

.NET Framework Security

Nicholas John Murison

January 6, 2005

Contents

1	Introduction	3
2	Overview of Microsoft .NET Framework	4
2.1	The Goal of .NET	4
2.2	The Anatomy of .NET	5
2.2.1	Development	5
2.2.2	Deployment	6
2.2.3	Execution	6
3	Code Access Security	8
3.1	Evidence	8
3.1.1	Strong Named Assemblies	10
3.2	Permissions	12
3.2.1	Enforcing Permissions Through Stack Walks	14
3.3	Policy	20
3.3.1	Code Groups	20
3.3.2	Policy Levels	21
4	.NET Assemblies	23
4.1	Portable Executable / Common Object File Format	23
4.2	Metadata	24
4.3	Common Intermediary Language	30
5	The Common Language Runtime	34
5.1	Assembly Loading	35
5.2	Policy Management	35
5.3	Class Loading	36
5.4	Just-in-Time Compilation and Verification	37
5.4.1	Verification	37
5.4.2	JIT compilation	40
5.5	Code Management and Security	41
6	Conclusions	43

Foreword

The following document was initially written as a dissertation for the MSc course in Information Security at Royal Holloway, University of London (www.isg.rhul.ac.uk), and was submitted on August 31, 2004. By publishing it more widely, I hope someone may find it interesting and useful.

I would like to take this opportunity to thank my supervisor Andreas Fuchsberger for all his constructive criticism and pointers. Thanks also go to Philippe Clairet for his support and fruitful discussions.

Even though I have moved on to other areas of information security, comments and questions regarding the topic of this report are most appreciated, and can be directed to me at nick@urgusabic.net.

Chapter 1

Introduction

Many computer security issues can arise from unintentional behaviour of computer programs [11, Ch. 8]. This unintentional behaviour can result from several types of issues; mainly buffer overflows, input validation errors and faulty default configurations [15, 11, 26].

Buffer overflows are caused when a process attempts to access a part of memory which is either not part of the process memory space, or is not part of the process memory which it expects it to be. Examples of this are situations when a process accidentally overwrites a value on the heap, causing corruption of the process variables, or when a process overwrites a value on the process stack, causing diversion of the execution to an unintended piece of code [20].

Input validation errors are caused when a process does not parse its input strictly enough, causing the process to behave abnormally. An example of this is when a process takes a file name as an argument, but does not check that the file name is for a valid file for the context. This could lead to overwriting of important system files, and is known as a luring attack [15, Ch. 9].

Default configurations of software often make broad assumptions about the target computer system. This can lead to poor securing of confidential information, as well as abnormal execution of the software. For examples of security issues due to default configurations, search the Computer Emergency Response Teams / Coordination Center database (<http://www.cert.org/>).

In the context of computer security, these issues can affect a computer system in several ways [11]:

Breach of Confidentiality Information that is contained or being processed on the computer system can become available to unauthorised entities.

Breach of Integrity Information that is contained or being processed on the computer system may be modified by unauthorised entities, causing misinterpretation and possibly further breaches of security.

Breach of Availability A process on the computer system may be so severely affected that it fails to operate effectively or at all. This can cause the process or entire computer system to become unavailable for further use. When used in a malicious attack, this is known as a Denial of Service (DoS) attack.

This report looks at how the Microsoft .NET Framework attempts to address these issues in the context of security. I will begin by giving an overview of the Framework, then describe the principles behind Code Access Security (CAS). After this more theoretical look at .NET, I will look at the practical implementations of these mechanisms; first by describing the .NET executable assemblies, and then looking at the process of executing the assemblies within the Framework. I will finish by summarising some of the key points I have addressed, and concluding with how successfully I believe the Framework provides a secure execution environment.

Chapter 2

Overview of Microsoft .NET Framework

This chapter will give a brief overview of the Microsoft .NET Framework. It will briefly describe the intentions behind it, and what the Framework consists of.

2.1 The Goal of .NET

The following quote from Microsoft [8] describes their intentions behind the .NET Framework:

The .NET Framework is an integral Windows component that supports building and running the next generation of applications and XML Web services. The .NET Framework is designed to fulfill the following objectives:

- To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a code-execution environment that promotes safe execution of code, including code created by an unknown or semi-trusted third party.
- To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- To make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications.
- To build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

The Framework has appeared as an alternative to Sun's Java platform (<http://java.sun.com>), but has yet to receive as wide acceptance as the more established Java. A comparison between Java and .NET is beyond the scope of this document.

.NET accommodates the use of several different programming languages, as long as they conform to the Common Type System (CTS), a part of the Common Language Infrastructure (CLI) [14]. It is argued [15, Ch. 2] that this allows developers with different language backgrounds to still

use their preferred languages and not get locked in to a specific language that the Framework supports.

The validity of this argument can be questioned. For a language to be supported as managed code under .NET, it must conform to the CTS and provide access to the Base Class Library (BCL) [15, Ch. 2]. The list of languages that fit under this categorisation is currently quite limited. A developer should use the tool most fitting for the problem, and if the best tool is a procedural language with a small library footprint requirement, an object-oriented CTS-compliant language supporting the BCL is probably not a good alternative. Nevertheless, the statement arguably holds in the context of medium to large scale application development, where the object-oriented paradigm has a wide following. Any further discussion on the choice of programming languages is beyond the scope of this document.

Microsoft's .NET initiatives also cover technologies other than the .NET Framework, like .NET Passport, a single sign-on authentication system. These will not be covered in this document.

The .NET Framework is a published standard (ECMA 335 [14]), and has not only been implemented by Microsoft. Other implementations are Mono (<http://www.mono-project.com/>) and dotGNU (<http://www.dotgnu.org/>). Microsoft also released a reference implementation of the standard, called the Shared Source Common Language Infrastructure (SSCLI).

When I discuss the theoretical aspects of .NET in this report, I will be discussing the ECMA standard. For practical aspects of the Framework, I have been looking specifically at the Microsoft implementations, i.e. both the SSCLI and the commercial Microsoft .NET Framework implementation.

2.2 The Anatomy of .NET

The .NET Framework encompasses development, deployment and execution of so-called managed code. Managed code is code that runs within the Common Language Runtime (CLR), while unmanaged code is code not controlled by the CLR, i.e. native code components (such as operating system libraries) which run independently from the .NET runtime.

I will briefly describe each of the stages, introducing the key concepts of the Framework as they appear.

2.2.1 Development

As briefly described in the previous section, .NET supports development in a variety of programming languages, as long they are able to support the CTS and provide access to the BCL. Languages that currently fit this description are C# [13], Microsoft Visual Basic .NET, Microsoft Visual J# and Microsoft Visual C++. The latter is a managed form of C++, which can be used in type safe manner by adhering to the CTS rules.

The CTS defines what fundamental types must be available at compile time in the compliant languages, as well as the fundamental functions that can be performed on those types. Two kinds of types are defined: value types and reference types.

Value types are primitive types like integers, floating point numbers and enumerated types. Instances of these types are stored in managed memory locations as raw bit-patterns, and it is not possible to deduce simply from the stored pattern what type it represents.

Reference types describe values by the location of the bit-patterns in memory, rather than the actual bit-patterns. This means that an instance of a reference type includes the location of the bit-pattern in memory, what operations can be used to manipulate and represent the bit-pattern, as well as information on how much access to the underlying value should be granted to functions that wish to access the value. Reference types include self-describing types such as classes and arrays,

interface types (describe a set of functions which a class can implement to provide a standard set of functionality), pointer types (both managed and unmanaged), and built-in compound types such as `Object` and `String`.

Based on the CTS, the .NET Framework CLR can verify the type safety of compiled code at runtime. This is described in a later chapter.

For more information on the CTS, please refer to [14, Chapter 8, Partition I]. A good overview is available at [10].

The BCL is a collection of namespaces that all compliant languages can expect access to. Please note that each language does not need to implement the BCL; linking to libraries occurs after the code has been compiled into Common Intermediary Language (CIL) (see below).

The BCL provides the basic functionality for the .NET Framework. Other commonly used functionality such as input/output to streams (`System.IO`) and common network protocol communications (`System.NET`) are provided by other standard libraries, but are not required for minimal implementations [14, Partition IV]. Cryptographic functionality (`System.Security.Cryptography`) and GUI development (`System.Windows.Forms`) are not part of the standard libraries, but are implemented in Microsoft's version of the Framework.

Using one of the compliant languages, developers can create large object-oriented software applications that can either be managed, unmanaged or partially managed. These applications are compiled in assemblies of CIL code and can be deployed in different ways.

2.2.2 Deployment

.NET code is deployed in the form of assemblies. Assemblies can consist of one or more files. The files conform to Microsoft's Portable Executable / Common Object File Format (PE/COFF). The files contain CIL code along with metadata describing the nature of the assembly. For more information on .NET Assemblies, see Chapter 4.

Assemblies can be deployed in a number of ways: publishing on the Internet, publishing on an Intranet and installing locally on the target machine. Depending on what method is used, and where the assembly is being accessed from, different permissions are granted to the assembly. Zones are used to describe groups of virtual or physical locations from which assemblies are loaded. Depending on an assembly's zone, it is granted permissions based on the security policy. For more information on this, see Chapter 3.

In principle, the .NET Framework is platform independent. Hence, an assembly compiled on one platform should run with no problems on any other platform, subject to availability of a CLR implementation for the target platform and equivalent access to unmanaged facilities. As an example, an assembly developed and compiled under Microsoft Windows XP should run under Apple Mac OS X if there is a CLR installed on the OS X machine, and all the required libraries are available. If the assembly were to make a call to some unmanaged code, like a COM+ resource, it would compile successfully on the Windows XP machine, but would most probably not run on the Mac OS X machine.

2.2.3 Execution

.NET assemblies are executed in the CLR. The CLR takes the CIL code from an assembly and compiles it into native code before executing it.

It is at the execution stage that security becomes a major issue. This is where malicious or erroneous code can cause breaches of confidentiality, integrity and accessibility. The CLR is designed to protect the host system from these breaches, by creating a sandbox for the code to be executed in. Microsoft call this sandbox an Application Domain.

The CLR performs several checks of the code before executing it as native code. These stages are briefly described below.

Assembly Loader Initial verification of the assembly. The files are checked for proper PE/COFF formatting, and metadata is inspected.

Policy Manager Evidence from assembly metadata and environment are used in conjunction with security policies to determine what permissions the code should be granted.

Class Loader Individual classes are loaded from the assemblies, and space is allocated for them and their member methods and variables are laid out in memory.

Just-In-Time Compiler and Verifier The CIL is verified for type safety and other security checks take place before the CIL is compiled into native code. Compilation happens on demand, so only methods that get used will be compiled.

Native Code Manager Native code is executed, while security policies are enforced.

The topic of execution and how the CLR operates in detail will be explored extensively in Chapter 5.

Chapter 3

Code Access Security

CAS is one of the fundamental principles behind the .NET Framework. Traditional software execution security has centred around the identity of the *user* who is running the software, rather than the identity of the *code* that is being run. CAS addresses this issue through assigning permissions to code based on evidence presented about the code and the security policy that has been put in place in the execution environment.

As outlined in [15, Ch. 4], the CAS system in .NET serves three purposes:

“Authenticate code identity” When presented with an assembly, the Framework should be able to verify where the assembly is being run from (i.e. URL) and where it originates from (through Strong Naming). The information used to authenticate the code is referred to as *evidence*.

“Authorize code, not users, to access resources” *Permissions* to access files, devices, network resources etc. are granted based on the evidence presented about an assembly and the security *policy* in place. This allows a system to restrict an assembly loaded from the Internet to only access certain local resources. It is important to note that permissions granted based on code identity cannot override permissions based on user identity. That is, if the user running an assembly does not have access to a specific resource, access to that resource will not be granted based on the identity of the assembly, even if the security policy allows it.

“Enforce the authorization decisions on particular pieces of code, not users” There is no point in granting and refusing permissions to an assembly if these are not enforced. The CAS system must ensure policies are not breached, and that assemblies are not given access to resources they have not been granted permission to access.

It is important to note that the .NET runtime environment also enforces permissions based on user permission policies. The environment supports traditional user-based access control as well as role-based access control. The details of these control mechanisms are more platform specific, and I therefore do not wish to delve too far into the area. Suffice it to say, a permission not granted to a user can not be granted to an assembly being run by that user.

The remainder of this chapter will take a more detailed look at the three concepts that make up CAS: evidence, permissions and policy.

3.1 Evidence

Evidence is information that authenticates an assembly or Application Domain. Permissions to access resources are granted based on the evidence available, so it is important that the evidence

is valid.

Evidence can be provided by the assembly that wishes to execute, or the host under which it will be executed. In this context, a host is not a physical computer, but a process which invokes the CLR or loads a piece of managed code.

A host can either be unmanaged or managed. An unmanaged host is a process which invokes the CLR. Microsoft .NET currently ships with several unmanaged hosts: a web browser (Internet Explorer), a server side scripting environment (ASP.NET), a command line host and more. Unmanaged hosts like these cannot directly load and inspect assemblies, but must invoke the CLR which will take care of code loading and execution. As a consequence, unmanaged hosts can not provide evidence on an assembly, but can only provide evidence on an Application Domain.

A managed host is a piece of managed code which can load other managed code. Such a host requires specific security permissions to be granted. This makes sense, so that arbitrary code loaded from an untrusted site on the Internet can not act as a host and control execution. Managed hosts can provide evidence both on Application Domains and assemblies. Classes are provided in the BCL to create new Application Domains and load assemblies, plus providing evidence for these instantiations.

Evidence comes in various forms, and custom evidence can be provided by developers if needed. Common evidence associated with an assembly and available in the BCL are:

Zone The assembly will have originated either from the local system, somewhere on the local network or intranet, or from the Internet. Zones distinguish between these various locations. Zones do not necessarily only distinguish between these broad categorisations of locations; locations on the Internet, for example, can be sub-categorised into a zone for trusted Internet sites and a zone for untrusted Internet sites. The concept of zones originates from Microsoft Internet Explorer [7].

Uniform Resource Locator (URL) URLs are a common concept on the World Wide Web [3]. An assembly loaded from a network resource will have a uniform resource locator akin to `http://some.server.net/assemblies/test.exe`, which informs the CLR what site and where on that site the assembly originated from. An assembly loaded from the local system will not have a URL.

Site As well as the URL evidence, the site from which the assembly has been loaded from features as a separate piece of evidence. This becomes useful for when you wish to set a security policy denying permission to execute for any assembly from a specific site `some.server.net`, but you wish to grant permission to execute to any assembly loaded from a specific location on that site, say `some.server.net/secure`. As with URL evidence, assemblies loaded from the local system will not have any Site evidence.

Application Directory This specifies the directory from which the code is being executed. This can be useful for server side execution, where a developer may want to ensure that their code is not being called from a web page on the server which it should not be called from.

Permission Request This simply provides information on what permissions an assembly has requested in order to execute. Using this, a policy could be enforced where an assembly that requested access to sensitive information on the local system as well as access to communicate over the network should not be allowed to execute, in case of an attempted or accidental information leak.

Hash This is a cryptographic hash of the assembly file. For assemblies that consist of multiple files, the hash represents the manifest file. MD5 and SHA1 are available by default, but other hash algorithms can be used if required. Using this evidence in a policy, a system administrator can allow full access to a heavily trusted assembly based on its hash value. This way the assembly can be loaded from anywhere, be it local system, intranet or Internet, and it will still receive the same privileges.

Publisher Certificate This provides information on the Authenticode signature [6] on the assembly, if there is one. An Authenticode signature is based on an X.509 certificate and Public Key Infrastructure [12] and authenticates the assembly as having been signed by a specific party. This evidence can be used to always trust assemblies from a specific software provider, regardless of where it is being loaded from.

Strong Name A strong name is a cryptographic representation of the assembly's name, version and culture. Strong names can be used to avoid so-called "DLL Hell", and are discussed in the section below.

The above evidence types are supported by the BCL, but other custom evidence types can be specified by the developer. Custom evidence will not affect the permissions granted unless the security policy handles the evidence.

3.1.1 Strong Named Assemblies

One of the features that Microsoft has emphasised is strong naming of assemblies. Strong naming is meant to provide a globally unique identifier for an assembly so as to prevent "DLL Hell"

"DLL Hell" is a bi-product of how Microsoft Windows manages its shared library resources [15, Ch. 9]. Shared libraries are stored in Dynamic Link Library (DLL) files identified purely by the library name followed by ".dll". Because no version information is stored as part of the file name, multiple versions of the same library can not be stored in the same directory. Consider the following scenario:

- Application X is installed on a system. As part of its installation, it installs version 1.1 of foo.dll in a directory where other applications can search for DLL files.
- Application Y is installed on the same system. It depends on version 2.3 of foo.dll, and thus the previous DLL file will not suffice. Version 2.3 is therefore installed over the existing copy of foo.dll.
- A major revision of the interface functions in foo.dll occurred at version 2.0. This breaks backward compatibility, meaning applications dependent on versions prior to 2.0 will not work with version 2.0 and newer.
- Application X gets executed, and tries to call a function in foo.dll that no longer exists.

The problem can also occur when several applications make use of identically named but completely different DLL files. I.e. two different development teams may have developed a library to deal with printers and both called the library printer.dll.

Some software companies also ship their software with modified versions of common DLL files, which can potentially break compatibility as well.

One solution to this problem would be for each application to store their DLL files in separate directories, for example the same directory as the application executable. However, this can lead to excessive redundant waste of disk space, as applications which share the same DLL versions will each have their own copy.

UNIX based systems like FreeBSD and Linux partly solve this problem by including version numbers in the actual file name. Using the previous example, application X would use foo.1.0.so while application Y would use foo.2.3.so under Linux or FreeBSD [28].

Of course, the UNIX approach does not solve the issue when applications try to install custom versions of a shared library that shares its version number with an official version, nor the situation where two libraries are identically named.

Strong naming an assembly strives to completely solve the problem by providing globally unique identities for all assemblies. Under the .NET Framework, a shared library is just another assembly, and an assembly can function as a library or an executable or both.

A strong name is the combination of the assembly's name, version and a public key which is unique to the assembly¹. This creates a namespace under each public key, ensuring collision free naming, as long as the owner of the public key has good naming and versioning policies, i.e. two assemblies that are named the same and have the same version number but are under different public keys will not collide. However, if a developer were to release two copies of an assembly with the same version numbers under the same public key, there would be a collision.

The addition of the public key to the naming does not solve the “DLL Hell” problem by itself. Imagine a situation where a third party software developer makes a change to an assembly developed by company X. The third party developer can still insist that the strong name is valid, as the name is not bound to the content of the code. Applications referring to the assembly would still consider the assembly code valid under the same strong name, even if the code has been corrupted, unless, of course, the content of the assembly is cryptographically bound to the public key belonging to company X, through a signature.

A strong name signature is a cryptographic representation of the entire assembly, including metadata, code and resources, computed over a private key belonging to the creator. This signature is appended to the assembly, and can be verified by finding the corresponding public key in the assembly's metadata and performing standard public key signature verification.

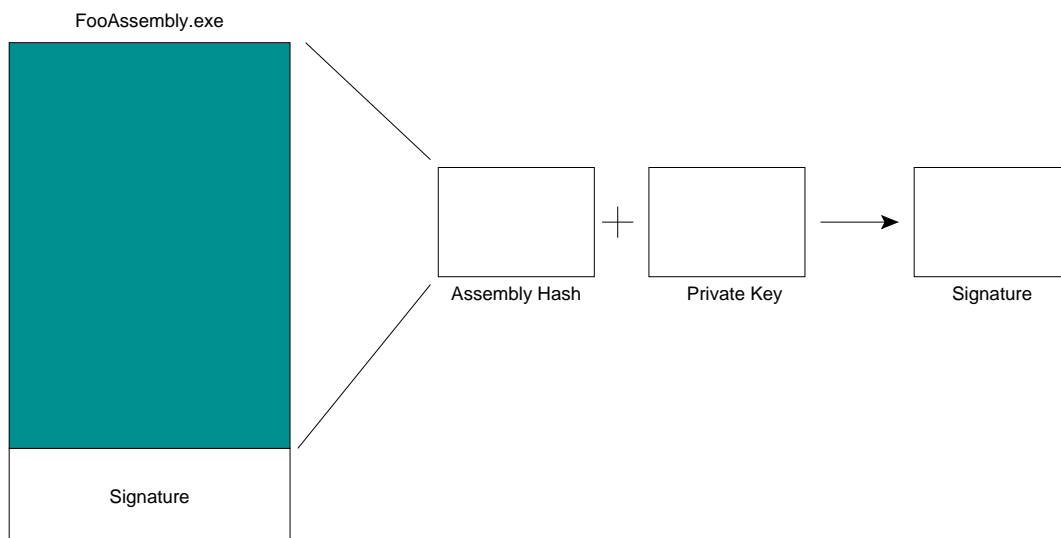


Figure 3.1: The strong name signature is computed from a hash of the assembly over a private key. Figure taken from [15, Ch. 9].

Assemblies which depend on a strong named assembly can refer to the assembly by its strong name, requiring the runtime to load that specific assembly. Because the cryptographic signature encompasses all the metadata, the strong name will represent the version information as well. In brief, a strong name signature identifies a specific assembly with specific metadata (including a specific version), the specific CIL in that assembly, signed under a specific private key. By verifying the signature on an assembly against its strong name, an application can ensure that it is referencing not only a specific version of a shared library, but an uncorrupted version of the library assembly.

¹The same public/private keypair can be used to Strong Name several assemblies, but this is not advised [22]

One problem exists with strong naming core library assemblies. In Microsoft's CLR implementation, all the core assemblies are signed by Microsoft, using a private/public keypair belonging to Microsoft. This in itself is fine, until someone else wishes to reimplement the core assemblies, possibly for a different architecture.

When an assembly calls a core assembly, it does so by its strong name. If all core assemblies were signed by the creator of that assembly – in this case Microsoft – the public key in the metadata of the core assembly would be the public key belonging to Microsoft. Thus, all derived assemblies would be hard wired to use only Microsoft's core assembly implementations because they would all require Microsoft's public key to be part of the strong name.

To solve this problem, the public key contained in the core assemblies' metadata is not a public key belonging to the creator, but a placeholder. This placeholder is known as the ECMA key and is the value 0x000000000000000040000000000000. Using this placeholder, the strong name of a specific version of a core assembly will always be the same, no matter who implemented it.

This causes an issue when wanting to verify the strong name signature on a core assembly. The ECMA public key does not have a corresponding private key that people can use to sign their core assembly implementation. Instead, the strong name signature is computed using a private key belonging to the CLR implementor. When a core assembly's signature needs to be verified, the CLR substitutes the ECMA placeholder in the metadata for the CLR implementor's public key, and then performs verification. It is important to note that one CLR implementor's core assembly will not verify under another CLR implementor's CLR, as the public key of one implementor will not verify a signature by another implementor.

Strong names by themselves do not provide any security, only a mechanism for dealing with library versioning and naming conflicts. However, the strong name signature does provide integrity of the assembly.

3.2 Permissions

In .NET, permissions act as a prerequisite for an assembly to access a certain resource. A piece of code cannot access any resources, or even execute, unless it has been granted a permission to do so by the security policy.

Permissions are used in three different ways. They can be (quoted from [15, Ch. 6]):

- Granted by the security policy
- Demanded by .NET assemblies
- Used for other security actions

Based on the evidence presented by the assembly and runtime, the security manager can grant permissions to an assembly that complies with the security policy in place. More can be found about this in the next section. Each assembly is granted a collection of permissions, known as a permission set. These sets can be combined using set unions, which becomes useful when resolving permissions based on security policy (next section).

An assembly's permission set accompanies the assembly code through its stages in the runtime environment. Once a piece of code in the assembly attempts to access a resource, the permission set is checked to see if the assembly has the appropriate permission to access that resource. The assembly's attempt to access a resource is preceded by a permission demand.

As an example, an assembly downloaded from the Internet may have been granted an extremely restricted permission set: only to execute and not access any local resources. If a piece of code in the assembly attempts to read from a local file, the assembly's permission set would be checked.

When no permission to access that file is found, the access is denied. An exception will be raised, and the assembly will need to deal with that exception, or fail to execute any further.

There are in fact two separate ways for an assembly to demand a permission: declaratively and imperatively.

Declarative permission demands are parsed at compile time, and can not be changed after the assembly has been compiled. These are useful in that you can determine what permissions an assembly will require by looking at the assembly metadata at load time [4]. If it turns out that the assembly will not be granted all the permissions it is demanding, then the runtime can save itself the trouble of executing the code at all.

Imperative permission demands are parsed at runtime, and are part of the actual assembly code. These allow greater flexibility in permission demands, giving the assembly the opportunity to demand permissions to access resources based on runtime conditions.

As an example, an assembly may wish to write to a specific file on local disk, but only if certain conditions are true. If it turns out the permission has not been granted, then the assembly can still run as usual if the required condition is false. This also gives the assembly control over what happens to its execution when a permission is denied, by handling the exception that gets raised. In the case of declarative permission demands, it becomes the responsibility of the calling assembly to handle any exceptions that are raised.

Below is some example C# code (Figure 3.2) that shows how these two approaches differ in code:

There are certain pre-defined permission sets, known as named permission sets [27]. These are used in the default security policy. Examples of named permission sets are described in [27] as follows:

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 perform limited user interface actions as well as use the safe file system called Isolated Storage.

Although most permissions are related to CAS, there is one predefined permission called `PrincipalPermission`, which deals with user identity. The permission can be used to check whether the user who is running the calling code is in a specific role or group, by using the Role Based Access Control functionality in the Framework.

Although .NET specifies standard permissions, which are used by the default security polices and demanded by the core assemblies, custom permissions can be easily created. The BCL contains mechanisms to create custom permissions which assemblies can demand. For code identity permissions, new permission classes can be created by inheriting from the abstract class `System.Security.CodeAccessPermission`. For user identity based permissions, new permission classes can be created by implementing the `System.Security.IPermission` interface and the `System.Security.ISecurityEncodable` interface. To use custom permissions in a security policy, they need to be referred to in custom membership conditions, which are discussed later in this chapter.

It is important to remember that permissions granted to an assembly will not be able to override platform specific access control. That is, even if the security policy grants an assembly write permission to a specific file, the underlying operating system can still deny access to the file based on traditional user based access control.

```
// Declarative permission demand
[FileIOPermission(SecurityAction.Demand, Write=@"tmp/dummy")]
public void DoSomething()
{
    // Do something involving writing to /tmp/dummyfile
    // Execution will not make it as far as this function if the above permission
    // demand can not be satisfied at load time.
}

// Imperative permission demand
public void DoSomethingElse()
{
    bool fileAvailable = true;

    try
    {
        // Try to demand the required permission
        new FileIOPermission(
            FileIOPermissionAccess.Write.Write, @"tmp/dummy").Demand();
    }
    catch (SecurityException e)
    {
        // If we get a SecurityException, handle it gracefully
        fileAvailable = false;
    }

    if (fileAvailable) {
        // We got the permission, do something with the file
        ...
    }
}
```

Figure 3.2: Example C# code showing declarative and imperative permission demands.

3.2.1 Enforcing Permissions Through Stack Walks

When a function attempts to access a resource, a permission check must be performed to ascertain that the function has the appropriate permissions granted to it. However, the runtime cannot just check that the function attempting to access the resource is allowed access, but that the function calling that function has suitable permissions as well, and so on.

The reason for this is to prevent the following type of scenario:

- An Internet based assembly with only permission to execute (i.e. no access to local resources) calls a function in an assembly on the local system.
- The local assembly has a much less restricted permission set, and can write to files on local disk.
- The called function in the local assembly takes a filename as an argument, and data to write to it.
- The Internet based assembly calls the local assembly function using an important system file as argument and scrambled or malicious data.

- The local assembly is granted permission to write to the system file, and overwrites the critical data.

Obviously, the implementor of the local assembly has not been thinking about security concerns in this instance, and if the above scenario were possible, it would have serious security implications. .NET tries to prevent these so-called luring attacks by checking that every function in the chain of function calls in the current process have suitable permissions. In the context of the above example, the runtime would deny the local assembly function access to write to the system file, because one of the calling functions (the one in the Internet based assembly) did not have the appropriate permissions.

The CLR uses the process stack to determine whether all calling functions have appropriate permissions. This is known as a stack walk. The CLR process stack contains a lot more metadata than a native code process stack, some of which specifically deals with permissions. A stack walk proceeds as follows:

When a set of permissions is required, inspect the stack

For each calling function on the stack:

 Check to see if it has the appropriate permissions

 If not, raise exception

 Otherwise, continue

 Check to see if the function asserts or denies any of the permissions

 If it asserts one of more of the permissions demanded, take those permissions off the list of demanded permissions and move on

 If it denies any of the permissions demanded, raise an exception

 Proceed to next function on stack

Please note that the function which raises the demand is not checked for the permission set; only the functions below it in the stack are checked.

A stack walk will usually continue until it has either found a function on the stack which does not fulfill the permission set required for the resource access, or until it runs out of functions on the stack. In the former case, an exception is raised, and the access is denied (Figure 3.4). In the latter, the access is allowed (Figure 3.3).

Stack walks can be modified by using asserts and denies. These are member functions of the permission objects as they are represented in the BCL. A permission can be asserted by a function on the stack, as long as that function has the appropriate permission.

By asserting a permission, the function says it vouches for the functions below it in the stack, and the runtime no longer needs to check the functions below for that specific permission. If the set of permissions that needs to be checked for is now empty, the stack walk ceases, and the access is permitted. If the assert only causes a subset of the permission set to be vouched for, those permissions are removed from the required permission set, and the stack walk continues with the remaining permission set. See Figure 3.5 for an example.

By denying a permission, a function states it will not accept that the functions below it be checked for that specific permission (possibly in case of fear of luring attacks), and the stack walk ceases immediately with the access being denied. See Figure 3.6.

Returning to the luring attack example from above, we now have a situation where a developer can allow that attack to happen, if they are unwitting in their use of asserts. That is, the developer of the local assembly may have included an assert in his function, which would then cause the stack walk to stop before getting to the lower privileged Internet based assembly. The motivation

for this may be that the developer noticed his function did not work with some less privileged assembly that he wanted to inter-operate with. Instead of coming up with a different security policy, or perhaps making an assert dependant on evidence from the calling assembly, a simple assert is a quick fix for the problem.

This is an extremely dangerous scenario, in that it essentially undermines the whole permission checking infrastructure by overriding it. This may be a major source of security related problems in .NET applications in the future.

```

namespace ClientAssembly          // Fully trusted assembly
{
    public class ClientClass
    {
        public void Bar()
        {
            ServerAssembly.Foo(); // Call the function in the other assembly
        }
    }
}

namespace ServerAssembly          // Fully trusted assembly
{
    public class ServerClass
    {
        public void Foo()
        {
            // This causes the stack walk:
            FileStream stream = new FileStream("tmp.dat", FileMode.Open,
                                             FileAccess.Read);
        }
    }
}

```

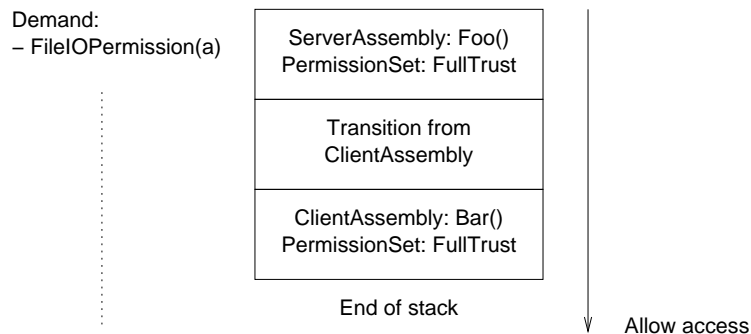


Figure 3.3: Straight forward stack walk where both assemblies on the stack have fully trusted permission sets, thus the access is allowed.

```

namespace ClientAssembly      // Untrusted assembly
{
    public class ClientClass
    {
        public void Bar()
        {
            ServerAssembly.Foo(); // Call the function in the other assembly
        }
    }
}

namespace ServerAssembly      // Fully trusted assembly
{
    public class ServerClass
    {
        public void Foo()
        {
            // This causes the stack walk:
            FileStream stream = new FileStream("tmp.dat", FileMode.Open,
                                              FileAccess.Read);
        }
    }
}

```

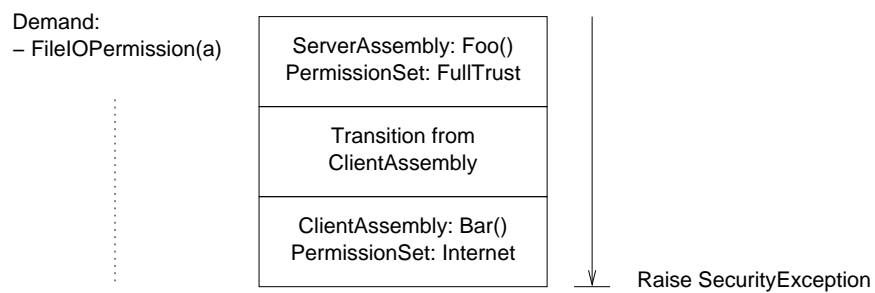


Figure 3.4: The calling assembly does not have the permissions that are demanded, thus the access is denied and an exception is raised.

```

namespace ClientAssembly          // Untrusted assembly
{
    public class ClientClass
    {
        public void Bar()
        {
            ServerAssembly.Foo(); // Call the function in the other assembly
        }
    }
}

namespace ServerAssembly          // Fully trusted assembly
{
    public class ServerClass
    {
        public void Foo()
        {
            // Assert the write permission (b)
            new FileIOPermission(FileIOPermissionAccess.Write, "tmp.dat").Assert();

            // Call the next function
            Caller();
        }

        public void Caller()
        {
            // Assert the read permission (a)
            new FileIOPermission(FileIOPermissionAccess.Read, "tmp.dat").Assert();

            // This causes the stack walk:
            FileStream stream = new FileStream("tmp.dat", FileMode.Open,
                                             FileAccess.ReadWrite);
        }
    }
}

```

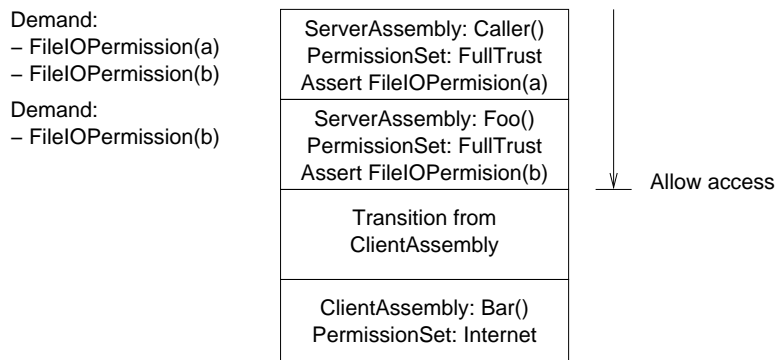


Figure 3.5: Even though the calling assembly at the bottom of the stack does not have the right permissions, the access is allowed because the two functions from the upper assembly each assert one of the demanded permissions each, depleting the demanded permission set.

```

namespace ClientAssembly          // Untrusted assembly
{
    public class ClientClass
    {
        public void Bar()
        {
            ServerAssembly.Foo(); // Call the function in the other assembly
        }
    }
}

namespace ServerAssembly          // Fully trusted assembly
{
    public class ServerClass
    {
        public void Foo()
        {
            // Denying a demanded permission will stop the stack walk
            new FileIOPermission(FileIOPermissionAccess.Write, "tmp.dat").Deny();

            // This causes the stack walk:
            FileStream stream = new FileStream("tmp.dat", FileMode.Open,
                                              FileAccess.ReadWrite);
        }
    }
}

```

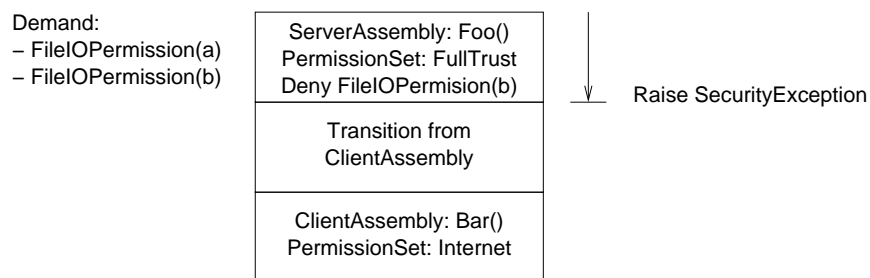


Figure 3.6: The called assembly does not trust its caller to use the FileIOPermission for "b" correctly, and thus explicitly denies the permission. Thus the demands fails and an exception is raised.

3.3 Policy

So far, we have discussed evidence and permissions. Evidence describes the identity of code, while permissions describe what access to resources a piece of code should have. Evidence is mapped to permissions through security policies.

As an example, a security policy may state that assemblies originating from a specific site on the Internet (use of evidence) should be allowed access to files in a specific directory on the local system (granting of permissions), while assemblies which originate from elsewhere on the Internet (use of evidence) will have no access to any files on the local system (granting – or rather lack of granting – of permissions).

A security policy does not need to be fine grained, specifying what permissions a specific set of evidence should yield. This could be quite time consuming, although slightly more efficient than specifying permissions based on the exact identity of an assembly. Instead, assemblies are grouped together in code groups, based on their common evidence, or membership conditions. This is explained in more detail in the section below on code groups.

To make policy management easier, there is not just one flat security policy for all assemblies running on a computer. There are in fact four different policy levels which interact to create the specific security policy relevant to a CLR instance. This is explained more closely in the section below on policy levels.

3.3.1 Code Groups

Code groups are a way of grouping a set of assemblies based on their common evidence, referred to as membership conditions. The point of the code groups is to provide a way to grant a set of permissions to a set of assemblies that all fit a certain common criteria, thus easing the management of the security policies.

Membership conditions are not simply evidence by another name, as a membership condition can refer to several pieces of evidence. For example, a membership condition which specifies that an assembly must be written by a specific developer may check both the assembly's Strong Name evidence and the assembly's Publisher Certificate evidence. Also, several membership conditions can use the same evidence.

The default security policies make use of a standard set of membership conditions, all of which are represented in the BCL. Custom membership conditions can easily be created by implementing the `System.Security.Policy.IMembershipCondition` interface, the `System.Security.ISecurityEncodable` interface and the `System.Security.ISecurityPolicyEncodable` interface from the BCL.

Code groups will often refer to the same membership conditions, for example several code groups will deal with assemblies originating from the Internet. It is therefore useful to organise code groups in a hierarchy, so that the process of granting a specific permission set to an assembly becomes a process of walking through a hierarchy of code groups which check for specific evidence. Thus the security policy becomes a hierarchy of mappings from evidence to permission sets.

In Figure 3.7, an example security policy is outlined. When the permission set for an assembly needs to be calculated, the root node is given the assembly evidence, and the node resolves the permission set by traversing its subnodes. An assembly with a publisher certificate from “ACME” will obtain the FullTrust permission set when downloaded from the Internet, while all assemblies from the “www.evil.com” site will not be granted any permissions whatsoever, due to the “Exclusive” code group attribute on the relevant code group. This may be to prevent malicious code that is running locally with privileges from accessing parts of code on that site. The code

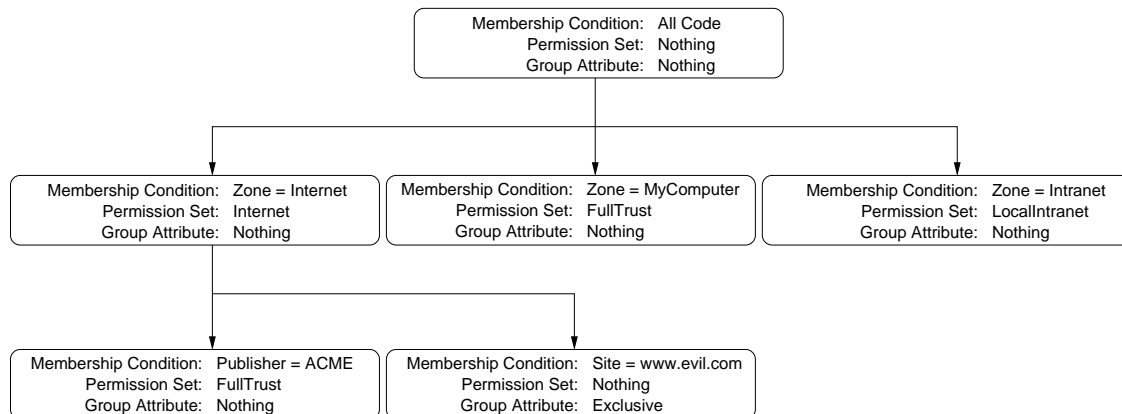


Figure 3.7: An example security policy code group hierarchy. Note the groups are hugely simplified for sake of example.

group will disallow execution of any code from that site, thus preventing the locally executing malicious code from running properly ².

The use of the “Exclusive” attribute in the example above works by ignoring all permission sets granted by any other code group, and only granting the permission set granted by the code group that is defined as exclusive. If an assembly’s evidence matches the membership conditions of more than one exclusive code group, an error occurs, and it is up to the system administrator to ensure that such contradictory code groups do not exist.

.NET comes with a set of default code group types [15, Ch. 8] used by the default security policies. Code groups have differing ways of handling its membership conditions, thus the code groups types differ. Custom code group types can also be created by inheriting from the abstract class `System.Security.Policy.CodeGroup`. The behaviour with regards to treating subnodes in the code group hierarchy can be altered by defining non-standard functionality in the `Resolve` member function.

3.3.2 Policy Levels

The .NET Framework does not operate with only one security policy. It has in fact four different policy levels, so as to accommodate the intentions of several parties: the specific Application Domain, the individual user, the machine administrator and the enterprise or organisation in which the computer exists. Respectively, the policy levels are referred to as the AppDomain, User, Machine and Enterprise levels. While the three former levels are specified and managed by users and administrators, the AppDomain level is computed automatically per Application Domain at runtime.

Although the policy levels are organised in a hierarchy, the policy levels are intersected to produce a policy for the specific context in which an assembly is being run (Figure 3.8). Each policy level suggests what permission sets should be granted to a specific assembly, but only the permissions suggested by all four levels are finally granted.

The hierarchy comes into play when a code group in one level has the “LevelFinal” attribute set. This attribute causes the permission sets in policy levels below the level being evaluated to not be considered. This way, a system administrator for an enterprise can ensure that a machine

²Unless of course the malicious calling code has successfully asserted all the permissions it needs before calling the remote code. See the section on stack walks earlier in the chapter for how this can happen.

administrator or individual user can not override the denying or granting of certain permission sets.

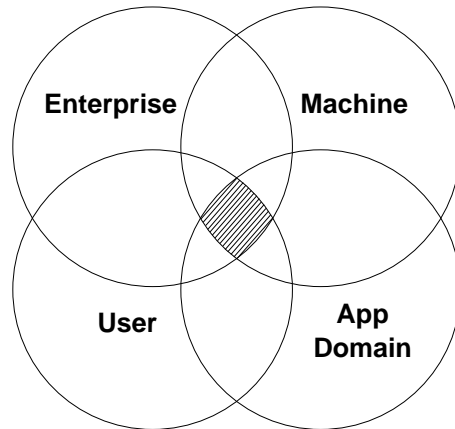


Figure 3.8: The resulting security policy is calculated as the intersection of the four policy levels. Figure taken from [4].

Each policy level contains a code group hierarchy, a list of named permission sets used in the hierarchy, and a list of fully trusted assemblies. The list of fully trusted assemblies is necessary so as not to cause recursion during policy resolution; the classes used in the policy levels need to be loaded from assemblies, which again will cause the security policy resolution to initiate unless explicitly fully trusted.

By default, the Enterprise and User policy levels have empty code group hierarchies, except for an all-catching code group granting the FullTrust permission set. The AppDomain policy level only exists if an application has specifically set one. Only the Machine policy level has a default code group hierarchy that contains anything more than a root node. The result, by default, is that only the contents of the Machine level policy is considered during policy resolution.

The Machine level code group hierarchy defaults can be found in [15, Ch. 8].

Chapter 4

.NET Assemblies

This chapter will discuss the internal format of assemblies, how they make use of the CAS principles, and how they assist the CLR in performing verification and validation.

I will first give an overview of the Portable Executable / Common Object File Format (PE/COFF), which is used by all assemblies. I will then describe the two major components of the assembly: the assembly metadata and the CIL.

4.1 Portable Executable / Common Object File Format

Assemblies and modules are defined in [25] as follows:

An *assembly* is the basic unit of deployment and versioning, consisting of a manifest, a set of one or more modules, and an optional set of resource.

I.e. assemblies are the logical units of deployment, which consist of one or several physical units of deployment; modules, which are better known as files. An assembly can consist of one module (or file), or several, all of which are in Portable Executable / Common Object File Format.

As shown in Figure 4.1 (derived from [19, 5, 25]) the .NET CLR data is entirely contained within the `.text` section of the PE/COFF module, including both the metadata and the managed code. Other native, unmanaged resources may also be present in the module.

The PE headers contain slots which are used in determining what type of module the file represents. In the case of .NET modules, slot number 14 contains a reference to a header containing information that describes the managed data parts of the module [15, Ch. 11]. When the operating system encounters this reference, it passes control to the .NET CLR.

A peculiarity associated with this design is how the header references the CLR executable, or more specifically “`mSCOREE.dll`”. Because Microsoft operating systems did not initially know how to handle managed modules, the managed code header has to contain a reference to the CLR executable for the OS to understand how to execute the managed module. This was the source of the “Donut virus” [16], which redirected the reference to “`mSCOREE.dll`” to point to malicious native code, which then infected other .NET modules on the victim machine. Revised version of Windows XP fixes this problem by no longer paying attention to what the header is referencing, but simply deducing that the module should be passed to “`mSCOREE.dll`” based on PE slot 14 not being `null`¹.

¹LaMachia et al [15, Ch. 11, p. 153] seem to disagree with this description of the issue, claiming that only versions of Windows prior to Windows XP were susceptible to the virus. However, most sources [16][18], including Microsoft [9] state that only Windows 2000 and XP were affected.

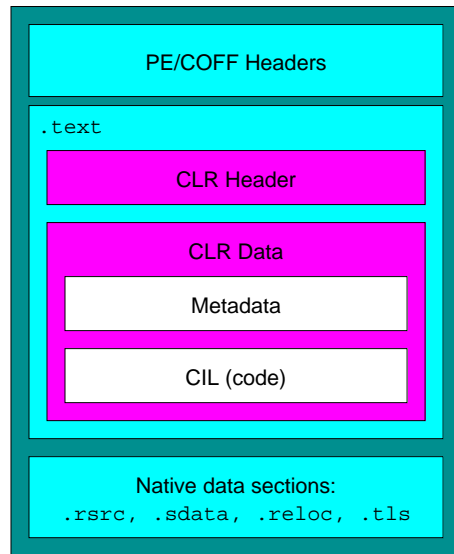


Figure 4.1: .NET CLR metadata and CIL are placed in the `.text` section of Microsoft PE/COFF modules.

For more detailed specification of PE/COFF as it is used in .NET modules and assemblies, please consult [14, Ch. 24, Partition II].

4.2 Metadata

Metadata is an essential part of .NET assemblies. They provide evidence about the code contained in the assembly, and also aid validation and verification of the assembly as it is loaded and JIT compiled.

There are two main types of metadata contained in an assembly and its modules: the assembly manifest and metadata about the data types defined. The manifest is contained in the main module of the assembly, and provides information on what other modules are part of the assembly, what external assemblies the assembly relies on, and the identity of the assembly. The type definition metadata provides a listing of the classes and their members defined in the modules, including information on inheritance. All classes inherit from `System.Object` for the sake of providing standard verification.

This metadata is organised in tables when stored in PE/COFF files. There is a different table type for each different purpose, ranging from the assembly describing table (`.assembly`) to the method parameter specification table (`.param`). These essentially provide the CLR with a highly specified object-oriented view of the code and all other aspects of the assembly. For a full listing of the tables the metadata can describe, see [14, Ch. 21, Partition II].

Probably the best way to describe metadata is by example. In Figure 4.2 is example C# for a rather overly complicated Hello World implementation. It contains two classes, one which simply acts as an entry point and calls the other. The `Printer` class demands two permissions, one imperative and one declarative. The reason for this will become more apparent in the later section on CIL.

The two files were compiled into modules and linked into a strong named assembly with the following commands using Microsoft Visual Studio .NET 2003:

```
csc /t:module Printer.cs
```

```
csc /t:module MainApp.cs /addmodule:Printer.netmodule
al /out:HelloWorld.exe /version:1.2.3.4 /target:exe
... /main:HelloWorld.MainApp.Main /keyfile:keypair.snk
... MainApp.netmodule Printer.netmodule
```

This results in three files: `HelloWorld.exe`, `MainApp.netmodule` and `Printer.netmodule`. The first file is the main module of the assembly, containing the manifest in its metadata section. This metadata section is given in Figure 4.3.

All metadata starts with a keyword with a period in front of it. The blocks denoted by `.assembly` make up the manifest of the assembly. These include a reference to the external assembly “mscorlib” using its Strong Name (with explicit version reference, assembly hash, and public key token), as well as a self-describing section. The self-describing section contains a few noteworthy keywords:

.custom This line refers to the file where the public key token was exported from. The data

MainApp.cs:

```
using System;

namespace HelloWorld
{
    public class MainApp
    {
        public static void Main()
        {
            Printer printer = new Printer();

            printer.PrintStatement();
        }
    }
}
```

Printer.cs:

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

namespace HelloWorld
{
    [FileIOPermission(SecurityAction.Demand, Read=@"C:\config.sys")]
    public class Printer
    {
        public void PrintStatement()
        {
            new FileIOPermission(FileIOPermissionAccess.Read,
                                 @"C:\autoexec.bat").Demand();
            Console.WriteLine("Hello World!");
        }
    }
}
```

Figure 4.2: Hello World, using two classes. In C#.

presented is in raw hex, but translates to a literal string giving the location of the original key file.

- .publickey** Contains the public key which the assembly has been signed with. This is considered part of the assembly's Strong Name (see Chapter 3).
- .hash algorithm** Refers to the type of algorithm used to create the hashes listed in the metadata.
- .ver** Gives the version of the assembly. This also contributes to the Strong Name.

Another field would have been present, denoting the culture of the assembly, if a culture had been defined. Culture information also contributes to the Strong Name of the assembly, but can only be specified in non-executable assemblies, i.e. assemblies that only contain resources. These assemblies are meant to be used to provide localised language support, and use the localisation codes from RFC 1766 [2].

The `HelloWorld.exe` metadata also contains references to the other modules which make up the assembly, plus information on the classes contained in those modules.

The metadata for the `MainApp` and `Printer` modules do not need to contain any manifest, but they do need to reference any external assemblies that they call. In our two modules (Figure 4.4 and Figure 4.5) we have references to “mscorlib”, as they both require that assembly. The `MainApp` module also references `Printer.netmodule`, because it directly relies on the functionality provided by it.

Other metadata is also present in the modules, and encapsulate the CIL code. Examples of this metadata can be found in Figure 4.6 and Figure 4.7 in the next section.

One noteworthy piece of metadata in Figure 4.7 is the `.permissionset` keyword. It describes a permission set which the class `Printer` demands before it can execute. If you look at Figure 4.2, you will see this maps back to the declarative permission demand for read access to “C:\\config.sys”.

The other metadata in Figure 4.6 and Figure 4.7 helps the assembly loader and class loader in verifying and validating the sanity of the assembly and classes. This type of metadata mostly describes type contracts, which describe the classes in the assembly and their sub-components. Information is provided about:

Classes Their accessibility (i.e. public, or private to the assembly), whether they are abstract, and what they inherit from and depend on.

Methods in classes Names, arguments, return types, accessibility (public, private, protected) and calling convention.

Fields in classes Names, types, accessibility (like methods), and whether they are static or not.

Declarative permissions What permissions have been declaratively hard-coded into the assembly, which the class requires to be granted.

Because this detailed information regarding the type declaration of the classes and its member methods and fields are present at runtime, the CLR can ensure that calls to and from methods are type safe. Of course, this only works if the metadata has not been corrupted, something the assembly and class loaders check.

For a comprehensive and in-depth description of what metadata can describe, and how it is encoded, please read [14, Partition II].

HelloWorld.exe Metadata Disassembled:

```

.module extern MainApp.netmodule
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .hash = (E6 8E F4 00 2B 3C 3C 88 D6 32 F2 72 A3 22 FA C8
          A7 7B 24 07 )
  .ver 1:0:5000:0
}
.assembly HelloWorld
{
  .custom instance void
    [mscorlib]System.Reflection.AssemblyKeyFileAttribute::.ctor(string)
    = ( 01 00 59 63 3A 5C 44 6F 63 75 6D 65 6E 74 73 20
      61 6E 64 20 53 65 74 74 69 6E 67 73 5C 6E 69 63
      6B 5C 4D 79 20 44 6F 63 75 6D 65 6E 74 73 5C 56
      69 73 75 61 6C 20 53 74 75 64 69 6F 20 50 72 6F
      6A 65 63 74 73 5C 48 65 6C 6C 6F 57 6F 72 6C 64
      5C 6B 65 79 70 61 69 72 2E 73 6E 6B 00 00 )
  .publickey = (00 24 00 00 04 80 00 00 94 00 00 00 06 02 00 00
    00 24 00 00 52 53 41 31 00 04 00 00 01 00 01 00
    F3 F8 19 0A 92 CF 22 B4 2C F3 2A 0B 97 88 FC 97
    D8 24 65 E4 AC 67 AD 44 4B A4 35 10 DE 65 CE 6B
    16 3B 6E 2B 26 53 63 FC 6D E6 F6 E6 50 48 95 4C
    97 A9 29 44 74 E9 44 57 F8 A6 FA D3 86 DB F3 F9
    D4 0D 6B 64 02 D4 FD C2 C2 F8 2F B3 12 8F B0 A3
    54 95 8D 67 6C CE 5C AE 16 A8 95 EE D9 66 86 00
    D8 C7 9F AF OF 3B D4 18 A9 OF A8 E3 50 81 88 68
    6E 40 B9 A6 50 93 3E EB C9 6A C2 4E 73 E8 C3 BD )
  .hash algorithm 0x00008004
  .ver 1:2:3:4
}
.file MainApp.netmodule
  .hash = (37 B9 C5 88 22 C7 0D 4C 75 6F 82 71 2A 32 5F 08
          5D 32 84 E5 )
.file Printer.netmodule
  .hash = (C3 73 99 7E D3 15 52 6E ED 33 CC 9B C2 88 28 48
          E5 CC 3D BA )
.class extern public HelloWorld.MainApp
{
  .file MainApp.netmodule
  .class 0x02000002
}
.class extern public HelloWorld.Printer
{
  .file Printer.netmodule
  .class 0x02000002
}
.module HelloWorld.exe
// MVID: {7F2C9415-960A-4278-A222-F3A0BB48B168}

```

Figure 4.3: The metadata portions of the HelloWorld.exe module from Figure 4.2.

MainApp.netmodule Metadata Disassembled:

```
.module extern Printer.netmodule
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .hash = (E6 8E F4 00 2B 3C 3C 88 D6 32 F2 72 A3 22 FA C8
          A7 7B 24 07 )
  .ver 1:0:5000:0
}
.module MainApp.netmodule
// MVID: {F477C914-5683-4033-8C3F-E09529494765}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x06b30000
//
// ===== CLASS STRUCTURE DECLARATION =====
//
.namespace HelloWorld
{
  .class public auto ansi beforefieldinit MainApp
    extends [mscorlib]System.Object
  {
  } // end of class MainApp
} // end of namespace HelloWorld
```

Figure 4.4: The metadata portions of the MainApp module from Figure 4.2.

Printer.netmodule Metadata Disassembled:

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .hash = (E6 8E F4 00 2B 3C 3C 88 D6 32 F2 72 A3 22 FA C8
          A7 7B 24 07 )
  .ver 1:0:5000:0
}
.module Printer.netmodule
// MVID: {F7D8630A-C0C8-4E24-9F4B-8DE8A2AC56B0}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x06b30000
//
// ===== CLASS STRUCTURE DECLARATION =====
//
.namespace HelloWorld
{
  .class public auto ansi beforefieldinit Printer
    extends [mscorlib]System.Object
  {
  } // end of class Printer
} // end of namespace HelloWorld
```

Figure 4.5: The metadata portions of the Printer module from Figure 4.2.

4.3 Common Intermediary Language

CIL can be described as an object-oriented assembly language for a stack-based virtual machine architecture. It provides an intermediary stage between the high level programming languages and the native assembly language for the architecture it is being run on. Due to this, it can both be easily validated and verified (thanks in part to metadata) and easily converted to local native code.

Example CIL can be found in Figure 4.6 and Figure 4.7, which are derived from the Hello World example in the section on metadata.

MainApp.netmodule CIL Disassembled:

```
.namespace HelloWorld
{
    .class public auto ansi beforefieldinit MainApp
        extends [mscorlib]System.Object
    {
        .method public hidebysig static void
            Main() cil managed
        {
            // Code size          13 (0xd)
            .maxstack 1
            .locals init (class [.module Printer.netmodule]HelloWorld.Printer V_0)
            newobj          instance void
                [.module Printer.netmodule]HelloWorld.Printer::.ctor()
            stloc.0
            ldloc.0
            callvirt        instance void
                [.module Printer.netmodule]
                HelloWorld.Printer::PrintStatement()
            ret
        } // end of method MainApp::Main

        .method public hidebysig specialname rtspecialname
            instance void .ctor() cil managed
        {
            // Code size          7 (0x7)
            .maxstack 1
            ldarg.0
            call            instance void [mscorlib]System.Object::.ctor()
            ret
        } // end of method MainApp::.ctor

    } // end of class MainApp
} // end of namespace HelloWorld
```

Figure 4.6: The CIL portion of MainApp.netmodule, compiled from the MainApp.cs code in Figure 4.2. Also contains metadata describing the type contracts.

In Figure 4.7 you can see an example of how the code executes in a stack-based manner. In the `PrintStatement` function, the 32-bit integer 1 (`ldc.i4.1`) is pushed onto the stack, followed by the string `"C:\\autoexec.bat"`. The integer represents the constant `FileIOPermissionAccess.Read` when passed as the first argument to the constructor of the `FileIOPermission` class. The string

Printer.netmodule CIL Disassembled:

```

.namespace HelloWorld
{
    .class public auto ansi beforefieldinit Printer
        extends [mscorlib]System.Object
    {
        .permissionset demand = (3C 00 50 00 65 00 72 00 ..... 3E 00 0D 00 0A 00 )
        .method public hidebysig instance void
            PrintStatement() cil managed
        {
            // Code size          27 (0x1b)
            .maxstack 3
            ldc.i4.1
            ldstr          "C:\\\\autoexec.bat"
            newobj         instance void
                [mscorlib]System.Security.Permissions.FileIOPermission::
                    .ctor(valuetype [mscorlib]System.Security.Permissions.
                        FileIOPermissionAccess,string)

            call          instance void
                [mscorlib]System.Security.CodeAccessPermission::Demand()
            ldstr          "Hello World!"
            call          void [mscorlib]System.Console::WriteLine(string)
            ret
        } // end of method Printer::PrintStatement

        .method public hidebysig specialname rtspecialname
            instance void .ctor() cil managed
        {
            // Code size          7 (0x7)
            .maxstack 1
            ldarg.0
            call          instance void [mscorlib]System.Object::.ctor()
            ret
        } // end of method Printer::.ctor

    } // end of class Printer
} // end of namespace HelloWorld

```

Figure 4.7: The CIL portion of Printer.netmodule, compiled from the Printer.cs code in Figure 4.2. This also contains metadata, including a `.permissionset` directive which specifies a permission set the module demands declaratively.

represents the second argument to the constructor. After the two values have been pushed onto the stack, the constructor for `FileIOPermission` is called. The constructor pops the two top values off the stack, creates an object based on its constructor logic and the two values, and then pushes the new object it creates onto the stack.

Because the top value on the stack is now the new `FileIOPermission` instance, the next instruction does not need to be preceded by a loading of the object onto the stack. Instead, the call to `CodeAccessPermission::Demand` can be made directly after the constructor call, as it will simply pop the top value off the stack and use it in its method logic.

What I have just described is not only an example of how CIL operates in a stack-based fashion,

but also how imperative permission demands appear in CIL form, just like any other function call.

Each CIL instruction is specified in the context of its action, the state transition it causes on the stack, any exception it may raise, and its verifiability [14, Chs. 2-4, Partition III]. This enables the CLR to verify that the stream of CIL it is loading is boundary and execution safe by checking the following:

Valid Stack State That each instruction leads from a valid stack state to another valid stack state. If an instruction requires two arguments, then there should be two arguments on the stack immediately before the instruction call. Also, the instruction should leave the stack in a valid state for the next instruction (i.e. there needs to be enough values on the stack for the next instruction).

Verifiability That the values on the stack are of the correct types for the instruction to accept them, and that the resulting stack state leaves the next instruction verifiable.

As an example, here is how the `cpobj` instruction is defined in [14, Ch. 4, p. 97, Partition III]:

Stack Transition:

$\dots, destValObj, srcValObj \rightarrow \dots,$

Description:

The `cpobj` instruction copies the value type located at the address specified by `srcValObj` (an unmanaged pointer, `native int`, or a managed pointer, `&`) to the address specified by `destValObj` (also a pointer). Behavior is unspecified if `srcValObj` and `destValObj` are not pointers to instances of the class represented by `classTok` (a `typeRef` or `typeDef`), or if `classTok` does not represent a value type.

Exceptions:

`NullReferenceException` may be thrown if an invalid address is detected.

Verifiability:

Correct CIL ensures that `classTok` is a valid `typeRef` or `typeDef` token for a value type, as well as that `srcValObj` and `destValObj` are both pointers to locations of that type.

Verification requires, in addition, that `srcValObj` and `destValObj` are both managed pointers (not unmanaged pointers).

The above specification declares that the `cpobj` instruction will transfer the stack from a state of having two pointer values at the top of the stack, to a state where these two values are no longer on the stack, and the rest of the stack remains as it was. This is illustrated in Figure 4.8.

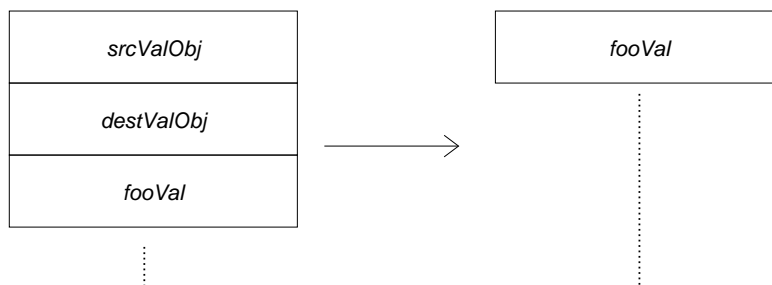


Figure 4.8: The `cpobj` instruction pops two values off the stack and pushes nothing onto it, causing the above stack state transition.

The specification also gives a human readable description of what the instruction does with the two pointers and the class token, declaring when behaviour is specified and when it is not.

As specified above, the instruction may throw a `NullReferenceException` if the instruction logic detects an invalid value either in `srcValObj` or `destValObj`.

The verifiability rules describe when the instruction is being used properly. It also specifies that for the code to pass CIL verification, the two pointers denoted by `srcValObj` and `destValObj` must be managed pointers. Notice that the human readable description is not this strict. This is an example of where code can execute as intended, but does not necessarily pass the verification tests.

For a complete listing of all CIL instructions and their verifiability rules, please consult [14, Partition II].

Chapter 5

The Common Language Runtime

This chapter will look closely at how code is treated in the .NET Framework CLR, and how the principles of CAS are enforced during the execution of managed code assemblies. The standard CLR infrastructure is specified in [14, Partition I].

The .NET CLR consists of several components through which assemblies pass before native code execution: the assembly loader, the policy manager, the class loader, and the Just-In-Time (JIT) compiler and verifier. Auxiliary components like the garbage collector, thread manager, exception manager and runtime security manager affect the native code execution through the code manager. The inter-operation of these components are laid out in Figure 5.1.

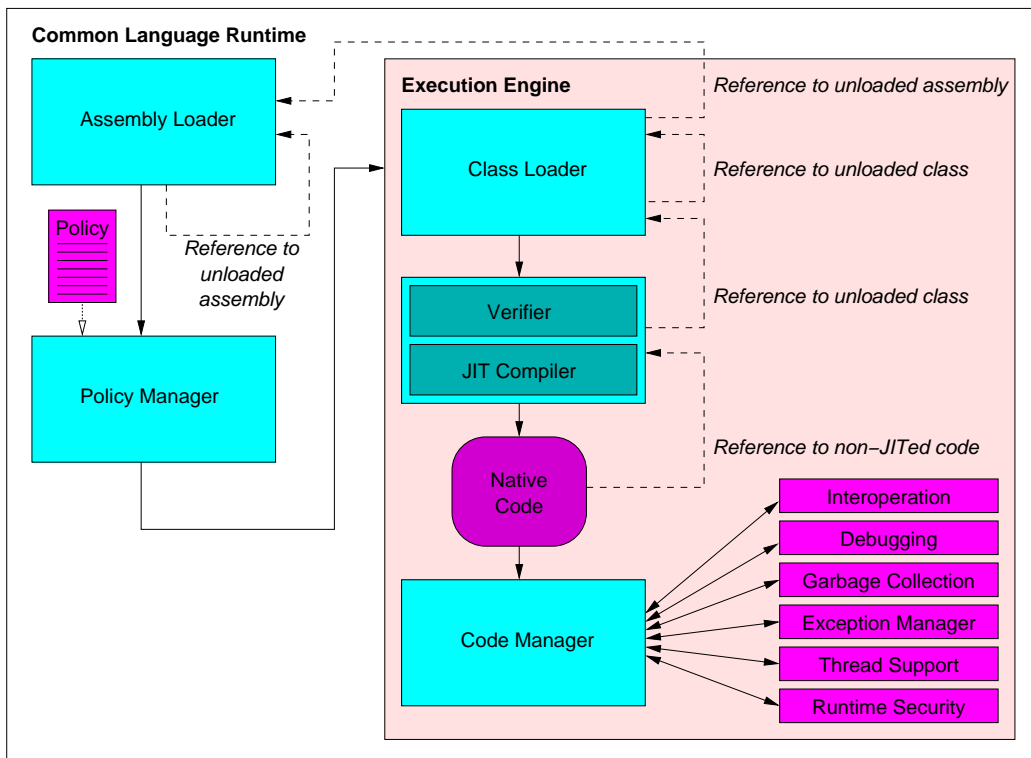


Figure 5.1: The stages of the .NET CLR execution environment. Note how references to unloaded or un-JITed code will cause recursive calls to previous components.

One important aspect of the CLR is how each component can recursively call other components. For example, if the JIT compiler encounters a reference to a class that has yet to be loaded, the class loader is called to load the missing class.

The following sections describe the different components in detail in the order they are called in the CLR. Each component serves a purpose in maintaining a valid secure runtime state, and the omission of one would depreciate the overall integrity of the CLR.

5.1 Assembly Loading

The Assembly Loader loads assemblies into memory and “binds” them to an Application Domain [19, Ch. 6]. It is the responsibility of the Assembly Loader to check the integrity of the assemblies and modules, so as not to introduce corrupted metadata or CIL into the CLR.

Assemblies can be loaded from the current directory, a sub-directory, the Global Assembly Cache (GAC), an intranet site, the Internet, or from a byte array (i.e. an assembly that has been dynamically emitted by managed code). The GAC is a local directory of assemblies which have been explicitly installed to be available to all assemblies running on the machine.

Assemblies can be referenced either by their “friendly” name (i.e. just assembly title), or by their Strong Name (title, version, public key and culture). It is up to the Assembly Loader to determine which version of the assembly to load if there are several versions available. When using Strong Naming, the Assembly Loader will also verify the signature on the assembly, based on the public key token found in the assembly’s metadata. If the verification fails, the assembly could have been tampered with, or has been corrupted. In this case, it is rejected, and no further parsing occurs.

In the case of single-module assemblies, the Assembly Loader only needs to load one module into memory. However, if the assembly consists of several modules, all the associated modules must be loaded. The assembly manifest may also contain references to other assemblies which the assembly relies on. As an example, most assemblies will reference `mscorlib`, from which the BCL is called. Both in the case of modules and other assemblies, the Assembly Loader must locate the files and check their integrity. Each module or assembly is referenced by their name and their cryptographic hash (as in Figure 4.3). The hash is verified against the module or assembly by computing fresh hashes of the files, and loading only succeeds if the reference hash and freshly computed hash are the same. Note in the case of references to other assemblies, the Assembly Loader will be called recursively by itself.

Only once these integrity checks are complete will the evidence, classes and resources be extracted from the assembly. Data structures are allocated in memory to represent the contents of the assembly, so as to be used by other components in the CLR.

5.2 Policy Management

The Policy Manager is responsible for assigning permissions to assemblies based on the security policies in place [15, Ch. 12]. Based on the permission set granted, the Policy Manager may not let the the assembly continue through to the Execution Engine components.

The Policy Manager takes three different inputs: the evidence from the assembly metadata and host environment (Section 3.1), the security policies (Enterprise, Machine, User, AppDomain - Section 3.3), and the declarative permission requests found in the assembly metadata (like in Figure 4.7).

Evidence comes in two forms: trusted evidence and untrusted evidence. Trusted evidence is evidence that has either been provided by the host environment (the host environment is assumed to be trustworthy), or that the CLR has been able to verify, like an assembly’s Strong Name or an Authenticode signature. Untrusted evidence is evidence which originates from the assembly and

which has not been verified by the CLR in some way. The default security policy for Microsoft's CLR does not consider untrusted evidence in its policy resolution.

To compute a security policy valid for the current assembly, the Policy Manager evaluates each policy level individually, before computing the intersection of the four policies (or three if no AppDomain level policy exists). The resulting policy describes the maximum set of permissions the Policy Manager will grant the assembly. Requests for permissions other than those granted by the policy will be denied.

Declarative permission requests are evaluated by the Policy Manager to determine whether the assembly will be able to execute at all. These requests give information on the following [15, Ch. 12]:

- Permissions the assembly does not wish to be granted.
- Permissions the assembly would like, but does not need for basic operation.
- Permissions the assembly must have to be able to execute properly.

If the minimum set of permissions required by the assembly is not a subset of the maximum set of permissions the Policy Manager has computed from the security policy, the assembly will not be able to execute. In this case, the Policy Manager will cease the parsing of the assembly, and will not allow it to enter the Execution Engine by throwing `PolicyException`.

There is another situation where the Policy Manager may not let the assembly continue to the Execution Engine: if the assembly is not granted the right to execute. This right to execute can be denied under conditions where the security policy resolves the assembly's permission set to not contain an instance of `SecurityPermission` with the `Execution` flag set. Again, `PolicyException` is thrown, and the Policy Manager will not let the assembly continue to the Execution Engine.

Once the set of granted permissions has been computed, and the cases where `PolicyException` may have been thrown have been passed, the Policy Manager associates the granted permission set with the in-memory data structures that represent the assembly contents, and passes control onto the Execution Engine through the Class Loader.

5.3 Class Loading

The Class Loader is responsible for allocating and laying out data structures in memory to represent classes and their member fields and methods, as well as verifying the visibility of class members. It does this based on the metadata provided by the modules of the assembly.

Classes are loaded in a lazy fashion, i.e. only when an instance of the class or a static member of the class is referenced will the class be loaded. Please note that the Class Loader does not deal with CIL, essentially deferring CIL parsing to the latest possible stage.

Once a class is loaded, the Class Loader must find the entry point in the class; the member method which is to be run on object instantiation (i.e. constructor). In the case of an executable assembly (as opposed to library assemblies), this includes finding the "Main" method.

The Class Loader instantiates the data structures for the member methods using stubs; pieces of native code that call the JIT compiler [19, Ch. 7]. Any native code reference to the member method references the location of the stub code. When called, the stub calls the JIT compiler, which compiles the member method's CIL into native code, and the stub then executes the compiled code. The next time the same member method is called, the stub simply executes the compiled method code rather than calling the JIT compiler.

There are four different types of stubs used in the Execution Engine [19, Ch. 7]:

Prestub The stub inserted by the Class Loader to invoke the JIT compiler for member methods.

Security stub Inserted to check the permission set granted by the Policy Manager.

Marshalling stub Inserted whenever a call to unmanaged code is made.

Remoting stub Inserted whenever a call across an Application Domain is made.

In summary, the Class Loader prepares the class and its members in memory for JIT compilation. It is important to note that the stub insertion and visibility verification are essential to the maintaining of a valid execution state, and thus security.

5.4 Just-in-Time Compilation and Verification

The JIT Compiler/Verifier is responsible for compiling CIL into native code, after having first verified the type safety of the CIL against the rules laid out in [14, Partition III].

5.4.1 Verification

There is no absolute algorithm which can verify whether CIL code is type-safe or not. Instead, a strict algorithm which will identify a majority of type-safe CIL is used in the verification of the assembly code. The downside of this is that some verifiable CIL is not permitted by the JIT Compiler/Verifier. The upside is that no type-unsafe code is let through to the actual stage of execution.

The Verifier is responsible for making sure the CIL instructions it encounters follow the verifiability rules. If an instruction is found to violate the rules, a `VerificationException` is thrown, and execution is halted unless the calling code catches the exception.

As an example, Figure 5.2 shows a slightly corrupted version of the constructor code for the `Printer` class from Figure 4.2. A `rem` instruction has been added to the start of the method. This instruction, in its verifiable use, pops two values off the stack, computes the remainder of dividing one by the other, and pushes the remainder onto the stack. The stack state transition diagram for this instruction [14, p. 80, Partition III] describes this state transition.

```
.method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
{
    // Code size          7 (0x7)
    .maxstack 2
    rem
    ldarg.0
    call      instance void [mscorlib]System.Object::.ctor()
    ret
} // end of method Printer::.ctor
```

Figure 5.2: Modified `Printer` constructor. Compare to Figure 4.7.

The problem with adding this instruction to the beginning of the method, is that there are no values on the stack. Such an instruction may have been inserted to test if the Execution Engine protects against stack underflows. The result of running the assembled code can be found in Figure 5.3.

One very important issue to note is that all assemblies executed locally skip verification, at least under the default security policy. This is because all local assemblies are by default granted the

```
C:\Documents and Settings\nick\My Documents\Visual Studio Projects\HelloWorld\
haxored>HelloWorld.exe
```

```
Unhandled Exception: System.Security.VerificationException: Operation could
destabilize the runtime.
   at HelloWorld.Printer..ctor()
   at HelloWorld.MainApp.Main()
```

Figure 5.3: The result of executing the HelloWorld assembly when using the corrupted `Printer` constructor from Figure 5.2. Please note the assembly has not been granted `SkipVerification`, otherwise the verification would not have been performed.

`FullTrust` permission set, which includes the `SkipVerification` permission. This introduces the possibility of malicious code being allowed to execute without verification, as long as the malicious code can be installed onto the local machine before execution. Depending on the user, this could easily be achieved by social engineering [17].

As an example of what happens when verification is skipped, the code in Figure 5.4 introduces another type of verifiability violation in `Printer.PrintStatement`, and Figure 5.5 shows the result when it is executed with `SkipVerification` having been granted.

```
.method public hidebysig instance void
    PrintStatement() cil managed
{
    // Code size      27 (0x1b)
    .maxstack 3
    ldc.i4.1
    ldstr      "C:\\autoexec.bat"
    newobj     instance void
                [mscorlib]System.Security.Permissions.FileIOPermission::
                .ctor(valuetype [mscorlib]System.Security.Permissions.
                FileIOPermissionAccess,string)
    call      instance void
                [mscorlib]System.Security.CodeAccessPermission::Demand()
    ldc.i4.1
    call      void [mscorlib]System.Console::WriteLine(string)
    ret
} // end of method Printer::PrintStatement
```

Figure 5.4: Modified `Printer.PrintStatement`. Loads the 32-bit integer value ‘1’ onto the stack, instead of a string. Compare to Figure 4.7.

The modified version of `Printer.PrintStatement` has replaced the loading of the “Hello World!” string onto the stack with the loading of the 32-bit integer value ‘1’. This kind of malicious code may be crafted to attempt to trick the runtime into parsing the integer as a reference to somewhere in memory, and possibly expose hidden values in memory. In this case, no integrity is breached (Figure 5.5), but the type discrepancy survives three method calls into the Standard Libraries.

Although the above example can cause lack of availability, it does not affect confidentiality of information, which can often be the goal of an attack. Confidentiality can be breached, however, by accessing member fields and methods which are not supposed to be visible outside a class instance. This is possible when `SkipVerification` is granted.

Figure 5.6 shows the C# code for a fairly simplistic example of a class with a private member

```
C:\Documents and Settings\nick\My Documents\Visual Studio Projects\HelloWorld\
haxored>HelloWorld.exe
```

```
Unhandled Exception: System.NullReferenceException: Object reference not set to
an instance of an object.
   at System.IO.TextWriter.WriteLine(String value)
   at System.IO.SyncTextWriter.WriteLine(String value)
   at System.Console.WriteLine(String value)
   at HelloWorld.Printer.PrintStatement()
   at HelloWorld.MainApp.Main()
```

Figure 5.5: The result of executing the HelloWorld assembly when using the corrupted `Printer.PrintStatement` method from Figure 5.4 and with `SkipVerification` granted.

field. In real life circumstances, this field would contain slightly more interesting information, such as authentication details, or access privileges, which the attacker either wishes to obtain (breach of confidentiality) or alter (breach of integrity).

```
public class VictimClass
{
    private String secret = "This is the secret string!";

    public String official = "This is the public string.";

    public String getString()
    {
        return official;
    }
}
```

Figure 5.6: The `VictimClass`, which contains a private field we want to access. The `getString` method returns a reference to the public string `official`, and its based on this reference that we can obtain the private string `secret`.

To gain access to the private field, the exploiting code must figure out the memory address where the field is stored, so as to access it directly. The exploit in Figure 5.7 calls the `getString` method on an instance of `VictimClass` to obtain a reference to the public string `official`. Through some educated guessing, I was able to determine that the private field `secret` was located 72 bytes ahead of the public field `official`. By subtracting 72 from the address of `official`, the top value on the stack is the memory address of the private field. To view the content of the private string, I use the standard `Console.WriteLine` method call.

Because of the pointer arithmetic involved, the exploit had to be written directly in CIL, as the C# compiler would not compile the exploit due to the unsafe operations. The result of executing the exploit both without and with `SkipVerification` granted can be found in Figure 5.8.

The exploit I have described is not unique to .NET, and can be performed on any native code which is compiled from object-oriented code and runs outside of a sandbox environment. However, it is important to note that .NET is such an environment that attempts to guard against these kinds of attacks. This type of attack should therefore ideally not be possible. Of course, changing the default security policy to not grant `SkipVerification` quite as frivolously should resolve the issue.

```
.method public hidebysig static void
    Main() cil managed
{
    .entrypoint
    // Code size      20 (0x14)
    .maxstack 2
    .locals init (class VerificationExploit.VictimClass V_0, string V_1)
    newobj      instance void VerificationExploit.VictimClass::.ctor()
    stloc.0
    ldloc.0
    callvirt   instance string VerificationExploit.VictimClass::getString()
    stloc.1
    ldloc.1
    ldc.i4     72          // Push 72 onto stack
    sub                // Subtract 72 from the address of victim.official
    call       void [mscorlib]System.Console::WriteLine(string)
    ret
} // end of method ExploitClass::Main
```

Figure 5.7: The assembly for the exploiting function. The code calls the `getString` method on the instance of `VictimClass` (Figure 5.6, subtracts 72 from the memory address, and prints what it finds. The result with and without `SkipVerification` granted can be found in Figure 5.8.

```
C:\Documents and Settings\nick\My Documents\Visual Studio Projects\
VerificationExploit>exploit2.exe

Unhandled Exception: System.Security.VerificationException: Operation could
destabilize the runtime.
   at VerificationExploit.ExploitClass.Main()

C:\Documents and Settings\nick\My Documents\Visual Studio Projects\
VerificationExploit>exploit2.exe
This is the secret string!
```

Figure 5.8: The result of running the exploit code from Figure 5.7. The first run shows the result of executing the exploit without `SkipVerification` granted, while the second run is with `SkipVerification` granted.

If a successful attack on a widely installed assembly can be found using the `SkipVerification` loophole, all that is required is some social engineering to trick the user into executing the malicious code locally. Recent e-mail worms have shown this to be relatively easy [24].

5.4.2 JIT compilation

If the CIL passes through the Verifier successfully, or verification is skipped, the Just-In-Time Compiler will compile the CIL to native code.

The major security activity of the JIT Compiler is to insert stubs into the native code where imperative security demands appear. These stubs call the Code Manager, which passes control to the Runtime Security component. The JIT Compiler also injects stubs for the other auxiliary components.

Coming back to the `SkipVerification` issues described in the previous chapter, I would speculate that it could be possible to override calls to the runtime stubs in specially crafted assemblies when `SkipVerification` has been granted. As an example, runtime permission checks are made by calling a security stub before the actual code which requires the permission. If a piece of code referenced the requiring code directly – i.e. by memory location rather than method name – the security stub would be bypassed (Figure 5.9). This is purely a theoretical attack, and I have not had the opportunity to investigate it further. There may possibly be integrity checks after the code has been called which would make this type of attack impossible.

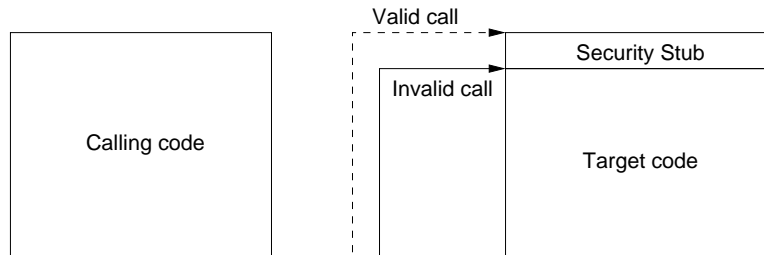


Figure 5.9: A theoretical attack where a direct reference to code which requires a permission bypasses the security stub. This has not been tested, and mechanisms may be in place to prevent this type of attack.

5.5 Code Management and Security

The Code Manager provides essential auxiliary functionality during the execution of native code, which include:

Garbage Collection Freeing up of memory when the objects contained within the memory are no longer referenced by any code on the execution stack.

Exception Handling Exceptions are used throughout the CLR and in the assemblies. Code to handle exceptions must be indexed and called when they are required.

Debugging Because of the amount of metadata and other information available at runtime, debugging can be performed as part of the runtime.

Threading Assemblies can choose to branch their execution into multiple threads. The Thread Support component maps these Application Domain threads to native operating system threads.

Interoperation When calls to native unmanaged code (for example COM components), these must be handled through marshaling.

As well as the above functionality, the Code Manager also provides a Runtime Security component. This component is called by all the stubs injected by the Class Loader and JIT Compiler whenever a security decision must be made. It also interacts with the other auxiliary components; the Exception Handler is called when the Runtime Security component throws a security related exception, the Interoperation component requires security checks to be successful before it can marshal code, and the Debugging component is ultimately called if an exception is raised but not caught.

It is during the execution of these stubs that refer to auxiliary components that stack walks occur.

The Code Manager is essentially responsible for the execution of any runtime security operations. Without these operations, the stub injection by the JIT Compiler and the Class Loader would be pointless, which in turn would render large parts of the CAS principles useless.

Chapter 6

Conclusions

The .NET Framework is a virtual machine execution environment which accommodates object-oriented cross-platform software development. It bases its security on the principles of CAS, which uses evidence about the executing assemblies and a four-level security policy to grant permissions to the assemblies. The principles are enforced in the metadata and CIL code found in the assemblies, and through the different components of the CLR.

Although the Framework appears to take security quite seriously, there are still a couple of issues I have identified which I believe need to be addressed.

Firstly, there is the ability for developers to override stack walks checking for granted permissions, by asserting permissions in their code which calling assemblies may not have. Although this type of feature is often necessary, it is a dangerous feature to use unwittingly. As an improvement, I would suggest that only highly trusted code is allowed to assert permissions on behalf of calling assemblies. This could be implemented by having a specific `AssertPermission`, which is only granted as part of the `FullyTrusted` permission set. Another solution would be in the form of developer education. The compilers could be implemented so as to produce warnings whenever an assert is found in the source files, explaining the hazards of using asserts in assemblies. It is questionable whether developers would pay attention to such warnings, of course.

The other issue I have identified is the ability of assemblies to be granted the `SkipVerification` permission when run locally. This issue is specific to the default security policy in the Microsoft implementation of the Framework, and may not exist in other default security policies. In the default security policy, all code running locally will be fully trusted, and thus will not be subject to verification when being processed by the CLR. As demonstrated in Chapter 5, this can have devastating consequences for the integrity of the execution environment.

My suggestion for fixing this issues would be to ship the Framework with a much stricter default security policy. This stricter policy would only grant `SkipVerification` to core assemblies that are part of the Framework implementation, or not grant the permission at all. This may cause a speed decrease in execution, but would safeguard against any malicious unverifiable code from doing any damage, even when run locally.

Of course, the security of the execution environment also depends on the implementation of the Framework. Proper auditing of Framework implementation code should take place, so as to prevent any future exploits like the Donut virus.

It remains to be seen whether any serious exploits can arise from the issues I have addressed above. Future work could look into what unverified code can do to the various Framework implementations, and also what verified code is able to do that may breach the security of the host system. The usability of security policy management should also be studied, to see if policies are actually changed from the default or not.

In my opinion, the Framework does not so much solve the issue of writing secure software, but rather shifts the responsibility of ensuring that the software cannot harm the target computer system. A lot of functionality is present in the Framework to safeguard execution, but it is up to the users and administrators to ensure that adequate policies that utilise the functionality are in place. This is in contrast to the more traditional view that it is the responsibility of the developer to ensure secure code execution [26]. As long as a secure default policy is in place, this shift of responsibility should not be an issue.

Bibliography

- [1] V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, April 1986.
- [2] H. Alvestrand. Tags for the Identification of Languages. March 1995. URL <http://www.ietf.org/rfc/rfc1766.txt>.
- [3] T. Berners-Lee, L. Masinter, and M. McCahill. RFC 1738 - Uniform Resource Locators (URL), December 1994. URL <http://www.ietf.org/rfc/rfc1738.txt>.
- [4] Don Box. The Security Infrastructure of the CLR Provides Evidence, Policy, Permissions, and Enforcement Services. *MSDN Magazine*, September 2002. URL <http://msdn.microsoft.com/library/default.asp?url=/msdnmag/issues/02/09%/SecurityinNET/TOC.ASP>.
- [5] Don Box and Chris Sells. *Essential .NET, Volume I: The Common Language Runtime*. Addison Wesley Professional, November 2002.
- [6] Microsoft Corporation. Introduction to Code Signing, . URL http://msdn.microsoft.com/library/default.asp?url=/workshop/security/au%thcode/intro_authenticode.asp.
- [7] Microsoft Corporation. Introduction to URL Security Zones (Internet Explorer - Security), . URL <http://msdn.microsoft.com/workshop/security/szone/overview/overview.asp%>.
- [8] Microsoft Corporation. Overview of the .NET Framework, . URL <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguid%e/html/cpovrIntroductionToNETFrameworkSDK.asp>.
- [9] Microsoft Corporation. Information About the .NET W32.Donut Virus. 2002. URL <http://support.microsoft.com/default.aspx?scid=kb;en-us;316287>.
- [10] Abhijit Dharia and Rahul Phadnavis. Internals of .NET/Rotor CLR, 2004. URL <http://wiki.cs.uiuc.edu/cs427/.Net+CLR+Internals>.
- [11] Dieter Gollman. *Computer Security*. John Wiley and Sons Ltd., July 2003.
- [12] R. Housley, W. Polk, W. Ford, and D. Solo. Internet X.509 Public Key Infrastructure - Certificate and Certificate Revocation List (CRL) Profile, April 2002. URL <http://www.ietf.org/rfc/rfc3280.txt>.
- [13] ECMA International. C# Language Specification, December 2002. URL <http://www.ecma-international.org/publications/standards/Ecma-334.htm>.
- [14] ECMA International. Common Language Infrastructure (CLI) Partitions I to V, December 2002. URL <http://www.ecma-international.org/publications/standards/Ecma-335.htm>.
- [15] Brian A. LaMachia, Sebastian Lange, Matthew Lyons, Rudi Martin, and Kevin T. Price. *.NET Framework Security*. Addison Wesley, 2002.

- [16] John Leyden. Donut virus highlights holes in .Net. January 2002. URL http://www.theregister.co.uk/2002/01/10/donut_virus_highlights_holes/.
- [17] Kevin D. Mitnick and William L. Simon. *The Art of Deception: Controlling the Human Element of Security*. John Wiley & Sons Inc, October 2002.
- [18] Inc. Networks Associates Technology. Virus Profile - W32/Donut. January 2002. URL http://us.mcafee.com/virusInfo/default.asp?id=description&virus_k=99300%.
- [19] Gary Nutt. *Distributed Virtual Machines: Inside the Rotor CLI DRAFT*. May 2004. URL <http://www.nuvolan.com/~nutt/DPRS/toc.html>.
- [20] Aleph One. Smashing The Stack For Fun and Profit. *Phrack Issue 49*. URL <http://www.insecure.org/stf/smashstack.txt>.
- [21] Frank Piessens. Developing Secure Software Applications - Draft, November 2003. URL <http://www.cs.kuleuven.ac.be/~frank/OVS/ovs.pdf>.
- [22] Denis Piliptchouk and Vince Dovydaitis. Securing .NET and Enterprise Java: Side by Side. *Computer Security Journal*, XVIII(3-4), 2002. URL <http://www.foliage.com/whitepapers/>.
- [23] James Seward. Under the Hood of .NET - Looking into assemblies. *HardCopy Issue 15*, 2002. URL <http://www.greymatter.com/developers/hardcopy/issue15/>.
- [24] Pete Simpson. Lessons We Can Learn From Recent Mass Mailing Worms. February 2002. URL <http://www.itsecurity.com/papers/mime2.htm>.
- [25] Thuan Thai and Hoan Q. Lam. *.NET Framework Essentials*. O'Reilly, 3rd edition, August 2003.
- [26] John Viega and Gary McGraw. *Building Secure Software*. Addison-Wesley, 2002.
- [27] Dr. Demien Watkins and Sebastian Lange. An Overview of Security in the .NET Framework, January 2002. URL <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnets%ec/html/netframesecover.asp>.
- [28] David A. Wheeler. Program Library HOWTO, April 2003. URL <http://www.tldp.org/HOWTO/Program-Library-HOWTO/index.html>.