From the January 2002 issue of MSDN Magazine

10/23/2019 • 24 minutes to read

msdn magazine

DHTML and .NET: Host Secure, Lightweight Client-Side Controls in Microsoft Internet Explorer

Jay Allen

This article assumes you're familiar with Visual Studio .NET, C#, Scripting

Level of Difficulty 1 2 3

Download the code for this article: UserCtrl.exe (69KB)

Browse the code for this article at **Code Center**: Multi-File Upload

SUMMARY In the past, Web developers often used ActiveX controls if they wanted customized client-side functionality incorporated into their Web applications. Now, they can build objects supported by the Microsoft .NET Framework which are more compact, lightweight, secure, and seamlessly integrated. By hosting .NET Windows Forms controls in Internet Explorer, developers can realize many of their client-side Web development goals.

This article adapts ActiveX concepts for use with Windows Forms, and builds a multifile upload application that demonstrates these techniques.

With all the talk about Microsoft® .NET these days, it's natural for developers to look to this new platform for alternatives to traditional client-side Web technologies, such as ActiveX®. While the immediate impact of .NET is greatest on the server, corporate intranet developers targeting Microsoft Internet Explorer 5.01 or higher can use .NET Windows® Forms technology to build lightweight, zero-impact, secure client-side objects that can utilize the powerful .NET Framework.

In this article, I'll discuss the benefits and drawbacks of building .NET components for Web page use, and I'll also demonstrate how common ActiveX concepts translate to the .NET Framework. The context for these discussions is MultiFileUploadCtrl, a simple .NET UserControl that I wrote in order to allow users to upload multiple files to a Web server.

Hosting .NET Windows Forms controls in Internet Explorer certainly isn't for everyone. Its use is most feasible to the corporate intranet where your IT department can assure widespread deployment of the Microsoft .NET runtime, which must reside on all target machines prior to control usage. (The .NET runtime—a 20MB distribution at its smallest—will not auto-install upon control detection.) Microsoft does not discourage the use of the .NET runtime on the Internet, however. At the end of this article I'll discuss how Web sites can detect a client's .NET capabilities and serve content accordingly.

From ActiveX to .NET

When I attended the Software Development East Conference in 1996, the Web was a much stranger place than it is now. Netscape was everyone's browser, Perl was synonymous with server-side programming, and I was a mere Unix developer trying to make sense of my borrowed laptop running Windows NT®. I lunched with developers who were abuzz (and confused) about ActiveX, a loose standard for lightweight controls that opened up a world of possibilities to developers using Win32® and who were faced with rudimentary HTML on one hand and platform-agnostic languages on the other.

Since then ActiveX has lost some of its sheen. Due to the large variability allowed by the ActiveX specification (controls are only required to implement IUnknown), many controls don't work properly in Microsoft Internet Explorer, which has very specific interface requirements. The ActiveX installation process itself is fragile and hard to roll back, as it involves numerous registry entries. Cabinets—the package and deployment mechanism for ActiveX controls hosted by Internet Explorer—can be difficult to configure, and installation problems hard to diagnose.

Finally, there are the problems with security. The user-centric trust model used by ActiveX controls can too easily allow a corporate network's security to be compromised by a single trusting employee. Worse yet, security flaws in otherwise trusted components can be used by untrusted Web pages or even e-mail messages to crack into a user's system. Worms like ILOVEYOU and Melissa are two infamous examples.

Using .NET in Internet Explorer via the Internet Explorer runtime host (IEHost.dll) can, with careful planning, help you avoid many of these problems while simultaneously shortening your development cycle via heavy reliance on the framework.

MultiFileUploadCtrl

MultiFileUploadCtrl provides an interface for uploading multiple files to a Web server. It accomplishes this by using the standard multipart/form-data message type specified in RFC 2388, in which a boundary string is declared in the Content-Type header of the request. The body of the POST message contains two or more boundary instances, between which resides the raw data for a single file.

I decided to code my Windows Forms control in C#. So I went to File | New | Project in

Visual Studio® .NET, selected Visual C#™ as the project type, and chose Windows Control Library. This created a default object, UserControl1, which subclasses System.Windows.Forms.UserControl, the base class for all Windows Forms controls. (You use the same process to create a Visual Basic® project—just change the project type.)

Writing client code to perform a MIME multipart POST is not rocket science, but it's a job I'd rather avoid if possible—there's always code to write, so the less needed, the better. This is where the power of .NET comes in. The class I need to perform a MIME multipart upload, WebClient, is already defined in the System.Net namespace. My job is reduced to defining the interface for file selection and delegating upload to WebClient. (If you want to upload all your files in a single HTTP request, you'll need to roll your own solution. WebClient.UploadFiles does not have an overload that accepts an array of strings.)

Since I'll need to read information from the hard drive, I restricted file selection to a dialog that's raised using the System.Windows.Forms.OpenFileDialog class. I'm writing privileged client-side code, so I can include some other information in the UI, such as a running tally of the total size of the upload in bytes. Assuming that the server application that will utilize this control places quotas on the amount of disk space each user can use for file uploads, I defined another property, MaxSessionUpload, which specifies the maximum upload size in bytes for the current instance of the HTML page. The default value is -1, which in this case means an unlimited upload size.

From here, it's all done but for the shouting. This control might appear on a page with other FORM elements; therefore, I don't give it a native Upload button. Instead, I define a public method, UploadFiles, that will spin off a worker thread to take the results of the user's file selection and upload them, one at a time, to the Web server (see Figure 1).

The control also relies on a tiny server component—an .aspx (ASP.NET) file that saves the uploaded files to the directory in which the .aspx resides. It does this in the Page_Load event handler, relying on the convenient Request.Files object to iterate over the uploaded collection, as you can see in Figure 2.

A file named testClient.htm is included in the project to demonstrate the control. As with ActiveX controls, .NET controls are loaded via the HTML OBJECT tag. Note the syntactical differences, though. A typical ActiveX OBJECT tag looks like this:

Copy

Copy

Copy

Copy

Copy

```
<OBJECT id="upload1" classid="CLSID:EE0C7AF7-6744-4017-93C5-
```

```
 560C7976F6A5"


codebase="Upload.cab#-1,-1,-1,-1">


•••


</OBJECT>
```

In ActiveX, all objects are identified by their CLASSID, a 128-bit Global Unique Identifier (GUID). The CODEBASE attribute is used to specify either the portable executable (PE) in which the object resides, or a Cabinet file containing the PE.

.NET, by contrast, uses what is called no touch deployment, meaning that .NET assemblies do not have to alter the registry or any other system files in order to be recognized by the runtime. If you run the Visual Studio-based utility DUMPBIN on a .NET DLL, you'll notice that the familiar COM exports, such as DllRegisterServer, are absent. So the OBJECT tag looks a little different:

```
Copy
```

```
Copy
```

```
Copy
```

```
Copy
```

```
Copy
```

```
Copy
```

```
Copy
```

Copy

```
<OBJECT id="upload1" classid=

"Upload.dll#TestCorp.ClientControls.MultiUploadCtrl"

width=800 height=300 style="font-size:12;">

<PARAM Name="FileUploadURL"

 Value="http://localhost/ServerFileUpload/UploadFile.aspx">

<PARAM Name="MaxSessionUpload"

Value="10646">

</OBJECT>
```

The CODEBASE attribute is gone; instead, I overload the CLASSID attribute with the following syntax

Copy

```
component-name#namespace-path.control-name
```

where component-name identifies the .NET assembly, namespace-path names the namespace in which the Windows Forms control is located, and control-name is the name of your Windows Forms control subclass. Internet Explorer parses this string and looks for the assembly in the specified location. By default, it looks in the current Web server directory (APPBASE), and then in that directory's bin directory.

There are two limitations with this variant use of OBJECT that might throw ActiveX-centric developers for a loop. First, the OBJECT tag cannot be used to load any preexisting Windows Forms controls installed to the Global Assembly Cache (GAC)—essentially, the entire .NET Framework. Windows Forms controls can (and must) use framework objects within their own implementations, but you cannot directly instantiate, say, the Windows Forms Listbox control in Internet Explorer without the use of a custom Windows Forms controls wrapper.

A corollary is that you cannot install assemblies from the Web directly into the GAC. Fusion is the name of the new no touch component installation technology; it made its premiere in Windows 2000 in the form of side-by-side installs. Fusion stores them in a separate location, the download directory. This is a separate directory from the standard Downloaded Program Files in Internet Explorer used for ActiveX controls. You can view the download directory via a Windows shell extension installed with .NET by browsing %SYSTEMDIR%\assembly\download. You must use the command gacutil to manipulate the contents of this directory. The following command clears the contents of the download cache (which becomes quite polluted during control development as well as during testing):

Copy

```
gacutil /cdl
```

Second, as noted in the .NET SDK documentation, a single CAB can currently only contain one .NET assembly. This complicates multi-assembly applications as one of the main benefits of creating CAB files, consolidating multiple network accesses into a single download, is lost. It also complicates mixed applications—Windows Forms controls with a .NET front end that utilize COM Interop to talk to existing COM objects that might be either too expensive to migrate to .NET or too inefficient unless programmed in C++. Developers deploying such

solutions to new clients will need to use a two-stage deployment process: installing the COM components and the .NET UserControl interface.

The New Security Model

The ambitious among you have probably already put these files on a development Web server and tried to run testClient.htm. If you did, you saw the dialog shown in **Figure 3** from the .NET Framework when you tried to select a file.

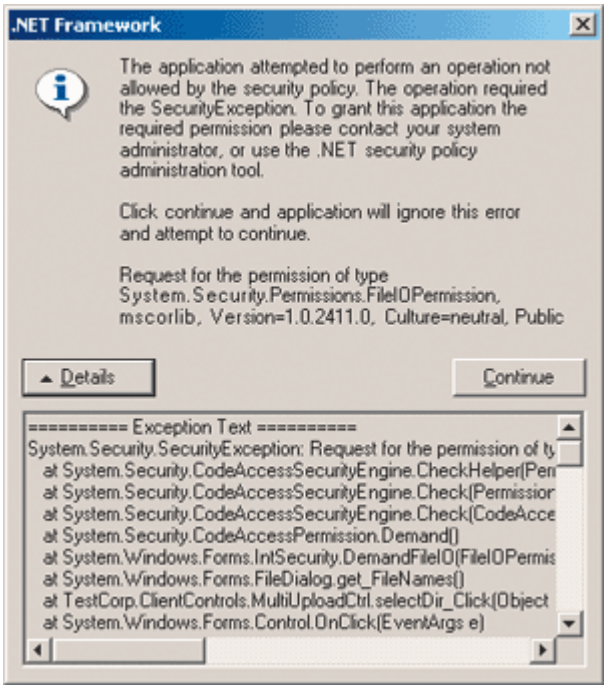

Figure 3 Error Dialog

This may come as a shock to many developers since file system access is trivial in ActiveX. Even if your ActiveX control is unsigned and not marked safe for scripting, there is nothing to prevent you from accessing privileged system resources if the user decides to run your control. Part of the power of ActiveX controls is that controls can access system resources with impunity. Safe controls, those that don't access privileged system resources, were encouraged to implement the IObjectSafety interface. This permission grant is all-or-nothing; allowing a control to run gives it access to all system resources.

Unfortunately, even a truly unsafe control can implement IObjectSafety and declare itself safe. This can have nasty consequences for a private user with private network access via dial-up or broadband. For a corporation, however, it can be disastrous. A single user's decision can allow an unsafe control to map internal network topology, trip the user's Ethernet card into Promiscuous mode, and sniff data for the entire subnet or access sensitive company data located on mounted shared drives. Even if a component is not intentionally malicious, a security hole can be exploited by unscrupulous script. Internet Explorer has been hit in recent years by security flaws that have cropped up in ActiveX components that ship with the browser.

.NET attacks these problems in three large strokes. First, security in corporate networks is no longer solely the province of individual users. Security permissions for controls can bind to

one of three hierarchical security policy levels: User, Machine, and Enterprise. Enterprise constrains Machine, which constrains User. In other words, the Machine policy cannot grant permissions to code that the Enterprise policy denies. System administrators (and users, if they're adventurous) can apply permissions to a fine level of granularity using code groups, which identify one or more pieces of code on the basis of some variety of evidence. Proper evidence for a Windows Forms control includes its strong name (public key, name, and version), zone (Internet versus intranet), and other such attributes.

Second, security permissions in .NET also have a fine granularity. .NET contains a permission object for every resource available on a user's machine, including File I/O, Web access, unmanaged code execution, and a host of others. Administrators or power users grant permissions to .NET code by grouping them together into a permission set, which is then applied to a code group.

Finally, the .NET runtime allows components to control how other components access them through the use of permission demands and the stack walk. Suppose an application object in an untrusted Assembly A calls an object in Assembly B that does in-file sorting; Assembly B, in turn, calls a .NET Framework class in trusted Assembly C to read data from and write data to the file system. When asked to write to the file system, Assembly C calls the method System.Security.Permissions.FileIOPermission.Demand, requiring that all components in its call stack have permission to write to the file system. Assembly A lacks that permission, so the method call will terminate with a security exception before it ever executes. This powerful mechanism prevents untrusted components from coercing trusted components into performing unsafe operations.

This has a large implication for .NET Windows Forms controls, since the framework component that hosts .NET Windows Forms controls in Internet Explorer, IEHost, has a restricted permission set that prevents many potentially unsafe operations. Since my control needs to read from the file system, I need to complete two steps, as outlined here.

1. Grant my Windows Forms control extended permissions using the .NET Framework Configuration Tool (mscorcfg).
2. Add an Assert to my assembly. The System.Security.SecurityAction.Assert method allows a component (or an object within) to short-circuit the stack walk, so that less trusted components can use trust-ed components without causing a SecurityException to be raised.

.NET supports specifying permission requests either at compile time using Attributes (a declarative request) or at runtime (an imperative request). I've incorporated Step 2 into Upload.dll using imperative requests so that I can grant permission only to those lines of code that require them, as you can see in the following example from the selectDir_Click event handler.

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

```
new FileIOPermission(PermissionState.Unrestricted).Assert();

foreach (string fileListItem in dirDialog.FileNames)

{

    // Has the user already selected this file?

    // If so, don't add it again.

    // !TODO - user notification of dupes?
```

```
    if (!fileTable.Contains(fileListItem))

    {

        AddFile(fileListItem);

    }

}

CodeAccessPermission.RevertAssert();
```

This prevents malicious script or other executable content from trying to abuse the enhanced level of trust.

Defining New Permission Sets

To complete Step 1 (mentioned earlier), I'll use the .NET Framework Configuration Tool, the Microsoft Management Console (MMC) snap-in that provides the GUI for security management. When I'm finished, my Windows Forms control should run in the browser without security exceptions. You can launch the tool (see Figure 4) by entering the following string into the Run dialog (depending on your Windows directory—if you're using Windows XP, use "windows" instead of "winnt"):

Copy

Copy

```
mmc c:\winnt\microsoft.net\Framework\<%FRAMEWORK_VERSION_DIR%>/

    mscorcfg.msc
```

The item that's of most interest in this list is "Runtime Security Policy," which, as you'd expect, breaks down to a Code Group folder and a Permission Sets folder.

.NET components inherit their default security permissions based on the Machine security permissions. By expanding Machine | Code Groups | All Code, selecting Intranet Zone, clicking on "Edit Code Group Properties" and selecting the Permission Set tab, you can see that Windows Forms controls in Internet Explorer operate by default in a pretty tight sandbox; they cannot perform file I/O on any disks, use the System.Net classes to connect to any server other than the hosting Web server, or make any calls to unmanaged code (for example, ActiveX objects or Win32 APIs not wrapped by the framework).

If you compare intranet and Internet Zone permissions, you'll notice that permissions in the intranet, as expected, are somewhat richer than in the Internet, with same-site Web access, Limited Reflection, and DNS all forbidden for controls downloaded from outside the corporate network.

By default, you do have permission to use the OpenFileDialog control in the Intranet Zone. You might ask yourself at this point: what good is this permission if you can't read from the file system? The answer is that if you're only reading a single file from the drive, you don't need file I/O permissions. The OpenFileDialog class has a method, OpenFile, that will return a Stream object for the file that the user has chosen, as shown in the following code.

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

```
using (OpenFileDialog fd = new

    OpenFileDialog())

{

if (fd.ShowDialog() == DialogResult.OK)

{

        using (Stream s = fd.OpenFile())

{
```

```
            StreamReader reader = new



            StreamReader(s);



            //do stuff



        }



    }



}
```

However, my component allows multiple file selection and requires at least Read and PathDiscovery access, so I need to implement a bit more. Therefore, I selected Permission Sets for the machine policy (if you were configuring this for a corporation, of course, you'd perform these operations at the enterprise level) and click Create New Permission Set (see Figure 5). To do this yourself, in the first dialog, give the set a name such as "Extended UserControl Permissions" (see **Figure 6**).

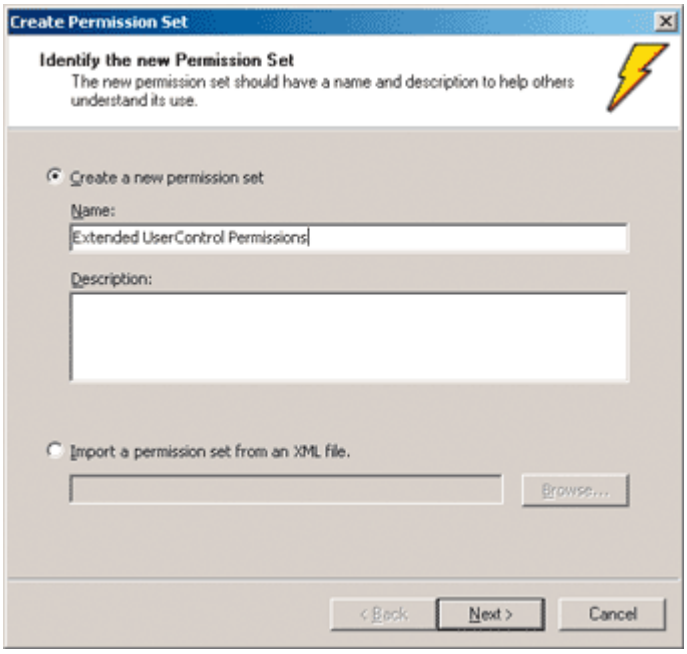

Figure 6 New Permission Set Dialog

Click Next, and from the Available Permissions dialog on the left, double-click file I/O. This pops up a dialog asking which permissions you want to grant for which directories. Check the radio button at the bottom for "Grant assemblies unrestricted access to the file system" (see **Figure 7**). Also, go ahead and add the Security permission by checking "Allow calls to unmanaged assemblies" in the next dialog. The reason for this will become clear when I discuss raising events from your control that are handled by scripts.

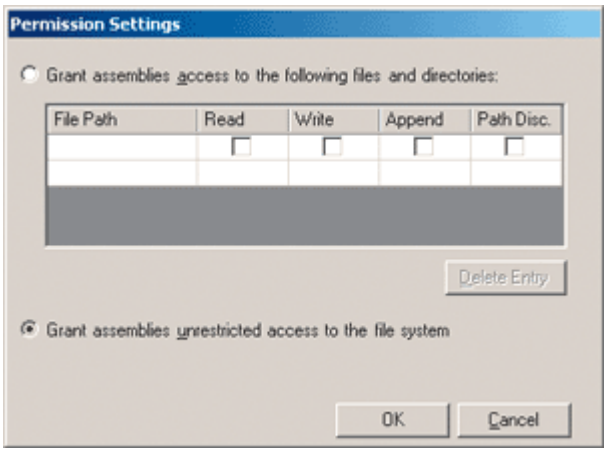

Figure 7 I/O Permissions

Defining a New Code Group

Now that the Permission Set is defined, I need to associate it with my Windows Forms control. I only want my control to run on intranet machines, so I select the LocalIntranet code group and, from the right-hand pane, click Add a Child Code Group (see Figure 8). I use the first screen to give this group a unique name like "MultiFileUploadCtrl V1" (see **Figure 9**).

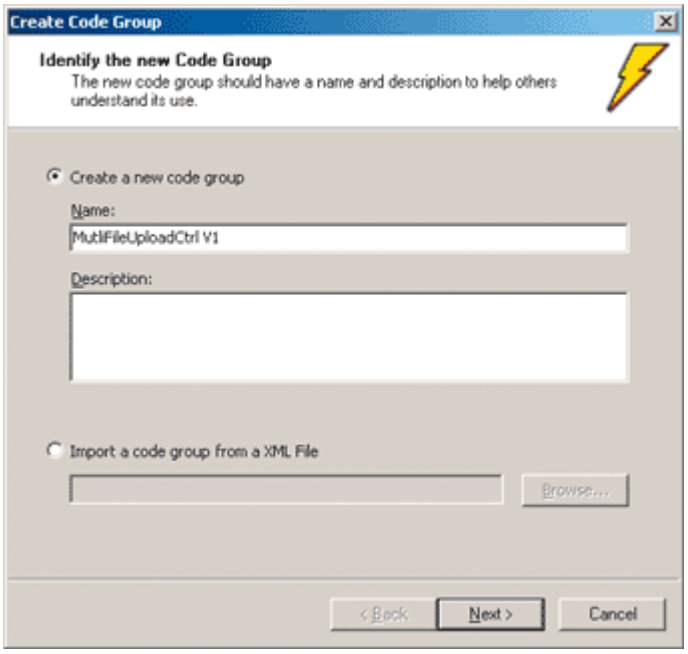

Figure 9 Create New Code Group

On the next screen, I need to supply the evidence that defines the code group. The best bet is Strong Name, a unique name consisting of a component name, component version, and a public key (see **Figure 10**).

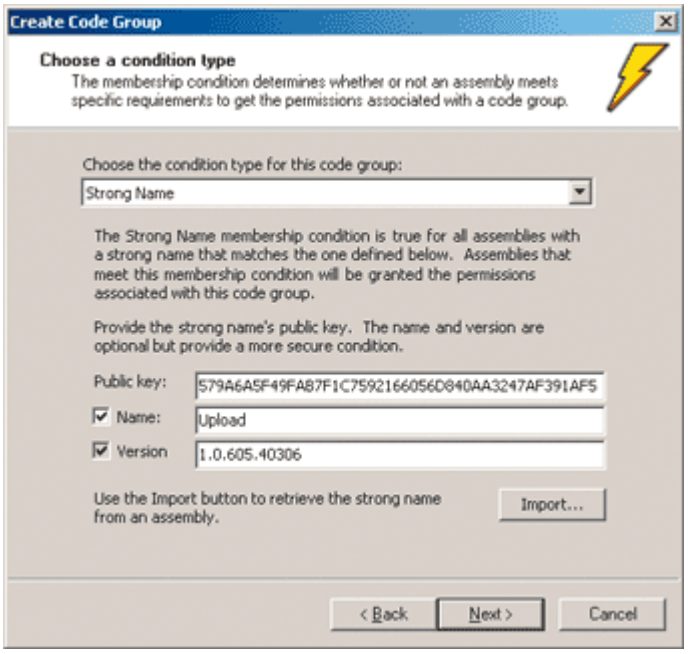

Figure 10 Choosing a Condition Type

I've already generated a public-private key pair using the command-line utility sn:

Copy

```
sn -k client.snk
```

To apply this to my assembly, I added the following line to AssemblyInfo.cs in my Visual Studio .NET-based C# project:

Copy

Copy

```
[assembly:


AssemblyKeyFile("client.snk")]
```

In mscorcfg, select Strong Name from the screen's dropdown, and use the Browse button that appears below it to select Upload.dll. The Strong Name information will populate automatically. Technically, you can base permissions on the public key alone. This can be useful, as it's one way of granting extended permissions to a control from a specific publisher, such as your company's IT group. I strongly recommend (no pun intended) binding permissions to the complete Strong Name. That way, if you discover a security flaw in your control at a later date, you can yank permissions for that control so that it no longer executes correctly in your enterprise, while still allowing future, fixed versions of the control to function.

Note that it is possible to specify greater and greater security through hierarchical nesting of Code Groups. If I so desired, I could have restricted access to my Windows Forms control to a specific Web site in the enterprise by creating a Code Group under LocalIntranet using the Site condition and default LocalIntranet permissions, and then creating a Strong Name-specific Code Group under that group with extended permissions.

To test a control to make sure it's been granted the appropriate permissions, right-click on Runtime Security Policy and select Evaluate Assembly. In the File name box at the top, specify the localhost path to your assembly, which should be http://localhost/ServerFileUpload/Upload.dll (see **Figure 11**). You'll see a list of all permissions that apply to your Windows Form control's assembly—double-click on any one to view its setting. If everything checks out, you should be able to open Internet Explorer, surf to the testClient.htm page, and test the control for yourself.

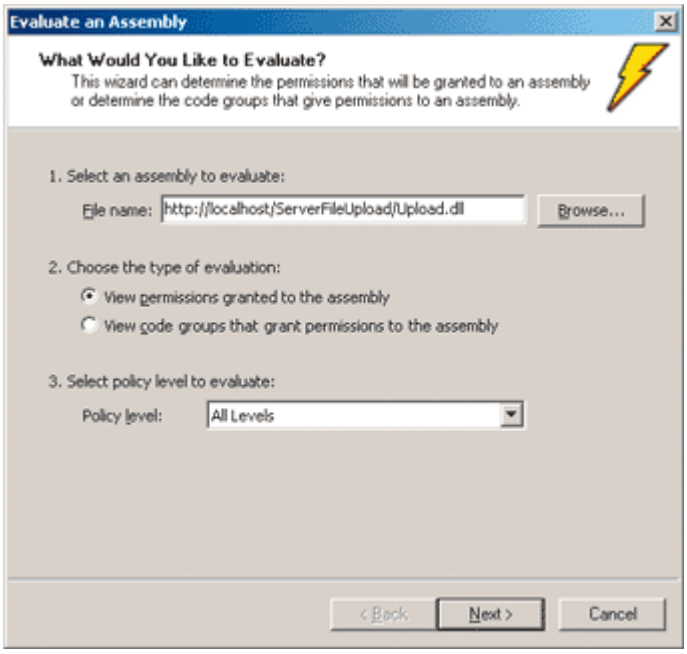

Figure 11 Evaluating an Assembly

PARAM Tag Initialization

You've now seen how .NET Windows Form control security gives you much greater flexibility and power than the user-centric ActiveX model. Let's take a quick look at some of the other features of .NET Windows Forms controls, starting with PARAM tags.

Since COM objects had no constructor per se, several mechanisms evolved to initialize and save objects in the form of the IPersistxxx family of interfaces: IPersistStream, IPersistStreamInit, IPersistMemory, IPersistFile, IPersistMoniker, and IPersistPropertyBag.

When ActiveX controls were introduced, the same mechanism was kept, but with a new interface added: IPersistPropertyBag. A property bag is a collection of name/value pairs extracted from the PARAM subtags of the DHTML OBJECT tag:

Copy

Copy

Copy

Copy

Copy

```
<OBJECT id="ComObj1" ClassID="CLSID:....">
```

```
    <PARAM Name="URL" Value="http://

    server.com/Resource.xxx">

    <PARAM Name="DataBind Value="Yes">

</OBJECT>
```

.NET Windows Forms controls also implement IPersistPropertyBag. However, this implementation is inaccessible to developers; instead, the .NET runtime host for Internet Explorer (IEHost.dll) uses Reflection to discover the names of all public properties defined on your control. If it finds a property with a name that matches the NAME attribute of a PARAM tag, it does a property set. There's no need to handle IPropertyBag directly, as you do in C++ ActiveX controls, or implement the ReadProperties and WriteProperties methods as in Visual Basic 6.0 UserControls; IEHost handles the busy work for you.

There are, of course, catches to all this convenience. First, your property must be typed as a string, or else you must supply a custom object that converts the string representation of the PARAM Value into the type of your choice.

Second, don't bother throwing exceptions within property procedures initialized via a PARAM tag, as Internet Explorer shuts off all exception handling during initialization. This is not a change for managed code: calling SetErrorInfo from within IPersistPropertyBag::Load in a C++ ActiveX control will also come to naught. (Calling Err.Raise in UserControl_ReadProperties in Visual Basic will generate an error—a Visual Basic runtime error, not a COM IErrorInfo-style error.) If you truly want property implementations to throw exceptions (thus halting script execution), you can initialize your properties in your onload event handler for your document body, as you can see in Figure 12. Otherwise, move your check and exception logic into a private method inside your Windows Forms control and call it in every public method that uses your property.

This begs the question whether you can use PARAM tags for one-time initialization. The MaxSessionUpload property of my Windows Forms control is a good example. There's little reason for client developers to change this value on the fly, especially if it's provided by a component that monitors user disk quotas on the Web server. Given the nature of corporate

turnover, you'd probably want to make this crystal clear to all developers present and future by not defining the set function for this property.

This is trivial under ActiveX in either C++ or Visual Basic. Unfortunately, there's no straightforward solution in .NET. One idea would be to throw an exception in the set block of your property procedure if the variable you're attempting to assign to is already initialized. This idea is functional, but very inelegant, as it raises more questions than it answers. Another idea would be to grant your assembly unmanaged code execution permission, pass in the IHTMLObject object in the Internet Explorer Document Object Model (DOM) corresponding to your control, and read the appropriate IHTMLParam tag. This would allow you to remove set from your property definition, causing .NET to fail silently during Reflection. A full discussion of such a workaround is beyond the scope of this article.

Handling Events

The last hurdle is to add event notifications to my Windows Forms control, so that it can inform its container when downloading has started, stopped, or aborted due to an internal error.

In the COM/ActiveX world, you add events by defining an event interface (also called a source interface). Controls that want to throw these events implement the interface, and their callers sink the interface—to register themselves as event handlers—using the Advise method of the IConnectionPoint corresponding to the control's default source interface. Regular .NET classes simplify this task greatly. Events no longer need to be grouped into interfaces, and listeners are provided for individual events using a type-safe function pointer called a delegate.

As of version 6.0, Internet Explorer provides no seamless translation between these two mechanisms. The best you can do is to use COM Interoperability to define a default source interface. Internet Explorer will sink a control's default source interface and expose event handlers through an IDispatch interface based on a dispid assigned to the events. This requires you to define, COM-style, an interface that implements the events:

Copy

Copy

```
[Guid("A59B958D-B363-454b-88AA-BE8626A131FB")]

[InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]

public interface IMultiUploadCtrlCOMEvents

{

    [DispId(0x60020000)]

    void UploadComplete();



    [DispId(0x60020001)]

    void BeginUpload();

}
```

It also requires you to define all of the control's public methods and properties on a separate interface, otherwise, Internet Explorer will attempt to resolve all method and property calls made by script against the source interface, with obvious bad consequences.

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

```
public interface IMultiUploadCtrlCOMIncoming

{

    void UploadFiles();

    bool FilesPending {get;}

    int MaxSessionUpload {get; set; }
```

```
    int BytesUploaded {get;}

    string FileUploadURL {get; set; }

}
```

From there, you define and raise an event just as you would any regular .NET event. To sink, you must use the DHTML <SCRIPT> tag with the FOR attribute:

Copy

Copy

Copy

Copy

Copy

Copy

Copy

```
<SCRIPT FOR="upload1" EVENT="BeginUpload">

    window.status = "Uploading files...please wait";
```

```
</SCRIPT>



<SCRIPT FOR="upload1" EVENT="UploadComplete">

    window.alert("Upload complete");

    window.status = "";

</SCRIPT>
```

Hence the reason for requiring unmanaged code execution permissions for my control: when I raise an event, I am calling unmanaged listeners written in JScript or VBScript.

Debugging in Internet Explorer

As of Beta 2 of .NET, debugging in Internet Explorer is still somewhat problematic. The intuitive approach is to set my control's Debug Mode to URL (right-click Upload in the Visual Studio .NET Solution Explorer, select Properties, and select Configuration Properties | Debugging from the Property Page). Unfortunately, this doesn't work.

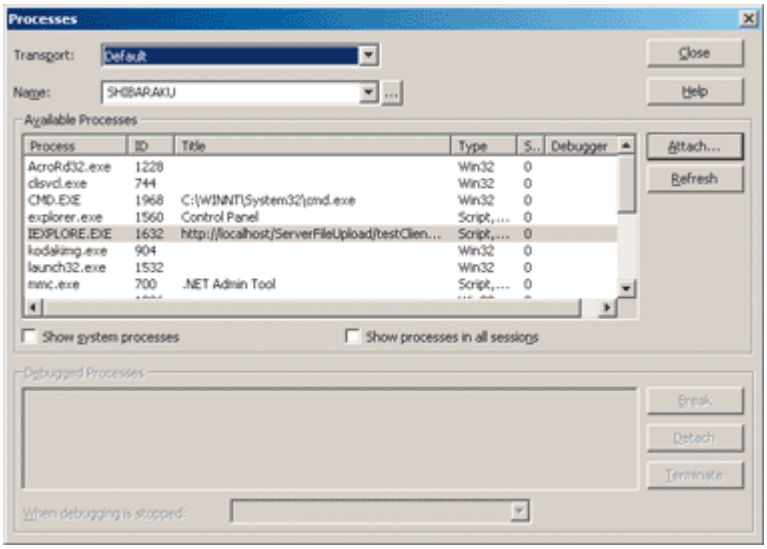

Figure 13 Debug Process Window

The best bet is to attach to the instance of IEXPLORE.exe hosting your control. The only drawback to this approach is that an instance of your Windows Forms control must be running in the browser, so that Visual Studio .NET can hook into a hosted instance of the .NET runtime. Once you have testClient.htm loaded, choose Debug | Processes from Visual Studio .NET, choose the appropriate instance of IEXPLORE.exe from the process list, and click the Attach button (see **Figure 13**). A new dialog will ask you which type of programs you wish to debug; make sure Common Language Runtime (CLR) is selected (see **Figure 14**). From this point on, you can insert breakpoints anywhere in your Windows Forms control code. The new unified development environment makes it a snap to step through script in testClient.htm and seamlessly step into method calls made by script against your Windows Forms control.

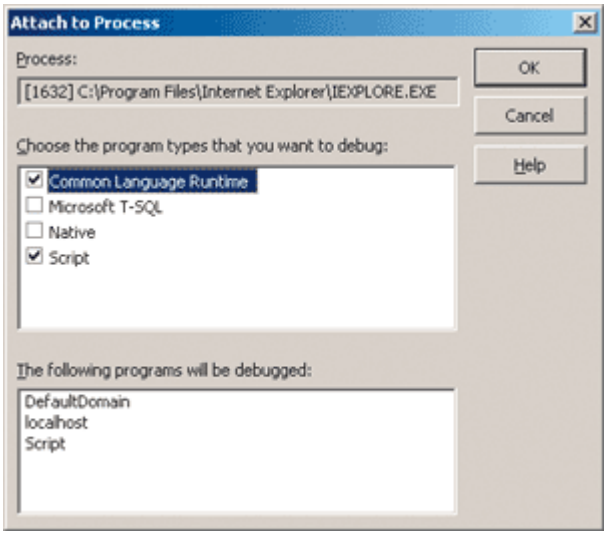

Figure 14 Attach Dialog

What happens if your control never displays in Internet Explorer? In the COM world there is a little utility called CDLLOGVW.exe, the Code Download Error Log Viewer, which retrieves log files of failed downloads from the Internet Explorer cache. In .NET the equivalent tool is called FUSLOGVW.exe, the Fusion Error Log Viewer. FUSLOGVW is self-explanatory to anyone who's used CDLLOGVW.exe. Those new to component download debugging can refer to

Knowledge Base article Q252937 "Find More Information About Why Code Download Failed" for some quick tips and pointers.

.NET and the UserAgent String

My control is served up in a static HTML file, partly because I don't need any dynamic capabilities in a simple sample, and partly because inserting .NET-style OBJECT tags in an .aspx in .NET Beta 2 makes Visual Studio .NET throw a host of COMExceptions. In a production environment, the PARAM values (at least) would likely need to be generated dynamically. I also might need to know whether a particular machine has the CLR installed; in the future, it'll be important to distinguish between different installed versions of the CLR.

If you perform a Network Monitor trace of a .NET request, you'll notice something interesting. The UserAgent string is modified to reflect the .NET CLR versions installed on the client's machine.

Copy

Copy

Copy

```
User-Agent: Mozilla/4.0 (compatible; MSIE


6.0b; Windows NT 5.0; .NET CLR 1.0.2901;


.NET CLR 1.0.2914)
```

If you're using ASP.NET, the work here is done for you. The HttpBrowserCapabilities class defines a property, ClrVersion, that provides this information:

Copy

Copy

```
string clrV;



clrV = Request.Browser.ClrVersion;
```

Those sites still using classic ASP can perform the same magic with some simple regular expressions, as shown here using server-side JScript®:

Copy

Copy

Copy

Copy

Copy

Copy

Copy

Copy

```
<%@Language="JScript" %>



<%



Response.Write("HTTP User Agent: " +
```

```
Request.ServerVariables("HTTP_USER_AGENT") + "<br>\n");

var mc = new String(Request.ServerVariables("HTTP_USER_AGENT"))

    .match("\\.NET CLR((?:\\d+(?:\\.)?)+)");

Response.Write("Version: " + RegExp.$1);

%>
```

Conclusion

.NET Windows Forms controls hosting holds some peril, but also great promise. The boost in the security system alone can be enough to convince an IT department to convert their existing ActiveX control base to .NET. Look for support to improve in future versions of the Windows Web client platform as the .NET story continues to unfold.

For related articles see:
Using Windows Forms Controls in Internet Explorer
For background information see:
Introduction to ActiveX Controls
IObjectSafety Extensions for Internet Explorer

Jay Allen *is a Technical Lead in Microsoft Developer Support's Internet Client Team, and a regular contributor to MSDN Online's Web Team Talking column. He has over six years experience developing n-tier Web-based applications for a variety of Fortune 500 companies.*