on a network drive. Because of the way that we installed the Demonstration_Strong_Name code group, the test applications will fail. However, you can easily change that outcome. Right-click the Demonstration_Strong_Name code group entry in the .NET Framework Configuration Tool and choose Cut from the context menu. Right-click the LocalIntranet_Zone entry and choose Paste from the context menu. These two steps move the Demonstration_Strong_Name code group entry.

WARNING Make sure you cut the entry and don’t copy it—if you copy the entry the test won’t work as anticipated.

Try the test application again. Notice that the attribute method still fails because the attribute hard codes the zone error reaction. The membership testing technique displays an error message. When you click Yes, this application succeeds because the custom coding enables the user to decide the severity of the security risk. Of course, you can choose any of a number of mechanisms to make the decision. The important point is that you can make a decision.
Adding New Permissions

The permissions that Microsoft provides as part of the .NET Framework will address many (if not most) common situations. However, one consistent element of programming is that the unexpected is the normal condition that developers face. A company usually has some kind of special need for their data that Microsoft can’t consider as part of the common scenario. The need could be something as simple as ensuring that no one outside the selected group can use the program. You may need to limit the functionality of a program based on the needs of the requestor. Perhaps managers have access to features that other employees don’t need.

In the Win32 environment, handling special situations is difficult because security is a hard-wired element of the operating system. You can only go so far in protecting an operating system because the role of the user and the nature of the environment aren’t considered. In fact, you can’t even garner enough information to develop a kludge to handle this situation in many cases.

The .NET Framework makes it possible to create a new permission. This permission can describe any set of security requirements you can imagine. When you want to limit application access based on some criteria, you can build a permission to handle that requirement. The permission acts as input to a code group that defines access to the permission based on a membership criteria. Finally, your application can demand the requisite permission and perform analysis of which features the requestor can access. It’s an extremely flexible way to define security.

The following sections take you through the process of creating a new permission and testing it within an application. In short, when you finish this section, you should know the basic process for creating any security requirement needed to ensure the integrity of your data, the safety of your resources, and the use of resources.

Creating a Permission Assembly

In general, you must create a new permission to implement a new security requirement that’s part of a code group. You can take several routes to achieve this goal. For example, you can create a new permission out of the available named permissions. We discussed that possibility in the “Making Permission Modifications” section of the chapter. In that example, we created an XML file to hold the existing definitions in a new combination. You can assign that definition to a code group and use it as part of an application setup.

This section takes a different approach. It assumes that the existing permissions are inadequate for your particular needs. Listing 5.5 shows the code used to create a custom permission. Of course, this is just the permission assembly. You still have more work to do to
implement the permission, but this is the starting point. (Note that this listing isn't complete. You can find the full listing for this example in the \Chapter 05\C#\CustomPermission or \Chapter 05\VB\CustomPermission folder of the source code located on the Sybex Web site.)

Listing 5.5  A Typical Custom Permission Assembly

```csharp
// This enumeration contains the acceptable permission values.
public enum SpecialSet
{
    None = 0,
    Some,
    All
}

[Serializable()]
public class Special : CodeAccessPermission, IUnrestrictedPermission
{
    // A required variable for unrestricted access.
    private Boolean _unrestricted;

    // A special variable for tracking access to this class.
    private SpecialSet _setValue;

    // At least one constructor should set the restricted state.
    public Special(PermissionState State)
    {
        if (State == PermissionState.Unrestricted)
            _unrestricted = true;
        else
            _unrestricted = false;

        // Make sure you also set any special property values.
        if (_unrestricted == true)
        {
            _setValue = SpecialSet.All;
        }
        else
        {
            _setValue = SpecialSet.None;
        }
    }

    // At least one constructor should set your special properties.
    public Special(SpecialSet SetValue)
    {
        // Set the property value.
        _setValue = SetValue;

        // You can assume an unrestricted state in some cases.
        if (SetValue == SpecialSet.All)
            _unrestricted = true;
        else
            _unrestricted = false;
    }
}
// This implementation also requires a special property.
public SpecialSet SetValue
{
    get { return _setValue; }
    set
    {
        // Set the new value.
        _setValue = value;

        // Determine whether unrestricted has changed.
        if (_setValue == SpecialSet.All)
            _unrestricted = true;
        else
            _unrestricted = false;
    }
}

public override IPermission Copy()
{
    // Create a new copy of the permission.
    Special Copy = new Special(PermissionState.None);

    // Define the properties.
    Copy._setValue = this.SetValue;

    // Set the restriction level.
    if (this.IsUnrestricted())
        Copy._unrestricted = true;
    else
        Copy._unrestricted = false;

    // Return the copy.
    return Copy;
}

public override void FromXml(SecurityElement elem)
{
    // Get the SetValue value.
    String Element;
    Element = elem.Attribute("SetValue");

    // Set the property values according to the
    // SetValue entry.
    if (Element != null)
    {
        switch (Element)
        {
            case "All":
                this._setValue = SpecialSet.All;
                this._unrestricted = true;
                break;
        } // end switch
    } // end if
} // end FromXml
break;

case 'Some':
    this._setValue = SpecialSet.Some;
    this._unrestricted = false;
    break;

case 'None':
    this._setValue = SpecialSet.None;
    this._unrestricted = false;
    break;

// Don't perform any more processing.
return;
}

// The XML file didn't contain a SetValue entry. Try
// the Unrestricted entry.
Element = elem.Attribute("Unrestricted");

if (Element != null)
{
    // Process using the Unrestricted value.
    this._unrestricted = Convert.ToBoolean(Element);

    // Set the SetValue value.
    if (_unrestricted)
        this._setValue = SpecialSet.All;
    else
        this._setValue = SpecialSet.None;
}
else
{
    // No one saved anything, so use defaults.
    this._setValue = SpecialSet.None;
    this._unrestricted = false;
}

public override IPermission Intersect(IPermission target)
{
    // Use a try...catch statement in case the caller sends
    // the wrong kind of permission.
    try
    {
        // If the permission is null, return null.
        if (target == null)
        {
            return null;
        }

        // Create a new copy of the permission. This is where the
// code will fail if the supplied permission is the wrong type.
Special Perm = (Special)target;

// Start checking permissions. Begin with the case where the incoming permission allows everything.
switch (Perm.SetValue)
{
    case SpecialSet.All:
        return this.Copy();
    case SpecialSet.Some:
        if (this.SetValue == SpecialSet.None)
            return this.Copy();
        else
            return Perm;
    case SpecialSet.None:
        return Perm;
}

// If all else fails, return a copy of this object.
return this.Copy();

catch (InvalidCastException)
{
    // Tell the caller there is an argument error.
    throw new ArgumentException("Wrong Argument Type ",
        this.GetType().FullName);
}

public override bool IsSubsetOf(IPermission target)
{
    // Use a try...catch statement in case the caller sends the wrong kind of permission.
    try
    {
        // If the permission is null, return false.
        if (target == null)
        {
            return false;
        }

        // Create a new copy of the permission. This is where the code will fail if the supplied permission is the wrong type.
        Special Perm = (Special)target;

        // Start checking permissions. Begin with the case where the incoming permission allows everything.
        switch (this.SetValue)
        {
case SpecialSet.All:
    return true;

case SpecialSet.Some:
    if (Perm.SetValue == SpecialSet.All)
        return false;
    else
        return true;

case SpecialSet.None:
    if (Perm.SetValue == SpecialSet.None)
        return true;
    else
        return false;

    // If all else fails, return false.
    return false;

catch (InvalidCastException)
{
    // Tell the caller there is an argument error.
    throw new ArgumentException("Wrong Argument Type ",
        this.GetType().FullName);
}

public override SecurityElement ToXml()
{
    // Create a security XML encoding element.
    SecurityElement SE = new SecurityElement("IPermission");

    // Determine the permission type.
    Type Inst = this.GetType();

    // Determine the assembly name.
    StringBuilder AssemblyName;
    AssemblyName = new StringBuilder(Inst.Assembly.ToString());

    // Replace double quotes with single quotes for the XML file.
    AssemblyName.Replace('"', '');

    // Create the required attributes.
    SE.AddAttribute("class", Inst.FullName + ", " + AssemblyName);
    SE.AddAttribute("version", "1");
    SE.AddAttribute("Unrestricted", _unrestricted.ToString());
    SE.AddAttribute("SetValue", _setValue.ToString());

    // Return the resulting security element.
    return SE;
}

public bool IsUnrestricted()
{
Notice that the Special class derives from the CodeAccessPermission class and implements the IUnrestrictedPermission interface. When you create a permission class, you must implement the IPermission interface in some way. Microsoft recommends that you use the CodeAccessPermission class to meet this requirement because otherwise you have to write code for all of the common IPermission interface members.

**TIP**

It may be tempting to say that the CodeAccessPermission class is the only class you can use to derive a new permission class. This statement isn’t true. You can derive from any permission class and it may be better if you do derive from other classes in some situations. The CodeAccessPermission class is code access specific. When you need to create a new identity permission, you may want to derive your new permission from one of the identity permissions such as the ZoneIdentityPermission class. The point is that you don’t want to limit your choices based on the methods used in this section or online—choose the best class to meet your specific need and then modify that implementation as needed.

The Special class also has the [Serializable()] attribute attached. You don’t have to include this attribute if you only intend to use imperative security with the new permission. However, it’s usually a good idea to include the [Serializable()] attribute in case you decide to implement declarative security later. Otherwise, you see a strange error message (it seems to vary by phase of the moon) that has nothing to do with the actual problem when you attempt to use the permission in declarative mode.

The code begins by creating two private variables. Part of the custom permission specification states that you must keep track of the unrestricted state. However, you’ll also want to track custom values.

The example provides two constructors. Look at most of the .NET Framework permission classes and you’ll notice that they also have a minimum of two constructors. The reason is that you must create one constructor that accepts a PermissionState enumeration value as input. This value sets the unrestricted value. Notice that the example code also sets the custom _setValue variable based on this input. Although this isn’t a requirement, you’ll find that your custom permission works better if you make some assumptions based on the unrestricted value when a caller uses the PermissionState enumeration as input. Likewise, the second constructor handles the unrestricted state based on the SetValue input.
It may not seem like a very important addition, but you should provide a property for every custom status value your permission supports. The example provides this feature to make it easier to implement declarative security. It's also a good addition when the caller wants to learn the current permission state. A component may rely on specific caller states to grant or deny permissions.

Another rule for creating permissions based on the CodeAccessPermission class is that you must override the Copy(), Intersect(), IsSubsetOf(), ToXml(), and FromXml() methods. It's also helpful to override the Union() method, but not absolutely required. The example code shows typical implementations that include functionality for a special security state property. Notice, for example, that the Copy() method sets the values of the new copy of the Special object before returning it.

The FromXml() and ToXml() methods are paired. Microsoft doesn't place any restrictions on the format of the XML file except that the FromXml() must be able to read anything produced by the ToXml() method. The XML tags must also conform to the format produced by the SecurityElement object. This XML data isn't strictly correct or even readable by a typical XML utility. Figure 5.14 shows typical SecurityElement object XML output. As you can see from the figure and the associated code, the general idea is to use an existing object to generate XML settings that the code can read in and use to create a replica object.

The Union(), Intersect(), and IsSubset() methods all perform manipulations of security settings based on the comparison of the existing permission with an input permission. The Union() method creates a result that's the complete set of settings of the two permissions. The Intersect() method returns a result that's the least of the two permissions. The IsSubset() method returns a Boolean value that states whether the input permission is a true subset (or an equal) of the existing permission. In all three cases, the code must check for a null input and act accordingly. After checking for a null input, the code coerces the input permission to an object of the same type as the current permission. If the coercion succeeds, the code performs the required comparison. As shown in the examples, you must provide handling for the InvalidCastException exception.

The IUnrestricted interface is relatively easy to implement. All you need to provide is an IsUnrestricted() method that returns the unrestricted value of your permission.

**Figure 5.14:**
Create the specialized XML used for security needs in the FromXml() and ToXml() methods.
Listing 5.5 showed the imperative portion of the permission. However, you'll likely want to implement declarative security too. Listing 5.6 shows a typical declarative security implementation. The attribute class appears as part of the Special class module, which makes it easy for the two to work together (and lessens the work you must perform to create the declarative version).

Listing 5.6 Declarative Security Addition

```csharp
// Add an attribute class so the permission includes both imperative
// and declarative security support.
[AttributeUsageAttribute(AttributeTargets.All, AllowMultiple = true)]
public class SpecialAttribute: CodeAccessSecurityAttribute
{
    // A required variable for unrestricted access.
    private Boolean _unrestricted;

    // A special variable for tracking access to this class.
    private SpecialSet _setValue;

    // You must include a property for changing the unrestricted value.
    public new Boolean Unrestricted
    {
        get { return _unrestricted; }
        set { _unrestricted = value; }
    }

    // This implementation also requires a special property.
    public SpecialSet SetValue
    {
        get { return _setParameter; }
        set { _setParameter = value; }
    }

    // Create the constructor.
    public SpecialAttribute(SecurityAction Action) : base (Action)
    {
    }

    // Override this method so the attribute returns the correct
    // object type.
    public override IPermission CreatePermission()
    {
        // Handle the case where the SetValue argument is supplied.
        switch (SetValue)
        {
            case SpecialSet.All:
```
The `SpecialAttribute` class derives from `CodeAccessSecurityAttribute`. However, as with the imperative version of this example, you can implement a completely custom declarative version.

You must provide properties for every value that the user can change as part of the attribute. These properties contain the values when the system calls your code. For example, when the caller uses the

```
[SpecialAttribute(SecurityAction.Demand, SetValue=SpecialSet.All)]
```

Attribute, the `SetValue` property receives the `SpecialSet.All` value. If you don't provide the requisite property, then your code won't receive the value.

The only method you must implement is `CreatePermission()`. The example shows one method of handling the input. If the `SetValue` property isn't set, the code looks at the `Unrestricted` property. You must handle the unrestricted setting at a minimum because many types of XML file handling rely on it. Note that in all cases, the `SpecialAttribute` class merely returns a new copy of the `Special` class.

**Designing a Component to Use the Custom Permission**

It's time to use the new `Special` permission class to protect a component. Listing 5.7 shows typical component code. It depends on the imperative method, but you could easily use the declarative method as well. (You can find this example in the \Chapter 05\C#\CustomPermissionComponent or \Chapter 05\VB\CustomPermissionComponent folder of the source code located on the Sybex Web site.)
Listing 5.7  A Custom Permission Test Component

```csharp
public class TestClass
{
    public TestClass()
    {
        // Demand the required permission.
        Special ClientCheck;
        ClientCheck = new Special(SpecialSet.All);
        ClientCheck.Demand();
    }

    public String Random()
    {
        // Create a random number and place it in a string.
        Random MyRand;
        MyRand = new Random(DateTime.Now.Second);
        return 'Next random number: ' + MyRand.Next().ToString();
    }
}
```

When a caller tries to instantiate a copy of the component, the code checks for the correct security level using the `Demand()` method. Incorrect credentials produce a `SecurityException` exception. The `Random()` method is only used to test component access—it generates a string that includes a random number.

**Designing an Application to Test the Custom Permission**

The test program checks grant and deny security conditions when using the Special permission. Listing 5.8 shows the code used to instantiate a copy of the test component and display a string using the `Random()` method. (You can find this example in the `\Chapter 05\C#\CustomPermissionTest` or `\Chapter 05\VB\CustomPermissionTest` folder of the source code located on the Sybex Web site.)

Listing 5.8  Custom Permission Test Application

```csharp
private void btnGrant_Click(object sender, System.EventArgs e)
{
    // Create a new permission.
    Special Perm;
    Perm = new Special(SpecialSet.All);
    Perm.Demand();

    // Try to create the custom component.
    try
    {
        TestClass TC;
```
TC = new TestClass();

    // Display a result.
    MessageBox.Show(TC.Random());
} catch (SecurityException SE)
{
    // Display the error.
    MessageBox.Show(SE.Message);
}

private void btnDeny_Click(object sender, System.EventArgs e)
{
    // Create a new permission.
    Special Perm;
    Perm = new Special(SpecialSet.All);
    Perm.Deny();

    // Try to create the custom component.
    try
    {
        TestClass TC;
        TC = new TestClass();
        // Display a result.
        MessageBox.Show(TC.Random());
    }
    catch (SecurityException SE)
    {
        // Display the error.
        MessageBox.Show(SE.Message);
    }
}

As you can see, both methods are essentially the same. The only difference is that one uses the Demand() method to gain access to the required permission and the other uses the Deny() method to refuse the permission. Clicking Deny produces a message box that contains a security error message. The message box correctly identifies the CustomPermission.Special class as the source of the error.

**Using Policy Objects**

Creating a permission lets you create custom policy objects using the CASPol command line tool or the .NET Framework Configuration Tool. As your organization depends more on .NET to ensure the safety of your data, creating custom policies becomes more important.
You may run into a situation where you want to return a system to the default security policy before you make any other changes. For example, this technique works well if you use a batch file for setup and want to ensure the system is in a baseline configuration before you begin the batch file. To return the system to a default security configuration, type `CASPol -reset`.

This section of the chapter relies on the custom permission created in the “Adding New Permissions” section of the chapter. In this case, we’ll use the Special permission class as the basis for defining a new policy.

### Installing a New Permission

When you create a code group, you begin by looking at the membership condition and the permission granted by fulfillment of that condition. However, before you can grant a permission, it must appear as part of the standard security policy used by the .NET Framework. You must make the system aware of the permission in order to use it as a basis for any policy. The first task is to add the permission class to the Policy Assemblies folder at one or more levels. Repeated attempts show that this part of the .NET Framework Configuration Tool doesn’t work as expected, so you must type `CASPol -machine -addfulltrust CustomPermission.DLL` and press Enter at the command prompt.

At this point, you need to create the permissions used with the code groups. Unfortunately, you can’t just use the existing `CustomPermission.DLL` file as you might for other purposes. Neither of the utilities discussed so far will accept anything other than an XML file as input, which means you need to build the requisite program. Listing 5.9 shows that this program isn’t difficult to write—just annoying. (This listing isn’t complete. However, you can find the complete listing for this example in the \Chapter 05\C#\CustomPermissionXMLGen or \Chapter 05\VB\CustomPermissionXMLGen folder of the source code located on the Sybex Web site.)

#### Listing 5.9  XML Generator for a Custom Permission

```csharp
private void btnTest_Click(object sender, System.EventArgs e)
{
    Special Perm; // Special permission object.
    NamedPermissionSet NPS; // Permission set container.
    StreamWriter Output; // Data storage to disk.

    // Create the All special permission.
    Perm = new Special(SpecialSet.All);

    // Define a named permission set.
    NPS = new NamedPermissionSet("SpecialPermissionAll",
                                PermissionState.None);

    // ... (rest of the code)
}
```

Copyright © 2003 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved.
NPS.Description = "Grants all permissions for special permission."
NPS.AddPermission(Perm);

// Generate the All permission output file.
Output = new StreamWriter('SpecialPermissionAll.XML');
Output.Write(NPS.ToXml());
Output.Close();

... Some and None Permission Generation ...

// Operation complete.
MessageBox.Show('Output Complete!');
}

As you can see, the code begins by building a named permission set and a permission to place in it. It’s important to create one permission for each permission state that you want a code group to support. Because the code is relatively easy, the example creates permission files for all three of the example permission states. The program uses a StreamWriter to output the information to an XML file. The result of running this program is three XML files that you can use to create permissions. The following steps show how to perform this part of the process using the .NET Framework Configuration Tool.

1. Select the Runtime Security Policy/Machine/Permission Sets folder.
2. Click Create New Permission Set. You’ll see a Create Permission Set dialog box.
3. Select the Import a permission set from an XML file option.
4. Click Browse to locate the XML file or type the location of the file. Your dialog box should look similar to the one shown in Figure 5.15.

FIGURE 5.15:
Define the location of the XML file containing the new permission you want to create.
5. Click Finish. The .NET Framework Configuration Tool will add the new permission to the list.

6. Repeat Steps 2 through 5 for each of the permissions you want to add (three for this example).

Figure 5.16 shows the Permission Sets folder with all three permissions added.

**Figure 5.16:** A view of the Permission Sets folder with custom permissions added.

Creating a Code Group Based on the Permission

Now that you have a new permission to use, you’ll likely want to create a code group to use it. Using a custom permission is like any other permission you might have used. The only difference is that you use a custom permission instead of one of the supplied permissions. The “Installing the CodeGroupComponent” section of the chapter provides an example of the techniques you use to create a new code group. Here are the quick steps for creating a code group for the custom permission.


2. Click Add a Child Code Group. You’ll see a Create Code Group dialog box.

3. Type `Special_Permission_Test` in the Name field and `Tests the CustomPermission .Special class permission` in the Description field, and then click Next.

4. Select All_Code as the membership condition and click Next.
5. Select SpecialPermissionAll as the permission and click Next.
6. Click Finish. The .NET Framework Configuration Tool creates the new code group.

**Designing a Named Permission Test Program**

It's the decisive moment. The example in this section shows how to check for the existence of a named permission on your system. In this case, the program only provides output for the custom permissions you just installed. Listing 5.10 shows the code that you'll need.

(The listing in this chapter is incomplete. You can find the complete listing for this example in the `Chapter 05\C#\NamedPermission` or `Chapter 05\VB\NamedPermission` folder of the source code located on the Sybex Web site.)

**Listing 5.10 Obtaining a Named Permission**

```csharp
private void btnTest_Click(object sender, System.EventArgs e)
{
    IEnumerator Policies;   // Security policies.
    PolicyLevel Policy;     // A single policy.
    NamedPermissionSet NPS;        // The special permission set.
    Special Perm;       // The special permission.
    StringBuilder Output;     // The output value.

    // Initialize the output.
    Output = new StringBuilder();

    // Get the all of the policies.
    Policies = SecurityManager.PolicyHierarchy();
    while (Policies.MoveNext())
    {
        // Get the current policy.
        Policy = (PolicyLevel)Policies.Current;

        // Check for the SpecialPermissionAll named permission set.
        NPS = Policy.GetNamedPermissionSet("SpecialPermissionAll");

        // Process the named permission set when available.
        if (NPS != null)
        {
            // Get the named permission and check its values.
            Perm = (Special)NPS.GetPermission(typeof(Special));
            Output.Append("SpecialPermissionAll\nUnrestricted: ");
            Output.Append(Perm.IsUnrestricted().ToString());
            Output.Append("\nSetValue: ");
            Output.Append(Perm.SetValue.ToString());
        }
    }

    // Other Permission Checks ...
}
```

Copyright © 2003 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved.
The code begins by accessing the SecurityManager and gaining access to the policy hierarchy. The code moves from one policy to the next in search of the named permission sets you created. You use the GetNamedPermissionSet() method to access a named permission set. If the permission set is missing, the return value is null.

Once the code locates the named permission set, it uses the GetPermission() method to obtain access to the actual permissions. The code coerces the IPermission output to the Special type. The permission has all the same features as the custom permission you created. Figure 5.17 shows the output from this example. Note that all of the values reflect the custom permission code.

**TIP**

You’ll likely find that the GetPermission() method doesn’t work as advertised in Visual Basic .NET. The technique to get around this problem is to use an enumeration. While using this method isn’t as fast or memory efficient as the C# alternative, it is reliable. In addition, you’ll run into casting problems with Visual Basic when using the Special permission. The alternative in this case is to use an Object instead of the actual type. See the Visual Basic .NET version of this example for details.
Summary

This chapter has demonstrated several new features of code groups and policies. Now that you’ve completed this chapter, you should have a better idea of how both .NET security features work at a low level. More important, you know how to manage these security features so that you can prevent most security problems and detect those that do get past your defenses. The goal of this chapter is to help you understand how policies and code groups work together to create a cohesive .NET security strategy.

Now that you have a better idea of how policies work, you should define a default security policy for your system at all three levels: enterprise, machine, and user. Make sure you use the tips found in the chapter to refine your security strategy. Remember that it’s important to have a written policy that works with the software policies you create.

Chapter 6 discusses the process of validation and verification. Validation is the process of checking that code, data, and resources are intact. You use validation to detect any changes that a cracker makes to your system. It’s also useful for ensuring your code doesn’t contain any changes before you run it. Verification is the process of determining identity. In this case, identity includes both the caller and recipient. It’s important to verify the identity of both servers and workstations in a world where crackers use both ends of the communication channel to overcome and overwhelm security measures.