SECURITY ARCHITECTURES IN THE MICROSOFT .NET FRAMEWORK

OLIVER ANDRE HOPPE
Microsoft Research Limited (Cambridge, United Kingdom) t-ohoppe@microsoft.com

Key words: Microsoft .NET framework, security, role–based security, code-based security

Abstract: The current trend in today's software development and deployment is to distribute software functionality within a distributed environment. This trend is clearly illustrated by Microsoft's drive towards promoting software functionality as loosely coupled web services that can be accessed by common Internet protocols. As a result, many computer systems are becoming increasingly complex leading to increased vulnerability to security threats. To address these concerns, Microsoft has integrated a rich series of security features within the .NET Framework. This paper will serve to provide an overview of the Microsoft .NET Framework security architecture, including code-based security, role-based security and verification.

1. INTRODUCTION

During the last few decades the use of information technology (IT) has become more widespread in all areas of society, and the types of activities that it performs or supports, have become increasingly more important. As a result, information systems are now heavily utilized by all organizations and relied upon to the extent that it would be impossible to manage without them (Hutchinson, B. & Warren, M. 1999, p. 42).

As computer systems become more interconnected, their complexity increases, and so does the threat to their security. The trend toward distributing software over the Internet so it can run on multiple computers increases security risks. Parts of programs previously hidden when run on a single computer are exposed to the outside world, and to attacks, when operated over networks.

Not only is the Internet increasing the level of exposure that software has to security threats, but customers trust more of their valuable data to computer systems, both at businesses and in the home. So now more than ever, secure software is becoming a critical concern.

To address these concerns, the Microsoft .NET Framework provides a rich security system, capable of confining code to run in tightly constrained, administrator-defined security contexts (Simon, D. 2001, p2). This paper examines some of the fundamental security features in the .NET Framework. However, before we discuss how the .NET framework accomplishes this, it is important to provide a conceptual overview of the .NET Framework itself.

2. THE MICROSOFT .NET PARADIGM

The .NET Framework promises to provide a new platform for building integrated, service-oriented applications to meet the needs of today's Internet businesses. The common language runtime and class libraries combine together to provide services and solutions that can be easily integrated within and across a variety of systems. The .NET Framework provides a fully managed, protected, and feature-rich application execution environment, simplified development and deployment, and seamless integration with a wide variety of languages (Microsoft Research Security Team. 2001, p3).

The .NET Framework has two main components: the common language runtime and the .NET Framework class library. The common language runtime is the foundation of the .NET Framework. The runtime is an agent that manages code at execution time, providing core services such as memory management, thread management, and remoting while also enforcing strict type safety and other forms of code accuracy that ensure security and robustness (Microsoft Cooperation. 2001).

The runtime executes both managed and unmanaged code. Managed code executes under the control of the runtime and therefore has access to services provided by the runtime, such as memory management, just-in-time (JIT) compilation, and, most importantly as far as this paper is concerned, security services, such as the security policy system and verification.

Unmanaged code is code that has been compiled to run on a specific hardware platform and cannot directly utilize the runtime. However, when language compilers emit managed code, the compiler output is represented as Microsoft intermediate language (MSIL). MSIL is often described as resembling an object-oriented assembly language for an abstract, stack-based machine. MSIL is said to be object-oriented, as it has instructions for supporting object-oriented concepts, such as the allocation of objects and virtual function calls. It is an abstract machine, as MSIL is not tied to any specific platform. That is, it makes no assumptions about the hardware on which it runs. MSIL is typically JIT-compiled to native code prior to execution.

The class library, the other main component of the .NET Framework, is a comprehensive, object-oriented collection of reusable types that can be used to develop applications ranging from traditional command-line or graphical user interface (GUI) applications to applications based on the latest innovations provided by ASP.NET, such as Web Forms and XML Web services (Microsoft Cooperation. 2001).

Security is one of the most important issues to consider when moving from traditional program development to an environment that allows for dynamic downloads and execution and even remote execution (Simon, D. 2001, p2). To support this model, the Microsoft .NET Framework provides a variety of rich security architectures.

3. THE MICROSOFT .NET FRAMEWORK SECURITY MODEL

The .NET Framework security solution is based upon the concept of managed code, with security rules enforced by the common language runtime (CLR). Most managed code is verified to ensure type safety, as well as the well-defined behaviour of other properties. Verification prevents code that is not type safe from executing, and catches many common programming errors before they cause damage. Common vulnerabilities - such as buffer overruns - are no longer possible (Simon, D. 2001, p2). This benefits end users, because the code they run is checked before it executes. It also benefits developers, who will find that many of the common bugs that have traditionally plagued development are now identified and prevented from causing harm. The following sub-sections will serve to provide greater insight into the security mechanisms employed by the .NET security framework.

3.1 VERIFICATION

There are two forms of verification done in the runtime. MSIL is verified and assembly metadata is validated. All types in the runtime specify the contracts that they will implement, and this information is persisted as metadata along with the MSIL in the managed PE/COEFF file. For example, when a type specifies that it inherits from another class or interface, indicating that it will implement a number of methods, this is a contract. A contract can also be related to visibility. For example, types may be declared as public from their assembly or not. Type safety is a property of code in so much as it only accesses types in accordance with their contracts. MSIL can be verified to prove it is type safe. Verification is a fundamental building block in the .NET Framework security system; currently verification is only performed on managed code (Microsoft Cooperation. 2001). Unmanaged code executed by the runtime must be fully trusted, as it cannot be verified by the runtime.

In order to understand MSIL verification, it is important to understand how MSIL can be classified. MSIL can be classified as invalid, valid, type safe, and verifiable (Microsoft Cooperation. 2001).

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

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

For code loaded from a disk, the verification process is part of the JIT compiler and proceeds intermittently within the JIT compiler. Verification and JIT compilation are not executed as two separate processes. If, during verification, a sequence of unverifiable MSIL is found within an assembly, the security system checks to see if the assembly is trusted enough to skip verification. For example, if an assembly is loaded from a local hard disk, under the default settings of the security model, then this could be the case. If the assembly is trusted to skip verification, then the MSIL is translated into native code. If the assembly is not trusted enough to skip verification then the offending MSIL is replaced with a stub that throws an exception if that execution path is exercised (Simon, D. 2001, p4).

In addition to MSIL verification, assembly metadata is also verified. In fact, type safety relies on these metadata checks, for it assumes that the metadata tokens used during MSIL verification are correct. Assembly metadata is either verified when an assembly is loaded into the Global Assembly Cache (GAC), or Download Cache, or when it is read from a disk if it is not inserted into the GAC. (The GAC is a central storage for assemblies that are used by a number of programs. The download cache holds assemblies downloaded from other locations, such as the Internet.) Metadata verification involves examining metadata tokens to see that they index correctly into the tables they access and that indexes into string tables do not point at strings that are longer than the size of buffers that should hold them, eliminating buffer overflow. The elimination, through MSIL and metadata verification, of type-safe code that is not type-safe is the first part of security on the runtime (Microsoft Cooperation. 2001).

The CLR also enables *unmanaged code* to run, but unmanaged code does not benefit from these security measures. Specific permissions are associated with the capability to call into unmanaged code, and a robust security policy will ensure that those permissions are conservatively granted. The migration from unmanaged code to managed code will, over time, reduce the frequency of calls to unmanaged code.

The .NET Framework also provides security on code and this is referred to as code access security (also referred to as evidence-based security). With code access security, a user may be trusted to access a resource but if the code the user executes is not trusted, then access to the resource will be denied. Security based on code, as opposed to specific users, is a fundamental facility to permit security to be expressed on mobile code. Mobile code may be downloaded and executed by any number of users all of which are unknown at development time. Code Access Security centres on some core abstractions, namely: evidence, policies, and permissions.

3.2 CODE ACCESS SECURITY

Most common security mechanisms give rights to users based on their logon credentials (usually a password) and restrict resources (often directories and files) that the user is allowed to access. However, this approach fails to address several issues: users obtain code from many sources, some of which might be unreliable; code can contain bugs or vulnerabilities that enable it to be exploited by malicious code; and code sometimes does things that the user does not know it will do. As a result, computer systems can be damaged and private data can be leaked when cautious and trustworthy users run malicious or error-filled software. Most operating system security mechanisms require that every piece of code must be completely trusted in order to run, except perhaps for scripts on a Web page. Therefore, there is still a need for a widely applicable security mechanism that allows code originating from one computer system to execute safely on another system, even when there is no trust relationship between the systems (Microsoft Cooperation. 2002).

To help protect computer systems from malicious mobile code, to allow code from unknown origins to run safely, and to protect trusted code from intentionally or accidentally compromising security, the .NET Framework provides a security mechanism called code access security.

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

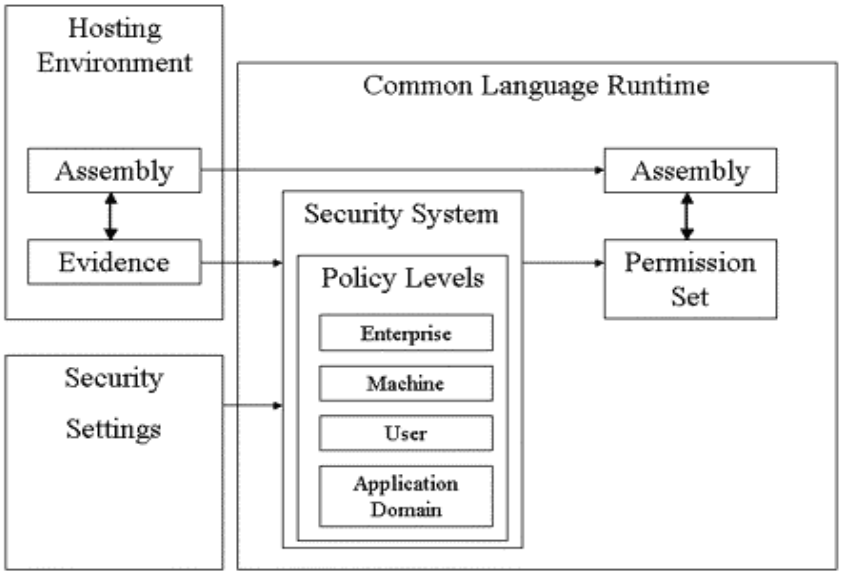
the file system. This mapping is based on an administrable security policy (Microsoft Cooperation. 2001).

A security policy configured by the administrator or user specifies rules of using evidence to determine permissions to grant to code. After ensuring that the minimum permissions can be satisfied, the code requests can be given, permissions are granted and the code runs limited by what the permissions allow it to do. If policy would grant code less than the minimum it requests, then the code is not run. The permission request also allows the code to be examined at deployment time to learn what permissions the developer declared it needs.

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

The process of determining the actual set of granted permissions to an assembly is a three-fold procedure (Microsoft Cooperation. 2001):

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



1.1 Calculation of the set of granted permissions of an assembly (Microsoft Cooperation. 2001).

Figure 1.1 provides an abstracted overview. The host of the runtime supplies evidence about an assembly that acts as one of the inputs for the calculation of the permission set the assembly receives. The administrable security policy (enterprise, machine and user policy), referred to above as security settings, is the second input that determines the calculation of a permission set of the assembly. The security policy code, then traverses the policy level settings given the evidence of the assembly and produces the permission set that represent an assembly's set of rights to access protected resources.

Policy levels express a self-contained, configurable security policy unit - each level mapping assembly evidence to a set of permissions. Each policy level has a similar architecture. Each level consists of three constituents that are used in combination to express the configuration state of a policy level (LaMacchia, B. 2002, p5):

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

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

3.2.1 Code Group

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

3.2.2 Named Permission Sets

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

- **FullTrust:** Allows unrestricted access to system resources.
- **SkipVerification:** Allows an assembly to skip verification.

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

3.2.3 Policy Assemblies

During security evaluation, other assemblies might need to be loaded to be used in the policy evaluation process. For example, an assembly can contain a user-defined permission class part of a permission set handed out by a code group. Of course, the assembly containing the custom permission also needs to be evaluated. If the assembly of the custom permission is granted the permission set containing the custom permission it itself implements, then a circular dependency ensues. To avoid this, each policy level contains a list of trusted assemblies that it needs for policy evaluation. The list of required assemblies is naturally referred to as the list of "Policy Assemblies", and contains the transitive closure of all assembly required to implement security policy at that policy level. Policy evaluation for all assemblies contained in that list is short circuited to avoid the occurrence of circular dependencies (Microsoft Cooperation. 2002).

This completes the examination of the configurable constituents of each policy level: a tree of code groups, list of named permission sets and a list of policy assemblies.

Code access security revolves around the identity of code, as opposed to user identity. This allows code to run under a single user context in an indefinite number of trust levels. For instance, code coming from the Internet can run in restrictive security boundaries, even if the operating system user context in which it runs would allow full access to all system resources. However, there is still a need to be able to express security settings based on user identities. The runtime security system therefore also ships with role-based security features.

3.3 ROLE BASED SECURITY

Role-based security utilizes the concepts of users and roles, which is similar to the implementation of security in many current operating systems. Two core abstractions in role-based security are Identity and Principal. Identity represents the user on whose behalf the code is executing. It is important to remember that this could be a logical user as defined by the application or developer and not necessarily the user as seen by the operating system. A Principal represents the abstraction of a user and the roles in which a user belongs (Microsoft Cooperation. 2002).

The .NET Framework role-based security supports authorization by making information about the principal, which is constructed from an associated identity,

available to the current thread. The identity (and the principal it helps to define) can be either based on a Windows account or be a custom identity unrelated to a Windows account. The .NET Framework applications can make authorization decisions based on the principal's identity or role membership, or both. A role is a named set of principals that have the same privileges with respect to security (such as a teller or a manager). A principal can be a member of one or more roles. Therefore, applications can use role membership to determine whether a principal is authorized to perform a requested action (Microsoft Research Security Team. 2001, p6).

The .NET Framework provides role-based security support that is flexible and extensible enough to meet the needs of a wide spectrum of applications.

4. CONCLUSION

Security is a fundamental aspect built into the .NET framework. The integration of the above mentioned security architectures, in combination with other forms of security such as cryptography, web application security and stack flow, provide a secure framework that ensures that the execution of all managed code, and to some extent unmanaged code, is restricted to a well administered security contexts, thereby providing a robust and secure platform for the development and deployment of both local and distributed applications.

5. REFERENCES

Hutchinson, B. & Warren, M. (1999). The Future of Australian & New Zealand Security Standard AS/NZA 4444?. In J.H.P. Eloff & L. Labuschagne & R. von Solms & J. Verschuren (Eds.). Information Security Management & Small Systems Security (pp. 41 - 49). United States of America : Kluwer Academic Publishers.

LaMacchia, B (2002). The .NET Framework Security Infrastructure [online]. Cited [April 19, 2002] Internal Microsoft Document.

Microsoft Cooperation (2002). Code Access Security [online]. Cited [February 28, 2002] Available from Internet URL http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconcodeaccesssecurity.asp.

Microsoft Cooperation (2002). Introduction to Role-Based Security [online]. Cited [May 11, 2002] Available from Internet URL http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconintroductiontorole-basedsecurity.asp.

Microsoft Cooperation (2001). Overview of the .NET Framework [online]. Cited [February 23, 2002] Available from Internet URL

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpovrintroductiontonetframeworksdk.asp.

Microsoft Research Security Team (2001). A Security Model for .NET [online]. [Cited March 22, 2002] Internal Microsoft Document.

Simon, D (2001). A Model for .NET Security [online]. Cited [March 04, 2002] Internal Microsoft Document.

Simon, D (2001). Security Mechanisms in .NET [online]. Cited [March 04, 2002] Internal Microsoft Document.