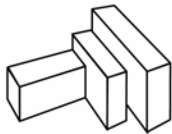
Security in the Microsoft® .NET Framework

An Analysis by Foundstone, Inc. and CORE Security Technologies



Abstract

This paper presents an overview of the security architecture of Microsoft's .NET Framework This paper is based on a long-term, independent security analysis performed by Foundstone, Inc. and CORE Security Technologies, beginning in the summer of 2000.

Our analysis revealed that, used properly, the .NET Framework gives developers and administrators granular security control over their applications and resources; provides developers with an easy-to-use toolset to implement powerful authentication, authorization, and cryptographic routines; eliminates many of the major security risks facing applications today due to flawed code (such as buffer overflows); and shifts the burden from having to make critical security decisions -- such as whether or not to run a particular application or what resources that application should be able to access -- from end users to developers and administrators.

In the course of this document, we will explain how the .NET Framework's evidence- and role-based security features, code access security, verification process, cryptography support, isolated storage, and application domains work together to achieve these outcomes, providing a robust platform for developing and running all types of software applications, both client- and server-side. We conclude that the .NET Framework can provide organizations with greater assurance that their applications can resist known security attacks today and in the future.

Table of Contents

Introduction 3

Scope & Objectives 3

Background: The Problem of Application Security 4

A Solution: An Architecture for Managing Software Risk 4

The Managed Code Paradigm 4

.NET Framework Security in Detail 5

Evidence-Based Security 6
Code Access Security 8
The Verification Process 8
Role-Based Security 9
Cryptography 11
Application Domains 12

Conclusion 12

Poor Design and Administration Can Still Lead to Security Risk 13
Security Is Mission-Critical -- To Everything 13

Resources for Further Reading 15

Introduction

From the early stages of the development of the .NET Framework, Foundstone, Inc. and CORE Security Technologies have assisted Microsoft Corp. with analyzing and assessing the security of its architecture and implementation.

Our analysis of the .NET Framework began in the summer of 2000, before the first beta release of the software and continued up through Beta 2. The entire engagement encompassed over 2,800 hours of rigorous, independent security auditing and testing by a team of ten experts, during which we had full access to the source code and Microsoft engineers and became intimately familiar with the security architecture of the .NET Framework, from design principles to code-level implementation.

The audit followed standard methodologies developed by Foundstone, Inc. and CORE Security Technologies over many years of experience testing, assessing, and securing complex software applications for organizations ranging from members of the Fortune 500 to newly-minted startups. We like to say that we have seen "the good, bad and the ugly" from our perch as security solution providers, and the .NET Framework bore the brunt of our collective knowledge during our year of exposure to its inner workings.

This white paper focuses on the broad security features of the .NET Framework. It is based largely on the results of the assessment we performed over the last year and our continued interaction with the .NET Framework development team. The thoughts and opinions expressed herein are solely our own independent observations based on rigorous analysis and testing of many builds of the software. It is our hope that this document will promote understanding of security in the .NET Framework, and convey our confidence in that architecture and its implementation.

Scope & Objectives

In this document, we will review many of the common security challenges enterprises face during the design and development of software solutions, and outline how the .NET Framework provides a reasonable solution to these issues through its security architecture.

At all times, we will seek to make the complexities of .NET Framework security approachable to readers with at least a moderate technical background. We assume at least a basic familiarity with the .NET Framework, and do not spend inordinate time with background information on the basic technology involved. We provide many references for further reading at the end of this document for those seeking more deeply technical coverage of the .NET Framework.

Background: The Problem of Application Security

Practically no one today questions that many software applications are mission-critical, especially those that are built using Internet-based technologies. They have evolved from simple, static, data-manipulation channels into complex, dynamic, transaction-oriented pillars of corporate commerce.

The ever-increasing complexity and functionality of modern software applications has driven an unfortunate and alarming counter-trend, however: a growing number of organizations have fallen victim to assaults against their software from internal and external interlopers.

A Solution: An Architecture for Managing Software Risk

The managed code architecture of the .NET Framework provides a compelling solution to the problem of software application security. It transparently controls the behavior of code even in the most adverse circumstances, so that the risks inherent in all types of applications – client- and server-side – are greatly reduced. In fact, used appropriately, we believe that it is one of the best platforms for developing enterprise and Web applications with strict security requirements.

At a high-level, the .NET Framework gives developers and administrators granular security control over their applications and resources; provides developers with an easy-to-use toolset to implement powerful authentication, authorization, and cryptographic routines; eliminates many of the major security risks facing applications today due to flawed code (such as buffer overflows); and shifts the burden from having to make critical security decisions -- such as whether or not to run a particular application or what resources that application should be able to access -- from end users to developers and administrators.

The Managed Code Paradigm

Before we discuss in detail how the .NET Framework accomplishes this, it's helpful to first review the basic components of the Framework itself, including:

- Common language runtime
- Class libraries
- Assemblies

The Common Language Runtime

The *common language runtime* (CLR) is the engine that runs and "manages" executing code. Thus, from a security perspective, the CLR enforces the .NET Framework's restrictions on executing code and prevents it from behaving unexpectedly.

More specifically, the CLR performs "just-in-time" compilation (JIT) when running managed code. JIT translates managed code into native code before it executes it. Since the JIT generates the code within the CLR, the CLR is uniquely positioned to ensure its

security, something that can't be done with code executing unprocessed in the native environment.

The Class Libraries

The .NET *Framework class libraries are* a collection of reusable classes, or types, that developers can use to write programs that will execute in the common language runtime. These implement many important security features, including permissions (i.e., the right to access one or more system resources),, authentication mechanisms, and cryptographic protocols and primitives. The large majority of applications could benefit from this security simply by using these libraries, with no security-specific code required. We will discuss these features in more detail later in this document.

Assemblies

An *assembly* is an executable or DLL compiled using one of the .NET Framework's many language compilers. .NET Framework assemblies can be written in nearly every major programming language, including Visual Basic, C#, C++, J#, Perl, and COBOL, to name just a few. Thus, developers may program in the language most appropriate to their task and skill set, and the same security infrastructure will support them, regardless of their selection.

Assemblies contain the code that the runtime executes in the form of Microsoft Intermediate Language (MSIL). We previously discussed how the CLR JITs MSIL to native code, providing a unique vantage point from which to apply security to executing code. Assemblies also contain *metadata*, which the CLR uses to locate and load classes, lay out instances in memory, resolve method invocations, generate native code, enforce security, and set runtime context boundaries.

Through assemblies, the CLR and class libraries implement the managed code architecture of the .NET Framework. The remainder of this document discusses this managed code architecture in greater detail.

.NET Framework Security in Detail

The security architecture of the .NET Framework is composed of a number of core elements, including:

- Evidence-based security
- Code access security
- The verification process
- Role-based security
- Cryptography
- Application Domains

Each element is discussed in detail below.

Evidence-Based Security

The key elements of the .NET Framework evidence-based security subsystem include *policy*, *permissions*, and *evidence*.

Policy

Anyone with any experience in information systems security will tell you that security is impossible to attain in a vacuum -- it must be driven by *policy*. All of the .NET Framework security thus rests ultimately on carefully defined, XML-inscribed policy. In essence, .NET Framework policy defines what resources code in executing assemblies may access, preventing software from errantly or maliciously harming the integrity of data. Policy in the .NET Framework is ubiquitous and well-secured from non-administrative users. It is installed automatically on every machine, for each user account, Optionally, it can be deployed across Windows domains via Group Policy.

The basic function of security policy in the .NET Framework is to match *permissions* to *evidence* (we will discuss both of these momentarily). The default security policy shipped with the .NET Framework was designed by Microsoft, and is intended to create a safe execution environment for a typical end user. It can also be customized by sufficiently privileged administrative accounts to address unique needs.

Permissions

Permissions lie at the root of policy. Permissions describe one or more resources and associated rights, and implement methods for demanding and asserting access. The .NET Framework includes permissions for the following objects: DataAccess; DNS; DirectoryServices; FileIO; EventLog; Environment; FileDialog; Registry; Reflection; Socket; Web; IsolatedStorage; UI; Printing; MessageQueue; and Security – whose members include AllFlags, Assertion, ControlAppDomain, ControlDomainPolicy, ControlEvidence, ControlPolicy, ControlPrincipal, ControlThread, Execution, Infrastructure, NoFlags, RemotingConfiguration, SkipVerification, and UnmanagedCode. The developer may extend these permissions definitions to include application-defined resources and methods for verifying access rights. This contrasts with other managed code architectures like Java 2, where such granular customizations cannot be made as easily.

Developers have some ability to control how their code reacts relative to permissions granted by policy by embedding *permission requests* within assemblies. There are three types of permission requests: Minimal, Optional, and Refuse. If policy does not grant an assembly everything listed in the "Minimal" set, the assembly will fail to load and will not run. Using the "Refuse" request, developers can explicitly decline access to resources that the application might otherwise be able to access but which it does not require in order to run. This means that developers can limit the scope of their application's permission set beyond even what the administrator-defined policies would allow. To the extent code can refuse permissions, it is exonerated for being involved in security problems that might arise involving those same permissions. This is a very

granular capability compared to current managed code architectures like Java 2, and it allows code to be designed to run with least privilege.

Isolated Storage

Of all the permissions covered by evidence-based security, the IsolatedStorage permission is worth particular mention. This provides support for a special file storage mechanism that is built on top of the underlying file system, but ensures that different application's repositories are kept isolated from each other and specific file system characteristics are not revealed (such as path names, available drives, and so on). Using isolated storage, semi-trusted assemblies that are not granted the FileIO permission can still be allowed to locally store application-specific data. Thanks to the strict isolation and limited accessibility of these storage areas, however, they do so in a way that does not risk compromising the local file system or machine itself. This is particularly useful for running semi-trusted code (e.g. Internet applications), while granting the powerful functionality of local storage capabilities.

Evidence

At runtime, the CLR determines which permissions can be assigned to a particular assembly by evaluating that assembly's *evidence*. Evidence can come from a variety of sources resident within an assembly, or it can be gathered from the local execution environment. Sources of evidence include:

- Cryptographically sealed namespaces (strong names)
- Software publisher identity (Authenticode®)
- Code origin (URL, site, Internet Explorer Zone)

Putting It All Together: Policy, Permissions, and Evidence in Action

So far, we have described each of the major components of the .NET Framework's evidence-based security model separately. It's important to note, however, that these components work together fluidly to provide an execution environment much different from what we are used to currently when we launch executables. With the .NET Framework, few security decisions have to be made at runtime by the user. The .NET Framework is already transparently ensuring that the code end users run enforces the design principles for the security of the application, ahead of time, relieving the end user from making important security decisions which he or she is most likely not qualified to make.

From the developer perspective, nearly all of the work involved is handled behind-the-scenes. As long as sufficient permissions[1] and properly configured policies cover all the resources involved, and as long as the developer uses managed code to access them, all of the required evidence-checking and policy enforcement is handled transparently.

[1] Remember, the minimal, optional, and refuse permission requests are strictly supplementary.

Code Access Security

Code access security (CAS) is the enforcement engine that ensures assembly code does not exceed its granted permissions while executing on a computer system. As managed code assemblies are loaded for execution, they are associated with a corresponding set of permissions. If a method in an assembly needs permission to access a resource, the code providing access to that resource will *demand* the appropriate permission object. When this occurs, a *stack walk* is initiated. This checks that each assembly in the call-chain has the demanded permission granted to it, not just the immediate caller. If any of the callers fail this test, a security exception is generated and the requested operation is not performed. Stack walking prevents "luring attacks" in which untrustworthy code attempts to "trick" code in another assembly, with greater access rights, to call a protected object and bypass security restrictions.

When using .NET Framework class libraries on resources for which policies and permissions are already defined, this work is all handled behind the scenes. There are two mechanisms by which developers can actively force permissions checks: imperative and declarative. Imperative checks are simply runtime method calls to the core security engine requesting a demand or to override portions of the stack walk operation. Declarative security checks are essentially the same. However they are expressed as custom attributes that are evaluated at compile time and embedded in metadata. Declarative checks cover the same operations as imperative, plus they allow for a few additional checks that are implemented strictly at JIT-time.

Under certain circumstances, code may need to call a permission's *assert* method in order to limit subsequent stack walks to this code's stack frame. This will allow it to access certain resources even when the method's callers do not have proper permissions. For example, the code providing file access will typically demand its callers have the FileIO permission, but then assert the Unmanaged code permission to access the underlying Windows file system. This technique should be used sparingly and is only available to highly trusted code granted the Assertion permission. Note that the assertion operation is fine-grained and only applies to the permission asserted

Code access security thus sets an extraordinarily high bar for intruders to surmount when attempting to abuse the behavior of running code.

The Verification Process

There is one final step in ensuring the runtime safety of managed code. This is known as the *verification process*. During JIT compilation, the CLR verifies all managed code to ensure memory type safety. This eliminates the risk of code executing or provoking "unexpected" actions that could bypass the common application flow and circumvent security checks.

The verification process prevents common errors from occurring, such as using an integer as a pointer to access arbitrary memory locations, treating an object as a different type to allow the reading of private state or memory outside the object boundary, accessing a

private field or method from outside its class, accessing a freshly created object before it has been initialized to cause incorrect operation or to access residual information in memory. Buffer overflows (supplying parameters that exceed the size expected by the called method), referring to memory containing anything other than defined variables or method entry points, referencing stack locations outside the allocated stack frame (invalid references), and transferring execution to arbitrary locations within a process are also prevented by the verification process. These common programming mistakes underlie a significant majority of today's security vulnerabilities, and no longer pose a threat within the type safe, managed environment provided by the .NET Framework. This in itself is probably one of the most compelling outcomes of designing applications using the .NET Framework.

A Note on Unmanaged Code

Code that runs outside the control of the CLR is referred to as "unmanaged" code. Unmanaged code by definition is not constrained by the security measures of the CLR, and is thus capable of obtaining unauthorized access to resources in the native environment via traditional attacks. .

Fortunately, most applications never will need to call native code directly. The .NET Framework class libraries implement managed code wrappers for many unmanaged code methods (i.e. Win32 API calls). These managed code wrappers take care of verifying the caller permissions and parameters and call the appropriate unmanaged code.

Role-Based Security

Up to this point, our discussion has been focused on how the security of the .NET Framework's code execution model relies heavily on evidence read from within an assembly or the local environment. *Role-Based Security* defines the way the .NET Framework establishes identity, and permits or denies that identity to access resources. These two processes are frequently referred to as *authentication* and *authorization*, the linchpins of secure application design for Web applications.

Authentication

Role-Based Security gives developers the freedom to construct highly customized authentication scenarios for their applications. All of the most common authentication routines are available to .NET Framework-based applications via a diverse range of *authentication providers*. These are code routines that verify credentials, create the proper Identity and Principal object, and attach it to the request's context. Once the user identity is determined, authorization decisions can be made when accessing resources. Authentication providers can also offer other functionality, such as cookie generation for session state maintenance. Authentication providers supported by the .NET Framework include:

- **Forms-based (Cookie) Authentication:** Using this provider causes unauthenticated requests to be redirected to a specified HTML form using client side redirection. The user can then supply logon credentials, and post the form back to the server. If the application authenticates the request (using application-specific logic), ASP.NET

issues a cookie that contains the credentials or a key for reacquiring the client identity. Subsequent request are issued with the cookie in the request headers, which means that subsequent manual authentications are unnecessary. The credentials can be custom checked against different sources, such as a SQL database or a Microsoft Exchange directory. This authentication module is often used when you want to present the user with a logon page.

- **Passport Authentication:** This is a centralized authentication service provided by Microsoft that offers a single logon facility and membership services for participating sites. ASP.NET, in conjunction with the Microsoft Passport Software Development Kit (SDK), provides functionality similar to Forms Authentication for Passport users.
- **IIS**: Microsoft's IIS server provides several built-in authentication mechanisms. These can be used to provide authenticated identities to IIS-hosted applications. If there are corresponding Windows accounts, IIS can also provide automatic account mapping based on the authenticated identity. Supported authentication mechanisms include Basic Authentication, NTLM, Kerberos, Digest Authentication, and X.509 Certificates (with SSL).
- **Windows Authentication:** Windows supports a number of authentication mechanisms that can be used by applications via the SSPI subsystems. These include Kerberos, NTLM, and X509 Certificates.

Developers can additionally write custom authentication and authorization code (for example, by combining IIS Anonymous authentication with ASP.NET's Form Authentication provider), or use one of the standard authentication modules already available in the ASP.NET Framework (by combining IIS NTLM or Kerberos authentication with ASP.NET's Windows authentication provider). Authentication providers can be configured per application and per virtual directory.

Authorization

Once identity is established reliably using one of these well-known methods, access to resources can be authorized through a similarly extensible and flexible architecture. ASP.NET provides two different methods of authorization to application code:

- File Authorization, where the request location is mapped to the physical file, denying or granting access by matching the file's ACLs with the identity making the request[2]
- URL Authorization, where access can be granted or revoked specifically by mapping users and roles to pieces of the URI namespace, including the request method (GET, HEAD, POST, etc.)

For example, to restrict access to the URL "http://servername.com/adminpage.aspx" to users in the role "Admin," one could perform the following runtime role checks in code:

```
if(HTTPContext.IsCallerInRole("Admin"){ … } )
```

[2] File authorization is used only in conjunction with Windows Authentication, since other authentication mechanisms typically do not set a per-user Windows access token

Principal and Identity

The .NET Framework provides a rich and robust object model for identity using its Principal and Identity concepts. A Principal represents the security context under which the code is running while an Identity represents the identity of the user associated with that security context. Normally, an Identity will be created after a user's successful authentication and attached to a Principal that will in turn be associated with an execution context. Code running in a specific context can then query the Principal about the Identity role(s), allowing or denying permissions according to role membership.

This architecture is flexible enough to permit custom definitions of roles, identities, and principals. For example, it is possible to map identities to username/password pairs stored in a database or text file. Implementing the GenericPrincipal object allows for these highly customized, platform-independent authorization scenarios.

Alternatively, .NET Framework can leverage the traditional Windows security subsystem via the WindowsPrincipal object, allowing the easy mapping of roles to existing Windows user accounts and groups.

Of course, the .NET Framework is capable of performing *impersonation* of client requests to access resources. Impersonation remains one of the key differentiators between Windows-based authorization architectures and competitive solutions like UNIX and Linux, and allows solutions architects to keep identity tied to one user account throughout the flow of an application, rather than periodically handing off control to the process under which the application runs.

Impersonation in ASP.NET can be implemented in two different ways:

- Per-request impersonation, which means that an application can run with the privileges of the identity making the request. This helps in reducing the impact of possible security breaches while improving auditing capabilities.
- Application-level impersonation, where the worker process running the application does so using the identity of a user specified in the configuration, diminishing the impact of application compromise by isolating and protecting other applications sharing the same server and system (i.e. application compromise doesn't necessarily leads to system compromise)[3].

Impersonation gives ASP.NET applications granularity and flexibility when accessing resources, homogeneously across the .NET Framework.

Cryptography

Similar to the ready availability of simple authentication and authorization features within the .NET Framework, cryptographic primitives are also easily accessible to developers via stream-based managed code libraries for encryption, digital signatures,

[3] It should be noted that credentials are stored in configuration in cleartext; a more appropriate way to achieve this is to configure the anonymous account or to call into a ServicedComponent running as a fixed identity in a COM+ server application

hashing, and random number generation. Wrappers for most CryptoAPI functionality are also available. Algorithm support includes:

- RSA and DSA public key (asymmetric) encryption
- DES, TripleDES, and RC2 private key (symmetric) encryption
- MD5 and SHA1 hashing

Besides the supported primitives, the .NET Framework supports encryption by means of cryptographic streaming objects based on the implemented primitives and various feedback modes. It also supports digital signatures, message authentication codes (MACs)/keyed hash, pseudo-random number generators (PRNGs), and authentication mechanisms. New or pre-standard primitives as SHA-256 or XMLDSIG are already supported. ASP.NET includes well-integrated support for signing and encrypting cookie content addressing long-standing sensitive issues of Web application security.

The ready availability and more than complete breadth of such libraries will hopefully drive more widespread reliance on the cryptography to fortify the security of everyday applications. Based on our own experiences, we can confidently state that well-implemented cryptography dramatically increases the security of many aspects of a given application.

Application Domains

Finally, the .NET Framework offers a compelling new way to segregate portions of applications through what is known as *application domains*. Usually, operating systems provide this isolation by running each application in a separate process, each one having a different address space, preventing them from directly interfering with each other. Unfortunately for highly loaded servers, processes are expensive in terms of system performance, and it may be prohibitive to run an individual process for each user that is accessing the server.

Thanks to the type-safety of verified managed code (which ensures, among other things, that the code cannot access or jump to arbitrary addresses in memory), the CLR is able to provide a great level of isolation *within* the process boundary. A single process can contain several application domains, with different evidence-based trust levels and associated principals, without danger of any kind of malicious interference between them. Code running in one domain cannot directly affect other applications in the same process, or access other application resources. All managed code is loaded into a single application domain and run according to that domain's security policy.

All in all, application domains are a tremendous boon for Application Service Providers and IT departments hosting networked applications. They offer powerful security control at a fraction of the resource costs of existing solutions.

Conclusion

There is a lot more detail we'd like to cover about the security of the .NET architecture, but we'd need several more whitepapers. We conclude with two parting thoughts.

Poor Design and Administration Can Still Lead to Security Risk

As we have shown throughout this paper, the .NET Framework transparently implements a great deal of security infrastructure via the key components of its security architecture. However, it still does not eliminate the need to thoughtfully design an application with security in mind. As with any application development environment, when implementing code that involves custom permission objects, authorization mechanisms, or any security-relevant functionality, the developer must be familiar with the .NET Framework's security architecture in order to ensure that the design principles are enforced.

In particular, unsafe usage of permission's security assert method must be avoided. We recommend strategically consolidating and unifying permission demands or asserts within an application to improve security and code auditing capabilities.

Another potentially sensitive design concern arises when implementing additional cryptographic functionality within the .NET Framework. Special care must be taken at these junctures, as design or implementation errors here may expose not only a new component's security, but also the security of other components that rely on common cryptographic elements. For example, one could design an application using cookie authentication in a manner that would make it feasible for outside parties to run chosen-plaintext cryptographic attacks against the authentication mechanism.

Besides application design, deployment and administration are critical to security. The networks and systems on which .NET Framework-based applications run are still potentially vulnerable, and must be secured according to best practices (strong account management policies, disable unnecessary services, regularly install patches, and so on). No managed code paradigm can account for sloppy system administration. Although the .NET Framework transparently eliminates many common code-level errors, it is powerless to prevent issues arising from inappropriately assigned account privileges, misconfigured resource access control lists, and similar errors in configuration.

Furthermore, as we have shown, unmanaged code continues to operate outside of the constraints of the .NET Framework security model, and can still be hazardous. Applications architects who rely on unmanaged code cannot enjoy the full security benefits provided by the managed environment. As a general rule, unmanaged code should be avoided, to be used only as a last resort, and subject to a thorough security review. Indiscriminate and improper calls to unmanaged code is one of the biggest potential points of failure in terms of the overall security of a .NET Framework application.

Security Is Mission-Critical -- To Everything

Security is but one part of the overall story of the .NET Framework, but a critically important one. As we have discussed in this paper, security is mission-critical to all networked systems today, and .NET Framework can, if used correctly, provide developers, administrators, and end users with much-needed assurance that their applications are resistant to common attacks, now and in the future. The .NET Framework delivers this assurance through novel approaches to managing software behavior, including evidence- and role-based security features.

We at Foundstone and CORE hope that this brief exploration of the .NET Framework security architecture has been informative and helpful to those of you who will design and build the next generation of software. Based on our own analysis and extended interactions with the .NET Framework architects at Microsoft, we are confident that application security can improve as the migration towards the .NET Framework continues, and also in the resources and motivation of the .NET Framework team to address security with the utmost priority as the computing technology continues to evolve.

Resources for Further Reading

MSDN .NET Developer Center	http://msdn.microsoft.com/net
GotDotNet Community	http://www.gotdotnet.com/
Visual Studio.NET	http://msdn.microsoft.com/vstudio/nextgen/default.asp
.NET Framework Reference	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguidnf/html/cpframeworkref_start.asp
Main ASP.NET Site	http://www.asp.net/
MSDN's ASP.NET Site	http://msdn.microsoft.com/net/aspnet
IBuySpy Developer Solutions Site by Vertigo Software	http://ibuyspy.com/
Foundstone	http://www.foundstone.com
CORE Security Technologies	http://www.corest.com