CHAPTER 14

Security Design and Management

In this chapter

THE IMPORTANCE OF APPLICATION SECURITY

For all the talk and attention that Internet security gets these days, for some reason it often takes a back seat to other considerations during application design. Maybe it's because the nonfunctional requirements often are overlooked due to the importance that is placed on "the product working like it's supposed to." Or maybe it's because of the overall complexity of designing and building a proper security framework. The amount of planning and forethought for security planning and construction can consume a large amount of a project's cycle. The irony about an application's security framework is that if it's working like it's supposed to, no one will notice it. When it's not working like it should, everyone will notice. This might be another reason why not enough attention is given to the application security requirements. Whatever the real reasons are, the results of not paying enough attention to the security considerations can be disastrous for the application and possibly the company.

Obviously, not all applications have the same exact requirements placed on them from a security perspective. However, for typical B2C and B2B Internet applications, there are many similarities when it comes to security design and constraints. Most of these applications are distributed component-based applications. The key point in that sentence is "distributed." Because these components are physically distributed over a network, there are more security holes that possibly can be exploited by attackers and unauthorized users.

The types of networks that these components use to communicate with one another can vary greatly, but often some portion of the application must be exposed to an unprotected open network such as the Internet. For example, a browser that makes a call to a servlet or JSP page typically will send the request, and the data within the request, over the Internet to the Web server, which usually is listening on a well-known port. As this request travels over the open Internet, many bad things can happen along the way. The request might contain the customer's credit card information for an order. If an unauthorized person were to intercept the request and get this information, you can imagine how unhappy this customer would be.

Because most Web servers listen on a common set of port numbers, extra precautions must be taken to protect the customer's information and requests. This is just one piece of the security puzzle with which application designers must deal. This chapter takes a closer look at some of the other security issues that you must consider when designing and building EJB applications. Like many other things in software development, the earlier you deal with these issues during analysis and design, the better the chances you'll have of building a more secure and resilient application.

UNDERSTANDING YOUR APPLICATION'S SECURITY REQUIREMENTS

As we stated earlier, not all target environments have the same security needs and constraints. However, there are some broad generalities we can make about typical EJB applications. The following list describes some of the common security-related features or aspects:

- Physically separated tiers
- User-level access based on username/password
- Different vendor products used throughout the application
- Sensitive and nonsensitive data being used

PHYSICALLY SEPARATED TIERS

A typical EJB application might have three or more physical tiers, all running on separate machines. The Web tier usually is on a server that is placed where Internet or intranet HTTP traffic can reach it. The Application tier usually is on a server located in the enterprise's protected network infrastructure. It's typically not exposed to the Internet directly, because the traffic to it usually comes from the Web server.

Many EJB vendors these days provide Web servers inside the EJB server itself. This usually can give better performance and provide for better maintenance because everything is centrally located. The problem with this approach, however, is that the entire tier might have to be exposed closer to the Internet because of this lack of separation between the two tiers. You should give plenty of thought to your security requirements before taking advantage of this configuration. Be sure you have other strong measures in place to protect someone from getting into your application server and causing damage to the system.

The third tier usually is a database server that is used explicitly by the application tier and possibly other enterprise resource planning (ERP) systems. The database houses the mission-critical data for the application, including important customer-sensitive data. The Database tier should be located deep in the company's protected network infrastructure with no path to it from the outside world. If an attacker does get at this data, it could spell the end for the company and many customers' credit reports. There have been several incidences lately where hackers were able to get a list of credit-card numbers for customers that did business with an online company. This is the worst possible thing that could happen for an Internet company and its product. Always be sure to protect this data and never expose it to unauthorized individuals. You probably want to go as far as encrypting sensitive information in the database to ensure that even if someone gets credit-card numbers, they won't be able to use them easily.

Continuing with the sweeping generalities, Figure 14.1 presents a physical network topology for a typical EJB application. The figure shows how and where security measures are usually applied.

Figure 14.1 shows that there is usually at least one demilitarized zone (DMZ) where components are somewhat exposed to the Internet or some other unprotected networks. The DMZ is the part of the network that is most susceptible to intruders and attacks. The DMZ area is given much more attention for security considerations than other areas that are located deeper in the company's intranet. This is usually done with a combination of software and hardware configurations.

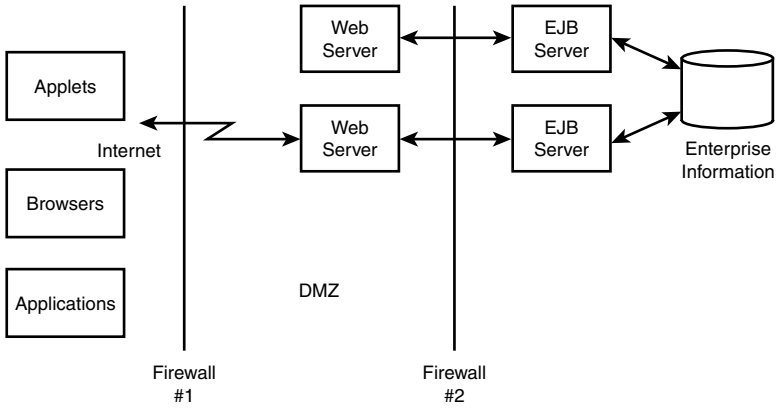

Figure 14.1
A typical Enterprise JavaBeans network security topology.

USER-LEVEL ACCESS BASED ON USERNAME/PASSWORD

Another common feature of EJB applications is that end users can be authenticated with a username and password. The username and password attributes are the only information that is typically provided by the end user to be identified. To protect sensitive information such as this, Web applications use digital certificates. Certificates are installed on the Web servers for the application and use the Secure Sockets Layer (SSL) protocol to protect customer data that must be sent from the client browser to the Web server. By using HTTPS rather than just the HTTP protocol, data will be sent encrypted and not in the clear. This helps ensure confidentiality and integrity of the user's data and requests.

Digital certificates are most often installed on the Web server, but usually not on the end user's browser. If a digital certificate is installed on both the Web server and the client's browser, this form of authentication is known as *mutual authentication* and is not commonly done on B2C or B2B applications. It might be more prevalent in B2B applications, but even this isn't the norm. SSL usually is sufficient.

DIFFERENT VENDOR PRODUCTS USED

Unless you are using an EJB server that includes the Web server and you are taking advantage of this feature, you generally have products from different vendors throughout the enterprise application. One of the goals of the EJB and J2EE architectures is to allow for developers to choose the best vendor for a specific technology. The problem associated with different vendors is that sometimes the integration process is immense.

Fortunately, interoperability has been given plenty of attention from the EJB and J2EE specifications, so many of the interoperability problems have been solved. However, security interoperability is one of the weakest parts of the specification. This is not to say that it can't and is not being done, it's just that part of the specification seems to be behind when compared to some of the other areas. If your components do have to communicate in a secure fashion, one choice is always to use the SSL protocol. Because RMI/IIOP is the standard

wire protocol between J2EE clients and containers, SSL is a nice solution because IIOP can be used on top of the protocol when communicating between the Web tier and EJB container, for example.

Sensitive and Nonsensitive Data Being Used

Not all applications need to encrypt data that is sent from tier to tier. In most cases, just the communications between the Client tier and Web tier might need to be protected. This is not always the case, but it's true more often than not. Encryption doesn't come without a price. There is a negative impact on performance and administration when you need to use encryption to protect the data. Most applications will change into a secured mode only when it's absolutely necessary. Others might use HTTPS from the moment the customer sends the username and password. You must think about when you actually need to use encryption to protect the data. It really depends on your customer base and when certain data is being sent to and from the user's browser.

Basic Security Concepts

One of hardest things about understanding security design and construction is figuring out what all the terms mean and how everything fits together. This section attempts to provide a clear, simple definition for these terms so that we can have a foundation for the rest of the chapter.

Authentication and Authorization

Authentication is the process of entities proving to one another that they are acting on behalf of specific identities. For example, when a Web user provides a username and password for a login, the authentication process verifies that this is a valid application user and that the password matches the real user's provided password. Various types of authentication mechanisms can be used. Other than no authentication, two main categories are employed in the various EJB products, although the actual naming conventions might be different.

Weak or *simple authentication* is where the user provides a username and password to be authenticated. The user provides no other authentication information. This probably is the most common form of authentication in EJB applications. One main concern with simple authentication is that if someone else gets your username and password, they can assume your identity.

As you might expect, *strong authentication* is more secure than simple or weak authentication. This is where the user provides a digital certification or other private means of being authenticated. It's much harder for someone to get your digital certificate from your machine. Even if they do, the certificate is good for only a particular machine and will be pretty much worthless to them.

Other authentication mechanisms can be used as well. Sometime within the next year or two, banks are planning to introduce automatic teller machines (ATMs) with a security measure that scans the user's iris. Although we might be a few years away from users of

eBay.com wanting to get their eyes scanned before they can log in, newer types of authentication are being developed. Another up and coming authentication mechanism involves fingerprint scans. This actually is used in some larger government-type systems that need more security for the system.

Authorization differs from authentication in that authentication is about ensuring only valid users get access into an application, whereas authorization is more about controlling what the authenticated user is allowed to do after they get into the application.

Authentication happens first, and then authorization should happen next, assuming authentication succeeds. For some simple EJB applications, it's possible that only authentication needs to be used. However, for many applications, there is some type of administrator functionality that a normal user should not have access to. One of the ways that this can be prevented is by creating a list of permissions for actions that a user can perform and then checking this permission list against the actions attempted by the user.

Authorization typically is much harder and more complex to perform. Some applications can get by without doing much authorization, although by adding authorization to the framework and making it possible, you will save yourself many headaches later trying to incorporate it.

DATA INTEGRITY

Data integrity is the means or mechanism of ensuring that data has not been tampered with between the sender and the receiver. It ensures that no third party could have modified the information, which is possible when it's sent over an open network. If the receiver detects that a message might have been tampered with, it would probably want to discard the message.

CONFIDENTIALITY AND DATA PRIVACY

Confidentiality is the mechanism of making the information available to only the intended recipient. Ensuring that the system you are communicating with is really the one that you intended to communicate with is the biggest part of this concept. There are many ways hackers can trick you into sharing sensitive data. There was a case recently where a lesser-known security hole allowed hackers to modify DNS entries and cause traffic from an actual bank to be rerouted to a fake site. The fake Web site set up the Web pages to look exactly like the bank's site and attempted to capture the user's username and password, which could then be used on the real site to gain access. Digital certificates help solve most of the associated problems, but you must keep your eyes open.

NONREPUDIATION

This is one of the most misunderstood security concepts. Nonrepudiation is the act of proving that a particular user performed some action. For example, if a user submitted a bid for an auction, through proper record keeping and audit trails, the system administrators could prove that the action was performed by the particular user's account. It doesn't mean that that owner of the account actually submitted the bid, but you can prove their account was used and that it's not just a data error.

Auditing is sometimes overlooked, but it's invaluable when an action that was performed on a user's account has to be verified. Other auditing features include invalid login attempts, which can point to possible attacks on the system.

PRINCIPALS AND USERS

A *principal* is an entity that can be authenticated by the system. This is typically an end user or another service requesting access to the application. The principal is usually identified by a name; most often the username that the end user uses to log in to the system.

SUBJECT

Subject is a term taken from other security technologies and applied to EJB recently with the introduction of Java Authentication and Authorization Service (JAAS) 1.0. A subject holds a collection of principals and their associated credentials. The idea of needing something broader than a principal came about because there are many systems where you might need different principals or credentials to access the various parts of an application. By using a subject that might hold on to these various principals and credentials, applications can support such things as single sign-ons.

CREDENTIALS

When an end user wants to be authenticated to the application, they must usually also provide some form of credential. This credential might be just a password when simple authentication is being used, or it might be a digital certificate when strong authentication is used. The credential usually is associated with a specific principal. The specifications don't specify the content or format of a credential, because both can vary widely.

GROUPS AND ROLES

Groups and roles sometimes can be thought of as the same thing, although they are used for different purposes. A *group* is a set of users who usually have something in common, such as working in the same department in a company. Groups are used primarily to manage many users in an efficient manner. When a group is granted permission to perform some action, all members of the group gain this permission indirectly.

A *role*, on the other hand, is more of a logical grouping of users. A bean provider might indicate that only an admin user can close an auction, but the bean provider doesn't usually have knowledge of the operational environment to establish the exact group to which a user must belong to close an auction, for example. There typically is a mapping of roles to the groups in the operational environment, but the deployer or application assembler handles this mapping.

ACCESS CONTROL LISTS (ACLS) AND PERMISSIONS

Permissions for an application represent a privilege to access a particular resource or to perform some action. An application administrator usually protects resources by creating lists of users and groups that have the permissions required to access this resource. These lists are

known as access control lists (ACLs). For example, a user with auction admin permissions may create, modify, or close an auction, but a user that has only bidder permissions may be allowed to participate only in the bidding process for an auction.

An ACL file is made up of AclEntries, which contain a set of permissions for a particular resource and a set of users and groups.

SECURITY REALM

A security realm is a logical grouping of users, groups, and ACLs. The physical implementation of a security realm normally is done by a relational database, an LDAP server, a Windows NT or Unix security realm, or, in some very simple cases, a flat file. The realm is usually checked when authentication or authorization must occur to allow access for a user to the application. With some EJB vendors, a caching realm is used and loaded from the original security realm to help increase performance. When authentication or authorization occurs, the caching realm is checked first and, if it can authenticate or authorize from there, there's no need to incur the database IO. The caching realm usually is flushed often to ensure that dirty reads do not occur.

JAVA SECURITY FUNDAMENTALS

To really understand the security mechanisms available to you in EJB, it would help to understand what security infrastructure is available from the core Java language and where it helps with EJB applications and where it falls short. This section introduces the security aspects of the Java language, but does so from a high enough level as not to complicate our discussion of EJB application security. Although the two have some ties, it's not absolutely necessary to understand the entire Java security model to program enterprise beans.

The security architecture in Java has evolved three significant times since it was first created. The changes were primarily made to ease some of the restrictions that were placed on Java applications and applets in the early releases. The Java security model has always been conservative, which you want from a security perspective, but the restrictions came at a price, which made it not so easy to get a consistent security policy for applications and applets alike. Although the use of applets is arguably less than it was in the early days of Java, it still helps to understand the reasoning for the changes.

Java 1.0 introduced the *security sandbox*, which confined untrusted code to run in a very protected area where it could not negatively affect other running systems or system resources. This was necessary because the client browser downloads applets and runs them on the local machine. A client didn't necessarily want an applet to be able to read and write to and from the file system, because severe damage to the user's data can take place. On the other hand, applications were given free reign to the system resources from a security perspective because they typically were launched locally. A component known as a `SecurityManager` is responsible for determining on which resources untrusted code is allowed to operate.

With Java 1.1, applets were allowed to run out of the sandbox, as long as they were signed with a private key. If an applet was unsigned, it was forced back into the sandbox model. Although this allowed signed applets to have the same possible resource access as an application, it still wasn't very flexible for developers.

JDK 1.2 (Java 2) introduced several improvements over the previous Java security models. First and foremost, it added the capability for applications and applets to use security policies in the same manner, which permitted a more flexible and consistent security mechanism for application developers. The Java security policy defines a set of permissions that grant specific access to resources such as the file system or sockets. Listing 14.1 shows some of the permissions in the default policy file that are installed with the SDK 1.3. Some of the lines have been wrapped to make them fit on the page.

LISTING 14.1 THE DEFAULT POLICY FILE INSTALLED FOR SDK 1.3

```
grant codeBase "file:${java.home}/lib/ext/*" {
  permission java.security.AllPermission;
};

grant {
  permission java.lang.RuntimePermission "stopThread";
  permission java.net.SocketPermission "localhost:1024-", "listen";
  permission java.util.PropertyPermission "java.version", "read";
  permission java.util.PropertyPermission "java.vendor", "read";
  permission java.util.PropertyPermission "java.vendor.url", "read";
  permission java.util.PropertyPermission "java.class.version", "read";
  permission java.util.PropertyPermission "os.name", "read";
  permission java.util.PropertyPermission "os.version", "read";
  permission java.util.PropertyPermission "os.arch", "read";
  permission java.util.PropertyPermission "file.separator", "read";
  permission java.util.PropertyPermission "path.separator", "read";
  permission java.util.PropertyPermission "line.separator", "read";
  permission java.util.PropertyPermission "java.specification.version", "read";
  permission java.util.PropertyPermission "java.specification.vendor", "read";
  permission java.util.PropertyPermission "java.specification.name", "read";
  permission java.util.PropertyPermission
      "java.vm.specification.version", "read";
  permission java.util.PropertyPermission
      "java.vm.specification.vendor", "read";
  permission java.util.PropertyPermission
      "java.vm.specification.name", "read";
  permission java.util.PropertyPermission "java.vm.version", "read";
  permission java.util.PropertyPermission "java.vm.vendor", "read";
  permission java.util.PropertyPermission "java.vm.name", "read";
  permission java.net.SocketPermission
      "1024-65535", "accept, connect, listen, resolve";
  permission java.net.SocketPermission
      "localhost", "accept, connect, listen, resolve";
};
```

The runtime system structures code into individual groups called *security domains*. Each domain contains a set of classes and, because permissions are defined at the domain level, all the classes within a particular domain have the same access permissions. This allows for a

much more flexible security model, while at the same time allowing for configuration similar to the sandbox approach. By default, applications still have unlimited access, but if required, they can be constrained within a domain by using a security policy and installing a `SecurityManager` for the application. You can specify a `SecurityManager` for an application either by supplying one on the command line as a system property or by setting one up programmatically at the start of the application.

By using security policies, the security implementation can be separated from the policy. Figure 14.2 shows a diagram of the Java 2 security architecture.

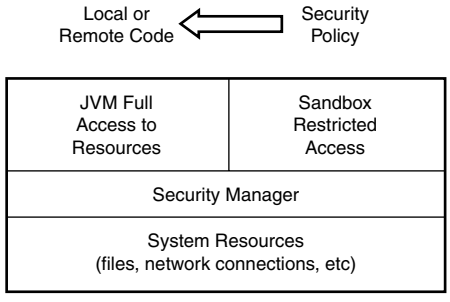

Figure 14.2
The Java 2 security architecture.

THE JAVA `ClassLoader`

The Java `ClassLoader` is responsible for loading Java byte codes into the Java Virtual Machine (JVM). It partners with the `AccessContoller` and the `SecurityManager` to ensure that the security policies are not violated. There are different types of class loaders, and third-party components can create a customer class loader to provide security features beyond those offered by the Java 2 standard security model.

One very important version of the class loader is called the "System" `ClassLoader`. This type of class loader helps launch the initial JVM by reading in classes and packages that are essential in starting the runtime system.

PERMISSION CLASSES

Permission classes are at the root of the Java security model. They allow or deny access to the system resources such as files, sockets, RMI objects, and so on. The set of permissions, when mapped to classes, can be conceptually thought of as the security policy for an application. A security policy file is used to configure the security rules for an application. The security policy file is a text file that can be viewed or edited by hand or by using the policy tool located in the `bin` directory of the Java home directory.

THE JAVA `SecurityManager`

The `SecurityManager` checks to ensure that the action that is being requested does not violate the security policies established in the security policy for an implementation. The

SecurityManager works with the AccessController to verify whether the permission should be granted or denied. If an unauthorized permission is attempted, it is the job of the SecurityManager to raise a security exception back to the requesting component.

THE AccessController CLASS

The AccessController class decides whether access to a system resource should be granted or denied based on the current security policy being used. The AccessController also has several static methods that can be used by an application to help check whether the calling component has the proper permission to access a resource. An AccessControlException will be raised if access is denied.

THE AccessControlContext CLASS

In normal situations, the SecurityManager delegates permission checks down to the AccessController class. The AccessController uses the context within the current thread to determine whether to grant the permission. In some situations, however, it's necessary to do work in a separate thread but still maintain the proper security context. This is where the AccessControlContext class can help. For example, if you needed to create a worker thread and allow it to have the same permissions as the parent thread, you can create an AccessControlContext object from the AccessController and pass it on to the worker thread to use for permission checks. This concept of obtaining the security context from the current thread and passing it or propagating it on to another thread will become very important when we talk about how J2EE containers propagate security information from one container to another during remote calls.

PRIVILEGED CODE

As the previous sections explained, the policy for an installation specifies what resources can be accessed based on the set of permissions for a protection domain. It sometimes is necessary for an application to override these restrictions and perform an otherwise unauthorized action. Marking code as *privileged* enables a piece of trusted code to temporarily grant access to more resources than are available directly to the code that called it.

Whenever a resource access is attempted, all the code that is called by the execution thread must have permission to access the particular resource, or an AccessControlException will be thrown. If the code for any caller in the call chain doesn't have the requested permission, the exception is thrown, unless one of the callers whose code does have the permission has been marked as privileged and all the callers called after this caller also have the permission.

To mark code as privileged, you can use the doPrivileged feature located on the AccessController class. The following code fragment illustrates how you might mark some code as privileged:

```
public class MyPrivilegedAction implements java.security.PrivilegedAction {

  public MyPrivilegedAction() {
    super();
  }
```

```
  public Object run(){
    // privileged code would go here
    FileInputStream stream = new FileInputStream( "aFile" );
    // do some work with the file

    // Nothing to return
  }
}

// In some other class here

public void someMethod(){
    // Other code here
    MyPrivilegedAction action = new MyPrivilegedAction();
    // Changed to privileged
    java.security.AccessController.doPrivileged( action );
    // Once the privileged action finished, back to normal mode
  }
```

If you need to return a value from the `run` method, you'll need to cast it to the correct class stereotype. If the code in the `run` method might possibly throw a checked exception such as a `FileNotFoundException`, you will need to use the `PrivilegedExceptionAction` instead. The following code fragment illustrates how this might be handled:

```
public class MyPrivilegedAction implements PrivilegedExceptionAction {

  public MyPrivilegedAction() {
    super();
  }

  public Object run(){
    // privileged would go here

    // Nothing to return
  }
}

public void someMethod() throws java.io.FileNotFoundException {
    // Other code here
    MyPrivilegedAction action = new MyPrivilegedAction();
    // Changed to privileged

    try{
      FileInputStream inStr =
        (FileInputStream)java.security.AccessController.doPrivileged( action );
      //Once the privileged action finished, back to normal mode

      // The PrivilegedActionException is just a wrapper around the
      // real exception that occurred.
    }catch( PrivilegedActionException ex ){
      // Assuming a FileNotFoundException although you might really
      // want to check for this to be safe
      throw (FileNotFoundException)ex.getException();
    }
  }
```

When the privileged code is finished, the application should go back to the normal policy and permission use. Be very careful when using this feature, and keep the section of code that is executing as privileged as small as possible to prevent security holes.

USING SECURITY WITH ENTERPRISE JAVABEANS AND J2EE

Security is an important part of the J2EE and EJB specifications, although many EJB developers argue that there is much more that the specifications need to account for from a security perspective. The J2EE 1.3 and EJB 2.0 Specifications are better than the previous versions when it comes to specifying standards for dealing with security issues. Three main security goals are set for the EJB architecture:

- Lessen the burden placed on the bean provider for dealing with security issues.
- Allow the EJB applications to be portable across different vendor's servers and allow the different vendors to use different security mechanisms.
- Allow support for security policies to be set by the deployer or assembler rather than by the bean provider.

The EJB and J2EE specifications describe two entirely different methods of handling security in enterprise beans and in other J2EE components. These two methods are called programmatic and declarative security.

USING PROGRAMMATIC SECURITY

The EJB 2.0 Specification recommends not using programmatic security in your enterprise beans because it's too easy to couple your application to the physical security environment. If you needed to deploy your application in other security domains with different roles, it might make it necessary to have to change source code to work correctly in this new environment.

Even though it's not recommended, there are still situations that arise that make it necessary to use programmatic security in your applications. Applications should use programmatic security mainly when the declarative method does not offer enough flexibility or when business requirements dictate the need.

For the most part, either an enterprise bean or a servlet can use programmatic security. When doing programmatic security within EJB, you can use the methods defined in the `EJBContext` interface:

```
public boolean isCallerInRole(String roleName);
public Principal getCallerPrincipal();
```

The `isCallerInRole` method tests whether the principal that made the call to the enterprise bean is a member of the role specified in the argument. The `Principal` is typically propagated over from another tier, and the security context information resides in the current thread. The following fragment shows an example of how you might use the

`isCallerInRole` method in an enterprise bean:

```
//Other enterprise bean code here
//...
public void submitBid( Integer auctionId,
     double newBidAmount, String bidderUserName )
     throws InvalidBidException, InvalidAuctionStatusException{

    // Make sure this user is a valid bidder
    if ( !getSessionContext().isCallerInRole( "bidder" )) {
      throw new InvalidBidException( "You must register first" );
    }

    // Get the home interface for the english auction bean
    EnglishAuctionHome auctionHome = getEnglishAuctionHome();

    try{

      // Locate the correct auction
      EnglishAuction auction = auctionHome.findByPrimaryKey( auctionId );

      // Locate the bidder
      Bidder bidder = null;

      // Try to submit the bid
      auction.submitBid( newBidAmount, bidder );

    }catch( FinderException ex ){
      ex.printStackTrace();
    }catch( RemoteException ex ){
      ex.printStackTrace();
    }
  }
```

This is the `submitBid` method in the `AuctionHouseBean` class. If the user is not a member of the `bidder` role, they are not allowed to submit a bid, and an `InvalidBidException` is thrown. This would force a user to register before submitting bids for an auction.

Depending on the setting in the `security-identity` element in the bean's deployment descriptor, the `getCallerPrincipal` method returns the `Principal` object for the current caller. When the `security-identity` element has a `use-caller-identity` value in it, the original caller of the enterprise bean will be propagated when the bean makes calls on itself.

If the deployer specifies a `run-as` element in the deployment descriptor, a different principal other than the initial caller might be returned from this method. A deployer can set the `security-identity` to another principal to execute with more permissions than the current caller. For example, it might need to invoke an operation as an administrator, but the operational environment doesn't want to map all callers to this group directly.

The following example shows an example of how you might use the `getCallerPrincipal` method in an enterprise bean:

```
//Other enterprise bean code here
//...
Principal principal = getSessionContext().getCallerPrincipal();
String bidderName = null;
if ( principal != null ) {
  bidderName = principal.getName();
}else{
  bidderName = "Unknown";
}

// Log the user's bid attempt to a file
String msg = bidderName + " submitted a bid for auction: " + auctionId );
logMessage( msg );
```

Programmatic security is done similarly in a servlet by using these two methods on the `HttpServletRequest` interface:

```
public boolean isUserInRole(String roleName);
public Principal getUserPrincipal();
```

These methods allow the components to make business logic decisions based on the security role of the caller or remote user. Whether the servlet makes a call to a security realm in the application tier or has the information cached on the Web tier is entirely up to the container's implementation.

Caution

The form and context of the principal names will vary greatly depending on the authentication and vendor used.

When an enterprise bean uses the `isCallerInRole` method within an enterprise bean, the bean provider must declare each security role referenced in the code using the `security-role-ref` element. The following example illustrates how an enterprise bean's references to security roles are used in the deployment descriptor:

```
<ejb-jar>
  <enterprise-beans>
    ...
    <entity>
      <ejb-name>EnglishAuction</ejb-name>
      ...
      <security-role-ref>
        <description>The auction restricts some operations to valid bidders
        </description>
        <role-name>bidder</role-name>
      </security-role-ref>
      ...
    </entity>
    ...
  </enterprise-beans>
  ...
</ejb-jar>
```

The deployment descriptor indicates that the enterprise bean `EnglishAuction` invokes the `isCallerInRole` method using the role of "bidder." There can be multiple `security-role-ref` elements for an enterprise bean, one for each different role used as an argument to the `isCallerInRole` method. The role name is scoped only to the enterprise bean that declares it, so if you use the same role name in a different enterprise bean, you'll need to declare a `security-role-ref` element in the deployment descriptor for that bean as well.

USING DECLARATIVE SECURITY

Declarative security is done by expressing an application's security policy, including which security role or roles have permission to access an enterprise bean, in a form that is completely external to the application code. The application assembler uses one or more `security-role` elements in the assembly instructions in the deployment descriptor. Here's a sample deployment descriptor that includes a `security-role` element added by the assembler:

```
<ejb-jar>
  <enterprise-beans>
    ...
    <entity>
      <ejb-name>EnglishAuction</ejb-name>
      ...
      <security-role-ref>
        <description>The auction restricts some operations to valid bidders
        </description>
        <role-name>bidder</role-name>
        <role-link>registered-bidder</role-link>
      </security-role-ref>
      ...
    </entity>
    ...
  </enterprise-beans>
  <assembly-descriptor>
    <security-role>
      <description>A role to represent users who have registered with the
        system as authorized auction participants
      </description>
      <role-name>registered-bidder</role-name>
    </security-role>
  </assembly-descriptor>
  ...
  ...
</ejb-jar>
```

The deployment descriptor includes a `security-role` element that defines a role of "registered-bidder."

Note

The roles defined in the `security-role` element do not represent roles in the physical operation environment. They are used only to define a logical security view of an application. They should not be confused with user groups, principals, and other security concepts that exist in the operational environment.

It's also a requirement for the application assembler to map any `security-role-ref` elements defined by the bean provider to the `security-role` elements. The assembler does this by inserting a `role-link` element in the `security-role-ref` element that references one of the valid `security-role` elements.

The application assembler is not required to add `security-role` elements to the deployment descriptor. The reason that the assembler would do it in the first place is to provide information to the deployer so that the deployer doesn't have to have intimate knowledge about what the business methods are for or are doing. If the assembler adds no `security-role` elements to the deployment descriptor, it's up to the deployer to understand the business methods in the enterprise beans and the operational environment to conduct the mapping. The `security-role` elements are scoped to the deployment descriptor and would need to be duplicated in other `ejb-jar.xml` files.

If the application assembler does provide one or more `security-role` elements in the deployment descriptor, they can also specify the methods of the home and remote interfaces that each role is authorized to invoke. The assembler defines the method permissions in the deployment descriptor using the `method-permission` element. Each `method-permission` element can contain one or more security roles and one or more methods. The following illustrates how an assembler might configure the method permissions:

```
<ejb-jar>
  <enterprise-beans>
    ...
    <entity>
      <ejb-name>EnglishAuction</ejb-name>
      ...
      <security-role-ref>
        <description>The auction restricts some operations to valid bidders
        </description>
        <role-name>bidder</role-name>
        <role-link>registered-bidder</role-link>
      </security-role-ref>
      ...
    </entity>
    ...
  </enterprise-beans>
  <assembly-descriptor>
    <security-role>
      <description>A role to represent users who have registered with the
        system as authorized auction participants
      </description>
      <role-name>registered-bidder</role-name>
    </security-role>
    <security-role>
      <description>
        A role to represent a user who has permission to close an auction
      </description>
      <role-name>authorized-agent</role-name>
    </security-role>
  </assembly-descriptor>
  ...
  ...
```

```
 <method-permission>
  <role-name>registered-bidder</role-name>
  <role-name>authorized-agent</role-name>
  <method>
    <ejb-name>EnglishAuction</ejb-name>
    <method-name>getLeadingBid</method-name>
  </method>
 </method-permission>
</ejb-jar>
```

There can be multiple `method-permission` elements in the deployment descriptor. The method permission for the enterprise beans is defined as the union of all the method permissions defined in the deployment descriptor.

There are three different ways of writing a `method-permission` element within the deployment descriptor. The first method is used for referring to all the remote and home methods of a specified bean:

```
<method-permission>
  <role-name>registered-bidder</role-name>
  <method>
    <ejb-name>AuctionHouse</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
```

The wildcard "`*`" is used to indicate the roles that can access all the methods on both interfaces. The second style of declaring a `method-permission` element is

```
<method-permission>
  <role-name>registered-bidder</role-name>
  <method>
    <ejb-name>AuctionHouse</ejb-name>
    <method-name>submitBid</method-name>
  </method>
</method-permission>
```

It is used to specify a particular method of the home or component interface. If there are multiple overloaded methods with the same name, this style would grant access to all the different overloaded methods with the same name.

If there are overloaded methods with the same name and you would like to reference a particular method, you can use the third method:

```
<method-permission>
  <role-name>registered-bidder</role-name>
  <method>
    <ejb-name>AuctionHouse</ejb-name>
    <method-name>submitBid</method-name>
    <method-params>
      <method-param>java.lang.Double</method-param>
    </method-params>
  </method>
</method-permission>
```

The `method-params` element contains a list of fully qualified java types of the method's input parameters in order. If you want to choose an overloaded method that takes no parameters, you would have an empty `method-params` element like this:

```
<method-params>
</method-params>
```

If the method contains an array, the `method-param` element would look like this:

```
<method-params>
  <method-param>int[]</method-param>
</method-params>
```

SPECIFYING IDENTITIES IN THE DEPLOYMENT DESCRIPTOR

The application assembler can specify whether the original caller's security identity should be used to execute methods within an enterprise bean or whether a specific run-as identity should be used. This doesn't affect the original caller's permission to call a bean, but it does affect the permissions associated with the bean when it calls other methods or beans. To do this, the assembler uses the `security-identity` element in the deployment descriptor. The value of this element can either be `use-caller-identity` or `run-as`. If `run-as` is specified, this element must include a `role-name` entry to define the security identity to be taken on by the bean. Because a message-driven bean doesn't interact directly with a caller, `run-as` is the only option if you want to control a message-driven bean's security identity. The assembler doesn't have to provide the `security-identity` element within the deployment descriptor. In this case, it's the responsibility of the deployer to determine which caller identity should be used when one component invokes an operation onto another.

MAPPING THE DEPLOYMENT ROLES TO THE PHYSICAL ENVIRONMENT

Up to this point in our discussion of setting up and defining the security view of our enterprise beans, we have said that the roles that are defined in the deployment descriptor are just logical roles and that the deployer would be responsible for performing the mapping of these logical roles to the ones that exist in the operational environment. The specifications leave it up to the vendor as to how this happens and, to be quite honest, not many of the vendors have provided a very flexible way to do this for anything but the most trivial security setups.

In some cases, the vendor expects you to put principal names directly into the deployment descriptor for the enterprise bean or servlet. Arguably, this is where EJB shows its immaturity the most. If you build an EJB application that you then sell and install for a customer, you don't want to have to modify the XML deployment descriptors just because the principals are different. Also, what about an existing customer that wants to add or delete an existing principal; does that mean you are going to have to redeploy a component?

You definitely should attempt to follow the intent of the EJB specification or suffer the consequences of lack of portability and interoperability if you don't. In some cases, however, you'll need to think out of the box and provide your own implementation. Security just

might be one of those places. The next section discusses how the auction's application needs have some special security requirements and how we'll fulfill those requirements by building in our own security model.

SKETCHING OUT THE AUCTION SECURITY

As you saw in the previous section, the security features provided by the EJB and servlet containers are sufficient for many types of EJB applications. However, as we pointed out in the beginning of this chapter, some things are not covered by the specifications. For example, what if your Web application wanted to cache the user's security context in the Web tier to prevent redundant network calls to the security realm, which is typically located in the application tier? Suppose that you had a set of requirements to not show certain buttons, hyperlinks, or tabs depending on the user's roles and permissions. If you had to make several network calls while dynamically spitting out a JSP page, your performance would definitely suffer.

To understand this a little better, let's take a look at what happens in a typical Web-enabled EJB application. Our auction application consists of a set of JSP pages located on the Web tier that will dynamically allow certain features depending on who you are and what role you are playing for a particular session. Two scenarios will emphasize the problem.

- Scenario 1—User Bob posts an auction for others to bid on.
- Scenario 2—User Bob submits a bid for an auction posted by someone else.

When Bob posts an auction for others to bid on, he's acting in the role of Auctioneer. This role has certain permissions when it comes to managing this particular auction. Bob would be an Auctioneer only for auctions that he created. He would be given the ability to cancel the auction, assign an early winner to the auction, and respond to questions about the auction. However, Bob must not have this ability when viewing other auctions. Therefore, the security framework should be capable of distinguishing the two roles based on the dynamic data for the user. Although the EJB security framework would prevent Bob from making invocations on particular servlets or methods within an enterprise bean, there's no real local way to get at the permissions that an auction user has been granted without going back and forth to the application tier.

Another problem is that servlets and enterprise beans are role-based. This means that you either are in a role or are not. If you are in the role, you have permissions for everything the role has been granted. If you don't belong to a particular role, you are restricted from all permissions granted to that role. There's no way to assign or remove a single permission without putting the user in a role or taking them out of a role. It would be nice and much more flexible if we could not only assign permissions to a group, but also assign them directly to the user. The EJB security architecture doesn't allow for this directly, so we'll have to design our own way of handling this set of requirements if we truly need this behavior.

For our auction application, we need to provide a way for the Web tier to get a set of groups and permissions that the client has been granted and then cache this information on the Web tier for performance. Because we are caching these on the Web tier, changes made to the security realm itself will not be reflected to the user during a user's live session. However, after the user logs out and then comes back in later, the changes will be reflected in the security context information that is marshaled to the Web tier.

So, the plan for our auction example is going to include a session bean called `SecurityManager` that will be called only by a special login servlet on the Web server. We also will create an `AdminSecurityManager` session bean that will be responsible for creating, updating, and deleting users and groups. This session bean could be used from an admin application within the Web tier or, maybe to add more security, it might be called only by an application installed within the intranet. This separation of responsibility helps with security and also provides a more cohesive interface for each component because the responsibilities are arranged in a logical manner. The following code fragments show the steps for a `login` method inside the `SecurityManagerBean` class:

```
public SecurityContext login( String userName, String password )
  throws InvalidLoginException {

  SecurityContext secCtx = null;

  // Get a database connection from the datasource and look for the user
  // and make sure the account is still active

  // If the user doesn't exist or is inactive, throw an exception

  // If the user does exist, build the security context information

     // Get the user's permissions and groups and build the collections

     // return the context back to the caller
     return secCtx;
}
```

CREATING THE AUCTION SECURITY REALM SCHEMA

For our example, we are going to be storing users, groups, and permissions in a relational database. We will need to create the database schema for these three tables. Listing 14.2 shows the DDL for our security schema.

LISTING 14.2 THE SAMPLE AUCTION SECURITY REALM SCHEMA

```
#=======================
# Table SecGroup
# Represents a Security Group for the Auction Application
#=======================
CREATE TABLE SecGroup (
  Id int NOT NULL,
  Name varchar (255) NOT NULL,
  Description varchar (255) NOT NULL
);
```

LISTING 14.2 CONTINUED

```
ALTER TABLE SecGroup ADD
    CONSTRAINT PK_SecGroup PRIMARY KEY (Id);

#========================
# Table SecUser
# Represents a Security User for the Auction Application
#========================
CREATE TABLE SecUser (
  Id int NOT NULL,
  FirstName varchar (255) NOT NULL,
  LastName varchar (255) NOT NULL,
  EmailAddress varchar (255) NULL,
  UserName varchar (255) NOT NULL,
  Password varchar (255) NOT NULL,
  AccountCreatedDate date NOT NULL,
  LastLoginDate date NULL,
  IsAccountActive varchar (1) NOT NULL
);
ALTER TABLE SecUser ADD
    CONSTRAINT PK_SecUser PRIMARY KEY (Id);

#========================
# Table SecUserSecGroup
# Represents a link table between User and Group
#========================
CREATE TABLE SecUserSecGroup (
  SecUserId int NOT NULL,
  SecGroupId int NOT NULL,
  IsGroupActive varchar (1) NOT NULL
);
ALTER TABLE SecUserSecGroup ADD CONSTRAINT
    PK_SecUserSecGroup PRIMARY KEY
    (SecUserId, SecGroupId);

ALTER TABLE SecUserSecGroup ADD
  CONSTRAINT FK_SecUserSecGroup_User FOREIGN KEY
    (SecUserId) REFERENCES SecUser (Id);

ALTER TABLE SecUserSecGroup ADD
  CONSTRAINT FK_SecUserSecGroup_Group FOREIGN KEY
      (SecGroupId) REFERENCES SecGroup (Id);

#========================
# Table Permission
# Represents a permission that a user or group can perform
#========================
CREATE TABLE Permission (
  Id int NOT NULL,
  Name varchar (255) NOT NULL,
  Description varchar (255) NOT NULL
);
ALTER TABLE Permission ADD
    CONSTRAINT PK_Permission PRIMARY KEY (Id);
```

LISTING 14.2 CONTINUED

```
#=======================
# Table SecUserPermission
# Represents a link table between User and Permission
#=======================
CREATE TABLE SecUserPermission (
  SecUserId int NOT NULL,
  PermissionId int NOT NULL
);
ALTER TABLE SecUserPermission ADD
    CONSTRAINT PK_SecUserPermission PRIMARY KEY
    (SecUserId, PermissionId);

ALTER TABLE SecUserPermission ADD
  CONSTRAINT FK_SecUserPermission_User FOREIGN KEY
    (SecUserId) REFERENCES SecUser (Id);

ALTER TABLE SecUserPermission ADD
  CONSTRAINT FK_SecUserPerm_Permission FOREIGN KEY
      (PermissionId) REFERENCES Permission (Id);

#=======================
# Table SecGroupPermission
# Represents a link table between Group and Permission
#=======================
CREATE TABLE SecGroupPermission (
  SecGroupId int NOT NULL,
  PermissionId int NOT NULL
);
ALTER TABLE SecGroupPermission ADD
    CONSTRAINT PK_SecGroupPermission PRIMARY KEY
    (SecGroupId, PermissionId);

ALTER TABLE SecGroupPermission ADD
  CONSTRAINT FK_SecGroupPermission_Group FOREIGN KEY
    (SecGroupId) REFERENCES SecGroup (Id);

ALTER TABLE SecGroupPermission ADD
  CONSTRAINT FK_SecGroupPerm_Permission FOREIGN KEY
      (PermissionId) REFERENCES Permission (Id);
```

In the schema in Listing 14.2, we've included only the necessary attributes to understand the design. You might need more attributes, depending on your requirements. This schema was tested on Oracle 8i. If you want to test this on other database vendors, you might have to make a few modifications to the schema to support these other vendors. Don't worry too much about the exact definition of this security schema. There could be some normalization or denormalization on it, depending on how much you like normalized databases. The schema isn't presented to show a good database design, but rather to give you an idea of what types of table and attributes must be supported for the auction security realm.

DESIGNING ACCESS TO THE SECURITY REALM

The security objects are pretty lightweight objects, which means they don't contain many attributes or even a great deal of business logic. Choosing whether the security objects are entity beans or not depends on several factors, one of which is your particular strategy for making things entity beans or not. Making concepts in your logical model an entity bean can solve many of the transactional and concurrency headaches associated with persistent objects. You also can gain much more scalability because the container handles the life cycle for the enterprise bean and is able to shuffle resources as needed. All these things are true; however, you still don't want everything from your logical model translating into an entity bean. For one thing, if no client needs to access the data remotely, this can be one argument for not being an entity bean. Of course, there are others.

➔ For more information on what types of objects should be entity beans, **see** "Entity Beans," **p. 105**.

If you don't want to use entity beans and you are using bean-managed persistence, an alternative solution is to access the data in the relational database directly from the session beans. The session beans could return immutable view classes back to the client by using JDBC directly from within the session bean methods. There are some benefits to using this approach; however, there are some transactional and concurrency problems that you must deal with. If the administrator is updating the data and the client is reading it, concurrency must be dealt with to ensure that no transactional problems occur.

There are several Object to Relational Mapping (ORM) frameworks that can provide help in this area. One such ORM is TOPLink from WebGain. TOPLink provides both a CMP and a BMP solution for EJB persistence and also deals with more complicated issues, such as data caching and transactional issues.

We are not going to provide the entire solution for the data-accessing problem here, but the recommendation for the auction example would be to use session beans to access the data and return immutable view classes to the client. This solution is not the most elegant, but it will definitely work for this situation.

USING SECURITY ATTRIBUTES IN THE WEB TIER

When the Web tier calls the `SecurityManager` session bean and attempts to log in, an object called `SecurityContext` will be returned if the login is successful. Each user will have its own `SecurityContext` instance cached in the `HttpSession`. The `SecurityContext` object will be used to validate the user's permission to perform actions within the auction Web site.

The `SecurityContext` object will contain a collection of roles or groups of which the user is a member, as well as a collection of permissions. The permission collection is a union of all the permissions from the groups to which the user belongs, as well as any extra permissions assigned directly to the user. This type of security design could also support negative permissions as well, rather than just additive. For example, if a user belongs to an "auctioneer" group that has a `cancelAuction` permission, we could easily add a column to the permission table called Additive that determines whether the permission should be added to the list of

permissions or subtracted from the list. This gives the administrator more flexibility to determine how permissions are assigned or removed.

Listing 14.3 shows the `SecurityContext` class that will be built by the security session bean and returned to the Web tier.

LISTING 14.3 THE `SecurityContext` SOURCE REPRESENTING A USER'S SECURITY CONTEXT INFORMATION

```
/**
 * Title:       SecurityContext<p>
 * Description: The user's security context information.<p>
 */
package com.que.ejb20.entity.businessobjects;

import java.security.Principal;
import java.util.Collection;

public class SecurityContext implements java.io.Serializable {

  private java.security.Principal principal;
  private java.util.Collection groups;
  private java.util.Collection permissions;

  public SecurityContext() {
    super();
  }

  public Principal getPrincipal() {
    return principal;
  }

  public void setPrincipal( Principal newPrincipal ) {
    principal = newPrincipal;
  }

  public void setGroups( Collection newGroups ) {
    groups = newGroups;
  }

  public Collection getGroups() {
    return groups;
  }

  public void setPermissions( Collection newPermissions ) {
    permissions = newPermissions;
  }

  public Collection getPermissions() {
    return permissions;
  }

  public boolean isUserInRole( String role ) {
    return this.groups.contains(role);
  }
```

LISTING 14.3 CONTINUED

```
  public boolean checkPermission( String permission ) {
    return this.permissions.contains( permission );
  }
}
```

The two most important methods in the `SecurityContext` class are `isUserInRole` and `checkPermission`. The client uses these two methods to determine to which security roles the user belongs and which security permissions have been granted to the user. Here's a code fragment that illustrates how a client can use these methods to hide or show a Close Auction button:

```
// Assume a SecurityContext has already been obtained
// Verify that the user is acting as the role auctioneer for this session
if ( secCtx.isUserInRole( "auctioneer" )) {
  // Check to see if they have the closeAuction permission
  if ( secCtx.checkPermission( "closeAuction" )) {
    // Show a close auction button here
  }
}
```

The `Principal` reference in the `SecurityContext` class is an interface from the Java 2 security package that represents the user. We are going to provide a `UserView` class that implements this interface and acts as the user in the system. Listing 14.4 shows the `UserView` class that is built by the `SecurityManager` and returned to the client.

LISTING 14.4 THIS CLASS REPRESENTS A USER WITHIN THE SYSTEM

```
/**
 * Title:       UserView<p>
 * Description: A view of the user in the system<p>
 */
package com.que.ejb20.entity.businessobjects;

import java.io.Serializable;
import java.security.Principal;

public class UserView implements Principal, Serializable {

  private String firstName;
  private String lastName;
  private String emailAddress;
  private String userName;
  private String password;
  private String accountCreatedDate;
  private String lastLoginDate;
  private String active;

  public UserView() {
    super();
  }
```

Listing 14.4 Continued

```
public String getFirstName() {
  return firstName;
}

public void setFirstName(String newFirstName) {
  firstName = newFirstName;
}

public void setLastName(String newLastName) {
  lastName = newLastName;
}

public String getLastName() {
  return lastName;
}

public void setEmailAddress(String newEmailAddress) {
  emailAddress = newEmailAddress;
}

public String getEmailAddress() {
  return emailAddress;
}

public void setUserName(String newUserName) {
  userName = newUserName;
}

public String getUserName() {
  return userName;
}

public void setPassword(String newPassword) {
  password = newPassword;
}

public String getPassword() {
  return password;
}

public void setAccountCreatedDate(String newAccountCreatedDate) {
  accountCreatedDate = newAccountCreatedDate;
}

public String getAccountCreatedDate() {
  return accountCreatedDate;
}

public void setLastLoginDate(String newLastLoginDate) {
  lastLoginDate = newLastLoginDate;
}

public String getLastLoginDate() {
  return lastLoginDate;
}
```

LISTING 14.4 CONTINUED

```
  public void setActive(String newActive) {
    active = newActive;
  }

  public String getActive() {
    return active;
  }

  // Method implementation needed because this class implements the
  // java.security
  public String getName() {
    return this.userName;
  }
}
```

If you were using JSP pages on the client, it might be a good idea to wrap the security checks inside a JSP Custom Tag library. This might make the JSP pages a little cleaner because they wouldn't have to access the `SecurityContext` object directly. If an instance of a `SecurityContext` class were stored in the session for each user, the JSP Tag handler would have direct access to it and could do all the checks for the JSP Page. The JSP page would just include the tag library information within it. You can find more information on JSP custom tags at

`http://java.sun.com/products/jsp/taglibraries.html`

PROPAGATING THE PRINCIPAL

There's one final note on implementing security in this manner. When a client invokes an operation on an enterprise bean, the principal is propagated to the EJB object from the client. This propagation is taken care of by the container or the stub classes, depending on the vendor's implementation. With the security design that we have discussed here, the `Principal` is not being associated with the current thread by our implementation, and it might not be propagated to the enterprise bean correctly. This would present some problems if the container has security attributes set up for the beans.

It might be a good idea to associate the `Principal` that is returned in the `SecurityContext` object with the current thread; this sometimes is referred to as *Thread-Specific Storage (TSS)*. Some EJB servers will associate the JNDI principal with the current thread when a client creates a remote interface and uses this principal to invoke calls on enterprise beans. In theory, the JNDI principal and credential are supposed to be used only to authenticate and authorize access to the naming and directory service. Several vendors use this security information for calls to the enterprise beans. Just be careful when taking advantage of this because chances are it will not be portable.

JAVA AUTHENTICATION AND AUTHORIZATION SERVICE (JAAS)

Within the J2EE 1.3 and EJB 2.0 Specifications, a new security-related technology for EJB applications called Java Authentication and Authorization Service (JAAS) is introduced. JAAS is a Java implementation of the standard Pluggable Authentication Module (PAM) framework. The goal of the PAM framework is to design an authentication mechanism that is independent of the application layer. In other words, an administrator should be able to plug in various authentication mechanisms on a per-application basis without affecting the application logic itself. You can find more information on the PAM framework at

```
http://java.sun.com/security/jaas/doc/pam.html
```

JAAS is a standard extension to the Java 2 SDK 1.3. The Java 2 security model only provides access controls based on where the code originated from and who signed the code. The Java 2 security model does not provide the capability to additionally enforce access controls based on who runs the code. JAAS compliments the Java 2 security model with this type of support. JAAS probably will be part of the core Java language with SDK 1.4 (code name Merlin) when it's released sometime in 2001.

As the name implies, JAAS can be divided into two main components: an authentication component and an authorization component.

AUTHENTICATION

The authentication component provides the capability to reliably and securely determine who is currently executing Java code. This is true regardless of whether the Java code being executed is an applet, an application, a JSP page, or a servlet.

Note

The authentication capability does not exist with the Java 2 security model. This is absolutely essential behavior for most EJB applications. Prior to JAAS, most applications had to build their own authentication support.

JAAS authentication supports different implementations to be plugged in without affecting the Java application using it. This allows applications to take advantage of the various security authentication technologies without having to rewrite your software. For example, if one customer needed to use a relational database to store user information and another used Lightweight Directory Access Protocol (LDAP), you could just plug in different implementations without negatively affecting the application.

AUTHORIZATION

The authorization component of JAAS extends the existing Java 2 security framework by restricting users from performing actions depending on who the user is and on the code source. After the user is authenticated, the system obtains the actions that are allowed for this user and remembers this throughout the life cycle of the user's current session with the application.

Note

JAAS supports a security policy similar to the Java 2 security policy. In fact, the JAAS policy is an extension and understands the permissions in the Java 2 policy file like `java.io.FilePermission` and `java.net.SocketPermission`.

JAAS CORE CLASSES

The main package for JAAS is the `javax.security.auth` package. Although three packages exist under the main package, it probably makes more sense to talk about JAAS from a logical grouping of classes, based on what tasks they perform in JAAS. A more logical grouping of classes for JAAS is

- Common classes
- Authentication classes
- Authorization classes

Caution

Don't be misled in believing that the classes are really separated into these groupings. It's more logical for us to discuss them this way, but they are grouped entirely differently.

THE COMMON CLASSES AND INTERFACES

Two common components are important to developers using JAAS: the `javax.auth.Subject` class and the interface `java.security.Principal`. The `Subject` represents an entity, such as an individual user or service. A `Subject` can have many principals, each one associated with a different application service. For example, if an application allowed a user to log in to two different parts of a site and the user used a different username for each part of the site, the user (`Subject`) would have two different principals. The `Principal` interface we are referring to here is actually the `Principal` interface that already exists in the Java 2 security framework.

The `Subject` class has two public constructors:

```
public Subject();
public Subject(boolean readonly, Set principals,
       Set publicCredentials, Set privateCredentials );
```

As you'll see later in this section, you also can obtain an authenticated `Subject` from a `LoginContext` class, which we haven't defined yet. The `Subject` class contains methods for getting the set of principals and public or private credentials.

Caution

If you modify the set that is returned from the `getPrincipals`, `getPublicCredentials`, or `getPrivateCredentials` methods in the `Subject`, the original set will also be modified. Make sure you get a copy if you don't want to affect the original set.

Public and private credentials are not part of the JAAS library. You can use any Java class to represent a credential, including something as simple as the `String` class.

Earlier you saw how to execute privileged actions using the `AccessController` class. The `Subject` class contains two static methods for executing privilege actions as a particular subject. The following methods associate the `Subject` with the current thread's `AccessControlContext` and then executes the privileged action by calling the methods on the `AccessController` class that you saw earlier in this chapter:

```
public static Object doAs(Subject subject, PrivilegedAction action );
public static Object doAs(Subject subject, PrivilegedExceptionAction action );
```

There also are two more methods on the `Subject` class that, instead of associating the `Subject` with the current thread's `AccessControlContext`, the `Subject` gets associated with the `AccessControlContext` provided as an argument. The two methods are

```
public static Object doAs(Subject subject,
                          PrivilegedAction action,
                          AccessControlContext ctx);

public static Object doAs(Subject subject,
                          PrivilegedExceptionAction action,
                          AccessControlContext ctx);
```

All these `doAs` methods play a very important role in how the security context information is propagated to a remote container. For example, if a `Subject` has already been authenticated in the Web tier and invokes a remote operation on an EJB server, the Web tier can use the `Subject` and `Principal` information and pass it along to the EJB container, which then can have access to the `Principal` information.

Note

Keep in mind that the behavior of propagating security context information from the current thread to other J2EE containers isn't unique to JAAS. This behavior is how many J2EE containers perform it already. JAAS merely uses the same techniques.

THE AUTHENTICATION CLASSES AND INTERFACES

The classes and interfaces in the authentication logical group deal exclusively with authenticating a `Subject` in the application. The classes and interface involved are `javax.security.auth.spi.LoginModule`, `javax.security.auth.LoginContext`, `javax.security.auth.callback.Callback`, and `javax.security.auth.callback.CallbackHandler`.

The `LoginContext` class provides methods to authenticate a `Subject`, regardless of the authentication mechanism being used. The `LoginContext` object uses a `javax.security.auth.login.Configuration` object to determine which authentication mechanisms to use to authenticate the `Subject`. The `Configuration` is associated with one or more classes that all implement the `LoginModule` interface. Each `LoginModule` is responsible for authenticating the `Subject` for a particular authentication service.

Here are the basic steps to authenticate a `Subject`:

1. Create an instance of the `LoginContext` class.
2. Specify the Configuration file for the `LoginContext` to use.
3. The Configuration loads all the `LoginModules` specified.
4. The client invokes the login method on the `LoginContext`.
5. Each login method in the different `LoginModules` can associate an authenticated principal with the `Subject` if the login succeeds.
6. The `LoginContext` returns the authenticated `Subject` to the client.
7. The client is then free to access the `Subject` and `Principals` from the `LoginContext` object.

We have left a few of the smaller details out here, but the most important steps have been listed and you should get the idea of how this works.

One thing that we have left out of the steps on purpose is discussing how the `Callback` and `CallbackHandler` interfaces are involved in the authentication process. These interfaces and the concrete classes are in the `javax.security.auth.callback` package. They can seem pretty confusing at first, but after you get the picture where they fit in during the authentication process, they make quite a bit of sense. The `LoginContext` class has four constructors. Two of the constructors take an instance of a class that implements the `CallbackHandler` interface. Here are the two methods that take an instance of the `CallbackHandler` interface:

```
public LoginContext(String name,CallbackHandler handler)
throws LoginException;

public LoginContext(String name,Subject subject, CallbackHandler handler)
throws LoginException;
```

The `CallbackHandler` is passed to each `LoginModule` in the `initialize` method. The `LoginModule` then can use the `CallbackHandler` instance to make a callback on the client to request information needed to continue with the authentication process. Typically, this information is a username and password. You might be wondering why you don't just pass

this information to the `LoginContext` or `LoginModule` in the first place. The main reason is that each authentication mechanism is going to be different. Some might use a device to scan the iris of your eyes or scan your fingerprints. By using a callback instead of letting the application handle this up front, the implementation of the authentication mechanism is further decoupled from the application.

There are several concrete classes of the `Callback` interface for doing things such as getting usernames and passwords. Of course, you can implement your own as well.

Note

There has been some debate on how Web-friendly the callback mechanism is. This is because of the differences between the typical synchronous Web page login and the asynchronous callback. There are some solutions to get around this slight mismatch. One solution involves blocking the original thread until the callback thread acquires the information necessary to complete the authentication process. These issues will be addressed in further implementations.

THE AUTHORIZATION CLASSES AND INTERFACES

The last logical grouping of classes deals with the authorization portion of JAAS. After a `Subject` has been authenticated, a client can obtain the permissions that are granted to the particular `Subject` and code source. The permissions granted to a `Subject` are configured in a JAAS policy. The `javax.security.auth.PolicyFile` class is a default file-based implementation provided by JAAS. This file is similar to the Java 2 policy file, which contains one or more grant statements, each of which can contain a set of permission statements.

Each grant statement specifies a codebase/codesigners/Principals triplet, including the permissions that have been granted to that triplet. What this means is that all the permissions will be granted to any code downloaded from the specified codebase and signed by the specified code signer, as long as the `Subject` running the code has all the specified principals in the `Principal` set. The following fragment shows a sample entry in the JAAS policy file:

```
// example entry in JAAS policy file
grant CodeBase http://java.sun.com,
    SignedBy "johndoe",
    Principal com.sun.security.auth.NTPrincipal "admin"
{
    Permission java.io.FilePermission "c:/winnt/stuff", "read, write";
};
```

Note

The `CodeBase` and `SignedBy` components are optional and, if absent, will allow any codebase and signer to match. This includes code that is unsigned as well.