Component Security Design Considerations for J2EE and .Net - An Architectural View Part 1

29

Gunnar Peterson

"An architect's first work is apt to be spare and clean. He knows he doesn't know what he's doing so he does it carefully and with great restraint... The second is the most dangerous system a man ever designs."

- Fred Brooks

Introduction

This article describes the security issues and considerations for the two most prominent middleware frameworks: Sun's *Java 2 Enterprise Edition* (J2EE) and Microsoft's *.Net*.

The series has three parts. The first two parts give an overview of the major security components for J2EE and .Net, respectively. Part three compares the two and details security issues regarding common interoperability scenarios.

This paper focuses on security issues for J2EE. It is written from an architectural viewpoint. We will examine design considerations and challenges facing enterprise component level developers.

J2EE in brief

J2EE is is based on the Java language, but it is different from Java. J2EE requires that Java is used as the programming language (this is a key difference from .Net which allows for many different languages to be used). J2EE itself is much more than just a programming language. J2EE is a standards-based platform which enables developers to solve enterprise problems in an efficient manner.

J2EE defines solutions to common enterprise issues. J2EE has classes to handle messaging, database access, clustering, and most of the tasks an enterprise developer deals with on a regular basis. The J2EE specification is very broad, for the purposes of this article the focus will be primarily on the J2EE component model Enterprise JavaBeans (EJB).

EJB Overview

EJBs are designed to insulate the enterprise developer from common low level problems like transaction monitoring, networking, clustering, and authentication. EJBs run inside of an application server. The application server vendor develops and packages the solution that enable these low level details to be hidden from the developer and dealt with at an administrative level.

Application servers are developed by 3rd parties. not directly by Sun. Sun's role in J2EE is to define the specification (although Sun does have an application server: iPlanet). Examples of application servers include BEA's Weblogic and IBM Websphere. Competitive advantage is gained by the application server vender based on how well their implementation scales, clusters and performs based on their implementation of the J2EE standard.

For a detailed look at EJBs, read Richard Monson-Haefel's *Enterprise JavaBeans* published by O'Reilly.

Tiers

A current trend in software architecture is to decouple the system into three or more tiers. Some examples of these modular designs are "3 Tier", N Tier" and "Model-View-Controller."

In a standard Model-View-Controller (MVC) pattern (see Figure 1), EJBs typically comprise the Controller and Model layers. EJBs do not handle any GUI tasks so these must be implemented elsewhere, typically these View operations would be handled by servlets, JSPs, or an applet.

It is often said that any problem in computer science can be solved by adding a layer of indirection. The controller layer (also known as "the middle tier") enables this indirection. The con-

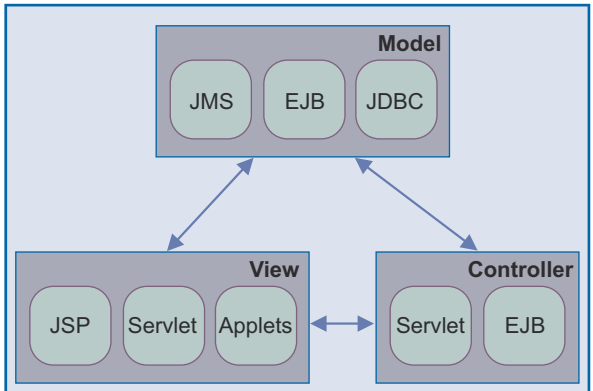

Figure 1 - The MVC Architecture

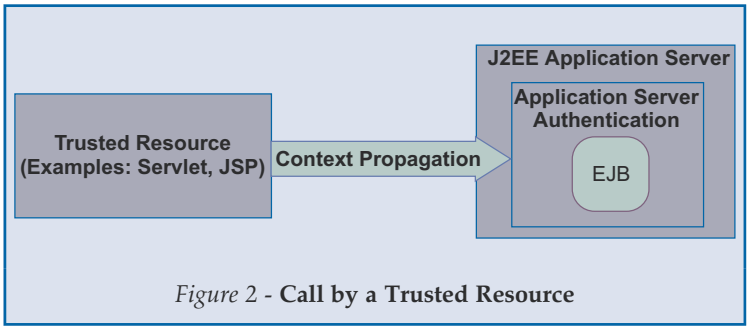

Figure 2 - **Call by a Trusted Resource**

troller layer is the area where the business logic and behavioral elements are stored in MVC pattern. The model layer is responsible for the data-centric operations. Since the controller layer does not have to be concerned with either data or view centric operations, it is free to call out to other subsystems, "glue" together disparate data sources, execute logical algorithms, and enfore enterprise policies and rules.

EJBs, then, are strategically located from a security standpoint. The logical, behavioral and data elements encapsulated in EJBs are a critical part of the security picture. These are the prized assets that the security system must protect.

Many organizations focus on security at a network level (firewall, IDS). If they deal with application security in depth at all, it is frequently at the front door level (View Layer) only. While it is vital to secure the front door, it is just as necessary to carry the security policy throughout the other tiers of the application.

In a bank, the front door is secure to be sure, but the safe (where the assets are) is even more formidable. In practice today, many systems have no safe. To take the bank analogy one step further, application level real time monitoring could be considered the bank's camera system. However, J2EE does not define a standard for monitoring. A custom solution can be cobbled together using logging and/or Java Management Extension (JMX).

Secure Components

Secure applications must have support for authentication, authorization, and privacy. There are many other factors to deal with regarding application security, but these elements generally comprise the building blocks for a secure structure.

At a Controller and Model level, the client application that is calling the EJBs has hopefully already been authenticated and authorized by a trusted source. This trusted source can be the application server as most J2EE server support the common clients in the View layer (JSP, Servlets). The most straightforward paradigm is if the call to the EJB is coming from a trusted resource in the same application server. If this is the case, the security context is propagated to the application server and the security principal (the security credentials) can be understood by the application server. The EJB layer then uses lazy authentication (see Figure 2).

If the call does not come from a resource that can be trusted or if the resource does not interoperate at an authentication level with your application server, then it may be necessary to implement a trusted resource between the calling client application and the EJB layer. This additional trusted resource, for example a servlet, can handle the authentication issues. The EJB is then configured to only accept requests from this servlet. The security context is then propagated from the servlet as in the above diagram. If your organization's architecture has multiple vendor's application servers, then the interoperability on a EJB to EJB call from disparate servers may require this additional authentication scenario (see Figure 3).

Portability

One of the primary goals of Java as well as J2EE is to be portable across different operating systems. Some cynics describe Java's "Write once, run anywhere" claim with the statement, "Write once, debug anywhere", however, in large part the promise has been fulfilled. Nevertheless, portability of an EJB application presents a different set of issues than porting a Java application. Portability with EJB applications is more of an issue with porting to different application servers than to different operating systems.

Since the J2EE specification leaves security and other low level plumbing up to the application

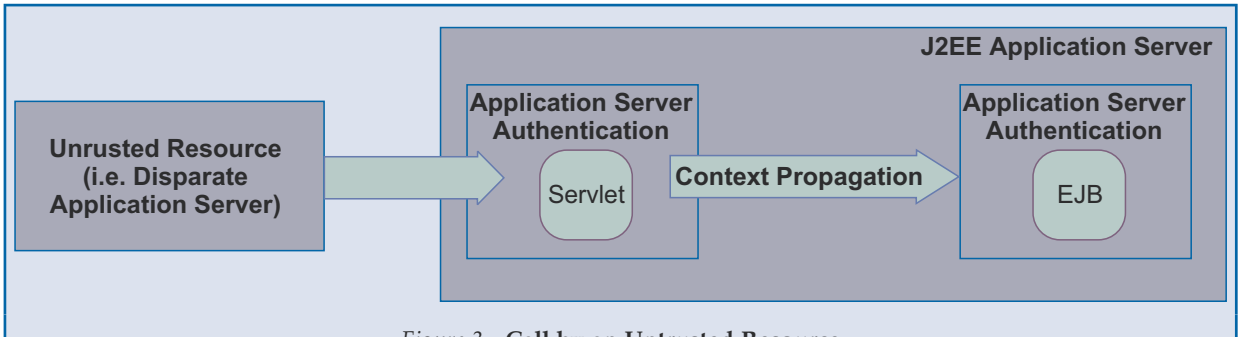

Figure 3 - **Call by an Untrusted Resource**

server vendor, when you develop security for your EJBs you can inadvertently tie your application to that specific application server vendor. Your application will usually port successfully from one operating system to another as long as it is still run within the same application server If you decide to port from Websphere to Weblogic, for example, you will need to closely inspect the security aspects of your EJBs to ensure a clean migration.

Note also that the issues regarding vendor-specific security plumbing also impact interoperability between disparate application servers. Due to the different implementations, communicating the security context from one vendor's application server to another can create interoperability issues. Some or all of the context may not be communicated correctly.

Added to the EJB 2.0 specification, the *Java Authentication Authorization Service* (JAAS) is designed to remedy many of the above problems.

Authentication

JAAS is a big step forward for EJB security. One important feature is that JAAS provides a facade that hides the low level authentication data stores from the client programmer. JAAS can be configured to authenticate against a variety of LDAP servers, RDBMS or other stores (like file systems). The client programmer only has to communicate with the interface and does not have to deal with connecting directly to LDAP.

This feature can also streamlines the development life-cycle. A team can quickly begin development against one type of user data store and switch the user data store out for a more "production ready" one later in the life cycle. This change should not require recoding from the client programmer's point of view.

For web clients, J2EE provides four different methods to authenticate: *Basic, Form-based* (can be encrypted), *Digest* (encrypted username and password), and *Certificate*. These types of authentication are implemented differently depending upon your application server vendor. Since this is the front door to the web, it is critical to understand what the choices are and how they are configured in your particular application server. The flexibility to choose different methods of authentication helps support more sophisticated security policies at an administrative rather than programmatic level.

Once the client is authenticated by the application server the context can be propagated to the EJB tier. Validating the propagated context can be handled in a variety of ways, as described in the next section.

In keeping with the overall design spirit of J2EE, JAAS insulates the client developer from authentication details. Splitting the roles of development and administration is a good idea for a number of reasons and security is high on this list. Since, J2EE supports a strict separation of duties it is possible to have one team do development, another team configure security and yet another team handle deployment and administration.

The overall design of J2EE supports a better balanced approach to software development. The key security components are componentized so they can be architected by a representative from the enterprise security team who does not necessarily have to have a developer level understanding of the entire J2EE platform. The enterprise security representative should, however, have a far deeper understanding of the letter and the spirit of the information security policy and related issues than a typical developer.

The power of this model has yet to be fully realized, because many organizations still leave all J2EE responsibilities up to developers who have to balance all of these concerns.

Authorization

J2EE and EJBs support the notion of Roles. Roles should map to logical and behavioral elements. In the bank example, a bank may have different roles defined for opening a new account and depositing money into an account. The bank can assign the "NewAccount" role to Sally and Bob and "DepositMoney" only to Sally. Sally can open an account and deposit money, while Bob can only open a new account.

EJBs can work with roles defined at declarative and programmatic levels. At a high level many of the design decisions regarding EJB security come down to when to use declarative or programmatic security for authorization.

Declarative security is generally considered by architects to be the holy grail. The developer is insulated from many of the security implementation details since the security is configured at deployment time by the administrator. At deployment time the administrator can configure the authentication and authorization settings for each EJB with no impact on the code.

More importantly, once the application is running in production the security model can be changed and roles and authorized users can be added, deleted or updated. Again, these changes can be done without reworking the code or rebuilding the system.

Logical roles that authorize the user to perform specific functions can be mapped to the EJBs at the method level. A given EJB can support multiple types of users and groups as well as different usage scenarios. A single EJB could implement read, write, and update methods with different combinations of roles mapped to each of these methods.

In this model the EJB container manages the authorization responsibilities and insulates the de-

veloper from dealing with them at a coding level. The tradeoff for being insulated from these issues is a lack of fine-grained control and the inability to provide instance level authorization.

With programmatic security, the developer is back in the driver's seat. The developer can implement authorization checks (typically calling `isCallerInRole`) however, wherever he or she chooses. As with declarative security the developer can implement roles on a method-by-method basis.

The developer has the flexibility to add logical algorithms based upon the Caller's role. This enables the system to descend below the high level role and enforce lower level rules based upon a given user's parameters and permissions.

The price to pay with this approach is that the developer's main concern is generally developing business logic and not security. Programming mistakes can be made and they are much harder to detect in the code than at the deployment level. The administrator is generally charged with being much more security aware and will more easily adapt to and find security issues in a declarative model. In the declarative model, the security configuration is visible and manageable at the application server administration console level rather than the code level only.

The other issue with heavy reliance on programmatic security is ongoing administration and changes to policies, roles, users and groups. When role checking is altered in a programmatic model, coding changes are necessary. It is also necessary to recompile, rebuild and deploy the system. This can cause the system to be brought down for redeployment and creates additional work for the development and administration teams.

This design decision comes down to the choice on how to balance convenience and control. The decision of how and when to use each of these is one of the first major design decisions you need to make.

The middle road is to implement both. Handle high level tasks that can be broken down into users, groups, and roles at the container level using declarative security. For lower level issues, finer grained roles and permissions, and attribute-based and instance level issues consider programmatic security.

RunAs

EJBs have one other important method regarding roles. The `RunAs` method enables the EJB deployer to assign attributes and credentials to the EJB at deployment time from an administrative level. This method allows the EJB components to use impersonation. `RunAs` is an effective way to manage component to component calls especially when calling from one layer to another. Imagine a bank employee who is a teller and is empowered to execute wire transfers as well. When he or she executes the wire trasfer this is "RunAs" as the authorised wire transfer user.

Privacy

Cryptographic functions are defined in the *Java Cryptography Extension* (JCE). The JCE supports public and private key systems (including standard algorithms such as Blowfish, DES, and RSA), message digests, digital signatures, and key management operations. JCE provides a full suite of tools to implement a sophisticated cryptographic subsystem. A full discussion of JCE is beyond the scope of this article, but Sun's Javasoft website has a good deal of documentation on JCE.

In addition to the standard JCE implementation, the Cryptix Foundation has developed the Cryptix JCE which provides additional algorithms and functionality.

One of the main issues with JCE is performance. If this is a concern in your application, it is possible for JNI to call out to a C or C++ based cryptographic application to enable better performance. JNI has its own share of isues, so it is important to balance your need for speed with the brittle nature of JNI.

As with any use of cryptography, you should consult the legal documents to see if there are any implications for usage in your locale.

Using SSL

In a standard J2EE application there are two non-obvious areas in which to consider using SSL (the obvious way is between the web server and client).

The first is in a basic and form authentication scenario where passwords are sent in clear text. SSL can provide privacy for these. The contents can then be encrypted using JCE to encrypt both the pipe and its contents.

The second is that the wire level protocol for EJB communication is RMI-IIOP. There is no direct support for SSL in RMI-IIOP, but for an additional layer of protection, J2EE applications can use RMI-IIOP over SSL to secure these network communications.

Classloaders

The Java language uses a class loader to load and verify byte codes as well as handling execution and security tasks. In a security conscious application, the Secure class loader should be used. The Secure Class loader enables permission checking for class loading operations.

Final Note on Application Server Assessment

One of the chief benefits of the J2EE architecture is the ability to choose "horses for courses." The various application servers each have strengths and weaknesses. If your organization is at the point of assessing application servers, then note that the security implementation is one of the primary differentiators between the application server brands.

Conclusion

EJBs are a critical part of Sun's J2EE strategy. The EJB security model is flexible enough to enable multiple different paradigms and patterns for securing an enterprise's valuable assets. The EJB model lends itself to an architectural approach, which empowers a wider audience, including the enterprise security team, to participate alongside developers and architects to design secure components solutions.

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 can be reached at gunnar@arctecgroup.net.