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, might only want to let a certain set of users access particular resources.

Setting up security also involves some usability 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 verify that a claimed identity is genuine
- Authorization, which lets you 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
- Nonrepudiation, 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 integrates existing security mechanisms, and makes it easier to design and implement secure applications.

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

7.1 Security Scenarios

Enterprise environments with Web site applications have a variety of security use case scenarios. Although the spread of Web services has given rise to additional security use cases, these Web services application use cases have similar security needs to those of Web-based enterprise applications (such as browser-based applications accessing Web sites). Typically, Web site and Web service application use cases involve access to services through the Internet or an intranet, allow users to access certain sets of resources but not others, and allow users to perform some set of actions. In addition, users might require access to other resources, such as a database, and they might need to interact with other applications.

Some of the security needs of Web site applications and Web services are very 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. While processing a client request, 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 a Web service interaction in which a client request to the service causes the service endpoint to interact with other components, resources, and systems. It illustrates that a Web service request can take many paths and result in interactions with different containers, components, and resources, including other Web services. Requests to a Web service start with a client sending a message to a Web service endpoint running in a Web or EJB container.

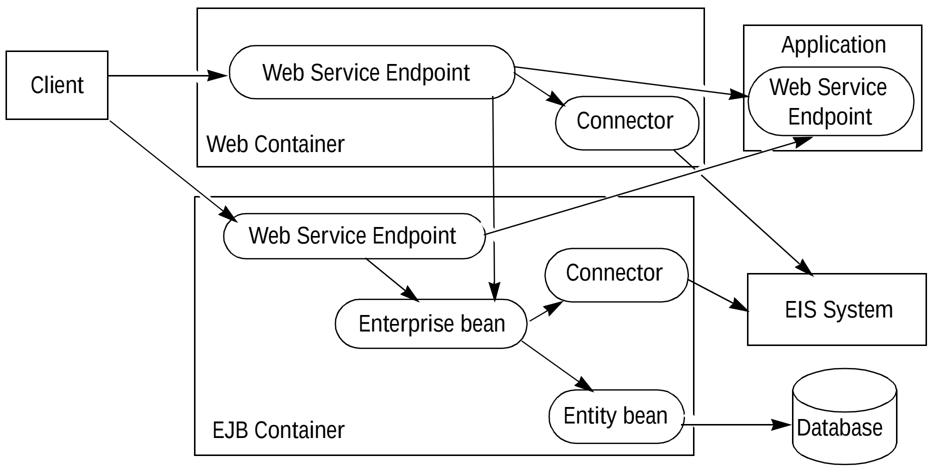

Figure 7.1 Anatomy of a Web Service Interaction

❐ However, designing a secure Web service involves more than just securing the 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 other J2EE components, resources, and so forth, that it undertakes to process the request.

Most client requests to a service require the service to access a series of components to fulfill the request and each call might have its own, unique security requirement. This results in a chain of calls to various components, some of which might be within the initiating component's security domain and others of which are outside that security domain. 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 might 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 might make a request to a Web service. The client call is to an endpoint, which in turn might 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.

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 might require that clients provide some credentials—such as a username and password, or a digital certificate such as an X.509 certificate—to help in proving their identity. The client of a Web service might require that a service provide it with some evidence to help establish its identity, which typically is done using a digital certificate.

Furthermore, since a Web service might 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 might need to provide identity information to authenticate itself to resources and components. The resources and components might 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 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 might want to let only certain authorized clients access its services. Or, an application might want to restrict different sets of its resources and functionality to different groups of clients. An endpoint might allow all clients to invoke its basic service, but it might grant some clients extra privileges and access to special functions. For example, you might want to limit access to only users who are classified as managers or to only users who work for a particular department. 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 and resources, the endpoint needs some way to control access to them. 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 whether 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 contain 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 might require integrity constraints. That is, message exchanges between a client and a service might require a digital signature to verify that the message was not altered in transit. The message recipient, by validating a signature bound 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 certificates to establish such secure channels. HTTPS provides a secure message exchange for one hop between two parties.

7.1.1.4 Message-Level Security

Besides creating a secure communication channel between a client and a Web service, some Web service message exchanges might require that security information be embedded within the SOAP 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 useful in XML document-centric applications, 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 might shield the Web service. The operational environment security might be sufficient to satisfy your security needs. Similarly, Web services in environments that require communication via a Virtual Private Network (VPN) might not need to worry about issues such as confidentiality, since the communication channel is already secure.

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

7.2 J2EE Platform Security Model

The J2EE platform container provides a set of security-related system services 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 is primarily declarative and is specified externally from the application code. *Declarative security* mechanisms used in an application are expressed via a declarative syntax in a configuration document called a deployment descriptor. The declarative security model has the advantage of enabling you to easily change these declarative settings to match security policy.

Declarative references in the deployment descriptor, rather than program code, define much of the security for a J2EE application. 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 application security policy declared 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 client calls and the application's components to perform security checks and otherwise manage the applications.

In addition to declarative security, the J2EE platform includes APIs to add security code into your components. *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.

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 and authentication mechanisms, handling existing user identity information, digital certificates, and so forth. The model provides a unifying layer above other security services, and its coherent programming model hides the security implementation details from application developers. For example, the J2EE security model provides mechanisms to leverage existing Internet security standards such as Secure Sockets Layer (SSL).

In addition, the J2EE platform security model gives you the ability to provide security boundaries. Once you have established these security boundaries, you can map users to their organizational roles and combine users into logical groups according to these roles.

Let's look in more detail at the J2EE platform security services and mechanisms. This security model applies to Web services as well as to the entire J2EE platform. "Security for Web Service Interactions" on page 308 describes how a Web service application can leverage these J2EE security mechanisms.

7.2.1 Authentication

Authentication is the mechanism by which a client presents an identifier and the service provider verifies the client's claimed identity. 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 resource, such as JSPs, enterprise beans, databases, and Web service endpoints. When these resources are protected, as is often the case, a client presents its identity and credentials, and the container determines whether the client meets the criteria for access specified by the authorization rules. 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.

Not only does the container enforce authentication and establish an identity when a client calls a component, but the container also handles authentication when the initially called component makes calls to other components and resources. Processing a client's request to a component might require the component to make a chain of calls to access other resources and components. Each subsequently called component might have its own authentication requirements, and these requirements might differ from those of the initially called component. The J2EE container handles this by establishing an identity with each call along the chain of calls. 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.

It is also possible to configure a component to establish a new identity when it acts as a client in a chain of calls. When so configured, a component can change the authenticated identity from the client's identity to its own identity. Regardless of how it is handled, the J2EE container establishes an identity for calls made by a component. Also, the J2EE container handles unauthenticated invocations that do not require a client to establish an identity. This mechanism can be useful for supporting use cases where a client does not have to authenticate.

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 logical boundary around 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.

The container ensures that the identity of a call is authenticated before it enters the protection domain; this is usually done with a credential, such as an X.509 certificate or a Kerberos service ticket. A credential 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.

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 whether authentication is required and to provide the mechanisms for handling identities and credentials.

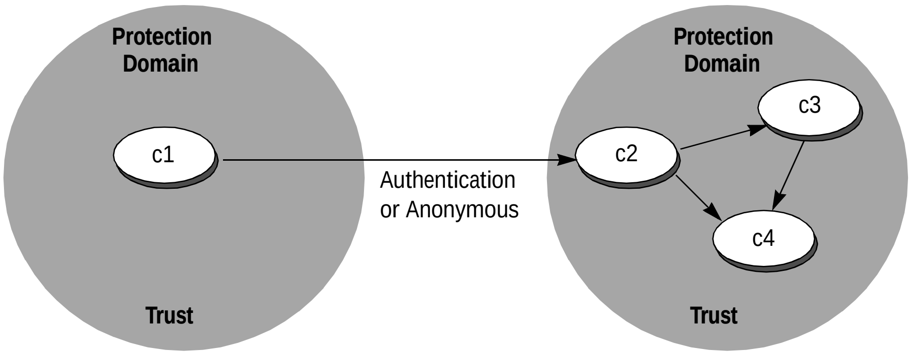

Figure 7.2 Protection Domain Established by Authentication Boundaries

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 the authentication 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 username and password obtained from the Web client. The username and password are included in the HTTP headers and are handled at the transport layer.
- 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. This type of authentication, geared toward Web page presentation applications, is not used for Web services.
- HTTPS mutual authentication—Both the client and the server use digital certificates to establish their identity, and authentication occurs over a channel protected by Secure Sockets Layer.

Generally, for Web tier authentication, the developer specifies an authorization constraint to designate those Web resources—such as Web service endpoints, 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 multiple 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 accessible to the user within a single application server without requiring the user to re-authenticate for each application. However, this mechanism is more appropriate for session-aware, browser-based Web applications; it is not as applicable to Web service interactions since Web services have no standard notion of session-oriented interactions. Other efforts provide similar security capabilities to Web services, such as the Liberty Alliance specifications (`http://www.projectliberty.org`).

7.2.1.3 EJB Tier Authentication

The EJB container has the ability to handle authentication. When a client directly interacts with a Web service endpoint implemented by an enterprise bean, the EJB container establishes the authentication with the client. Optionally, you can structure an application so that a Web container component may handle authentication for an EJB component. Several use case scenarios describe these situations.

One common 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. This is often done in browser-based Web applications and also with Web services applications that have a JAX-RPC Web endpoint. In these cases, the application developer places a Web component in front of the enterprise bean and lets the Web component handle the authentication. Thus, the Web container vouches for the identity of those clients who want to access enterprise beans, and these clients access the beans via protected Web components. Figure 7.3 illustrates how an application can be structured 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.

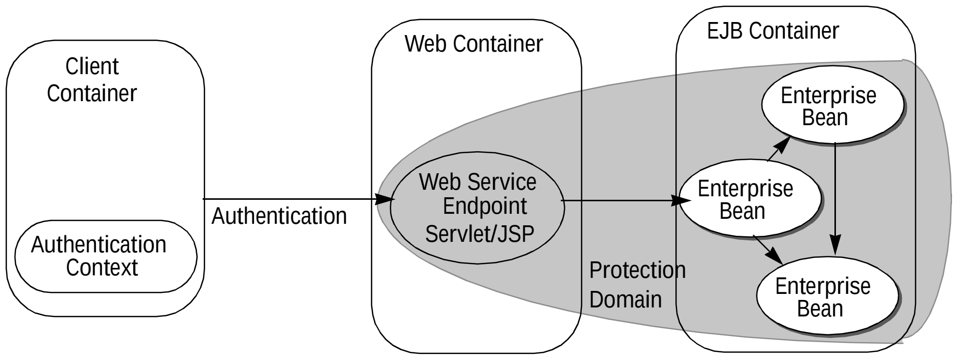

Figure 7.3 Using the Web Container to Establish an EJB Tier Protection Domain

Another use case scenario involves sending a SOAP request to an EJB service endpoint. Since the caller is making the SOAP request over HTTP, 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 container directly handles the authentication. Note that in the J2EE platform, both Web and EJB tier endpoints support the same mechanisms for Web service authenication.

A third use case entails calls made directly to an enterprise bean using RMI-IIOP. This scenario is not common for Web services since they are not accessed with RMI-IIOP. However, some Web service endpoints, while processing a request, may need to access a remote enterprise bean component 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, client authentication is enforced just above the transport layer. The Security Attribute Service also permits identity assertion, which is an impersonation mechanism, so that an intermediate component can use an identity other than its own.

7.2.1.4 Enterprise Information System Tier Authentication

Many application components and Web service endpoints need to access enterprise information systems, such as databases or JMS resources. J2EE components get access to the connections of these resources through a resource manager connection factory. For example, the `javax.sql.DataSource` interface provides a resource manager factory interface to obtain a `javax.sql.Connection` for a database. JMS, JavaMail, and URL connection factories are also available for these common types of resources.

When 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 the application. With application-managed resource manager sign-on, it is possible for components that programmatically manage resource sign-on to 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 enterprise 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 with a capability to sign-on to another resource is secured by appropriate authorization rules. Otherwise, that component can be misused to gain unauthorized access to the resource.

The J2EE Connector architecture offers a standard API for application-managed resource manager sign-on. This 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. Since the J2EE application programming model focuses on permissions, which indicate who can do what function, authentication and identity establishment occur before authorization decisions are enforced.

After successful authentication, a credential is made available to the called component. The credential contains information describing the caller through its identity attributes. Anonymous callers are represented by 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 attributes and shared authorization attributes in the credential are collectively represented as *security attributes.* Comparing the security attributes of the credential associated with a component invocation 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 credential associated with a component invocation to the access control rules for the target component. If the rules are satisfied, the container allows the call; otherwise, it rejects the call.

7.2.2.1 Declarative Authorization

Deployment establishes the container-enforced access control rules associated with a J2EE application. Generally, a deployment tool maps an application permission model, which is 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. At deployment, the security roles are mapped to identities in the operational environment to establish the capabilities of users in the runtime environment. Callers authenticated by the container as 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. The Web container enforces authorization requirements similar to those for an EJB container. Security constraints with associated roles also protect Web resource collections, that is, a URL pattern and an associated HTTP method, such as `GET or POST`.

Although deployment descriptors define authorization constraints for an application, security mechanisms often require more refinement plus careful mapping to mechanisms in the operational environment in which the application is ultimately deployed. Both the EJB and Web tiers define access control policy at deployment, rather than during application development. The access control policy can be stated in the deployment descriptors, and the policy is often adjusted at deployment to suit the operational environment.

It is also possible during deployment to refine the privileges required to access components. At the same time, you can define the correspondence between the security attributes presented by callers and the container privileges. The mapping from security attributes to container 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 interacts with resources hosted in a Web or EJB container. 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, clients 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 320.

7.2.2.2 Programmatic Authorization

A J2EE container decides access control before dispatching method calls to a component. In addition to these container pre-dispatch access control decisions, a developer might need to include some additional application logic for access control decisions. This logic may be based on the state of the component, the parameters of the invocation, or some other information. A component can use two methods, `EJBContext.isCallerInRole` (for use by enterprise bean code) and `HttpServletRequest.isUserInRole` (for use by Web components), to perform additional access control within the component code.

To use these functions, a component must specify in the deployment descriptor 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, deployment 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.

Additionally, a component might want to use the identity of the caller to make decisions about access control. As noted, a component can use the methods `EJBContext.getCallerPrincipal` and `HttpServletRequest.getUserPrincipal` to obtain the calling principle. Note that containers from different vendors may represent the returned principal differently. If portability is a priority, then care should be taken when code is embedded with a dependence on a principle.

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 that a particular person is responsible for the content: In addition, the modification of the message by anyone other than the creator of the content is detectable by the receiver.

Configuring the containers to apply confidentiality and integrity mechanisms is done when an application is deployed into its operational environment. Components that need to be protected are noted as such. The corresponding containers can be configured to employ the required confidentiality and integrity mechanisms when interactions with these components occur over open or unprotected networks. Containers can also be configured to reject call requests or responses with message content that should be protected but is not.

The J2EE platform requires that containers support transport layer integrity and confidentiality mechanisms based on SSL so that security properties applied to communications are established as a side effect of creating a connection. SSL 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. Details about interacting with a J2EE component using SSL are discussed in the next section. When a component's interactions with an external resource include sensitive information, these sensitivities should be described in the `description` subelement of the corresponding `resource-ref`. These elements make sensitive information available when security requirements are set at deployment.

7.3 Security for Web Service 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. The J2EE platform supports the WS-I Basic Profile 1.0 specifications for secure interoperable Web service interactions. WS-I security compliance requires HTTPS and single hop security for a request and reply between a client and service. The Basic Profile requires that the transport layer of HTTPS be combined with additional mechanisms for basic and mutual authentication.

The J2EE platform provides Web tier and EJB tier endpoints with similar security mechanisms for Web services. Most J2EE developers should already be familiar with its security mechanisms, since the platform already provides transport layer security and authentication support for non-Web service interactions involving browsers and Web pages.

With Web service interactions, both the request and the reply may have security requirements. In addition, Web service endpoints must interact securely with other components and resources when processing requests. Developers may also leverage other J2EE platform security mechanisms, such as authorization, to design and build secure Web services.

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 interaction on the J2EE platform, that is, how to authenticate and establish a secure HTTPS channel. As with any J2EE component, you can use declarative mechanisms to define the security for a Web service endpoint. Similarly, you may include programmatic security mechanisms in your Web service endpoints, and your service endpoint can leverage the platform's declarative mechanisms.

The key requirements for a secure Web service interaction are authentication and establishing a secure SSL channel for the interaction. Let's first examine how to secure the transport layer, and then we'll look at the available authentication mechanisms.

7.3.1.1 Securing the Transport Layer

SSL and Transport Layer Security (TLS) are key technologies in Web service interactions, and it is important to understand how to establish an SSL/TLS-protected interaction and authenticate clients. Note that TLS is an enhanced specification based on SSL. References to SSL refer to both SSL and TLS.

SSL is a standard mechanism for Web services that is available on virtually all application servers. This widely used, mature technology, which secures the communication channel between client and server, can satisfy many 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.

Authentication is an important aspect of establishing an HTTPS connection. The J2EE platform supports the following authentication mechanisms for Web services using HTTPS:

- The server authenticates itself to clients with SSL and makes its certificate 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.

While browser-based Web applications rely on these same authentication mechanisms when accessing a Web site, Web services scenarios have some additional considerations. With Web services, the interaction use case is usually 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 individuals involved in the deployment of the Web service interaction must distribute and exchange the server certificate, and possibly the client certificate if mutual authentication is required, prior to the interaction occurrence. Since an interoperable standard for Web service certificate distribution and exchange does not exist, the J2EE platform does not require one. Certificates must be handled in a manner appropriate to the specific operational environment of the application.

A Web service can be implemented and deployed in either the Web tier or EJB tier. The security mechanisms are the same at the conceptual level but differ in the details. The endpoint type determines the mechanism for declaring that a Web service endpoint requires SSL. For a Web tier endpoint (a JAX-RPC service endpoint), you indicate you are using SSL by setting to `CONFIDENTIAL` the `transport-guarantee` subelement of a `security-constraint` element in the `web.xml` deployment descriptor. This setting enforces an SSL interaction with a Web service endpoint. (See Code Example 7.1.)

```
<web-app>
    <security-constraint>
        ...
        <web-resource-collection>
            <web-resource-name>orderService</web-resource-name>
            <url-pattern>/mywebservice</url-pattern>
            <http-method>POST</http-method>
            <http-method>GET</http-method>
        </web-resource-collection>
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
</web-app>
```

Code Example 7.1 Requiring SSL for Web Tier Endpoints

Setting up SSL for EJB tier endpoints varies according to the particular application server. Generally, for EJB endpoints a developer uses a `description` subelement of the target EJB component to indicate that the component requires SSL when deployed. Although EJB endpoints are required to support SSL and mutual authentication, the specifications have not defined a standard, portable mechanism for enabling this. As a result, you must follow application server-specific mechanisms to indicate that an EJB endpoint requires SSL. Often, these are application server-specific deployment descriptor elements for EJB endpoints that are similar to the `web.xml` elements for Web tier endpoints.

7.3.1.2 Specifying Mutual Authentication

You can also specify HTTPS with mutual authentication for a Web service endpoint. For Web tier endpoints, you first specify a secure transport (see the previous section) and then, in the same deployment descriptor, set the `auth-method` element to `CLIENT-CERT`. (See Code Example 7.2.)

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

Code Example 7.2 Requiring Mutual Authentication for Web Tier Endpoints

The combination of the two settings—`CONFIDENTIAL` for `transport-guarantee` (see Code Example 7.1) 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 client-specific identifying information.)

Specifying mutual authentication for EJB service endpoints is specific to each application server. Usually it is done in a similar manner to specifying mutual authentication for Web tier endpoints.

7.3.1.3 Specifying Basic and Hybrid Authentication

With basic authentication, a Web service endpoint requires a client to authenticate itself with a username and password. The type of the Web service endpoint determines how to specify requiring basic authentication for the service. For a Web tier (JAX-RPC) service endpoint, set the `auth-method` element to `BASIC` for the login configuration (`login-config`) element in the `web.xml` deployment descriptor, as shown in Code Example 7.3:

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

Code Example 7.3 Requiring Basic Authentication for Web Tier Endpoints

For a Web service with an EJB endpoint, you use 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.

A Web service may also require hybrid authentication, which is when a client authenticates with basic authentication and SSL is the transport. The client authenticates with a username and password, the server authenticates with its digital certificate, and all of this occurs over a HTTPS connection. 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 for all message content, including the client authenticators, such as username and password.

Enabling hybrid authentication for a Web service endpoint generally requires two operations (both previously discussed): setting the transport to use the confidentiality mechanism of HTTPS and setting the authentication of the client to use basic authentication. For EJB endpoints, you use application server-specific mechanisms. For Web endpoints, you set deployment descriptor elements. Code Example 7.4 demonstrates how to configure hybrid authentication by combining the deployment descriptor choices for basic authentication and confidential transport.

```
<web-app>
    <security-constraint>
        ...
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
...
<login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>some_realm_name</realm-name>
</login-config>
...
</web-app>
```

Code Example 7.4 Requiring SSL Hybrid Authentication for Web Tier Endpoints

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

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.

❒ Ensure that you set up an SSL transport for each endpoint that requires basic authentication. Otherwise, the client authenticator is not fully protected. For example, for Web endpoints, ensure that the `transport-guarantee` element of each protected Web endpoint is set to `CONFIDENTIAL` for an application using a hybrid authentication mechanism.

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 endpoint also needs to describe its security policy and make that information available to clients. If the WSDL document does not express the policy information, then the service must use other means to make its requirements known so that clients can be designed and implemented with those requirements in mind and be able to interact with the service.

At the present time, a WSDL description contains minimal information about the security characteristics of an endpoint—just the HTTPS location specified in the endpoint URL. The security functionality specified by the WS-I Basic Profile 1.0 only requires that Web services using HTTPS have `https` in the URI of the location attribute of the `address` element in its `wsdl:port` description. See Code Example 7.5.

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

Code Example 7.5 WSDL Security Description

Since current WSDL documents have no standard mechanism to indicate whether an endpoint requires basic or mutual authentication, such information needs to be made available through service-level agreements between the client and endpoint. Future versions of the WSDL description may be extended to include descriptions of endpoint security requirements, perhaps by using metadata or annotations similar to CSIv2.

Since the present WSDL description for security is limited, you need to consider what other mechanisms you can use today to define security policies for endpoints. Generally, you should try to use the security mechanisms included with a particular vendor's application server. You have available options such as providing some metadata in another location, making some security assumptions among your partners, including security descriptions as a nonstandard part of JAXR entries, or even extending the WSDL description yourself. Not only that, your application and its endpoints may have built-in implicit assumptions, and you may need to provide a description of these unique security requirements. Clients need to be aware of all the requirements of a service so that they can be designed and implemented to interact properly with the service.

❒ 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 available for endpoint developers a separate document that describes the security policy for an endpoint. In this document, clearly describe the information needed by a client.

7.3.2 Client Programming Model

Client developers must handle some security requirements for their applications. The mechanisms for handling security vary according to the type of client. We focus on J2EE components, including enterprise bean and servlet components, acting as clients of Web services. 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 in 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 stand-alone J2SE clients, since they are not run within a J2EE container generally cannot use the services of the J2EE platform. Stand-alone J2SE clients can use the JAX-RPC technology outside of the J2EE platform if they include the JAX-RPC runtime in their stand-alone environment. If you develop a stand-alone 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 just noted, you can also include the JAX-RPC runtime, then use its mechanisms to set up username and password properties in the appropriate stubs and 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 that also satisfies the WS-I interoperability requirements.

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 accessed 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 308, a target endpoint defines some security constraints or requirements that a client must meet. For example, the client’s interaction with the service endpoint might require basic authentication and 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 WSDL document may not describe all security requirements (see “Publicizing Security Policy” on page 313), discovering the target endpoint’s security policy is specific to each situation. Once you know the client’s security requirements for interacting with the service, you can set up the client component environment to make available the appropriate artifacts. For example, if the Web service endpoint requires basic authentication, the calling client container places the username and password identifying information in the HTTP headers. Let’s take a closer look at what happens with both basic and mutual authentication.

For HTTP basic authentication, application server-specific mechanisms, such as additional deployment descriptor elements, are used to set the client username and password. These vendor-specific deployment descriptors may statically define at deployment the username and password needed for basic authentication. However, at runtime this username and identifier combination may have no relation to the principal associated with the calling component. When the JAX-RPC call is made, the container puts the username and password values into the HTTP header. Keep in mind that the J2EE 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 components programmatically setting these identifier values.

If the endpoint requires mutual authentication, the application server instance environment is set at deployment with the proper certificates such that they are available to the J2EE container. Since a client component's deployment descriptors have no portable, cross-platform mechanism for setting these security artifacts, they must be set using the particular application server's own mechanisms. In other words, an enterprise bean or servlet component that interacts with a Web service requiring mutual authentication must, at deployment, make the appropriate digital certificates available to the component's host container. The client's container can then use these certificates when the component actually places the call to the service.

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 5 for details.) The J2EE container not only manages the HTTPS transport, it handles the authentication for the call using the digital certificate or the values specified in the deployment descriptor.

7.3.3 Propagating Component Identity

Web service endpoints and other components 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, non-Web service J2EE components can make calls to Web services. Each call between components and endpoints may have an identity associated with it, and this identity may need to be propagated.

There are two cases of identity propagation, differentiated by the target of the call. Both cases start with a caller that is a J2EE component—including a component that is a Web service endpoint. In the first case, the J2EE component or endpoint calls a J2EE component that is *not* a Web service. In the second case, the J2EE component or Web service makes JAX-RPC calls to a Web service.

7.3.3.1 Propagating Identity to Non-Web Service Components

All J2EE components have an invocation identity, established by the container, that identifies them when they call other J2EE components. The container establishes this invocation identity using either the `run-as(role-name)` or `use-caller-`

identity identity selection policy, both defined in the deployment descriptor. The container then uses either the calling component's identity (if the policy is to use the use-caller-identity) or, for run-as(role-name), a static identity previously designated at deployment from the principal identities mapped to the named security role.

Developers can define component identity selection policies for J2EE Web and EJB resources, including Web service endpoints. If you want to hold callers accountable for their actions, you should associate a use-caller-identity policy with component callers. 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 7.6 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 7.6 Configuring Identity Selection Policies for Enterprise Beans

Code Example 7.7 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 7.7 Configuring Identity Selection Policies for Web Components

7.3.3.2 Propagating Identity to a Web Service

Protection domains help to understand how clients set identity for Web service calls. (See “Protection Domains” on page 300.) Recall that 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 considered outside of any protection domain.

❐ When calling a Web service, be prepared to satisfy its security requirements. Web services are loosely coupled and it is more likely that a call to a service will cross protection domains.

Since Web service calls are likely to cross protection domains, identity propagation mechanisms (such as `run-as` and `use_caller_identity`) and security context are not useful and are not propagated to service endpoints. When a J2EE component acting as a Web service client specifies the `run-as` identity or the `use-caller-identity`, the container applies that identity only to the component’s interactions with non-Web service components, such as enterprise beans. Some vendors may provide mechanisms to propagate identity across protection domains, but these mechanisms may not be portable.

This brings us to the question of how to establish identity for Web services. For the client making calls to a service that requires authentication, the client container provides the necessary artifacts, whether username and password for basic authentication or a digital certificate for mutual authentication. The container of the target Web service establishes the identity of calls to its service endpoint. The Web service bases this identity on the mapping principals designated by when the service was deployed, which may be based on either the client's username and password identity or the digital certificate attributes supplied by the client's container. However, since no standard mechanism exists for a target Web service to map an authenticated client to the identity of a component, each application server handles this mapping differently.

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 entity bean `K` calls a method on service endpoint `Y`, since the Web service is not in the same protection domain, reauthentication must occur. Similarly, when endpoint `X` calls endpoint `Z`, the caller identifier cannot be propagated.

Applications can also use programmatic APIs to check client identity, and use that client identity to make identity decisions. For example, a Web tier endpoint, as well as other Web components, can use the `getUserPrincipal` method on the `HttpServletRequest` interface. An EJB endpoint, just like other enterprise bean components, 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.

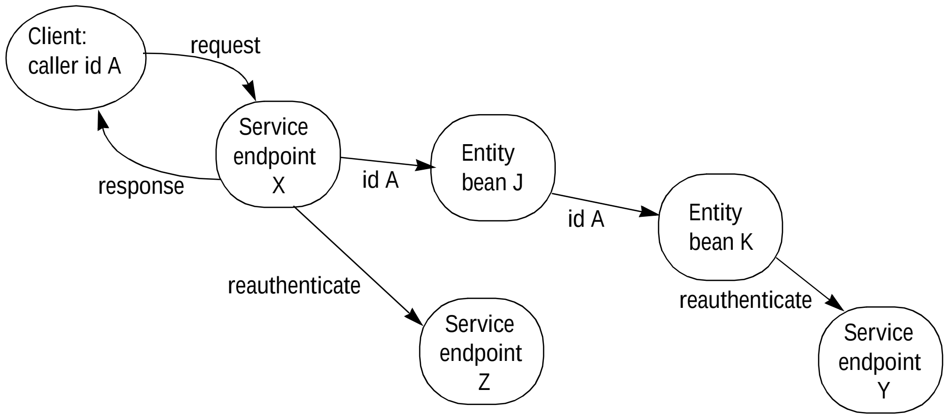

Figure 7.4 Security Propagation

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 since the standard J2EE authorization mechanisms can be leveraged. 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. A Web service is also free to leave its resources unprotected so that anyone can access its service.

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's interaction with other components and resources is handled by the same mechanisms used by any J2EE component. That is, the authorization mechanisms for Web service endpoints are the same as for other components in the J2EE platform.

The tier on which your endpoint resides determines how you specify and configure access control. In general, to enable access control you specify a role and the resource you want protected. Components in both tiers specify a role in the same manner, using the `security-role` element as shown in Code Example 7.8. With Web tier endpoint components, access control entails specifying a 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.

```
<security-role>
    <role-name>customer</role-name>
</security-role>
```

Code Example 7.8 Configuring a Role for an Authorization Constraint

What does this mean in terms of a Web service's access control considerations? Your Web service access control policy may influence whether you implement the service as a Web tier or an EJB tier endpoint. For Web tier components, the granularity of security is specific to the resource and based on the URL for the Web resource. For EJB tier components, security granularity is at the method level, which is typically a finer-grained level of control.

Let's consider a Web service with an interface containing multiple methods, such as the one shown in Code Example 7.9, where you want different access policies for each method. For a service endpoint interface such as this you might want to permit the following: Any client can browse the catalog of items available for sale, only authorized customers—for example, those clients who have set up accounts—can place orders, and only administrators can alter the catalog data. If you implement the service with a Web tier endpoint, then each method has the same protection because access control is the same for all methods that are bound to the port at the endpoint's URL. To handle a service with an interface containing multiple methods and different access policies, consider creating separate Web services where each service handles a different set of authorization requirements.

You have more flexibility if you implement the same Web service that has an interface containing multiple methods with an EJB endpoint. By using an EJB endpoint, you can set different authorization requirements for each method. See the next section, "Controlling Access to Web Tier Endpoints," and "Controlling Access to EJB Tier Endpoints" on page 323.

```
public interface OrderingService extends java.rmi.Remote {
    public Details getCatalogInfo(ItemType someItem)
            throws java.rmi.RemoteException;
    public Details submitOrder(purchaseOrder po)
            throws java.rmi.RemoteException;
    public void updateCatalog(ItemType someItem)
            throws java.rmi.RemoteException;
}
```

Code Example 7.9 Interface Methods Requiring Different Access 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 can be designed to handle the same security capabilities. However, it is generally discouraged to embed security code and use the programmatic security APIs in a component. A better approach keeps the security policy externalized form the application code and uses the declarative services with deployment descriptors.

❐ 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 Tier Endpoints

To control access to a Web component such as a Web service endpoint, the Web deployment descriptor specifies a `security-constraint` element with an `auth-constraint` subelement. Code Example 7.10 illustrates the definition of a protected resource in a Web component deployment descriptor. The descriptor specifies that only clients acting in the role of `customer` can access the URL `/mywebservice`. Note that this URL maps to all the methods in the service endpoint interface. Hence, all methods have the same access control.

```
<web-app>
....
<security-constraint>
    <web-resource-collection>
        <web-resource-name>orderService</web-resource-name>
        <url-pattern>/mywebservice</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>
...
<login-config>
    ...choose either basic or client(for mutual authentication)
</login-config>
<security-role>
    <role-name>customer</role-name>
</security-role>

</web-app>
```

Code Example 7.10 Web Resource Authorization Configuration

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

7.3.4.2 Controlling Access to EJB Tier Endpoints

The EJB deployment descriptors define security roles for an enterprise bean. These descriptors also specify, via the `method-permission` elements, the methods of a bean's home, component, and Web service endpoint interfaces that each security role is allowed to invoke.

Code Example 7.11 shows how to configure method-level access. The example specifies that the method `submitOrder`, which occurs on an interface of an enterprise bean Web service endpoint, requires that a caller belonging to the `customer` role must have authenticated to be granted access to the method. 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-intf>ServiceEndpoint</method-intf>
        <method-name>submitOrder</method-name>
    </method>
</method-permission>
```

Code Example 7.11 Enterprise Bean Authorization Configuration

Some applications also feature unprotected EJB endpoints 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 7.12 demonstrates the use of the `unchecked` element.

```
<method-permission>
    <unchecked/>
    <method>
        <ejb-name>PurchaseOrder</ejb-name>
        <method-name>getCatalogInfo</method-name>
    </method>
    <method>
    ...
</method-permission>
```

Code Example 7.12 Enterprise Bean Unchecked `method-permission`

In addition to defining authorization policy in the `method-permission` elements, you may also add method specifications to the `exclude-list`. Doing so denies access to these methods independent of caller identity and whether the methods are the subject of a `method-permission` element. Code Example 7.13 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 7.13 Enterprise Bean Excluded `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. Security programming is complex and difficult to write correctly, leading to a false sense of security. Use programmatic mechanisms for access control only when extra flexibility is required.

❐ If you have multiple Web tier endpoints with varying authentication requirements, consider bundling them in different `.war` files. 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.

❐ Provide security policy descriptions in addition 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 when configuring a client component to use HTTP basic authentication. 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 if deployment descriptors do include a username and password, be sure to store the deployment descriptors 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

Message-level security, or securing Web services at the message level, addresses the same security requirements—identity, authentication, authorization, integrity, confidentiality, non-repudiation, and basic message exchange—as traditional Web security. Both traditional Web and message-level security share many of the same mechanisms for handling security, including digital certificates, encryption, and digital signatures. Today, new mechanisms and standards are emerging that make it not only possible but easier to implement message-level security.

Traditional Web security mechanisms, such as HTTPS, may be insufficient to manage the security requirements of all Web service scenarios. For example, when an application sends a document with JAX-RPC using HTTPS, the message is secured only for the HTTPS connection, that is, during the transport of the document between the service requester (the client) and the service. However, the application may require that the document data be secured beyond the HTTPS connection, or even beyond the transport layer. By securing Web services at the message level, message-level security is capable of meeting these expanded requirements.

7.4.1 Understanding Message-Level Security

Message-level security, which applies to XML documents sent as SOAP messages, makes security part of the message itself by embedding all required security information in a message's SOAP header. In addition, message-level security can apply security mechanisms, such as encryption and digital signature, to the data in 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 that information to handle security needs. The SOAP message also contains information relevant to 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, the security mechanisms are independent of a transport protocol such as HTTPS.

JAX-RPC hides the details of a SOAP message exchange, but, to understand message-level security, it's helpful to examine a SOAP message in more detail.

(See “Simple Object Access Protocol” on page 33 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 can be embedded at the message level. The diagram expands a SOAP header to show the header’s security information contents and artifacts related to the message. It also expands the body entry to show the particular set of elements being secured.

The client adds to the SOAP message header security information that applies to that particular message. When the message is received, the Web service endpoint, using the security information in the header, applies the appropriate security mechanisms to the message. For example, the service endpoint might verify the message signature and check that the message has not been tampered with. It is possible to add signature and encryption information to the SOAP message headers, as well as other information such as security tokens for identity—for example, an X.509 certificate—that are bound to the SOAP message content.

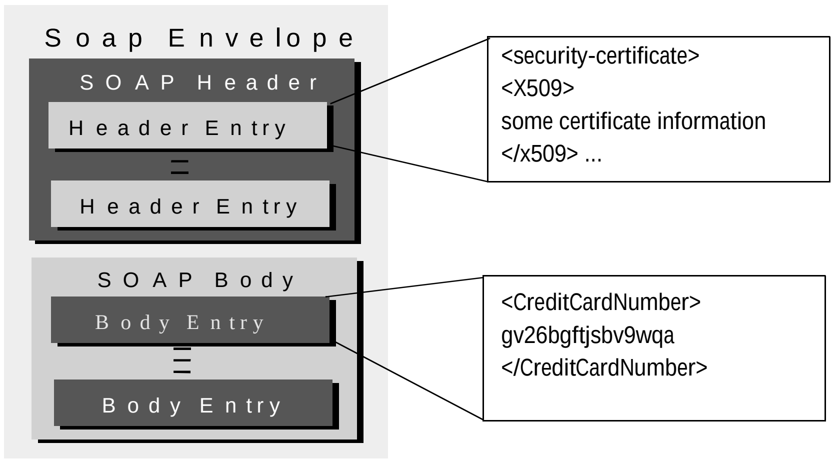

Figure 7.5 Embedding Security at the Message Level

In summary, message-level security technology lets you embed into the message itself a range of security mechanisms, such as identity and security tokens and certificates, and message encryption and signature mechanisms. The technology associates this security information with the message and can process and apply the specified security mechanisms. Message-level security uses encryption and it uses a digital signature to bind the claims—the identity attributes—from a security token to message content. It is possible to layer additional functionality on top of these basic mechanisms.

7.4.2 Comparing Security Mechanisms

The JAX-RPC over SSL (discussed in “Security for Web Service Interactions” on page 308) primarily concerns securing peer-to-peer communication. It relies on HTTP over SSL to create a secure channel between two peers.

Message-level security takes a different approach, since it embeds the security information within each message. Message-level security has different characteristics from SSL security. Let’s compare these two approaches.

7.4.2.1 Transport Layer Security and SOAP Messages

HTTP over SSL protocol is a transport layer security mechanism that applies security protection to messages only when they are “on the wire,” that is, during transport. A message is encrypted—and thus protected—while it is on the wire. However, the message data is decrypted at the transport layer boundary. At that point, the message is unprotected and vulnerable while it is passed to other system layers, whether operating system, application server, or J2EE application layers. Thus, the duration of protection using HTTP is the lifetime of the message on the wire at the transport layer.

Message-level security not only persists beyond the transport layer, it lasts for as long as the XML content is perceived as a SOAP message. Since the security is applied to the SOAP message, the protection remains and the security information is available to the application server container and to applications that have access to SOAP messages through mechanisms and APIs such as JAX-RPC handlers and SAAJ. The duration of protection for message-level security is the lifetime of the SOAP message, and this can span the transport boundary.

Message-level security has other advantages in addition to providing a longer duration of protection. Because security is part of the SOAP message, applications can support Web service interactions that require maintaining protection throughout the entire system or into the application layer. Having security as part of the message also makes it possible to persist both the message data and its security information. For example, perhaps to prove that a message was sent, an application may need to persist the message data and the digital signature bound to the message. Or, to protect against internal threats, an application may need to keep data in a SOAP message confidential, even to the application layer. HTTPS, since it only protects a message during transport, cannot give the application layer this encryption protection for the data.

7.4.2.2 Peer Entity and Data Origin Authentication

Two kinds of authentication in a network are: peer entity authentication and data origin authentication. With peer entity authentication, the security service verifies that the identity of a peer—in an association such as a session between a sender and receiver—is the identity claimed. Note that there must be an association between the two parties.

Data origin authentication verifies that the original source of a received message is as claimed, but, unlike peer entity authentication, no association between the sender and receiver is required. With data origin authentication, a target receiver can verify the identity of a message as belonging to the original message creator even if the message passes from its initial source through multiple participants before arriving at the target receiver.

A Web service interaction that uses HTTPS supports peer entity authentication, because the interaction covers just the connection between two peers. Message-level security supports data origin authentication, since its security is tied to the SOAP message itself rather than the transport mechanism.

Using HTTPS is disadvantageous in multi-hop scenarios where a message passes through numerous intermediate participants between the initial sender and target receiver, because each message exchange requires establishing a new association between the communicating participants. Furthermore, SSL requires that each participant decrypt each received message, then encrypt the same message before transmitting it to the next participant in the workflow. SSL, relying on peer entity authentication, does not support end-to-end multi-hop message exchange. (See Figure 7.6.)

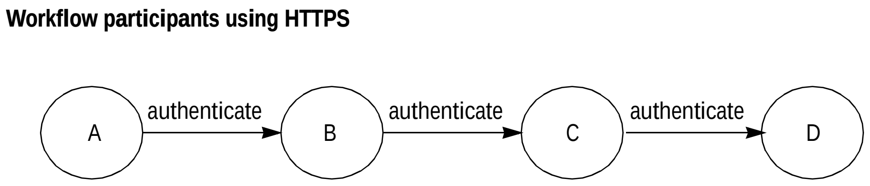

Figure 7.6 Authentication Between Point-to-Point Participants

Web service scenarios that pass messages to multiple participants lend themselves to using message-level security. Since message-level security is based on data origin authentication, an application that passes messages to numerous intermediary participants on the way to the target recipient can verify, at each intermediary point and at the final target recipient, that the initial message creator identity associated with the message is as claimed. In other words, the initial message content creator's identity moves with the message through the chain of recipients.

7.4.2.3 Levels of Granularity

In addition to handling end-to-end use cases, message-level security can provide security at a more granular level than HTTPS. The ability to apply security at different levels of granularity is important. Consider scenarios that handle XML documents, which are composed of a nested hierarchy of elements and subelements making them inherently granular. Message-level security, which is XML aware, can be more flexible in applying security mechanisms to a message than HTTPS.

For example, suppose you need to encrypt only certain elements or fragments of an XML document to be sent as a SOAP message. If you use HTTPS, you essentially encrypt the entire SOAP message since HTTPS encrypts everything passed on the wire. If you use message-level security, you can encrypt just a portion of the XML document, then send a SOAP message that is partially encrypted. Since encryption is computationally intensive, encrypting an entire document (particularly a large one) can impact performance.

Message-level security's finer-grained control in applying security protections is useful in common Web service interactions, such as end-to-end scenarios. You can apply different security to portions of a document so that participants in a workflow may only access those fragments applicable to their separate functions. For example, an application handling purchase orders may encrypt just credit card information. As the order passes among numerous workflow participants—customer relations, message brokers, supplier—only the appropriate participants, such as a financial department, can read the encrypted information. You could also apply different security mechanisms, such as different encryption algorithms, to various parts of a message, ensuring that only intended recipients can decrypt those parts of the message. Finer-grained control also supports intermediaries whose processing requires access to a small part of the message data, such as intermediaries that route messages to appropriate recipients.

7.4.2.4 Maturity of the Security Technologies

Message-level security is still an emerging technology, with relatively new specifications, some of which are not yet standardized. Moreover, these new specifications may not completely cover all security considerations.

HTTP over SSL is a mature, widely used and well understood standard technology. It is a technology that has been analyzed extensively and has held up against varied security threats. This technology supports both client and server authentication, data integrity, data confidentiality, and point-to-point secure sessions. The J2EE 1.4 platform relies on this technology to provide Web service interactions with standard portable and interoperable support.

Keep in mind that message-level security mechanisms are designed to integrate 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 together to satisfy your security requirements. For example, you might use a message-level digital signature while at the same time exchanging the message using HTTP over SSL.

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 the Organization for the Advancement of Structured Information Standards (OASIS), the World Wide Web Consortium (W3C), the Internet Engineering Task Force (IETF), and other standards bodies, concentrate on message-level security for XML documents. New Java APIs are also emerging to support these industry Web service security standards. These APIS are developed as Java Specification Requests (JSRs) through the Java Community Process, and future versions of the Java platform may include them.

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—Web Services Message Security Java APIs, which enable applications to construct secure SOAP message exchanges.

Let's look at how to apply message-level security mechanisms to build and send a SOAP message with some message-level security. For example, let's examine how message-level security APIs 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 7.14 shows the message header for this purchase order XML document with the embedded digital signature, per the XML Digital Signature specification. This code example is derived from the OASIS *Web Service Security: SOAP Message Security, Working Draft 17* document. See `http:www.oasis-open.org/committees/documents.php`.

```
<?xml version="1.0" encoding="utf-8"?>
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
    xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/06/secext"
xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">

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

      <S:Body wsu:Id="myBody">
         <myPO:PurchaseDetails xmlns:myPO=
            "http://www.someURL.com/purchaseOrder">
            some message content here ...
         </myPO:PurchaseDetails>
```

```
        </S:Body>
    </S:Envelope>
```

Code Example 7.14 Embedding a Digital Signature in an SOAP Message

This XML document shows the SOAP envelope containing the message body for the purchase order details and the message header, with the digital signature for the message. The security portion of the header, which is part of the SOAP message itself, includes or references all the information necessary to describe and validate the signature details and artifacts, including:

- Information specifying the security token, which is an X.509 certificate associated with the message. This information is enclosed within the `wsse:BinarySecurityToken` element.
- A description of the signature algorithm and its details, enclosed within the `ds:Signature` element.
- References to the signed message body elements. This is shown in the `<ds:Reference URI="#myBody>` element and associated attribute, which references the body of this message.
- The signature value itself, which is inside the `ds:SignatureValue` element.
- Information about the key or keys used for signing, enclosed with the `ds:KeyInfo` element. In this case the keys are from the associated X.509 certificate.

Since the message-level security specifications are still evolving, the details in this example may change. Regardless, the example does highlight how to associate security information with a particular SOAP message and include the security information as part of the message itself. As message-level security JSRs and specifications finalize and are incorporated in the J2EE platform, the corresponding Java APIs and the J2EE containers will hide many of these details from the application developer.

7.4.3.1 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. Although this is the portable approach, it may require more work. You should consider this approach if your situation necessitates it.
2. You can use application server-specific extensions that explicitly provide message-level security. This is the preferred approach. Since vendors try to make new features available before standards are finalized, some application servers may offer nonstandard extensions that integrate some message-level security capabilities. Eventually these specifications may become part of the standard J2EE platform, but they may differ from the implementations offered by these early adopters. Although it may not be portable, it is the easier approach and more likely to provide the intended security.

Some of these technologies are more mature than others. For example, the Java Web Services Developer Pack (Java WSDP) toolkit has already incorporated some of the digital signature standards. Java WSDP is an integrated toolkit from Sun Microsystems that allows Java developers to build and test XML applications, Web services, and Web applications using the latest Web service technologies and standards implementations. The Java WSDP toolkit is available at `http://java.sun.com/webservices/`. In addition, some Apache Foundation projects include 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 since it is usually quite difficult to write truly secure code. 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, try to plug in your security code to the SOAP messages that JAX-RPC exchanges. This may enable your Web services with message-level security. You can then leverage the JAX-RPC built-in mechanisms to manipulate the XML messages being exchanged.

Recall from Chapter 2 that JAX-RPC has handlers that provide a mechanism to intercept a SOAP message at various points during processing of request and response messages. You can use the JAX-RPC handler to interpose on the message exchange at the points in the interaction where handlers are invoked. 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

Handlers intercept all requests and responses that pass through a Web service endpoint, providing access to the actual SOAP message exchanged as part of the Web service request and response. Handlers let you apply different logic for service requests, responses, and faults. To do so, you add the appropriate code to the handler methods `handleRequest`, `handleResponse`, and `handleFault`. You can use handlers to apply message-level security to messages exchanged as part of your service. Since they are configurable on both the client and the endpoint, you can customize handlers to apply security services at both the client and service sides.

You 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. Also consider using existing implementations of message-level security functionality, such as the digital signature capability.

For portability, you must 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 the functionality of existing implementations of message-level security. This library of actions should provide a higher level interface to these security functions. When providing a security library around existing message-level security implementations, 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. Once the library is in place, you can use the SAAJ API from within the handler logic to access the SOAP message. Then, apply the message-level security with your security library. Figure 7.7 shows the main participants in this process.

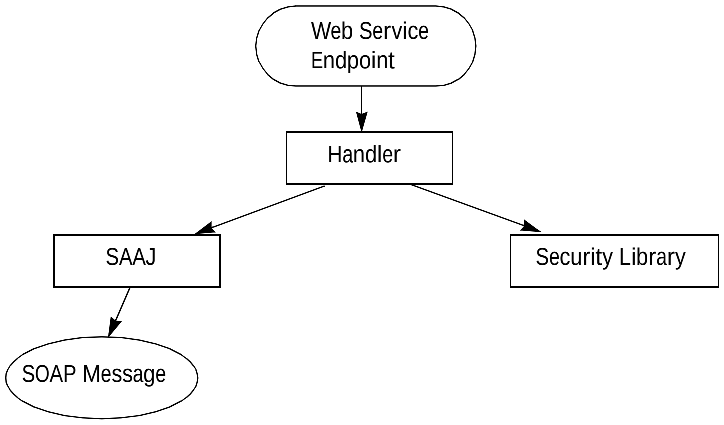

Figure 7.7 Implementing Message-Level Security

You may want to combine message-level security with other J2EE declarative and programmatic security 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.

You can also combine security mechanisms by adding 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 confidentiality. Adding this security at the message level ensures that integrity or confidentiality persist beyond the transport layer.

7.5 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, authorization, and transport layer security.

The chapter described how to implement a secure environment using the JAX-RPC technology. It discussed the JAX-RPC endpoint and client programming models, and how each model handles authentication, authorization, and transport layer security. The chapter also introduced the message-level Web service security model and provided guidelines for using this approach to security.

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