CHAPTER 7

Security

VIRTUALLY every enterprise application exposed through a Web service has a need for security at some level. An enterprise's data is an important asset to every business, and a good security system is necessary to ensure its safety and integrity. Businesses need to safeguard their systems and their data resources from malicious use by unauthorized intruders, both internal and external to the business, and from inadvertent or unintended mischief. Businesses also must keep message exchanges with other entities secure.

Security for Web services is two-fold: it encompasses both the security requirements of a typical enterprise as well as the particular security needs of the Web services themselves. An enterprise's business security requirements are well known, and it is just as important to identify the security needs of a service. For example, the developer of a Web service, after assessing its business needs, may only want to let a certain set of users access particular resources. Setting up security also involves some quality issues. In particular, because Web services have a high degree of interaction with varied clients, it is important to keep security measures from being overly intrusive and thus maintain the ease of use of a service. A service needs to promote its interoperability and make its security requirements and policy known to clients. Often, a service needs to keep a record of transactions or a log of access made to particular resources. The service must not only guarantee privacy, but it must also keep these records in case a claim is made at some later date about a transaction occurrence.

To address security needs, enterprise platforms use well-known mechanisms to provide for common protections, as follows:

- Identity, which enables a business or service to know who you are
- Authentication, which enables you to prove you are who you claim to be
- Authorization, which lets you can establish who has access to specific resources
- Data integrity, which lets you establish that data has not been tampered with
- Confidentiality, which restricts access to certain messages only to intended parties
- Non-repudiation, which lets you prove a user performed a certain action such that the user cannot deny it
- Auditing, which helps you to keep a record of security events

These are just some of the concepts important to security, and there are others such as trust, single sign-on, federation, and so forth. The chapter describes mechanisms to address and handle the threats to security, including credentials for establishing identity, encryption to safeguard the confidentiality of messages, digital signatures to help verify identity, and secure communication channels (such as HTTPS) to safeguard messages and data.

Keep in mind that the J2EE 1.4 platform does not invent new security mechanisms. Rather, the platform provides a programming model that, because it integrates existing security mechanisms, makes it easier to design and implement secure applications.

This chapter begins with an examination of some typical Web service security scenarios and use cases. It then covers the security technologies available on the J2EE 1.4 platform. Once the technologies are described, it shows how to design and implement secure Web services that rely on the JAX-RPC runtime. The chapter also covers the emerging technologies for Web service security, in particular message-level security.

7.1 Security Scenarios and Use Cases

Enterprise environments with Web site applications have a variety of security use case scenarios. The spread of Web services has given rise to additional, new use cases that require securing. Fortunately, Web services application use cases have similar security needs to those of Web-based enterprise applications. Typically, Web-based application use case scenarios involve browser-based applications that access services through the Internet. These applications allow users to access certain sets of Web pages—and not others—plus they allow users to perform some set of transactions. In addition, users may need access to other resources, such as a database, and they may need to interact with other applications.

Some of the security needs of Web site applications and Web services are strikingly similar. For example, a Web site application must authenticate its users, and a Web service application must authenticate its clients. However, Web services applications have additional security needs, because their use cases are typically application to application rather than user to application and because their communication interaction uses new technologies. Later in this chapter we examine security issues specific to Web services, plus we look at the specific details for implementing Web services-specific security mechanisms.

Let's first look at some typical Web services scenarios and examine the secure interactions between clients and services. Not only do we look at security issues relevant to client and service interactions, we also examine how service endpoints interact in a secure manner with resources and components of an enterprise to process requests. Before doing so, however, we examine basic security requirements.

7.1.1 General Security Requirements

Although varying greatly in implementation and functionality, J2EE Web services scenarios have common security requirements. They require certain security constraints for message exchange interactions and data passing between a client and a service. In addition to securing service and client interactions, Web service endpoints must be able to securely access other J2EE components (such as entity beans) and external resources (such as databases and enterprise information systems) to process client requests. Service endpoints may also need to interact with other Web services, and this, too, must be done in a secure manner.

Figure 7.1 shows an example of a Web service interaction, one in which a client request to the service causes the service endpoint to interact with other com-

ponents, resources, and systems. This illustrates that a Web service request can take many paths and result in interactions with different containers, components, and resources, including other Web services.

While requests to a Web service start with a client message sent to a Web or EJB container, designing a secure Web service involves more than just securing that initial interaction between the client and the service. For a truly secure service, you must also consider the security needs of the Web service endpoint's subsequent interactions with components, resources, and so forth, that it undertakes to process the request. A client's Web service invocation may result in a chain of subsequent calls, and each call may have its own, unique security requirement.

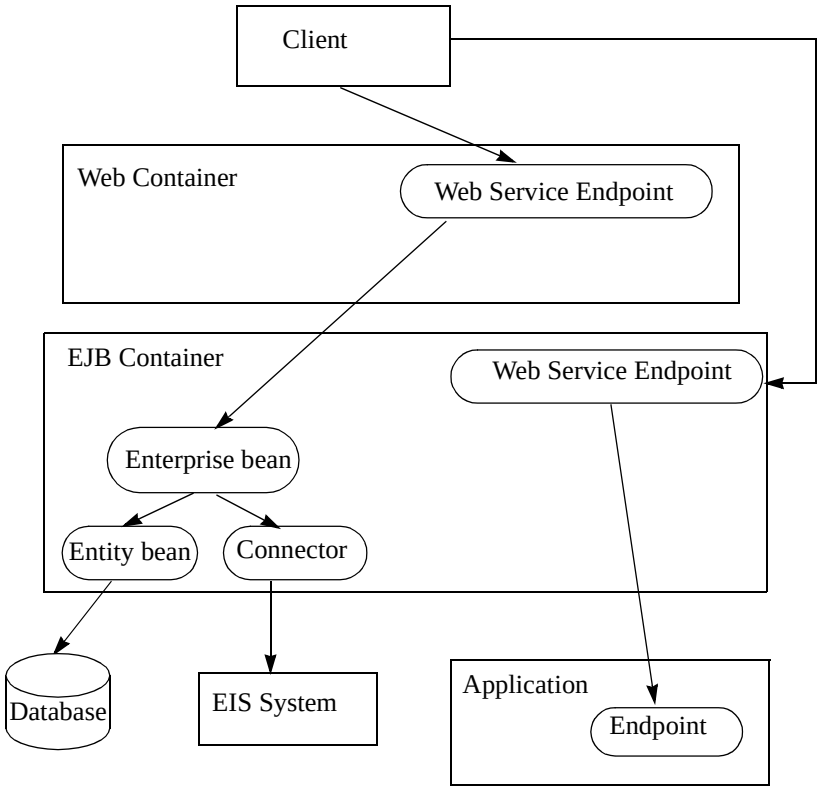

Figure 7.1 Anatomy of a Web Service Interaction

Most client requests to a service require the service to access a series of components to fulfill the request. This results in a chain of calls to various compo-

nents, some of which may be within the initiating component's security domain and others of which are outside that security domain. (A *security domain* is a general term for an environment in which a set of entities are presumed to trust one another.) With such a chain of component calls, each cooperating component in the chain must be able to negotiate its security requirements. In addition, components along the chain may use different security protocols. In short, security needs to flow from a client to a called component, then to other components and resources, while passing through different security policy domains.

A J2EE application must be able to integrate its own security requirements and mechanisms with those of different components and systems. For example, a client may make a request to a Web service. The client call is to an endpoint, which in turn may call other Web services, make IIOP calls, access resources, and access local components. Each component—other Web services, local and remote components, and resources—has its own security requirements. If it interacts with an EIS system, a Web service endpoint must be able to handle the security requirements and mechanisms that the EIS system requires for authentication and authorization.

Thus, some of the common security requirements for a Web service are authentication, access control, establishing a secure channel for exchanging messages, message-level security, and securing the interaction with other components when processing requests. Let's examine how these security requirements express themselves with Web services.

7.1.1.1 Authentication

Authentication, or proving one's identity, is often required by both a Web service and a client for an interaction to occur. A Web service may require that clients provide some credentials—such as a user name, password, or a digital or X.509 certificate—to prove their identity. The client of a Web service may require that a service provide it with proof of its identity, which typically is a digital certificate.

Furthermore, since a Web service may need to access other components and resources to process a client's request, there are authentication requirements between a service and resources that it uses. The service may need to provide identity information to authenticate itself to resources and components. The resources and components may also have to prove their identity to the service. The same authentication requirements hold true between Web services if the service endpoint needs to access other Web services.

Thus, authentication occurs across different layers and different types of systems and domains. Passing principal identity along the chain may also require that the identity change or be mapped to another principal.

7.1.1.2 Access Control

Controlling access to a service is as important as authentication. A service endpoint may want to let only certain clients access its services. Or, an application may want to restrict different sets of its resources and functionality to different groups of clients. An endpoint may allow all clients to invoke its basic service, but it may grant some clients extra privileges and access to special functions. In short, all clients are not equal in terms of their permissions to access or use services or resources.

Because a service endpoint also needs to interact with other components, such as enterprise beans and resources, the endpoint needs some way to control access to those resources. For example, you may want to limit access to only users who are classified as managers or to only users who work for a particular department. That is, the endpoint needs to be able to specify resources that have restricted access, to group clients into logical roles and map those roles to an established identity, and, while processing a service request, to decide if clients with a particular identity can access a particular resource.

7.1.1.3 Secure Channel for Message Exchange

A client's utilization of a Web service entails numerous message exchanges, and such messages may be documents, input parameters, return values, and so forth. Since not all messages require security, an application needs to identify those messages requiring security and ensure that they are properly protected.

Some message exchanges, such as passing credit card information, require confidentiality. For these messages, the interaction between a client and a Web service must be encrypted so that unintended parties, even if they manage to intercept the message, cannot read the data.

Interactions between a client and a Web service may require integrity constraints. That is, message exchanges between a client and a service may require a digital signature to verify that the message was not altered in transit. The message recipient, by applying a signature to a message, verifies the integrity of the message.

To handle interactions requiring integrity and confidentiality, it is important to establish secure channels for exchanging messages. Applications use HTTPS and

digital certifications to establish such secure channels. HTTPS provides a secure message exchange for one hop between two parties.

7.1.1.4 Message-Level Security

Some Web service message exchanges may require that security information be embedded within the message itself. This is often the case when a message needs to be processed by several intermediary nodes before it reaches the target service, or when a message must be passed among several services to be processed.

Message-level security can be an issue when XML documents are the form of the message, since different sections of the XML document may have different security requirements or be intended for different users.

7.1.2 Security Implications of the Operational Environment

The operational environment within which Web services interactions occur is an important factor in your security design. Service interactions occurring entirely within an enterprise have very different security requirements than service interactions open to everyone on the Internet. Thus, the relationship among Web service participants—such as Internet, intranet, and extranet—is an important consideration. When participants are closely aligned, you have a greater ability to negotiate security requirements.

In essence, the more control you have over the environment in which the Web service participants run, the easier it is to solve your security design. For example, if the Web services limit communication to applications inside your enterprise, then the network's physical security shields the Web service. The operational environment security may be sufficient to satisfy your security needs. Similarly, Web services in environments that require communication via a Virtual Private Network (VPN) need not worry about confidentiality, since the communication channel is already secure.

When all participants are trusted, such as within one enterprise, it is an easy matter to set up and exchange security keys. However, this is a difficult challenge for untrusted participants with an open Internet Web service.

7.1.3 Example Credit Card Processing Service

Let's see how these general security requirements, apply to the Adventure Builder application example discussed in Chapter 1. The Adventure Builder Web site, prior to adding Web services, allowed its affiliated credit card service to expose its func-

tionality—validating and processing credit card purchases—through HTTP. The credit card service wants to expose its functionality through a SOAP-based Web service interface and retain its existing security setup.

Let's first look at the new requirements of the Adventure Builder to credit card service interaction. Since both client and service want proof of identity, the interaction requires mutual authentication. That is, both parties in the message exchange authenticate themselves to each other. Since exchanged messages contain sensitive financial data, an encrypted channel is needed to maintain confidentiality and integrity.

Since a credit card service client can be another application, such as a travel service processing the payment portion of an order it is fulfilling, this potentially can be an application to application interaction. Many different client applications can use the credit card service, but none of its clients may be anonymous. The credit card service may also want to allow for different categories or levels of clients, giving some clients special services and perhaps charging these clients higher fees. There needs to be a way to restrict access to special services to only authorized users. The credit card service might have to set up different roles for different users, for example differentiating between Visa and MasterCard users.

The credit card service might have multiple endpoints, such as a second endpoint for online registration. It might also provide a tracking service for store administrators to check usage and fees.

Not only does the credit card service need to securely transport messages, the service endpoint needs to access other resources—such as a database, an EIS, external banks—to process requests. It needs to interact with the security systems of these other resources. The credit card service endpoint must also interact with the existing credit card application components for its processing. As a result, it must deal with the existing security policies of these components.

Finally, the credit card service must keep a record of transactions to prevent any non-repudiation of services by customers.

7.2 J2EE Platform Security Model

The J2EE platform container provides a set of security-related system services at deployment and runtime to its applications and clients. These built-in container services simplify application development because they remove the need for the application developer to write the security portion of the application logic.

Security on the J2EE platform can be either declarative or programmatic. *Declarative security* mechanisms used in an application are expressed via a declarative syntax in a document called a deployment descriptor. An application deployer employs container-specific tools to map the application requirements that are in a deployment descriptor to security mechanisms in J2EE. The declarative security model has the advantage of enabling you to change declarative settings to match security policy. *Programmatic security* refers to security decisions that are made by security-aware applications. Programmatic security, which allows an application to include code that explicitly uses a security mechanism, is useful when declarative security alone cannot sufficiently express the security model of an application. The J2EE programming model offers some programmatic services that help you to write security functionality into the application code.

Declarative references in the deployment descriptor, rather than program code, define much of the J2EE application security. The collection of security declarations forms the security policy for an application. When security is defined declaratively, the container is responsible for performing security and the application does not include code specifically for security operations. Since security references are in the deployment descriptor, developers can modify the security for an application by using tools or changing the deployment descriptor. At deployment, the container uses the declarative references in the deployment descriptor to set up the security environment for the J2EE application, just as it uses other references in the deployment descriptor to perform similar services for transactions, remote communication, and so forth. During runtime, the container interposes itself between the applications and the platform to perform security checks and otherwise manage the applications and support the distributed object protocols.

The J2EE platform security model gives you the ability to provide boundaries for security domains. Once you have established these security domains, you can map users to identities and combine users into logical groups according to their roles within the organization.

As noted, rather than inventing new security mechanisms, the J2EE platform facilitates the incorporation of existing security mechanisms into an application server operational environment. That is, the J2EE security model integrates with existing authorization mechanisms, handling existing user identity information, authorization certificates, and so forth. The model provides a unifying layer above other security services, and its coherent programming model hides much of the security implementation details from application developers. For example, the J2EE security model links into the security mechanisms of data resources. It thus lets you use the username and password security provided by a database system.

Let's look in more detail at the J2EE platform security services and mechanisms. This security model applies to Web services as well as the entire J2EE platform. "Security for JAX-RPC Interactions" on page 286 describes how a Web service application can leverage these J2EE mechanisms.

7.2.1 Authentication

Authentication is the mechanism by which clients and service providers prove to one another that they are acting on behalf of specific users or systems. When the proof occurs in two directions—the caller and service both prove their identity to the other party—it is referred to as mutual authentication.

Typically, a client interaction with a J2EE application accesses a set of components and resources (including JSPs, enterprise beans, data bases, and Web service endpoints). When these resources are protected, as is often the case, authorization rules restrict a client's access to the resources. A client presents its identity and credentials and the container determines if the client meets the criteria for access specified by the authorization rules.

Since processing a request may require a chain of calls to access other resources and components, the J2EE platform allows the client identity established with the initial call's authentication to be associated with subsequent method calls and interactions. That is, the client's authenticated identity can be propagated along the chain of calls.

The J2EE platform has a mechanism by which a component along a chain of calls can change the authenticated identity to its own identity. The platform also allows lazy authentication, which allows unauthenticated clients to access unprotected resources but forces authentication when these clients try to access protected resources.

The platform additionally permits authentication to occur at different points, such as on the Web or EJB tier. The J2EE container handles the authentication based on the requirements declared in the deployment descriptor.

7.2.1.1 Protection Domains

The J2EE platform makes it possible to group entities into special domains, called protection domains, so that they can communicate among themselves without having to authenticate themselves. 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 7.2 illustrates an environment using protection domains. It shows how authentication is required only for interactions that cross the boundary of a protection domain. Interactions that remain within the protection domain do not require authentication. Although authentication is not required within this realm of trust, there must be some means to ensure that unproven or unauthenticated identities do not cross the protection domain boundary. In the J2EE architecture, a container provides an authentication boundary between external callers and the components it hosts. Furthermore, the architecture does not require that the boundaries of protection domains be aligned with the boundaries of containers. The container's responsibility is to enforce the boundaries, but implementations are likely to support protection domains that span containers.

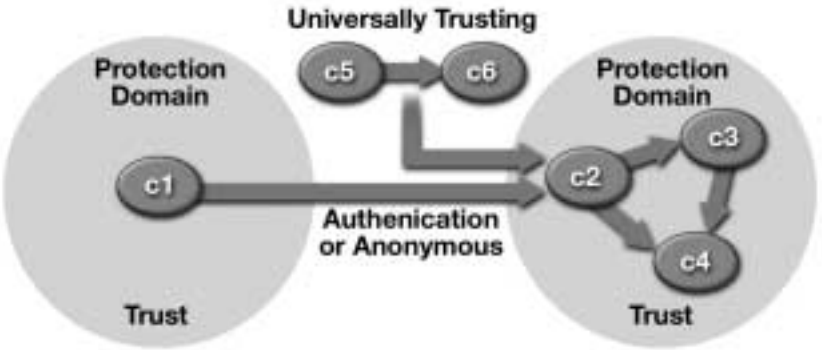

Figure 7.2 Protection Domain Established by Authentication Boundaries

The container ensures that calls entering the domain can authenticate their identity, which they usually do with a credential (such as an X.509 certificate or a Kerberos service ticket). A credential such as this is analogous to a passport or driver's license. The container also ensures that outgoing calls are properly identified. Maintaining proper proof of component identity makes it easier for interacting components to trust each other. A J2EE developer can declaratively specify the authentication requirements of an application for calls to its components (such as enterprise beans or JSPs) and for outbound calls that its components make to access other components and resources.

Maintaining the protection boundary integrity requires exposing the authentication boundaries. The deployment descriptor holds declarations of the references made by each J2EE component to other components and to external resources. These declarations, which appear in the descriptor as `ejb-ref` elements,

`resource-ref` elements, and `service-ref` elements, indicate where authentication may be necessary. The declarations are made in the scope of the calling component and they serve to expose the application's inter-component or resource call tree. Deployers use J2EE platform tools to read these declarations, and they can then use these references to properly secure interactions between the calling and called components. The container uses this information at runtime to determine if authentication is required and to provide the mechanisms for handling identities and credentials.

7.2.1.2 Web Tier Authentication

Developers can specify that authentication be performed on the Web tier when certain components and resources are accessed, in which case it is handled by the J2EE Web container. J2EE Web containers must support three different authentication mechanisms:

- HTTP basic authentication—The Web server authenticates a principal using the user name and password obtained from the Web client.
- Form-based authentication—A developer can customize a form for entering username and password information, and then use this form to pass the information to the J2EE Web container.
- HTTPS mutual authentication—Both the client and the server use recognized certificates (X.509) to establish their identity, and authentication occurs over a channel protected by Secure Sockets Layer (SSL).

In addition, Web containers may support HTTP digest authentication, and they are encouraged to do so. With digest authentication, a Web client authenticates itself to a Web server with a message digest containing the client password that it sends along with its HTTP request message.

Generally, for Web tier authentication, the developer specifies an authorization constraint to designate those Web resources—such as HTML documents, Web components, image files, archives, and so forth—that need to be protected. When a user tries to access a protected Web resource, the Web container applies the particular authentication mechanism (either basic, form-based, or mutual authentication) specified in the application's deployment descriptor.

It is important to note that J2EE Web containers provide single sign-on among applications within a security policy domain boundary. Clients often make many

requests to an application within a session. At times, these requests may be among different applications. In a J2EE application server, when a client has authenticated in one application, it is also automatically authenticated for other applications for which that client identity is mapped. Web containers allow the login session to represent a user for all applications that the user can access without requiring the user to re-authenticate for each application. However, this type of single sign-on does not span multiple security domains. Other specifications, such as the Liberty Alliance Standard (`http://www.projectliberty.org`), cover single sign-on that spans multiple security policy domains.

7.2.1.3 EJB Tier Authentication

Authentication can also be performed directly on the EJB tier, or a component on a different tier can perform the authentication for the EJB tier. Several use case scenarios describe these situations.

One common use case scenario involves a Web tier component that receives a user request sent to it over HTTP. To handle the request, the Web component calls an enterprise bean component on the EJB tier, a typical scenario since many Web applications use enterprise beans. In these cases, the application developer places a Web component in front of the enterprise bean and lets the Web component provide the authentication.

Often, too, the network firewall technology in place in many environments prevents direct interaction, such as with RMI, between client containers and enterprise beans. When the firewall prevents such direct interaction, the EJB container may rely on the Web container's authentication mechanisms and network accessibility capabilities. Thus, the Web container vouches for the identity of those users who want to access enterprise beans, and these users access the beans via protected Web components. Figure 7.3 illustrates how a system can be configured to use the Web container to enforce protection domain boundaries for Web components, and, by extension, for the enterprise beans called by the Web components.

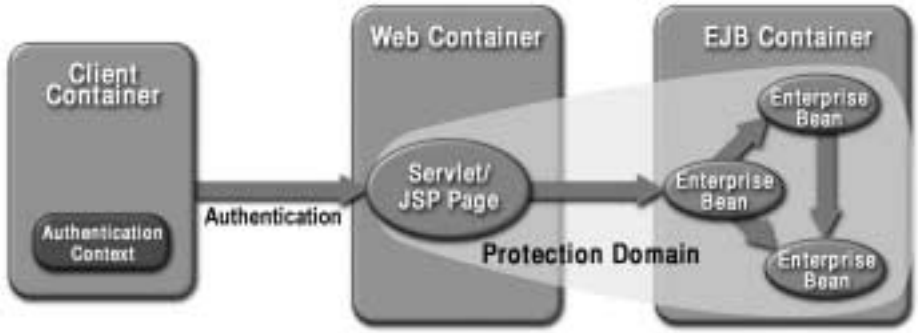

Figure 7.3 Using the Web Container to Establish an EJB Tier Protection Domain

In another use case scenario, calls can be made directly to an enterprise bean using RMI-IIOP. The Common Secure Interoperability (CSIv2) specification, which is an Object Management Group (OMG) standard supported by the J2EE platform, defines a protocol for secure RMI-IIOP invocations. Using the CSIv2-defined Security Attribute Service (SAS), client authentication is enforced just above the transport layer. SAS also permits identity assertion, which is an impersonation mechanism, so that an intermediate component can use an identity other than its own.

A possible third use case involves sending a SOAP request to an EJB service endpoint. In this case, the Web service authentication model handles authentication using similar mechanisms—basic authentication and mutual SSL—to the Web tier component use case. However, rather than use a Web component in front of the EJB component, the EJB programming model handles the authentication.

7.2.1.4 Client Identity Selection

J2EE components have an invocation identity, established by the container, that identifies them when they call other J2EE components. The container uses either the `run-as(role-name)` or `use-caller-identity` identity selection policy defined in the deployment descriptor to establish this invocation identity. The container uses a static identity selected by the deployer from the principal identities mapped to the named security role.

Only J2EE components that are not Web service endpoints handle caller identity as described here. When a component calls a Web service endpoint using

JAX-RPC, reauthentication is required since caller identity is not propagated. See “Propagating Component Identity” on page 293.

Developers can define component identity selection policies for J2EE Web and EJB resources. You should associate a `use-caller-identity` policy with component callers that should be held accountable for their actions. Using the `run-as(role-name)` identity selection policy does not maintain the chain of traceability and may be used to afford the caller with the privileges of the component. Code Example 7.1 shows how to configure client identity selection policies in an enterprise bean deployment descriptor.

```
<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>
```

Code Example 7.1 Configuring Enterprise Bean Identity Selection Policies

Code Example 7.2 shows how to configure client identity selection policies in Web component deployment descriptors. If `run-as` is not explicitly specified, the `use-caller-identity` policy is assumed.

```
<web-app>
    <servlet>
        <run-as>
```

```
            <role-name> guest </role-name>
        </run-as>
        ...
    </servlet>
    ...
</web-app>
```

Code Example 7.2 Configuring Identity Selection Policies for Web Components

Applications can also use various programmatic APIs to check client identity. For example, a servlet component can use the `getUserPrincipal` method on the `HttpServletRequest` interface. An enterprise bean component can use the `EJBContext` method `getCallerPrincipal`. An application can use these methods to obtain information about the caller, and then pass that information to business logic or use it to perform custom security checks. Keep in mind, however, that there is no way to programmatically change client identity; changing identity can only be done declaratively with the `run-as` element.

7.2.1.5 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, you can configure the calling container to manage for the calling component the authentication to the resource, a form of authentication called *container-managed resource manager sign-on*. The J2EE architecture also recognizes that some components need to directly manage the specification of caller identity and the production of a suitable authenticator. For these applications, the J2EE architecture provides a means for an application component to engage in what is called *application-managed resource manager sign-on*. Use application-managed resource manager sign-on when the ability to manipulate the authentication details is fundamental to the component's functionality.

The `resource-ref` elements of a component's deployment descriptor declare the resources used by the component. The value of the `res-auth` subelement declares whether sign on to the resource is managed by the container or application. Components that manage resource sign on can use the `EJBContext.getCallerPrincipal` or `HttpServletRequest.getUserPrincipal` methods to obtain the identity of their caller. A component can map the identity of its caller to a new identity or authentication secret as required by the target enter-

prise information system. With container-managed resource manager sign-on, the container performs *principal mapping* on behalf of the component.

Care should be taken to ensure that access to any component that encapsulates or is provided by its container with a capability to sign on to another resource is secured by appropriate authorization rules.

The Connector architecture offers a standard API for application-managed resource manager sign-on. The Connector-provided API ensures portability of components that authenticate with enterprise information systems.

7.2.2 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. You can establish caller identity by selecting from the set of authentication 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. Anonymous callers use a special credential. These attributes uniquely identify the caller in the context of the authority that issued the credential. Depending on the type of credential, it may contain other attributes that define shared authorization properties (such as group memberships), which distinguish collections of related credentials. The identity and shared authorization attributes in the credential are the caller's *security attributes*. In the J2SE platform, the caller's security attributes may also include the identity attributes of the code used by the caller. Comparing the caller's security attributes with those required to access the called component determines access to the called component.

In the J2EE architecture, a container serves as an authorization boundary between the components it hosts and their callers. 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 container allows the call; otherwise, it rejects the call.

The J2EE application programming model focuses on permissions, which indicate who can do what function. In the J2EE architecture, the deployer maps

the permission model of the application to the capabilities of users in the operational environment.

7.2.2.1 Declarative Authorization

The deployer establishes the container-enforced access control rules associated with a J2EE application. The deployer uses a deployment tool to map an application permission model, which is typically supplied by the application assembler and defined in the deployment descriptor, to policy and mechanisms specific to the operational environment.

The deployment descriptor defines logical privileges called *security roles* and associates them with components. Security roles are ultimately granted permission to access components. The deployer assigns logical privileges to specific callers, based on the values of their security attributes, to establish the capabilities of users in the runtime environment. For example, a deployer might map a security role to a security group in the operational environment. As a result, any caller whose security attributes indicate membership in the group is 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. Callers authenticated by one of these identities are 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, rather than during 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 kept to the scope of the application. Thus, the mapping applied to the components of one application may be different from that of another application.

A client typically uses a J2EE application's container to interact with enterprise resources in the Web or EJB tiers. These resources may be protected or unprotected. Protected resources have authorization rules defined in deployment descriptors that restrict access to some subset of non-anonymous identities. To

access protected resources, users must present non-anonymous credentials to enable their identities to be evaluated against the resource authorization policy.

You control access to Web resources by properly defining their access elements in the deployment descriptor. For accessing enterprise beans, you define method permissions on individual bean methods. See "Handling Authorization" on page 296.

7.2.2.2 Programmatic Authorization

A J2EE container decides access control before dispatching method calls to a component. The logic or state of the component is not a factor in these 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 within the component code.

To use these functions, a component must declare the complete set of distinct `roleName` values used in all 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. Ultimately, the deployer 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.

7.2.3 Confidentiality and Integrity

Confidentiality mechanisms ensure private communication between entities by encrypting the message content so that a third party cannot read it. *Integrity mechanisms* ensure that another party cannot tamper with communication between entities; in particular, that a third party cannot intercept and modify communications. Integrity mechanisms can also ensure that messages are used only once. Attaching a *message signature* to a message ensures message integrity: the modification of the message by anyone other than the caller will be detectable by the receiver.

The deployer configures the containers to apply confidentiality and integrity mechanisms. The application assembler supplies the deployer with information about the components that should be protected. The deployer then configures the corresponding containers to employ the needed confidentiality and integrity mechanisms whenever interactions with these components occurs over 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 is not. Message integrity may be verified as a side effect of enforcing confidentiality.

The J2EE platform requires that containers support transport layer integrity and confidentiality mechanisms based on Secure Sockets Layer and Transport Layer Security (SSL/TLS) so that security properties applied to communications are established as a side effect of creating a connection. SSL/TLS can be specified as requirements for Web components and EJB components, including Web service endpoints.

The deployment descriptor conveys information to identify those components with method calls whose parameters or return values should be protected. For enterprise beans, this is done in a `description` subelement of the target EJB component. For servlets and JSPs, this is done in the `transport-guarantee` subelement of the `user-data-constraint` subelement of a `security-constraint`. When 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`.

7.3 Security for JAX-RPC Interactions

Developers that rely on JAX-RPC to exchange messages between Web service endpoints and clients leverage the security services provided by the J2EE platform. Let's begin with a closer look at how the J2EE platform implements the WS-I Web services security mechanisms and requirements.

The J2EE Web service security model, which is based on the J2EE platform security model, encompasses mechanisms for securing Web service interactions. This Web service security model shares some of the J2EE platform's authentication and authorization mechanisms, particularly servlet and enterprise bean mechanisms. Similar to the platform, the Web service security model relies on HTTPS to maintain integrity and confidentiality from connection to connection, or hop to hop. Due to restrictions and other limitations, however, the Web service security model provides a subset of the platform's overall security capabilities.

The Web service security model, similar for enterprise bean or Web-based endpoints, supports the interoperability and security standards specified by the WS-I Basic Profile 1.0. With Web services, it is important that both the request and the reply are secure. The model meets current security requirements for WS-I compliance—it covers single hop security for a request and reply between the client and server, and pertains just to the transport layer (HTTP or HTTPS)—and

combines this with additional mechanisms in the J2EE platform, such as basic and mutual authentication.

7.3.1 Endpoint Programming Model

Let's first look at the endpoint programming model and see how to design and implement a secure Web service on the J2EE platform; that is, how to authenticate and establish a secure HTTPS interaction.

The J2EE platform's security services enable you to implement a secure Web service without writing your own security code. As with any J2EE component, you can use declarative mechanisms to define the security for a service. Similarly, you also have the option of including programmatic security mechanisms in your Web service endpoints.

The key requirements for a Web service interaction are authentication and establishing a secure SSL channel for the interaction. Identity propagation and authorization are also important requirements, as is publishing and discovery of a service's security policy.

7.3.1.1 Setting Up a Secure Transport Layer

SSL/TSL is a key technology in Web service interactions, and it is important to understand how to properly set up SSL and authenticate users. (Although supported by the J2EE platform, the form-based login option for authentication is not available for Web service endpoints.)

SSL is a standard mechanism for Web services that is available on virtually all application servers. This widely-used, mature technology, which both secures the communication channel between client and server and supports client and server authentication, can satisfy most use cases for secure Web service communications. Since it works at the transport layer, SSL covers all information passed in the channel as part of a message exchange between a client and a service, including attachments.

The J2EE platform supports the following use cases with SSL:

- The server authenticates itself with SSL and the service's certificate is available.
- The client uses basic authentication over an SSL channel.
- Mutual authentication with SSL, using the server certificate as well as the client certificate, so that both parties can authenticate to each other.

With Web services, the interaction use case is machine to machine; that is, it is an interaction between two application components with no human involvement. Machine-to-machine interactions have a different trust model from typical Web site interactions. In a machine-to-machine interaction, trust must be established proactively, since there can be no real-time interaction with a user about whether to trust a certificate. Ordinarily, when a user interacts with a Web site via a browser and the browser does not have the certificate for the site, the user is prompted about whether to trust the certificate. The user can accept or reject the certificate at that moment. With Web services, the deployers involved in the Web service interaction must distribute the server certificate, and possibly the client certificate if mutual authentication is required, prior to the interaction occurrence.

The mechanism for implementing a Web service endpoint that requires SSL varies according to the endpoint type. For a Web tier endpoint (a JAX-RPC service endpoint), you indicate you are using SSL by setting the `transport-guarantee` element in the `web.xml` deployment descriptor to `CONFIDENTIAL`. This setting enforces an SSL interaction with a Web service endpoint. (See Code Example 7.3.)

```
<user-data-constraint>
  <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
```

Code Example 7.3 Using SSL in a Web Tier Endpoint

You can also require a client of a service endpoint to authenticate with a client certificate. To do so, set the `auth-method` element to `CLIENT-CERT` in the same deployment descriptor. (See Code Example 7.4.)

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

Code Example 7.4 Requiring Client Authentication

The combination of these two settings (`CONFIDENTIAL` for `transport-guarantee` and `CLIENT-CERT` for `auth-method`) enables mutual authentication. When set to these values, the containers for the client and the target service both

provide digital certificates sufficient to authenticate each other. (These digital certificates contain user-specific identifying information.)

Setting up SSL for EJB tier endpoints varies according to the particular application server. Although EJB endpoints are required to support SSL and mutual authentication, the specifications have not defined a standard, portable mechanism for enabling this. The deployer must follow application server-specific mechanisms to make an EJB endpoint require basic authentication. In many cases, there are application server-specific deployment descriptor elements for EJB endpoints that are similar to the `web.xml` elements.

7.3.1.2 Implementing Basic Authentication

All J2EE-compliant application servers provide basic authentication, the standard security mechanism for J2EE Web services. With basic authentication, a Web service endpoint requires a client to authenticate itself. The client does so by providing a username and password. Basic authentication, referred to as `BASIC` in the descriptor, requires the name of a user realm.

As with SSL, the type of the Web service endpoint determines the mechanism for requiring basic authentication for the service. For a Web tier (JAX-RPC) service endpoint, set the `auth-method` element for the login configuration (`login-config`) in the `web.xml` deployment descriptor to `BASIC`, as shown here:

```
<web-app>
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>somerealmname</realm-name>
    </login-config>
</web-app>
```

Code Example 7.5 Basic Authentication Mechanism for Web Tier Endpoints

For a Web service with an EJB endpoint, the deployer uses the application server-specific mechanisms to require basic authentication. Often, each application server's deployment descriptor includes an element for authentication for an EJB service endpoint that is analogous to the `web.xml` `auth-method` element.

7.3.1.3 Setting Hybrid Authentication

A Web service may also require hybrid authentication, which is when a client authenticates with basic authentication and SSL is the transport. Enabling hybrid authentication requires setting the `transport-guarantee` element to `CONFIDENTIAL` and the `login-config` `auth-method` element to `BASIC`.

When setting authentication requirements for a client, keep in mind that an endpoint can require a client to authenticate by either supplying a username and password or with a digital certificate. An endpoint cannot require a client to use both mechanisms.

Hybrid authentication compensates for HTTP basic authentication's inability to protect passwords for confidentiality. This vulnerability can be overcome by running the authentication protocols over an SSL-protected session, essentially creating a hybrid authentication mechanism. The SSL-protected session ensures confidentiality protection for all message content, including the client authenticators. Code Example 7.6 demonstrates how to configure HTTP basic authentication over SSL using the `transport-guarantee` element.

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

Code Example 7.6 SSL Hybrid Authentication Mechanism

When deploying an application that uses this type of hybrid authentication mechanism, it is important to properly set the security elements of the Web resource's deployment descriptor.

- The application deployer should ensure that the `transport-guarantee` element of each protected Web resource is set to `CONFIDENTIAL` for an application using

a hybrid authentication mechanism. Otherwise, the client authenticator is not fully protected.

7.3.1.4 Publicizing Security Policy

Just as it needs to describe its methods and related information in a WSDL document, a Web service also needs to describe its security policy and make that information available to clients. Without this information, clients cannot interoperate with the service.

At the present time, a WSDL description contains just minimal information about the security characteristics of an endpoint—just the HTTPS location specified in the endpoint URL. (The WSDL specification requires the location attribute of the `soapbind:address` element in its `wsdl:port` description to be a URI whose scheme is `https`.) See Code Example 7.7.

```
<service name =”SomeService”>
<port name=”SomeServicePort” binding=”tns:SomeServiceBinding”>
<soap:address location=”https://myhostname:7000/adventurebuild-
er/opc/getOrderDetails”/>
</port>
</service>
```

Code Example 7.7 WSDL Security Description

Currently, the WSDL description has no standard mechanism for indicating if an endpoint requires basic or mutual authentication. Such information needs to be made available through service-level agreements between the client and endpoint. (It is expected that later versions of the WSDL description will be extended to include descriptions of endpoint security requirements, perhaps by using meta-data or annotations similar to CSIv2.)

Since WSDL is so limited, you need to ask yourself what to do now to define security policies for endpoints. Do you provide some meta-data somewhere, make some security assumptions among partners, include security descriptions as a non-standard part of JAXR entries, or somehow extend the WSDL description yourself? Not only that, your application and its endpoints may have built-in implicit assumptions, and it may need to provide a description of its own unique security requirements for its endpoints. Some of these requirements may be negotiable.

- It is recommended that you list security assumptions and requirements in the description elements that are part of a service component's deployment descriptor.
- In addition, have a separate document available for endpoint developers describing the security policy for an endpoint. Clearly document the information needed by a client in this document.

7.3.2 Client Programming Model

In addition to implementing security for an endpoint, client developers must also implement security. You implement security according to the type of client. We focus on J2EE clients, also called managed clients, and these include enterprise bean and servlet components. J2EE clients can take advantage of the J2EE platform mechanisms when interacting with a Web service endpoint. You design and implement security for J2EE clients the same way regardless of whether they interact with Java-based or non-Java-based Web services.

Other types of clients, such as non-Java or standalone J2SE clients, generally cannot use the services of the J2EE platform. (With one exception—standalone J2SE clients can use the JAX-RPC technology if they include the JAX-RPC runtime in their standalone environment.) If you develop a standalone J2SE client to be a Web service client, keep in mind that the J2SE platform provides its own set of services and tools to help you. You can use the Java Authentication and Authorization Service (JAAS), along with tools such as `keytool`, to manage certificates and other security artifacts. As noted, you can also include the JAX-RPC runtime, use its mechanisms to set up username and password properties in the appropriate stub files, and then make calls to the Web service. It is important to have your client follow the WS-I interoperability requirements, since doing so ensures that your client can communicate with any Web service endpoint.

7.3.2.1 Implementing Security for J2EE Clients

The J2EE container provides support so that J2EE components, such as servlets and enterprise beans, can have secure interactions when they act as clients of Web service endpoints. The container provides this support regardless of whether or not the Web service endpoint is based on Java. Let's look at how J2EE components use the JAX-RPC client APIs to invoke Web service endpoints in a secure manner.

As indicated in the section "Endpoint Programming Model" on page 287, a target endpoint defines some security constraints or requirements that a client

must meet. For example, the client's interaction with the service endpoint may require basic authentication or HTTPS, or the client must provide certain information to access the endpoint.

The first step for a client is to discover the security policy of the target endpoint. Since the J2EE platform does not mandate how the WSDL should describe security requirements, this discovery process varies with each situation. Once it knows the security requirements for interacting with the service, the client sets up its environment to make available the appropriate artifacts. For example, if the Web service endpoint requires basic authentication, the client must place its username and password identifying information in the HTTP headers. If the endpoint requires mutual authentication, the client deployer must establish the proper certificates and make them available to the J2EE container, by correctly setting up the application server instance environment.

For basic authentication, the deployer sets application server-specific deployment descriptor elements that are used to set the client username and password. These vendor-specific deployment descriptors statically define at deployment the username and password identifier, since this identifier has no relation to the principal associated with the calling component at runtime. The container, at the time the JAX-RPC call is made, puts the username and password values into the HTTP header. Keep in mind that the specifications recommend *against* using programmatic JAX-RPC APIs to set the username and password properties on stubs for J2EE components. Thus, J2EE application servers are not required to support programmatically setting these identifier values.

For mutual authentication, the deployer sets up the appropriate digital certificates and makes them available to the container hosting the client component.

Once the environment is set, a J2EE component can make a secure call on a service endpoint in the same way that it ordinarily calls a Web service—it looks up the service using JNDI, sets any necessary parameters, and makes the call. (See Chapter 4 for details.) The J2EE container handles the security for the call using the digital certificate or the values specified in the deployment descriptor by the deployer.

7.3.3 Propagating Component Identity

Web service endpoints can be clients of other Web services and J2EE components. Any given endpoint may be in a chain of calls between components and Web service endpoints. Also, any non-Web service J2EE component can make calls to Web services.

A Web service endpoint follows the same rules as other J2EE components when it accesses J2EE components or resources. Thus, when an endpoint is a client of another J2EE component, the security considerations are the same as for any J2EE component. However, there are differences when an endpoint component calls another Web service endpoint.

To understand how clients set identity for Web service calls, let's first examine protection domains. A protection domain establishes an authentication boundary around a set of entities that are assumed to trust each other. Entities within this boundary can safely communicate with each other without authenticating themselves. Authentication is only required when the boundary is crossed. However, Web services are always considered outside of any protection domain.

- When making calls to any Web service, regardless of its location or relationship to your application, you must assume that the call to the service crosses an authentication boundary and design your security policies accordingly.

Keep in mind that Web service interactions differ from Web site interactions. Web sites may be designed to store session state, particularly when a user's browser session requires multiple calls to the Web site. Such design permits a user to authenticate just one time per session and not have to re-authenticate on subsequent calls during the session. With non-session-based Web services using JAX-RPC, authentication is valid for only one method call. You should design your application or component to assume that authentication is required for each call to a Web service endpoint.

How does a service endpoint establish its identity when it acts as a client calling another component? J2EE components can invoke other components under their own identity or, by using the `run-as` declarative identity property, they can set the caller identity to a different component. In general, endpoint components have this same capability. However, when an endpoint component calls a Web service, the container does not propagate the caller's security nor any `run-as` identity to the service. Thus, neither the `run-as` or caller identities determine the identity of a component calling a Web service. When an endpoint component uses

the `run-as` identity, the container applies that identity only to the component's interactions with non-Web service components, such as enterprise beans.

- Identity and security context are not propagated on calls to Web service endpoints. Each call to a Web service endpoint has its own identity and security context.

The target Web service establishes the identity of calls to its service endpoint. It bases this identity on the mapping principals designated by the service deployer, either the basic authentication username and password identity of the client or the digital certificate attributes supplied by the client's container. The `run-as` identity in the deployment descriptor does not affect the identity of a client that calls a Web service.

A Web service endpoint, once it has established an identity of a calling client, can propagate that identity (or its own identity, as the case may be) when making calls to other J2EE components that are not Web service endpoints. The same holds true for an endpoint's interactions with resources such as databases, queues, and so forth. Because a service endpoint functions like any J2EE component when it interacts with other non-Web service components and resources, it is easy to add a service endpoint in front of an existing application or system and integrate its security model.

For example, Figure 7.4 illustrates how a caller identifier is propagated from clients to Web service endpoints and J2EE components. The initial client makes a request of Web service endpoint X. To fulfill the request, endpoint X makes a call on entity bean J, which in turn invokes a method on entity bean K. The client caller identifier A propagates from the endpoint through both entity beans. However, when bean K calls a method on service endpoint Y, the identifier cannot be propagated and reauthentication must occur. Similarly, when endpoint X calls endpoint Z, the caller identifier cannot be propagated.

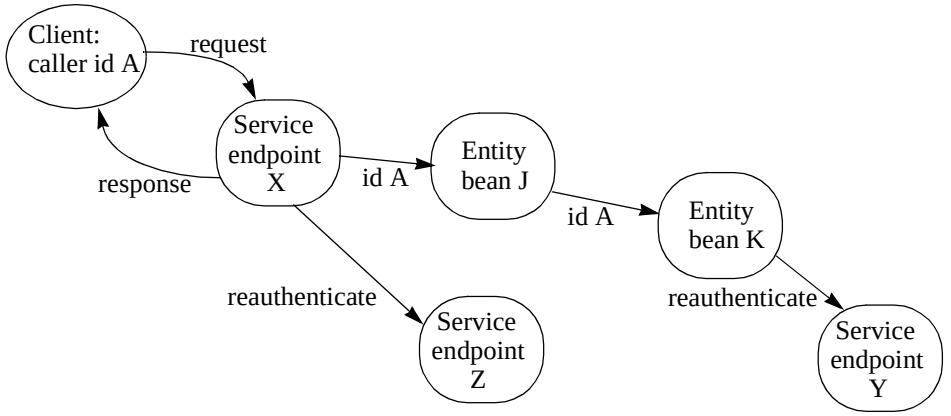

Figure 7.4 Security Propagation

7.3.4 Handling Authorization

Web service endpoints can restrict access to resources using the same declarative authorization mechanisms available to other J2EE components. From a security point of view, this capability facilitates integrating Web services with J2EE applications. When a Web service is called—and the calling client has been authenticated and its identity established—the container has the capability to check that the calling principal is authorized to access this service endpoint.

Furthermore, components and resources accessed by the Web service endpoint may have their own access control policies, and these may differ from the endpoint's policies. The service endpoint interacts with these components and resources in an identical manner as other J2EE components. A Web service is also free to leave its resources unprotected so that anyone can access its service.

Your access control policy may influence whether you implement the service as a Web tier or an EJB tier endpoint, since the different tiers have different access control mechanisms. (Conversely, the tier on which your endpoint resides determines how you specify access control.) With Web tier endpoint components, access control entails specifying an URL pattern that determines the set of restricted resources. For EJB tier endpoints, you specify access control at the method level, and you can group together a set of method names that you want protected.

What does this mean in terms of a Web service's security considerations? Let's consider permissions, for example. For Web tier components the granularity of security is specific to the resource and based on the URL for the resource. For EJB tier components, security granularity is at the method-scope level, which is a finer-grained level of control. Keep in mind, however, that both Web and EJB tier endpoints can use programmatic APIs for finer-grained security. If you are willing to write code for access control, then both types of endpoints have the same security capabilities as other Web and EJB tier components.

- ❒ If you require finer-grained control for your access control policy, consider using an EJB endpoint, since it utilizes method-level control.

7.3.4.1 Controlling Access to Web Endpoints

To control access to a Web endpoint, the deployer specifies a `security-constraint` element with an `auth-constraint` subelement in the Web deployment descriptor. Code Example 7.8 illustrates the definition of a protected resource in a Web component deployment descriptor. The descriptor specifies that only users acting in the role of `customer` can access the URL `/control/placeorder`.

```
<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 7.8 Web Resource Authorization Configuration

In addition to controlling access to Web components, an application can provide unrestricted access to unprotected resources, such as a JSP for browsing, by omitting an authentication rule. Omitting authentication rules allows unauthenticated users to access Web components.

7.3.4.2 Controlling Access to EJB Endpoints

A deployer, after defining security roles for an enterprise bean, can also specify via the `method-permission` elements the methods of the bean's remote, home, and local interfaces that each security role is allowed to invoke. Ultimately, the assignment of users to roles 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 7.9 shows how to configure method-level access. The example specifies that the method `submitOrder`, which occurs on an interface of an enterprise bean, requires that a caller belonging to the `customer` role must have authenticated to be granted access to it. If multiple methods have the same overloaded name, this example would grant the same access to all overloaded methods. It is possible to further qualify method specifications so as to identify methods with overloaded names by parameter signature, or to refer to methods of a specific interface of the enterprise bean. For example, you can specify that all methods of all interfaces (that is, remote, home, local, local home, and service) for a bean require authorization by using an asterisk (`*`) for the value in the `method-name` tag.

```
<method-permission>
    <role-name>customer</role-name>
    <method>
        <ejb-name>PurchaseOrder</ejb-name>
        <method-name>submitOrder</method-name>
    </method>
    <method>
    ...
</method-permission>
```

Code Example 7.9 Enterprise Bean Authorization Configuration

In addition to the authorization policy defined in `method-permission` elements, method specifications may be added to the `exclude-list` to indicate that access to them is to be denied independent of caller identity and whether the methods are the subject of a `method-permission` element. Code Example 7.10 demonstrates the use of the `exclude-list`.

```
<exclude-list>
    <method>
        <ejb-name>SpecialOrder</ejb-name>
        <method-name>*</method-name>
    </method>
    <method>
    ...
</exclude-list>
```

Code Example 7.10 Enterprise Bean Excluded `method-permission`

Some applications also feature unprotected enterprise beans and allow anonymous, unauthenticated users to access certain EJB resources. Use the `unchecked` element in the `method-permission` element to indicate that no authorization check is required.

Code Example 7.11 demonstrates the use of the `unchecked` element.

```
<method-permission>
    <unchecked/>
    <method>
        <ejb-name>Catalogue</ejb-name>
        <method-name>browseSpecials</method-name>
    </method>
    <method>
    ...
</method-permission>
```

Code Example 7.11 Enterprise Bean Unchecked `method-permission`

7.3.5 JAX-RPC Security Guidelines

In addition to the guidelines noted previously, the following general guidelines sum up the JAX-RPC authentication and authorization considerations.

- Apply the same access control rules to all access paths of a component. In addition, partition an application as necessary to enforce this guideline, unless there is some specific need to architect an application in a different fashion. 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. Be particularly careful that a less-protected access method does not undermine the policy enforced by a more rigorously protected method.

- Declarative security is preferable to programmatic security. Try to use declarative access control mechanisms since these mechanisms keep the business logic of the application separate from the security logic, thus making it easier for the deployer to understand and change the access policy without tampering with the application code. Generally, programmatic security is hard to maintain and enhance, plus it is not as portable as declarative security. Use programmatic mechanisms for access control only when extra flexibility is required.

- Bundle endpoints in `.war` files based as much as possible on authentication requirements. An application (deployed within an `.ear` file) may use multiple Web service endpoints. It is possible that you may require different authentication for these endpoints—some endpoints may require basic authentication, others may require a client certificate. Since a `web.xml` file can have only one type of authentication associated with its login configuration, you cannot put endpoints that require different authentication in a single `.war` file. Instead, group endpoints into `.war` files based on the type of client authentication they require. Because the J2EE platform permits multiple `.war` files in a single `.ear` file, you can put these `.war` files into the application `.ear` file.

- It is good practice to provide additional security policy descriptions to those that the standard WSDL file provides. The WSDL file is required to publish only a Web service's HTTPS URL. It has no standard annotation describing whether the service endpoint requires basic or mutual authentication. Use the description elements of the deployment descriptor to make known the security requirements of your endpoints.

- Be careful with the username and password information, because these properties can create a vulnerability if set when using stubs. Username and password are sensitive security data and the security of your system is compromised if they become known to the wrong party. For example, do not store username and password values in the application code or the deployment descriptor, and be sure to store the deployment descriptor in a secure manner.

- Consider using a "guarding" component between the interaction and processing layers. Set up an application accessor component with security attributes and place it in front of a set of components that require protection. Then, allow

access to that set of components only through the guarding or front component. A guarding component can make application security more manageable by centralizing security access to a set of components in a single component.

7.4 Message-Level Web Service Security

Traditional JAX-RPC security mechanisms may not be sufficient to manage the security requirements of all scenarios. However, new mechanisms and standards are emerging that allow securing Web services at the message level. Message-level security addresses the same security threats as more traditional mechanisms, including identity, authentication, authorization, and message signatures, encryption, and basic message exchange.

Message-level security applies to messages that are sent as documents. It makes security part of the document itself by embedding all required security information in a message's SOAP header. Message-level security also applies security mechanisms, such as encryption, to the message itself. With message-level security, the SOAP message itself either contains the information needed to secure the message or it contains information about where to get whatever is needed to handle security services. The SOAP message also contains the protocols and procedures for processing the specified message-level security. However, message-level security is not tied to any particular transport mechanism: since they are part of the message document data, the security mechanisms are independent of the transport mechanism.

JAX-RPC handles SOAP message exchanges with Web services, but it hides the details of these messages. To understand message-level security, it's necessary to examine a SOAP message in more detail. (See "Simple Object Access Protocol" on page 37 for more details about SOAP.) A SOAP message is composed of three parts:

- An envelope
- A header that contains meta information
- A body that contains the message contents

Figure 7.5 illustrates how security information is embedded at the message level. The diagram expands a SOAP header to show the header's security infor-

mation contents and artifacts related to the message. It also expands the body entry to show the particular set of elements being secured.

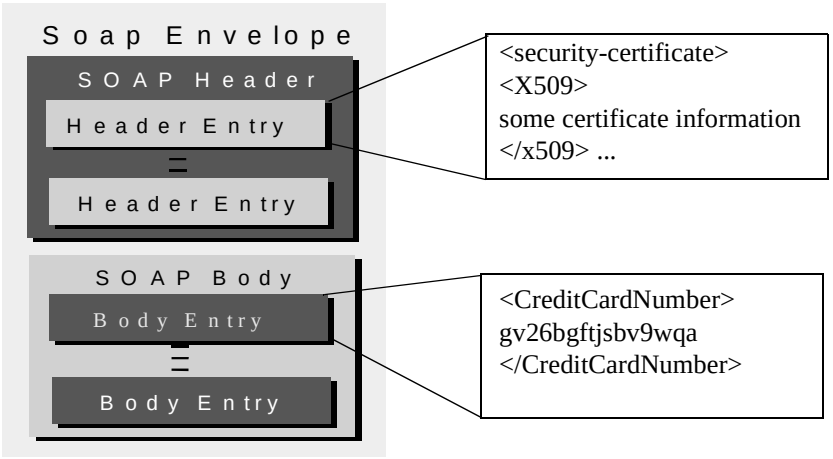

Figure 7.5 Embedding Security at the Message Level

The client adds security information that applies to that particular message to the SOAP message header. When the message is received, the service endpoint uses the header information to apply the appropriate security mechanisms and check that the message has not been tampered with. It is possible add signature and encryption information to SOAP headers, as well as other information such as security tokens—for example, an X.509 certificate—to a SOAP message.

In summary, message-level security technology lets you embed into the message itself a range of security mechanisms, including identity and security tokens and certificates and even message encryption and signature mechanisms. The technology associates this security information with the message and can process and apply the specified security mechanisms.

7.4.1 Benefits of Message-Level Security

Web service scenarios that involve passing documents or messages to multiple participants lend themselves to using message-level security. An application that needs to pass a message or document to numerous recipients often relies on the services of an intermediary between the sender and the recipient. The existence of such an

intermediary is often referred to as an end-to-end use case scenario, and it differs from what is called point-to-point, or peer-to-peer, use cases. Not only is it designed for handling end-to-end use cases, message-level security can provide security at a more granular level.

For end-to-end scenarios, a message does not go directly between a requestor (a client) and the service. Instead, the message may pass through one or several intermediaries as it progresses from the original requestor to the target service. With a point-to-point scenario, the message travels directly from the requestor to the receiver. These two types of scenarios have different security implications. Security technologies that handle point-to-point communication secure just the channel between two points. Such security is inadequate for an end-to-end scenario, since the transport mechanism security is not sufficient to protect the document or message during its subsequent transfers until it reaches its final destination. While the message may be secure when sent between two points, security falls short when the document is sent and received to subsequent points.

7.4.2 Scenarios for Using Message-Level Security

Why would a Web service use intermediaries and an end-to-end scenario, since it may pose added security constraints? Intermediaries can be useful in some Web service scenarios because they can provide value-added service. For example, they may validate parts of a document structure. An intermediary might also be used to determine a document's routing, accessing and modifying the document in between the document's various messaging hops.

A Web service that processes a purchase order document is a good example of an end-to-end use case scenario. Many participants are involved in the purchase order processing. However, each participant plays a distinct role and may be entitled to access only certain portions of the document. The application must pass the document to each participant, but, at the same time, it must limit a participant's access to only the appropriate portion of the document. This necessitates a more granular approach for security—that is, you want to secure just portions of the document.

In another scenario, although it may maintain numerous internal Web service endpoints, an application may expose only a single Web service endpoint through which it receives many different documents. The exposed service endpoint acts as a broker for all documents, while the internal endpoints actually perform the business processing on the received documents. For example, an order processing center may expose one Web service endpoint for all its supplier interactions. This

broker endpoints needs to view some identifying part of the incoming documents, but it may not be entitled to access all the information, particularly payment information, contained within the document. The broker endpoint may, for non-repudiation purposes, persist the purchase order along with its security information, but leave the actual order processing to the internal endpoints.

Web services that handle XML documents may also fall into the end-to-end use case category. XML documents are inherently granular since they are composed of a nested hierarchy of elements and subelements. It is easy to divide such a document into separate portions for processing, passing only the appropriate portion to some endpoint in the workflow. In the same way, you can use the document's granularity to apply different security to these different portions, as required. For example, a purchase order may need to encrypt only the credit card information. By encrypting only the credit card information, the order could pass among numerous participants—customer relations, call center, warehouse—but only the appropriate participants, such as a bank, could read the encrypted information.

7.4.3 Emerging Message-Level Security Standards

Since it is a new technology, there are a number of emerging standards for message-level security. These new specifications, which are part of OASIS and other standards bodies, concentrate on message-level security for XML documents. There are also new Java APIs just coming out to support these technologies.

The emerging specifications address security issues—such as identity, security tokens and certificates, authentication, authorization, encryption, message signing, and so forth—as they apply at the message level. For those who want to explore these emerging standards on their own, here is a partial list of the more significant JSRs and specifications:

- JSR 105—XML Digital Signatures for signing an XML document. It also includes procedures for computing and verifying signatures.
- JSR 106—XML Digital Encryption for encrypting parts of an XML document.
- JSR 155—Web Services Security Assertions, which is based on Security and Assertions Markup Language (SAML), is used for exchanging authentication and authorization information for XML documents.
- JSR 183—WS-Security provides a Java API for the WS-Security specification.

- XML Key Management Services (XKMS)—A protocol for distributing and registering public keys.
- XML Access Control Markup Language (XACML)

Let's look at how you might apply one of these message-level security mechanisms to a SOAP message, such as how you might put an XML digital signature on a purchase order document. A client first embeds the digital signature into the XML message, signing the message before sending it. The signature, which is an X.509 certificate, is embedded into the purchase order XML document along with other information. Code Example 7.12 shows the message header for this purchase order XML document with the embedded digital signature, per the XML Digital Signature specification.

```
<wsse:Security> <wsse:BinarySecurityToken ValueType="wsse:X509v3"
EncodingType="wsse:Base64Binary" Id="X509Token">
MIIEZzCCA9CgAwIBAgIQEmtJZc0rqrKh5i... </wsse:BinarySecurityToken>
<ds:Signature>
<ds:SignedInfo>
<ds:CanonicalizationMethod Algorithm= "ht-
tp://www.w3.org/2001/10/xml-exc-c14n#"/>
<ds:SignatureMethod Algorithm= "http://www.w3.org/2000/09/xmld-
sig#rsa-sha1"/>
<ds:Reference> <ds:Transforms>
<ds:Transform Algorithm= "http://...#RoutingTransform"/>
<ds:Transform Algorithm= "http://www.w3.org/2001/10/xml-exc-
c14n#"/>
</ds:Transforms> <ds:DigestMethod Algorithm= "ht-
tp://www.w3.org/2000/09/xmldsig#sha1"/>
<ds:DigestValue>EULddytSo1...</ds:DigestValue> </ds:Reference>
</ds:SignedInfo>
<ds:SignatureValue> BL8jdfToEb1l/vXcMZNNjPOV... </ds:SignatureVal-
ue>
<ds:KeyInfo>
<wsse:SecurityTokenReference> <wsse:Reference URI="#X509Token"/>
</wsse:SecurityTokenReference> </ds:KeyInfo>
</ds:Signature>
</wsse:Security>
```

Code Example 7.12 Embedding a Digital Signature in an XML Document

The code snippet shown in Code Example 7.12 is included in the message header for the message that has been digitally signed. The security portion of the header, which is part of the message itself, includes or references all the information necessary to describe and validate the signature details and artifacts, including:

- The X.509 certificate used
- A description of the signature algorithm and its details
- The signature value itself
- References to the message body elements that were signed
- Information about the key or keys used for signing

Later, as these JSRs and specifications finalize, the corresponding Java APIs will hide much of these details from the application developer.

7.4.4 Using Message-Level Security Mechanisms

How can you make use of these emerging technologies and Java API implementations as they become available? There are two ways to approach this problem:

1. You can make the security code and any supporting framework for message-level security part of your application, by placing it in the application's `.ear` file. This is the portable approach.
2. You can use application server-specific extensions that explicitly provide message-level security. Since they try to make new features available before standards are finalized, some application server vendors may offer non-standard extensions that integrate some message-level security capabilities. Eventually, these specifications become part of the standard J2EE platform, but they may differ from the implementations offered by these early adapters. For obvious reasons, this approach is not portable.

Keep in mind that some of these technologies have already matured more than others. For example, the Java Web Services Developer Pack toolkit has begun incorporating some of the digital signature standards. In addition, some Apache Foundation projects include some implementations of emerging message-level security capabilities.

Let's look at how you might implement a portable strategy to incorporate message-level security into your J2EE application. Note that while this is possible, it is not a task for every application developer. You should attempt this only if you feel comfortable handling security code, since it involves writing a framework for security. However, it may be a useful strategy if you need to use message-level security today and cannot wait for it to be incorporated into the J2EE platform.

Suppose you want to add a digital signature to a message involved in a single exchange between two participants. First, try to leverage existing J2EE technologies and mechanisms. For example, because JAX-RPC is the primary message exchange technology for Web service interactions, we extend it so that we can use it to exchange some XML messages with message-level security.

Recall that JAX-RPC has handlers that provide a mechanism to intercept a SOAP message at various points in the request and response stack. We can use the handlers' intercepting filter pattern to interpose on the message exchange at the points in the interaction where handlers occur. These points are:

- On the client side:
  - after parameters are marshalled into the request
  - before unmarshalling values returned in the response
- On the server side:
  - before unmarshalling parameters for dispatch
  - after marshalling return values into the response

Since handlers are configurable on both the client and the endpoint, you can customize them to apply security services at these interceptions points. To do so, you need to declaratively add the security code to the methods `handleRequest`, `handleResponse`, and `handleFault` on the client and server.

You can also use the SAAJ API to inspect and manipulate raw SOAP messages. SAAJ also gives you a compound message view capability that lets you examine MIME-based attachments. With SAAJ, you can also embed the digital signature information into the XML document and add the necessary security information to the header and message.

You can also make use of existing implementations of message-level security functionality, such as the digital signature capability. For portability, you need to include the message-level security implementations in the application's `.ear` file.

At this early stage, it is also recommended that you create a library of actions that wrap security tasks and functionality, such as the digital signature implementation. It is also a good idea to provide multiple defaults for common use cases, such as for obtaining X.509 certificates, handling verification faults, and so forth.

You may want to combine message-level security with other J2EE security mechanisms, both declarative and programmatic mechanisms. For example, you may want to use HTTPS as the transport protocol even though the document is signed by a message-level mechanism. If you choose to use any of the J2EE declarative or programmatic security mechanisms along with JAX-RPC handlers, keep in mind the order in which the security constraints are enforced:

1. The container applies the declarative security mechanisms first.
2. The handlers run and apply their checks.
3. J2EE programmatic security mechanisms run after the handler checks.

Another way to combine security mechanisms is to add some secure message-level functionality to an existing transport-level security solution. For example, if you have an existing Web service that uses SSL, you may want to add message-level integrity or persistent confidentiality functionality. Adding this security at the message level ensures that integrity or confidentiality persist beyond the transport layer.

7.5 Comparing Security Mechanisms

The JAX-RPC approach to security (discussed in “Security for JAX-RPC Interactions” on page 286) primarily concerns securing peer-to-peer communication. It relies on SSL and HTTPS to create a secure channel between two peers. Message-level security embeds the security information with each message. Let’s see how these two approaches compare.

7.5.1 Advantages and Limitations of SSL and HTTPS

SSL, which secures the communication channel between a client and a server, supports both client and server authentication. SSL provides authentication, data integrity, data confidentiality, and point-to-point secure sessions.

However, there are some limitations to SSL. Since it covers just a single connection between two peers (or one “hop” in a communication network), SSL does

not support end-to-end message exchange when intermediaries are involved. SSL is also bound to the transport protocol, which means that it might not work if a Web service uses a different transport protocol.

SSL has some limitations when a protocol translation is required between a sender and a recipient. In these cases, some security information—identity, authentication, and confidentiality, for example—stops at the HTTPS endpoint. This can be a particular problem for a Web service interaction that requires maintaining integrity throughout the entire system, even into the persistence of the message. Along with persisting a message, you may want to persist the security associated with the message to prove that the message had been sent. HTTPS does not provide an inherent signature that can be used for non-repudiation.

SSL and HTTPS may not provide sufficient granularity for applying security. You may want to encrypt only certain elements or fragments of an XML document, particularly when dealing with a large document, but HTTPS encrypts the entire document. Since encryption is resource intensive, this characteristic of HTTPS may have a significant performance impact.

In a multi-hop scenarios, especially one where your application uses an intermediate recipient just to route messages, SSL requires that the message be encrypted and decrypted for each hop. (This is because SSL does not truly support end-to-end messaging.) Even when the intermediary hop just routes a message, HTTPS and SSL requires that you first decrypt the message, then encrypt it, before forwarding it to the target service. Not only do these decryption and encryption actions impact performance, it may also compromise the security of the message.

7.5.2 Advantages and Limitations of Message-Level Security

Message-level security has different characteristics from SSL security. Message-level security is not tied to a particular session, plus the data is secured independently from the communication mechanism. Rather than using peer-based authentication as SSL does, message-level security uses data-origin authentication. With data-origin authentication, the security service verifies that the original source of a received message is as claimed. (With peer-based authentication, by contract, the security service verifies that the identity of a peer—in an association between a sender and receiver—is the identity claimed.)

Message-level security is most advantageous when used in situations involving multi-hop message exchange—that is, a message is sent sequentially to multiple recipients. It is also best used when sending different portions of a document

to different recipients, especially since you can limit encryption to just the fragment of the document for that recipient rather than having to encrypt and send the entire document to each recipient. Furthermore, each fragment of the document may be encrypted differently.

Message-level security is also helpful when you want to associate non-repudiation with a document, such as when you want to prove that a particular customer submitted a certain purchase order.

Using message-level security, you can also more easily persist security information, since the security is already part of the message. That is, if you want to persist the message, you can easily persist the security associated with the message. With SSL, however, the security is transient since it lasts for just the single session.

However, although it should make it easier to design and implement certain use cases, message-level security is still an emerging technology. SSL is a mature technology that is well understood and widely used today.

Keep in mind that message-level security mechanisms are designed to integrate in with existing security mechanisms, such as transport security, public key infrastructure (PKI), and X.509 certificates. You can also use both message-level security and transport layer security simultaneously.

7.6 Conclusion

This chapter explained the J2EE platform security model as it applies to Web service endpoints, and showed how to use the platform security model in different security scenarios. In particular, it described the declarative security approach and mechanisms used in the platform, and how the platform handles authentication and authorization.

The chapter described how to implement a secure environment using the JAX-RPC APIs and runtime. It discussed the JAX-RPC endpoint and client programming models, and how each model handles authentication and authorization and how to set up a secure environment. The chapter also introduced the message-level Web service security model, and provided guidelines for using this approach to security. The chapter concluded with a discussion of the advantages and limitations of the different security mechanisms available with the J2EE platform.

The next chapter, about the architecture of an actual Web service application, puts all the conceptual information covered so far into practice.