CODE ACCESS SECURITY IN MICROSOFT'S .NET

Petar Atanasov

Abstract: This paper introduces basic concepts of code access security, using and implementing security features, as well as types of security syntax and mechanism of checking and requesting specific permissions.

Preface

Role-Based security is at the heart of Microsoft Windows 2000/XP operating systems, but it isn't enough to depend on the code itself and to neglect the skills and awareness of the user.

This security model cares about user access secure resources and any code usually runs under the credentials of the logged on user.

Common scenario for Windows users:

1. Installed ActiveX runs under user's security permission set or it can do pretty much anything with the system that the user can. (Delete, update files, etc.)
2. The user has no idea of what the ActiveX does and most likely it doesn't cross his/her mind, but what matters to the user is that the computer remains secure.

The recent virus "*Sasser*" does not require user interaction for the computer to get infected; simply plug an unprotected machine to the net and in a matter of minutes it becomes infected.

Code Access Security reveals where Role-Based security needs assistance for adequate performance. Code Access Security builds upon Role-Based security.

It provides it with the mechanism of securing the code based on who wrote it and where it came from, or where it is executed (evidences).

These evidences are mapped to the permissions (rights), which can be administered by four different policies, which correspond to the role user represents:

1. Domain Administrator – Enterprise Policy
2. Machine Administrator – Machine Policy
3. Actual User of the machine - User Policy
4. Developer - Application domain Policy

These policies are configurable after the application is deployed and can be modified at any point in time.

One major concept was introduced with CAS - Partially trusted code is code that has been granted only access to the resources it needs to execute successfully and no more.

Code Access Security Basics

Every application that targets the common language runtime must interact with the runtime's security system. When an application executes, it is automatically evaluated and given a set of permissions by the runtime. Depending on the permissions that the application receives, it either runs properly or generates a security exception. The local security settings on a particular computer ultimately decide which permissions code receives. Because these settings can change from computer to computer, a developer can never be sure that his code will receive sufficient permissions to run. This is in contrast to the world of unmanaged development, in which it's not necessary to worry about code's permission to run.

Every developer must be familiar with the following code access security concepts in order to write effective applications targeting the common language runtime:

- Writing type-safe code: To enable code to benefit from code access security, must be used a compiler that generates verifiably type-safe code.
- Imperative and declarative syntax: Interaction with the runtime security system is performed using imperative and declarative security calls. Declarative calls are performed using attributes; imperative calls are performed

using new instances of classes within code. Some calls can be performed only imperatively, while others can be performed only declaratively. Some calls can be performed in either manner.

- Requesting permissions for the code: Requests are applied to the assembly scope, where code informs the runtime about permissions that it either needs to run or specifically does not want. Security requests are evaluated by the runtime when code is loaded into memory. Requests cannot influence the runtime to give code more permissions than the runtime would have given to the code and had the request not been made. However, requests are what code uses to inform the runtime about the permissions it requires in order to run.
- Using secure class libraries: Class libraries use code access security to specify the permissions they require in order to be accessed. The developer should be aware of the permissions required to access any library that code uses and make appropriate requests in itself.

Type-safe code

Type-safe code is code that accesses types only in well-defined, allowable ways. For example, given a valid object reference, type-safe code can access memory at fixed offsets corresponding to actual field members. However, if the code accesses memory at arbitrary offsets outside the range of memory that belongs to that object's publicly exposed fields, it is not type-safe.

JIT compilation performs a process called verification that examines code and attempts to determine whether the code is type-safe. Code that is proven during verification to be type-safe is called verifiably type-safe code. Code can be type-safe, yet not be verifiably type-safe, due to the limitations of the verification process or of the compiler. Not all languages are type-safe, and some language compilers, such as Microsoft Visual C++, cannot generate verifiably type-safe managed code. To determine whether the language compiler is used generates verifiably type-safe code, should be consulted the compiler's documentation. If is used a language compiler that generates verifiably type-safe code only when developer avoids certain language constructs, it might be useful to be used the .NET Framework SDK PEVerify tool to determine whether code is verifiably type-safe.

Code that is not verifiably type-safe can attempt to execute if security policy allows the code to bypass verification. However, because type safety is an essential part of the runtime's mechanism for isolating assemblies, security cannot be reliably enforced if code violates the rules of type safety. By default, code that is not type-safe is allowed to run only if it originates from the local computer. Therefore, mobile code should be type-safe.

Security Syntax

Code that targets the common language runtime can interact with the security system by requesting permissions, demanding that callers have specified permissions, and overriding certain security settings (given enough privileges). There are two different forms of syntax to programmatically interact with the .NET Framework security system: declarative syntax and imperative syntax. Some operations can be done using both forms of syntax while other operations can be performed using only declarative syntax. A developer should be familiar with both forms

Declarative Security

Declarative security syntax uses attributes to place security information into the metadata of code. Attributes can be placed at the assembly, class, or member level, to indicate the type of request, demand, or override developer want to use. Requests are used in applications that target the common language runtime to inform the runtime security system about the permissions that the application needs or does not want. Demands and overrides are used in libraries to help protect resources from callers or to override default security behavior.

In order to use declarative security calls, developer must initialize the state data of the permission object so that it represents the particular form of permission he needs. Every built-in permission has an attribute that is passed a SecurityAction enumeration to describe the type of security operation wanted to perform. However, permissions also accept their own parameters that are exclusive to them.

The following code fragment shows declarative syntax for requesting that code's callers have a custom permission called `MyPermission`. This permission is a hypothetical custom permission and does not exist in the .NET Framework. In this example, the declarative call is placed directly before the class definition, specifying that this permission be applied to the class level. The attribute is passed a SecurityAction.Demand structure to specify that callers must have this permission in order to run.

Visual Basic

```
<MyPermission(SecurityAction.Demand, Unrestricted = True)> Public Class MyClass1

   Public Sub New()
      'The constructor is protected by the security call.
   End Sub
   Public Sub MyMethod()
      'This method is protected by the security call.
   End Sub
   Public Sub YourMethod()
      'This method is protected by the security call.
   End Sub
End Class
```

C#

```
[MyPermission(SecurityAction.Demand, Unrestricted = true)]
public class MyClass
{
   public MyClass()
   {
      //The constructor is protected by the security call.
   }
   public void MyMethod()
   {
      //This method is protected by the security call.
   }
   public void YourMethod()
   {
      //This method is protected by the security call.
   }
}
```

Imperative Security

Imperative security syntax issues a security call by creating a new instance of the permission object wanted to invoke. Developer can use imperative syntax to perform demands and overrides, but not requests.

Before making the security call, developer must initialize the state data of the permission object so that it represents the particular form of the permission he need. For example, when creating a FileIOPermission object, can be used the constructor to initialize the FileIOPermission object so that it represents either unrestricted access to all files or no access to files. Or, developer can use a different FileIOPermission object, passing parameters that indicate the type of access he wants the object to represent (that is, read, append, or write) and what files he wants the object to protect.

In addition to using imperative security syntax to invoke a single security object, developer can use it to initialize a group of permissions called a permission set. For example, this technique is the only way to reliably perform assert calls on multiple permissions in one method. Use the PermissionSet and NamedPermissionSet classes to create a group of permissions and then call the appropriate method to invoke the desired security call.

Developer can use imperative syntax to perform demands and overrides, but not requests. It might be useful to use imperative syntax for demands and overrides instead of declarative syntax when information that is needed in order to initialize the permission state becomes known only at run time. For example, if developer wants to ensure that callers have permission to read a certain file, but he does not know the name of that file until run time, use an imperative demand. Developer might also choose to use imperative checks instead of declarative checks when he needs to determine at run time whether a condition holds and, based on the result of the test, make a security demand (or not).

The following code fragment shows imperative syntax for requesting that code's callers have a custom permission called `MyPermission`. This permission is a hypothetical custom permission and does not exist in the .NET Framework. A new instance of `MyPermision` is created in `MyMethod`, guarding only this method with the security call.

Visual Basic

```
Public Class MyClass1

   Public Sub New()
   End Sub
   Public Sub MyMethod()
      'MyPermission is demanded using imperative syntax.
      Dim Perm As New MyPermission()
      Perm.Demand()
      'This method is protected by the security call.
   End Sub
   Public Sub YourMethod()
      'YourMethod 'This method is not protected by the security call.
   End Sub
End Class
```

C#

```
public class MyClass {
   public MyClass(){
   }
   public void MyMethod() {
       //MyPermission is demanded using imperative syntax.
       MyPermission Perm = new MyPermission();
       Perm.Demand();
       //This method is protected by the security call.
   }
   public void YourMethod() {
       //This method is not protected by the security call.
   }
}
```

Manifest stores Metadata information that can be read without running the assembly; therefore if Developer was using Declarative security to enforce security than Administrator can simply run command-line utility (Permview.exe) to view what Permission is needed to have to run produced code.

In comparison, Imperative is more flexible and is stored as MSIL code, which will be compiled in JIT and given a Security Exception at run-time.

Requesting Permissions

Requesting permissions is the way the developers let the runtime know what code needs to be allowed to do. He request permissions for an assembly by placing attributes (declarative syntax) in the assembly scope of code. When the assembly is created, the language compiler stores the requested permissions in the assembly manifest. At load time, the runtime examines the permission requests, and applies security policy rules to determine which permissions to grant to the assembly. Requests only influence the runtime to deny permissions to code and never influence the runtime to give more permissions to the code. The local administration policy always has final control over the maximum permissions code is granted.

Although code does not have to request permissions in order to compile, there are important reasons the code should always request permissions:

- Requesting permissions increases the likelihood that code will run properly if it is allowed to execute. Code that request a minimal set of permissions will not run unless it receives those permissions. If developer does not identify a minimum set of permissions, code must gracefully handle any and all situations where not being granted some permission might prevent it from executing properly.
- Requesting permissions helps ensure that code is granted only the permissions it needs. If code is not granted extra permissions, it cannot damage the resources protected by those extra permissions, even if it is exploited by malicious code or has bugs that can be leveraged to damage resources. Developer should request only those permissions that his code needs, and no more.
- Requesting permissions lets administrators know the minimum permissions that the application needs so that they can adjust security policy accordingly. Administrators use the Permission View Tool (Permview.exe) to examine assemblies and set up security policy to issue required permissions. If developer does not explicitly

request the permissions that application requires, the Permission View tool cannot return any information about the permissions that produced application requires. If an administrator does not know this information, the application is difficult to administer.

Requesting permissions informs the runtime which permissions the application needs to function or specifically does not want. For example, if the application writes to the local hard disk without using isolated storage, application must have FileIOPermission. If the developer does not request FileIOPermission and the local security settings do not allow the application to have this permission, a security exception is raised when the application attempts to write to the disk. Even if the application can handle the exception, it will not be allowed to write to the disk. This behavior might be frustrating to users if the application is a text-editing program that they have been using for an extended period of time. On the other hand, if the application requests FileIOPermission and the local security settings do not allow the application to have FileIOPermission, the application will generate the exception when it starts and the user will not face the problem of losing any work. Additionally, if the application requests FileIOPermission and if it is a trusted application, the administrator can adjust security policy to allow it to execute from the remote share.

If the code does not access protected resources or perform protected operations, developer does not need to request any permissions. For example, a permission request might not be necessary if the code simply computes a result based on inputs passed to it, without using any resources. If code does access protected resources but does not request the necessary permissions, it might still be allowed to execute, but it could fail at some point during execution if it attempts to access a resource for which it does not have the necessary permission.

To request permissions, developer must know which resources and protected operations code uses, and he must also know which permissions protect those resources and operations. In addition, he needs to keep track of any resources accessed by any class library methods that are called by the solution components.

The following table describes the types of permission requests.

Permission request	Description
Minimum permissions (RequestMinimum)	Permissions code must have in order to run.
Optional permissions (RequestOptional)	Permissions code can use but can run effectively without. This request implicitly refuses all other permissions not specifically requested.
Refused permissions (RequestRefuse)	Permissions that is wanted to ensure will never be granted to the code, even if security policy allows them to be granted.
Perform any of the above requests on built-in permission sets (Requesting Built-in Permission Sets).	Built-in permission sets, including Nothing, Execution, FullTrust, Internet, LocalIntranet, and SkipVerification.
Perform any of the above requests on XML-encoded permission sets (Requesting XML-Encoded Permissions).	XML representation (either a string containing the XML-encoded permission set or the location of an XML file containing the encoded permission set) of a desired permission set.

If developer specifies required permissions (using RequestMinimum), the code will be granted each required permission that security policy allows. The code will be allowed to run only if it is granted all the permissions it requires.

Requesting optional permissions without also requesting required permissions can, in some cases, severely restrict the permissions granted to an assembly. For example, suppose security policy normally grants Assembly A the permissions associated with the Everything named permission set. If the developer of Assembly A requests Permission A as optional and does not request any required permissions, Assembly A will be granted either Permission A (if security policy allows it) or no permissions at all.

Using Secure Class Libraries

Secure library is a class library that uses security demands to ensure that the library's callers have permission to access the resources that the library exposes. For example, a secure class library might have a method for creating files that would demand that its callers have permissions to create files. The .NET Framework comprises secure class libraries.

If code requests and is granted the permissions required by the class library, it will be allowed to access the library and the resource will be protected from unauthorized access; if code does not have the appropriate permissions, it will not be allowed to access the class library, and malicious code will not be able to use developer's code to indirectly access the resource. Even if code has permission to access a library, it will not be allowed to run if code that calls the code does not also have permission to access the library.

Code access security does not eliminate the possibility of human error in writing code; however, if applications use secure class libraries to access protected resources, the security risk for application code is decreased because class libraries are closely scrutinized for potential security problems.

Java – Security Manager

In JDK 1.1, local applications and correctly digitally signed applets were generally trusted to have full access to vital system resources, such as the file system, while unsigned applets were not trusted and could access only limited resources.

A security manager was responsible for determining which resource accesses were allowed.

The Java 2 SDK security architecture is policy-based, and allows for fine-grained access control. When code is loaded, it is assigned "permissions" based on the security policy currently in effect.

Each permission specifies a permitted access to a particular resource, such as "read" and "write" access to a specified file or directory, or "connect" access to a given host and port.

The policy, specifying which permissions are available for code from various signers/locations, can be initialized from an external configurable policy file.

Unless permission is explicitly granted to code, it cannot access the resource that is guarded by that permission.

These new concepts of permission and policy enable the SDK to offer fine-grain, highly configurable, flexible, and extensible access control.

Such access control can now not only be specified for applets, but also for all Java code, including applications, beans, and servlets.

Conclusion

Even with these powerful features code access security doesn't implement the process of creating and envisioning the whole solution, nether in the web nor in the software solutions. The weak implementation of security in the sense of too much references (and invocations) to objects in the security stack could result in performance penalty, while not securing the application could lead to unpredictable results. Code access security gives only a well defined, structured and clean way of performing security, while the security itself is still a priority task number one for the software/web developer.

Bibliography

MSDN

Code Access Security (CAS) and Design Patterns

Security Managers and the Java™ 2 SDK

Author's Information

Petar P. Atanasov – Software developer at Unicontsoft (http://www.unicontsoft.com); phone: ++359-88-7208893; e-mail: ppa@unicontsoft.com