Journal of Information Engineering and Applications www.iiste.org
ISSN 2224-5782 (print) ISSN 2225-0506 (online)
Vol.4, No.10, 2014

Making the NET. Framework More Secure Depending on Architecture of NET. Framework and its Security Levels

Baydaa Jaffer AL-Khafaji Nadia Mohammed Abdulla Zainab F. Hassan
1.Computer Science, College of Education for Pure Science Ibn-AL-Haitham University of Baghdad
2.Computer & Internet Unit,College of langauges, University of Baghdad

Abstract
The .NET Framework is a managed type-safe environment for application development and execution. The .NET Framework manages all aspects of your program's execution, it allocates memory for the storage of data and instructions, grants or denies the appropriate permissions to your application, initiates and manages application execution, and manages the reallocation of memory from resources that are no longer needed.

1. Introduction

The .NET Framework. Development platform introduces many new concepts, technologies, and terms. The goal of this research is to give an overview of the .NET Framework to show how it is architected, to introduce some of the new technologies, and to define many of the new terms. If you decided to use the .NET Framework as your development platform, the first step is to determine what type of application or component you intend to build. After that you must decide what programming language to use [1]. This is usually a difficult task because different languages offer different capabilities. For example, in unmanaged C/C++, you have pretty low-level control of the system. You can manage memory exactly the way you want to, create threads easily if you need to, and so on. Visual Basic 6, on the other hand, allows you to build UI applications very rapidly and allows the easy control of COM objects and databases.

2. Architecture of 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 [1, 4, and 8]:-
• 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 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.

If you use the .NET Framework, your code targets the common language runtime (CLR), which affects your decision about a programming language. The common language runtime is just what its name says it is: A runtime that is usable by different and varied programming languages. The features of the CLR are available to any and all programming languages that target it period. If the runtime uses exceptions to report errors, then all languages get errors reported via exceptions. If the runtime allows you to create a thread, then any language can create a thread. In fact, at runtime, the CLR has no idea which programming language the developer used for the source code. This means that you should choose whatever programming language allows you to express your intentions most easily.

Regardless of which compiler you use, the result is a *managed module*. A managed module is a standard Windows Portable Executable (PE) file that requires the CLR to execute. In the future, other operating systems may use the PE file format as well. A Managed Module is composed of the following parts [2, 3]:

- ***PE Header*** :- This is the standard Windows PE file header, which is similar to the Common Object File Format (COFF) header. The PE header indicates the type of file -GUI, CUI, or DLL- and also has a timestamp indicating when the file was built.
- ***CLR header*** :- This header contains the information (interpreted by the CLR and utilities) that makes this a managed module. It includes the version of the CLR required, some flags, and the location/size of the module's metadata, resources, strong name, some flags, and other less interesting stuff.
- ***Metadata*** :- Every managed module contains metadata tables, of which there are two main types: those that describe the types and members *defined in* your source code, and those that describe the types and members *referenced by* your source code.
- ***Intermediate Language (IL) Code*** **:-** This is the code that was produced by the compiler as it compiled the source code. IL is later compiled by the CLR into native CPU instructions.

In brief, *metadata* is simply a set of data tables that describe what is defined in the module, such as types and their members. In addition, metadata also has tables indicating what the managed module references, such as imported types and their members. Metadata is a superset of older technologies such as type libraries and IDL files. Metadata has many uses. Here are some of them [3]:-

1. Metadata removes the need for header and library files when compiling, because all the information about the referenced types/members is contained in one file along with the IL that implements those type/members.
2. The CLR code verification process uses metadata to ensure that your code performs only “safe” operations.
3. Metadata allows the garbage collector to track the lifetime of objects. For any object, the garbage collector can determine the type of the object, and from the metadata it knows which fields within that object refer to other objects.

3. Loading the Common Language Runtime

Each assembly that you build can either be an executable application or a DLL containing a set of types (components) for use by an executable application. The CLR is responsible for managing the execution of code contained within these assemblies. This means that the .NET Framework must be installed on the host machine. Microsoft has created a redistribution package that you can freely ship to install the .NET Framework on your customer's machines. Future versions of Windows will include the .NET Framework, at which point you will no longer need to ship it with your assemblies.

You can tell whether the .NET Framework has been installed by looking for the MSCorEE.dll file in the %windir%\system32 directory. The existence of this file tells you that the .NET Framework is installed [4]. However, several versions of the .NET Framework may be installed on a single machine simultaneously. If you want to determine exactly which versions of the .NET Framework are installed, examine the sub keys under the following registry key:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NET Framework\policy

When you build an EXE assembly, the compiler/linker emits some special information into the resulting assembly's PE File header and the file's .text section. When the EXE file is invoked, this special information causes the CLR to load and initialize. Then the CLR locates the entry point method for the application and lets the application start executing.

4. Executing the Code in Your Assembly

Managed modules contain both metadata and intermediate language (IL) code. IL is a CPU-independent machine language created by Microsoft after consultation with several external commercial and academic language/compiler writers. IL is much higher-level than most CPU machine languages. IL understands object types and has instructions that create and initialize objects, call virtual methods on objects, and manipulate array elements directly. It even has instructions that throw and catch exceptions for error handling. You can think of IL as an object-oriented machine language. Usually, developers prefer to program in a high-level language, such as C# or Visual Basic.NET [5].

Any high-level language will most likely expose only a subset of the facilities offered by the CLR. IL assembly language, however, gives a developer access to all the facilities of the CLR. So if your programming language of choice hides a CLR feature that you really wish to take advantage of, you can write that portion of your code in IL assembly or in another programming language that exposes the CLR feature you seek. The only way for you to know what facilities are offered by the runtime is to read documentation specific to the CLR itself. An important thing to note about IL is that it is not tied to any specific CPU platform. This means that a managed module containing IL can run on any CPU platform as long as the operating system running on that CPU platform hosts a version of the CLR.

5. The .NET Framework Class Library

Included with the .NET Framework is a set of Framework Class Library (FCL) assemblies that contains several thousand type definitions, where each type exposes some functionality. All in all, the CLR and the FCL allow developers to build the following kinds of applications [4, and 6]

- Web Services. Components that can be accessed over the Internet very easily. Web services are, of course, the main thrust of Microsoft's .NET initiative.
- Web Forms. HTML-based applications (web sites). Typically, web form applications make database queries and web service calls, combine and filter the returned information, and then present that information in a browser using a rich HTML-based UI. Web forms provide a Visual Basic 6- and InterDev-like development environment for web applications written in any CLR language.
- Windows Forms. Rich Windows GUI applications. Instead of using a web form to create your

application's UI, you can use the more powerful, higher-performance functionality offered by the Windows desktop. Windows form applications can take advantage of controls, menus, mouse and keyboard events, and can talk directly to the underlying operating system.

- Windows Console Applications. For applications with very simple UI demands, a console application provides a quick and easy solution. Compilers, utilities, and tools are typically implemented as console applications.

6. The Common Type System

Types expose functionality to your applications and components. Types are the mechanism by which code written in one programming language can talk to code written in a different programming language. Because types are at the root of the CLR, Microsoft created a formal specification –the common type system (CTS)-that describes how types are defined and behave.

The CTS specification states that a type may contain zero or more members. A brief introduction to Type Member can be explained here [6]:

1. ***Field***. A data variable that is part of the object's state. Fields are identified by their name and type.
2. ***Method***. A function that performs an operation on the object, often changing the object's state. Methods have a name, signature, and modifiers. The signature specifies the calling convention, number of parameters (and their sequence), the types of the parameters, and the type of value returned by the method.
3. ***Property***. To the caller, this member looks like a field. But to the type implementer, this member looks like a method (or two). Properties allow an implementer to validate input parameters and object state before accessing the value and to calculate a value only when necessary; they also allow a user of the type to have simplified syntax.
4. ***Event.*** A notification mechanism between an object and other interested objects.

7. The Common Language Specification

If you intend to create types that are easily accessible from other programming languages, then it is important that you use only features of your programming language that are guaranteed to be available in all other languages. To help you with this, Microsoft has defined a *common language specification* (CLS) that details for compiler vendors the minimum set of features that their compilers must support if they are to target the runtime. Note that the CLR/CTS supports a lot more features than the subset defined by the common language specification, so if you don't care about language interoperability, you can develop very rich types limited only by the capabilities of the language.

8. Interoperability with Unmanaged Code

The CLR supports three interoperability [7]:-

1. Managed code can call an unmanaged function in a DLL. Managed code can easily call functions contained in DLLs using a mechanism called P/Invoke (for Platform Invoke). After all, many of the types defined in the FCL internally call functions exported from Kernel32.dll, User32.dll, and so on. Many programming languages will expose a mechanism that makes it easy for managed code to call out to unmanaged functions contained in DLLs.

Example: A C# or VB application can call the Create Semaphore function exported from Kernel32.dll.

2. Managed code can use an existing COM component (server). Many companies have already implemented a number of unmanaged COM components. Using the type library from these components, a managed assembly can be created that describes the COM component. Managed code can access the type in the managed assembly just like any other managed type.

Example: Using DirectX COM components from a C# or VB application.

3. Unmanaged code can use a managed type (server). A lot of existing unmanaged code requires that you supply a COM component for the code to work correctly. You can implement these components easily using managed code, avoiding all the code that has to do with reference counting and interfaces.

Example: Creating an ActiveX control or a shell extension in C# or VB.

9. The Security of .NET Framework

Many computer security issues can arise from unintentional behavior of computer programs. This unintentional behavior can result from several types of issues; mainly buffer over-flows, input validation errors and faulty default configurations [1 and 6].

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.

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.

Default configurations of software often make broad assumptions about the target computer sys tem. This can lead to poor securing of confidential information, as well as abnormal execution of the software.

10. Code Access Security CAS

CAS is one of the fundamental principles behind the .NET Framework. Traditional software execution security has centered on the identity of the user who is running the software, rather than the identity of the code that is being run. The CAS system in .NET serves three purposes [9]:

- "***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.

11. 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 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 un-trusted 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.

12. 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 [3, 8]:

- 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.

13. Policy

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 [7]. 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. 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 next section on policy levels.

14. 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 organization in which the computer exists [3, 9]. Respectively, the policy levels are referred to as the App Domain, User, and Machine and Enterprise levels. While the three former levels are specified and managed by users and administrators, the Application Domain level is computed automatically per Application Domain at runtime. Although the policy levels are organized in a hierarchy, the policy levels are intersected to produce a policy for the specific context in which an assembly is being run. 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 "Level Final" 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 administrator or individual user can not override the denying or granting of certain permission sets.

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 [9]. 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 Full Trust 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 are considered during policy resolution.

15. Policy Management

The Policy Manager is responsible for assigning permissions to assemblies based on the security policies in place. Based on the permission set granted, the Policy Manager may not let 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, the security policies (Enterprise, Machine, User, AppDomain), and the declarative permission requests found in the assembly metadata [4, 7].

Evidence comes in two forms: trusted evidence and un-trusted 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. Un-trusted 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 un-trusted 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 [9]:

- 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 Policy Exception. 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 Security Permission with the Execution flag set. Again, Policy Exception 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 Policy Exception 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.

Conclusions

• The primary unit of a .NET application is the assembly, which includes an assembly manifest. The assembly manifest describes the assembly and one or more modules, and the modules contain the source code for the application.

• A .NET executable is stored as an IL file. When loaded, the assembly is checked against the security policy of the local system. If it is allowed to run, the first assembly is loaded into memory and compiled into native binary code, where it is stored for the remainder of the program's execution.

• 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.

• Security checks can be applied imperatively or declaratively. Declarative security is applied by associating attribute declarations that specify a security action with classes or methods. Imperative security is applied by calling the appropriate methods of a Permission object that represents the Principal (for role-based security) or system resource (for code access security).

• We can use declarative security to request permissions for the entire assembly. We can use the Assembly (assembly) directive to make a declarative security request for the entire assembly.

References

1. Don Box and Chris Sells. *Essential .NET, Volume I: The Common Language Runtime*. Addison Wesley Professional, November 2002.
2. Don Box. The Security Infrastructure of the CLR Provides Evidence, Policy, Permissions, and Enforcement Services. *MSDN Magazine*, September 2002. URL http://msdn.microsoft.
3. Microsoft Corporation. Information About the .NET W32.Donut Virus. 2002. URL http://support.microsoft.com/default.aspx?scid=kb;en-us;316287.
4. Abhijit Dharia and Rahul Phadnavis. Internals of .NET/Rotor CLR, 2004. URL http://wiki.cs.uiuc.edu/cs427/.Net+CLR+Internals.
5. Dieter Gollman. *Computer Security*. John Wiley and Sons Ltd., July 2003.
6. ECMA International. C# Language Specification, December 2002. URL http://www.ecma-international.org/publications/standards/Ecma-334.htm.
7. Brian A. LaMachia, Sebastian Lange, Matthew Lyons, Rudi Martin, and Kevin T. Price.*.NET Framework Security*. Addison Wesley, 2002.
8. John Leyden. Donut virus highlights holes in .Net. January 2002. URL http://www.theregister.co.uk/2002/01/10/donut_virus_highlights_holes/.
9. Kevin D. Mitnick and William L. Simon. *The Art of Deception: Controlling the Human Element of Security*. John Wiley & Sons Inc, October 2002.