CHAPTER 8

Security

by Ron Monzillo

IN an enterprise computing environment, failure, compromise, or lack of availability of computing resources can jeopardize the viability of the enterprise. An organization must take steps to identify threats to security. Once they are identified, steps should be taken to reduce these threats.

Although J2EE products, and hence J2EE applications, may not displace existing enterprise security infrastructures, they do offer significant value when integrated with these existing infrastructures. The J2EE application programming model attempts to leverage existing security services rather than require new services or mechanisms.

This discussion begins with a review of some security concepts and mechanisms. We describe the security concerns and characteristics of enterprise applications and explore the application of J2EE security mechanisms to the design, implementation, and deployment of secure enterprise applications.

8.1 Security Threats and Mechanisms

Threats to enterprise-critical assets fall into a few general categories:

- Disclosure of confidential information
- Modification or destruction of information
- Misappropriation of protected resources
- Compromise of accountability

- Misappropriation that compromises availability

Depending on the environment in which an enterprise application operates, these threats may manifest themselves in different forms. For example, in a traditional single system environment, a threat of disclosure might manifest itself in the vulnerability of information kept in files. In a distributed environment with multiple servers and clients, a threat of disclosure might also result from exposures occurring as the result of networking.

Although not all threats can or need be eliminated, there are many circumstances where exposure can be reduced to an acceptable level through the use of the following security mechanisms: authentication, authorization, signing, encryption, and auditing. The following sections describe J2EE platform security mechanisms and indicate how to use the mechanisms to support security policies in an operational environment.

8.2 Authentication

In distributed component computing, *authentication* is the mechanism by which callers and service providers prove to one another that they are acting on behalf of specific users or systems. When the proof is bidirectional, we refer to it as *mutual authentication.* Authentication establishes the call identities and proves that the participants are authentic instances of these identities. An entity that participates in a call without establishing and/or proving an identity (that is, *anonymously*), is called *unauthenticated*.

When a client *program* run by a user makes the calls, the caller identity is likely to be that of the *user.* When the caller is an *application component* acting as an intermediary in a call chain originating with some user, the identity may be associated with that of the user, in which case the component would be *impersonating* the user. Alternatively, one application component may call another with an identity of its own and unrelated to that of its caller.

Authentication is often achieved in two phases. First, an *authentication context* is established by performing a service-independent authentication requiring knowledge of some secret. The authentication context encapsulates the identity and is able to fabricate *authenticators* (proofs of identity*)*. Then, the authentication context is used to authenticate with other (called or calling) entities. The basis of authentication entails controlling access to the authentication context,

and thus the ability to authenticate as the associated identity. Among the possible policies and mechanisms for controlling access to an authentication context are:

- Once the user performs an initial authentication, the processes the user starts inherit access to the authentication context.
- When a component is authenticated, access to the authentication context may be available to other related or trusted components, such as those that are part of the same application.
- When a component is expected to *impersonate* its caller, the caller may *delegate* its authentication context to the called component.

8.2.1 Protection Domains

Some entities may communicate without requiring authentication. A *protection domain* is a set of entities that are assumed or known to trust each other. Entities in such a domain need not be authenticated to one another.

Figure 8.1 illustrates that authentication is only required for interactions that cross the boundary of a protection domain. When a component interacts with components in the same protection domain, no constraint is placed on the identity that it can associate with its call. The caller may *propagate* the caller's identity, or *choose* an identity based on knowledge of authorization constraints imposed by the called component, since the caller's ability to *claim* an identity is based on trust, not authentication. If the concept of protection domains is employed to avoid the need for authentication, there must be a means to establish the boundaries of protection domains, so that trust in unproven identities does not cross these boundaries. Entities that are universally trusting of all other entities should not be trusted as a member of any protection domain.

In the J2EE architecture, a container provides an authentication boundary between external callers and the components it hosts. The boundaries of protection domains don't always align with those of containers. Containers enforce the boundaries, and implementations are likely to support protection domains that span containers. Although a container is not required to host components from different protection domains, an implementation may choose to do so.

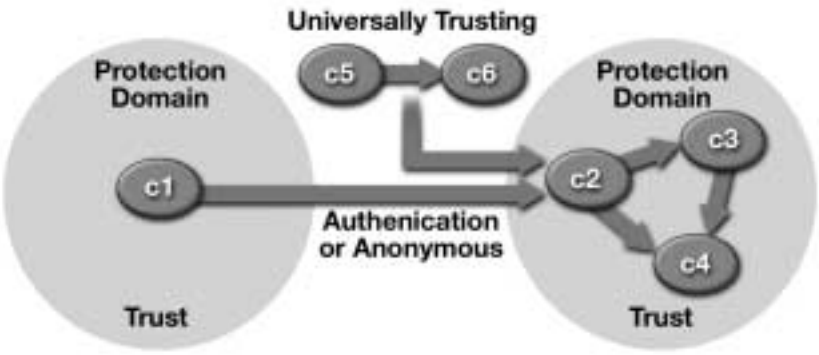

Figure 8.1 Protection Domain

For *inbound* calls, it is the container's responsibility to make an authentic representation of the caller identity available to the component in the form of a *credential*. An X.509 certificate and a Kerberos service ticket are examples of credentials. A passport or a driver's licence are analogous artifacts used in person-to-person interactions.

For *outbound* calls, the container is responsible for establishing the identity of the calling component. In general, it is the job of the container to provide bidirectional authentication functionality to enforce the protection domain boundaries of the deployed applications.

Without proof of component identity, the interacting containers must determine if there is sufficient inter-container trust to accept the container-provided representations of component identity. In some environments, trust may simply be presumed, in others it may be more explicitly evaluated based on inter-container authentication and possibly the comparison of container identities to lists of trusted identities. If a required proof of identity is not provided, and when a sufficient inter-container trust relationship is absent, a container should reject or abandon a call.

Figure 8.2 illustrates these authentication concepts in two scenarios: an authenticated user scenario and an unauthenticated user scenario.

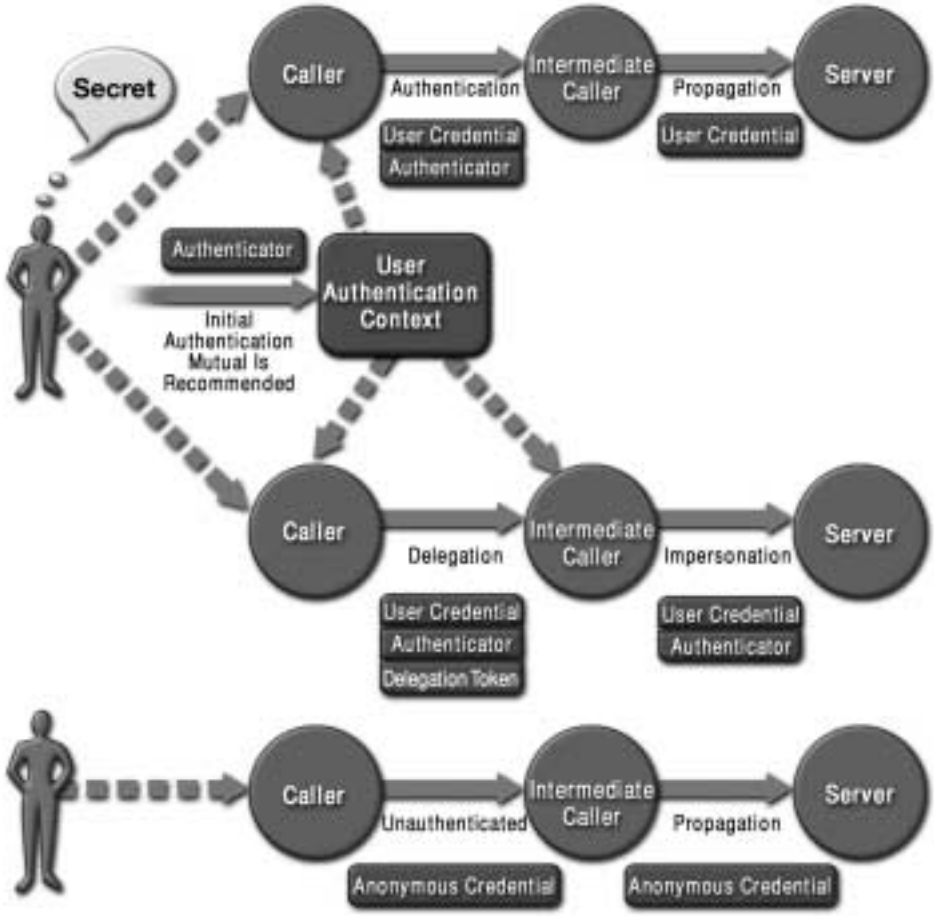

Figure 8.2 Authentication Scenarios

The authenticated user invokes a calling component that employs the user's authentication context to prove its identity to an intermediate component. When the called component makes a call it propagates the identity of its caller. The propagated identity is unproven, and so will be accepted only if the targets trust the caller, that is, if they reside in the same protection domain. The figure also differentiates identity propagation from delegation and subsequent impersonation. In propagation, the service providers bear the burden of determining whether they should accept propagated identities as authentic. In delegation, the user provides the called component with access to its authentication context, enabling the called

component to impersonate the user in subsequent calls. Impersonation requires the user to trust the impersonator to act in its behalf. The lower portion of the figure depicts the propagation of an unauthenticated user identity in the form of an anonymous credential. An anonymous credential is the one form of unproven identity that may be propagated independent of trust.

8.2.2 Authentication Mechanisms

In a typical J2EE application, a user would go through a client container to interact with enterprise resources in the Web or EJB tiers. Resources available to the user may be protected or unprotected. Protected resources are distinguished by the presence of *authorization rules* (see Section 8.3 on page 229) that restrict access to some subset of non-anonymous identities. To access a protected resource, a user must present a non-anonymous credential such that its identity can be evaluated against the resource authorization policy. In the absence of a trust relationship between the client and resource containers, the credential must be accompanied by an authenticator that confirms its validity. This section describes the various authentication mechanisms supported by the J2EE platform and how to configure them.

8.2.2.1 Web Tier Authentication

An Application Component Provider can designate that a collection of Web resources (Web components, HTML documents, image files, compressed archives, and so on) is protected by specifying an authorization constraint (described in Section 8.3.7.1 on page 235) for the collection. When an unauthenticated user tries to access a protected Web resource, the Web container will prompt the user to authenticate with the Web container. The request will not be accepted by the Web container until the user identity has been proven to the Web container and shown to be one of the identities granted permission to access the resource. Caller authentication performed on the first access to a protected resource is called *lazy authentication.*

When a user tries to access a protected Web-tier resource, the Web container activates the authentication mechanism defined in the application's deployment descriptor. J2EE Web containers must support three authentication mechanisms: HTTP basic authentication, form-based authentication, and HTTPS mutual authentication, and are encouraged to support HTTP digest authentication.

In *basic authentication,* the Web server authenticates a principal using the user name and password obtained from the Web client. In *digest authentication* a Web

client authenticates to a Web server by sending the server a message digest along with its HTTP request message. The digest is computed by employing a one-way hash algorithm to a concatenation of the HTTP request message and the client's password. The digest is typically much smaller than the HTTP request, and doesn't contain the password.

Form-based authentication lets developers customize the authentication user interface presented by an HTTP browser. Like HTTP basic authentication, form-based authentication is a relatively vulnerable authentication mechanism, since the content of the user dialog is sent as plain text, and the target server is not authenticated.

In single-signon environments, discretion must be exercised in customizing an application's authentication interface. It may be preferable to provide a single enterprise-wide custom user authentication interface, rather than implementing a set of application-specific interfaces.

With *mutual authentication*, the client and server use X.509 certificates to establish their identity. Mutual authentication occurs over a channel protected by SSL. Hybrid mechanisms featuring either HTTP basic authentication, form-based authentication, or HTTP digest authentication over SSL are also supported.

Authentication Configuration

An authentication mechanism is configured using the `login-config` element of the Web component deployment descriptor. Code Example 8.1, Code Example 8.2, and Code Example 8.3 illustrate the declaration of each type of authentication mechanism.

```
<web-app>
    <login-config>
        <auth-method>BASIC|DIGEST</auth-method>
        <realm-name>jpets</realm-name>
    </login-config>
</web-app>
```

Code Example 8.1 HTTP Basic and Digest Authentication Configuration

```
<web-app>
    <login-config>
        <auth-method>FORM</auth-method>
```

```
        <form-login-config>
            <form-login-page>login.jsp</form-login-page>
            <form-error-page>error.jsp</form-error-page>
        </form-login-config>
    </login-config>
</web-app>
```

Code Example 8.2 Form-Based Authentication Configuration

```
<web-app>
    <login-config>
        <auth-method>CLIENT-CERT</auth-method>
    </login-config>
</web-app>
```

Code Example 8.3 Client Certificate Authentication Configuration

Hybrid Authentication

In both HTTP basic and form-based authentication, passwords are not protected for confidentiality. This vulnerability can be overcome by running these authentication protocols over an SSL-protected session, which ensures that all message content, including the client authenticators, are protected for confidentiality. Code Example 8.4 demonstrates how to configure HTTP basic authentication over SSL using the `transport-guarantee` element. Form-based authentication over SSL is configured in the same way.

```
<web-app>
    <security-constraint>
        ...
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
</web-app>
```

Code Example 8.4 SSL Hybrid Authentication Mechanism

8.2.2.2 EJB Tier Authentication

The J2EE 1.2 platform specification requires that EJB containers and EJB client containers support interoperable caller authentication at the EJB container, even if network firewall technology prevents direct Internet interaction (via RMI) between client containers and enterprise beans. One way that an EJB container can protect access to enterprise beans is to entrust the Web container to vouch for the identity of users accessing the beans via protected Web components. As illustrated in Figure 8.3, such configurations use the Web container to enforce protection domain boundaries for Web components and the enterprise beans that they call.

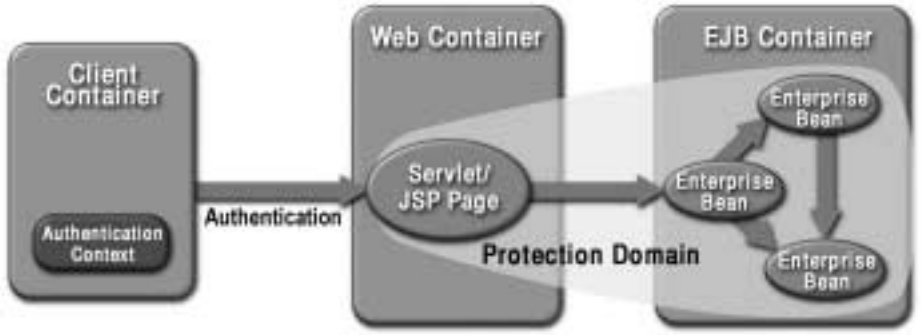

Figure 8.3 Typical J2EE Application Configuration

8.2.2.3 Common Secure Interoperability

The Common Secure Interoperability, Version 2 (CSIv2) protocol specifies how to secure EJB invocations.

8.2.3 Authentication Call Patterns

In this section, we describe authentication call patterns that can safely be followed and those that should be avoided.

8.2.3.1 Changing Authentication Identity

It is sometimes necessary for a user to change authentication identity. This may arise when a user attempts to visit a protected resource and is rebuffed because of a lack of access authority. While a user can exit the browser and restart the authentication process, this is not always the optimal solution. Instead, it is preferable to have the application client container apply proactive authentication.

8.2.3.2 Patterns to Avoid

In a multitier, multicomponent application, certain call patterns should be avoided for security reasons. For example, an application that calls protected EJB resources from unprotected Web resources can run into problems, because the Web tier's lazy authentication paradigm doesn't require user authentication except when the user attempts to access a protected resource. While the protection requirement can be moved to the EJB tier, care must be taken to ensure that users who are capable of authenticating can do so. With lazy authentication, a user who wants to visit a protected EJB resource must have visited a protected Web resource. One way to ensure this would be to front every protected EJB resource with a protected Web resource. Another approach would be to link to a protected Web resource (perhaps appearing as an authenticate button) on every Web resource that calls EJB resources. This approach gives the user the option of authenticating (by visiting the protected Web resource linked behind the button) prior to accessing an EJB resource, especially after having been denied access by the EJB resource through an unprotected page.

When an application is deployed with a hybrid authentication mechanism, the Deployer must ensure that the `transport-guarantee` element of each protected Web resource is set to `CONFIDENTIAL`. Otherwise, the client authenticator won't be fully protected.

8.2.3.3 Enterprise Information System Tier Authentication

In integrating with enterprise information systems, J2EE components may use different security mechanisms and operate in different protection domains than the resources they access. In these cases, the calling container can be configured to manage the authentication to the resource for the calling component. This form of authentication is called *container-managed resource manager signon*. The J2EE architecture also recognizes that some components require an ability to manage the specification of caller identity, and the production of a suitable authenticator directly. For these applications, the J2EE architecture provides a means for an application component to engage in what is called *application-managed resource manager signon*. Application-managed resource manager signon is used when manipulating the authentication details is a fundamental aspect of the component's functionality.

The `resource-ref` elements of a component's deployment descriptor (described in greater detail in Section 8.3 on page 229) declares the resources used by the component. The subelement `res-auth` specifies the type of signon authentication. Components can use the `EJBContext.getCallerPrincipal` and `HttpServ-`

`letRequest.getUserPrincipal` methods to obtain the identity of their caller. The component may map the caller identity to a new identity and/or authentication secret as required by the target enterprise information system.

With container-managed resource manager signon, the container would perform the *principal mapping* on behalf of the calling component. Container-managed principal mapping isn't explicitly defined in any of the J2EE specifications. Whether it is performed by the container or embedded in the caller, the mapping of caller identity to an identity and authentication secret capable of accessing resources in the enterprise information system tier should be modeled as a protected resource, and secured by appropriate authorization rules (see Section 8.3.6 on page 233).

The Connector architecture discussed in Section 6.3 on page 154 offers a standard API for application-managed resource manager signon. The Connector provided API will ensure portability of components that authenticate with enterprise information systems.

8.2.3.4 Authentication Gateway Self-Registration Pattern

Some applications must be prepared to admit users whose identities cannot be known until the user's first use of the application. Such applications require an automated means for users to register an authentication identity for themselves. For example, many e-commerce applications are designed to make it as easy as possible for a user to become a customer. In contrast to typical computer user authentication environments, where a user must wait for an administrator to set up the user's account, many e-commerce applications enable users to set up their own accounts without administrative intervention. Frequently the user is required to provide his or her identity and location, agree to some contractual obligations, provide credit card information for payment, and establish a password to protect the account. Once the registration dialog is complete, the user can access the protected resources of the site.

In the future, although client certificates may replace the identity and password elements of the registration to improve the accountability of the authentication, all security functionality should not be moved into the application. An authentication gateway, self-registration pattern would utilize the following components and steps:

- A Front Component. negotiates user identity and shared secret.
- The Front Component instantiates an object with an embedded secret, plus

supplementary user information.

- The Front Component authenticates the user and then operates as a Front Component on behalf of the user.
- The component uses a `CallsAs` identity that is authorized to call all target objects.
- The component may evaluate if its caller is distinguished.

An example of such a self-registration pattern might be a shopping cart application requiring portable self-registration.

Web resources that provide the user interface for auto-registration must be protected. This is accomplished by setting the `transport-guarantee` of these resources to `CONFIDENTIAL`.

8.2.4 Exposing Authentication Boundaries with References

The Application Component Provider is responsible for declaring references made by each component to other J2EE components and to external resources. These declarations are made in the deployment descriptor. In addition to their role in locating services, such declarations inform the Deployer of all the places in the application where authentication may be necessary. Enterprise bean references are declared using `ejb-ref` elements. Enterprise information system references are declared with `resource-ref` elements. In both cases, the declarations are made in the scope of the calling component, and the collection of declared references serves to expose the application's inter-component/resource call tree.

It is important to know when to set up inter-container authentication and data protection mechanisms or trust relationships.

8.3 Authorization

Authorization mechanisms limit interactions with resources to collections of users or systems for the purpose of enforcing integrity, confidentiality, or availability constraints. Such mechanisms allow only authentic caller identities to access components. Mechanisms provided by the J2EE platform can be used to control access to code based on identity properties, such as the location and signer of the calling code, and the identity of the user of the calling code. As mentioned in the section on authentication, caller identity can be established by selecting from the set of authen-

tication contexts available to the calling code. Alternatively, the caller may propagate the identity of its caller, select an arbitrary identity, or make the call anonymously.

In all cases, a credential is made available to the called component. The credential contains information describing the caller through its identity attributes. In the case of anonymous callers, a special credential is used. These attributes uniquely identify the caller in the context of the authority that issued the credential. Depending on the type of credential, it may also contain other attributes which define shared authorization properties (for example, group memberships) that distinguish collections of related credentials. The identity attributes and shared authorization attributes appearing in the credential are referred to together as the caller's *security attributes.* In the J2SE platform, the identity attributes of the code used by the caller may also be included in the caller's security attributes. Access to the called component is determined by comparing the caller's security attributes with those required to access the called component.

In the J2EE architecture, a container serves as an authorization boundary between callers and the components it hosts. The authorization boundary exists inside the container's authentication boundary, so that authorization is considered in the context of successful authentication. For inbound calls, the container compares security attributes from the caller's credential with the access control rules for the target component. If the rules are satisfied, the call is allowed. Otherwise, the call is rejected.

There are two fundamental approaches to defining access control rules: *capabilities* and *permissions.* The capabilities approach focuses on what a caller can do. The permissions approach focuses on who can do something. The J2EE application programming model focuses on permissions. In the J2EE architecture, the job of the Deployer is to map the permission model of the application to the capabilities of users in the operational environment.

8.3.1 Declarative Authorization

The container-enforced access control rules associated with a J2EE application are established by the Deployer. The Deployer uses a deployment tool to map an application permission model (typically) supplied by the Application Assembler to policy and mechanisms that are specific to the operational environment. The application permission model is contained in a deployment descriptor.

The deployment descriptor defines logical privileges called *security roles* and associates them with components to define the privileges required to be granted

permission to access components. The Deployer assigns these logical privileges to specific callers to establish the capabilities of users in the runtime environment. Callers are assigned logical privileges based on the values of their security attributes. For example, a Deployer might map a security role to a security group in the operational environment such that any caller whose security attributes indicate that it is a member of the group would be assigned the privilege represented by the role. As another example, a Deployer might map a security role to a list containing one or more principal identities in the operational environment such that a caller authenticated as one of these identities would be assigned the privilege represented by the role.

The EJB container grants permission to access a method only to callers that have at least one of the privileges associated with the method. Security roles also protect Web resource collections, that is, a URL pattern and an associated HTTP method, such as `GET`. The Web container enforces authorization requirements similar to those for an EJB container.

In both tiers, access control policy is defined at deployment time, rather than application development. The Deployer can modify the policy provided by the Application Assembler. The Deployer refines the privileges required to access the components, and defines the correspondence between the security attributes presented by callers and the container privileges. In any container, the mapping from security attributes to privileges is scoped to the application, so that the mapping applied to the components of one application may be different from that of another application.

8.3.2 Programmatic Authorization

A J2EE container makes access control decisions before dispatching method calls to a component. As a result, the logic or state of a component doesn't affect the access decisions. However, a component can use two methods, `EJBContext.isCallerInRole` (for use by enterprise bean code) and `HttpServletRequest.isUserInRole` (for use by Web components), to perform finer-grained access control. A component uses these methods to determine whether a caller has been granted a privilege selected by the component based on the parameters of the call, the internal state of the component, or other factors such as the time of the call.

The Application Component Provider of a component that calls one of these functions must declare the complete set of distinct `roleName` values used in all of its calls. These declarations appear in the deployment descriptor as `security-role-ref` elements. Each `security-role-ref` element links a privilege name

embedded in the application as a `roleName` to a security role. It is ultimately the Deployer who establishes the link between the privilege names embedded in the application and the security roles defined in the deployment descriptor. The link between privilege names and security roles may differ for components in the same application.

In addition, there is also the concept of instance-scoped access control. An application can use the `getCallerPrincipal` and `IsUserInRole` methods to acheive instance-scoped access control. Use these methods following the Distinguished Caller pattern. Establish the distinguished caller identities, obtained from call arguments, and embed them in the instance when creating the instance. Then, if `getCallerPrincipal.getName` evaluates to a distinguished caller, allow the caller to proceed. This pattern can also be combined with class-scoped policies.

Keep in mind that relying on `getCallerPrincipal` raises portability issues.

8.3.3 Declarative Versus Programmatic Authorization

There is a trade-off between the external access control policy configured by the Deployer and the internal policy embedded in the application by the Component Provider. The former is more flexible after the application has been written. The latter provides more flexibility, in the form of functionality, while the application is being written. The former is transparent and completely comprehensible. The latter is buried in the application such that it may only be completely understood by the those who developed the application. These trade-offs should be considered in choosing the authorization model for particular components and methods.

8.3.4 Isolation

When designing the access control rules for protected resources, take care to ensure that the authorization policy is consistently enforced across all the paths by which the resource may be accessed. When method-level access control rules are applied to a component, care must be taken that a less protected method does not serve to undermine the policy enforced by a more rigorously protected method. Such considerations are most significant when component state is shared by disparately protected methods. The simplifying rule of thumb is to apply the same access control rules to all the methods of a component, and to partition an application as necessary to enforce this guideline unless there's some specific need to architect an application otherwise.

CSIv2 IORs define per object annotations rather than Enterprise JavaBeans method level annotations. Using CSIv2 IORs, you cannot differentially protect the methods of an enterprise bean for authentication. As a result, the overall programming model includes objects which may or may not have protected methods. To handle this, it is best to have all client's authenticate even if they are not required to do so.

8.3.5 Identity Selection

When setting an application's access control policy, the Application Component Provider bases policy decisions on assumptions about the call identities selected by the application callers. When a call passes through intermediary components, the caller identity at the destination component may depend on the identity selection decisions made by the intermediaries. The destination component may assume that caller identities have been propagated along the call chain such that the identity of its caller will be that of the caller who initiated the chain. In other cases, the called component must assume that one or more of the callers in its call path will employ an identity selection policy other than identity propagation. The Application Assembler is responsible for communicating these assumptions to the Deployer. The assembler explicitly specifies a component's `runAs` identity—the caller identity for inter-component calls—in the deployment descriptor. Unless the Deployer has other instructions from the Application Assembler, they should assume that each caller will propagate the identity of the caller's identity.

8.3.6 Encapsulation for Access Control

The component model of an application may be used to impose authorization boundaries around what might otherwise be unprotected resources. This can be done by using accessor components to implement the authorization barrier. If accessor components are used to create an authorization boundary, access control can either be done externally by the container, or internally by the component, or both.

An accessor component may encapsulate the mapping to an authentication context suitable for interacting with an external resource. Considered in the context of principal mapping for the purpose of authenticating and gaining access to enterprise information system resources, encapsulation for access control can be used to control who is authorized to access a mapping. Depending on the form of the mapping, the authorization rules may be more or less complex. For example, if all access to a resource is performed via a single conceptually omnipotent

enterprise information system tier identity, then the J2EE application can implement secure access to the resource by limiting who can access the accessor. If the mapping of authentication context is many-to-many, then the authorization configuration of the accessor may need to define which of a collection of mappings are accessible to the caller, and which should be assumed by default (if the caller does not assert which mapping it requires).

8.3.6.1 Shared Accessor Identity

A component may be made capable of accessing an external resource, either by container-managed sign on or bean-managed sign on. In addition, method permissions can be associated to the component's methods for accessing the resource. Later, access can be granted only to those J2EE principals that have the capability to access this external resource.

8.3.6.2 Private Accessor Identity

An enterprise bean, such as a stateful session bean, can use bean-managed sign on to an external resource. The session bean relies on a protected entity bean to map the J2EE principal to the corresponding principal in the external resource's realm, and also to the corresponding authenticator if necessary. In this scenario, either one protected entity bean holds all the mappings, and that bean limits access to a particular mapping to a specific principal (returned by `getCallerPrincipal`), or there is one entity bean per mapping. If using the former approach, keep in mind the restrictions on portability of applications when relying on `getCallerPrincipal`.

8.3.7 Controlling Access to J2EE Resources

In a typical J2EE application, a client would go through its container to interact with enterprise resources in the Web or EJB tiers. Resources available to the user may be protected or unprotected. Protected resources are distinguished by the presence of authorization rules defined in deployment descriptors that restrict access to some subset of non-anonymous identities. To access a protected resource, a user must present a non-anonymous credential such that its identity can be evaluated against the resource authorization policy.

8.3.7.1 Controlling Access to Web Resources

To control access to a Web resource, an Application Component Provider or Application Assembler specifies a `security-constraint` element with an `auth-constraint` subelement in the Web deployment descriptor. Code Example 8.5 illustrates the definition of a protected resource in a Web component deployment descriptor. The descriptor specifies that the URL `/control/placeorder` can only be accessed by users acting in the role of `customer`.

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>placeorder</web-resource-name>
        <url-pattern>/control/placeorder</url-pattern>
        <http-method>POST</http-method>
        <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>customer</role-name>
    </auth-constraint>
</security-constraint>
```

Code Example 8.5 Web Resource Authorization Configuration

8.3.7.2 Controlling Access to Enterprise Beans

An Application Component Provider or Application Assembler that has defined security roles for an enterprise bean can also specify the methods of the remote and home interface that each security role is allowed to invoke. This is done in the form of `method-permission` elements. Ultimately, it is the assignment of users to roles that determines if a resource is protected. When the roles required to access the enterprise bean are assigned only to authenticated users, the bean is protected.

Code Example 8.6 contains two styles of method specifications. The first refers to all of the remote and home interface methods of an enterprise bean. The second is used for referring to a specific method of the remote or home interface of an enterprise bean. If there are multiple methods with the same overloaded name, this style refers to all of the overloaded methods. Method specifications can be further qualified with parameter names for methods with an overloaded name.

```
<method-permission>
    <role-name>admin</role-name>
        <method>
            <ejb-name>TheOrder</ejb-name>
            <method-name>*</method-name>
        </method>
</method-permission>

<method-permission>
    <role-name>customer</role-name>
    <method>
        <ejb-name>TheOrder</ejb-name>
        <method-name>getDetails</method-name>
    </method>
    <method>
    ...
</method-permission>
```

Code Example 8.6 Enterprise Bean Authorization Configuration

Enterprise beans can now have local and local home interfaces. To accommodate this, the deployment descriptor includes the `unchecked` and `exclude-list` elements. The Application Assembler uses the `unchecked` element instead of a role name in the `method-permission` element to indicate that a method should not be checked for authorization. Similarly, the Application Assembler uses the `exclude-list` element to indicate a set of methods that should not be called. The Deployer configures the bean's security to disallow access to these methods.

8.3.7.3 Unprotected Resources

Many applications feature unprotected Web-tier content, available to any caller without authentication. Unprotected resources are characterized by the absence of a requirement that their caller be authenticated. In the Web tier, unrestricted access is provided simply by leaving out an authentication rule.

Some applications also feature unprotected enterprise beans. For example, the sample application allows anonymous, unauthenticated users to access certain EJB resources. In the EJB tier, the caller must be authenticated, although the authentication does not have to occur on the call to the enterprise bean. It is possi-

ble for the EJB container to accept a propagated identity based on trust in the calling container. In EJB 2.0, unrestricted access is accomplished by mapping at least one role which is permitted access to the resource is accomplished by designating the method as `unchecked` in the method permission element.

8.3.8 Example

To understand how each application, and each component within an application can apply its own authorization requirements, consider the following examples.

One application is assembled from two enterprise beans, EJB 1 and EJB 2, each with one method. Each method calls `isCallerInRole` with the role name `MANAGER`. The deployment descriptor includes a `security-role-ref` element for the call to `isCallerInRole` in each enterprise bean. The `security-role-ref` for EJB 1 links `MANAGER` to the role `good-managers` and the `security-role-ref` element for EJB 2 links `MANAGER` to the role `bad-managers`. The deployment descriptor defines two method-permission elements, one establishes that the role `employees` can access all methods of EJB 1 and the other does the same for EJB 2. The deployment descriptor has 3 `security-role` elements: `employees`, `good-managers`, and `bad-managers`. The Deployer assigns User 1 to roles `employees` and `good-managers` and assigns User 2 to roles `employees` and `bad-managers`.

A second application, with one enterprise bean EJB 3, is also deployed in the container. EJB 3 also makes a call to `isCallerInRole` with the role name `MANAGER`. The deployment descriptor for this second application contains a `security-role-ref` element that links `MANAGER` to the role `good-managers`. Similarly, the deployment descriptor defines one `method-permission` element that establishes that the role `employees` can access all the methods of EJB 3. The deployment descriptor has 2 role elements, `employees` and `good-managers`. The Deployer assigns User 2 to roles `employees` and `good-managers`.

Figure 8.4 illustrates the configuration of method permissions as a relationship between roles and methods. It also illustrates the mapping of caller security attributes to roles, and the link between privilege names embedded in the application and roles.

Table 8.1 lists the authorization decisions that occur when different users initiate method calls on these enterprise beans. For example, when User 1 initiates a method call on EJB 2's method, the container dispatches the call because the `method-permission` element specifies the security roles `employees` and `good-managers`, and the Deployer has assigned User 1 to the former security role.However, the `isCallerInRole(MANAGER)` method returns false, because the `security-role-`

`ref` element for EJB 2 links `MANAGER` to the security role `bad-managers`, which is not satisfied for User 1. When User 1 invokes a method on EJB 3, the call isn't even dispatched, because User 1 isn't assigned to any security roles.

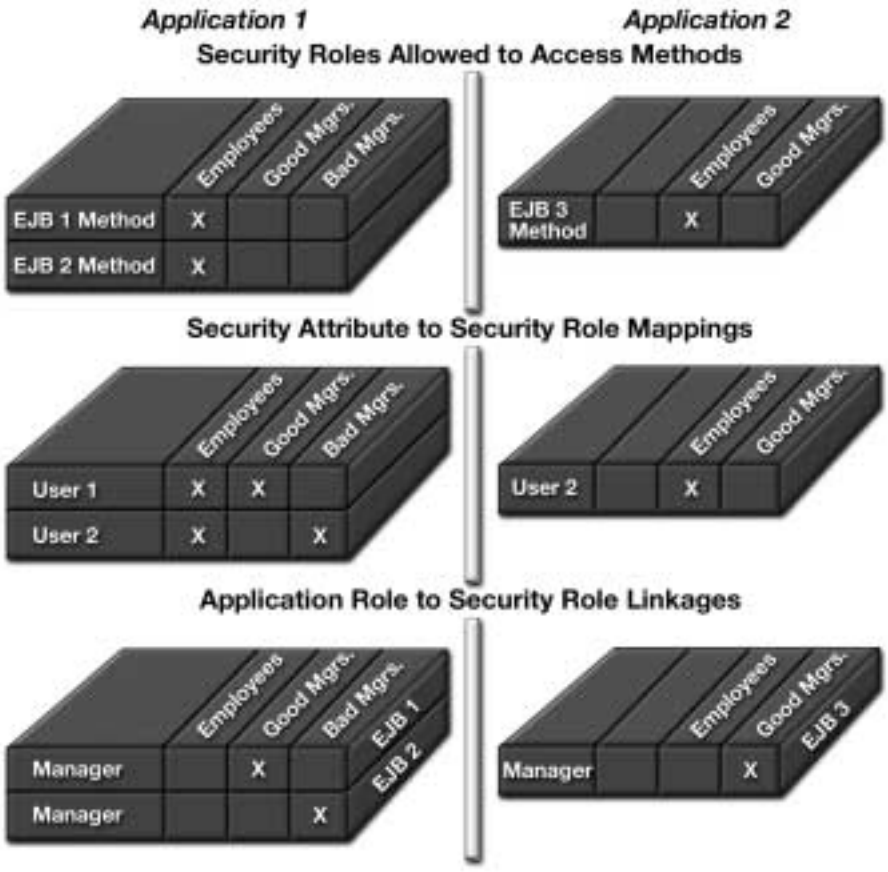

Figure 8.4 Authorization Scenario

Table 8.1 Authorization Decisions

Call	Call Dispatched?	isCallerInRole?
User 1 - EJB 1	yes	true
User 1 - EJB 2	yes	false
User 1 - EJB 3	no	never called

Table 8.1 Authorization Decisions (continued)

Call	Call Dispatched?	isCallerInRole?
User 2 - EJB 1	yes	false
User 2 - EJB 2	yes	true
User 2 - EJB 3	yes	true

8.4 Protecting Messages

In a distributed computing system, a significant amount of information is transmitted through networks in the form of messages. Message content is subject to three main types of attacks. Messages might be intercepted and modified for the purpose of changing the affects they have on their recipients. Messages might be captured and reused one or more times for the benefit of another party. Messages might be monitored by an eavesdropper in an effort to capture information that would not otherwise be available. Such attacks can be minimized by using integrity and confidentiality mechanisms.

8.4.1 Integrity Mechanisms

Integrity mechanisms ensure that communication between entities is not being tampered with by another party, especially one that can intercept and modify their communications. Integrity mechanisms can also be used to ensure that messages can only be used once.

Message integrity is ensured by attaching a *message signature* to a message. The message signature is calculated by using a one-way hash algorithm to convert the message contents into a typically smaller, fixed length *message digest* that is then *signed* (that is, cryptographically enciphered, typically using a public key mechanism). A message signature ensures that modification of the message by anyone other than the caller will be detectable by the receiver. Although there are always things a sender can do (including publishing its private authentication keys), to compromise a receiver's ability to hold it accountable for a received message, both parties to the communication would be wise to select an integrity mechanism that appends a message confounder (typically a sequence number and a timestamp) to the message before the digest. The purpose of the confounder is to make the message authenticator useful only once. This prevents a malicious recip-

ient from claiming that it received a message more times than it did or from reusing an intercepted message for its own purpose. In exchange for these receiver-side limitations, a measure of accountability is transferred to the sender.

In the J2EE architecture, a container serves as an authentication boundary between callers and the components it hosts. Information may flow in both directions on a call (that is, a call may have input, output, or input and output parameters). The Deployer is responsible for configuring containers to safeguard interactions between components. A Deployer must configure the containers involved in a call to implement integrity mechanisms either because the call will traverse open or unprotected networks, or because the call will be made between components that do not trust each other. The latter is necessary to ensure that messages can only be used once, and to reduce the plausibility of arguments made by either of the communicants that they did not send the messages claimed to have been received. When the Deployer configures the integrity mechanisms, the calling container must attach a digital signature to the message stream only when the message stream contains XML messages. Keep in mind that enterprise beans and servlets rely on SSL for integrity, which means that J2EE containers process message streams rather than individual messages.

The performance cost associated with applying integrity protection to all message communication is as much a property of the operational environment as it is a consequence of the cost of the protection. One way to safeguard the integrity of application messages without unnecessarily limiting the space of operational environments, is to capture application-specific knowledge identifying which messages must be integrity protected. The place to capture this information is in the application's deployment descriptor.

8.4.2 Using XML Digital Signatures

8.4.3 Confidentiality Mechanisms

Confidentiality mechanisms ensure that communication between entities is kept private. Privacy is achieved by encrypting the message contents. Because symmetric (that is, shared secret) encryption mechanisms are generally much less expensive (in terms of compute resources) than are asymmetric (that is, public key) mechanisms, it is quite common for an asymmetric mechanism to be used to secure the exchange of a symmetric encryption key, which is then used to encrypt the message contents. Note that this does not apply when using SSL to achieve confidentiality for EJB and servlet invocations, though it does apply for SOAP.

The Deployer is responsible for configuring containers to apply confidentiality mechanisms to ensure that sensitive information is not disclosed to third parties. Despite the improved performance of the shared secret mechanisms, the costs of message encryption are significant, and should be expected to have an adverse effect on performance when confidentiality mechanisms are applied where they are not needed. The Application Assembler should supply the Deployer with information on those components that should be protected for confidentiality. The Deployer then must configure the containers involved in a call to employ a confidentiality mechanism whenever one of the method calls identified by the Application Assembler will traverse open or unprotected networks. In addition to applying confidentiality mechanisms where appropriate, the Deployer should configure containers to reject call requests or responses with message content that should be protected but isn't protected. Message integrity is typically verified as a side effect of confidentiality.

8.4.4 Identifying Sensitive Components

We recommend that the Application Assembler identify the components whose method calls feature parameters or return values that should be protected for integrity and/or confidentiality. The deployment descriptor is used to convey this information. For enterprise beans, this would be done in a `description` subelement for the entire object. For servlets and JSP pages, this would be done in the `transport-guarantee` subelement of the `user-data-constraint` subelement of a `security-constraint`. In cases where a component's interactions with an external resource are known to carry sensitive information, these sensitivities should be described in the `description` subelement of the corresponding `resource-ref`.

8.4.5 Ensuring Confidentiality of Web Resources

In addition to understanding how to configure Web transport guarantees, it is important to understand the properties of HTTP methods, and the effects these properties have when a link is followed from one Web resource to another. When a resource contains links to other resources, the nature of the links determines how the protection context of the current resource affects the protection of requests made to the linked resources.

When a link is *absolute* (that is, the URL begins with `https://` or `http://`), the HTTP client container will ignore the context of the current resource and access the linked resource based on the nature of the absolute URL. If the URL of

the link begins with `https://`, a protected transport will be established with the server before the request is sent. If the URL of the link begins with `http://`, the request will be attempted over an insecure transport. When the link is *relative*, the HTTP client container will protect an access to a linked resource based on whether the resource in which the link occurs was protected.

The application developer should consider these link properties most carefully when a linked request must carry confidential data back to the server. There are a few choices available to ensure security in such cases. For example, an application developer might choose to use secure absolute links to ensure the transport protection of requests that carry confidential data. This would solve the security problem, at the expense of constraining the application to a very specific naming environment.

Another option, assuming that an application opts for portability and uses relative links, is for the Deployer to configure the application so that wherever there is a confidential interaction from one resource to another, both are deployed with a confidential transport guarantee. This approach will ensure that an HTTP client container will not send a request to a protected resource without protecting it.

As a related point, the `POST` method is favored over the `GET` method for delivering confidential request data, since data sent via `GET` appears in both client- and server-side logs.

8.5 Auditing

Auditing is the practice of capturing a record of security-related events for the purpose of being able to hold users or systems accountable for their actions. A common misunderstanding of the value of auditing is evident when auditing is used solely to determine whether security mechanisms are serving to limit access to a system. When security is breached, it is usually much more important to know who has been allowed access than who has not. Only by knowing who has interacted with the system do we have a chance of determining who should be held accountable for a breach of security. Moreover, auditing can only be used to evaluate the effective security of a system when there is a clear understanding of what is audited and what is not.

The Deployer is responsible for configuring the security mechanisms that will be applied by the enterprise containers. Each of the configured mechanisms may be thought of as a constraint that the containers will attempt to enforce on interactions between components. It should be possible for the Deployer or System

Administrator to review the security constraints established for the platform, and to associate an audit behavior with each constraint so that the container will audit one of the following:

- All evaluations where the constraint was satisfied
- All evaluations where it was not satisfied
- All evaluations independent of outcome
- No evaluations

It would also be prudent to audit all changes (resulting from deployment or subsequent administration) to the audit configuration or the constraints being enforced by the platform. Audit records must be protected so that attackers cannot escape accountability for their actions by expunging incriminating records or changing their content.

The J2EE programming model aims to shift the burden of auditing away from developers and integrators to those who are responsible for application deployment and management. Therefore, although not currently mandated by the J2EE specification, we recommend that J2EE containers provide auditing functionality that facilitates the evaluation of container-enforced security policy.

8.6 Summary

A primary goal of the J2EE platform is to relieve the application developer from the details of security mechanisms and facilitate the secure deployment of an application in diverse environments. The J2EE platform addresses this goal by defining a clear separation of responsibility between those who develop application components, those who assemble components into applications, and those who configure applications for use in a specific environment. By allowing the Component Provider and Application Assembler to specify the parts of an application that require security, deployment descriptors provide a means outside of code for the developer to communicate these needs to the Deployer. They also enable container-specific tools to give the Deployer easier ways to engage the security constraints recommended by the developer.

An Application Component Provider identifies all of the security dependencies embedded in a component including:

- The names of all the role names used by the component in calls to `IsCallerInRole` or `isUserInRole`
- References to all of the external resources accessed by the component
- References to all the inter-component calls made by the component

An Application Component Provider may also provide a method permission model, along with information that identifies the sensitivity with respect to privacy of the information exchanged in particular calls.

An Application Assembler combines one or more components into an application package and then rationalizes the external view of security provided by the individual components to produce a consistent security view for the application as a whole. The objective of the Application Assembler is to provide this information so that it can inform the actions of a Deployer.

A Deployer is responsible for taking the security view of the application provided by the Application Assembler and using it to secure the application in a specific operational environment. The Deployer uses a platform-specific deployment tool to map the view provided by the assembler to the policies and mechanisms that are specific to the operational environment. The security mechanisms configured by the Deployer are implemented by containers on behalf of the components hosted in the containers.

J2EE security mechanisms combine the concepts of container hosting, plus the declarative specification of application security requirements, with the availability of application-embedded mechanisms. This provides a powerful model for secure, interoperable, distributed component computing.

For our latest thinking on security, see

`http://java.sun.com/j2ee/blueprints/security/`