CLR INSIDE OUT

Security In Silverlight 2

Andrew Dai – October 2008

Contents

In the August 2008 *MSDN Magazine* CLR Inside Out column, Andrew Pardoe gave a brief description of the CoreCLR security model. He described why we chose not to use code access security (CAS) policy in Silverlight and the difference between Transparent, SafeCritical, and Critical code. In this installment I'll go into more detail about how the Transparency model was born, how it works in Silverlight, and what you can expect to see from it in the future.

Permissions and Sandboxing Untrusted Code

Before I dive into the Transparency model, it helps to have some background on CLR security within the full Microsoft .NET Framework. The CLR provides mechanisms for determining what abilities or rights, represented by permissions, code has based on evidence from the code—the location from which the code is running, the signature of the code if one exists, and so forth. This ensures that code coming from an untrusted location, such as the Internet, doesn't have the same privileges as code coming from a trusted location, such as the local machine's Program Files folder.

The CLR runs each managed application in its own application domain. The preferred mechanism to run code safely from untrusted locations is to run it in a sandboxed application domain. The code running in these domains is restricted to the rights of the sandbox—an environment with limited permissions appropriate for partially trusted code. In general, sandboxed application domains separate code into two groups—full trust and partial trust. I'll go into more detail as to the purpose of these two groups and how the distinction is drawn later in this section.

To make this separation easier to implement and use, we introduced what we call the simple sandboxing APIs in the .NET Framework 2.0, which create each application domain with a given permission set for its sandbox and a list of fully trusted assemblies that are not in the Global Assembly Cache (GAC), as all assemblies in the GAC are already fully trusted. Anything loaded into this application domain that is not on the full-trust assemblies list would get the permissions of the sandbox's granted set. In the case of the application running from the Internet, this would be the built-in Internet permission set. Shawn Farkas, a Software Design Engineer on the CLR security team, has an excellent blog post on how to use this API at go.microsoft.com/fwlink/?LinkId=124952.

With this model, a host (for example, a browser plug-in) would create an application domain for each application running under it—these would load as partially trusted and get the sandbox's permission set. The host may also want to (safely) wrap some full-trust managed or native functions and provide them to its hosted applications in the form of libraries—these library assemblies would be on the full-trust assembly list. For example, file system access is a privileged operation, but saving information locally is useful and can be safe if done correctly (fixed location, never executed, and so on). Safely exposing this functionality, though, can be challenging.

The Transparency Model

The Transparency model is not actually new to CoreCLR (the Silverlight version of the CLR)—it's been around since the .NET Framework 2.0. It was initially introduced to address the issue of the growing number of framework libraries exposing services to partially trusted code and the internal security audits they were incurring within the .NET Framework teams. Libraries with the AllowPartiallyTrustedCallers (APTCA) attribute expose full-trust services to partially trusted code—something potentially dangerous that requires proper security checks on the part of the APTCA library.

Prior to the introduction of Transparency, entire APTCA assemblies were subject to code audits. This was extremely expensive, especially considering that much of the library could be calling a common privileged function but otherwise be completely safe.
We developed Transparency with the intention of creating a strong isolation boundary between privileged and unprivileged code. In this model, there are two kinds of code—Transparent and Critical. Transparent code is subject to the following restrictions:

- Cannot assert permissions or elevate.
- Should not contain unsafe or unverifiable code. This is considered to be a Critical operation and causes demands to be injected if they occur from Transparent code.
- Cannot directly call Critical code. Exceptions to this are explained later in this section.
- Cannot call native code or APIs with the SuppressUnmanagedCodeSecurity attribute.
- Cannot satisfy LinkDemands. All LinkDemands become full, stack-walking demands.

All permission demands flow through Transparent code and go against the application domain; Transparent code cannot assert these demands away. Therefore, Transparent code is always restricted to the rights of the sandbox. Critical code is not restricted in this way and can do the things mentioned above, and therefore the security audit burden for APTCA assemblies is reduced to the Critical surface area of the assembly.
So how do the two types of code interact? If a developer deemed a Critical API as safe to be called from Transparent code (for example, an API that calls into the operating system to retrieve the date and time), he could mark it as TreatAsSafe (actual attribute in code: SecurityTreatAsSafe) to enable this. It is also important to note that in the 2.0 Transparency model, the Transparency rules are only enforced within an assembly and, as a result, public Criticals are implicitly TreatAsSafe. (Throughout this column, I'll be referring to this Transparency model as the 2.0 Transparency model. This Transparency model is, in fact, also part of the .NET Framework versions 3.0, 3.5, and 3.5 SP1.) Therefore, a public Critical method can be called by Transparent code.
The 2.0 Transparency model helped a lot in the later versions of the desktop version of the .NET Framework, making security easier to reason about and making it much easier to enable our teams to expose their services to partial trust applications. With CoreCLR, we had an opportunity not only to improve on the Transparency system, but also to make it the primary mechanism for enforcing security across the runtime and the code built against it.

Transparency in CoreCLR

Although it helps a great deal in assessing the security of an assembly, Transparency is not the only determinant of security in the .NET Framework. Different application domains may have different sandboxes with different sets of permissions—an application running from the Internet and an application running from a corporate intranet do not have the same privileges. Therefore, it is necessary to check for and assert these permissions in code.
CoreCLR allowed us to make some simplifying assumptions due to the fact that we know all Rich Internet Applications built against it run within a Web browser. The first assumption is that each application domain has a fixed sandbox, and we don't have to deal with individual permissions since we know exactly what rights each Silverlight application will have. There are only ever two permission sets—full trust and the Silverlight sandbox. The second assumption we made is that all Silverlight applications are partially trusted and, therefore, cannot elevate beyond the permissions of the Silverlight sandbox.
These are significant simplifications that provide the following results: all Silverlight applications are Transparent and receive the Silverlight sandbox permission set, Silverlight platform libraries (most of which are fully trusted) will only ever be called by Silverlight applications, so we don't need to demand specific permissions; marking dangerous APIs as Critical is sufficient. Finally, since there are no demands, we will never need to stop a permission stack walk and, therefore, we don't need asserts.
Thus, Transparency becomes the sole security enforcement mechanism in CoreCLR, and we are able to ignore much of CAS. To make it a stronger enforcement mechanism, we tightened the rules for Transparent code and made additional improvements to the Transparency system to make it more intuitive and effective.

You may have noticed that in the 2.0 Transparency model, unsafe or unverifiable code that executes in Transparent code causes an injected demand for UnmanagedCodePermission instead of an outright failure/exception. Generally, this permission should only be available in full trust, so this code would fail in partial trust but work otherwise.

With CoreCLR, however, we don't have the concept of permission demands, so an exception will be thrown when this happens (instead of a demand being injected in the new Transparency model). This type of code fails in both models, but is more performant now since we don't have to deal with the demand's stack walk before finally failing.

Also, in Silverlight, all code is Transparent by default. This includes Silverlight platform libraries; all Critical code must be explicitly annotated. This provides us with a secure default that reduces the chances of creating an implicitly Critical method accidentally. More rules were added with regard to type inheritance and method overrides, which I'll explain in the Inheritance Rules section.

I mentioned earlier that a Critical API marked TreatAsSafe was considered to be callable from Transparent code. I also pointed out that, since Transparency on the desktop framework only applies within an assembly, public Critical APIs are implicitly TreatAsSafe. In CoreCLR, we created a new attribute, SafeCritical (again, the actual annotation is SecuritySafeCritical), to describe code that was both Critical and callable from Transparent code. Transparency now applies within and across assemblies, so Critical code, whether public or not, cannot be called from Transparent code.

This change helps to clear up confusion around how to expose Critical code to Transparent code. First, all Critical code is restricted—there is no exception for public code, so adding the Critical annotation will always mean that Transparent code cannot call it. Furthermore, TreatAsSafe only ever makes sense if the Critical attribute is already there—requiring two attributes may lead to some confusion, as it is possible to put TreatAsSafe on a Transparent method even though it does nothing. Finally, having only one way to specify safe, Critical code simplifies the model further.

One thing to realize regarding this new attribute is that when we talk about Critical code, we often include SafeCritical code in the discussion. This is because SafeCritical code is Critical code—it is just as capable as Critical code, but it is also callable by Transparent code. Therefore, to truly make it safe and Critical, the library developer must be sure that the proper input validation or other security checks are being made before Critical code (not safe) is called and that the necessary output validation and side-effect checks are being made afterward. SafeCritical becomes a contract that says this method accesses Critical code but does all the appropriate checks beforehand and afterword. With the new Transparency model, the focus of security audits is the SafeCritical code.

Inheritance Rules

For the Transparency model to be effective, we also had to ensure that the closure of a type or method was secure as well. Deriving types or methods with resource access rights that were different from the rights of the base type or method raised some access protection issues, especially when these were being cast to more accessible parent types or interfaces. For example, you could imagine someone creating a Critical override of a Transparent virtual method, such as ToString, and wanting to prevent Transparent code from calling the override by casting to Object.ToString.

You can imagine that there is an access and capability hierarchy for Transparency levels whereby access becomes more restricted and/or capabilities are increased as you move from Transparent to SafeCritical to Critical. (Here access refers to the ability to contain as well as call this level of code). The inheritance rule for types is that derived types must be at least as restrictive as the base type. This prevents developers from accidentally creating overloads of Critical types callable by Transparent code. **Figure 1** shows what's allowed according to this pattern.

Figure 1 Allowed Type Inheritance Patterns

Base Type	Allowed Derived Type
Transparent	Transparent, SafeCritical, Critical
SafeCritical	SafeCritical, Critical
Critical	Critical

The virtual method override rule is different—the derived method must have the same accessibility from Transparent code as the base method. **Figure 2** shows what's allowed according to this pattern.

Figure 2 Allowed Method Inheritance Patterns

Base (Virtual/Interface) Method	Allowed Override Method
Transparent	Transparent, SafeCritical
SafeCritical	Transparent, SafeCritical
Critical	Critical

Note that we don't allow Critical overrides of Transparent virtual methods—one could simply cast to the base method and bypass the Critical check on the override when calling it. We allow SafeCritical overrides of Transparent functions because the cast would buy you nothing—Transparent code can access SafeCritical code anyway. We also allow Transparent overrides of SafeCritical virtual methods—this may seem surprising at first, but it's actually safe because, again, nothing dangerous can be exposed via casts.

To summarize the Silverlight Transparency model, remember that all code is Transparent by default. All Silverlight applications are completely Transparent. All markings indicating otherwise are ignored. In addition, a fixed permission set means Transparency is the only enforcement mechanism needed for CoreCLR. **Figure 3** summarizes the three types of code and their abilities and traits.

Figure 3 Comparing Transparent, SafeCritical, and Critical Code

	Transparent	SafeCritical	Critical
Access Restrictions	Cannot directly call Critical code.	Can do everything that Critical code can. Must make sure to perform the applicable checks before and after calling into Critical code.	Is privileged and able to call any code. Equivalent of full trust only. Can never be called directly by Transparent code, but may be exposed via SafeCritical code.
Unsafe Code	Cannot contain unsafe or unverifiable code.	May contain unsafe or unverifiable code.	May contain unsafe or unverifiable code.
Availability to Application or Platform Code	Present in both application and platform code.	Only available to Silverlight platform assemblies.	Only available to Silverlight platform assemblies.
Type Inheritance Characteristics	Types must derive from other Transparent types.	Types must derive from Transparent or SafeCritical types.	Types can derive from all types regardless of annotations.
Method Override Characteristics	Methods must override Transparent or SafeCritical virtual/interface methods.	Methods must derive from Transparent or SafeCritical virtual/interface methods.	Methods must derive from Critical virtual/interface methods.

Exposure to Silverlight Transparency

Much of the benefit of the Silverlight Transparency model manifested itself in the simplicity of security enforcement in the Silverlight platform. Since Silverlight is a closed platform, there is currently no notion of trusted third-party libraries. Therefore, a developer's interaction with the Silverlight Transparency model is limited to the APIs he is allowed to call—those that are Transparent or SafeCritical—as his applications are completely Transparent. On the .NET Framework, however, the platform is not closed, and so it is possible to develop additional libraries that could benefit from simpler security enforcement.

However, developers may see these benefits in the next major release of the desktop framework. The Transparency model we introduced in .NET Framework 2.0 made the tasks of analyzing and judging the security of APTCA libraries much simpler; the Silverlight Transparency model took that another step up, allowing us to make assumptions and guarantees that previously had to be confirmed through painstaking review. In the future, we plan to introduce the tightened rules around Transparent code, the SafeCritical attribute, and the Transparency inheritance rules to the full .NET Framework. We also plan to make them available to external developers as well.

One major difference between the desktop framework and Silverlight is that most applications on the desktop framework are built for full-trust environments, while all Silverlight applications are built for a partial-trust environment. Most desktop applications can do anything they want and would continue to be

able to do so—they won't have to think about Transparency at all. Implementationwise, these applications will be considered Critical.

APTCA library developers will experience many of the benefits of our Silverlight platform assemblies. The auditable surface area of an APTCA library is now focused on the SafeCritical layer, as the Critical layer is not accessible from partially trusted, Transparent code. Furthermore, the introduction of the inheritance rules ensures that the library overrides and derivatives aren't unintentionally exposing any security holes. (Just keep in mind that, due to the fact that there's a variety of possible sandboxes on the desktop framework, APTCA libraries will have to handle individual permissions. Therefore, SafeCritical APIs may have to perform permission demands as well as other input validation checks.)

The stricter Transparent code rules also mean that partially trusted applications will be safer and more performant than before. In the .NET Framework 2.0 Transparency model, Transparent code that called or contained unsafe or unverifiable code faced injected demands for UnmanagedCodePermission and the expensive stack walk that resulted when the unsafe/unverifiable code was called. This typically results in a failure, as most sandboxes do not contain this permission in their permission grant sets.

In the new Transparency model, these will be hard failures—as in the CoreCLR, there would be no injected demand and, therefore, no stack walk. By bringing the new Transparency model to the next version of the desktop framework, we hope to make it easier to develop APTCA libraries as well as improve the experience around developing partially trusted applications.

Send your questions and comments to clrinout@microsoft.com.

Andrew Dai is a Program Manager for the CLR at Microsoft. He works on security for Silverlight and the desktop framework.