6
ClickOnce Security

WHEN PLANNING FOR DEPLOYMENT, you need to consider a number of different aspects with respect to security. You need to consider

- How to protect the client machine from being compromised by your application's installation or execution
- How to protect the application files from being tampered with on the deployment server
- How to implement authentication and authorization based on the user's identity
- What you want to allow the application to do based on the identity of the application publisher

ClickOnce, the .NET Framework, and the Windows operating system provide facilities to address all of these considerations. This chapter will discuss these different aspects and give you a solid understanding of what protections ClickOnce provides, and how you can customize those protections to suit the needs of your particular application.

ClickOnce Security Overview

ClickOnce is designed to be a trustworthy deployment mechanism for smart client applications. This means that ClickOnce is designed to protect

the client machine from being harmed by applications that it deploys. ClickOnce provides protection for the client machine at install time and at runtime, ensures that the client machine and users can identify who the publisher of the application is, and protects the application's files to ensure than no one can tamper with them after the publisher has published the application.

ClickOnce runtime protection is based on the application's identity, not on the user. ClickOnce is specifically designed to enable low-privilege users to deploy and launch smart client applications without administrator intervention. The user identity is not used directly by ClickOnce in any way. However, that does not mean that your ClickOnce application will be unprotected with respect to user privileges either. You can take advantage of .NET role-based security to prevent users from using functionality in your application if they do not have sufficient rights. Additionally, the client machine's operating system will still enforce access controls based on the logged-in user, such as limiting access to files, folders, or the registry if the user is not part of the access control list for those resources.

ClickOnce Deployment-Time Protections

ClickOnce security protection comes into play as soon as an application or update is deployed to the client machine. When files are deployed to the client machine through ClickOnce, they are isolated per user, per application, and per version under the user's profile. The application deployment itself is nothing more than a series of files copied into an isolated folder under the user's profile. If you have worked with .NET isolated storage before, the ClickOnce cache folders are similar in concept, but located in a different place under the user's profile. You cannot execute any custom installation steps that make modifications to the local machine as part of the ClickOnce deployment itself (see Chapters 7 and 8 for more information on custom installation requirements). As a result of this design, there is no way that the act of deploying an application to a client machine through ClickOnce can harm other applications or data on the machine.

ClickOnce Runtime Protections

ClickOnce and the .NET runtime provide runtime protections for the client as well. ClickOnce relies on the Code Access Security (CAS) infrastructure of the .NET Framework for enforcing those runtime protections, but ClickOnce security is configured and managed a little differently than for non-ClickOnce deployed applications. For a quick overview of CAS, see the sidebar entitled A Short Primer on Code Access Security.

A Short Primer on Code Access Security

The .NET Framework provides Code Access Security (CAS), which is a security mechanism that complements Windows access control and .NET role-based security. Both of these are based on the identity of the user who is executing the code. CAS lets you control the permissions the executing managed code has based on evidence associated with the executing code. CAS is a fairly complicated topic that many developers do not even know exists, or if they do, they do not really understand how it works.[1]

Evidence is based on either the identity of the code itself or the launch location of the code. The code's identity is determined by one of several forms of code identity that are embedded in the assembly manifest at compile time. These forms of identity include a hash of the assembly contents, an assembly strong name, or a publisher certificate. The location the code is being executed from can be determined by the .NET runtime based on the path used to load an assembly, and can be associated with a local machine directory, URL, site, or zone.

CAS is driven by security policies that can be administratively configured through the .NET Configuration tool or set programmatically. A complex schema of security objects drives the application of CAS by the runtime. Each machine has several security policies defined for it. Each ***security policy*** is composed of a collection of ***code groups,*** and each

Continues

1. For a deep understanding of Code Access Security, including how it works, how to configure it, and how to programmatically invoke it, I recommend *Programming .NET Components, Second Edition*, by Juval Löwy (O'Reilly & Associates, 2005).

code group is composed of a form of ***evidence*** (also called *membership criteria*) and an associated ***permission set.*** Each permission set is composed of a collection of individual ***permission types,*** which correspond to the discrete operations or resources that you want to protect, such as the file system, user interface, security capabilities, or the registry. For each permission type, there are fine-grained options that you can control. For example, you can control what specific set of URLs an application is allowed to access for a permission of type `WebPermission`.

For applications that are not launched through ClickOnce, CAS evaluates the evidence available for an assembly as the runtime loads it. It compares the evidence for the assembly against each code group's membership criteria (form of evidence associated with the code group). If the evidence presented by the assembly meets the code group's membership criteria, the assembly will be granted the code group's permission set. The runtime follows this process for all of the code groups in all of the policies on the system to come up with a final set of permissions that the assembly is granted for execution. As the code in the assembly is invoked, security demands can be made programmatically (typically by Framework assemblies) that ensure that the assembly and all of its callers have been granted the required permission for the operation or resource access that is being performed. If the assembly itself or any of the calling assemblies do not have the demanded permission, a security exception will be thrown.

By controlling the configuration of CAS, you can explicitly grant or deny permissions to any assembly on a machine for various operations and resources. For example, you can configure a server machine so that only .NET Framework code and code signed by your development organization's strong name are allowed to run on that machine. By doing so, you can ensure that if any other managed assemblies make it onto that server somehow, they will be unable to run unless an administrator intervenes and grants those assemblies a set of permissions.

When you install .NET, there are a predefined set of CAS policies and code groups installed on the machine. The built-in code groups are all based on the location that code is executing from. By default, any assemblies

installed on the local machine have full trust (unrestricted permissions). This corresponds to the My Computer CAS zone. If assemblies are loaded from somewhere other than a local disk, the runtime can evaluate the path used to load the assembly and will compare it to the membership criteria of the other code groups. The other built-in code groups include ones for paths that evaluate to the LocalIntranet zone, Internet zone, Trusted Sites, or Restricted Sites. The LocalIntranet and Internet are the most common other security zones applied, based on the path to the assembly being loaded. The Trusted and Restricted Sites zones are based on URLs configured through Internet Explorer security settings.

ClickOnce security is applied at the application level, instead of at the individual assembly level as it is in a normal .NET application. Your entire ClickOnce application (the application executable and all assemblies that it loads) are treated as a single unit for the purposes of deployment, versioning, and security. When an application is deployed through ClickOnce, the application manifest specifies what security permissions the application needs to run. These permissions are based on CAS.

As the application is launched by ClickOnce, the runtime first evaluates what URL or UNC path was used to deploy the application to the client machine (the path to the deployment manifest on the deployment server). This path is treated as the launch path. Based on this path, the runtime associates your application with one of the built-in location-based code groups (My Computer, LocalIntranet, Internet, Trusted Sites, or Restricted Sites zones). The runtime determines what set of permissions should be granted to your application based on the zone that it was launched from and compares that to the set of permissions requested by the application.

If the requested permissions in the application manifest are less than or equal to the set that would be granted based on the launch zone, then no elevation of permissions needs to occur and the application can simply launch and run. If the application attempts to perform an operation that exceeds the granted permissions, then a `SecurityException` will be thrown.

To see this in action, do the following.

1. Create a new Windows Application project in Visual Studio, and name the project **RuntimeProtectionApp**.
2. From the toolbox, add a button to the form.
3. Double-click on the button to add a Click event handler for the button.
4. Add the following code to the event handler:

```
private void button1_Click(object sender, EventArgs e)
{
    StreamWriter writer = new StreamWriter("AttemptedHack.evil");
    writer.WriteLine("If I can do this, what else could I do??");
    writer.Close();
}
```

5. Add a using statement for the `System.IO` namespace to the top of the file:

```
using System.IO;
```

6. Open the project properties editor (choose Project > RuntimeProtectionApp Properties).
7. On the Security tab, check the checkbox labeled *Enable ClickOnce Security Settings*, and click the radio button labeled *This is a partial trust application* (see Figure 6.1).
8. Publish the application by choosing Build > Publish RuntimeProtectionApp.
9. When the Publish wizard appears, click the Next button.
10. In the second step of the Publish wizard, select the option to make the application available online only (see Figure 6.2) and then click Finish.
11. Click the Run button in the publish.htm test page when it appears in the browser. This launches the application.
12. Press the button that you added to the form in step 2, causing the application to try to write a text file to the current working directory (which in this case is the C:\Windows\Microsoft.NET\Framework\v2.0.50727 folder, since the application is marked for partial trust as discussed in Chapter 5).

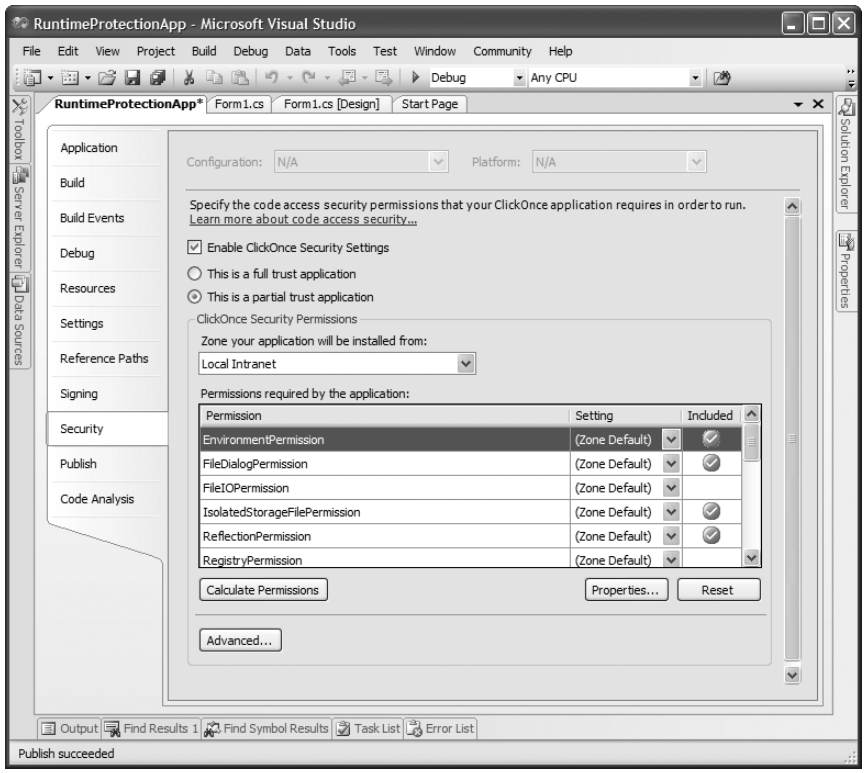

FIGURE 6.1: ClickOnce Security Settings

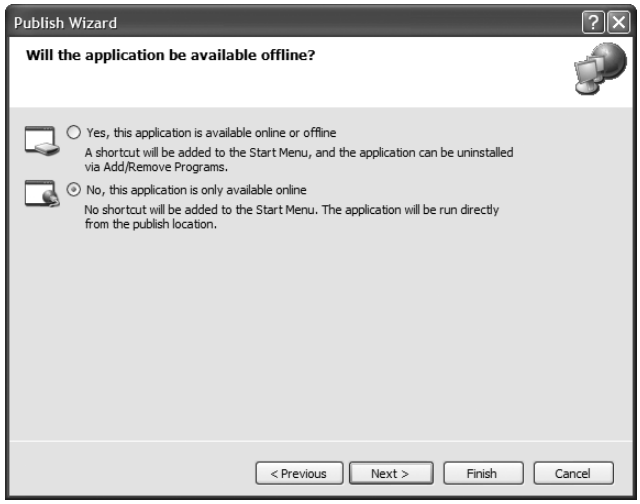

FIGURE 6.2: Selecting Install Mode in the Publish Wizard

13. A `SecurityException` will be thrown for the `FileIOPermission` type, because the default LocalIntranet zone security permissions do not include that permission. The permission is demanded by the `StreamWriter` class when you construct an instance of the `StreamWriter`. Since the application does not catch the exception, the dialog shown in Figure 6.3 will display.
14. Click the Quit button to exit the application.

In this example, the application requested permissions that did not exceed the permissions granted by the launch zone. This is because you selected partial trust and the default zone for partial trust is the Local Intranet zone. When you installed the application by clicking on the Run button in the publish.htm test page, the address used was http://<your-machine-name>/RuntimeProtectionApp/RuntimeProtectionApp.application. The runtime evaluates this address to the Local Intranet zone (based on the server address portion of the URL: http://<your-machine-name>/) and compares the requested permissions in the application manifest to the permissions for that zone. Since they match, no additional prompting is needed based on security and the application launches.

However, just because the application only requests a certain set of permissions based on its manifest does not mean that there is not code in that application that might try to do some operation that exceeds the granted set of permissions. In this example, the application contains code that tries to perform a file write to the local directory. That operation triggers a check for `FileIOPermission` for the file that is being written. Since the Local Intranet zone does not include that permission, a `SecurityException` is thrown at that point.

These protections are designed to ensure that your application does not inadvertently do something on the user's machine that it was not designed to do. This could result from bugs in your code, debug code that was left behind unintentionally, or it could happen if your application manages to load some other assembly that does something more than you expect it to. For example, suppose you design a smart client application that acts as a data entry client for a distributed application. Based on your design, that application should only present a rich interactive user interface for the user

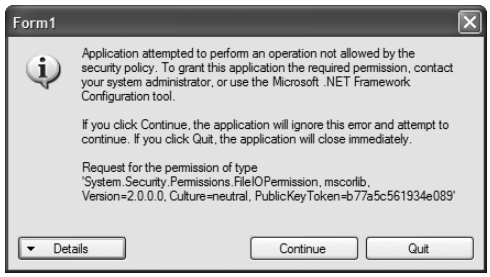

Figure 6.3: Unhandled Exception Dialog

to view, enter, and manipulate data that gets passed to your middle-tier application server through Web services.

Suppose you choose to use some third-party UI component to speed your development. Unknown to you, the code inside that component collects any values that are entered through its controls and transfers that data to some unknown location via a Web request for intelligence gathering. If you deployed this application with full trust, the component would be able to do just that and you may never even know it is happening behind the scenes. However, if you deployed your application with partial trust and restricted the `WebPermission` options to only allow calls to your middle-tier servers, then a security exception would be thrown when that nefarious component tried to do its evil deeds. By restricting the permission set, you would be protecting the user from that hidden data transfer.

Using a restricted set of permissions through partial trust is an excellent way to prevent your application from doing anything it was not designed to do. Unfortunately, for a lot of meaningful things that you might want to do in your application, such as doing on-demand updates through ClickOnce or making remote calls through Windows Communication Foundation, you will be required to set your application for full trust due to the more advanced things the Framework does for you under the covers to provide those capabilities. You can still lock down permissions for specific sections of your code, however (see the section Adding Restricted Code Sections later in this chapter for an example of how to do that).

If the application manifest requests permissions that exceed the launch zone permissions, such as full trust, then those permissions need to be granted to the application somehow so it can launch. This can be done either

through user prompting (the default) or automatically based on trusted publishers. Both of these approaches are covered later in this chapter.

> **NOTE** **ClickOnce Evaluates Permissions at the Application Level**
>
> An important distinction between the way CAS evaluates permissions and the way ClickOnce does so is where the scoping boundary is for a set of granted permissions. CAS evaluates permissions on an assembly-by-assembly basis, at the point where the assembly is loaded. ClickOnce evaluates permissions at the application boundary when the application is launched, and further checks are not done when each assembly is loaded. Additionally, ClickOnce only considers the built-in location-based code groups (My Computer, Internet, Local Intranet, Trusted Sites, and Restricted Sites zones) to determine what set of permissions the application should be given by default based on its launch URL. If you have custom code groups defined for which your assemblies would normally meet the membership criteria, those code groups will not be factored into what set of permissions the runtime will give your application by default.

ClickOnce Size Limitations for Online-Only Applications

A partial trust online-only application can run without any user prompting, depending on the permissions the application requires and the zone it is running from. To prevent such an application from filling up the hard disk by downloading many large files, ClickOnce restricts the total size of a partial-trust online-only application to be half the online cache quota on the machine. This size is checked at download time as bits are being downloaded, and the ClickOnce launch will fail once the limit is exceeded. The default cache quota is 250MB, so partial-trust applications larger than 125MB should ask for full trust.

ClickOnce Tamper Protections

ClickOnce protects the files that your application is composed of by using digital signatures. When you publish an application with ClickOnce, you have to sign the deployment and application manifest with an Authenticode Class 3 Code Signing publisher certificate. Authenticode certificates are

based on public-private key cryptography. Publisher certificates contain both a public and a private key. The public and private keys have a mathematical relationship that makes it so anything you encrypt with one of the keys, you can decrypt with the other. However, the complexity of the mathematical relationship is such that it is extremely difficult to come up with one key when you just have the other. With the strength of current cryptographic keys, it would take hundreds or thousands of years of heavy-duty computing to figure out the value of one key if you just know the value of the other.

As the names imply, the intent is that you keep one key (the private key) to yourself, but you can freely hand out the public key to anyone who wants it. Once others have your public key, you can encrypt a message or file with your private key and give the message or file to them, and they can decrypt it using the public key with a strong assurance that the message or file they decrypted actually came from you (or at least someone who has access to your private key). Likewise, they can encrypt a message or file with your public key and give it to you, and they can be sure that only you can decrypt that message or file and see the contents.

When you sign a file with a certificate, the signing mechanism computes a hash of the file's contents using cryptographic methods. In computing the hash, it disregards a reserved section of the file into which it will insert the digital signature once is has been computed. Once the hash has been computed, the hash is encrypted with the private key of the publisher certificate. The encrypted version of the hash is the digital signature. This signature and the public key from the certificate used to encrypt the hash are inserted into the reserved location in the file. Now anyone who receives that file can compute the file's current hash using the same algorithm that was used to generate the original hash. They can then extract the digital signature and decrypt it using the public key embedded in the file with the signature. After they have decrypted the signature, they have the original hash that was computed by the publisher. If they compare the original hash and the hash they just computed, they can confirm that no one has tampered with the file since it was signed by the publisher, because any modifications to any part of the file will modify the computed hash and it will be different from the original hash.

This approach is used by ClickOnce to digitally sign your deployment and application manifests when you publish your application. It is also used by .NET for strong naming assemblies. Strong naming is just a similar digital signature approach. In the case of ClickOnce, the digital signature is embedded in the manifests as XML. In the case of strong naming, the digital signature is computed when an assembly is compiled, and is embedded in the assembly manifest in binary form.

In addition to digital signatures providing a guarantee that the manifests have not been tampered with since you published your application, they also provide tamper protection for all of your application files. When your application manifest is generated, a hash of each of the files in the application is put into the application manifest along with the rest of the file information. When ClickOnce deploys or updates your application, it computes the hash of each file as it is downloaded from the server and compares the hash to the one embedded in the downloaded application manifest. Since the application manifest is signed and can't be tampered with to change the hash values for application files, there is no way for someone to tamper with any of your application files, because ClickOnce will refuse to launch your application if the application file hashes don't match after they have been downloaded.

Internet Explorer Security Settings Affecting ClickOnce

Internet Explorer has several zone security settings that will impact your users' ability to launch a ClickOnce application on their machines.

- **Script Activation:** By default, script activation is disabled for ClickOnce applications coming from the Internet zone on Windows XP with Service Pack 2 and later platforms. This means that an Internet Web site cannot launch a ClickOnce .application file with a script. The setting that controls this in Internet Explorer is in the Tool > Internet Options > Security Tab > Custom Level button > Downloads > Automatic Prompting for File Downloads. If this is set to *Enable*, script activation of ClickOnce applications is allowed. If this is set to *Disable*, script activation is disallowed. The default setting is *Enable* for Intranet and *Disable* for Internet.

- **Disable ClickOnce MIME Handler:** If Downloads > File Download is set to *Disable*, launching any ClickOnce application over the Web (http or https) will result in the Security Alert message, "Your current security settings do not allow this file to be downloaded." By default this setting is enabled for all zones, so this will not usually be a problem.
- **Disable Managed Code:** If .NET Framework-Reliant Components > Run Components Not Signed with Authenticode is set to either *Disable* or *Prompt*, ClickOnce will be disabled. This setting must be set to *Enabled* for ClickOnce to work. The default value for this setting for all zones is *Enabled*.

Another Internet Explorer-related setting that you may want to be aware of is a registry key setting that determines whether users are prompted with a download dialog when they click on a link that points to a ClickOnce deployment manifest. The registry key in question is HKEY_CURRENT_USER\Software\Policies\Microsoft\Internet Explorer \Restrictions\AlwaysPromptWhenDownload. When this DWORD value is set to 1, you will always get a file download prompt before the ClickOnce launch process starts. This registry key is not set by default, which lets ClickOnce start the launch process immediately when a link is clicked on.

Configuring ClickOnce Security Permissions

The permissions a ClickOnce application requires to run are determined by its application manifest. These permissions are populated by Visual Studio when you publish your application based on the project properties. You can configure these security permissions on the project properties editor's Security tab. You can also modify them to a certain degree after you have published from Visual Studio using the Mage SDK tools. I'll cover both approaches in this section.

Regardless of which tool you use, you have two choices at the top level—you can request full trust or partial trust. Full trust means that you do not want your application constrained by CAS in any way at runtime on the client. This corresponds to the *unrestricted* permission set in CAS. When you select this setting, your application code and any code it calls

will not be restricted in any way based on CAS. Keep in mind that CAS is separate and distinct from user-based security. So depending on the users' rights, they may still be prevented from doing certain things, either by role-based security code in your application or by the operating system if they try to access something on the system through your application that they do not have Windows access control privileges to use. But as far as ClickOnce and CAS are concerned, if the application has full trust, it can do whatever it likes.

When you choose partial trust, you have to specifically select a set of permissions that you want to include in the requested permissions for the application. You can base this on one of the predefined zone-based permission sets, such as Local Intranet or Internet, or you can use a custom set of permissions to request the specific permissions that correspond to the operations and resources your application uses by design. The latter is a better approach from a security vulnerability perspective.

Whether you use Visual Studio or Mage, what you end up with is a specification inside your application manifest file that says to the runtime, "My application needs these permissions to run."

Configuring ClickOnce Security Permissions with Visual Studio

Figure 6.1 showed the Security tab of the project properties editor. This is where you configure the set of permissions that are placed in your application manifest at the time that you publish your application. Checking the Enable ClickOnce Security Settings checkbox makes the rest of the options available. This box will be checked automatically the first time you publish your application from Visual Studio.

Enabling ClickOnce security settings also affects the way your application runs in the debugger. Once enabled, each time you run your application, the selected security settings will be applied to the debug executable process, so your debug runtime environment will have the same security restrictions as your target environment. For example, if you select Local Intranet as the target zone for partial trust and make no modifications to the permissions list below the partial-trust selection, and then run a debug session and your code tries to do file I/O, you will get an exception in the

debugger because the process will run with only the permissions for the Local Intranet zone. This is extremely helpful in debugging and fixing problems that would otherwise only occur in the deployed environment.

If you select partial trust, you can then select the target zone as Local Intranet, Internet, or Custom. Selecting either Local Intranet or Internet selects the permissions in the ClickOnce Security Permissions table to match the target zone. Once those permissions are selected, you can then customize the settings to something different than the defaults for that zone as needed, using the zone permissions as a starting point for a custom set of permissions. Remember that ClickOnce ignores any custom security policy code groups, so setting fine custom permissions through the ClickOnce partial trust settings are the only way to explicitly grant specific permissions in a partial-trust scenario.

So, for example, if you were going to deploy an application to the local Intranet, but the application needed to call a Web service on your network other than the one where the application is being deployed from, you would do the following.

1. Check the *Enable ClickOnce Security Settings* checkbox in the project properties editor's Security tab (see Figure 6.4).
2. Select *This is a partial trust application*.
3. Select *Local Intranet* in the drop-down list labeled *Zone your application will be installed from*.
4. Scroll down in the grid of permissions required by the application to find the *WebPermission* type.
5. In the Setting column drop-down list, select *Include*.

After doing this, your application will request all of the permissions in the Local Intranet zone as well as the `WebPermission` permission with unrestricted access to the Web.

As mentioned, most permission types have a number of additional options that you can set to customize exactly what options in that permission type you need. `WebPermission` includes the ability to set a list of URLs that you will let your application either call out to or be called

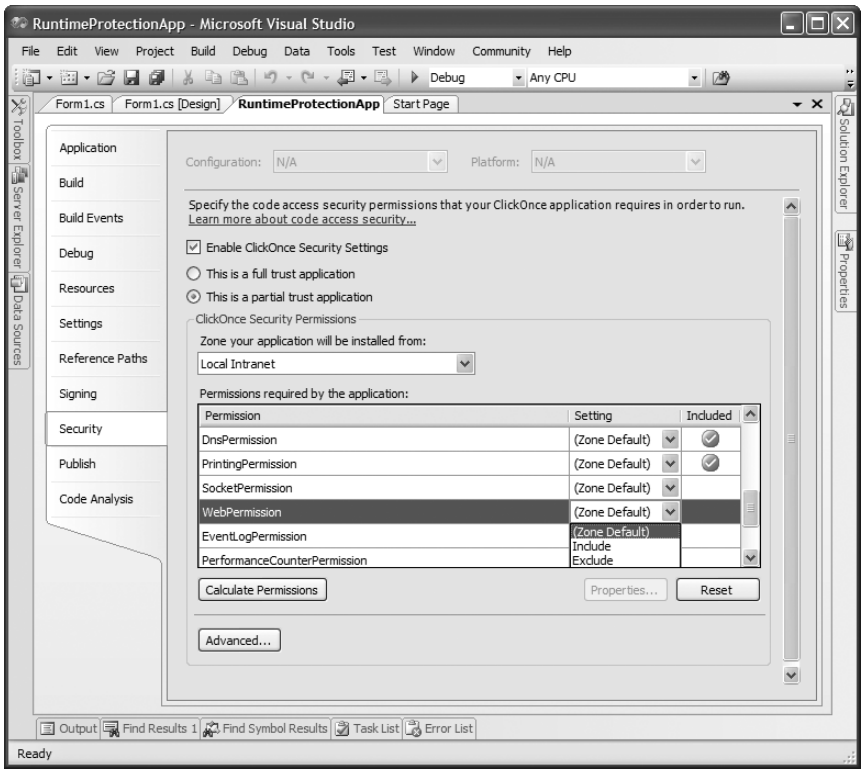

FIGURE 6.4: Adding Permissions to a Selected Zone

on. Unfortunately, the permissions editor in the Security tab for ClickOnce does not allow you to access all the options for all of the displayed permission types. `WebPermission` is an example. You can see in Figure 6.4 that the Properties button below the grid is disabled. If there are configurable properties for a selected permission type, this button will be enabled and will take you to a dialog that lets you edit the finer-grained options for that permission type. Figure 6.5 shows an example of one of these dialogs for the `SecurityPermission` type.

There are some permission types that are not shown in the grid at all. The only way to add these permission types is to either select full trust for the application (see Figure 6.4) or use the Mage tool to configure the permission type based on its XML declaration. See the next section for more information on how to do that.

FIGURE 6.5: Options in the Permissions Settings Dialog

If you do not want to run your debug sessions in the ClickOnce security zone selected, you can disable this capability by doing the following (see Figure 6.4).

1. Open the project properties editor for your application.
2. Select the Security tab.
3. Check the *Enable ClickOnce Security Settings* checkbox if it is not already checked.
4. Select *This is a partial trust application* if it is not already selected.
5. Click on the Advanced button at the bottom of the Security tab. This brings up the Advanced Security Settings dialog shown in Figure 6.6.
6. Uncheck the box labeled *Debug this application with the selected permission set*.

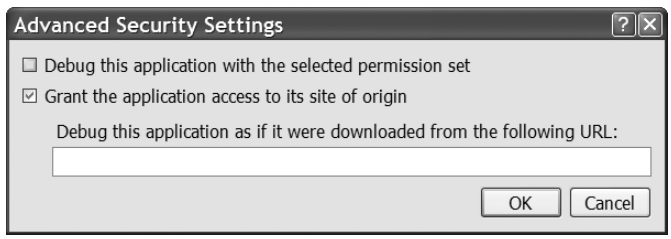

FIGURE 6.6: Advanced Security Settings Dialog

Notice that you also have the options in the Advanced Security Settings Dialog to do the following.

- Grant the application access to its site of origin (selected by default)
- Debug the application as if it were downloaded from a different URL

Granting the application access to its site of origin lets you expose a Web service from the same site that the application is launched from that the application calls for back-end services. You can also use this to download additional files on demand. By doing this, you do not need to ask for `WebPermission` specifically to make those calls. Debugging the application as if it were downloaded from a different URL lets you test and see what will happen with different security zones based on the URL—without needing to understand the exact logic that the runtime is using to evaluate the URL and match it against the location-based security zones.

The settings that you select in the Security tab are saved as part of your Visual Studio project file and will be used each time you publish your application from ClickOnce. The appropriate entries in the application manifest will be created when you publish.

Calculating Permissions with Visual Studio

At the bottom of the project properties editor's Security tab, there is a Calculate Permissions button (see Figure 6.4). If you click this button, Visual Studio will do a static analysis of your code, and every assembly that your code calls out to, in an attempt to determine what permissions your application will require to run. After you run the permissions calculator, it will configure the individual security permissions required for your application to include the permissions that it determined your application needs.

Using the permissions calculator is only appropriate if you plan to deploy your application under partial trust and are not sure what permissions your application requires based on its design. The thing to be aware of with the permissions calculator is that it makes a conservative estimate of what permissions your application will require. Based on my experience trying to use this tool, it always overestimates the permissions required by your application. In fact, it often grossly overestimates the permissions required.

As a result of this overestimation, you will be better off keeping track of what permissions your application needs based on its design and configuring only those permissions. Then test the application rigorously, running under the debugger with ClickOnce security enabled, to ensure you did not miss any required permissions.

If you use the permissions calculator to set the required permissions for your application, your application manifest will likely state that many more permissions are required than really are. This means you are removing some of the protections that running under partial trust brings you. However, running with a set of permissions determined by the permissions calculator under partial trust will still offer more protection to the client machine than running under full trust. So if you are unsure what permissions you need and don't want to jump the security requirements all the way to full trust as a result, go ahead and use the permissions calculator.

Configuring ClickOnce Security Permissions with Mage

It should be a fairly rare thing that you would change the permissions your application requests after you have published from Visual Studio. After all, the permissions required are determined by the code that executes, not based on administrative whims. However, if you find that you need to modify the set of permissions that your application requests without publishing a new version from Visual Studio, you have some ability to do so with the Mage tools.

Using the command line mage.exe tool, you can only set the security to one of the predefined zone levels of Internet, LocalIntranet, or FullTrust. You do this by running mage with a command line switch of `–TrustLevel` (or `–tr` for short). Because you are editing the manifest by doing this, you will also need to re-sign the manifest with a certificate. You do this with other command line options. The following example shows how to set the security zone to full trust and re-sign the manifest.

```
mage.exe -Update MyApp.exe.manifest -TrustLevel FullTrust -CertFile
MyCert.pfx -Password SomeSecretPwd
```

You can use the Mage UI tool, mageui.exe, to edit the permission settings in a dialog-based user interface. Start mageui.exe from a Visual Studio

command prompt and open the application manifest that you want to edit. Select the Permissions Required category in the list on the left and you will see the view shown in Figure 6.7.

You can drop down the list of permission set types on the right in Figure 6.7 to select one of the predefined zones, including Internet, Local Intranet, and Full Trust. When you select one of these values, the Details pane below it will display the XML `PermissionSet` element that will be placed in the application manifest. Under this element, there will be individual `IPermission` elements for each permission you require, except in the case of Full Trust, which just sets the permission set to unrestricted. You can also select a value of Custom in the permission set type list to manually enter whatever settings you would like. This requires that you understand the full schema of the `PermissionSet` element and its child elements to determine what to put into the Details box. This schema is beyond the scope of this book to describe in detail, but follows a common convention with the way permission sets are defined for security configuration files. Consult the MSDN Library documentation for more information on manually creating XML permission set entries.

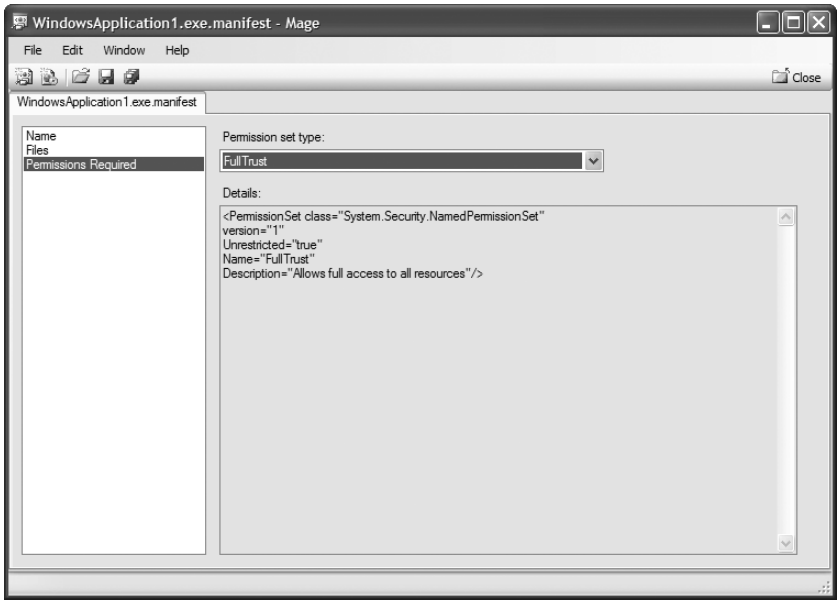

FIGURE 6.7: Setting Permissions with Mage UI

Note that using the Custom permission set type and figuring out the right XML elements and attributes to add is the only way to go beyond the permission sets and options that are exposed to you in Visual Studio. You could also write a custom tool to set these through the APIs exposed in the `Microsoft.Build.Tasks.Deployment` namespace, but that is not a trivial task either.

Understanding and Managing Publisher Certificates

The publisher certificates used to sign ClickOnce manifests are Authenticode Class 3 Code Signing certificates. This is just one form of an Authenticode certificate.[2] There are many different kinds for various forms of authentication and authorization security scenarios.

Publisher certificates are generated with a public-private key pair and additional metadata about the publisher. The organization that creates a certificate is called the **certificate authority** or **certificate issuer.** The organization the certificate represents is the publisher. The Windows operating system has a built-in infrastructure for storing and authenticating certificates. There are a number of built-in certificate stores in the operating system, and you can create additional custom stores as needed.

Certificates are based on the concept of a trust chain. If you are presented with a certificate, you can determine from the certificate who the publisher organization is that the certificate represents, as well as who issued that publisher the certificate. From the issuer's certificate, you can determine the issuer's identity, as well as who issued the issuer their certificate. You can follow this chain of issuers back to what is called a **Trusted Root Certification Authority.** This chain of issuers provides a path of discovery that ensures that if you can verify the identity of all of the issuers in the chain, you have a way to track down and contact the publisher.

This way, if you deploy an application to your machine that is signed with a publisher certificate, and that application does harmful things to your machine, you can track down the publisher through the information in the certificate, or through the information that is retained by the issuer

2. For an overview of Authenticode code signing, see http://msdn.microsoft.com/workshop/security/authcode/intro_authenticode.asp.

when issuing the certificate. Some certificate issuers include liability insurance as part of their certificate issuance services; this guarantees that if you cannot contact a publisher that was issued a certificate by that authority (to pursue a liability claim), the certificate issuer will assume the liability up to some limited degree.

To support this concept, a number of companies are in the business of verifying the identity of other organizations for the purposes of issuing certificates to them. VeriSign and thawte are two well-known companies who perform these services. The issued certificate (whether a code signing or publisher certificate, or one of the many other forms of certificates) becomes a digital representation of the organization's identity. Certificates from well-known and trusted certificate authorities are installed with the operating system or can be added later, which identifies them as a trusted issuer of other certificates. As a result, if you obtain a publisher certificate from an application vendor, and that certificate has been issued by an organization like VeriSign, you can be relatively certain that the company is who they say they are (they are a legal business entity), and that the organizational information contained in the certificate has been verified by the issuer (which includes the location of the organization or where its business license information can be verified).

The verification chain may be deeper than one level, however. A Trusted Root Certification Authority can issue certificates to themselves or other certification authorities to issue specific kinds of certificates. For example, see Figure 6.8 for the trust chain for a code-signing certificate for my company, Software Insight. You will see that the root VeriSign Class 3 Public Primary CA (certificate authority) certificate was used to issue a VeriSign Class 3 Code Signing CA certificate, which was then used to issue my Software Insight Class 3 Code Signing publisher certificate. To issue that certificate, VeriSign had to verify my existence as a legal business entity. They can do this through articles of incorporation or by verifying that a legal business license has been issued by your state or city, for example.

You do not have to purchase a certificate from a third-party certificate issuer to use ClickOnce. In an enterprise environment, your domain administrators can generate a certificate for themselves and configure that

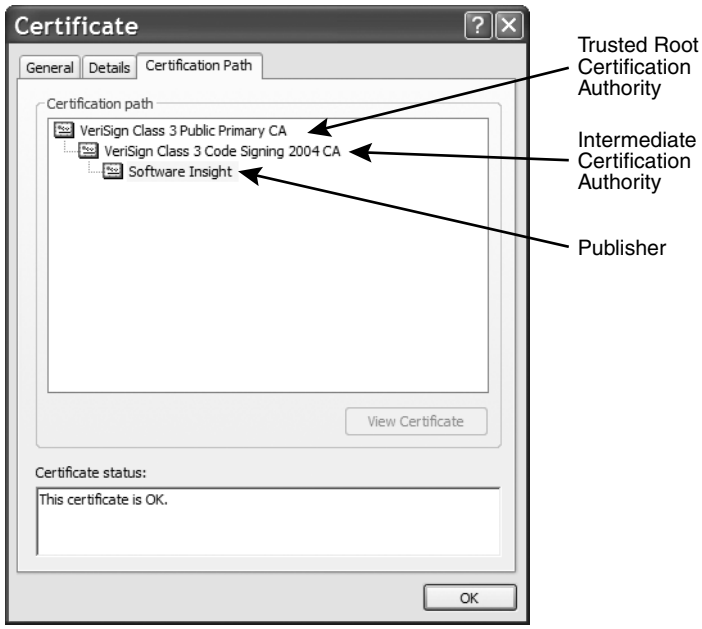

Figure 6.8: Certification Path

certificate as a Trusted Root Certification Authority (CA) on all the machines in the enterprise, allowing them to issue publisher certificates to your development organization with a single-level trust chain back to a known CA. Or if you are not concerned with providing any kind of assurances of identity with your ClickOnce publication, you can generate your own publisher certificate with either Visual Studio 2005 or with command line tools.

To make matters even more complicated, there are a number of different file formats that are used for delivering certificates. Third-party certificate issuers usually issue a certificate in the form of a .cer or .spc file. These certificate files usually only contain the public key portion of the certificate, so you can freely distribute them to client machines and install them in those machine's certificate stores. When you purchase a certificate, you also usually receive a separate .pvk file that contains the private key corresponding to that public key. You will need both the public and private key portions of a certificate, in a single .pfx file format, to use it for ClickOnce publishing. You can combine .cer or .spc file portions with

the .pvk portion by using the pvkimprt.exe tool that is available from Microsoft downloads.[3]

There are several certificate stores on your Windows machines that you will use with ClickOnce deployment. Any certificate you use for ClickOnce publishing will be added to the Personal certificate store for the logged-in user when you publish the application. Additionally, if you want to avoid user prompting on the client machine, you will want to install your publisher certificate into the Trusted Publishers store on the client machine as discussed in the section Trusted Publishers' Permission Elevation later in this chapter. If you are installing a publisher certificate into the Trusted Publishers store, you will want to make sure the certificate's issuer is in the Trusted Root Certification Authorities store or the Intermediate Certification Authorities store, and that the root issuer of the trust chain is in the Trusted Root Certification Authorities store (see Figure 6.8).

Generating or Configuring a Publisher Certificate with Visual Studio 2005

If you publish a Windows Application project with Visual Studio without configuring a publisher certificate ahead of time, Visual Studio will generate a self-signed publisher certificate for you. In this kind of certificate, the identity of the issuer and the publisher are set to the logged-in Windows identity of the user.

The public and private key portions of the certificate are placed in a file with a .pfx file extension and the file is added to your project. The certificate is then configured as the signing certificate for ClickOnce publication, and is also added to your Personal certificate store on the development machine. When your application is published, the deployment and application manifest files are signed with this certificate. The .pfx file that is generated is a password-protected file, but when Visual Studio automatically generates the file for you the first time you publish, the password of the generated file is set to an empty password.

You can generate your own certificates through Visual Studio (with the option to password-protect the file), or you can select an existing certificate

3. Search for *pvkimprt* at www.microsoft.com/downloads.

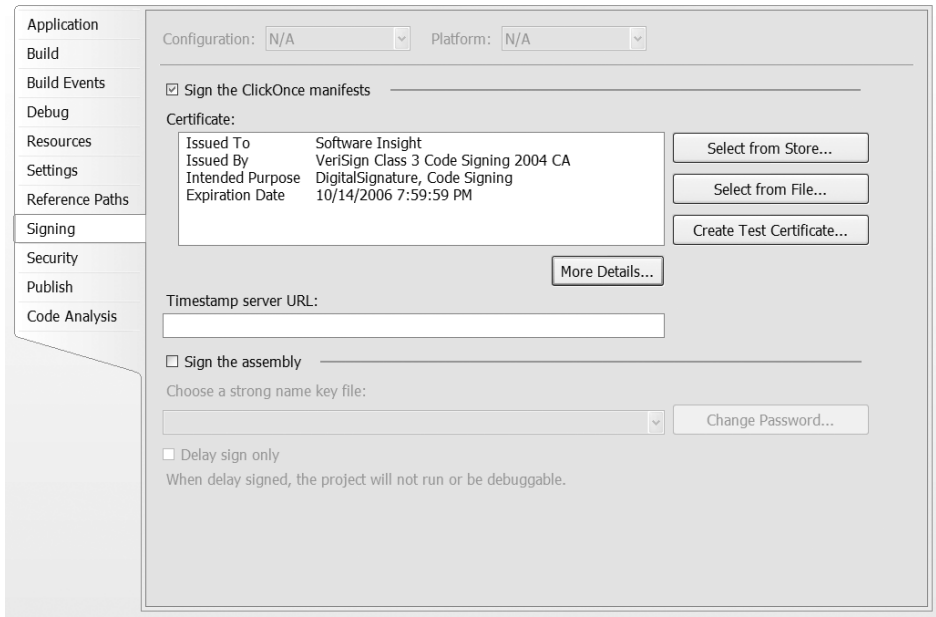

FIGURE 6.9: ClickOnce Signing Settings

to use for signing as well. You do this through the Signing tab of your project properties editor (see Figure 6.9).

After checking the box that is labeled *Sign the ClickOnce manifests*, you can either select a certificate from the logged-in user's Personal certificate store on the development machine, from a .pfx file, or generate a new certificate. If you click the Select from Store button, you will see the dialog shown in Figure 6.10 to select a certificate.

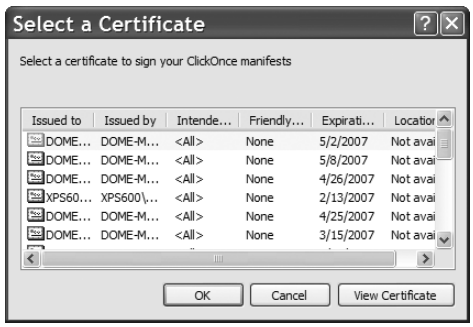

FIGURE 6.10: Select a Certificate Dialog

You can see that there are several small challenges to using the Select from Store option. The first is that if you test publishing an application with ClickOnce without first configuring the Signing tab to use an existing certificate, a new certificate is generated each time. Each of those certificates have a different public-private key pair and are distinct certificates, but they all have the same common name, known as *CN* for short, which will be your logged-in Windows account name (e.g., DOME-M200\Brian Noyes on my current machine). As a result, it is almost impossible to tell which one is which. The other challenge is that this dialog will not let you resize it, and you can see that there are a lot of columns, each with long content, so the usability of the dialog is extremely low.

An alternative to selecting a certificate from the Personal certificate store is to just point to an existing .pfx file for a publisher certificate. This will extract the information in the certificate and use it for signing, as well as install it in the Personal certificate store if it is not already there. You can see an example of this in Figure 6.10 as well—the entry that starts with *XPS600* is from a certificate generated on a different machine of mine (named XPS600), and was automatically imported into my Personal certificate store on the current machine when I selected that .pfx file for my certificate. Clicking the Select from File button on the Signing tab gives you a standard file dialog to navigate to the location of your certificate file.

If you click the Create Test Certificate button on the Signing tab, you will be prompted for a password as shown in Figure 6.11. The dialog does not enforce strong passwords; you can leave it blank if desired.

After you click OK in the Create Test Certificate dialog, the process is similar to what Visual Studio does if you do not configure a certificate and publish the application.

- A test certificate is generated with the issuer and publisher (labeled *Issued By:* and *Issued To:*, respectively, in most places in the UI) set to your logged-in Windows identity.
- The certificate is placed in a .pfx file with the name <appname>_TemporaryKey.pfx with the password you provided set on the file.
- The .pfx file is added to the Visual Studio project files.
- The certificate is imported into your Personal certificate store.

WARNING Selecting a Certificate File Copies It to Your Project Folder

One important thing to be aware of is that if you point to a certificate file by using the Select from File option on the project properties editor's Signing tab, the file you point to will be copied to your project folder and added to your project. A .pfx file for a publisher certificate includes both public and private keys. If you subsequently zip up your project and send it off to a friend, you will have compromised that certificate because now someone else has physical access to the certificate's private key. If the file is password protected, as it should be, that person will have a very difficult time using the certificate, but the possibility now exists. If it is a self-generated test certificate, then it is no big deal. But if it is a real company certificate, such as one purchased from VeriSign, then you should revoke that certificate. Unfortunately, I mention this from the vantage point of one who has done exactly this. I zipped up the code from a demo I did using my VeriSign company certificate and forgot to remove my certificate file from the project directory first.

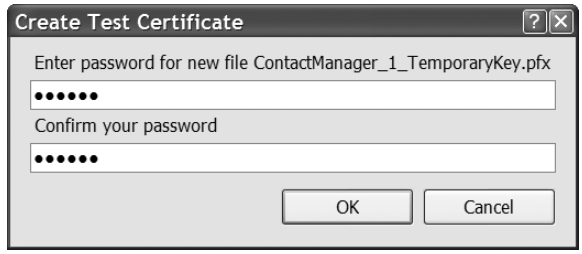

FIGURE 6.11: Create Test Certificate Dialog

Once you have configured a certificate through the Signing tab, that certificate will be used for any subsequent publications of your application to sign the ClickOnce manifests.

Installing a Certificate in a Store with Visual Studio 2005

Visual Studio lets you install a certificate into any certificate stores on your machine if desired. As described earlier, any certificate that you configure to sign your ClickOnce manifests by generating the file or selecting a file will be installed into your Personal certificate store on your development machine. Additionally, if the certificate is password protected, then you

can use Visual Studio to manually install that certificate into other stores on your machine.

Do the following if you want to install a signing certificate into a different store on your machine.

1. Click on the More Details button on the Signing tab (see Figure 6.9). This will display the certificate information dialog shown in Figure 6.12.
2. At the bottom of the dialog, click the Install Certificate button.
3. This will bring up the Certificate Import wizard shown in Figure 6.13. (This same wizard can be accessed by running certmgr.exe and clicking the Import button in that tool). Click Next to start the process.
4. The second step in the wizard (see Figure 6.14) lets you select which store the certificate will be placed into. Select the radio button labeled *Place all certificates in the following store* and click the Browse button.
5. The Select Certificate Store dialog shown in Figure 6.15 will display, and you can pick from the available stores on the machine. Select the store you want to place the certificate into, such as the Trusted Publishers store, and click the OK button.

FIGURE 6.12: Certificate Information Dialog

FIGURE 6.13: Certificate Import Wizard

FIGURE 6.14: Certificate Import Wizard – Store Selection

FIGURE 6.15: Select Certificate Store Dialog

6. You will return to the wizard and the selected store will be displayed in the Certificate store field in the middle of the form (see Figure 6.16). Click the Next button to continue.
7. The final step of the wizard (see Figure 6.17) will display, summarizing the import that is about to happen. Click the Finish button to complete the installation of the certificate into the selected store.

FIGURE 6.16: Completed Store Selection Step

FIGURE 6.17: Certificate Import Wizard Completion

Command Line Certificate Tools

There are several command line tools that come with the .NET Framework SDK or that you can download to assist you in generating, configuring, and managing publisher certificates. To generate a test certificate from the command line, you can use the makecert.exe command line tool. This tool offers more fine-grained options for generating publisher certificates. Run `makecert.exe` from a command line with the `-?` switch for a brief summary of options or with the `-!` command line switch for more detailed options. The makecert.exe tool uses the CryptoAPI under the covers and is available in the .NET Framework SDK binaries (\Bin) folder underneath your Visual Studio 2005 installation (C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin path with a default installation).

To configure certificates with respect to the machine certificate stores, you can use the certmgr.exe tool. If you run certmgr.exe without any arguments, it launches a UI version of the tool as shown in Figure 6.18.

This tool provides a graphical management console for importing, exporting, and removing certificates from the named stores on your development machine. Clicking the Import button shown in Figure 6.18 launches the same wizard discussed in the previous section for installing certificates in stores.

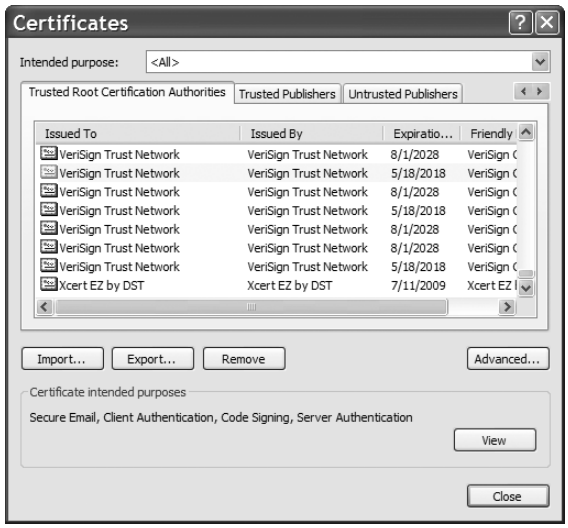

FIGURE 6.18: Certificate Manager Tool

You can also incorporate certmgr.exe in a Windows Installer installation package and use it to configure certificates on a client machine as well using command line options. For example, the following command line will install a certificate in the Trusted Publishers store on a target machine if the certmgr.exe tool is available in the command prompt PATH environment variable.

```
certmgr.exe -add MyCompany.cer -s TrustedPublisher
```

Another command line tool to be aware of that was mentioned earlier is the pvkimprt.exe tool. This tool is available for download from Microsoft (www.microsoft.com/downloads) or through the Platform SDK. Pvkimprt.exe lets you take a .cer or .spc file that just contains the public key portion of a publisher certificate, combine it with a .pvk file that contains the private key portion of the certificate, and generate a .pfx password-protected certificate file that contains the entire certificate. To do this, you run the tool with a `–pfx` switch, also passing the .spc or .cer file path and the .pvk file path. This will bring up a wizard that will step you through the process of providing a password and then exporting the keys to a .pfx file.

Signing Application Updates

An important security limitation to understand with ClickOnce is that you cannot deploy updates to an application that are signed by a different publisher certificate than the one that was originally used to sign previous deployments of the application. If ClickOnce sees that an update is available, but that the update is signed by a different publisher certificate than the one used to sign the version currently installed on the client machine, ClickOnce will disable the application and present the message box shown in Figure 6.19.

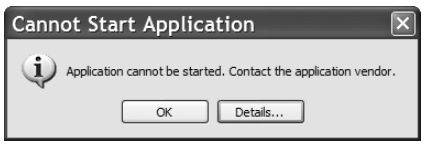

FIGURE 6.19: Changed Publisher Certificate Message

If you click the Details buttons to see the error, it will specify that the problem is, "The deployment identity does not match the subscription." This behavior was designed to protect the tamper assurances that were discussed earlier in the section ClickOnce Tamper Protection. If someone were to gain access to your published application directory on the server, he could introduce a virus or malware into one of the application files, and then just regenerate and re-sign the manifests with his own certificate. Because updates do not typically prompt users (unless elevating the permissions of the application beyond their current trust level), there would be no way for users to realize that they could be introducing the compromised application onto their machines. Disabling the application if the certificate has changed is designed to make it as safe as possible and not allow updates to be signed with a different publisher certificate from the original certificate.

As a result, if you do publish an update to an application from Visual Studio 2005 that is signed with a different certificate than the previous version, you will be prompted as shown in Figure 6.20 to ensure this is your intent.

If you do have a previous version of a ClickOnce application that users have installed and you need to put out a new version signed by a different certificate (perhaps your company changed names or was acquired by a new parent company), you will need to instruct users to launch the new version using the full URL to the deployment manifest on the server. This will force a fresh install of the application, and the new install will not be related to the old version at all. Before users do this, it is recommended that you have them uninstall the previous version first from the Add or Remove Programs item. If they do not, they will end up with two application installations. The old one will have the original program name (e.g., My Application), and the new one will have the old name with a – 1 appended to it (e.g., My Application – 1). This is likely to confuse users when they go to

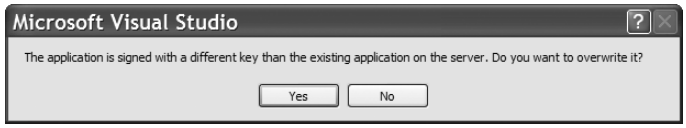

FIGURE 6.20: Visual Studio Publisher Certificate Change Warning

launch the application from the Start menu. Unfortunately, there is no way to automate this process. For online-only applications this is not a problem because if the certificate has been changed, it will just be treated as a new application being launched, and users will be prompted with the security warning again.

User Prompting

Despite the name *ClickOnce*, often users will need to click twice to get the application deployed and running. The first click is the one that starts the process of deployment and launching of the application. Users click on a URL provided in an e-mail or on a Web site to start the deployment process. If the application is configured to run offline, then users will also be prompted because the application will create a Start menu item and an Add or Remove Programs entry, which causes visible changes on their machines. As a result, users are notified of this change before it occurs and has the opportunity to refuse the application. Additionally, if the application requires elevated permissions to run, users will also be prompted to decide whether they should allow the application's elevated permissions on their machine. The kind of prompt presented to users in both of these cases also gives an indication of whether the publisher is verifiable through a trust chain on the machine.

Table 6.1 summarizes the prompting dialogs users will see and the associated risk levels, and Figures 6.21 through 6.26 show the range of prompting dialogs that users will see, starting from lowest risk to highest risk to the local machine. For any of these prompts, if users click the Install button, the installation will complete and the application will launch. If users click the Don't Install button, no modifications to the client machine will be made. The different levels of prompting are just intended to convey different degrees of risk to users based on how well known the publisher is and whether the application requires elevated permissions on the local machine.

Once users have been prompted and they click the Install button, they will not be prompted again for running that application, even if updates are installed, unless an update requests higher permissions than the currently installed version. In that case, the prompting will follow the same

TABLE 6.1: Security Dialog Risk Levels

Risk Level	Icon	Cause
Low	Green check mark	Known publisher, no security permission elevation, only adding Start menu shortcut and Add or Remove Programs item.
Medium	Yellow exclamation point	Known publisher, security permissions elevation needed to run, may also be adding Start menu shortcut and Add or Remove Programs item.
Medium	Yellow exclamation point	Unknown publisher, no security permission elevation, but adding Start menu shortcut and Add or Remove Programs item.
High	Red X	Unknown publisher, security permission elevation needed to run, may also be adding Start menu shortcut and Add or Remove Programs item.

logic as an initial install in determining which prompts to provide. This is true for both installed and online-only applications. The following subsections describe those different prompts.

Low-Risk Install

Figure 6.21 shows the install prompt users will see if an application is being launched for the first time where:

- The application is an installed application
- The application is signed by a publisher that was issued its certificate by a Trusted Root Certification Authority known by the client machine
- And the application is not requesting any permissions beyond what it would be granted by default by CAS based on its launch URL

Figure 6.22 shows the More Information dialog for this deployment scenario. You can see that the only thing the dialog is really cautioning users about is that it will add a Start menu item and an Add or Remove Programs item.

FIGURE 6.21: Low-Risk Install Prompt

FIGURE 6.22: Low-Risk Install More Information Dialog

Medium-Risk Install

Figure 6.23 shows the install prompt users will see if an application is being launched for the first time where:

- The application is an installed application
- The application is signed by a publisher that was issued its certificate by a Trusted Root Certification Authority known by the client machine
- And the application is requesting elevated permissions beyond what it would be granted by default by CAS based on its launch URL

Figure 6.24 shows the More Information dialog for this deployment scenario. You can see that in this case users are being warned that the application requires access to "additional resources on your computer," meaning elevated permissions. It also adds the normal caution that it will

FIGURE 6.23: Medium-Risk Install Prompt

FIGURE 6.24: Medium-Risk Install More Information Dialog

add a Start menu item and an Add or Remove Programs item. However, you can see that this dialog makes it clear that the publisher is considered to be a known entity since its certificate was issued by a known certificate authority (CA).

High-Risk Install

Figure 6.25 shows the install prompt users will see if an application is being launched for the first time where:

- The application is an installed application
- The application is signed by a publisher that is unknown (meaning its certificate was issued by an unknown certificate authority)
- And the application is requesting elevated permissions beyond what it would be granted by default by CAS based on its launch URL

FIGURE 6.25: High-Risk Install Prompt

FIGURE 6.26: High-Risk Install More Information Dialog

Figure 6.26 shows the More Information dialog for this deployment scenario. You can see that in this case users are being warned that the publisher of the application is unknown, and the application requires access to "additional resources on your computer," meaning elevated permissions. It also adds the normal caution that it will add a Start menu item and an Add or Remove Programs item.

The high-risk prompts shown in Figures 6.25 and 6.26 are what users will see if you deploy a ClickOnce application using a self-generated test certificate (created with Visual Studio or the makecert.exe tool).

User Prompting for Online-Only Applications

When users click on a link to an online-only ClickOnce application, they will only be prompted if the application needs to elevate permissions. If

FIGURE 6.27: Online-Only High-Risk Install More Information Dialog

the application does not need to elevate permissions, users will not be prompted at all after they click on the link to the application, even if the publisher is unknown. The application will just download and launch.

If the application does need to elevate permissions, then users will be prompted with a dialog similar to either Figure 6.23 or 6.25, depending on whether the publisher is known (certificate issued by a trusted root CA) or unknown. The only difference in the prompting dialogs in this case is that the buttons will be labeled *Run* and *Don't Run* for the online-only application instead of *Install* and *Don't Install* for the installed application. If users inspect the More Information dialog, they will see the green status for installation, indicating that no modifications to their Start menu or Add or Remove Programs items will be made (see Figure 6.27).

Trusted Applications' User Security Policies

When an application gets installed or run, a user security policy is created to record the set of permissions that have been granted to that application. This policy can be viewed using the Microsoft .NET Framework 2.0 Configuration tool. If you open this tool (from the Administrative Tools menu) and expand the Runtime Security Policy node down to the user level, you will see a child node under User for Trusted Applications. If you select this and click the link in the right pane labeled *View List of trusted applications*, you will see something like Figure 6.28.

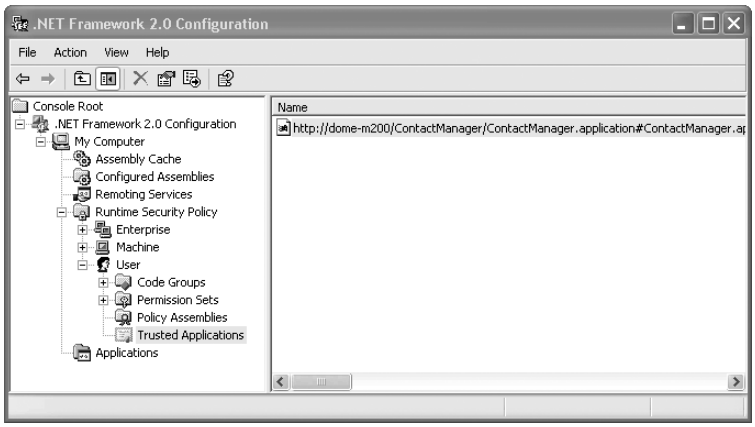

FIGURE 6.28: Trusted Applications' Security Policies

Depending on how many different ClickOnce applications and how many different versions of those applications have been deployed to your machine, you may see many more entries in the list on the right. You will get one entry in the list for each application version for which a different set of permissions were issued. This always includes the first install or run of a ClickOnce application, and then additional entries will be made for subsequent versions of an application only if they elevate permissions beyond what the previous version required.

You can inspect the permissions set for a given application version by double-clicking on the entry in the list. This displays the Properties dialog shown in Figure 6.29, where you can inspect and browse through the assigned permissions.

Trusted Publishers' Permission Elevation

The problem with the default security model for an enterprise environment is that it puts the trust decision of whether to elevate permissions or not into the users' hands. If an application needs elevated permissions, it prompts the users, and if they click the Install button, the application can elevate its permissions all the way to full trust if it wants to, effectively removing the runtime protections that ClickOnce is capable of providing.

This is often not what the IT administrators for the enterprise want—they want to have explicit control over the machines that they are responsible for.

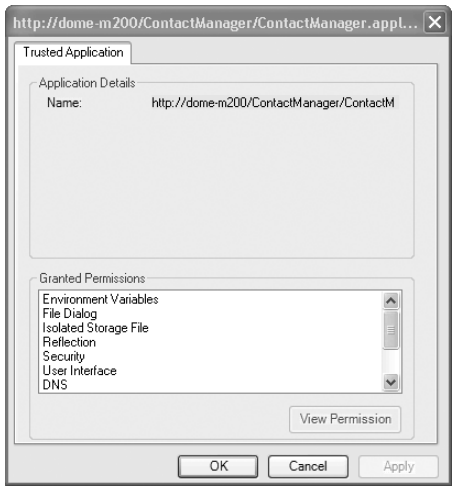

FIGURE 6.29: Trusted Application Permissions

Many users do not have the experience to discern a true high-risk scenario from one that is acceptable. From an IT administrator's perspective, an application should not run on a user's desktop unless the administrator has configured it to do so, either directly or implicitly through a trust relationship with the publisher of the application.

There is also a downside to the default prompting model from the users' perspective as well. If they are going to be launching various applications from a known and trusted publisher, why should they be repeatedly prompted when those applications launch the first time? It would be good to let end users establish a trust relationship with a publisher as well to avoid unnecessary prompting.

ClickOnce supports giving the administrators more explicit control and avoiding unnecessary prompting through a model known as **trusted publishers.** A trusted publisher is a publisher whose certificate has been installed in the Trusted Publishers certificate store on a client machine. This can be done manually using the techniques discussed early in the chapter in the section Understanding and Managing Publisher Certificates, or it can be done automatically through something like Microsoft Systems Management Server or group policy.

If an application is deployed through ClickOnce to the client machine and that application's manifest has been signed by a trusted publisher's

certificate, then the runtime can use that to automatically elevate the permissions for the application rather than prompting the user. As a result, the prompting model is actually even more complex than the different levels of prompting discussed in the User Prompting section.

When the ClickOnce runtime determines that a user prompt is required to elevate permissions, it will also check to determine if it is allowed to prompt the user. The settings that drive this decision are based on the zone that the application is being launched from. For each launch zone (My Computer, Local Intranet, Internet, Trusted Sites, Restricted Sites), there is a default setting that says when the runtime is allowed to prompt the user. This setting can take on one of the following values.

- **Enabled**—The runtime can prompt users if needed to elevate the permissions for the application. However, if the application's publisher is a trusted publisher, then the application will automatically elevate its permissions and users will not be prompted to install the application (i.e., it will be a true Click*Once* application). This is the default value for the My Computer, Local Intranet, Trusted Sites, and Internet zones.
- **Authenticode Required**—The runtime can prompt users if needed to elevate the application's permissions only if the publisher certificate for the application has a trust chain back to a Trusted Root Certification Authority. If the publisher is unknown, the application will be disabled. If the publisher is a trusted publisher, users will not be prompted to install the application and the permissions will be automatically elevated.
- **Disabled**—The application can only run if signed by a trusted publisher, in which case the permissions will automatically elevate. With this setting, users will never be prompted and only applications from trusted publishers will be allowed to launch through ClickOnce. This is the default value for the Restricted Sites zone.

If you want to change the default prompting behavior for a given zone, you will have to add a registry key with values set for the zones for which you want to change the defaults. You will need to create a TrustManager key

under the HKLM\Software\Microsoft\.NETFramework\Security key, and another key under that named PromptingLevel. Once you have created those keys, you add named string values under the PromptingLevel key for each zone that you want to modify. The name of the key should match the zone name: MyComputer, LocalIntranet, Internet, TrustedSites, or RestrictedSites. The value for the string value should be set to one of the three levels discussed earlier: Enabled, Authenticode Required, or Disabled.

Using these registry settings, you can achieve a much more secure configuration for client machines in an enterprise environment. If you add the named values just described and set them all to Disabled, it means that the only ClickOnce applications that are allowed to run on users' machines are those that are signed by publisher certificates that have been installed in the Trusted Publishers store on those machines. In other words, the only ClickOnce applications that get to run are those for which the administrator established a trust relationship on the client machine with the publisher of the application. When that is the case, the application will download, automatically elevate its permissions to the level needed, and execute. Any application not coming from a trusted publisher will not be allowed to launch through ClickOnce. Figure 6.30 depicts this configuration in the registry.

Adding Restricted Code Sections

The set of permissions an application has is determined by the permissions its application manifest says it requires to run and the permissions that would be granted to it based on the zone it is launching from. However, if

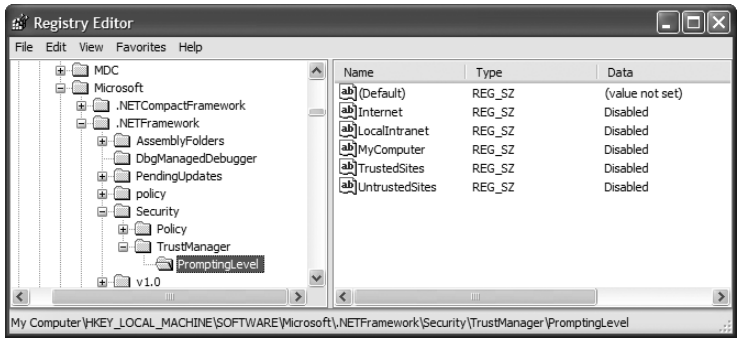

Figure 6.30: User Prompting Registry Settings

you need to elevate permissions for certain parts of your application, you may not want to leave your entire application vulnerable because of those elevated permissions. For example, you may have a very limited scope to perform on-demand updates from your application. As discussed in Chapter 4, on-demand updates require full trust. As a result, you will need to configure your application for full-trust permissions as far as ClickOnce deployment is concerned. However, there may be other portions of your code where you are doing things such as calling out to third-party components, and you want to make sure that they cannot perform any operations or access any resources other than what they are supposed to based on their design purpose.

For example, say for a limited scope you were going to call out to a component that is supposed to perform certain kinds of computation for you, and perhaps it may also present a user interface in doing so or to present error messages to users. However, by design, the component should never do anything other than execute and present user interfaces.

You can use the capabilities of Code Access Security to lock down the security context for a given scope of code. Listing 6.1 shows a section of code where a component is going to be used for a limited scope of code, and you want to ensure that the only thing that component does is to present a user interface to users. To do this, you can use a stack walk modifier, based on the `IStackWalk` interface, to turn on only a particular permission type.

Listing 6.1: Restricting Permissions for a Code Scope

```
private void OnCallUntrustedCode(object sender, EventArgs e)
{
   IStackWalk stackWalk =
      new UIPermission(PermissionState.Unrestricted);
   stackWalk.PermitOnly();
   UIOnlyComponent uoc = new UIOnlyComponent();
   uoc.DoUIStuff();
   CodeAccessPermission.RevertPermitOnly();
}
```

With the PermitOnly stack walk modifier in place, any code that is called out to in that scope will be restricted based on the specifications that the stack walk modifier puts in place. You can use stack walk modifiers to permit only

a specific permission or set of permissions as shown in Listing 6.1, deny a specific permission or set of permissions, or prevent checking for specific permissions beyond the local scope of code. The latter capability is called a *security assertion* and requires elevated permissions to be performed.[4]

Securing the Application Based on User Roles

A ClickOnce deployed application is no different than any other .NET application when it comes to role-based security. If you want to control what users can do with your application based on the roles they are associated with, you can use .NET principal permission demands. .NET principals can be based on a user's logged-in identity, or they can be based on custom authentication credentials and roles.

You can use role-based security in .NET to make security demands through either attributes applied to methods, classes, or assemblies, or you can make them imperatively with code. The following code shows an example of using principal-based security demands to ensure that only users in an appropriate role get to execute certain sections of code.

```
[PrincipalPermission(SecurityAction.Demand, Role = @"Managers")]
private void DoSomethingPrivileged()
{
   // Do manager stuff
   if (Thread.CurrentPrincipal.IsInRole(@"BUILTIN\Administrators"))
   {
      // Do admin stuff
   }
}
```

In the preceding code, only users who are associated with the Managers role will be allowed to execute the `DoSomethingPrivileged` method. The inline check using the `CurrentPrincipal` property on the thread lets you check whether the user is in a particular role and execute conditional code based on that.[5]

---

4. For more information on Code Access Security, stack walk modifiers, and permissions control, see *Programming .NET Components,* Second Edition, by Juval Löwy (O'Reilly & Associates, 2005).
5. For more information on role-based security in .NET applications, see *Programming .NET Components,* Second Edition, by Juval Löwy (O'Reilly & Associates, 2005).

Securing Access to ClickOnce Application Files on the Server

ClickOnce does not include any built-in provisions for restricting access to the application files on the deployment server. If you are using ClickOnce to deploy an application to the local Intranet, and the Windows identity of the logged-in user can propagate to the deployment server via Windows networking (i.e., no firewalls between the client and the deployment server), then you can use Windows Access Control Lists (ACLs) to restrict access at the file or folder level on the deployment server to specific users or groups. If you do this and a user attempts to launch an application to which she has not been granted file access rights, one of two things will happen.

If the user has been denied access to the .application file (the deployment manifest) and she tries to launch a ClickOnce application through a link or URL, she will see an HTTP 401 error (access denied) in the browser. If the user is allowed to get to the deployment manifest, but is denied access to the application manifest or any of the application files, she will get a launch error dialog like the one shown in Figure 6.31. The details under this dialog will specify that there was an HTTP 401 access denied error.

If you have an application that you are deploying over the open Internet or a network where you cannot rely on the logged-in identity of the user to get passed to the deployment server, then there is no practical way to secure access to the server files. When you deploy an application with ClickOnce, it is a set of individual file requests from the client machine to the deployment server. There are actually two for the deployment manifest (due to a level of indirection supported by the runtime where one deployment manifest can point to another), one for the application manifest, and then one for each application file. These file requests are not correlated in any way.

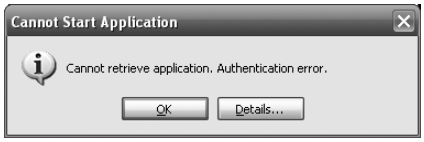

FIGURE 6.31: Authentication Error Launch Dialog

You can prevent users from being able to do a normal ClickOnce application launch by using an online-only application that you have to launch from a Web application that requires a login to access, but you can only restrict access to the .application file in that case. The request for the .application file will be made by the browser, but the subsequent file requests are made by the ClickOnce runtime and will not contain any cookies or headers from the previous file requests.[6] Each file request is completely isolated from the others.

Where Are We?

This chapter discussed the complex security mechanisms and options for ClickOnce deployment. You learned about the way ClickOnce applications identify the set of permissions that they require to run, how the .NET runtime determines what permissions they would be granted based on their launch URL, and how permissions can be elevated through user prompting or trusted publishers. You learned about certificates and their role in protecting application files from tampering and in determining trust relationships with application publishers.

The following are some of the key takeaways from this chapter.

- ClickOnce provides both install-time and runtime protections to the client machine.
- ClickOnce checks to see if an application requires more permission than it would be granted by default based on its launch URL. If so, it will elevate permissions based on either user prompting (the default) or trusted publishers.
- Manifests are signed by Class 3 Code Signing Authenticode certificates. Configuring a certificate as a trusted publisher on a client machine will avoid user prompting for most deployment zones.

---

6. For more information on possible strategies to secure deployment server files and to track application usage, see the "Administering ClickOnce Deployments" whitepaper by Brian Noyes at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwinforms/html/admincodep.asp.

- You can restrict access to deployment server application files based on Access Control Lists if all users will come from the local network and their logged-in user identities can be determined based on the individual file requests.

In the next chapter, we will look at how to configure and deploy prerequisites for your ClickOnce applications. Prerequisites include setup steps that require administrative privileges you do not want to require your users to have.