Component Security Design Considerations for J2EE and .Net – An Architectural View Part 2

Gunnar Peterson

Introduction

This series of articles aims to provide information regarding key security design elements for developers, architects, security and development team stakeholders who work on component projects. In the previous issue, we discussed design considerations for Sun Microsystems' middleware platform Java 2 Enterprise Edition (J2EE). In this article we will describe the key issues facing any team implementing components for Microsoft's .Net middleware platform.

As discussed in the previous issue, component security is something that many organizations are just beginning to focus on. Historically, an enterprise's may be comprised of firewalls, intrusion detection, information security policy, configuration guidelines, and perhaps PKI. The software industry's move towards component programming enables organizations to engage the security team (composed of individuals who are not necessarily developers) to extend their security model and policies to the application layer.

Well-designed component systems abstract the low level code up to higher level logical objects, methods and attributes. Representatives from the business, security, administration or development team are then able to influence design decisions related to their area of expertise without having to have the ability to read source code. The result of this collaboration is a significantly more mature finished product.

As stated in the previous issue, the power implied by components on development team structure has yet to be realized. Two factors that are responsible for this are:

1. Organizations have been slow to fully comprehend the implications of components. The optimal organization of a component development team is fundamentally different from a traditional procedural or object oriented development team. Component design reflects a strict decoupling of roles and responsibilities, especially the separation of logic from data and user interface and the organizational structure of the development team should mimic this distinction.
2. With the lack of clear definition of roles in their organizations, the individuals best suited to adding value to these more focused and specialized development teams have for the most part not stepped up to define or fulfill these roles. Couple this with the fact that there are not a lot of good examples in the way of books, conferences, or documentation for individuals to follow.

The full power of component development will remain under-realized until organizations define a more holistic approach to development and members of the development step into this breach. Today, enterprise security team members have a good opportunity to provide more value and more layers of protection to their organizations that use component-based designs like J2EE and .Net. First, the enterprise security team members must learn some of the key security elements in the framework.

.Net in brief

As stated above, .Net is Microsoft's latest foray into component development. Unlike J2EE, .Net represents an evolution from a long history (in software time) of component frameworks. .Net's history can be traced from its predecessors COM/DCOM/COM+. COM was Microsoft's first effort at enterprise level interoperability. COM grew out of Microsoft's work on ActiveX, which primarily was designed to provide functionality such as embedding Excel spreadsheets in Word documents. COM added support for enterprise concerns such as transaction and security support.

"COM was like an ugly woman with a heart of gold. You really had to want to fall in love, but once you did, it was forever. Java was like a beautiful woman with horrible breath. Sure she looks great from across a crowded room, but you really need to overlook certain annoyances if you plan on spending any time with her. The CLR is like a great looking woman with nice breath but since you are her first fling, you can't ask her ex for pointers on how to get along with her. Colorful analogies aside, the CLR seems to please everyone who gets near it. Perfect? No. Imminently better than anything else out there? Absolutely. [Please note this was originally written in gender-neutral terms, but it reads much better with gender-specific pronouns. Feel free to mentally replace the female pronoun with male pronouns.]"

-Don Box

So what is this wonderful CLR you may ask? The CLR is the *Common Language Runtime*. It represents one the of the major design differences between COM+ and .Net. Conceptually, the CLR is similar Java's *Java Virtual Machine* (JVM). Like the JVM, the CLR abstracts the developer from the machine level code. The CLR "manages" the code at runtime and handles tasks such as memory management, object lifecycle, and security. The CLR is one of the foundational elements of .Net.

Like J2EE, .Net specifies solutions to common enterprise development problems. The .Net platform has a rich framework that assists developers who are working on database access, messaging, transactions, or other enterprise type tasks. The platform also supports web applications through *Internet Information Server* (IIS) as well as the ASP.NET runtime.

Unlike J2EE, you are not able to choose your application server vendor. J2EE architects can compare and contrasts the merits of the product from such application server's as IBM's *Websphere* and BEA's *Weblogic*. Today, if you are developing for the .Net platform your application server is made by Microsoft, period. Obviously, this alleviates the portability issues found in porting J2EE applications from one vendor to another. Note that there are two projects from the Open Source/Free Software world that are actively working to port .Net to Linux and other non-Windows platforms.

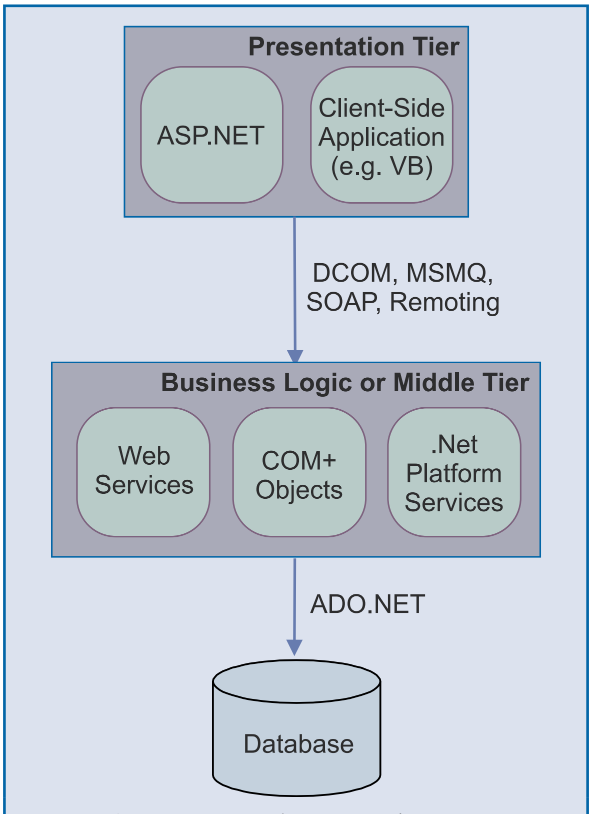
Presentation Tier
ASP.NET
Client-Side Application (e.g. VB)
DCOM, MSMQ, SOAP, Remoting
Business Logic or Middle Tier
Web Services
COM+ Objects
.Net Platform Services
ADO.NET
Database

Figure 1 - **Example .Net Architecture with Tiers and Services**

- Mono (www.go-mono.com)
- DotGNU (www.dotgnu.org)

From a design point of view, .Net offers some additional flexibility that J2EE does not in terms of language support. J2EE applications must be written in the Java programming language, while .Net applications can be written using a host of languages including C#, C++, Visual Basic, and even COBOL. This makes it possible for a development team that does not have a uniform background in the same language to develop components that can still interoperate. No matter which .Net supported language you are working with, you compile your code to an *intermediate language* (commonly referred to as "IL") instead of machine code. The IL and the CLR provide type safety for the component interoperation across the disparate programming languages. The IL and CLR ensure that while data types such as strings and integers are handled in different manners by the different programming languages (C# and Visual Basic support different data types, for example), these types are translated and handled in a consistent manner within the CLR.

Tiers (again)

In the previous issue, we examined the Model-View-Controller (MVC) architecture since this is used as a reference for J2EE applications. In the .Net world the terminology is slightly different, but the concepts are generally similar. The classical approach to application architecture from the COM/COM+ world has been N-tier architecture. Like MVC, N-tier requires strict separation between the GUI, logical and data elements in the application. .Net adds a more service-based approach to the traditional N-tier architecture. Since .Net is still new there is not yet a detailed, formalized architecture in place like N-tier or MVC that articulates the optimal way to leverage all of its features.

In terms of communication options between the tiers of the application, .Net adds two new methods in addition to DCOM which is how COM objects communicate over the wire. .Net supports DCOM as well as:

- .Net Remoting (using TCP or HTTP)
- ASP.NET (using SOAP)

For an in depth examination of Architectural options in .Net, please see msdn.Microsoft.com.

Now we will move on to the building blocks of a secure component system: *Authentication, Authorization, and Privacy*.

Authentication in .Net

So, .Net has a well defined model for component developers to program against and it also supports a strict decoupling of GUI, business logic, and data code, now how do you get a user logged on to your system? Authentication is the front door to your component system. Proper identification of the user enables more detailed security checks for authorization and confidentiality.

As with J2EE, .Net pushes component developers and architects to leverage the framework rather than coding all functionality from scratch. The spirit of .Net is to utilize the standard framework for low level plumbing and allow the developers to focus on writing business logic. Behind the scenes at authentication time, the CLR handles the task of creating the user's context and manages and associates the user's *Identity* and *Prinicpal*.

The .Net framework defines four methods for authenticating a user onto the system.

- Form-based authentication: utilizes HTML forms and cookies to manage session state

- Passport authentication: this is the much-maligned service that has Microsoft managing user's identity. For an in depth discussion of the risks inherent in centralizing identities, refer to: http://avirubin.com/passport.html.

- Windows authentication: utilizes standard Windows authentication mechanisms such as Kerberos

- Internet Information Server (IIS): IIS can authenticate using a Windows directory or different user store depending on configuration. IIS can also support certificate-based authentication. Many projects use IIS as an authentication server, but it is important to assess whether this functionality necessitates its use.

  Described in the ISB (Volume 6, Issue 9) articles "IIS Web Servers: It's Time to Just Be Careful" and "IIS: It's Time to Just Say 'No'" by E. Eugene Schultz and Ric Steinberger, respectively, IIS comes with a host of security issues, and it should not be introduced to your .Net application unless it is providing some value beyond just authentication.

While the ability to use the user stores and management tools in Active Directory can be appealing, from a design point of view the flexibility to not be bound to a specific user store is a very valuable asset in .Net.

Choosing the proper authentication mechanism is an important first step in securing your system. Note also it is critical to understand the ongoing impact on user administration based on which route you choose. While it is generally straightforward to change authentication program code, it can be problematic to migrate back end user stores from one implementation to another.

Authorization in .Net

Once the user is authenticated it is then possible to authorize users to perform certain actions based upon the design and configuration of the system. Authorization in .Net can be handled using:

- ACLs on the Windows file system

- URL authorization: Users and roles are given permissions based upon the URI namespace. The permissions can grant or deny access

- Principal objects: .Net can authorize calls based on Windows or Generic principal objects based upon the object's context at runtime

For more fine-grained authorization, it is necessary to use roles. At a logical level, roles should be mapped to the actions that a user can perform. The design of the system governs how roles are mapped to components. If your development team uses *Use Cases* to describe functionality, then the *Actors* in the diagrams are a good starting point to specify the *Users* and *Roles* for a given component.

As with J2EE, .Net roles can be implemented in a declarative or programmatic fashion. The same design guidelines apply to .Net authorization implementation. Declarative authorization provides a cleaner and simpler administrative interface and enables method level authorization. Programmatic authorization (via the `IsInRole` call) can enable different authorization strategies and decompose the security checks even further within the methods. As was stated in part 1 of this paper [ISB0705, May 2002], a blended approach that uses high level roles at a declarative level and more detailed checks at a programmatic level is worthwhile considering.

In an instance where a specific user can run as multiple users, impersonation can be used. As its name implies, impersonation provides the ability for a user, object or context to impersonate key attributes in a given scenario. This may desirable when traversing from one tier to another, for example.

Evidence-Based Security

At this point, you may ask yourself: "Where does the CLR get the information it needs to determine the validity of the information it is authenticating and authorizing?" The CLR gathers evidence to determine if the inbound call meets the criteria defined in the security policy. The evidence that is examined includes the URL, site, and zone as well as signatures. Based upon the CLR's assessment of the evidence and the policy, the CLR can determine access rights to a given piece of functionality.

Privacy

Cryptographic operations for adding privacy to .Net systems are available to developers in a variety of ways. As with the rest of .Net, the cryptographic functionality is equally available to any .Net supported language whether the developer is using Visual Basic or C++. So a developer using Visual Basic who may not be particularly well versed in the guts of crypto can still take advantage of it via the .Net framework. This approach has its pluses and minuses obviously, it is up to the security and architecture teams to ensure a well designed crypto system. Cryptography systems are too complex to be left to the "lone developer" as if the system were just another database access class.

.Net supports standard encryption algorithms such as RC2, DES, Triple DES, Rijndael, and RSA. Encryption functionality is available to developers either though the CryptoAPI which is a holdover from the COM days or as a stream in the .Net framework.

Streams can be implemented as back end streams or passthrough streams. Back end streams typically write data through to a file or memory stream. Passthrough system read data in and then write out a separate stream.

Simple Object Access Protocol (SOAP)

SOAP is one of the key technologies in the .Net platform. In the next part of the paper we will discuss SOAP, its role in .Net and its role in interoperability between .Net and J2EE.

Conclusion

There are many keys to a secure .Net component deployment. The team structure and methodology must support input on critical design decisions from multiple stakeholders not just the development team. It is crucial for the security team to understand the .Net terminology and components so that they can contribute to a secure system.

The .Net framework is flexible enough to support a sophisticated security implementation, but this flexibility is only valuable if analysis and design work is done to take advantage of the security functionality.

Gunnar Peterson is a Software Architect. He designs secure, stable, and scalable solutions for complex problem spaces. Over his ten year career, he has been dedicated to design and development of distributed middleware Object-Oriented and Component systems for clients ranging from large enterprises to start ups. Currently, Gunnar is CTO of Arctec Group. Arctec Group's primary focus is "Strategies for Industrial Strength, Integrated Architectures." He is shown here on the trail of the wily cutthroat trout deep in the Rocky Mountains. He can be reached at gunnar@arctecgroup.net.