J2EE and .Net security

1. Introduction

1.1. Document Revision History

Author	Document Version	Last Modified Date	Note
Ger Mulcahy	1.2	12/02/2002	Third draft
Ger Mulcahy	1.1	01/02/2002	Added information on JAAS, JSSE, Project Liberty, DotGNU, etc.
Ger Mulcahy	1.0	21/01/2002	Second Draft – added contributors
Ger Mulcahy	0.1	04/01/2002	First draft

1.1.1. Contributors

My thanks to the following for their assistance with this article:
Alan Danziger, Mark Curphey, Alan Faber, Elias Levy, Tony Northrup

1.2. Overview

A number of general comparative articles have been written discussing the pros and cons of these two competing technological platforms. The intention of this paper is to discuss J2EE and .Net at a high level from a security perspective, examining the tools and methodologies the platforms use to provide secure development and deployment environments.

This introduction section covers a brief, incomplete discussion of key features of both platforms. It will not discuss areas that are not analogous between platforms. For more information on both, see the references section of this document.

Note that .Net is a product platform, whereas J2EE is a standard specification, which is implemented to varying degrees of fidelity by a number of vendors. For this reason, direct comparisons may be difficult in certain areas without going into vendor specifics.

For the purposes of this article no real distinction is made between .Net and the .Net Framework, which forms one part of the .Net strategy.

While Microsoft is pushing .Net as their strategy for Web Services, this document will not discuss the two platforms from the point of view of Web Services, nor does it describe COM+, as this is not part of the .Net Framework.

1.3. What is .NET made of?

The .Net initiative, a significant undertaking by Microsoft, is an attempt to tie together applications, development languages, operating systems and data stores into a unified, distributed enterprise platform.

The .Net Framework is a reworking of Windows DNA, Microsoft's previous enterprise development platform, consisting of the CLR(Common Language Runtime), base classes and presentation through ASP.Net. .Net is tightly integrated into the Windows OS environment (with .Net support integrated into all of Microsoft's enterprise applications), but the portability presented by the CLR/CLS (Common Language Specification)/CTS (Common Type Specification) combination means that if Microsoft wanted, .Net could be implemented on other platforms.

1.3.1. C#

C# is an object-oriented language, derived from C++, and syntactically similar (with some exceptions) to the Java language. C# is a "brand new" language, and may suffer from teething troubles as a result. While it is not a part of the .Net framework, it is a part of the .Net strategy, and mirrors Java's function within J2EE.

1.3.2. Common Language Runtime (CLR)

All .Net components are compiled into an interim language called Intermediate Language (IL), which is then executed in the runtime environment provided by the CLR. A growing number of compilers for languages such as C#, Visual Basic.Net, C++, etc. are available for use with .Net, giving programmers the ability to work in .Net using familiar languages. The CLR is part of the core of the .Net Framework, providing a secure execution environment through the principles of what Microsoft describes as managed code and code access security. The CLR also provides housekeeping functionality through garbage collection.

1.3.3. ASP. Net

ASP.Net is an onward progression from ASP (Active Server Pages). Differences from ASP include programmatical changes and the fact that ASP.Net pages are now compiled and executed in a CLR.

1.3.4. ADO.Net

ADO.Net is the successor to Microsoft's ActiveX Data Objects (ADO). Whereas ADO provided two-dimensional access to data, through the use of rows and columns, ADO.Net describes data as objects and utilises XML for transmitting data between application tiers.

1.3.5. Assemblies

An assembly is a collection of code used as the building block of a .Net framework deployment. Assemblies are designed to include security information (in terms of permissions requested and strong name), versioning information (which enables multiple versions of software to co-exist on a single system, theoretically eliminating DLL conflicts), code and supporting resources. Assemblies can be built using the Visual Studio.Net IDE and the Assembly Generation Tool.

1.4. What is J2EE made of?

J2EE is Sun's reference standard for enterprise development, first introduced in December 1999, and now at version 1.3. Core elements of J2EE are described in this section.

1.4.1. Java language

Java is an object-oriented language derived from C++, with features that simplify coding such as memory management through garbage collection, no pointers, etc. Java is designed to be "Write once, run anywhere", this goal being achieved through the use of portable bytecode.

1.4.2. JVM/JRE

The JRE (Java Runtime Environment) consists of the Java Virtual Machine, a just-in-time compiler and some foundation classes. Java classes are compiled into platform-independent bytecode that is executed in the JVM.

1.4.3. JSPs and Servlets

Java Server Pages (JSPs) are analogous to ASP technology, providing the capability to build dynamic web pages composed of HTML with embedded dynamic components, e.g. references to Beans.
Servlets are described as applets that run on the web server, and are used where CGI would traditionally have been employed to build dynamic web applications.

1.4.4. EJB

Enterprise Java Beans (EJB) are used to build distributed applications by providing the communications and execution framework for distributed components. Critical services provided by an EJB container include transaction management, security, resource management and persistence.

1.4.5. JDBC

Java Database Connectivity (JDBC) is an API for connecting to relational databases, manipulating their contents, and processing the output of issued SQL statements. Numerous database vendors have developed drivers based on the JDBC API.

2. Security Models

Enterprise development requires that security targets be met for the development and deployment of code. Development platforms and runtime environments must provide for authentication and authorisation, data integrity, audit trails, non-repudiation and privacy of data.

The responsibilities for these tasks are spread across multiple platform elements in a typical n-tier application architecture. This section will attempt to discuss the approaches taken by .Net and J2EE, focussing mainly on the runtime environments.

2.1. .NET Framework security architecture

The CLR and base classes are responsible in large part for security functions within the .Net framework. The CLR uses a number of criteria to determine the security permissions of code to be executed, and obtains some security information from XML configuration files.

The base classes control access to resources such as the file system by determining the permissions of the caller.

.Net's two primary security functions are described in the following sections.

2.1.1. Code access security

Code access security is a core part of the CLR's security function. Code access security determines the level of trust assigned to a piece of code based on its origins, publisher and other factors. Some key concepts used to define code access security are described below.

2.1.1.1. Evidence-based security

.Net uses the concept of "Evidence-based security", to describe the process by which assemblies are examined by the CLR at runtime. The CLR queries assemblies using the following primary criteria:

- Where did this code originate?
- Who created this assembly?

The first question can be broken out into "Which URL did the assembly originate from?" and "Which zone did the assembly originate from?". Microsoft uses the concept of zones to describe security environments like the Internet, local networks or intranets, the local machine, etc.
Evidence can also be in the form of a digital certificate signed by the publisher of the assembly or the strong name (a unique identifier, consisting of a text name, digital signature, and a public key) of the assembly.

The second question is reasonably straightforward – "What information is available on the creator of this assembly?".

The evidence gathered is included as part of an assembly's metadata. Metadata can include information on types, relationships with other assemblies, security permissions requested as well as a description of the assembly. Metadata is examined by the CLR at various points, including by the verifier, which ensures that types are correct prior to compilation of IL to native code. The verifier is also responsible for ensuring that the metadata associated with an assembly is valid.

The CLR examines the metadata to establish an identity for the assembly and determines the permissions allocated to the assembly based on the security policy.

2.1.1.2. Permissions

Permissions within the CLR are the building blocks of access control. A code access permission in this context is the right of a piece of code to access a particular resource or perform a particular operation. When assemblies are built, the permissions that it requires to run can be included as part of its description.

At runtime the assembly requests these permissions, and those permissions that it requests are either granted or denied by the CLR. An assembly will never be given greater permissions than the current security policy allows, but may be given lesser permissions than it requests.

Examples of permissions are SecurityPermission (the permission to view or modify the security policy), UIPermission (the right to create sub-windows and make use of the clipboard) and the RegistryPermission class (which gives access to Windows Registry details).

Permissions can be grouped into permission sets for ease of administration. These permission sets are then associated with code groups, which are described in the next section. Examples of permission sets are Nothing (no permissions at all, so code cannot execute), Internet (the permissions assigned to untrusted code) and Everything (the set of permissions that grants all standard permissions, with the exception of avoiding code verification).

At runtime, the CLR performs what is known as a stack walk to determine whether the calling assembly has permission to access a particular resource, checking requested permissions against granted permissions for each caller on the stack.

If an assembly's request for access to a resource is denied, a SecurityException is thrown.

2.1.1.3. Security Policy

The security policy is administered using the Code Access Security Policy Tool (Caspol.exe) or the .Net Framework configuration tool. Administrators set the policy for assemblies and application domains, the CLR uses the evidence described above to identify an assembly, and then uses the security policy to determine the permissions the assembly has at runtime.

The policy identifies code by organising code into groups that categorise based on the evidentiary information described, such as the zone from which the code is loaded. If no other information is available, the default policy for the zone from which the code was obtained will be used to determine the permissions that an assembly has.

2.1.2. Role-based security

User membership in a role within a .Net application helps to determine the access that the user has to perform particular operations and access resources. For example, in the case of a financial application, a Broker role might have permission to initiate, authorise and cancel trades on behalf of an individual or financial institution.

Role-based security describes the means by which the .Net framework identifies, authenticates and authorises an individual user from any of a number of authoritative sources. When a user is identified, their authenticated identity (and role membership) is denoted by the term *principal*. A principal can be a member of one or more roles; for example, an individual could be a member of a Broker and an Executive role.

Role-based security employs a permission structure similar to that of code access security, with permissions managed through the PrincipalPermission object.

Authentication and authorisation sources for users can be through Windows machine security, domain security or some custom authentication source.

2.1.3. Programmatic security

.Net uses the IsInRole method to determine whether a user belongs to a particular role. For example, to determine if a user has membership of a Broker role, the following could be used :

```
If User.IsInRole("Broker")
' Permit requested function
Else
' Bounce back to login
End If
```

The classes responsible for the determination of principal identity are stored in the System.Security.Principal namespace.

2.2. J2EE security architecture

The J2EE security architecture is defined as part of the platform specification document. It details security management roles, and specifies goals of the security architecture, but does not specify security policy or implementation details (such as the use of a particular security technology to meet the described goals).

2.2.1. Code management through the JVM and the class file verifier, the class loader and the Security Manager

Basic Java security employs the concept of a "sandbox" to limit the abilities of untrusted code to cause damage to the system on which it runs. Historically, an untrusted piece of code such as an applet would be disallowed from accessing local disk, opening network connections, etc. With the introduction of certificate support through the Java Plug-In, the origin and author of a signed applet could be established definitively, enabling fine-grained permissions to be assigned to individual applets based on the security policy. This means that applets are no longer confined to the default sandbox.

As described in the coming paragraphs, the sandbox is implemented through the JVM and its class verifier, but also through the class loader and the Security Manager/ACL manager.

2.2.1.1. JVM security

The JVM provides a secure runtime environment by managing memory, providing isolation between executing components in different namespaces, array bounds checking, etc. The dynamic way in which the JVM allocates the various memory areas (method area, GC heap, thread stacks) means that it is almost impossible for a would-be attacker to determine what memory areas to attempt to insert malicious instructions into. Bounds checking on arrays prevent unreferenced memory accesses.

The JVM's class file verifier examines classes for basic class file structure upon loading. While bytecodes produced by Sun's compiler should be relatively free of errors (type errors, for example), it is possible for an attacker to manually create malicious bytecode.

The class file verifier examines each loaded class file in four passes; from the most basic check, where the physical attributes of the file are checked (size, magic number, length of attributes) to checking the constant pool to ensure that method and field references have correct attributes, to parsing the instructions for each method. Note that, by default, the only trusted classes are the base classes. All other classes, including those loaded from the application classpath are considered untrusted and must be verified.

2.2.1.2. The Class Loader architecture

There are two types of class loader – the primordial class loader, which is a part of the JVM, and class loader objects, which are used to load non-essential classes.
There can only be one primordial class loader, which is used to bootstrap the JVM by loading the base classes, sometimes using native OS-dependent means.
There can be many instances of class loader objects in the JVM, which can be instantiated on the fly, and used to load objects from sources such as the network, local disk, or data stores. Controlling the creation of class loader objects is therefore important due to the function of the class loader.

Class loaders are responsible for locating classes requested by the JVM for loading into the runtime environment. Part of their responsibility is to prevent unauthorised or untrusted code from replacing trusted code that makes up the base classes. As an example, a class loader should prevent the replacement of the Security Manager class. The attempted replacement of a base class by a maliciously coded class is referred to as class spoofing.

All class loaders, with the exception of the primordial class loader, are loaded by other class loaders, which become the loaded class loader's parent. Thus, the loading of class loaders describes a tree structure with the primordial class loader at the root and classes as the leaf nodes (obviously class loaders are themselves classes).

If a class loader loads a class, all subsequent requests for related classes are directed through that class loader. For example, if a class loader "A" loads a class "Building", and "Building" makes calls to methods in a class called "Cubicle", the JVM will use class loader "A" to load "Cubicle" and any other classes that are referred to by "Building".

Class loaders prevent class spoofing by passing requests back up the tree through their parent class loader until the class loader that loaded the requested class is reached. In the case of the Security Manager class, the primordial class loader is responsible, and consequently the malicious class described above will not be loaded. Classes are loaded only once, and the base classes are only loaded from the local disk using the system classpath.

Class loaders also provide security by managing namespaces. A class that is loaded by a particular class loader can only reference other classes in the same namespace, i.e., loaded by the same class loader or its parents.

2.2.1.3. The Security Manager and Access Controller

The Security Manager was responsible for examining and implementing the security policy, which is specified by policy files. The security policy, as with .Net, determines the permissions that code has at runtime.

In more recent versions of Java, the decisions on whether to grant permissions based on security policy are delegated to the Access Controller. When a class makes a request for permissions, it is received by the Security Manager which passes the request to the Access Controller.

In a similar fashion to the way that .Net groups permissions into permission sets, which it then associates with code groups, permissions in the Java world are grouped into

protection domains, associated with code sources. In other words, groups of permissions are associated with groups of classes, the classes being grouped by their origin.

Signed Java code is assigned permissions based on the system policy as applied to the code's protection domain. Depending on the permissions associated with the source of the code, the applet may have full access to system resources, or may be restricted to a small subset.
Java applications, by default, have no associated security manager, and therefore have full access to system resources.

2.2.2. Platform Roles

The J2EE platform specification describes Organisational or Platform Roles that can be used to divide up responsibilities in a J2EE development and deployment cycle. The Roles described in the platform specification are Product Provider, Application Component Provider, Application Assembler, Deployer and Systems Administrator. These Roles are not absolutes – the responsibilities of the various roles could be divided differently to fit a company's development and deployment methodologies.

Of these roles, most have clear security implications. The Product Provider and Application Component Provider roles are responsible for developing secure code, the Assembler is responsible for defining method permissions and security roles, the Deployer verifies the security view of the deployed application and assigns principals to roles, and the Systems Administrator administers principals and ensures that the local security environment is correct for the J2EE platform.

2.2.3. Security Roles and the Deployment Descriptor

The deployment descriptor is an XML file that ships with each EJB[1], and describes declaratively many of the aspects of the EJB's function and makeup, and its relationship with other beans.

One of the elements in the descriptor is the <security-role-ref> element. This element type is used by the bean developer to define all of the security roles used in the EJB code. Security role names are associated with links, which are then called elsewhere in the descriptor.

The <security-role> element is used to call roles described in the <security-role-ref> elements. For example:

```
.
.
.
<security-role-ref>
  <role-name>root</role-name>
  <role-link>super-user</role-link>
</security-role-ref>
.
.
<security-role>
<description>This is the security-role for the role "root", defined above</description>
<role-name>super-user</role-name>
</security-role>
```

[1] Deployment descriptors can also be used with other Java components (e.g. Apache SOAP deployment decriptors) but every EJB is required to have a deployment descriptor.

The description field in the <security-role> descriptor element is optional.

Membership of a role confers a set of permissions for the duration of the role membership. Principals can be in several roles at the same time, e.g. employee and manager.

Method permissions are also described in the deployment descriptor.

2.2.4. Programmatic security

Role membership can be determined programmatically in the J2EE environment using the *isUserInRole* and *getUserPrincipal* methods of the HTTPServletRequest object for the web container.

As part of the bean-container contract, the container provides the EJBContext object. The corresponding EJBContext methods are *isCallerInRole* and *getCallerPrincipal. getCallerPrincipal* returns the principal associated with the security context, while, predictably enough, *isCallerInRole* is a boolean method used to determine whether the caller belongs to a specified security role.

2.3. Cryptography support

2.3.1. .NET

The .Net base classes provide support for encryption, key generation, hashing and message authentication. Supported algorithms include DES, SHA, AES(Rijndael), RC2, etc.
The .Net Framework provides a number of tools for certificate management and manipulation.
makecert.exe can be used to create X.509 certificates for testing purposes, certmgr.exe (Certificate Manager) is used to manage X.509 certificates, trust lists and revocation lists, and secutil.exe can be used to extract public key information for a certificate from an assembly.

2.3.2. J2EE

JCE (Java Cryptography Extension) is a collection of packages that provides support for encryption, key exchange and MAC algorithms. JCE is an optional package for J2SDK 1.3, but has been integrated into v1.4. While the J2SDK includes some cryptography functions, the JCE separates out much of the functionality due to US Government export restrictions. The JCE uses the concept of CSPs (Cryptographic Service Providers) to plug in different encryption algorithm implementations.

Another set of add-on encryption packages for J2SDK 1.3 that are being integrated into v1.4 is JSSE, the Java Secure Sockets Extension, providing SSL and TLS functionality to Java developers.

3. Conclusion

J2EE and .Net both provide quite comprehensive security services, though with a different focus. Authentication and authorisation services in .Net are provided through Microsoft OSes and identification repositories. J2EE, on the other hand, does not specify which methods or identification stores should be used to perform these functions, leaving these decisions up to vendors and developers. Although its use is not mandated, authentication and authorisation functionality is provided by Sun through JAAS (Java Authentication and Authorisation Service), which is based on PAM.

Both platforms use similar concepts for handling user and code access to resources, with permissions being key to both, and the concept of roles being used to associate permissions with prinicipals in both environments.

While J2EE uses the concept of Organisational Roles to delineate responsibilities at various stages of the development and deployment process, .Net does not define the hierarchy as clearly.

While .Net provides a solid security model through managed code in the CLR, the ability to run unmanaged code confers the ability to bypass CLR security through direct calls to the underlying OS APIs. In the Java world, signed, trusted code has unrestricted access to system resources. Java calling out to native (C/C++) code through JNI confers the ability to bypass the JRE's security as surely as running unmanaged code in the CLR can be used to bypass.Net's security.

The reliance on Microsoft-specific identification stores such as Passport or Windows domains for authentication and authorisation purposes means that .Net applications may require end-users to subscribe to Microsoft's Passport service. As an example of why this might be a "bad thing", Microsoft had to temporarily shut down part of its Passport service during November 2001 due to a security issue. Current copyright restrictions prevent non-Microsoft vendors from providing alternatives, forcing .NET application developers to rely upon a single vendor for back-end services. Concern at this stance in the wider developer and Internet communities has resulted in projects like DotGNU and Project Liberty. DotGNU is an FSF (Free Software Foundation) initiative to provide an open-source implementation of the .Net framework, and a non-vendor controlled Passport alternative (Virtual Identities), which is intended to use encryption wherever possible to protect user details. Project Liberty is an alliance whose members include Sun, AOL, Vodafone, American Express, HP and RSA, to name but a few, with the stated aim of creating a federated single-sign-on scheme allowing authentication and authorisation from any type of network connected device.

Java has also had its share of security problems in the past, especially in the area of certain JREs and malicious applets; problems have also been reported in the class loading process, a fundamental part of JVM security, as described above.

One of the crucial challenges for both Microsoft and J2EE vendors when developing their respective platforms is securely handling code obtained from multiple sources, outside of the local machine. The code verification functions of the JVM are quite mature at this stage and mistakes made in the past have been learned from. The CLR model is similar, but the implementation is relatively untested. It will be interesting to see how the .Net environment stands the test of time from a security perspective, once .Net deployments become more widespread.

4. References

4.1. J2EE

Enterprise Security with EJB and CORBA. Hartman, Flinn, Beznosov
JDance – http://www.jdance.com
JGuru – http://www.jguru.com
JMiddleware – http://www.jmiddleware.com
Java Security Architecture – http://java.sun.com/j2se/1.3/docs/guide/security/security-specTOC.fm.html
JavaWorld – http://www.javaworld.com
Sun's Java site – http://java.sun.com
The J2EE 1.3 Specification (7/27/01)– http://java.sun.com/j2ee/j2ee-1_3-fr-spec.pdf
The ServerSide – http://www.theserverside.com

4.2. .Net

4GuysFromRolla – http://www.4guysfromrolla.com
CLR Security – http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnmag01/html/CAS.asp
C-sharp corner – http://www.csharpcorner.com
DevX – http://www.devx.com
Gotdotnet – http://www.gotdotnet.com
MSDN library – http://msdn.microsoft.com/library
.Net Development Centre – http://www.oreillynet.com/dotnet
.Net Framework Security – http://msdn.microsoft.com/vstudio/nextgen/technology/framework.sec.asp

4.3. Other

DotGNU – http://www.gnu.org/projects/dotgnu
Mono – http://www.go-mono.org
Project Liberty – www.projectliberty.org