21a

Code Access Security

Code Access Security (CAS) allows the CLR to create a locked-down or *sandboxed* environment that prevents code from performing certain kinds of operations (such as reading operating system files, performing reflection, or creating a user interface). The sandboxed environment created by CAS is referred to as a *partial trust* environment, whereas the normal unrestricted environment is referred to as *full trust*.

CAS was considered strategic in the early days of .NET, as it enabled the following:

- Running C# ActiveX controls inside a web browser (like Java applets)
- Lowering the cost of shared web hosting by allowing multiple web sites to run inside the same .NET process
- Deploying permission-restricted ClickOnce applications via the Internet

The first two are no longer relevant, and the third was always of dubious value, because end-users are unlikely to know or understand the consequences of restricted permission sets prior to installation. And while there are other use-cases for CAS, they are more specialized. A further problem is that the sandbox created by CAS is not entirely robust: Microsoft stated in 2015 that CAS should not be relied upon as a mechanism for enforcing security boundaries (and CAS has been largely excluded from .NET Standard 2.0). This is in spite of the improvements to CAS introduced with CLR 4 in 2010.

Sandboxing that does not rely on CAS is still well and alive: UWP applications run in a sandbox, as do SQL CLR libraries. These sandboxes are enforced by the operating system or hosted CLR, and are more robust than CAS sandboxes, as well as being simpler to understand and manage. Operating system security also works with unmanaged code, so a UWP app cannot read/write arbitrary files, whether written in C# or C++.

For these reasons, we omitted CAS from *C# 7.0 in a Nutshell*, and have published the material on CAS is this addendum. If you're a library author, you may still need to cater for partial trust environments to support older platforms.

The types covered in this chapter are defined in the following namespaces:

```
System.Security;
System.Security.Permissions;
System.Security.Principal;
System.Security.Cryptography;
```

Permissions

We discussed permissions in Chapter 21 of C# 7.0 in a Nutshell, in the context of identity and role security. We're now going to revisit them in the context of CAS.

The Framework uses permissions for both sandboxing and authorization. A *permission* acts as a gate that conditionally prevents code from executing. Sandboxing uses *code access* permissions; authorization uses *identity* and *role* permissions.

Although both follow a similar model, they feel quite different to use. Part of the reason for this is that they typically put you on a different side of the fence: with code access security, you're usually the *untrusted* party; with identity and role security, you're usually the *untrusting* party. Code access security is most often forced upon you by the CLR or a hosting environment such as ASP.NET or ClickOnce, whereas authorization is usually something you implement to prevent unprivileged callers from accessing your program.

CodeAccessPermission and PrincipalPermission

There are essentially two kinds of permissions:

`CodeAccessPermission`
: The abstract base class for all code access security (CAS) permissions, such as `FileIOPermission`, `ReflectionPermission`, or `PrintingPermission`

`PrincipalPermission`
: Describes an identity and/or role (e.g., "Mary" or "Human Resources")

The term *permission* is somewhat misleading in the case of `CodeAccessPermission`, because it suggests something has been granted. This is not necessarily the case. A `CodeAccessPermission` object describes a *privileged operation*.

For instance, a `FileIOPermission` object describes the privilege of being able to `Read`, `Write`, or `Append` to a particular set of files or directories. Such an object can be used in a variety of ways:

- To verify that you and all your callers have the rights to perform these actions (`Demand`)
- To verify that your immediate caller has the rights to perform these actions (`LinkDemand`)
- To temporarily escape a sandbox and `Assert` your assembly-given rights to perform these actions, regardless of callers' privileges

> You'll also see the following security actions in the CLR: `Deny`, `RequestMinimum`, `RequestOptional`, `RequestRefuse`, and `PermitOnly`. However, these (along with link demands) have been deprecated or discouraged since Framework 4.0, in favor of the new *transparency* model (see "The Transparency Model").

`PrincipalPermission` is much simpler. Its only security method is `Demand`, which checks that the specified user or role is valid given the current execution thread.

IPermission

Both `CodeAccessPermission` and `PrincipalPermission` implement the `IPermission` interface:

```
public interface IPermission
{
  void Demand();
  IPermission Intersect (IPermission target);
  IPermission Union (IPermission target);
  bool IsSubsetOf (IPermission target);
  IPermission Copy();
}
```

The crucial method here is `Demand`. It performs a spot-check to see whether the permission or privileged operation is currently permitted, and it throws a `SecurityException` if not. If you're the *untrusting* party, *you* will be `Demand`ing. If you're the *untrusted* party, code that you *call* will be `Demand`ing.

We saw in Chapter 21 of C# 7.0 in a Nutshell how to use a `PrincipalPermssion` to `Demand` that only Mary can run management reports:

```
new PrincipalPermission ("Mary", null).Demand();
// ... run management reports
```

In contrast, suppose your assembly was sandboxed such that file I/O was prohibited. The following line would then throw a `SecurityException`:

```
using (FileStream fs = new FileStream ("test.txt", FileMode.Create))
  ...
```

The Demand, in this case, is made by code that you call—in other words, FileStream's constructor:

```
...
new FileIOPermission (...).Demand();
```

A code access security Demand checks right up the call stack, in order to ensure that the requested operation is allowed for every party in the calling chain (within the current application domain). Effectively, it's asking, "Is this application domain entitled to this permission?"

With code access security, an interesting case arises with assemblies that run in the GAC, which are considered *fully trusted*. If such an assembly runs in a sandbox, any Demands that it makes are still subject to the sandbox's permission set. Fully trusted assemblies can, however, temporarily *escape* the sandbox by calling Assert on a CodeAccessPermission object. After doing so, Demands for the permissions that were asserted always succeed. An Assert ends either when the current method finishes or when you call CodeAccessPermission.RevertAssert.

The Intersect and Union methods combine two same-typed permission objects into one. The purpose of Intersect is to create a "smaller" permission object, whereas the purpose of Union is to create a "larger" permission object.

With code access permissions, a "larger" permission object is *more* restrictive when Demanded, because a greater number of permissions must be met.

(With principle permissions, a "larger" permission object is *less* restrictive when Demanded, because only *one* of the principles or identities is enough to satisfy the demand.)

PermissionSet

A PermissionSet represents a collection of differently typed IPermission objects. The following creates a permission set with three code access permissions, and then Demands all of them in one hit:

```
PermissionSet ps = new PermissionSet (PermissionState.None);

ps.AddPermission (new UIPermission (PermissionState.Unrestricted));
ps.AddPermission (new SecurityPermission (
                      SecurityPermissionFlag.UnmanagedCode));
ps.AddPermission (new FileIOPermission (FileIOPermissionAccess.Read, @"c:\docs"));
ps.Demand();
```

PermissionSet's constructor accepts a PermissionState enum, which indicates whether the set should be considered "unrestricted." An unrestricted permission set is treated as though it contained every possible permission (even though its collection is empty). Assemblies that execute with unrestricted code access security are said to be *fully trusted*.

AddPermission applies Union-like semantics in that it creates a "larger" set. Calling AddPermission on an unrestricted permission set has no effect (as it already has, logically, all possible permissions).

You can Union and Intersect permission sets just as you can with IPermission objects.

Declarative Versus Imperative Security

So far, we manually instantiated permission objects and called Demand on them. This is *imperative security*. You can achieve the same result by adding attributes to a method, constructor, class, struct, or assembly—this is *declarative security*. Although imperative security is more flexible, declarative security has three advantages:

- It can mean less coding.
- It allows the CLR to determine in advance what permissions your assembly requires.
- It can improve performance.

For example:

```
[PrincipalPermission (SecurityAction.Demand, Name="Mary")]
public ReportData GetReports()
{
  ...
}

[UIPermission(SecurityAction.Demand, Window=UIPermissionWindow.AllWindows)]
public Form FindForm()
{
  ...
}
```

This works because every permission type has a sister attribute type in the .NET Framework. `PrincipalPermission` has a `PrincipalPermissionAttribute` sister. The first argument of the attribute's constructor is always a `SecurityAction`, which indicates what security method to call once the permission object is constructed (usually `Demand`). The remaining named parameters mirror the properties on the corresponding permission object.

Code Access Security Permissions

The `CodeAccessPermission` types that are enforced throughout the .NET Framework are listed by category in Table 21a-1 through Table 21a-6. Collectively, these are intended to cover all the means by which a program can do mischief!

Table 21a-1. Core permissions

Type	Enables
`SecurityPermission`	Advanced operations, such as calling unmanaged code
`ReflectionPermission`	Use of reflection
`EnvironmentPermission`	Reading/writing command-line environment settings
`RegistryPermission`	Reading or writing to the Windows Registry

`SecurityPermission` accepts a `SecurityPermissionFlag` argument. This is an enum that allows any combination of the following:

```
AllFlags                    ControlThread
Assertion                   Execution
BindingRedirects            Infrastructure
ControlAppDomain            NoFlags
ControlDomainPolicy         RemotingConfiguration
ControlEvidence             SerializationFormatter
ControlPolicy               SkipVerification
ControlPrincipal            UnmanagedCode
```

The most significant member of this enum is `Execution`, without which code will not run. The other members should be granted only in full-trust scenarios, because they enable a grantee to compromise or escape a sandbox. `ControlAppDomain` allows the creation of new application domains (see Chapter 24); `UnmanagedCode` allows you to call native methods (see Chapter 25).

`ReflectionPermission` accepts a `ReflectionPermissionFlag` enum, which includes the members `MemberAccess` and `RestrictedMemberAccess`. If you're sandboxing assemblies, the latter is safer to grant while permitting reflection scenarios required by APIs such as LINQ to SQL.

Table 21a-2. I/O and data permissions

Type	Enables
`FileIOPermission`	Reading/writing files and directories
`FileDialogPermission`	Reading/writing to a file chosen through an Open or Save dialog box
`IsolatedStorageFilePermission`	Reading/writing to own isolated storage

`ConfigurationPermission`	Reading of application configuration files
`SqlClientPermission`, `OleDbPermission`, `OdbcPermission`	Communicating with a database server using the `SqlClient`, `OleDb`, or `Odbc` class
`DistributedTransactionPermission`	Participation in distributed transactions

`FileDialogPermission` controls access to the `OpenFileDialog` and `SaveFileDialog` classes. These classes are defined in `Microsoft.Win32` (for use in WPF applications) and in `System.Windows.Forms` (for use in Windows Forms applications). For this to work, `UIPermission` is also required. `FileIOPermission` is not also required, however, if you access the chosen file by calling `OpenFile` on the `OpenFileDialog` or `SaveFileDialog` object.

Table 21a-3. Networking permissions

Type	Enables
`DnsPermission`	DNS lookup
`WebPermission`	`WebRequest`-based network access
`SocketPermission`	`Socket`-based network access
`SmtpPermission`	Sending mail through the SMTP libraries
`NetworkInformationPermission`	Use of classes such as `Ping` and `NetworkInterface`

Table 21a-4. Encryption permissions

Type	Enables
`DataProtectionPermission`	Use of the Windows data protection methods
`KeyContainerPermission`	Public key encryption and signing
`StorePermission`	Access to X.509 certificate stores

Table 21a-5. UI permissions

Type	Enables
`UIPermission`	Creating windows and interacting with the clipboard
`WebBrowserPermission`	Use of the `WebBrowser` control
`MediaPermission`	Image, audio, and video support in WPF
`PrintingPermission`	Accessing a printer

Table 21a-6. Diagnostics permissions

Type	Enables
`EventLogPermission`	Reading or writing to the Windows event log
`PerformanceCounterPermission`	Use of Windows performance counters

`Demand`s for these permission types are enforced within the .NET Framework. There are also some permission classes for which the intention is that `Demand`s are enforced in your own code. The most important of these are concerned with establishing identity of the calling assembly, and are listed in Table 21a-7. The caveat is that (as with all CAS permissions) a `Demand` always succeeds if the application domain is running in full trust (see the following section).

Table 21a-7. Identity permissions

Type	Enforces
`GacIdentityPermission`	The assembly is loaded into the GAC
`StrongNameIdentityPermission`	The calling assembly has a particular strong name
`PublisherIdentityPermission`	The calling assembly is Authenticode-signed with a particular certificate

How Code Access Security Is Applied

When you run a .NET executable from the Windows shell or command prompt, it runs with unrestricted permissions. This is called *full trust*.

If you execute an assembly via another hosting environment—such as a SQL Server CLR integration host, ASP.NET, ClickOnce, or a custom host—the host decides what permissions to give your assembly. If it restricts permissions in any way, this is called *partial trust* or *sandboxing*.

More accurately, a host does not restrict permissions to your *assembly*. Rather, it creates an application domain with restricted permissions, and then loads your assembly into that sandboxed domain. This means that any other assemblies that load into that domain (such as assemblies that you reference) run in that same sandbox with the same permission set. There are two exceptions, however:

- Assemblies registered in the GAC (including the .NET Framework)
- Assemblies that a host has nominated to fully trust

Assemblies in those two categories are considered *fully trusted* and can escape the sandbox by `Assert`ing any permission they want. They can also call methods marked as `[SecurityCritical]` in other fully trusted assemblies, run unverifiable (`unsafe`) code, and call methods that enforce link demands, and those link demands will always succeed.

So when we say that a *partially trusted* assembly calls a *fully trusted* assembly, we mean that an assembly running in a sandboxed application domain calls a GAC assembly—or an assembly nominated by the host for full trust.

Testing for Full Trust

You can test whether you have unrestricted permissions as follows:

```
new PermissionSet (PermissionState.Unrestricted).Demand();
```

This throws an exception if your application domain is sandboxed. However, it might be that your assembly is, in fact, fully trusted and so can `Assert` its way out of the sandbox. You can test for this by querying the `IsFullyTrusted` property on the `Assembly` in question.

Allowing Partially Trusted Callers

Allowing an assembly to accept partially trusted callers creates the possibility of an elevation of privilege attack, and is therefore disallowed by the CLR unless you request otherwise. To see why this is so, let's look first at an elevation of privilege attack.

Elevation of Privilege

Let's suppose the CLR didn't enforce the rule just described, and you wrote a library intended to be used in full-trust scenarios. One of your properties was as follows:

```
public string ConnectionString
  => File.ReadAllText (_basePath + "cxString.txt");
```

Now, assume that the user who deploys your library decides (rightly or wrongly) to load your assembly into the GAC. That user then runs a totally unrelated application hosted in ClickOnce or ASP.NET, inside a restrictive sandbox. The sandboxed application now loads your fully trusted assembly—and tries to call the `ConnectionString` property. Fortunately, it throws a `SecurityException` because `File.ReadAllText` will demand a `FileIOPermission`, which the caller won't have (remember that a `Demand` checks right up the calling stack). But now consider the following method:

```
public unsafe void Poke (int offset, int data)
{
  int* target = (int*) _origin + offset;
  *target = data;
  ...
}
```

Without an implicit `Demand`, the sandboxed assembly can call this method—and use it to inflict damage. This is an *elevation of privilege* attack.

The problem in this case is that you never intended for your library to be called by partially trusted assemblies. Fortunately, the CLR helps you by preventing this situation by default.

APTCA and [SecurityTransparent]

To help avoid elevation of privilege attacks, the CLR does not allow partially trusted assemblies to call fully trusted assemblies by default.[1]

To allow such calls, you must do one of two things to the fully trusted assembly:

- Apply the `[AllowPartiallyTrustedCallers]` attribute (called APTCA for short).
- Apply the `[SecurityTransparent]` attribute.

Applying these attributes means that you must think about the possibility of being the *untrusting* party (rather than the *untrusted* party).

Prior to CLR 4.0, only the APTCA attribute was supported. And all that it did was to enable partially trusted callers. From CLR 4.0, the APTCA also has the effect of implicitly marking all the methods (and functions) in your assembly as *security transparent*. We'll explain this in detail in the next section; for now, we can summarize it by saying that security transparent methods can't do any of the following (whether running in full or partial trust):

- Run unverifiable (`unsafe`) code.
- Run native code via P/Invoke or COM.
- Assert permissions to elevate their security level.
- Satisfy a link demand.
- Call methods in the .NET Framework marked as `[SecurityCritical]`. Essentially, these comprise methods that do one of the preceding four things without appropriate safeguards or security checks.

> The rationale is that an assembly that doesn't do any of these things cannot, in general, be susceptible to an elevation of privilege attack.

The `[SecurityTransparent]` attribute applies a stronger version of the same rules. The difference is that with APTCA, you can nominate selected methods in your assembly as nontransparent, whereas with `[SecurityTransparent]`, all methods must be transparent.

> If your assembly can work with `[SecurityTransparent]`, your job is done as a library author. You can ignore the nuances of the transparency model and skip ahead to "Operating System Security"!

Before we look at how to nominate selected methods as nontransparent, let's first look at when you'd apply these attributes.

The first (and more obvious) scenario is if you plan to write a fully trusted assembly that will run in a partially trusted domain. We walk through an example in "Sandboxing Another Assembly."

The second (and less obvious) scenario is writing a library without knowledge of how it will be deployed. For instance, suppose you write an object relational mapper and sell it over the Internet. Customers have three options in how they call your library:

1. From a fully trusted environment
2. From a sandboxed domain
3. From a sandboxed domain, but with your assembly fully trusted (e.g., by loading it into the GAC)

It's easy to overlook the third option—and this is where the transparency model helps.

[1] Before CLR 4.0, partially trusted assemblies could not even call other partially trusted assemblies if the target was strongly named (unless you applied the APTCA). This restriction didn't really aid security, and so was dropped in CLR 4.0.

The Transparency Model

> To follow this, you'll need to have read the previous section and understand the scenarios for applying APTCA and `[SecurityTransparent]`.

The security transparency model makes it easier to secure assemblies that might be fully trusted and then called from partially trusted code.

By way of analogy, let's imagine that being a partially trusted assembly is like being convicted of a crime and being sent to prison. In prison, you discover that there are a set of privileges (permissions) that you can earn for good behavior. These permissions entitle you to perform activities such as watching TV or playing basketball. There are some activities, however, that you can never perform—such as getting the keys to the TV room (or the prison gates)—because such activities (methods) would undermine the whole security system. These methods are called *security-critical*.

If writing a fully trusted library, you would want to protect those security-critical methods. One way to do so is to `Demand` that callers be fully trusted. This was the approach prior to CLR 4.0:

```
[PermissionSet (SecurityAction.Demand, Unrestricted = true)]
public Key GetTVRoomKey() { ... }
```

This creates two problems. First, `Demand`s are slow because they must check right up the call stack; this matters because *security-critical* methods are sometimes *performance-critical*. A `Demand` can become particularly wasteful if a security-critical method is called in a loop—perhaps from another fully trusted assembly in the Framework. The CLR 2.0 workaround with such methods was to instead enforce *link demands*, which check only the immediate caller. But this also comes at a price. To maintain security, methods that call link-demanded methods must themselves perform demands or link demands—or be audited to ensure that they don't allow anything potentially harmful if called from a less trusted party. Such an audit becomes burdensome when call graphs are complicated.

The second problem is that it's easy to forget to perform a demand or link demand on security-critical methods (again, complex call graphs exacerbate this). It would be nice if the CLR could somehow help out and enforce that security-critical functions are not unintentionally exposed to inmates.

The transparency model does exactly that.

> The introduction of the transparency model is unrelated to the removal of CAS *policy* (see sidebar, "Security Policy in CLR 2.0").

How the Transparency Model Works

In the transparency model, security-critical methods are marked with the `[SecurityCritical]` attribute:

```
[SecurityCritical]
public Key GetTVRoomKey() { ... }
```

All "dangerous" methods (containing code that the CLR considers could breach security and allow an inmate to escape) must be marked with `[SecurityCritical]` or `[SecuritySafeCritical]`. This comprises:

- Unverifiable (`unsafe`) methods
- Methods that call unmanaged code via P/Invoke or COM interop
- Methods that `Assert` permissions or call link-demanding methods
- Methods that *call* `[SecurityCritical]` methods
- Methods that *override* virtual `[SecurityCritical]` methods

`[SecurityCritical]` means "this method could allow a partially trusted caller to escape a sandbox".

`[SecuritySafeCritical]` means "this method does security-critical things—but with appropriate safeguards and so is safe for partially trusted callers".

Methods in partially trusted assemblies can never call security critical methods in fully trusted assemblies. `[SecurityCritical]` methods can be called only by:

- Other `[SecurityCritical]` methods
- Methods marked as `[SecuritySafeCritical]`

Security-safe critical methods act as gatekeepers for security-critical methods (see Figure 21a-1), and can be called by any method in any assembly (fully or partially trusted, subject to permission-based CAS demands). To illustrate, suppose that as an inmate you want to watch television. The `WatchTV` method that you'll call will need to call `GetTVRoomKey`, which means that `WatchTV` must be *security-safe-critical*:

```
[SecuritySafeCritical]
public void WatchTV()
{
  new TVPermission().Demand();
  using (Key key = GetTVRoomKey())
    PrisonGuard.OpenDoor (key);
}
```

Notice that we `Demand` a `TVPermission` to ensure that the caller actually has TV-watching rights, and carefully dispose of the key we create. We are wrapping a *security-critical* method, making it *safe* to be called by anyone.

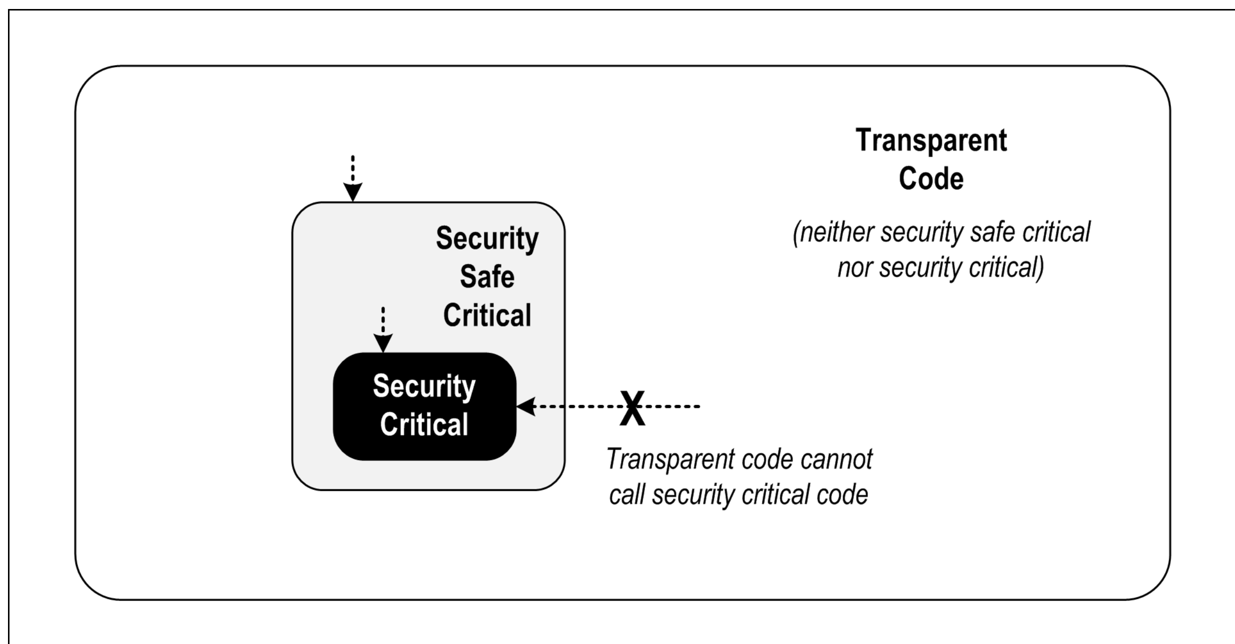

Figure 21a-1. Transparency model; only the area in gray needs security auditing

> Some methods partake in the activities considered "dangerous" by the CLR, but are not actually dangerous. You can mark these methods directly with `[SecuritySafeCritical]` instead of `[SecurityCritical]`. An example is the `Array.Copy` method: it has an unmanaged implementation for efficiency, and yet cannot be abused by partially trusted callers.

The UnsafeXXX Pattern

There's a potential inefficiency in our TV-watching example in that if a prison guard wants to watch TV via the `WatchTV` method, he must (unnecessarily) satisfy a `TVPermission` demand. As a remedy, the CLR team recommends a pattern whereby you define two versions of the method. The first is security-critical and is prefixed by the word *Unsafe*:

```
[SecurityCritical]
public void UnsafeWatchTV()
{
  using (Key key = GetTVRoomKey())
    PrisonGuard.OpenDoor(key);
}
```

The second is security-safe-critical, and calls the first after satisfying a full stack-walking demand:

```
[SecuritySafeCritical]
public void WatchTV()
{
  new TVPermission().Demand();
  UnsafeWatchTV();
}
```

Transparent code

Under the transparency model, all methods fall into one of three categories:

- *Security-critical*
- *Security-safe-critical*
- Neither (in which case, they're called *transparent*)

Transparent methods are so called because you can ignore them when it comes to auditing code for elevation of privilege attacks. All you need to focus on are the `[SecuritySafeCritical]` methods (the gatekeepers) which typically comprise just a small fraction of an assembly's methods. If an assembly comprises entirely transparent methods, the entire assembly can be marked with the `[SecurityTransparent]` attribute:

```
[assembly: SecurityTransparent]
```

We then say that the *assembly itself* is transparent. Transparent assemblies don't need auditing for elevation of privilege attacks and implicitly allow partially trusted callers—you don't need to apply APTCA.

Setting the transparency default for an assembly

To summarize what we said previously, there are two ways to specify transparency at the assembly level:

- Apply the APTCA. All methods are then implicitly transparent except those you mark otherwise.
- Apply the `[SecurityTransparent]` assembly attribute. All methods are then implicitly transparent, without exception.

The third option is to do nothing. This still opts you *into* the transparency rules, but with every method implicitly `[SecurityCritical]` (apart from any virtual `[SecuritySafeCritical]` methods that you override, which will remain safe-critical). The effect is that you can call any method you like (assuming you're fully trusted), but transparent methods in other assemblies won't be able to call you.

How to Write APTCA Libraries with Transparency

To follow the transparency model, first identify the potentially "dangerous" methods in your assembly (as described in the previous section). Unit tests will pick these up, because the CLR will refuse to run such methods—even in a fully trusted environment. (The .NET Framework also ships with a tool called *SecAnnotate.exe* to help with this.) Then mark each such method with:

- [SecurityCritical], if the method might be harmful if called from a less trusted assembly
- [SecuritySafeCritical], if the method performs appropriate checks/safeguards and can be safely called from a less trusted assembly

To illustrate, consider the following method, which calls a security-critical method in the .NET Framework:

```
public static void LoadLibraries()
{
  GC.AddMemoryPressure (1000000);   // Security critical
  ...
}
```

This method could be abused by being called repeatedly from less trusted callers. We could apply the [SecurityCritical] attribute, but then the method would be callable only from other trusted parties via critical or safe-critical methods. A better solution is to fix the method so that it's secure and then apply the [SecuritySafeCritical] attribute:

```
static bool _loaded;

[SecuritySafeCritical]
public static void LoadLibraries()
{
  if (_loaded) return;
  _loaded = true;
  GC.AddMemoryPressure (1000000);
  ...
}
```

(This has the benefit of making it safer for trusted callers, too.)

Securing unsafe methods

Next, suppose we have an unsafe method that is potentially harmful if called by a less trusted assembly. We simply decorate it with [SecurityCritical]:

```
[SecurityCritical]
public unsafe void Poke (int offset, int data)
{
  int* target = (int*) _origin + offset;
  *target = data;
  ...
}
```

> The CLR throws a VerificationException ("Operation could destabilize the runtime") if you attempt to execute a transparent method that contains unsafe code.

We then secure the upstream methods, marking them with [SecurityCritical] or [SecuritySafeCritical] as appropriate.

Next, consider the following unsafe method, which filters a bitmap. This is intrinsically harmless, so we can mark it SecuritySafeCritical:

```
[SecuritySafeCritical]
unsafe void BlueFilter (int[,] bitmap)
{
  int length = bitmap.Length;
  fixed (int* b = bitmap)
  {
    int* p = b;
    for (int i = 0; i < length; i++)
      *p++ &= 0xFF;
  }
}
```

Conversely, you might write a function that doesn't perform anything "dangerous" as far as the CLR is concerned, but poses a security risk nonetheless. You can decorate these, too, with [SecurityCritical]:

```
public string Password
{
  [SecurityCritical] get { return _password; }
}
```

P/Invokes and [SuppressUnmanagedSecurity]

Finally, consider the following unmanaged method, which returns a window handle from a `Point` (`System.Drawing`):

```
[DllImport ("user32.dll")]
public static extern IntPtr WindowFromPoint (Point point);
```

Remember that you can call unmanaged code only from `[SecurityCritical]` and `[SecuritySafeCritical]` methods.

> You could say that all `extern` methods are implicitly `[SecurityCritical]`, although there is a subtle difference: applying `[SecurityCritical]` explicitly to an `extern` method has the subtle effect of advancing the security check from runtime to JIT time. To illustrate, consider the following method:
>
> ```
> static void Foo (bool exec)
> {
>   if (exec) WindowFromPoint (...)
> }
> ```
>
> If called with `false`, this will be subject to a security check only if `WindowFromPoint` is marked explicitly with `[SecurityCritical]`.

Because we've made the method public, other fully trusted assemblies can call `WindowFromPoint` directly from `[SecurityCritical]` methods. For partially trusted callers, we expose the following secure version, which eliminates the danger, by `Demand`ing UI permission and returning a managed class instead of an `IntPtr`:

```
[UIPermission (SecurityAction.Demand, Unrestricted = true)]
[SecuritySafeCritical]
public static System.Windows.Forms.Control ControlFromPoint (Point point)
{
  IntPtr winPtr = WindowFromPoint (point);
  if (winPtr == IntPtr.Zero) return null;
  return System.Windows.Forms.Form.FromChildHandle (winPtr);
}
```

Just one problem remains: the CLR performs an implicit `Demand` for unmanaged permission whenever you P/Invoke. And because a `Demand` checks right up the call stack, the `WindowFromPoint` method will fail if the caller's caller is partially trusted. There are two ways around this. The first is to *assert* permission for unmanaged code in the first line of the `ControlFromPoint` method:

```
new SecurityPermission (SecurityPermissionFlag.UnmanagedCode).Assert();
```

Asserting our assembly-given unmanaged right here will ensure that the subsequent implicit `Demand` in `WindowFromPoint` will succeed. Of course, this assertion would fail if the assembly itself wasn't fully trusted (by virtue of being loaded into the GAC or being nominated as fully trusted by the host). We'll cover assertions in more detail in "Sandboxing Another Assembly."

The second (and more performant) solution is to apply the `[SuppressUnmanagedCodeSecurity]` attribute to the unmanaged method:

```
[DllImport ("user32.dll"), SuppressUnmanagedCodeSecurity]
public static extern IntPtr WindowFromPoint (Point point);
```

This tells the CLR to skip the expensive stack-walking unmanaged `Demand` (an optimization that could be particularly valuable if `WindowFromPoint` was called from other trusted classes or assemblies). We can then dump the unmanaged permission assertion in `ControlFromPoint`.

> Because you're following the transparency model, applying this attribute to an `extern` method doesn't create the same security risk as in CLR 2.0. This is because you're still protected by the fact that P/Invokes are implicitly security-critical, and so can be called only by other critical or safe-critical methods.

Security Policy in CLR 2.0

Prior to CLR 4.0, the CLR granted a default set of permissions to .NET assemblies based on a complex set of rules and mappings. This was called CAS *policy* and was defined in the computer's .NET Framework configuration. Three standard grant sets resulted from policy evaluation, customizable at the enterprise, machine, user, and application domain levels:

- "Full trust," which was granted to assemblies that ran on the local hard drive
- "LocalIntranet," granted to assemblies that ran over a network share
- "Internet," granted to assemblies that ran within Internet Explorer

Only "Full trust" was fully trusted by default. This meant that if you ran a .NET executable over a network share, it would run with a limited permission set and usually fail. This was supposed to offer some protection, but in reality it offered none—because a malicious party could simply replace the .NET executable with an unmanaged executable and be subject to no permission restrictions. All that this restriction achieved was to frustrate people who wanted to run full trust over a network share.

Therefore, the designers of CLR 4.0 decided to abolish these security policies. All assemblies now run in a permission set defined entirely by the host. Executables that you double-click or run from the command prompt will always run in full trust—whether on a network share or on a local hard drive.

In other words, it's now *entirely up to the host* as to how permissions should be restricted—a machine's CAS policy is irrelevant.

Transparency in Full-Trust Scenarios

In a fully trusted environment, you might want to write critical code and yet avoid the burden of security attributes and method auditing. The easiest way to achieve this is not to attach any assembly security attributes—in which case all your methods are implicitly `[SecurityCritical]`.

This works well as long as *all* partaking assemblies do the same thing—or if the transparency-enabled assemblies are at the *bottom* of the call graph. In other words, you can still call transparent methods in third-party libraries (and in the .NET Framework).

To go in the reverse direction is troublesome; however, this trouble typically guides you to a better solution. Suppose you're writing assembly `T`, which is partly or wholly transparent, and you want to call assembly `X`, which is unattributed (and therefore fully critical). You have three options:

- Go fully critical yourself. If your domain will always be fully trusted, you don't need to support partially trusted callers. Making that lack of support *explicit* makes sense.
- Write `[SecuritySafeCritical]` wrappers around methods in `X`. This then highlights the security vulnerability points (although this can be burdensome).
- Ask the author of `X` to consider transparency. If `X` does nothing critical, this will be as simple as applying `[SecurityTransparent]` to `X`. If `X` does perform critical functions, the process of following the transparency model will force the author of `X` to at least identify (if not address) `X`'s vulnerability points.

Sandboxing Another Assembly

Suppose you write an application that allows consumers to install third-party plug-ins. Most likely you'd want to prevent plug-ins from leveraging your privileges as a trusted application, so as not to destabilize your application—or the end user's computer. The best way to achieve this is to run each plug-in in its own sandboxed application domain.

For this example, we'll assume a plug-in is packaged as a .NET assembly called *plugin.exe* and that activating it is simply a matter of starting the executable. (In Chapter 24, we describe how to load a library into an application domain and interact with it in a more sophisticated way.)

Here's the complete code, for the *host* program:

```
using System;
using System.IO;
using System.Net;
using System.Reflection;
using System.Security;
using System.Security.Policy;
using System.Security.Permissions;

class Program
{
  static void Main()
  {
    string pluginFolder = Path.Combine (
      AppDomain.CurrentDomain.BaseDirectory, "plugins");

    string plugInPath = Path.Combine (pluginFolder, "plugin.exe");

    PermissionSet ps = new PermissionSet (PermissionState.None);

    ps.AddPermission
      (new SecurityPermission (SecurityPermissionFlag.Execution));

    ps.AddPermission
      (new FileIOPermission (FileIOPermissionAccess.PathDiscovery |
                             FileIOPermissionAccess.Read, plugInPath));

    ps.AddPermission (new UIPermission (PermissionState.Unrestricted));

    AppDomainSetup setup = AppDomain.CurrentDomain.SetupInformation;
    AppDomain sandbox = AppDomain.CreateDomain ("sbox", null, setup, ps);
    sandbox.ExecuteAssembly (plugInPath);
    AppDomain.Unload (sandbox);
  }
}
```

You can optionally pass an array of `StrongName` objects into the `CreateDomain` method, indicating assemblies to fully trust. We'll give an example in the following section.

First, we create a limited permission set to describe the privileges we want to give to the sandbox. This must include at least execution rights and permission for the plug-in to read its own assembly; otherwise, it won't start. In this case, we also give unrestricted UI permissions. Then we construct a new application domain, specifying our custom permission set, which will be awarded to all assemblies loaded into that domain. We then execute the plug-in assembly in the new domain, and unload the domain when the plug-in finishes executing.

© Joseph Albahari, 2007-2017

In this example, we load the plug-in assemblies from a subdirectory called *plugins*. Putting plug-ins in the same directory as the fully trusted host creates the potential for an elevation of privilege attack, whereby the fully trusted domain implicitly loads and runs code in a plug-in assembly in order to resolve a type. An example of how this could happen is if the plug-in throws a custom exception whose type is defined in its own assembly. When the exception bubbles up to the host, the host will implicitly load the plug-in assembly if it can find it— in an attempt to deserialize the exception. Putting the plug-ins in a separate folder prevents such a load from succeeding.

Asserting Permissions

Permission assertions are useful when writing methods that can be called from a partially trusted assembly. They allow fully trusted assemblies to temporarily escape the sandbox in order to perform actions that would otherwise be prohibited by downstream `Demands`.

Assertions in the world of CAS have nothing to do with diagnostic or contract-based assertions. Calling `Debug.Assert`, in fact, is more akin to `Demand`ing a permission than `Assert`ing a permission. In particular, asserting a permission has *side-effects* if the assertion succeeds, whereas `Debug.Assert` does not.

Recall that we previously wrote an application that ran third-party plug-ins in a restricted permission set. Suppose we want to extend this by providing a library of safe methods for plug-ins to call. For instance, we might prohibit plug-ins from accessing a database directly, and yet still allow them to perform certain queries through methods in a library that we provide. Or we might want to expose a method for writing to a log file—without giving them any file-based permission.

The first step in doing this is to create a separate assembly for this (e.g., *utilities*) and add the `AllowPartiallyTrustedCallers` attribute. Then we can expose a method as follows:

```
public static void WriteLog (string msg)
{
  // Write to log
  ...
}
```

The difficulty here is that writing to a file requires `FileIOPermission`. Even though our *utilities* assembly will be fully trusted, the caller won't be, and so any file-based `Demand`s will fail. The solution is to first `Assert` the permission:

```
public class Utils
{
  string _logsFolder = ...;

  [SecuritySafeCritical]
  public static void WriteLog (string msg)
  {
    FileIOPermission f = new FileIOPermission (PermissionState.None);
    f.AddPathList (FileIOPermissionAccess.AllAccess, _logsFolder);
    f.Assert();

    // Write to log
    ...
  }
}
```

> Because we're asserting a permission, we must mark the method as [SecurityCritical] or [SecuritySafeCritical] (unless we're targeting an earlier version of the Framework). In this case, the method is safe for partially trusted callers, so we choose SecuritySafeCritical. This, of course, means that we can't mark the assembly as a whole with [SecurityTransparent]; we must use APTCA instead.

Remember that `Demand` performs a spot-check and throws an exception if the permission is not satisfied. It then walks the stack, checking that all callers also have that permission (within the current `AppDomain`). An assertion checks only that the *current assembly* has the necessary permissions, and if successful, makes a mark on the stack, indicating that from now on, the caller's rights should be ignored and only the current assembly's rights should be considered with respect to those permissions. An `Assert` ends when the method finishes or when you call `CodeAccessPermission.RevertAssert`.

To complete our example, the remaining step is to create a sandboxed application domain that fully trusts the *utilities* assembly. Then we can instantiate a `StrongName` object that describes the assembly, and pass it into `AppDomain`'s `CreateDomain` method:

```
static void Main()
{
  string pluginFolder = Path.Combine (
    AppDomain.CurrentDomain.BaseDirectory, "plugins");

  string plugInPath = Path.Combine (pluginFolder, "plugin.exe");

  PermissionSet ps = new PermissionSet (PermissionState.None);

  // Add desired permissions to ps as we did before
  // ...

  Assembly utilAssembly = typeof (Utils).Assembly;
  StrongName utils = utilAssembly.Evidence.GetHostEvidence<StrongName>();

  AppDomainSetup setup = AppDomain.CurrentDomain.SetupInformation;
  AppDomain sandbox = AppDomain.CreateDomain ("sbox", null, setup, ps,
                                              utils);
  sandbox.ExecuteAssembly (plugInPath);
  AppDomain.Unload (sandbox);
}
```

For this to work, the *utilities* assembly must be strong-name signed.

> Prior to Framework 4.0, you couldn't obtain a StrongName by calling GetHostEvidence as we did. The solution was to instead do this:

```
AssemblyName name = utilAssembly.GetName();
StrongName utils = new StrongName (
  new StrongNamePublicKeyBlob (name.GetPublicKey()),
  name.Name,
  name.Version);
```

> The old-fashioned approach is still useful when you don't want to load the assembly into the host's domain. This is because you can obtain an AssemblyName without needing an Assembly or Type object:

```
AssemblyName name = AssemblyName.GetAssemblyName
                    (@"d:\utils.dll");
```