

: .NET Security Overview

Presented by:

Kirk Fertitta
Chief Technical Officer
Pacific MindWorks

THE PROBLEM OF MOBILE CODE

The motivation for the new security model in .NET stems largely from the need to deal with what is termed *mobile code*. Simply stated, mobile code is code that you have not explicitly installed on your local machine. This code may come to you via a network or it may be delivered via web pages. The two major sources of mobile code are corporate intranets and the public internet. In the COM (or conventional DLL) era, mobile code was packaged as native code. In the .NET era, mobile code is packaged as assemblies.

The chief problem with mobile code in the COM era is that the level of trust a user could ascribe to a piece of code is too coarse-grained. Specifically, mobile code in COM is either fully trusted or completely untrusted. For instance, consider what a user experiences when browsing to a vendor's web site that uses an ActiveX control. The user is presented with the vendor's digital certificate and asked whether they trust the vendor. The only choices the user has are to answer "Yes" or "No". Answering "No" usually means the website won't work properly. Answering "Yes" could mean many things. It often means the user gets a rich browser experience, but it might also mean the user gets a virus or Trojan horse installed on his system.

A user that allows COM-based mobile code to execute on their system is subject to security attacks because of the way the native Windows loader works. Each DLL that is loaded into a running application becomes an integral part of the process. Practically speaking, you can't tell the difference between code in the DLL and code in the original application. *Every line of code in every DLL in the process runs with the same level of privilege*, irrespective of the DLL's origin. In a world where most users run with admin privileges (you know who you are), this means mobile code in the COM era could easily take control of the local machine. .NET introduces a completely new security model known as Code Access Security to better deal with the problem of mobile code.

CODE ACCESS SECURITY BASICS

The first thing to understand about Code Access Security (henceforth referred to simply as CAS) is that it does not replace the underlying Windows native security system. The security measures enforced by .NET are in addition to, not in lieu of, those enforced by the underlying OS. For instance, if a .NET assembly is granted FileIOPermission to a particular file on disk but the underlying Windows ACL (access control list) denies the current user access to that same file, then the OS "wins" and the running application is denied access. In other words, Windows security has full "veto power" over CAS.

The goal of CAS is to enable a component-based security model that gives users finer-grained control over the privileges granted to code running in an application. As opposed to the process-centric security mod-

el of Windows where all code in an application runs with the privileges of the current user, CAS allows each assembly in an application to operate with its own set of privileges.

When the .NET loader loads an assembly, it gathers *evidence* about the assembly. Evidence can take many forms, but the most common forms of evidence used in CAS security decisions are the location from which the assembly was loaded and the strong name with which the assembly is signed. Once the loader gathers all of the evidence for an assembly, the evidence is fed into the .NET security *policy*. Though the details of .NET security policy management are somewhere involved, you can think of policy as a simple mapping of evidence to *permissions*. Users express what types of protected operations (such as file access, registry access, etc.) they wish to grant to assemblies through policy. In other words, policy allows users to establish the level of *trust* they place in specific assemblies.

Once an assembly's evidence has been pumped through policy, then the assembly operates with whatever permissions were granted to it. In this way, a running application consisting of multiple assemblies will have code that operates at various levels of privilege. This arrangement allows mobile code loaded from a website or another computer on the intranet to operate with less privilege than trusted code loaded locally.

CAS ENFORCEMENT

At runtime, the .NET Framework classes demand permissions before performing sensitive operations. For example, the file I/O classes will perform what is termed a security *demand* for FileIOPermission. If the calling assembly has not been granted FileIOPermission through policy, then the operation will not be allowed and a SecurityException will be thrown. Demands can be performed in code programmatically using the .NET Framework permission classes or they may be performed declaratively using special security attributes.

One problem that this component-based security mechanism introduces is the *luring attack*. This luring attack describes the situation where a less-trusted component uses a more-trusted component to perform protected (and usually malicious) operations. Consider the situation where a less-trusted assembly A downloaded from the internet invokes a method in fully-trusted assembly B loaded from the local machine. The .NET common language runtime (CLR) will check that assembly B has permission to perform the operation, not knowing that the ultimate caller is the less-trusted assembly A. If this was all that the CLR did, then the entire component-based security mechanism would fall apart, since assembly A would essentially have the ability to do anything the trusted assembly B could do. Indeed, the CLR does in fact do more than check the permission of the immediate caller. The CLR will check that all callers in the call stack have the required permission to perform a specific operation. If any caller in the call stack does not have the required permission, then the operation is not allowed and a SecurityException is thrown. This is known as a CAS *stack walk* and its purpose is to prevent the luring attack. As you might imagine, the stack

walk can impose a considerable performance penalty on running code. This is addressed later in the section on the `SuppressedUnmanagedCodeSecurity` attribute.

The last thing to consider in understanding CAS security enforcement is the CLR's *assert* mechanism. From the above discussion, we know that the CLR will demand permissions before performing sensitive operations and this demand will percolate up the stack through the stack walk mechanism ensuring that all callers have the permission being demanded. Consider the case where a method on one of the .NET Framework file I/O classes is called. Ultimately, these classes make calls into the native Win32 file I/O API, which is unmanaged code. The ability to call into unmanaged code is one of the most privileged operations that exist because it requires the caller to have full trust. When the CLR demands the `UnmanagedCode` permission, the stack walk will be performed and all callers will be required to have unmanaged code access. Even if the top-level caller is only trying to perform a benign file-read operation on a public directory, the stack walk mechanism will check that every line of code in the call stack is fully trusted. Once again, it seems the CAS security enforcement mechanism is broken if all callers must be fully trusted to perform even the simplest operations.

Rather than allowing the demand for unmanaged code access to percolate up the call stack, the .NET Framework classes perform an *assert* for the `UnmanagedCode` permission. This assert effectively puts a "cap" on the stack walk, so that callers of the code performing the assert *do not* need to have the permission being asserted. As with the demand operation described earlier, asserts can be performed programmatically with permission classes or declaratively with security attributes.

Following with our example of the .NET Framework file I/O classes, an assert would be performed for the `UnmanagedCode` permission because the file I/O class knows it ultimately needs to call into the unmanaged Win32 API. If the file I/O class itself has unmanaged code access permission (they do, as the Framework is installed with full trust), then the assert will succeed and the stack walk will not progress up the call stack. The file I/O class will then perform a demand for `FileIOPermission`, allowing the stack walk to proceed as normal so that all callers are ensured of having the ability to perform file I/O. This arrangement requires only a single component (the file I/O class) to be fully-trusted, while callers need only have the specific permission directly associated with the desired operation (`FileIOPermission`). The file I/O class is an example of what is termed a *secure gateway*, because it is a trusted component that is providing less trusted components limited indirect access to highly privileged operations.

THE `SuppressUnmanagedCodeSecurity` ATTRIBUTE

While the assert mechanism can be used to shorten the stack walk, the `SuppressUnmanagedCodeSecurityAttribute` can eliminate it completely. This attribute can be applied to methods that need to call into unmanaged code without incurring the performance penalty associated with the normal CAS stack walk, which can be significant. At runtime, no demand for the `UnmanagedCode` permission is performed. Ra-

ther, this demand is performed at just-in-time (JIT) compilation time, which occurs only once – the first time a method is called. This is known as a *link demand*.

A developer that applies this attribute to their methods is implicitly taking responsibility for assuring that their code is safe from attack by malicious code that would seek to exploit the unmanaged code privileges the developer's code has. That being said, this attribute must be used with extreme caution. It is most commonly used in the development of class libraries containing secure gateways to unmanaged code (as described in the previous section).

It is also important to note that this attribute applies only to methods that use PInvoke to call into unmanaged code. It has no effect on COM interop calls.

THE AllowPartiallyTrustedCallers ATTRIBUTE

With all of the aforementioned CAS security machinery in place, one would think our .NET code is secure enough. Nevertheless, Microsoft decided to implement an additional protection feature. By default, partially trusted code cannot call strongly named assemblies (e.g. assemblies with a digital signature). The reason for this is that strongly named assemblies can be deployed in the global assembly cache (GAC) and are thus accessible by mobile code. By contrast, local assemblies without a strong name cannot be seen by typical mobile code. These assemblies can only be loaded by an application in a parent directory (the AppBase).

Imagine if mobile code were allowed to call the following innocent-looking but dangerously poorly written class:

```
public class HopeThisIsntUsedByEvilCodeBecauseItsNotRobust
{
    public string Name;
    public string Password;

    // This entire function consists of
    // horribly bad code you should NEVER EVER use
    public bool IsValidUser()
    {
        SqlConnection conn = new SqlConnection(
            "initial catalog=accounts;user id=sa;password=");
        conn.Open();
        cmd.CommandText = string.Format(
            "select count(*) from users where name='{0}' +
            "and password='{1}'", Name, Password);
        return ((int)cmd.ExecuteScalar()) > 0;
    }
}
```

There are at least five problems with this code that could be exploited by malicious mobile code calling this class.

1. The SA account is used to get to the SQL Server database.
2. There's no password on the SA account.
3. The database connection is opened but never closed.
4. String concatenation is used to build the SQL query.
5. Unfiltered input is used as part of the SQL query.

As an example, because the developer didn't bother to validate the input parameters, an attacker could use this code to delete the entire database, by setting Name to be a string that included SQL statements in it. The point is that, although this class itself is not malicious, it is poorly designed and easily exploited by malicious callers. By not allowing this class to be called from partially-trusted assemblies by default, Microsoft is trying to protect you from publishing poorly written code in shared directories (such as the GAC) where fiendish callers can get at it.

To allow your strongly named assemblies to be called from partially-trusted code, you must take an explicit step -- applying the assembly-level attribute `AllowPartiallyTrustedCallers`. There is no real magic in this attribute and the CLR does not perform any checks to see if your code is really safe or robust. It is simply an explicit statement you are making about your code. It's as if, before publishing your assembly in the GAC, Microsoft wants you to raise your right hand and say, "I've thought about the security implications of my assembly and I've scrubbed it of poorly written code." The `AllowPartiallyTrustedCallers` (henceforth referred to as APTC) attribute is merely a way for Microsoft to force you to think about security before exposing your assemblies to partially trusted mobile code.

RECOMMENDATIONS

SuppressUnmanagedCodeSecurity ATTRIBUTE

It has been suggested that drivers should use the `SuppressUnmanagedCodeSecurity` attribute as a performance optimization. The rationale given at the time was that most drivers will need to call into an I/O library, such as VISA or VISA-COM and that each of those calls would produce a stack walk, thereby degrading performance.

One problem with this approach is that, in applying the `SuppressUnmanagedCodeSecurity` attribute at the driver level, the concomitant security measures must also be implemented at the driver level. That is to say, the entire driver must be developed as a secure gateway so that less privileged clients cannot exploit it. This is a big burden for driver developers and introduces more possibility of security holes than it offers the promise of improved performance. Furthermore, the driver itself must be fully trusted to use this attribute, and I do not believe it should necessarily be the case that drivers operate with full trust. Finally,

drivers developed using VISA-COM would get no benefit from use of this attribute as it has no effect on COM interop stack walks.

An alternate approach would have the I/O libraries themselves use this attribute. After all, these components are the ones that actually need the UnmanagedCode permission – not necessarily the driver. In fact, drivers could be designed to operate with minimal permissions if the I/O libraries made use of the SuppressUnmanagedCodeSecurity attribute. In general, drivers should follow the principal of least privilege. Specifically, drivers (and client applications that call them) should only be required to have enough privilege to perform the sensitive operations they know about. The fact that the I/O libraries require full trust because they make interop calls is an I/O implementation detail that neither the driver nor the client cares about. Drivers can perform demands, asserts, and link demands based upon the privileged operations they know they need to perform. This will necessarily vary from driver to driver, but if drivers mostly perform SCPI string manipulation and I/O library calls, then it is likely drivers can be designed to operate with much less than full trust.

AllowPartiallyTrustedCallers ATTRIBUTE

At the last IVI Foundation meeting, there was some discussion about whether or not to require the APTC attribute on IVI.NET instrument drivers. A few scenarios are possible.

1. Say nothing in the specs regarding the APTC attribute.
2. Require the APTC attribute.
3. Recommend use of the APTC attribute.

I believe it is a good idea to require use of the APTC attribute on IVI.NET drivers. Users may find it surprising if some drivers are available from partially trusted code and others are not. In addition, users generally like assurances that drivers have been designed with some degree of security in mind. However, it is critical, in my opinion, that the requirement to use the APTC attribute not be an empty one. In other words, we must not simply state in the specifications that driver developers need merely apply this attribute to their driver assembly. That would be worse than not applying the attribute at all, because it would give a false sense of security to driver client applications. Rather, the section of the specification that discusses the APTC requirement must also include guidelines and practices for ensuring that the driver is indeed safe to be called from partially trusted clients. While it is impractical to enumerate a detailed list of things driver developers must do and/or not do in their driver implementation to avoid security holes, it should be possible to provide useful practices and highlight common security mistakes. If such guidelines are available in the standard .NET documentation, then that could likely be leveraged in developing the IVI specifications.