Auto-Deploying WinForms .NET Applications

...the revenge of the fat client

Hector J. Correa
hector@hectorcorrea.com
www.hectorcorrea.com

Overview

.NET provides developers with new tools to make deployment of fat client .NET applications much easier than typical fat client applications. .NET Automatic-Deployment technology allows posting executables and DLLs to a web server and let the .NET application automatically download these files as users use the application. .NET is smart enough to cache (on the user's machine) the downloaded DLLs and automatically re-download them as soon as a new version is posted on the web server.

This document describes the basics of .NET Auto-Deployment technology and the security mechanism that .NET has to prevent users from *inadvertently* running code distributed by hackers and virus writers and only run code that has explicitly been trusted.

What is Automatic Deployment?

.NET Automatic Deployment is a feature built-in into the .NET Framework that allows applications to download (via HTTP) assemblies from remote locations on an as needed basis.

With this feature you can just post your fat client application to a web server and be sure that users will automatically be using this latest version next time they run the application. Imagine that, no more CDs to ship with updates and no more support calls from users that are a few versions behind.

The .NET Framework provides out-of-the-box a lot of the functionality required for automatic deployment. For example, .NET knows how to download an assembly from a remote location, caches it in the user's disk, and only download it again when a new version is available in the remote location. All these features come built-in into the .NET framework.

Automatic Deployment is one of those technologies that has many different names. You will find references to this technology under Automatic Code Distribution, No-Touch Deployment and Zero-Touch Deployment. They all refer to the same technology.

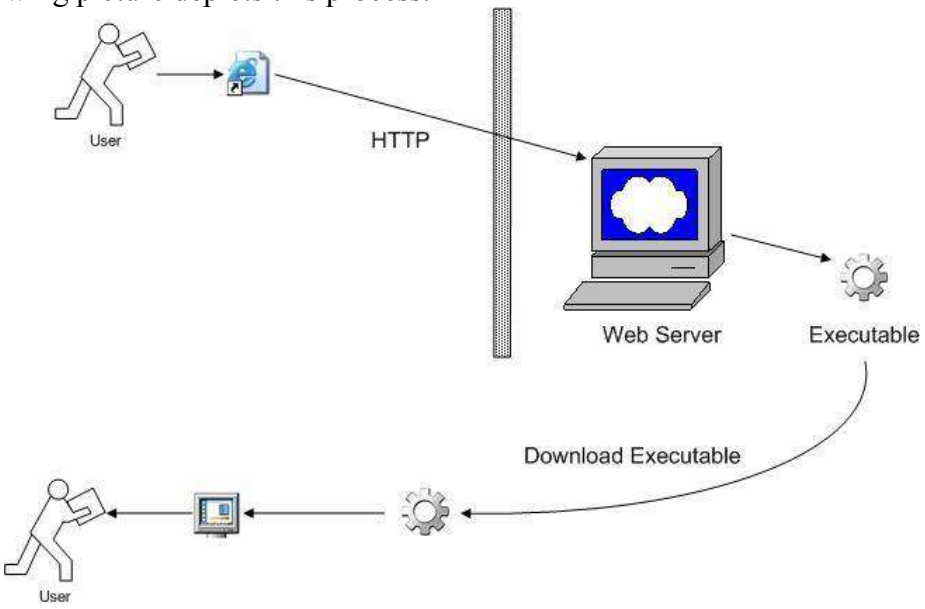
Auto-Deploying An Executable

There are two pieces to auto-deploying a WinForms .NET application. One is to let your users have access to your main executable. The second piece is letting your users download the rest of the application as they use it.

Let's talk about the first piece. Giving users access to the main executable is fairly easy to do with .NET. Basically, you just need to post your executable to a web server and let the user know the URL where the executable resides.

For example, let's say that you have an executable called "loader.exe". You can create a virtual folder in your company's web server and let the user know that the executable is available at http://www.mycompany.com/myapp/loader.exe

The following picture depicts this process:



User can now run the executable by just pointing their browser to this URL. They can also run it by entering the URL via the Run option in the Start menu. Likewise, you can create a web page with an HTML HREF tag that references this URL and have users visit this web page.

Let's use this technique to run one of the .NET executables that comes with the download files for this article:

1. Unzip `CodeDownloadDemo.zip` file to `c:\CodeDownloadDemo` folder.
2. Create a virtual folder in IIS and name it `CodeDownloadDemo`
3. Map this virtual folder to your `c:\CodeDownloadDemo\Loader\Bin` folder
4. Launch Internet Explorer and enter the following URL:
   `http://localhost/CodeDownloadDemo/loader.exe`

You will see how the loader application comes up outside the browser. Once the application is running you can even close the browser.

You can do the same with any .NET executable that you might have. Just drop it on a web server and launch it with your browser.

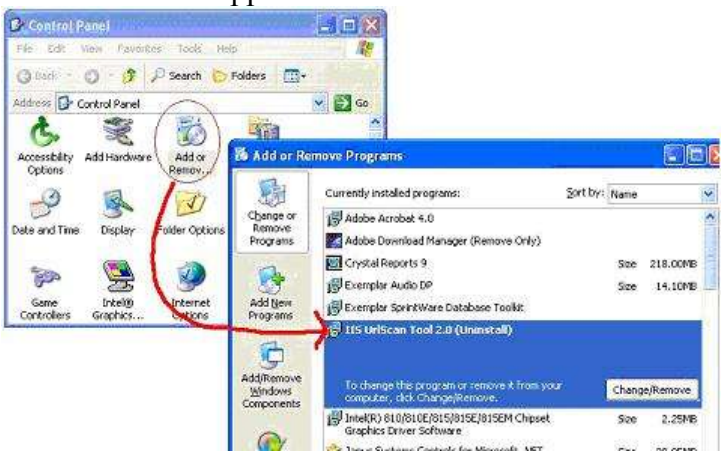
Watch Out For UrlScan Tool

The example above will run seamlessly as long as you don't have the UrlScan Tool installed in your web server. UrlScan Tool was distributed as part of the IIS Lockdown tool to stop invalid URL requests made to web servers running IIS. By default, URL Scan Tool considers invalid URLs that include an executable name on it. Therefore, the URL `http://localhost/CodeDownloadDemo/loader.exe` will be denied by the web server if you have this tool installed.

The following article on MSDN provides more information about UrlScan Tool:

http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/tools/tools/urlscan.asp

You can detect whether you have UrlScan Tool installed in your system by reviewing the list of applications installed on your computer. The following screen shot shows UrlScan Tool in the list of applications installed:



UrlScan Tool can be configured to allow requests for executables. To do so, you can modify the INI file that UrlScan Tool uses to detect valid and invalid requests. This file is usually named `c:\windows\system32\inetsvr\urlscan\urlscan.ini`. Below is a fragment of this file.

```
[DenyExtensions]
; Deny executables that could run on the server
.was_exe
.bat
.cmd
.com
```

Notice how the `.exe` file extension was changed to `.was_exe` to indicate UrlScan Tool that `.exe` extensions are not to be denied. Keep in mind that this is a quick-and-dirty way of working around this restriction. You should read the UrlScan Tool documentation and consult with your network administrator before making this (or any other) change to a production web server.

You might need to restart IIS after modifying UrlScan's INI file in order for IIS to pick up the change. You can restart IIS by running `IISRESET.EXE` program from the DOS prompt.

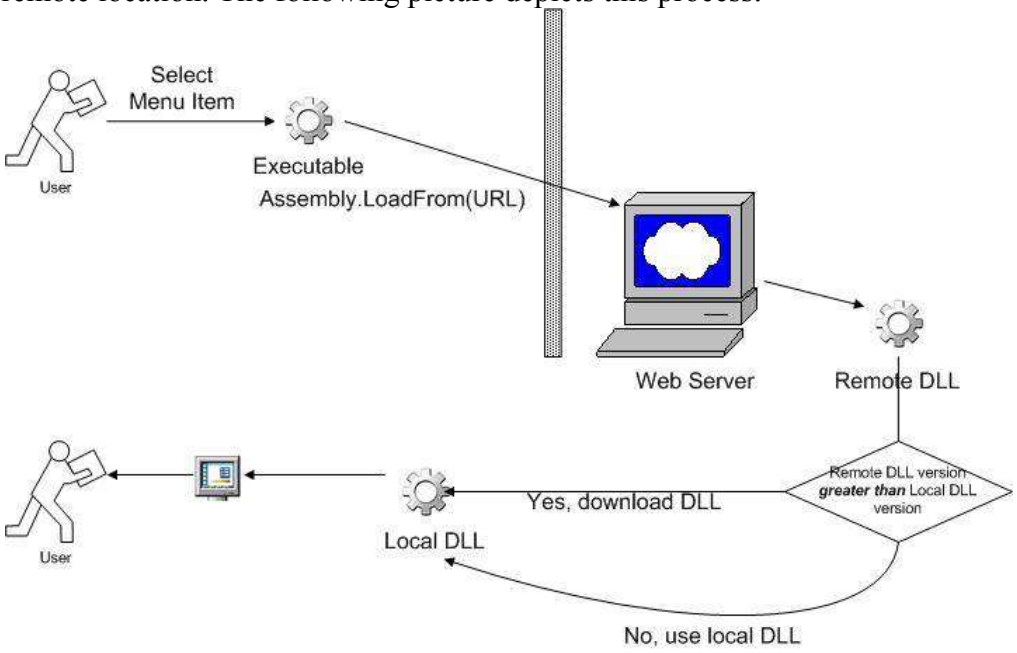
Auto-Deploying the Rest of Your Application

So far so good, you've seen how to deploy an executable over HTTP. This works fine for small applications like the loader (45 KB.) However, real applications are typically much larger than 45 KB. What if your application is 3 MB large? You wouldn't

want your users download 3 MB from your web server *every time* they run the application. This brings us to the second piece of auto-deployment.

A better approach is to install a copy of the main executable on the users' hard disk and let this main executable download the rest of the application on an as needed basis. For example, download a few kilobytes when they run the Invoicing module of your application, another few kilobytes when they run the Employees module, and so forth.

This type of deployment is commonly known as trickle down deployment. .NET provides the tools to incorporate this trickle down deployment out of the box. The classes that provide this functionality are in the `Assembly` class in the `System.Reflection` namespace. `Assembly` class knows, among other things, how to download an assembly from a remote location. The following picture depicts this process:



You could use the following code to download assembly `ModuleA.DLL` from `http://localhost/CodeDownloadDemo/Loader/` and instantiate class `EmployeeForm` from this assembly:

```
'Define URL.
Dim URL As String
URL = "http://localhost/CodeDownLoadDemo/ModuleA.DLL"

'Load assembly from the URL defined above.
Dim a As [Assembly]
a = [Assembly].LoadFrom( URL )

'Get a reference to EmployeeForm class.
Dim t As Type = a.GetType( "ModuleA.EmployeeForm" )

'Create an instance of the class/form.
Dim o As Object = Activator.CreateInstance( t )
```

```
'Show the form.
o.Show()
```

This simple code snipped gives an idea of several interesting features of the .NET framework. Let's review them.

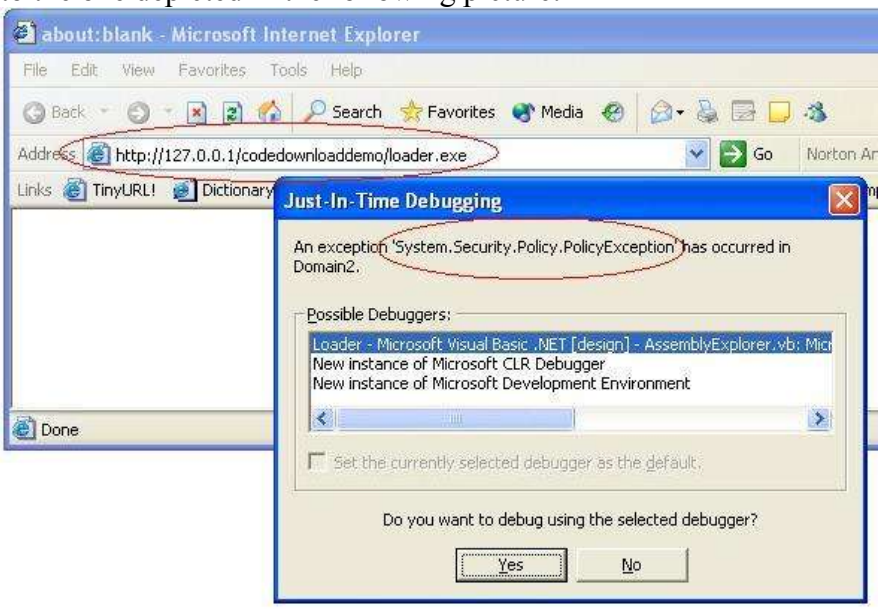
First of all, line `[Assembly].LoadFrom` in previous code is downloading a DLL via HTTP. In this particular example, the URL happen to be `localhost,` but it could as well had been a remote location, like http://www.mycompany.com/myapp. Just like that - one line of code with the right URL and .NET does the rest of the work regardless of the assembly's location.

Secondly, code `a.GetType` is getting a reference to a class inside this DLL that you have just downloaded from a remote location. Notice that the name of a class is a string, which means that we could define the name of the class that we want to reference at runtime. This is very important because now, at runtime, you can decide whether you want to reference `ModuleA.EmployeeForm` or `ModuleA.InvoiceForm`. For strong-types languages like VB.NET and C# this is a very important feature in .NET since it provides a degree of flexibility not common to strongly-typed languages.

Finally, code `Activator.CreateInstance( t )` is creating an actual instance of the `EmployeeForm` class. Once you have an instance of the class then you can start calling methods of this class and setting its properties. In the previous code we are calling method `Show` of the `EmployeeForm` class.

When `localhost` is Different from `127.0.0.1`

The previous two examples will work just fine as long as you use `http://localhost` as the URL. However, as soon as you try to run them using `http://127.0.0.1` you will start experiencing some difficulties. For example, trying to run the loader.exe with the URL `http://127.0.0.1/CodeDownloadDemo/loader.exe` will generate an error similar to the one depicted in the following picture:



The exact screen might vary depending on your configuration, but regardless of how the screen looks like, you will receive some sort of security error. In the previous picture, the error was `System.Security.Policy.PolicyException`.

When you run code from a URL you are (potentially) running code from a remote location. .NET does not really have idea whether the code that you are trying to run comes from a location that you trust or not.

The Security Policies applied by .NET to remote code are rather complex, but at its most basic level .NET considers a URL that does not include a dot inside it (like in `http://localhost`) an *intranet* URL. By default, .NET will let you run code coming from an intranet location. On the other hand .NET considers a URL with a dot inside it (like in `http://127.0.0.1`) an *internet* URL. By default, .NET will not let you run code coming from the internet unless you explicitly tell .NET that the indicated location is a trusted location.

It's All About Security

Security Policies in .NET are a necessary pain. Without them, letting users point their browser to any URL, download a .NET executable, and automatically run it on their

computers would be every virus writer's dream come true. With Security Policies you can protect users from accidentally running code from unknown sources.

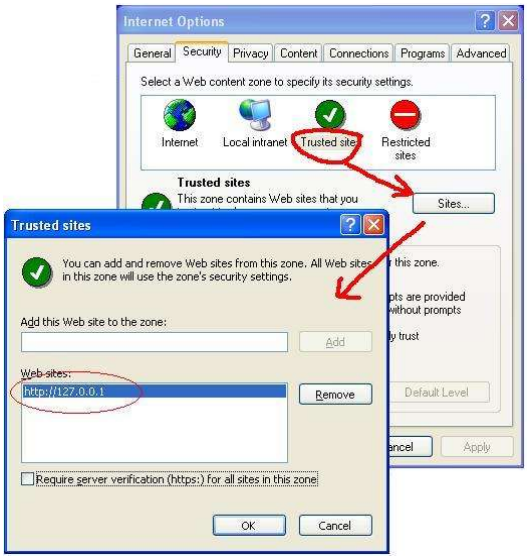
You have probably heard before that .NET is highly integrated with the operating system. Security is probably one of those areas where this integration is more evident.

.NET Security Policies are a very powerful and highly configurable security mechanism. Let's take a look on how these policies are structured and how you can configure them to allow users to download executables from trusted URLs.

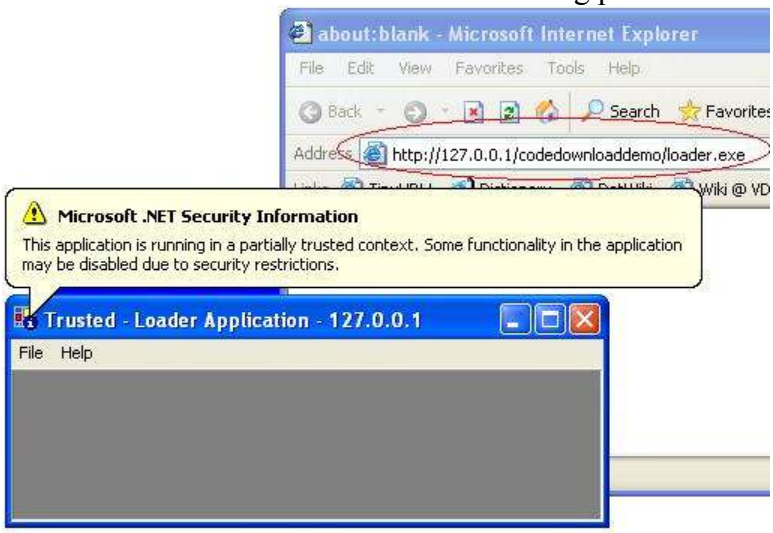
.NET Security and Internet Explorer Security Settings

The first place where you can configure security settings to allow users to run executables via HTTP is, not surprisingly, in Internet Explorer. To do so, launch Internet Explorer, go to Tools menu option, select Internet Options, click on the Security tab, select Trusted sites, and click on the Sites button.

In this case let's add http://127.0.0.1 as one trusted site. Notice that the "Require server verification…" checkbox is unchecked in order to be able to add this site to the list. The following picture shows this.

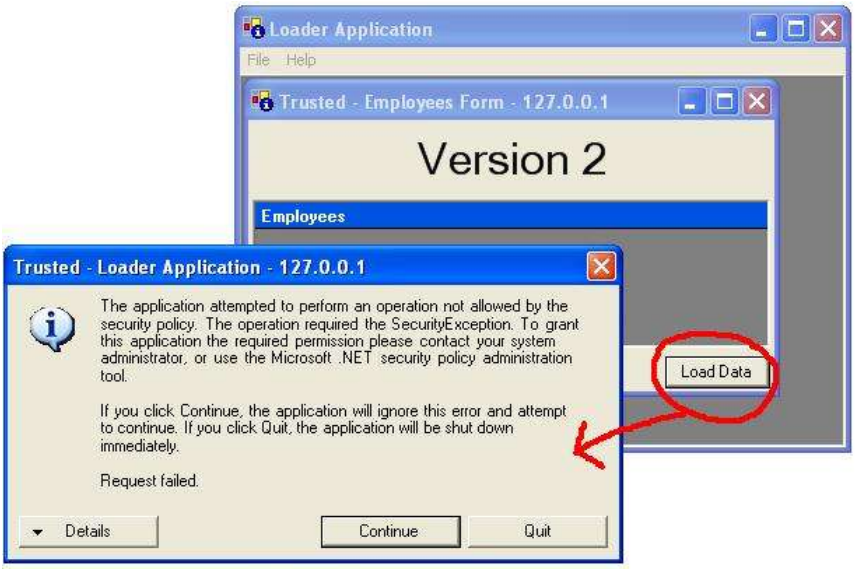


Now that http://127.0.0.1 is a trusted site, let's try to run our executable back again with the URL http://127.0.0.1/CodeDownloadDemo/Loader.exe In this case our application will actually run, however, .NET will let us know that some security issues are still unresolved. This can be seen in the following picture:



Notice how the loader application actually ran this time. However, there is a small information icon on the upper left corner that indicates that this application is running in a *partially trusted* context and therefore some features of the application might not work. Notice also how the URL (`127.0.0.1`) and the word "`trusted`" had been added to the application's title bar.

Since this application is currently *partially trusted* there are some things that might not work. For example, if you run the `Employees Form` from the `File` menu and click on the `Load Data` button you will receive a security message informing that you are performing an operation not allowed by the security policy. In this particular case, the `Employee Form` is trying to read data from SQL Server's Northwind database in your computer, and evidently, the application does not have the security permissions to do it.



.NET Security and the Framework Configuration Tool

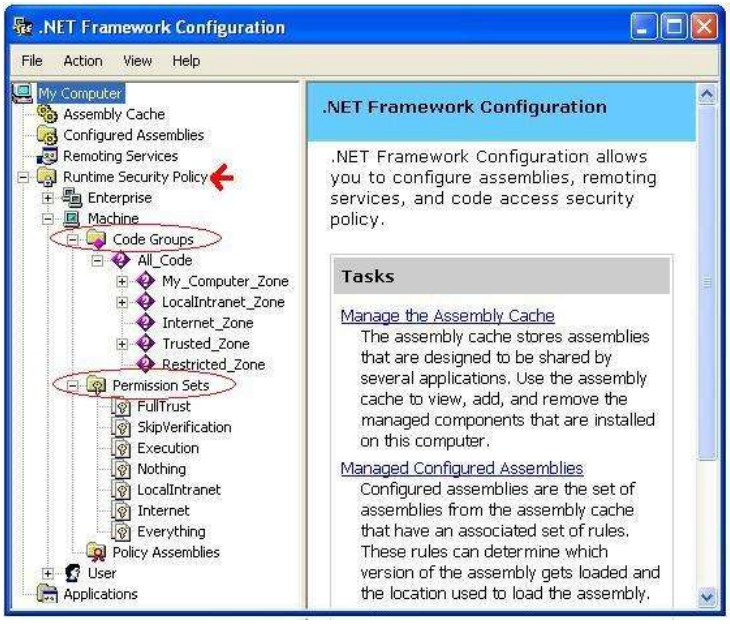
Although Internet Explorer allows configure some of the security settings for . NET applications, it does not provide with a comprehensive list of things that can be allowed and/or denied to .NET applications.

In order to get access to the full range of security options available to .NET applications you need to use the .NET Framework Configuration Tool. This tool is part of the .NET Framework (not of VS.NET) which means that every user that has the .NET Framework installed on his/her computer has this tool installed as well. To launch this tool go to Window's Control Panel + Administrative Tools + Microsoft .NET Framework Configuration.

Using this tool you can configure .NET Security Policies and decide what locations are to be trusted and the degree of privileges that they will receive.

A complete explanation of .NET Security Policies and the tools to configure them would probably require a book. In this section however, we'll have a brief tour of how to use .NET Framework Configuration Tool to configure some of these security settings.

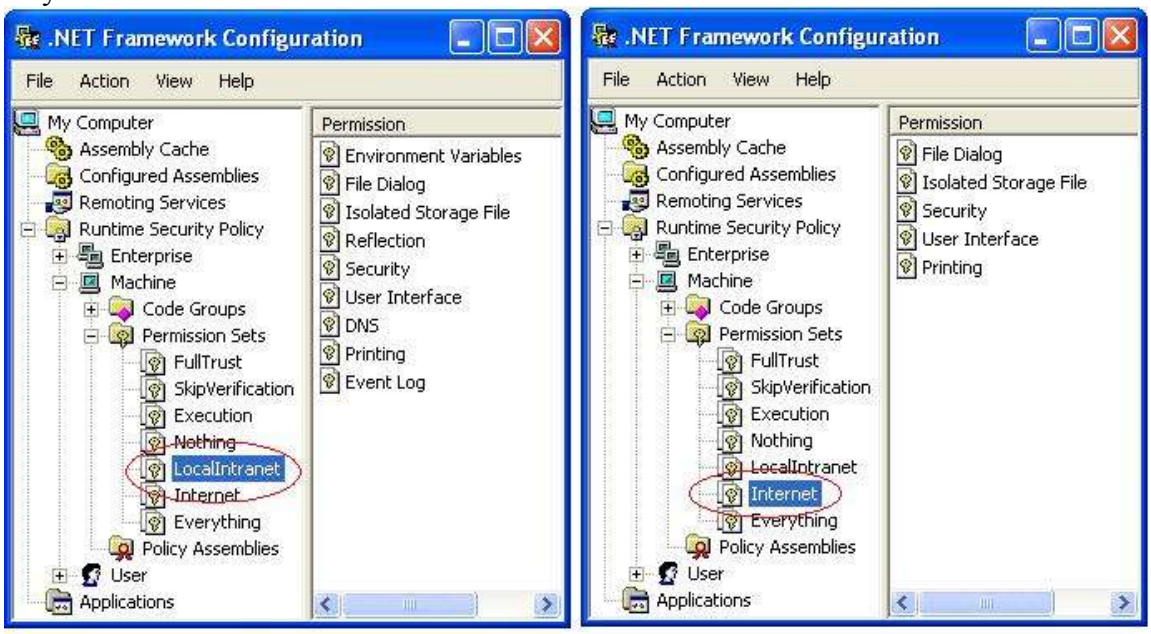
The .NET Framework Configuration Tool can be found in the Control Panel under Administrative Tools. The following picture shows the sections where the Security Policies are defined in this tool.



Security in this tool is defined in two pieces: *Code Groups* and *Permission Sets*.

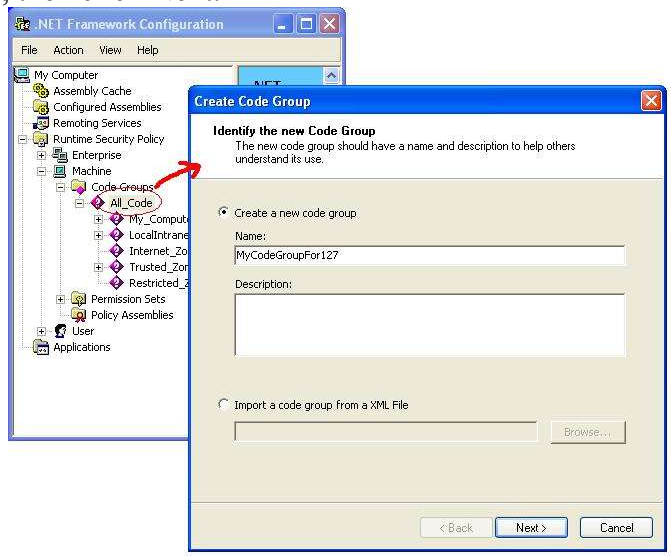
Code Groups is where you categorize what code is to be trusted and what code is to be denied access. You can probably recognize that the predefined code groups in the screen shot correspond to the Zones that you saw in the Internet Explorer Security Setting.

Permission Sets are a mean to assemble different permissions under a single name. For example, the following picture shows the different permissions that have been assigned by default to Permission Sets LocalIntranet and Permission Sets Internet.

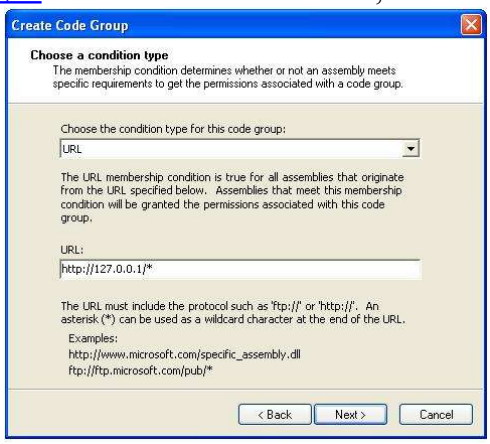


You can define custom Code Groups and Permission Sets by right-clicking in the appropriate node. For example, let's say that you want to give full trust to code coming from URL `http://127.0.0.1` so that the previous examples work with no problems regardless of whether they are executed with `http://localhost` or `http://127.0.0.1`.

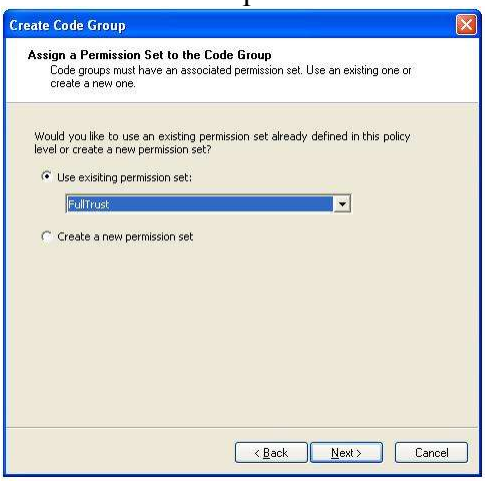
To create a new code group, open Code Groups branch, and right click on the All_Code branch and select New. Enter a name for the new Code Group, for example MyCodeGroupFor127, then click Next.



In the next step, select URL as the condition type for the code group and then enter http://127.0.0.1/* as the URL to trust. Then, click Next.



In the next step, select the Permission Set that you want to give to code coming from URL http://127.0.0.1. For this example, let's select Full Trust. Click Next and then Click finish to save the new Code Group.



Now, if you go back to Internet Explorer and launch our loader program with the URL http://127.0.0.1/CodeDownloadDemo/Loader.exe the application will run with no Security problems. You can even go to the File menu, launch the Employee Form and load Employee data with no problems.

In the previous example you gave Full Trust to a URL. This is not a problem for a demo, however in production environments you need to be more selective and give locations the minimal permissions that you can give them to work. Your network administrator will probably not let you do it otherwise.

I highly encourage you to play with this configuration tool and learn about how to create groups and permissions sets that will match particular security requirements.

In addition to the .NET Framework Configuration Tool, there is another program (a command line utility) that you can use to handle these security settings. This tool is named Code Access Security Policy Tool (CASPOL.EXE) and you can find more information about it at:

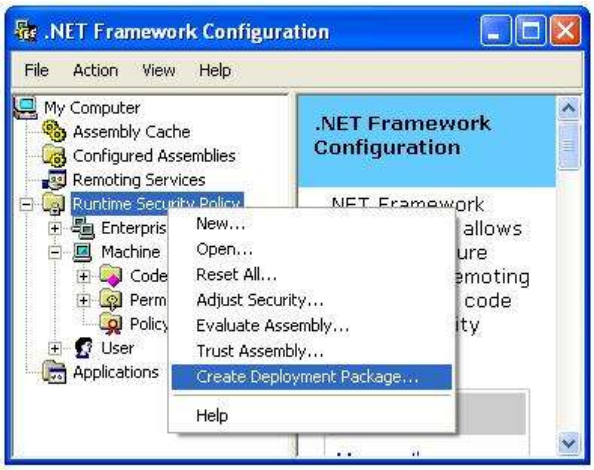
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpgrfcodeaccesssecuritypolicyutilitycaspolexe.asp

Deploying .NET Security Policies

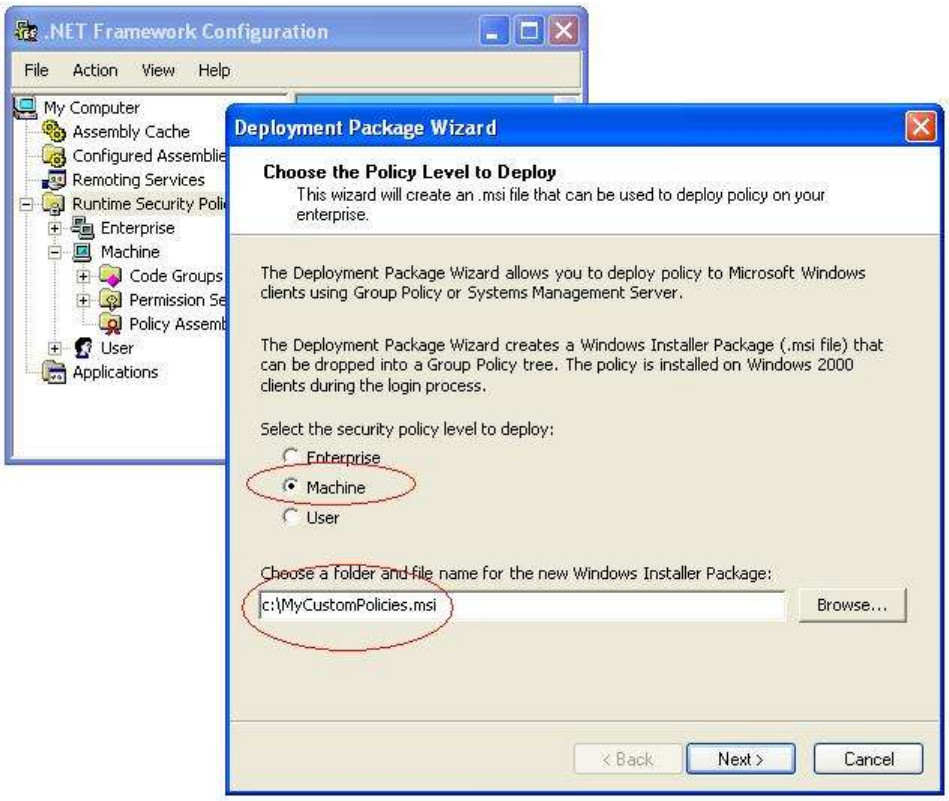
In the previous section you created a security policy to trust code coming from URL `http://127.0.0.1`. Now, you wouldn't want to have to ask your users to follow the same steps as you did to trust your assembly on their computer. Would you?

Fortunately, once you have defined a Security Policy you can distribute this policy to the users and have them add it to their system with a single click.

Once you have defined a Security Policy you can create a Deployment Package. This Deployment Package is a file that will install the Security Ppolicy via a Windows Installer File (an .MSI file.) To create a Deployment Package, simple right-click on the Runtime Security Policy branch, and select Create Deployment Package option. The following picture shows this.



When creating a Deployment Package you can select whether you want to use Enterprise Policies, Machine Policies, or User Policies. For this example, let's create a Deployment Package using Machine Policies. The following picture shows how to do this. Notice that the extension of the Deployment Package is .MSI.



Once you have created a Deployment Package, you can ship this file to your users and have they run it on their computer. Running this file will automatically create on *their* computer the security policies that you defined on *your* computer.

When users run this .MSI file (e.g. by double-clicking on it on the Windows Explorer) the Windows Installer will pick it up and execute it automatically.

In addition to this, a network administrator can push these files automatically to users by configuring it on their network.

Assembly.LoadFrom Revisited

Now, let's review what .NET does behind the scenes when you use Assembly.LoadFrom method to load an assembly from a remote location. Let's take this code for example:

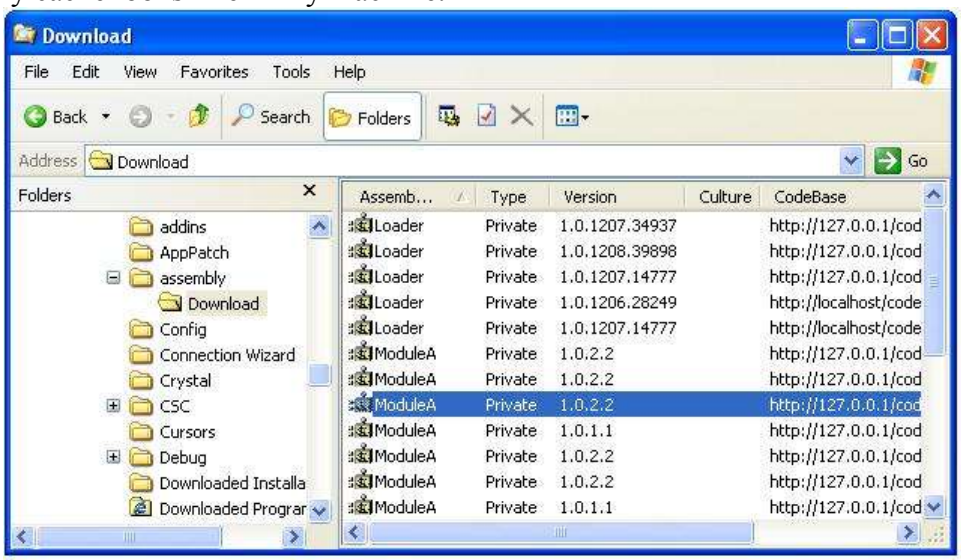
```
'Define URL.
Dim URL As String
URL = "http://localhost/CodeDownLoadDemo/ModuleA.DLL"

'Load assembly from the URL defined above.
Dim a As [Assembly]
a = [Assembly].LoadFrom( URL )
```

When .NET executes the LoadFrom method shown above, it goes to the URL `http://localhost/CodeDownloadDemo` and looks at the ModuleA.DLL assembly on the web server. .NET verifies if you have downloaded this assembly before. If you never have, .NET downloads this assembly and stores it the .NET Assembly Download Cache.

The second time you run this same code, .NET realizes that the assembly has been downloaded before. If the version in the web server is the same one as the one that you have in your Assembly Download Cache then .NET will be smart enough to not download it again and use the one that we already have on disk.

Let's say that you run this code a third time and .NET realizes that the version in the web server is more recent than the one that we have on disk (e.g. a developer posted a new version of ModuleA.DLL on the web server.) At that point, .NET will automatically download the new version, store it in the Assembly Download Cache and run it.

The Assembly Download Cache is a special folder located under `c:\windows\assembly\download`. The following picture shows how the assembly cache looks like in my machine.



A very important consideration to keep in mind is that Assembly.LoadFrom downloads and assembly from the server if its *datetime stamp* is more recent than the one located on the user's disk. This is very important: .NET will compare *dates*, not *version* numbers. Keep this in mind if you have multiple developers compiling executables and DLLs from different computers and their time zones are different.

What if the web server is not there?

What if you execute Assembly.LoadFrom and the web server at `http://localhost/CodeDownLoadDemo/` is not available. For example, what if your user lost his Internet connection or she is on the road with her laptop. In this case LoadFrom will fail and you won't be able to load your classes from this assembly.

There is a workaround for this. It is not a pretty workaround, but it works. You (or your user) just need to switch Internet Explorer to work offline. Once you have done that, then Assembly.LoadFrom will automatically know that it needs to look for the assembly in the Assembly Download Cache.

Requirements

There is a downside to all of techniques described above. The problem is that for automatic deployment to work on the users' computers, they must have installed the . NET Framework on their computer.

Although the .NET Framework is freely available not everybody has a copy installed on their computers. Therefore having to ask users to download a 20MB file and run it on their computers might not be a pleasant addition to your application.

This will be less of an issue in the future as new computers running Windows operating systems start having the .NET Framework already installed on them. In the mean time, you can address this by creating typical installation packages using tools like Install Shield or Wise Installer and have these tools automatically install the .NET Framework on the users' computers.

Summary

.NET Automatic deployment is a very powerful technology that you can use to reduce the hassle associated with deploying fat client applications. Now you can provide users with rich user interface applications that always run with the latest version available without them having to keep going to a web site to download updates.

.NET Framework provides you with the foundation that is needed to make this automatic deployment happen. .NET knows how to download code from a remote location, when to download it and how to cache it for future reference.

A very important feature of this technology is security. .NET lets users download code from remote locations while at the same time protecting them from inadvertently executing code developed by somebody they don't know.

This document has barely touched the surface of what .NET Automatic Deployment can do. I highly encourage you to start experimenting with this new technology and see how you can take advantage of it.