An Overview of Security in the .NET Framework

(also at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/netframesecover.asp, but may be dated)

Dr. Demien Watkins, Project 42
Sebastian Lange, Microsoft Corporation

January 2002

Summary: The fundamental features in the Microsoft .NET Framework security system are profiled, including the ability to confine code to run in tightly constrained, administrator-defined security contexts for a dynamic download and execution, and remote execution, model.

Introduction

Security is one of the most important issues to consider when moving from traditional program development, where administrators often install software to a fixed location on a local disk, to an environment that allows for dynamic downloads and execution and even remote execution. To support this model, the Microsoft .NET Framework provides a rich security system, capable of confining code to run in tightly constrained, administrator-defined security contexts. This paper examines some of the fundamental security features in the .NET Framework.

Many security models attach security to users and their groups (or roles). This means that users, and all code run on behalf of these users, are either permitted or not permitted to perform operations on critical resources. This is how security is modeled in most operating systems. The .NET Framework provides a developer defined security model called role-based security that functions in a similar vein. Role Based Security's principal abstractions are Principals and Identity. Additionally, the .NET Framework also provides security on code and this is referred to as code access security (also referred to as evidence-based security). With code access security, a user may be trusted to access a resource but if the code the user executes is not trusted, then access to the resource will be denied. Security based on code, as opposed to specific users, is a fundamental facility to permit security to be expressed on mobile code. Mobile code may be downloaded and executed by any number of users all of which are unknown at development time. Code Access Security centers on some core abstractions, namely: evidence, policies, and permissions. The security abstractions for role-based security and code access security are represented as types in the .NET Framework Class Library and are user-extendable. There are two other interesting challenges: to expose the security model to a number of programming languages in a consistent and coherent manner and to protect and expose the types in the .NET Framework Class Library that represent resources whose use can lead to security breaches.

The .NET Framework security system functions atop traditional operating system security. This adds a second more expressive and extensible level to operating system security. Both layers are complementing each other. (It is conceivable that an operating system security system can delegate some responsibility to the common language runtime security system for managed code, as the runtime security system is finer grain and more configurable then traditional operating system security.)

This paper provides an overview of security in the .NET Framework and in particular, role-based security, verification, code access security and stack walking. Concepts are exposed with small programming examples. This paper does not cover other runtime security facilities such as cryptography and isolated storage.

As an aside, this paper generally describes the default behavior of the above security features. However, the .NET Framework security system is highly configurable and extensible. This is a major strength of the system, but cannot be covered in detail in this conceptual overview.

Execution Overview

The runtime executes both managed and unmanaged code. Managed code executes under the control of the runtime and therefore has access to services provided by the runtime, such as memory management, just-in-time (JIT) compilation, and, most importantly as far as this paper is concerned, security services, such as the security policy system and verification. Unmanaged code is code that has been compiled to run on a specific hardware platform and cannot directly utilize the runtime. However, when language compilers emit managed code, the compiler output is represented as Microsoft intermediate language (MSIL). MSIL is often described as resembling an object-oriented assembly language for an abstract, stack-based machine. MSIL is said to be object-oriented, as it has instructions for supporting object-oriented concepts, such as the allocation of objects (newobj) and virtual function calls (callvirt). It is an abstract machine, as MSIL is not tied to any specific platform. That is, it makes no assumptions about the hardware on which it runs. It is stack-based, as essentially MSIL executes by pushing and popping values off a stack and calling methods. MSIL is typically JIT-compiled to native code prior to execution. (MSIL can also be compiled to native code prior to running that code. This can help start-up time of the assembly, though typically MSIL code is JIT-compiled at the method level.)

Verification

There are two forms of verification done in the runtime. MSIL is verified and assembly metadata is validated. All types in the runtime specify the contracts that they will implement, and this information is persisted as metadata along with the MSIL in the managed PE/COEFF file. For example, when a type specifies that it inherits from another class or interface, indicating that it will implement a number of methods, this is a contract. A contract can also be related to visibility. For example, types may be declared as public (exported) from their assembly or not. Type safety is a property of code in so much as it only accesses types in accordance with their contracts. MSIL can be verified to prove it is type safe. Verification is a fundamental building block in the .NET Framework security system; currently verification is only performed on managed code. Unmanaged code executed by the runtime must be fully trusted, as it cannot be verified by the runtime.

In order to understand MSIL verification, it is important to understand how MSIL can be classified. MSIL can be classified as invalid, valid, type safe, and verifiable.

Note: It should be stated that the following definitions are more informative than normative. More precise versions can be found in other documentation, such as the ECMA standard:

- *Invalid MSIL* is MSIL for which the JIT compiler cannot produce a native representation. For example, MSIL containing an invalid opcode could not be translated into native code. Another example would be a jump instruction whose target was the address of an operand rather than an opcode.
- *Valid MSIL* could be considered as all MSIL that satisfies the MSIL grammar and therefore can be represented in native code. This classification does include MSIL that uses non-type-safe forms of pointer arithmetic to gain access to members of a type.
- *Type-safe MSIL* only interacts with types through their publicly exposed contracts. MSIL that attempted to access a private member of a type from another type is not type-safe.
- *Verifiable MSIL* is type-safe MSIL that can be proved to be type-safe by a verification algorithm. The verification algorithm is conservative, so some type-safe MSIL might not pass verification. Naturally, verifiable MSIL is also type-safe and valid but not, of course, invalid.

In addition to type-safety checks, the MSIL verification algorithm in the runtime also checks for the occurrence of a stack underflow/overflow, correct use of the exception handling facilities, and object initialization.

For code loaded from disk, the verification process is part of the JIT compiler and proceeds intermittently within the JIT compiler. Verification and JIT compilation are not executed as two separate processes. If, during verification, a sequence of unverifiable MSIL is found within an assembly, the security system checks to see if the assembly is trusted enough to skip verification. For example, if an assembly is loaded from a local hard disk, under the default settings of the security model, then this could be the case. If the assembly is trusted to skip verification, then the MSIL is translated into native code. If the assembly is not trusted enough to skip verification then the offending MSIL is replaced with a stub that throws an exception if that execution path is exercised. A commonly asked question is, "Why not check if an assembly needs to be verified before verifying it?" In general, verification, as it is done as part of the JIT compilation, is often faster than checking to see if the assembly is allowed to skip verification. (The decision to skip verification is smarter than the process described here. For example, results of previous verification attempts can be cached to provide a quick lookup scenario.)

In addition to MSIL verification, assembly metadata is also verified. In fact, type safety relies on these metadata checks, for it assumes that the metadata tokens used during MSIL verification are correct. Assembly metadata is either verified when an assembly is loaded into the Global Assembly Cache (GAC), or Download Cache, or when it is read from disk if it is not inserted into the GAC. (The GAC is a central storage for assemblies that are used by a number of programs. The download cache holds assemblies downloaded from other locations, such as the Internet.) Metadata verification involves examining metadata tokens to see that they index correctly into the tables they access and that indexes into string tables do not point at strings that are longer than the size of buffers that should hold them, eliminating buffer overflow. The elimination, through MSIL and metadata verification, of type-safe code that is not type-safe is the first part of security on the runtime.

Code Access Security

Essentially, code access security assigns permissions to assemblies based on assembly evidence. Code access security uses the location from which executable code is obtained and other information about the identity of code as a primary factor in determining what resources the code should have access to. This information about the identity of an assembly is called evidence. Whenever an assembly is loaded into the runtime for execution, the hosting environment attaches a number of pieces of evidence to the assembly. It is the responsibility of the code access security system in the runtime to map this evidence into a set of permissions, which will determine what access this code has to a number of resources such as the registry or the file system. This mapping is based on administrable security policy.

Default code access security policy has been designed to be secure and sufficient for most application scenarios of managed code. It strongly limits what code from semi- or untrusted environments, such as the Internet or local intranet is capable of doing when executed on the local machine. The code access security default policy model thus represents an opt-in approach to security. Resources are secure by default; administrators need to take explicit action to make the system less secure.

Why we need yet another security paradigm? Code access security revolves around the identity of code, as opposed to user identity. This allows code to run under a single user context in an indefinite number of trust levels. For instance, code coming from the internet can run in restrictive security boundaries, even if the operating system user context in which it runs would allow full access to all system resources.

Let us now look at the main input and output of the code access security system: evidence and permissions.

Permissions

Permissions represent authorization to perform a protected operation. These operations often involve access to a specific resource. In general, the operation can involve accessing resources such as files, the registry, the network, the user interface, or the execution environment. An example of a permission that does not involve a tangible resource is the ability to skip verification.

Note: The System.Security.Permissions.SecurityPermission class contains a flag that determines whether recipients of the permission instance are allowed to skip verification. The SecurityPermission class contains other similar permission flags that cover core runtime technology that could open security vulnerability if misused, such as the ability to control the evidence given to assemblies running in a specific application domain. Core runtime technologies are protected by demanding callers to have the requisite SecurityPermission class with the appropriate permission flags set.

The fundamental abstraction of a permission is the IPermission interface, and it requires a specific permission type to implement a set of standard permission operations, such as returning the union or subset with other permission instances of the same permission type.

Permissions can be collated into a set of permissions, representing a statement of access rights to a variety of resources. The class System.Security.PermissionSet represents the collection of permissions. Methods on this class include Intersect and Union. These methods take another PermissionSet as a parameter and provide a PermissionSet that is either the union or intersection of all the permissions in both sets. (A collection of permissions in the runtime is represented as a simple, non-ordered set.) With these facilities the security system can work with permission sets and not have to understand the semantics of each individual type of permission. This allows developers to extend the permission hierarchy, without needing to modify the functionality of the security engine.

Note: Each individual permission type must derive from the IPermission interface that requires any permission types to implement standard permission operations such as the union, intersection, and subset and demand methods. Permission types are free to implement the semantics specific to the type of permission state they contain. For instance, a permission containing file names will be intersected different from one containing a simple Boolean state. When permission set A is intersected with permission set B, and A and B contain different instances of the same permission type X, then the intersection methods on an instance of X is called by the permission set class A without having to know anything about the semantics of X.

Based on the evidence presented to the security system at assembly load time, the security system grants a permission set that represents authority to access various protected resources. Conversely, resources are protected by a permission demand that triggers a security check to see that a specific permission has been granted to all callers of the resource; if the demand fails, an exception is raised. (There is a specific security check, called a link demand, which only checks the immediate caller. Typically though, the whole call stack of callers is checked.)

Evidence

Whenever an assembly is loaded into the runtime, the hosting environment presents the security system with evidence for the assembly. Evidence constitutes the input to the code access security policy system that determines what permissions an assembly receives.

A number of classes that ship with the .NET Framework are used as standard forms of evidence in the security system:

- *Zone:* The same concept as zones used in Internet Explorer.
- *URL:* A specific URL or file location that identifies a specific resource, such as http://www.microsoft.com/test
- *Hash:* The hash value of an assembly generated with hash algorithms such as SHA1.
- *Strong Name:* The strong name signature of an assembly. Strong names represent a versioned, cryptographically strong way to refer and identify an assembly or all assemblies of a particular signing party. For more information, please consult the .NET Framework SDK.

- *Site:* The site from which the code came. A URL is more specific than the notion of a site; for example, www.microsoft.com is a site.
- *Application Directory:* The directory from which the code is loaded.
- *Publisher certificate:* The Authenticode digital signature of the assembly.

Note: Theoretically, any managed object can constitute evidence. The above are just types that have corresponding membership conditions in the .NET Framework, and can thus be integrated into security policy without having to write custom security objects. See below for more detail on security policy and code groups.

The following program is a simple example of the evidence passed to the runtime security system when an assembly is loaded. In this case, mscorlib is the loaded assembly; this is the assembly that contains many of the runtime types, such as Object and String.

```
using System;
using System.Collections;
using System.Reflection;
using System.Security.Policy;

namespace AssemblyEvidence
{
   class Class1
   {
      static void Main(string[] args)
      {
         Type t = Type.GetType("System.String");
         Assembly a = Assembly.GetAssembly(t);
         Evidence e = a.Evidence;
         IEnumerator i = e.GetEnumerator();
         while(i.MoveNext())
            Console.WriteLine(i.Current);
      }
   }
}
```

The output from the program shows what evidence was passed to the security system for this assembly. The output below has been edited for brevity. The security system takes this evidence and produces a permission set for the assembly based on security policy as set by the administrator.

```
<System.Security.Policy.Zone version="1">
   <Zone>MyComputer</Zone>
</System.Security.Policy.Zone>
<System.Security.Policy.Url version="1">
   <Url>
     file:///C:/winnt/microsoft.net/framework/v1.0.2728/mscorlib.dll
   </Url>
</System.Security.Policy.Url>
<StrongName version="1"
            Key="00000000000000000400000000000000"
            Name="mscorlib"
            Version="1.0.2411.0"/>
<System.Security.Policy.Hash version="1">
   <RawData>4D5A90000300000004000000FFFF0000B8000000000000...
     0000000000000000000000000000000000000000000000000000
   </RawData>
</System.Security.Policy.Hash>
```

Security Policy

Administrable security policy determines the mapping between the assembly evidence that a host provides for an assembly, and the set of permissions granted to an assembly. The System.Security.SecurityManager class implements this mapping functionality. You can therefore think of the code access security policy system as a function with two input variables (evidence, and administrable security policy) and an assembly specific permission set as output value. This section focuses on the administrable security policy system.

There are a number of configurable policy levels the Security Manager recognizes, namely:

- Enterprise Policy Level
- Machine Policy Level
- User Policy Level
- Application Domain Policy Level

The enterprise, machine, and user policy levels are configurable by security policy administrators. The application domain policy level is programmatically configurable by hosts.

When the security manager needs to determine the set of permissions that an assembly is granted by security policy, it starts with the enterprise policy level. Supplying the assembly evidence to this policy level will result in the set of permissions granted from that policy level. The security manager typically continues to collect the permission sets of the policy levels below the enterprise policy in the same fashion. These permission sets are then intersected to generate the policy system permission set for the assembly. All levels must allow a specific permission before it can make it into the granted permission set for the assembly. For example, if, during the evaluation of an assembly, the enterprise policy level does not grant a specific permission, regardless of what the other levels specify, the permission is not be granted.

Note: There are corner cases in which a policy level, such as the enterprise policy level, can contain a directive to not evaluate any policy levels below it, such as the machine and user policy levels. In that case, neither machine nor user policy level would produce a permission set and is not taken into account in the calculation of the set of granted permissions for an assembly.

Developers of an assembly can impact the permission calculation at runtime of the assembly. Although an assembly cannot simply take the permissions it needs to run, it can state a minimum required set of permissions or refuse certain permissions. The security manager ensures that the assembly will run only if the requested permission, or permissions, is part of the set of permissions granted by the policy level hierarchy. Conversely, the security manager ensures that the assembly does not receive any permissions it refuses to get. Minimum required, refused, or optional permission requests are put on an assembly by the developer of the assembly using security custom attributes. See the Declarative and Imperative Style section below or the .NET Framework SDK for more information.

The process of determining the actual set of granted permissions to an assembly is a three-fold procedure:

1. Individual policy levels evaluate the evidence of an assembly and generate a policy level specific granted set of permissions.
2. The permission sets calculated for each policy level are intersected with each other.
3. The resulting permission set is compared with the set of permissions the assembly declared necessary to run, or refuses and the permissions grant is modified accordingly.

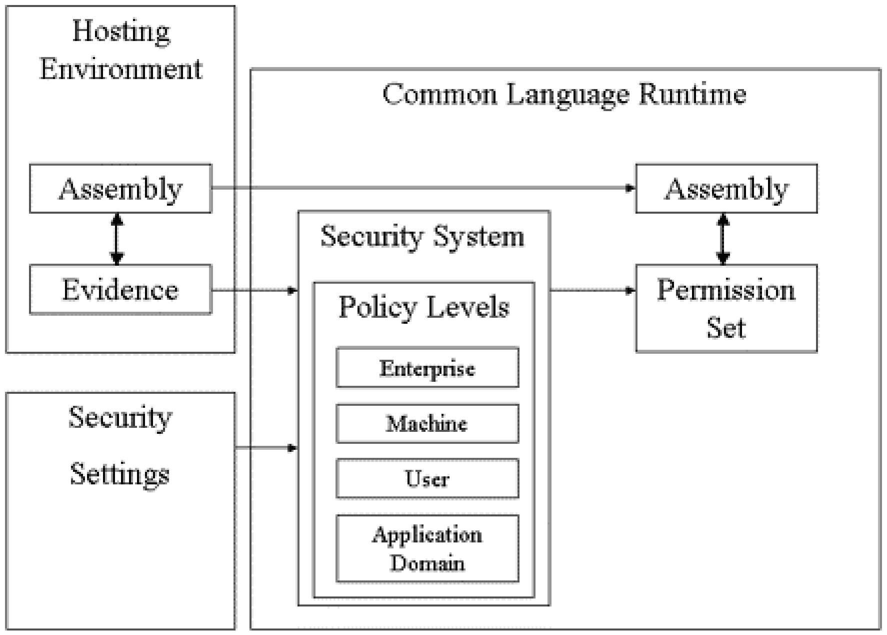


Figure 1. Calculation of the set of granted permissions of an assembly

Figure 1 shows you the big picture. The host of the runtime supplies evidence about an assembly that acts as one of the inputs for the calculation of the permission set the assembly receives. The administrable security policy (enterprise, machine and user policy), referred to above as security settings, is the second input that determines the calculation of a permission set of the assembly. The security policy code (contained in the SecurityManager class), then traverses the policy level settings given the evidence of the assembly and produces the permission set that represent an assembly's set of rights to access protected resources.

How is each of the policy levels administered? Policy levels express a self-contained, configurable security policy unit - each level mapping assembly evidence to a set of permissions. Each policy level has a similar architecture. Each level consists of three constituents that are used in combination to express the configuration state of a policy level:

- A tree of Code Groups
- A list of Named Permissions
- A list of Policy Assemblies

These constituents of all policy levels are now explained in detail.

Code Group

The heart of each policy level is a tree of code groups. It expresses the configuration state of the policy level. Essentially, a code group is a conditional expression and a permission set. If an assembly satisfies the conditional expression, then it is granted the permission set. The set of code groups for each policy level is arranged in a tree. Every time a conditional expression evaluates to true, the permission set is granted and the traversal of that branch continues. Whenever a condition is not satisfied, the permission set is not granted and that branch is not examined any further. For example, a tree of code groups that works like the following scenario.

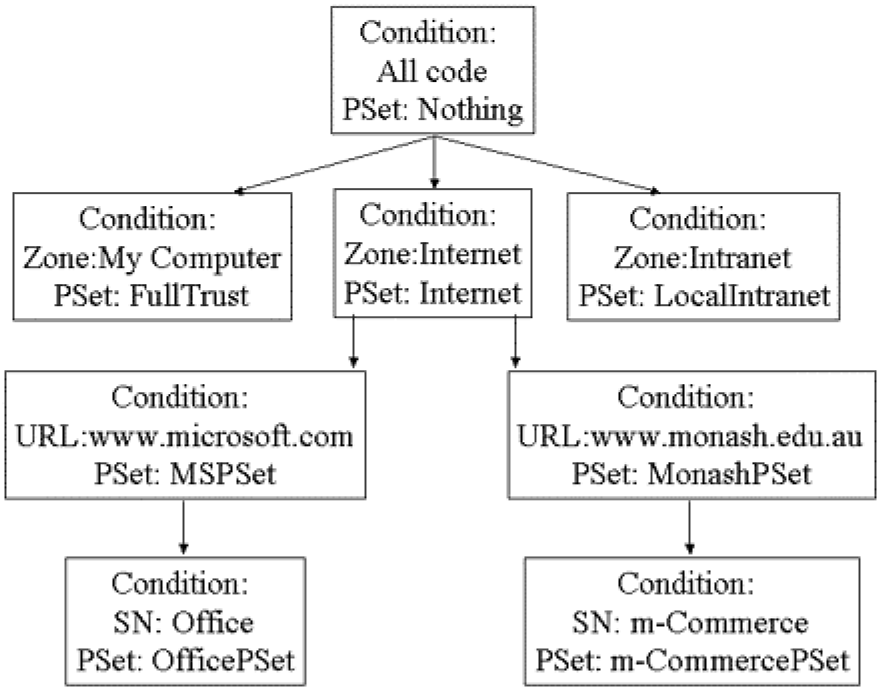


Figure 2. Policy level as a tree of code groups

Note: Here we are discussing only the code group semantics of the code group types used to implement default security policy. It is possible to include custom-authored code groups having a wildly different semantics than what is described here. Again, the security system is fully extendable and thus offers an indefinite potential for the introduction of new policy evaluation semantics.

Assume there is an assembly with the following evidence: it came from www.monash.edu.au and, as it came from the m-Commerce Center at Monash University, it has a strong name of mCommerce.

The code group tree traversal will proceed as follows:
The root node has a condition of 'all code' that is satisfied by any code. The permission for all code is granted to our assembly, this permission set is called "Nothing" and permits code no permissions. The next code group checked is the one requiring that the code be loaded from My Computer. As this condition fails, the permission set is not granted and no subordinated nodes of this condition are checked. We then return to the last successfully granted code group, all code in this case, and continue checking its subordinate nodes. The next code group is Zone: Internet. As our code was downloaded from the Internet, this condition is satisfied, the permission set, possibly the Internet permission set, is granted and you can continue on to the next subordinate code group in this branch. The next code group has a condition of Url: stating that the code came from www.microsoft.com. This condition fails, since the code came from www.monash.edu.au. At this point we return to the Zone: Internet code group and look for other nodes beneath it. We find a node for URL: www.monash.edu.au. As this condition is satisfied, we receive the MonashPSet permission set. Next we find a node for Strong

Name: m-Commerce center. As this condition is satisfied, we receive the m-Commerce permission set. As there are no code groups below this level we return to last code group that a condition matched and that has subordinate code groups and continue.

Eventually the conditions satisfied and the granted permission sets from this policy level would include:

- Condition: All code, Permission Set: Nothing
- Condition: Zone: Internet, Permission Set: Internet
- Condition: URL: www.monash.edu.au, Permission Set: MonashPSet
- Condition: Strong Name: m-Commerce, Permission Set: m-CommercePSet

All permission sets that are found to apply to a specific assembly in a policy level are typically union-ed to yield the total set of permissions granted by that policy level.

Inspecting the code group tree of a policy level is very simple. Appendix A describes a Microsoft Management Console snap-in that provides a visual interface for viewing and modifying code group hierarchies (and all other configurable constituents of a policy level, see below).

Named Permission Sets

A policy level contains a list of named permission sets. Each permission set represents a statement of trust to access a variety of protected resources. Named permission sets are the permission sets that a code group references by name. If the condition of a code group is met, the referenced named permission set is granted (see example above). Examples of predefined named permission sets are:

- *FullTrust:* Allows unrestricted access to system resources.
- *SkipVerification:* Allows an assembly to skip verification.
- *Execution:* Allows code to execute.
- *Nothing:* No permissions. Not granting the permission to execute effectively stops code from running
- *Internet:* A set of permissions deemed appropriate for code coming from the Internet. Code will not receive access to the file system or registry, but can do limited user interface actions as well as use the safe file system called Isolated Storage.

To view permission sets of a policy level, simply open the policy level node in the GUI tool mentioned in Appendix A and open the permission set folder.

Below is a little sample program that lists all known named permission sets at all policy levels.
The following program displays the named permission set lists at all policy levels. The application is a C# program run from the local disk, so it receives a fairly powerful permission set from default policy settings.

```
using System;
using System.Collections;
using System.Security;
using System.Security.Policy;

namespace SecurityResolver
{
   class Sample
   {
      static void Main(string[] args)
      {
         IEnumerator i = SecurityManager.PolicyHierarchy();
         while(i.MoveNext())
         {
            PolicyLevel p = (PolicyLevel) i.Current;
            Console.WriteLine(p.Label);

            IEnumerator np = p.NamedPermissionSets.GetEnumerator();
            while (np.MoveNext())
            {
               NamedPermissionSet pset = (NamedPermissionSet)np.Current;
               Console.WriteLine("\tPermission Set: \n\t\t Name: {0} \n\t\t Description {1}",
                                   pset.Name, pset.Description);
            }
         }
      }
   }
}
```

The output of the program is shown below. The output has been edited for brevity and clarity.

```
Enterprise
  Permission Set:
    Name: FullTrust
    Description: Allows full access to all resources
  Permission Set:
    Name: LocalIntranet
    Description: Default rights given to applications
                 on your local intranet
  ...
Machine
  Permission Set:
    Name: Nothing
    Description: Denies all resources, including the right to execute
  ...
User
  ...
    Name: SkipVerification
    Description: Grants right to bypass the verification
  Permission Set:
    Name: Execution
    Description: Permits execution
  ...
```

Policy Assemblies

During security evaluation, other assemblies might need to be loaded to be used in the policy evaluation process. For example, an assembly can contain a user-defined permission class part of a permission set handed out by a code group. Of course, the assembly containing the custom permission also needs to be evaluated. If the assembly of the custom permission is granted the permission set containing the custom permission it itself implements, then a circular dependency ensues. To avoid this, each policy level contains a list of trusted assemblies that it needs for policy evaluation. The list of required assemblies is naturally referred to as the list of "Policy Assemblies", and contains the transitive closure of all assembly required to implement security policy at that policy level. Policy evaluation for all assemblies contained in that list is short circuited to avoid the occurrence of circular dependencies. The list can be modified using the GUI admin tool mentioned in Appendix A.

This completes our examination of the configurable constituents of each policy level: a tree of code groups, list of named permission sets and a list of policy assemblies. It is now time to see how the permission grant set derived from the security policy state as instantiated by the configuration of each policy level, interfaces with the infrastructure of security enforcement. In other words, so far we have only looked at how assemblies get to have granted permission sets. Without an infrastructure that requires assemblies to have a certain level of permissions the security system would be as useful as a one legged chair. In fact, the technology that makes security enforcement possible is a security stack walk.

Stack Walk

Stack walks are an essential part of the security system. A stack walk operates in the following manner. Every time a method is called a new activation record will be put on the stack. This record contains the parameters passed to the method, if any, the address to return to when this function completes and any local variables. As a program executes, the stack grows and shrinks as functions are called. At certain stages during execution, the thread might need to access a system resource, such as the file system. Before allowing this access the protected resource may demand a stack walk to verify that all functions in the call chain have permission to access the system resource. At this stage a stack walk will occur and typically each activation record is checked to see that callers do indeed have the required permission. In contrast to a full stack walk the code access security system also allows developers to annotate resources with a link time check that checks only the immediate caller.

Modifying the Stack Walk

At any stage during execution, a function might need to check the permissions of its callers before it accesses a particular resource. At this stage, the function can demand a security check for a specific permission or set of permissions. This triggers a stack walk, the result of which is either that the function continues if all callers have the permission granted or an exception is thrown if the callers do not have the permission, or permissions, demanded. The following diagram represents this process.

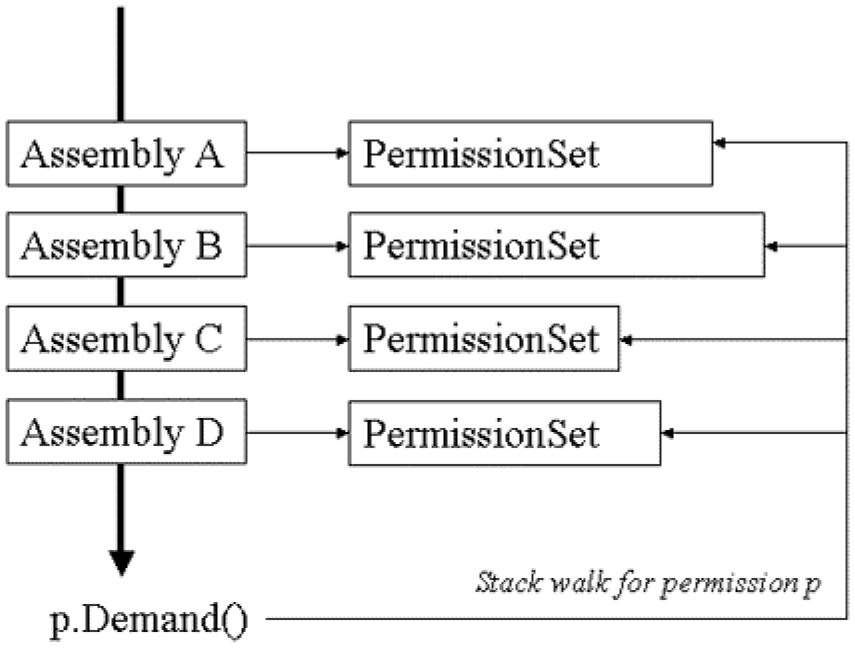


Figure 3. Stack walk example

Functions can choose to modify the stack walk, and there are a few mechanisms to do this. First, one function might need to vouch for the functions that called it. In this case it can assert a specific permission. If a stack walk occurs looking for the asserted permission, then when the activation record for this function is checked for the permission, the check will succeed and the stack walk will terminate if the function has the permission that it asserts. An assert is itself a protected operation, as it will open the access to a protected resource to all callers of the function asserting permission to access that resource. At runtime, the security system therefore checks if the assembly containing the function that asserts itself has the permission it tries to assert.

Another way to modify the stack walk is for a function to deny permissions. This situation can occur when a function knows that it should not be able to access a resource and it denies this permission. PermitOnly provides similar functionality to deny in that it causes a stack walk to fail. However, while deny specifies a set of permissions that would cause the stack walk to fail, PermitOnly specifies the set of permissions required to continue the stack walk.

Note: You should use caution when using Deny stack modifiers. If an earlier stack frame is an assert, the deny modifier will be ignored. Also, denying path-based permissions is difficult at best since there are often various different path strings that equivalently point to the same location. Denying one particular path expression will still leave open others.

There is one last point that is important to know. A stack frame can only have one Deny, one PermitOnly and one Assert in effect at any time. For example, if developers need to assert a number of permissions, they should create a PermissionSet to represent that set and do a single assert. There are methods to remove the current PermissionSet setting for a single stack walk modifier so that another permission set can be registered. An example of such a method is System.Security.CodeAccessPermission.RevertPermitOnly().

The sample below demonstrates the various stack modification techniques introduced above:

```
using System;
using System.Security;
using System.Security.Permissions;

namespace PermissionDemand
{
   class EntryPoint
   {
      static void Main(string[] args)
      {
         String f = @"c:\System Volume Information";
         FileIOPermission p = new FileIOPermission(FileIOPermissionAccess.Write, f);
         p.Demand();
         p.Deny();
         p.Demand();
         CheckDeny(p);
         p.Assert();
         CheckDeny(p);
      }
      static void CheckDeny(FileIOPermission p)
      {
         try
         {
            p.Demand();
         }
         catch(SecurityException)
         {
            Console.WriteLine("Demand failed");
         }
      }
   }
}
```

The previous program produces the following output, which appears very counterintuitive at first:

```
Demand failed
Demand failed
```

The first Demand in the code sample above succeeds, even though it is accessing a restricted system directory. Remember that the runtime security system operates above the underlying operating system settings. It is therefore possible to have the runtime security policy grant access to directories that the managed code actually trying to access it will get operating system access violations for. The next Demand, straight after the Deny also succeeds. The demanding function's activation record is not checked when a Demand is done, only its callers. Therefore, although the function has denied access this is not detected by the Demand. The call to CheckDeny and the subsequent Demand does fail. Now the Deny in the previous method is checked, as it is in a caller's stack frame. Next we return to Main and do an Assert. Here a permission has been asserted that has also been denied in this stack frame. When we enter CheckDeny, the Demand again raises an exception, but why? Essentially, a Deny overrides an Assert; the reason is that the Deny permission set is always checked before the Assert permission set.

In summary, the ability for managed resources to cause a managed security stack walk is the runtime security system way of protecting resources. A set of granted permissions an assembly received from the grant calculations run over each policy level is checked against the permissions demanded by a resource. If the latter forms a subset of the former, then the protected resource can be accessed. This subset check is undertaken for all callers in the call chain to a managed resource, unless the stack has been modified as described above. Security stack walks therefore bring together both aspects of the runtime security system:
1) the configurable mapping between evidence and permissions, and
2) the protection of resources by enforcing all callers to possess a certain level of permissions.

There are actually two different ways to programmatically express stack walk demands and stack modification operations, declarative and imperative security.

Declarative and Imperative Style

The .NET Framework allows developers to express security constraints in two styles. Expressing security constraints in the Declarative style means using the custom attribute syntax. These annotations are persisted in the metadata for the type, effectively being baked into the assembly at compile time. An example of declarative style security is:

```
[PrincipalPermissionAttribute(SecurityAction.Demand, Name=@"culex\damien")]
```

Expressing security requirements in the Imperative style means creating instances of Permisson objects at run time and invoking methods on them. An example of imperative style security from a previous sample in this paper was:

```
FileIOPermission p = new FileIOPermission(FileIOPermissionAccess.Write, f);
p.Demand();
```

There are a few reasons for choosing one style over another. First, all security actions can be expressed using the declarative style; this is not true for the imperative style. However, the declarative style requires that all security constraints be expressed at compile time and only allows annotation at the full method, class or assembly scope. The imperative style is more flexible as it allows constraints to be expressed at runtime, such as for a subset of the possible execution paths in a method. A side effect of persisting declarative security requirements in the metadata is that tools are able to extract this metadata information and provide functionality based on this fact. For example, a tool may display a list of the declarative security attributes set on an assembly. This is not possible for imperative security. Developers need to be aware and fluent with both.

A Word of Warning

As code executed from a local hard disk by default policy receives significantly more trust than code executed from any other location, downloading code, storing it to disk and executing it has far different semantics than executing code from a remote location. This is counterintuitive to previous systems where, for example, browsers have chosen to download code rather than execute it remotely. The assumption here was that after the code was downloaded it would be inspected, such as with virus scanners, before execution. With code access security, this scenario is reversed, executing the code as remote code provides significantly more security using default security policy. However, at this time, this can place a burden on the system or user to know the difference between managed and unmanaged code. Finally, there is another aspect of the runtime security system. This system should be more familiar to users of legacy security systems as it is based on user identity: role-based security.

Role-Based Security

The code access security system introduced so far fundamentally revolves around the identity of code, and not users or roles. However, there is still a need to be able to express security settings based on user identities. The runtime security system therefore also ships with role-based security features.

Role-based security utilizes the concepts of users and roles, which is similar to the implementation of security in many current operating systems. Two core abstractions in role-based security are Identity and Principal. Identity represents the user on whose behalf the code is executing. It is important to remember that this could be a logical user as defined by the application or developer and not necessarily the user as seen by the operating system. A Principal represents the abstraction of a user and the roles in which a user belongs. Classes representing a user’s Identity implement the IIdentity interface. A generic class providing a default implementation of this interface within the Framework is GenericIdentity. Classes representing Principals implement the IPrincipal interface. A generic class providing a default implementation of this interface within the Framework is GenericPrincipal.

At run time each thread has one and only one current principal object associated with it. Code may of course access and change this object, subject to security requirements. Each Principal has one and one only Identity object. Logically the run time structure of the objects resembles the following:

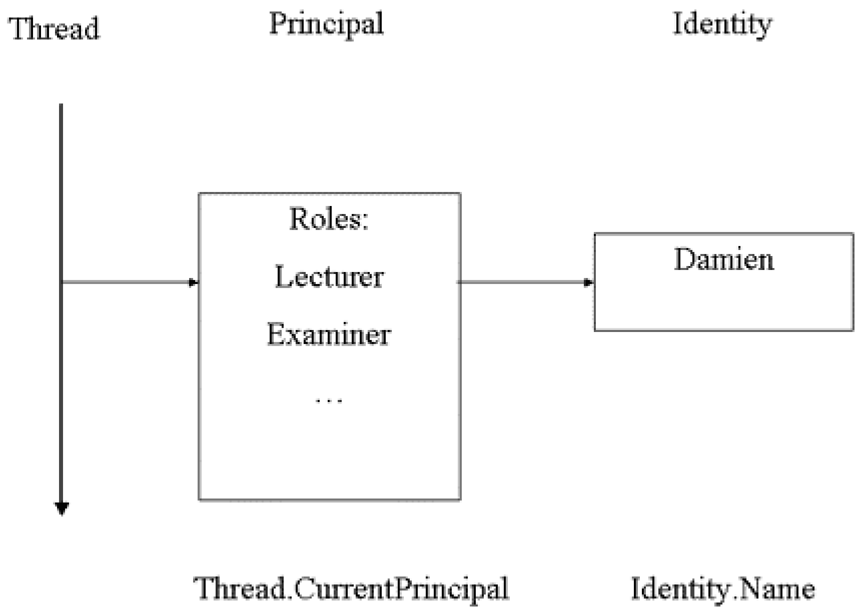


Figure 4. Role-based security structure

The following program demonstrates how developers can use the generic classes. In this case the developer is supplying the security model. The name "Damien" and the roles "Lecturer" and "Examiner" are not, for example, tied to any user or roles that the operating system can support.

```
using System;
using System.Threading;
using System.Security;
using System.Security.Principal;

namespace RoleBasedSecurity
{
   class Sample
   {
      static void Main(string[] args)
      {
         String [] roles = {"Lecturer", "Examiner"};
         GenericIdentity i = new GenericIdentity("Damien");
         GenericPrincipal g = new GenericPrincipal(i, roles);
         Thread.CurrentPrincipal = g;
         if(Thread.CurrentPrincipal.Identity.Name == "Damien")
            Console.WriteLine("Hello Damien");
         if(Thread.CurrentPrincipal.IsInRole("Examiner"))
            Console.WriteLine("Hello Examiner");
         if(Thread.CurrentPrincipal.IsInRole("Employee"))
            Console.WriteLine("Hello Employee");
      }
   }
}
```

The program produces the following output:

```
Hello Damien
Hello Examiner
```

It is also possible to use the Microsoft Windows security model if developers wish. In this situation, users and roles are tied to those on the hosting machine therefore accounts may need to be created on the hosting system. The following example uses the local machine's user accounts. The example also uses some syntactic sugar; the .NET Framework class PrincipalPermissionAttribute effectively encapsulates the calls to methods such as "IsInRole" to allow developers to use a simplified syntax.

```
namespace RoleBased
{
   class Sample
   {
      [PrincipalPermissionAttribute(SecurityAction.Demand, Name=@"culex\damien")]
      public static void UserDemandDamien()
      {
         Console.WriteLine("Hello Damien!");
      }
      [PrincipalPermissionAttribute(SecurityAction.Demand, Name=@"culex\dean")]
      public static void UserDemandDean()
      {
         Console.WriteLine("Hello Dean!");
      }
      static void Main(string[] args)
      {
         AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.WindowsPrincipal);
         try
         {
            UserDemandDamien();
            UserDemandDean();
         }
         catch(Exception)
         {
            Console.WriteLine("Exception thrown");
         }
      }
   }
}
```

The PrincipalPermissionAttribute ensures that a runtime check occurs each time the methods UserDemandDamien() and UserDemandDean() are called. Naturally, the program can be executed by Dean or Damien or someone else, so the security check should fail on at least one of the two method calls, if not both. The first line of Main sets the Principal Policy to that of Windows, the operating system on which this example was executed. The program produces the following output when executed by the user "culex\damien":

```
Hello Damien!
Exception thrown
```

Summary

Security is a fundamental and built-in aspect of the .NET Framework. This paper attempts to provide an overview of the Security System. Major concepts to take away from this paper are:

- The security system is extensible, with many concepts represented as types in the .NET Framework that developers can extend and tailor to satisfy their own needs.
- The security system provides different types of security models, specifically role-based and evidence-based. These different models address different needs and complement each other.
- Code access security revolves around the identity of code, and therefore allows the execution of code in semi-trusted security context, even if the operating system user context the code is run in gives it administrative rights on the machine

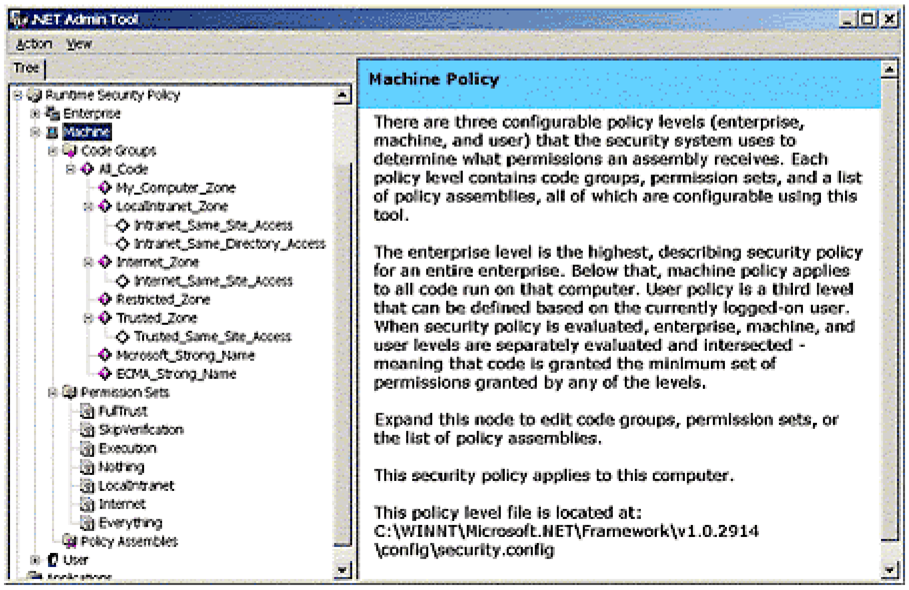
This paper has not covered some aspects of the Security System, such as Cryptography, nor has it covered any aspects in complete detail. Please refer to white papers specifically about these subjects for the missing details.

Acknowledgements

We would gratefully like to acknowledge the help and assistance received while writing this paper from Brian Pratt, Loren Kohnfelder and Matt Lyons.

Appendix A: MsCorCfg.msc

There is a Microsoft Management Console snap-in that allows visual manipulation of Code Access Security policy. The following figure shows the interface to this snap-in highlighting some of the concepts covered in this paper.



There are three configurable policy levels (enterprise, machine, and user) that the security system uses to determine what permissions an assembly receives. Each policy level contains code groups, permission sets, and a list of policy assemblies, all of which are configurable using this tool.

The enterprise level is the highest, describing security policy for an entire enterprise. Below that, machine policy applies to all code run on that computer. User policy is a third level that can be defined based on the currently logged-on user. When security policy is evaluated, enterprise, machine, and user levels are separately evaluated and intersected - meaning that code is granted the minimum set of permissions granted by any of the levels.

Expand this node to edit code groups, permission sets, or the list of policy assemblies.

This security policy applies to this computer.

This policy level file is located at:
C:\WINNT\Microsoft.NET\Framework\v1.0.2914
\config\security.config

Figure 5. Microsoft Management Console snap-in interface

You can access this tool in Control Panel by clicking Administrative Tools and then clicking the Microsoft .NET Framework Configuration shortcut.