10

Security

CERTIFICATION OBJECTIVES

In an enterprise computing environment, the failure, compromise, or lack of availability of computing resources can jeopardize the life of the enterprise. To survive, an enterprise must identify, minimize, and, where possible, eliminate threats to the security of enterprise computing system resources. *Resources* for our purposes refer to goods and services. A *good* is a tangible property—that is, the physical server. A *service* is an intangible property such as software or data. A threat against a resource is basically an unauthorized use of a good or a service.

Security

Out of the box, Java provides the ability for class code to be easily downloaded and executed. From the point of view of security, the easily downloaded code poses a threat because it may be possible for the code to access enterprise data resources. Therefore, it is important that your system be able to distinguish between code that can be trusted and code that cannot.

The Java security model takes into consideration the origin of the classes, and perhaps who signed them, when it permits or denies operations. This chapter concentrates on threats to services (software and data) and how Java and Java Platform, Enterprise Edition (JEE) fit into the scheme of things. JEE applications do not obviate existing enterprise security infrastructure; they do, however, have value when integrated with existing infrastructures. The JEE application model leverages existing security services as opposed to requiring new services.

This chapter begins with a brief review of threats to security, followed by a look at the security restrictions that Java technology environments normally impose on applets running in a browser. Then an overview of Java and some of its security and related APIs is presented. The rest of the chapter describes the security concerns and explores the application of JEE security mechanisms to the design, implementation, and deployment of secure enterprise applications, this includes the use of Java 5's Annotation facility.

Threats to enterprise resources fall into a few general categories that can overlap, as shown in Table 10-1.

Depending on the environment in which an enterprise application operates, these threats manifest themselves in different forms. For example, in a nondistributed system environment, a threat of disclosure might manifest itself in the vulnerability of information kept in files—for example, a client/server .INI file with user identities, passwords, IP addresses, and listener ports for enterprise databases.

TABLE 10-1 Threats to Enterprise Resources

Threat	Description	Example Result of Threat Execution
Compromise of accountability	In legal parlance, this is known as "fraud in the impersonation" or "identity theft." Someone is masquerading as another user.	UserX logs on as UserY. UserX uses UserY's identity to make system requests and is afforded all rights and permissions of UserY.
Disclosure of confidential information	Enterprise data is intentionally, negligently, or accidentally made available to parties who have no legal "right to know."	Patient medical record compromised; bank account number compromised.
Modification of information	Enterprise data is intentionally, negligently, or accidentally modified.	Corporate money account balance modified; computer virus stored on an enterprise server.
Misappropriation of protected resources	In legal parlance, this is known as "theft of service"; the perpetrator is accessing an enterprise computer system and using the system to perform, on its behalf, services that provide illegal gain or purpose.	UserX gains access to the lottery system and causes the system to create a winning ticket for UserX.
Misappropriation that compromises availability	The service misappropriation or data modification causes an interruption of the enterprise system.	Computer virus causes enterprise server to be unusable; a hacker causes the Amazon.com e-commerce server(s) to be unavailable.

In a distributed system, the code that performs business operations may be spread across multiple servers. A request will trigger the execution of code based on a server, and that code could possibly manipulate enterprise data. To prevent a threat to security, it is important that trusted requests be distinguished from those that are not. The server must verify the identity of the caller to evaluate whether the caller is permitted to execute the code. The client may also want to verify the identity of the server before engaging in the transaction—for example, the consumer will not want to send a credit card number to *www.stealyourcreditcard.com*.

A distributed system is typically made up of code executing on behalf of different *principals* (uniquely identified users or machines within the system). To obviate threats, the server requires that the caller provide credentials that are known only to the caller, as proof of identity. The credentials are then checked and verified with an authority, in what is known as the *authentication process*. Authenticated callers are then checked to determine whether they are permitted to access the requested resource; this is known as *authorization*. These are the fundamental phases in security threat prevention.

Obviously, it is impractical to believe that all threats can be eliminated (because new ones are developed every day). The objective is to reduce the exposure to a minimal and therefore acceptable level through using proper authentication and authorization augmented by the use of the security techniques including signing, encryption, and postfacto auditing. Java also provides some packages that can facilitate the security techniques used by an enterprise, as shown in Table 10-2.

TABLE 10-2 Security Packages of the Java Platform

Package	Description
java.security	Framework of classes and interfaces for security, including access control and authentication. Also provides support for cryptographic operations, including message digest and signature generation.
java.security.acl	Deprecated as of Java 1.2, replaced by classes in *java.security*.
java.security.cert	Classes and interfaces for parsing and managing X.509 certificates, X.509 certificate revocation lists (CRLs), and certification paths.
java.security.interfaces	Interfaces for RSA (Rivest, Shamir, and Adleman AsymmetricCipher algorithm) and DSA (Digital Signature Algorithm) key encryption.
java.security.spec	Classes and interfaces for DSA, RSA, DER, and X.509 keys and parameters used in public-key cryptography.
javax.crypto	Classes and interfaces for encrypting and decrypting data.
javax.crypto.interfaces	Interfaces for Diffie-Hellman public/private keys.
javax.crypto.spec	Classes and interfaces for key specifications and algorithm parameter specifications used in cryptography.
javax.net.ssl	Classes for encrypted communication across a network using the Secure Sockets Layer (SSL).
javax.security.auth	A framework for authentication and authorization used by Java Authentication and Authorization Service (JAAS).
javax.security.auth.callback	Classes providing low-level functionality that obtains authentication data and displays information to a user.
javax.security.auth.kerberos	Classes that support the Kerberos network authentication protocol.
javax.security.auth.login	Provides a plug-in framework for user authentication.
javax.security.auth.spi	`LoginModule` interface for implementing plug-in user authentication modules.
javax.security.auth.x500	Classes for representing X.500 Principal and X.500 Private Credentials.
New or modified packages for Java 5 and above	
javax.annotation	Common classes to support Java 5's annotation facility.
javax.annotation.security	Common security classes to support Java 5's annotation facility.
javax.ejb	Additionally contains classes to support Java 5's annotation facility.

CERTIFICATION OBJECTIVE 10.01

Identify Security Restrictions That Java Technology Environments Normally Impose on Applets Running in a Browser

We will now take a look at the restrictions that are normally imposed upon Java applets that execute within the confines of a browser.

Applets in a Browser

There are two flavors of applet, signed and unsigned. *Unsigned* applets work in a sandbox, are severely restricted, and are very safe to run. A *signed* applet is an applet with a digital signature added that proves that it came untampered from a particular publisher. The user of the signed applet can remove the default sandbox restrictions by indicating so on a dialog that is presented when the applet is first loaded in the browser.

A common misconception of most newcomers to Java is that security restrictions apply only to *applets* (Java classes downloaded and executed within a web browser). In fact, security restrictions can apply to *all* Java classes. (However, they do not apply to classes loaded from the boot *classpath*.) Before the Java API performs an action that is potentially unsafe, it calls the Java Security Manager (JSM) to determine whether the action is permitted. Here is a partial list of the actions for which checks take place:

- Reading, writing, or deleting a file
- Opening, waiting for, or accepting a socket connection
- Modifying a thread attribute (for example, priority)
- Accessing or updating system properties

If the Java Security Manager does not permit the action, the Java API will not allow the action to take place. Now, you might ask, how is my application able to do one of these so-called unsafe calls, such as read or write a file? The answer is that the Java Security Manager is not installed by default; but it can be by calling it within your class or specifying a parameter to the Java command line. To establish the Java

Security Manager within code, place the following as the first line in the `main()` method:

```
System.setSecurityManager( new SecurityManager() );
```

To establish the Java Security Manager via the command line, add the following parameter to the command line:

```
-Djava.security.manager
```

Once installed, the Java Security Manager checks whether a particular permission is granted to the specific requesting class; it throws a *SecurityException* if the permission is denied. The Java Security Manager checks by examining the call trace, so if an untrusted piece of code is invoked as part of a call to a secured method, it will fail because of the presence of the untrusted code. The permission is itself an abstract class representing access to a system resource. The permission can optionally contain a name and an action. When specified, these optional attributes further refine the permission being granted. For example, *java.io.SocketPermission* can be established with a host name of *66.108.43.211:9080* and an action of *accept,connect,listen,* which will allow the code to accept connections on, connect to, or listen on port 9080 on a host specified by IP address 66.108.43.211.

Here is a list of the security restrictions that Java technology environments normally impose on an unsigned applet running in a browser:

- Can make network connections only to the host from which it was downloaded.
- Can utilize only its own code and is not allowed to load libraries or define native methods.
- Cannot change thread priority.
- Cannot execute any native code.
- Cannot install software.
- Cannot issue an RMI call to a remote object running on a different server than the applet's.
- Cannot monitor mouse motion.
- Cannot programmatically read from or write to the clipboard.
- Cannot read or write local files on the host that is executing it.
- Cannot read the system properties specified in Table 10-3.
- Cannot send e-mail to a server other than the host from which it was downloaded.

TABLE 10-3 Security Packages of the Java Platform

Property	Description
java.home	Java installation directory
java.class.path	Java classpath
user.name	User account name
user.home	User home directory
user.dir	User's current working directory

- Cannot start any program on the local host.
- Cannot talk to a serial or parallel port.
- Cannot use `System.setOut()` or `System.setErr()` methods to redirect the console.
- Cannot use the Preferences API.
- Cannot use the Reflection API.

CERTIFICATION OBJECTIVE 10.02

Given an Architectural System Specification, Identify Appropriate Locations for Implementation of Specified Security Features and Select Suitable Technologies for Implementation of Those Features

We will now take a look at the authentication and authorization security features that are part of a distributed network environment. We'll look at specific implementations and provide example code for review.

Authentication

In distributed computing, *authentication* is the device used by callers and service providers to prove to one another that they are to be "trusted." When the proof is bidirectional, it is called *mutual* authentication. Authentication establishes an actor's identities and proves that each instance is "authenticated." An entity participating in a call without establishing an identity is "unauthenticated."

Authentication is achieved in phases. Initially, an *authentication context* is established by performing authentication, requiring knowledge of a secret password. The authentication context encapsulates the identity and is able to fabricate an *authenticator*—a proof of identity. The authentication context is then used to provide authentication to other entities with which it interacts. The utility of authentication context should be well planned by the enterprise security team. Of late, security and identity management has become a critical enterprise function. Most large enterprises now have an adjunct group responsible for maintaining a user identity throughout the enterprise environment. Some large enterprises will have thousands of applications, each with its own authentication and identity maintenance. Software such as Thor *(www.thortech.com)* is designed to maintain users, groups, and security policy that provisions authentication for all of the secured resources (programs and data) within an enterprise.

Potential policies for controlling access to an authentication context are listed here:

- Once the user performs an authentication, the processes the user invokes inherit access to the authentication context.
- When a component is authenticated, access to the authentication context may be available to other trusted components.
- When a component is expected to impersonate its caller, the caller delegates its authentication context to the called component.

The whole issue of *propagation* of authentication context from client to the Enterprise JavaBeans (EJB) server to the Enterprise Information System (EIS) server is still evolving, both in terms of the specification as well as vendor offerings. According to the current Java specification, the *container* is the authentication boundary between callers and components hosted by the container. To this end, JAAS is a package that enables services to authenticate and enforce access controls upon clients. It implements a Java version of the standard, pluggable authentication module framework and supports client-based authorization. JAAS was integrated into Java in version 1.4. The core facilities of Java's security design are intended to protect a client from developers. The client gives permissions to developers to access resources on the client machine. JAAS allows developers to grant or deny access to their programs based on the authentication credentials provided by the client. The JAAS specification extends the types of principals and credentials that can be associated with the client, but it is also evolving.

Java Protection Domains

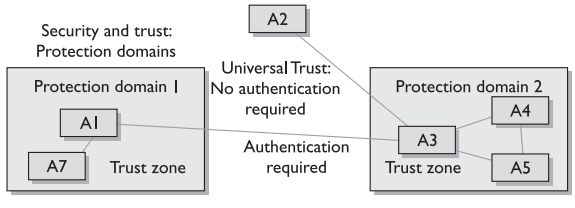
Some components communicate without the need for authentication. This is because the communication is based on a preestablished trust mechanism. A *protection domain* is the name given to a group of components that have this trust established. Components within the same protection domain do not need to be authenticated with each other, and consequently no constraint is placed on the identity associated during a call. Authentication is required only for components that interact across the boundary of the protection domain. Figure 10-1 illustrates the authentication requirements for interactions that are contained within and that cross the boundary of a protection domain.

The container in the JEE environment provides the authentication boundary between external and internal components. It is possible that the authentication boundary is not synchronized with the boundaries of protection domains. Even though it is the responsibility of the container to enforce the boundaries, you may encounter an environment that contains protection domains that span multiple containers. The issue of protection boundaries is extremely important, as enterprise requirements fuse JEE web and EJB components with back-end EIS resources that have preexisting application security in place—for example, IBM RACF (Resource Access Control Facility) security for CICS (Customer Information Control System) transactions interacting with JEE.

In general, it is the responsibility of the container to authenticate component calls and police the boundaries of the protection domain. On an inbound call, the container passes the authenticated credentials to the component being called. The credentials can be a simple identity or a more complicated item such as an X.509 certificate. Similarly, when a component makes an outbound call, the container is responsible for establishing the identity of the component making the call. When a call is made across containers and the identity of the calling component has not been authenticated, the containers will check whether an existing trust exists between the interacting containers. If the trust exists, the call is permitted; if not, the call is rejected.

FIGURE 10-1

The Java protection domain



It is important that you differentiate the identity "propagation" model from the "delegation/impersonation" model. In the propagation model, the providers must determine whether or not to accept propagated identities as authentic. In delegation/propagation, the called component is given access to the caller's authentication context, thus enabling the called component to use the passed credentials to act on behalf of (or to impersonate) the caller.

Authentication in JEE

In a JEE application, the user's client container communicates with containers in the web, EJB, or EIS layers to access resources in their respective zones. These resources can be either protected or unprotected. A resource is protected when an authorization constraint is defined to the container that hosts it. When a user wishes to access a protected resource, the client container must present credentials (along with, when no trust exists between the containers, an authenticator that proves the caller has a right to use the identity) so that the container can validate them against the defined authorization constraint and then either permit or deny access to the resource.

Authentication in the Web Container

Collections of web resources, such as JSPs, servlets, HTML documents, image files, and compressed archives, are protected in the JEE environment when the deployer specifies one or more authorization constraints for the collection at deployment time. In the deployed or target environment, when a user of a browser attempts to access protected resources, the web container determines whether the user has been authenticated; if not, the user will be prompted to identify himself using the mechanism specified in the application's deployment descriptor. When the user is successfully authenticated, he still will not be able to access the resource unless his identity is one that is granted permission according to the authorization constraint.

As already mentioned, the deployer specifies the authentication mechanism in the application deployment descriptor. A JEE web container typically supports the following types of authentication mechanisms:

- HTTP basic
- HTTP digest
- FORM based
- HTTPS mutual

In HTTP basic authentication, the web server authenticates a principal using the user name and password obtained from the web client. The following process shows the conversation between the client browser and the web container to help elaborate on the basic authentication mechanism.

1. Client browser attempts to access a protected resource by sending an HTTP GET request—for example:

   ```
   GET /secure/declarative.html HTTP/1.1 Host: ucny.com
   ```

2. The web container sends back a challenge to the client to authenticate. The `WWW-Authenticate` header within the response contains the type of the authentication mechanism required and the security realm:

   ```
   HTTP/1.1 401 Unauthorized WWW-Authenticate: Basic
   realm="weblogic"
   ```

3. The user enters a user ID and a password for the security realm, and the request is resubmitted along with an additional HTTP header whose value contains the authentication mechanism, the security realm, and the credentials. The credentials are formed by concatenating the user ID, a colon, and the password and then encoding this using the base-64 encoding algorithm. The following HTTP GET request contains the base-64 encoded credentials:

   ```
   GET /secure/declarative.html HTTP/1.1 Host: ucny.com
   Authorization: Basic c3lzdGVtOnBhc3N3b3Jk
   ```

4. The server will then attempt to authenticate the credentials within the security realm. If unsuccessful, the server will prompt again for valid credentials. If the credentials are valid, the identity will be checked against the authorization constraint. If the identity is permitted, access to the resource is allowed; otherwise, it is denied.

Basic authentication is limited, because HTTP is a *stateless* protocol. Once authenticated, a browser has to send this authentication data along with each and every client request. This is clearly a security threat because the request is not encrypted and can be captured and then retransmitted by a determined unauthorized individual. What's more, base-64 encoding is simple to decode and gives the hacker a real user ID and password that can be used to gain access to other protected resources. This potentially opens up the enterprise to the threat based upon a "compromise of accountability." For these reasons, it is pragmatic to use basic authentication with an encrypted link and server authentication, more commonly known as *digest authentication*.

Digest authentication is an improvement over basic authentication because it allows the client to prove knowledge of a password without actually transmitting it across the network. The web client authenticates by sending the server a message digest as part of the HTTP request. This message digest is calculated by taking parts of the message along with the client's password and passing them though a one-way hash algorithm. The mechanism works similarly to basic authentication, but in this case, the web container sends back some additional data with the challenge to the client to authenticate.

```
HTTP/1.1 401 Unauthorized WWW-Authenticate: Digest realm="ucny",
qop="auth", nonce="7fef9f6789b0526151d6efbd12196cdc",
opaque="c8202b69f571bdf3eerft43ce6ee2466"
```

The `WWW-Authenticate` header contains the name of the authentication mechanism (`Digest`), the realm (`"ucny"`), and some additional parameters to authenticate. These additional parameters include the `nonce`, or number once, which is a value that is used by the server and is valid for the current authentication sequence only. The browser client must then take the user name, password, realm, nonce, HTTP method, and request Uniform Resource Identifier (URI) and calculate a digest. The digest, a fixed-length encoding, has the properties that hide the actual data. The client will then resubmit the HTTP request along with a response parameter that is the calculated digest:

```
GET /secure/declarative.html HTTP/1.1 Host: ucny.com
Authorization: Digest username="system", realm="weblogic",
qop="auth", nonce="7fef9f6789b0526151d6efbd12196cdc",
opaque="c8202b69f571bdf3eerft43ce6ee2466",
response="5773a30ebe9e6ce90bcb5a535b4dc417"
```

The server in turn calculates the message digest from the inbound request and then compares it to the response value. If the values are not equal, the server responds with a "401 Unauthorized" error. If the values are equal, the credentials are deemed valid and then subsequently used for the authorization check to determine whether the client should have access to the protected resource. If the user is authorized, access to the resource is granted. If the authorization step fails, the server responds with a "403 Access Denied" error.

Form-based authentication allows for the use of a custom HTML form as the user interface for capturing the authentication information. However, as in basic authentication, the target server is not authenticated, and the authentication information is transmitted as plain text and as such is still vulnerable.

With mutual authentication, X.509 certificates are used to establish their identity on the client and on the server. The transmission occurs over a secure channel (SSL) and is much more difficult for a hacker to break into.

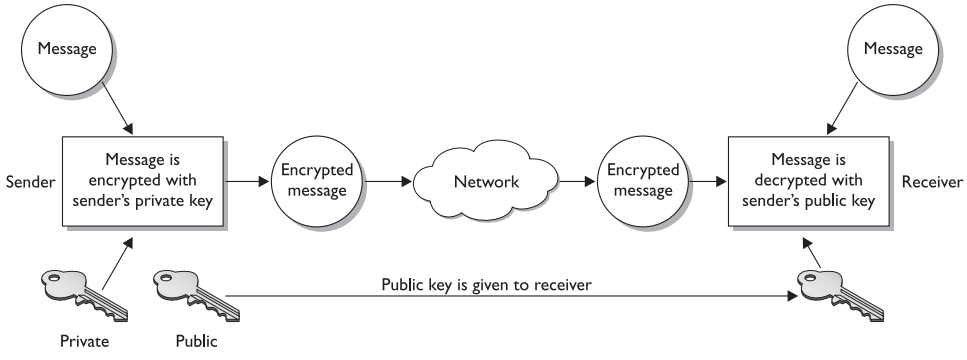
Encrypted Communication

Cryptography is a mechanism whereby data is encrypted using a key such that it can be decrypted only with a matching key. The two types of encryption are known as *symmetric* and *asymmetric*. In symmetric encryption, both sender and recipient know a common key, and this is used to encrypt and decrypt messages. In asymmetric encryption, also known as *public-key cryptography*, a key is split into two parts and referred to as a *key pair*, or *private* key and *public* key. Their most interesting feature is that each key is able to decrypt data that was encrypted by the other. The private key is obviously kept private and known only to a single individual or business, and the public key is given to all those who wish to communicate securely back and forth with the private key holder. So the private key holder is the only one who can decrypt data encrypted by the public key holders, and the public key holders are the only ones who can decrypt data encrypted by the private key holder. Figure 10-2 shows how asymmetric cryptography works.

Several choices can be made regarding which type of encryption to use and how much data should be encrypted in any given communication. For example, all the data can be encrypted with a private key so that only the public key holder can decrypt it, or it can be encrypted using a symmetric key known to both sides. Another possibility is to append an encrypted piece on to the communication—in effect, a signature or seal—so that the recipient will know that the sender genuinely sent the data and that the data was not tampered with on the way. In this case, the sender produces a hash code result by executing an algorithm on the complete message.

FIGURE 10-2

Asymmetric cryptography



This hash code result then gets encrypted and appended with the original data. Once the message is received, the recipient will attempt to decrypt the encrypted portion of the message to obtain the sender's hash code result. If successful, the recipient knows the message came from the sender. The recipient then executes an algorithm on the complete message, producing a hash code result to be compared with the sender's hash code. If they are the same, the message has been received without any tampering along the way.

Asymmetric encryption is slower than symmetric encryption when dealing with large amounts of data. This is due in part to the increased length of the keys required in asymmetric cryptography to achieve the same level of protection as the symmetric variety. The longer keys demand more computing resources. Because of this, the bulk of data that needs to be secured is usually encrypted using symmetric cryptography, and a smaller amount is encrypted with asymmetric cryptography. In fact, a large number of hardware manufacturers sell SSL accelerator boards to avoid the overhead of key generation, encryption, and decryption.

Digital Certificates

A big issue with public key (asymmetric) cryptography is the way that recipients obtain the public key. If a public key is received in person, it can be deemed trustworthy. However, if it is received via some other means, it may not be deemed trustworthy. To alleviate the logistical issue of all public keys being handed out in person, a popular solution is for the private key holder to place the public key into a package known as a *digital certificate*, and then sign it with the private key of a trusted authority, known as a *Certificate Authority (CA)*. These digital certificates can then be sent securely to recipients that are willing to trust the same CA and have access to the CAs public key. Several CAs are universally trusted, and their public keys are well known. VeriSign *(www.verisign.com)* is a leader in issuing digital certificates globally. The VeriSign public keys get distributed with commercial client and server software such as browsers and web servers.

Secure Sockets Layer

Created by Netscape, SSL is a security protocol that ensures privacy on the Internet. The protocol allows applications to communicate in such a way that eavesdropping cannot easily occur. SSL offers data encryption, server authentication, and message integrity. Servers are authenticated for each request, and clients can be optionally authenticated as well. SSL is independent of the application protocol, and as such, protocols such as HTTP or FTP can transparently execute on top of it. All the data is encrypted before it is transmitted.

SSL uses a series of *handshakes* to establish trust between two entities using asymmetric encryption and digital certificates. These handshakes finish with the two entities negotiating a code set, including a set of session keys to be used for bulk encryption and data integrity. SSL has two authentication modes, *mutual* and *server*. In mutual authentication mode, the client and server exchange digital certificates to verify identities. In server authentication mode, the server sends a certificate to the client to establish the identity of the server. Note that HTTPS (HTTP running over SSL) typically uses port 443 instead of HTTPs default port 80. A digital certificate, provided by a CA such as VeriSign, must be installed on the server before server authentication can take place.

Authentication Within the EJB Container

Although EJB containers implement authentication mechanisms, they often rely on the web container to have authenticated a user already. The web container enforces the protection domain for web components and the EJBs that they call. A typical configuration is shown in Figure 10-3.

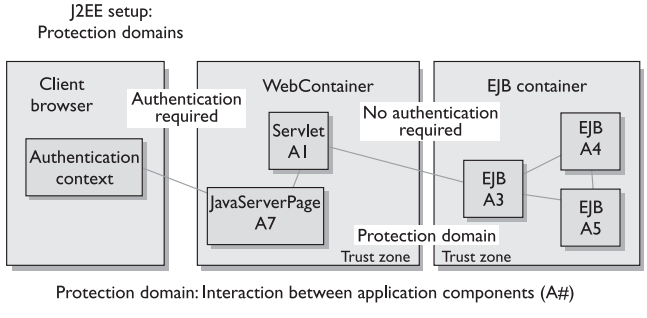
EJB containers and EJB client containers support version 2 of the Object Management Group's (OMG) Common Secure Interoperability (CSI) protocol. CSIv2 is a standard wire protocol for securing calls over the Internet Inter-ORB Protocol (IIOP). At its core, CSIv2 is an impersonation or "identity assertion" mechanism. It provides the mechanism for an intermediate entity to impersonate, or assert, an identity other than its own. This feature is based on the fact that the target trusts the intermediate entity.

Here is a summary of CSIv2:

- An Interoperable Object Reference (IOR) holds a component that identifies the security mechanisms supported by the object. The IOR also includes information about the transport, client authentication, and identity and authorization tokens.

FIGURE 10-3

Typical J2EE application configuration



- The security mechanisms contained within the IOR are examined and the mechanism that supports the options required by the client is selected.
- The client uses CORBAs Security Attribute Service (SAS) protocol to set up a security context. This protocol defines messages contained within the service context of requests and replies. The security context established can be either stateful (used for multiple invocation) or stateless (used for a single invocation).
- CSIv2 supports Generic Security Services API (GSSAPI) initial context tokens, but to comply with conformance level 0, only the user name and password must be supported.

CSIv2 is intended for situations where protecting the integrity of data and the authentication of clients are carried out at the transport layer level, along with, for example, SSL. JEE containers use CSIv2 impersonation to assert the identity of a caller to a component. Figure 10-4 illustrates the CSIv2 architecture components and the enterprise beans that they invoke.

When a JEE application is deployed to an application server, the deployer defines the CSIv2 security policy to be enforced by the application server. This includes specifying certain security requirements, such as the following:

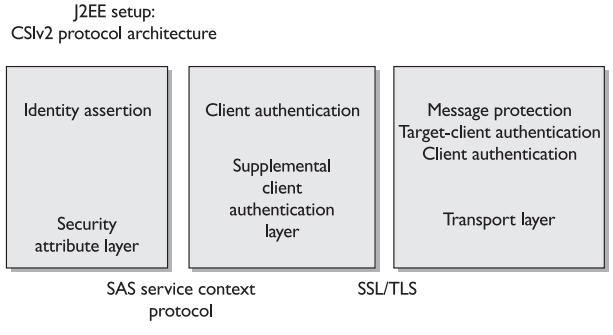
- Does the target require a protected transport?
- Does the target require client authentication?
- If so, what type of authentication is required?

Configuring Authentication in JEE

As part of the web application's authentication mechanism, the servlet container uses the *login-config* element contained in the application deployment descriptor

FIGURE 10-4

CSIv2 protocol architecture



file *(web.xml)*. The container performs many checks, some of the most common of which are listed here:

- Does the request need to be decrypted?
- Does the request have authorization constraints?
- Does the request have authentication or authorization requirements?

Typically, resources in an application are protected via a combination of authentication and authorization constraints. Specifying authentication constraints without authorization constraints (and vice versa) does not add any value and therefore does not make sense.

If the container cannot determine the caller's identity, it uses the *<login-config>* element specified in the application deployment descriptor. The following code listing example is an excerpt from an application deployment descriptor. It contains the authentication entries required to enforce the constraint using FORM-based authentication. Note that the associated authorization constraints, specified within the web-resource-collection element, are covered in detail in the section "Authorization" found later in this chapter.

```
<web-app>
  <security-constraint>
    <web-resource-collection>
      ...(resources to be protected)
    </web-resource-collection>
    <auth-constraint>
      <role-name>secure_role</role-name>
    </auth-constraint>
    <user-data-constraint>
      <transport-guarantee>NONE</transport-guarantee>
    </user-data-constraint>
  </security-constraint>
  <login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
      <form-login-page>login.html</form-login-page>
      <form-error-page>loginError.html</form-error-page>
    </form-login-config>
  </login-config>
  <security-role>
    <role-name>secure_role</role-name>
  </security-role>
</web-app>
```

Here is the <*login-config*> excerpt that uses BASIC authentication. Note that the realm-name is used only in BASIC authentication.

```
...
   <login-config>
   <auth-method>BASIC</auth-method>
   <realm-name>weblogic</realm-name>
</login-config>
...
```

Here is the <*login-config*> excerpt that uses DIGEST authentication:

```
...
<login-config>
   <auth-method>DIGEST</auth-method>
</login-config>
... Here is the <login-config> excerpt that uses CLIENT-CERT authentication.

...
<login-config>
   <auth-method>CLIENT-CERT</auth-method>
</login-config>
...
```

With BASIC authentication, the browser displays and controls the login process and user interface. The browser will display a simple dialog box prompting for user name and password. With FORM authentication, the web application defines and therefore controls the login process to a greater extent. Here is some example code for a login form:

```
<head><title>Security Demo: login</title></head>
<h2>Login</h2>
Please authenticate yourself:
<form method="POST" action="j_security_check">
Username: <input type="text" name="j_username"><br />
Password: <input type="password" name="j_password"><br />
<br />
<input type="submit" value="Login">
<input type="reset" value="Reset">
</form>
<p><a href="index.html">home</a>
</html>
```

Here is the code for *loginError.html:*

```
<head><title>Security Demo: login error</title></head>
<h2>Login Error</h2>
<hr width="100%">
Invalid username/password.
<br />
<p><a href="index.html">home</a>
</body>
</html>
```

In FORM-based authentication, the web container performs the authentication check. It does so according to the servlet specification, which specifies that the form method must be a POST, the name of the action must be *j_security_check,* and the names of the user name and password fields must be *j_username* and *j_password,* respectively. When the container sees the *j_security_check* action, it uses an internal mechanism to authenticate the caller. If the logon is authenticated and authorized to access the secured resource, the container produces a session ID to identify a logon session for the caller. The container maintains the logon session ID within a cookie. The server sends the cookie back to the client, and the client caller must then send this cookie back on all subsequent requests. If the authentication fails, the page identified by the `<form-error-page>` is returned to the client.

As mentioned, FORM-based authentication is still not secure by default. But it can be made more secure by conducting it over a secure channel by specifying a transport guarantee for the secured resource. For example, use `<transport-guarantee>SSL</transport-guarantee>`.

Authentication in the Enterprise Information System Layer

When JEE components need to access and therefore integrate with EISs, they may need to employ alternative mechanisms for security. In addition, they most likely will be operating from protected domains that do not cover the EIS resources they need to access. To provide for these situations, the calling container can be set up to manage the calling component's authentication for the resource. This is known as *container-managed resource manager sign-on.* The JEE architecture also provides the ability to specify the caller's credentials. This is known as *application-managed resource manager sign-on.*

Within the deployment descriptor, the *<resource-ref>* element specifies a resource called by a component. The *<res-auth>* element specifies whether the resource sign-on is to be handled by the container or the application. Components that use application-managed resource manager sign-on can use either the `getUserPrincipal()`

(for web components) or `getCallerPrincipal()` (for EJB components) method to access the identity of the caller. This identity can then be mapped according to the requirements of the EIS. When container-managed resource manager sign-on is used, the container takes care of the mapping for the component.

Identity Selection

In a JEE server-side component, the container sets up the identity when the component calls another JEE component. The identity that is created is dependent on the identity selection policy specified in the deployment descriptor. For the identity selection policy, the deployer can specify either a *<use-caller-identity>* element or a *<run-as>* element. Component identity selection policies may be defined for web and EJB resources. When *<use-caller-identity>* is specified, the container uses the identity of a component's caller in all subsequent calls made by the component. When the *<run-as>* element is specified, the container uses the identity specified within the element. In short, *<use-caller-identity>* maintains accountability and traceability for actions taken by components, and *<run-as>* can quickly give the caller privileges that their own identity lacks.

The following EJB deployment descriptor snippet shows examples of both types of client identity selection policy:

```
//Configuring EJB Component Identity Selection Policies
<enterprise-beans>
  <entity>
    <security-identity>
      <use-caller-identity/>
    </security-identity>
    ...
  </entity>
  <session>
    <security-identity>
      <run-as>
        <role-name>guest</role-name>
      </run-as>
    </security-identity>
    ...
  </session>
  ...
</enterprise-beans>
```

The following deployment descriptor snippet shows an example of client identity selection policy for a web component. Note that when a *<run-as>* element is not specified, the *use-caller-identity* policy is assumed.

```
//Configuring Web Component Identity Selection Policies
<web-app>
  <servlet>
    <run-as>
      <role-name>guest</role-name>
    </run-as>
    ...
  </servlet>
  ...
</web-app>
```

Authorization

Authorization is the mechanism that controls caller access and interaction with application resources or components. The caller's credentials (identity), which can also be anonymous or arbitrarily set by the caller, can be determined via authentication contexts that are available to the called component. Access can then be determined by comparing the caller's credentials with the access control rules for the required component or resource.

These access control rules are in effect a matching of the application's capabilities with the caller's permissions. The application's capabilities define what can be performed within the application, and the caller's permissions define what the caller is allowed to perform.

In the JEE architecture, the container provides the "border patrol" between callers requiring access to the target resources and components that execute within the container. So on an inbound call, the container compares the caller's credentials with the access control rules for the target component or resource. If the rules are satisfied, the call will continue; if not, the call is rejected.

Authorization in the JEE environment can be enforced in two ways: *declaratively*, configured by the deployer and managed by the container, or *programmatically*, embedded in and managed by the component.

Declarative authorization controls access from outside of the application code, whereas programmatic authorization controls access from within the application code. The pros and cons for each technique are detailed in Table 10-4.

The client to a JEE application typically uses the application container to interact with enterprise resources in the web or EJB layer. Resources that are secured (or protected) have authorization rules that are either declared in deployment descriptors or embedded within component code. These rules control the access to the components, and clients will need to present credentials to be evaluated against the access rules that are in place.

TABLE 10-4 Pros and Cons for Declarative and Programmatic Authorizations

Technique	Pros	Cons
Declarative authorization (external)	Continued flexibility once application is developed. Easily viewed and interpreted by deployer.	May not provide enough fine-grained flexibility.
Programmatic authorization (internal)	Provides fine-grained flexibility.	No flexibility after application is developed. Functionality is buried within code.

Authorization Enforced by the Container (Declarative)

As mentioned, declarative authorization is established externally to the web or EJB component. It is defined within the deployment descriptor files. Entries within these files map the application permissions (usually defined by the assembler) to the policies or mechanisms that exist in the actual target environment.

The *deployment descriptor* file contains definitions that associate the security roles (logical privileges) with components and the privileges required for permission to access components. The deployer assigns security roles to specific callers, thus establishing the abilities of users in the target environment.

Using Declarative Authorization

A client typically uses a JEE application's container to access enterprise resources in the web or EJB tier. To control access to a web resource declaratively, an application component provider or application assembler must specify the *security-constraint* element along with the *auth-constraint* subelement in the application deployment descriptor. The following deployment descriptor excerpt shows the specification of a protected web resource:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>SecurePages</web-resource-name>
    <description>Security constraint for protected resources</description>
    <url-pattern>/secure/*</url-pattern>
    <http-method>POST</http-method>
    <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
    <description>Users in this role can login</description>
    <role-name>secure_role</role-name>
  </auth-constraint>
  ...
</security-constraint>
```

This excerpt indicates to the container that the URL conforming to the pattern */secure/** can be accessed only by users that are in the *secure_role* role. However, some web content typically does not need to be protected with authorization rules. This unrestricted access is achieved simply by not adding an authentication rule.

To protect or declaratively control access to an enterprise bean resource, the application component provider or application assembler can declare security roles and the methods of the bean's interfaces (remote, home, local, and local home) that each security role is allowed to call. This is declared using *method-permission* elements in the deployment descriptor.

The following deployment descriptor excerpt shows two *method-permission* elements. The first refers to *method2* of all of the interfaces (which could be remote, home, local remote, and local home) of the enterprise bean. The second refers to *method3* on the remote interface of the same enterprise bean.

```
<assembly-descriptor>
  ...
  <security-role>
    <role-name>usr_role</role-name>
  </security-role>
  <security-role>
    <role-name>adm_role</role-name>
  </security-role>

  ...
  <method-permission>
    <description>remote method2 access</description>
    <role-name>usr_role</role-name>
    <method>
      <ejb-name>DeclarativeSecurity</ejb-name>
      <method-name>method2</method-name>
    </method>
  </method-permission>
  <method-permission>
    <description>remote method3 access</description>
    <role-name>adm_role</role-name>
    <method>
      <ejb-name>DeclarativeSecurity</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>method3</method-name>
    </method>
  </method-permission>

  ...
</assembly-descriptor>
```

Note that if another method were to use the same name (that is, overloaded methods were in the bean code), this permission scope would refer to both methods. You can refine the scope further by identifying methods with overloaded names by parameter signature, or you can refer to methods of a specific interface of the enterprise bean (such as remote, local).

You can also indicate to the container that it should allow the call to a method to proceed regardless of the caller's identity. By adding the unchecked element to the *method-permission* element, the container authorizes the use of a method to anybody. Here is a deployment descriptor excerpt showing the unchecked element:

```
<assembly-descriptor>
  ...
  <method-permission>
    <unchecked/>
    <method>
      <ejb-name>DeclarativeSecurity</ejb-name>
      <method-name>method1</method-name>
    </method>
  </method-permission>
  ...
</assembly-descriptor>
```

Additionally, method specifications may be added to an *exclude-list*. This indicates to the container that access to these methods is denied regardless of the caller's identity, even if the methods have been specified in the *method-permission* element. Here is a deployment descriptor excerpt showing the *exclude-list* element:

```
<assembly-descriptor>
  ...
  <exclude-list>
    <method>
      <ejb-name>DeclarativeSecurity</ejb-name>
      <method-name>method4</method-name>
    </method>
  </exclude-list>
  ...
</assembly-descriptor>
```

Authorization Enforced by the Component (Programmatic)

There may be a time when declarative authorization is not sufficient. For example, if a more fine-grained authorization model is required, the developer of a web component can use a combination of the `getUserPrincipal()` and `isUserInRole()`

methods that exist on the `HttpServletRequest` object, and the developer of an EJB component can use a combination of the `getCallerPrincipal()` and `isCallerInRole()` methods that exist on the EJBs `context` object, to carry out access control at the component level. This is known as *programmatic* authorization. The web or EJB component can use these methods to determine whether the caller is allowed to perform the functionality that has been called within the component.

Using Programmatic Authorization

As mentioned, the web component developer will use `getUserPrincipal()` and `isUserInRole()` methods within a JSP or servlet to control access to the web resource's functionality. These methods typically require that the client also be authenticated, so it makes sense to use the technique in conjunction with declarative authorization.

The following example shows the use of the programmatic authorization methods `getUserPrincipal()` and `isUserInRole()` in a JSP. The JSP is part of an application that also has some resources protected with declarative authorization. Here is example code for a JSP:

```
<html>
<head><title>Security Demo</title></head>
<body bgcolor="#FFFFFF">
<head><title>Security Demo: programmatically protected page</title></head>
<h2>Programmatically Protected Page</h2>
<hr width="100%">
<%
  java.security.Principal principal = request.getUserPrincipal();
  if (principal != null) {
   boolean inRole = request.isUserInRole("secure_role");
   if (inRole) {
     if ("system".equals(principal.getName())) {
       out.write("<br>You are the correct user ("+principal.getName()
         +") and role, so access is granted!");
       // This is where code for the protected
       // functionality would reside...
     } else {
       out.write("<br>You are NOT the correct user ("+principal.getName()
         +"), so access is denied!");
     }
   } else {
     out.write("<br>You are NOT in the correct role, so access is denied!");
   }
```

```
  } else {
    out.write("<p>You are not authenticated, so access is denied!");
  }
%>
<p><a href="../index.html">home</a>
</body>
</html>
```

To control access programmatically to an enterprise bean resource, the enterprise bean provider uses the `isCallerInRole()` and `getCallerPrincipal()` methods to determine whether the caller is within the specified role or is, in fact, a specific user that is authorized to perform the called functionality. Within the EJB deployment descriptor, the assembler must add a *security-role-ref* element for every role that is referred to within the bean code. The assembler must also add a *security-role* element for the *role-link* in every *security-role-ref* element that has been added. Here is an excerpt for an EJB deployment descriptor:

```
<ejb-jar>
   ...
   <enterprise-beans>
     ...
     <ejb-name>ProgrammaticSecurity</ejb-name>
     ...
     <security-role-ref>
       <role-name>programmatic</role-name>
       <role-link>programmatic_role</role-link>
     </security-role-ref>
     ...
   </enterprise-beans>
   ...
   <assembly-descriptor>
     <security-role>
       <role-name>programmatic_role</role-name>
     </security-role>
     ...
   </assembly-descriptor>
   ...
</ejb-jar>
```

When deployed, each *role-name* specified in the *assembly-descriptor* element of the EJB deployment descriptor must be mapped to actual resources in the target server. The following excerpt is from a WebLogic server deployment descriptor that resolves the *role-name* to specific resources within the server:

```
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>ProgrammaticSecurity</ejb-name>
    ...
  </weblogic-enterprise-bean>
  <security-role-assignment>
    <role-name>programmatic_role</role-name>
    <principal-name>system</principal-name>
    <principal-name>auser</principal-name>
  </security-role-assignment>
</weblogic-ejb-jar>
```

In this excerpt, the *security-role-assignment* declares and resolves the *role-name (programmatic_role)*, specified within the *assembly-descriptor* element of the EJB deployment descriptor, to one or more principal identities (*system* and *auser*) that are known to an authentication realm within the WebLogic server.

Here is an example of enterprise bean code that programmatically determines whether the caller is permitted to execute the method:

```
package javaee.architect.ProgrammaticSecurity;
import javax.ejb.*;
// A stateless session bean.
public class ProgrammaticSecurityBean implements SessionBean {
  SessionContext sessionContext;
  private static final String ROLE_REQUIRED = "programmatic";
  // Bean's methods required by EJB specification.
  public void ejbCreate() throws CreateException {log("ejbCreate()");}
  public void ejbRemove() {log("ejbRemove()");}
  public void ejbActivate() {log("ejbActivate()");}
  public void ejbPassivate() {log("ejbPassivate()");}
  public void setSessionContext(SessionContext parm) {
    this.sessionContext = parm;
  }
  // Bean's business methods.
  public String method1() {
    log("method1() called by user "
      +sessionContext.getCallerPrincipal().getName());
    return " method1 executed.";
  }
  public String method2() throws EJBException {
    log("method2() called by user "
      +sessionContext.getCallerPrincipal().getName());
    if (!sessionContext.isCallerInRole(ROLE_REQUIRED))
      throw new EJBException ("insufficient permission to access method2");
```

```
    // Place method functionality here...
    return " method2 executed.";
  }
  private void log(String parm) {
    System.out.println(new java.util.Date()
      +":ProgrammaticSecurityBean:"+this.hashCode()+" "+parm);
  }
}
```

Authorization Summary

By defining a clear separation of the responsibilities for securing an application among those that develop components, those that assemble components, and those that deploy application components, the JEE platform achieves its goal of making the details of security much more simple and easy to implement.

The *component-provider* role identifies all the security dependencies embedded in the component, including the following:

- The role names used in method `isUserInRole()` for web components and `isCallerInRole()` for EJB components
- References made by the component to other components
- References to external resources accessed by the component
- The method permission model, including information that identifies the sensitivity of the information exchanged or processing that occurs in individual methods

The *application-assembler* role combines one or more components into an application package and then produces an overall security view for the whole application. The deployer role takes this overall security view and uses it to secure the application for target environment. The deployer does this by mapping the security view elements to the actual policies and mechanisms that exist in the target environment. How this mapping occurs will depend on the vendor for the web and EJB containers. In some cases, additional deployment descriptors resolve this mapping, and in other cases a vendor-specific tool must be used.

Java 5 Annotation Facility

With the arrival of Java 5, one of the most significant changes in authorization is in the use of annotations. It is now possible to define custom annotations and

then apply these annotations to fields, methods, classes, etc. Annotations do not directly affect program semantics, but compile or runtime tools can examine them to generate additional constructs (for example, deployment descriptors) or implement required runtime behavior (for example, EJB component's state).

Using Java 5 Annotations with Web components With Java version 5 and beyond, in addition to the declarative and programmatic security specifications mentioned already, application developers can now programmatically set up the security for web components by using Java's annotation facility.

If a value is specified in an annotation and also in the deployment descriptor, the value in the deployment descriptor takes precedence. The granularity of overriding is on the per-method basis.

In addition, the web-app element of the web application deployment descriptor can now contain a `full` attribute. This `full` attribute states whether deployment descriptor is complete, or whether the class files of the web archive (WAR) file should be examined for annotations that specify deployment information. If the `full` attribute is missing or set to `false`, the deployment descriptors examine the class files of applications for annotations that specify deployment information. If the `full` attribute is set to `true`, the deployment descriptor ignores any servlet annotations present in the class files. This feature allows the application deployer to customize or override the values specified in annotations.

Web application class files can specify the following annotations:

- Declare roles for the EJB module using the `@DeclareRoles` annotation.
- Set the security identity using the `@RunAs` annotation.

Using Java 5 Annotations with EJB components One of the goals in EJB 3.0 is to reduce the number of pieces that a bean provider must provide. In the EJB 3.0 world, all kinds of enterprise beans are plain old Java objects (POJO) with appropriate annotations. Annotations can be used to define the bean's business interface, Object/Relational (O/R) mapping information, resource references, and practically anything else that was previously defined by deployment descriptors or interfaces in the prior EJB specifications. Consequently, deployment descriptors are no longer required, the home interface is no more, and you don't necessarily have to implement a business (remote) interface because the container can generate it for you.

Applications that have been written using the EJB 3.0 specification can specify the following authorization security options with the annotation facility:

- Declare roles for the EJB module using `@SecurityRoles` annotation.
- Assign roles to the bean class and/or bean methods with `@MethodPermissions` annotation.
- Override bean role permissions to allow unrestricted access with `@Unchecked` annotation.
- Disable access to a business method with `@Exclude` annotation.
- Set the security identity of a bean with `@RunAs` annotation.

CERTIFICATION SUMMARY

The security mechanisms covered in this chapter show how the features of Java Platform, Enterprise Edition provide a robust solution for interoperable and distributed security.

TWO-MINUTE DRILL

Here are some of the key points from each certification objective in Chapter 10.

Identify Security Restrictions That Java Technology Environments Normally Impose on Applets Running in a Browser

- ❑ An unsigned applet can make network connections only to the host from which it was downloaded.
- ❑ An unsigned applet can utilize only its own code and is not allowed to load libraries or define native methods.
- ❑ An unsigned applet cannot change thread priority.
- ❑ An unsigned applet cannot execute any native code.
- ❑ An unsigned applet cannot install software.
- ❑ An unsigned applet cannot issue an RMI call to a remote object running on a different server than the applet's.
- ❑ An unsigned applet cannot monitor mouse motion.
- ❑ An unsigned applet cannot programmatically read from or write to the clipboard.
- ❑ An unsigned applet cannot read or write local files on the host that is executing it.
- ❑ An unsigned applet cannot read the following system properties: *java.home*, *java.class.path*, *user.name*, *user.home*, and *user.dir*.
- ❑ An unsigned applet cannot send e-mail to a server other than the host from which it was downloaded.
- ❑ An unsigned applet cannot start any program on the local host.
- ❑ An unsigned applet cannot talk to a serial or parallel port.
- ❑ An unsigned applet cannot use the `System.setOut()` or `System.setErr()` method to redirect the console.
- ❑ An unsigned applet cannot use the Preferences API.
- ❑ An unsigned applet cannot use the Reflection API.

Given an Architectural System Specification, Identify Appropriate Locations for Implementation of Specified Security Features and Select Suitable Technologies for Implementation of Those Features

- Authentication
  - Authentication method: BASIC, FORM, DIGEST, and CLIENT-CERT
  - Digital certificates, certificate authorities
  - Secure Sockets Layer (SSL)
  - Common Secure Interoperability (CSIv2)
  - Identity selection: *<run-as>* or *<use-caller-identity>* or *@RunAs* annotation
  - Security roles, including *@DeclareRoles* and *@SecurityRoles* annotations
- Authorization
  - Authorization enforced by the container (declarative), defined in the deployment descriptor and/or within the component itself via annotations
  - Authorization enforced by the component (programmatic), defined within the application code

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all the choices carefully because there may be more than one correct answer. Choose all correct answers for each question.

Identify Security Restrictions That Java Technology Environments Normally Impose on Applets Running in a Browser

1. Which of the following properties cannot be read by an unsigned applet?
   A. *os.name*
   B. *file.separator*
   C. *java.home*
   D. *java.version*
2. Which of the following is possible with an unsigned applet?
   A. Connect to the host from which it was downloaded.
   B. Draw a button.
   C. Load a library.
   D. Read from the clipboard.
3. A company is building an application that allows its sales force to access and process sales information via a web browser. As part of the application, the plan is to develop an applet that will upload data read from a directory on the salesperson's machine. What are your recommendations on the use of applets for this purpose?
   A. The Applet technology is not a viable solution because it is not allowed to access local resources.
   B. The Applet technology is a viable solution provided that it is packaged as a signed Applet and the salesperson explicitly allows (trusts) the Applet to be run in its signed state.

Given an Architectural System Specification, Identify Appropriate Locations for Implementation of Specified Security Features and Select Suitable Technologies for Implementation of Those Features

4. What is a message digest?
   A. A digital fingerprint value that is computed from a message, file, or byte stream
   B. A shortened summary of a message

C. The subject line of a message
D. A processing function of the mail server

5. What method can be used to help programmatically determine the caller's identity within enterprise bean code?
   A. `getIdentity()`
   B. `getCallerPrincipal()`
   C. `getCallerIdentity()`
   D. `getUserId()`
6. Within enterprise bean code, what method can be used to determine whether the caller is in a security role and authorized to execute the method?
   A. `inRole()`
   B. `isAuthorized()`
   C. `isCallerInRole()`
   D. `isValid()`
7. What method can be used to help programmatically determine the caller's identity within a JSP?
   A. `getUserPrincipal()`
   B. `getPrincipal()`
   C. `getUser()`
   D. `getIdentity()`
8. Within a JSP, what method can be used to determine whether the caller is programmatically authorized to execute its functionality?
   A. `inRole()`
   B. `okToExecute()`
   C. `isValid()`
   D. `isUserInRole()`
9. What role maps the declarative authorization rules to the target environment?
   A. Deployer
   B. Component provider
   C. Application assembler
   D. Authorizer
10. What role maps the programmatic authorization rules to the target environment?
   A. Application assembler
   B. Component provider

C. Coder
D. Deployer

11. For Enterprise JavaBeans (EJBs), where are the declarative authorization rules defined?
A. Application properties
B. EJB deployment descriptor
C. JNDI
D. Enterprise bean code

12. For web resources, where are the declarative authorization rules defined?
A. EJB deployment descriptor
B. Application deployment descriptor
C. In the web resource
D. JMS

13. For Enterprise JavaBeans (EJBs), where are the programmatic authorization rules implemented?
A. JNDI
B. EJB deployment descriptor
C. In the enterprise bean code
D. JMS

14. For web resources, where are the programmatic authorization rules defined?
A. Java Security Manager
B. Security policy file
C. JNDI
D. Within the JSP or servlet

15. Which of the following is not a valid authentication method (`auth-method`)?
A. FORM
B. HTTP
C. DIGEST
D. CLIENT-CERT

SELF TEST ANSWERS

Identify Security Restrictions That Java Technology Environments Normally Impose on Applets Running in a Browser

1. ☑ **C** is correct. An unsigned applet cannot read the *java.home* system property.
   ☒ **A, B,** and **D** are incorrect. *os.name*, *file.separator*, and *java.version* can be read by an unsigned applet.
2. ☑ **B** is correct. An unsigned applet is allowed to draw a button.
   ☒ **A, C,** and **D** are incorrect because they are actions that are not permitted by an unsigned applet.
3. ☑ **B** is correct. A signed applet is permitted to access local resources.
   ☒ **A** is incorrect because the signed type of applet is allowed to access local resources.

Given an Architectural System Specification, Identify Appropriate Locations for Implementation of Specified Security Features and Select Suitable Technologies for Implementation of Those Features

4. ☑ **A** is correct. A message digest is a digital fingerprint value that is computed from a message, file, or byte stream.
   ☒ **B, C,** and **D** are not definitions of a message digest.
5. ☑ **B** is correct. The `getCallerPrincipal()` method returns the principal object. The `getName()` method can then be used to determine the caller's name (identity) from within enterprise bean code.
   ☒ **A, C,** and **D** are incorrect because `getCallerIdentity()` is a deprecated method, and `getIdentity()` and `getUserId()` are not valid methods.
6. ☑ **C** is correct. The `isCallerInRole()` method can be used to determine if the caller is within the specified role and therefore able to execute the EJB functionality.
   ☒ **A, B,** and **D** are incorrect because `inRole()`, `isAuthorized()`, and `isValid()` are not valid methods.
7. ☑ **A** is correct. The `getUserPrincipal()` method returns the principal object. The `getName()` method can then be used to determine the caller's name (identity) from within a JSP.
   ☒ **B, C,** and **D** are incorrect because `getPrincipal()`, `getUser()`, and `getIdentity()` are not valid methods.

8. ☑ **D** is correct. The `isUserInRole()` method can be used to determine whether the caller is within the specified role and therefore able to execute the JSP functionality.
☒ **A, B,** and **C** are incorrect because `inRole()`, `okToExecute()`, and `isValid()` are not valid methods.

9. ☑ **A** is correct. The deployer is responsible for mapping declarative authorization rules to the target environment.
☒ **B, C,** and **D** are incorrect because the Component Provider and Application Assembler are not responsible for providing this mapping. Authorizer is not a JEE role, so it can't be correct either.

10. ☑ **D** is correct. The deployer is responsible for mapping programmatic authorization rules to the target environment.
☒ **A, B,** and **C** are incorrect because the application assembler and component provider are not responsible for providing this mapping. Coder is not a JEE role, so it can't be correct either.

11. ☑ **B** is correct. The declarative authorization rules for EJBs are defined within the EJB deployment descriptor.
☒ **A, C,** and **D** are incorrect because EJB authorization rules are not declaratively defined in application properties, JNDI, or Enterprise bean code.

12. ☑ **B** is correct. The declarative authorization rules for web resources are defined within the application deployment descriptor.
☒ **A, C,** and **D** are incorrect because authorization rules for web resources are not declaratively defined in the EJB deployment descriptor, in the web resource, or in JMS.

13. ☑ **C** is correct. The programmatic authorization rules for EJBs are implemented within the enterprise bean code.
☒ **A, B,** and **D** are incorrect because EJB authorization rules are not programmatically implemented in JNDI, the EJB deployment descriptor, or JMS.

14. ☑ **D** is correct. The programmatic authorization rules for web resources are implemented within the JSP or servlet.
☒ **A, B,** and **C** are incorrect because authorization rules for web resources are not programmatically implemented in the Java Security Manager, security policy file, or JNDI.

15. ☑ **B** is correct. HTTP is not a valid authentication method.
☒ **A, C,** and **D** are incorrect. FORM, DIGEST, and CLIENT-CERT are valid authentication methods.