Building Web Services with Java

MAKING SENSE OF XML, SOAP, WSDL, AND UDDI

Second Edition

Steve Graham
Doug Davis
Simeon Simeonov
Glen Daniels
Peter Brittenham
Yuichi Nakamura
Paul Fremantle
Dieter König
Claudia Zentner

DEVELOPER'S LIBRARY

Sams Publishing, 800 East 96th Street, Indianapolis, Indiana 46240

Building Web Services with Java, Second Edition

Copyright © 2005 by Sams Publishing

International Standard Book Number: 0-672-32641-8

Library of Congress Catalog Card Number: 2004091343

Printed in the United States of America

First Printing: July 2004

07 06 05 4

Trademarks

Warning and Disclaimer

Bulk Sales

Associate Publisher
Michael Stephens

Acquisitions Editor
Todd Green

Development Editor
Tiffany Taylor

Managing Editor
Charlotte Clapp

Senior Project Editor
Matthew Purcell

Indexer
Larry Sweazy

Proofreader
Eileen Dennie

Technical Editors
Alan Moffet
Alan Wexelblat
Marc Goldford
Kunal Mittal

Publishing Coordinator
Cindy Teeters

Designer
Gary Adair

9

Securing Web Services

In Part I, "Web Services Basics," you saw how SOAP enables applications to interact with each other, and how Web Services Definition Language (WSDL) and Universal Description, Discovery, and Integration (UDDI) integrate applications among businesses. With Web services technologies, applications can be coupled loosely—that is, in a decentralized manner beyond the enterprise boundary. This situation suggests a new challenge related to security: Most existing technologies are only concerned with how to protect applications within a single *security domain* 📖. Soon, however, we'll have to be concerned with how to federate security domains, because each enterprise has its own security boundary. The Web services security roadmap document (referred as *the roadmap document* in this chapter) addresses this issue; it not only provides a security model, but also shows a collection of specifications to be published.

In this chapter, we'll look at the concept of Web services security and review its pending specifications. Because the specification drafts are often abstract (especially for non–security experts), we'll provide concrete examples using SkatesTown scenarios. In particular, we'll first review existing security technologies and take a closer look at the mapping from Web services security onto those technologies. In addition, we'll review how Web services security technologies are integrated into enterprise applications using the J2EE model.

Example Scenario

In our discussion of security, we'll continue our SkatesTown example. SkatesTown's CTO, Dean Caroll, is becoming concerned with security now that the business is expanding. SkatesTown is doing business with a large number of companies, and most of them aren't Fortune 500 companies. Currently, SkatesTown's Web services are secured only with Secure Socket Layer (SSL) and HTTP Basic Authentication. The combination of the two won't be enough in the future because SkatesTown has to transact with various business partners, some of which may want to use other security technologies such as Kerberos. Specifically, when SOAP messages travel through intermediaries, end-to-end security may be required. Dean can see that the combination of SSL and HTTP Basic Authentication can support point-to-point security. However, he doesn't know what kind of security mechanisms SkatesTown should support.

To ease Dean's concern, Al Rosen of Silver Bullet Consulting has been asked to advise Dean about what kind of security features to address in the next development phase. This hasn't been an easy task for Al either, because numerous security technologies and specifications are available; it won't be possible or meaningful for him to cover all of them. Therefore, he has addressed Web services security and considered how its mechanisms would fit into SkatesTown's SOAP-based transactions, as we'll present throughout this chapter.

Security Basics

In this section, we'll introduce basic security concepts and technologies that are relevant to Web services security. We'll begin by reviewing security requirements, and then discuss a collection of security technologies. Finally, we'll describe a requirement specific to Web services: the federation of security domains.

Note that we won't discuss security risks in a generic sense; instead, we'll address network security. For example, physical site security and insider problems are outside the scope of this chapter. It's also worthwhile to clarify that we're assuming machine-to-machine interactions rather than human-to-machine interactions.

Security Requirements

E-business relies on the exchange of information between trading partners over insecure networks (often the Internet). There are always security risks, because messages could be stolen, lost, or modified.

Four security requirements must be addressed to ensure the safety of information exchanged among trading partners:

- *Confidentiality* 📖 guarantees that exchanged information is protected against eavesdroppers. For example, purchase orders and invoices shouldn't be exposed to outsiders. Your credit card information shouldn't be wiretapped by third parties.
- *Integrity* 📖 refers to the assurance that a message isn't modified accidentally or deliberately in transit. For example, an invoice or order shouldn't be modified as it moves between a buyer and a seller.
- *Authentication* 📖 guarantees that access to e-business applications and data is restricted to those who can provide appropriate proof of identity. For example, when a buyer accesses a seller's site for a purchase order, the buyer is typically required to give an ID and password as proof of their identity.
- *Nonrepudiation* 📖 guarantees that the message's sender can't deny having sent it. This requirement is important when you exchange business documents such as a purchase order or bill, because document recipients want transaction records with

proof. With nonrepudiation, once a purchase order is submitted, the buyer can't repudiate it.

In addition to these requirements for message protection, you must consider how to protect resources such as data and applications such that only appropriate entities are allowed to access them. A fifth requirement is as follows:

- *Authorization* is a process that decides whether an entity with a given identity can access a particular resource. For example, authorized buyers can view a product list and submit a purchase order. The former indicates authorization on data, and the latter indicates authorization on an application—that is, an order-management system.

Message protection requires you to define how to include security information in exchanged messages. On the other hand, authorization often doesn't affect messages, but is provided as a mechanism embedded in a platform such as a Web server, an application server, or a database management system. We'll use the J2EE platform as an example to explain authorization in the "Enterprise Security" section of this chapter.

Cryptography

Cryptography technologies provide a basis for protecting messages exchanged between trading partners. Confidentiality and integrity can be ensured with *encryption* and *digital signature* technologies, respectively. These cryptography technologies can be categorized into two types in terms of an orthogonal dimension: symmetric and asymmetric keys. Table 9.1 shows four categories based on the two dimensions; the table classifies widely used algorithms. The following subsections review each category in more detail.

Table 9.1 **Classification of Cryptography Algorithms**

Technology	Symmetric Key	Asymmetric Key
Encryption	3DES, AES, RC4	RSA15
Digital signature	HMAC-SHA1, HMAC-MD5	RSA-SHA1

Symmetric Encryption

Symmetric encryption requires that you use the same key for encryption and decryption. For example, assume that Alice wants to send data to Bob. According to standard cryptography terminology, the original data is called *plaintext* and the encrypted data is called *ciphertext*. As shown in Figure 9.1, Alice encrypts the plaintext with a key to send it to Bob, and Bob decrypts the ciphertext with the same key to extract the plaintext. Because the same keys are used at the both endpoints, this kind of encryptions is referred to as symmetric, and the keys used are often called *symmetric keys*.

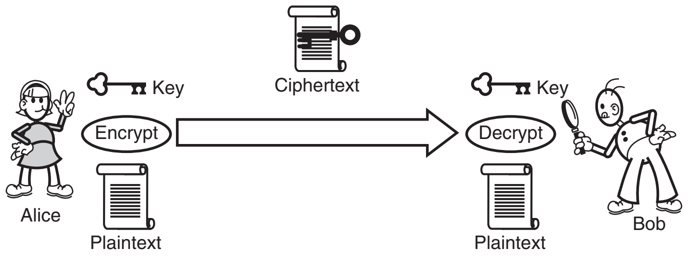

Figure 9.1 Symmetric encryption

This category of encryption includes Triple DES (3DES), which is a minor variation of the Data Encryption Standard (DES) developed by an IBM team 30 years ago; and the Advanced Encryption Standard (AES), which has been proposed as a replacement for 3DES by the National Institute of Standards and Technology (NIST). RC4, which was designed at RSA Laboratories by Ron Rivest in 1987, is also widely used with SSL.

Asymmetric Encryption

Asymmetric encryption 🕮 allows you to make your encryption key public and thus simplifies key distribution. Two different keys are used: a *public key* 🕮 and a *private key* 🕮. Assume that Bob has a pair of private and public keys, and he only publishes the public key. Alice encrypts her plaintext with the public key to send it to Bob, and Bob decrypts the ciphertext with his private key (see Figure 9.2).

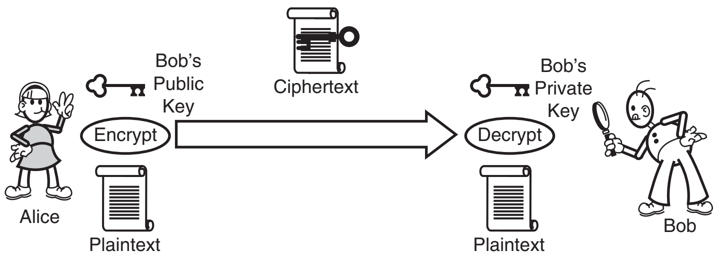

Figure 9.2 Asymmetric encryption

Unlike symmetric encryption schema, different keys are used at the endpoints. The keys are called *asymmetric keys*, reflecting their asymmetric nature. An example of an algorithm in this category is RSAES-PKCS1-v1_5 (RSA-15), which is specified in RFC 2437.

Message Authentication Code

Although the next category of security technology is symmetric digital signature, it's called *Message Authentication Code (MAC)* 🕮 in cryptography terminology. It relies on mathematical algorithms known as *hashing functions* 🕮 to ensure data integrity. A hashing function takes data as input and produces smaller data called a *digest* 🕮 as output. If the original data changes even slightly, the digest is different. MAC is an extension of this idea: A digest is created with a key in addition to the input data. Such an extension is necessary because an attacker could otherwise capture both the data and the digest, and then tamper with the data and construct a new digest. As shown in Figure 9.3, MAC requires the same key at both ends; hence Bob can check the integrity of Alice's data with the key.

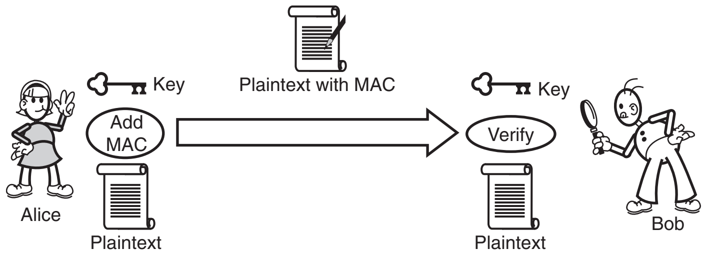

Figure 9.3 Message Authentication Code (MAC)

Keyed-Hashing for Message Authentication Code (HMAC) is an example of MAC. HMAC must be combined with hashing functions such as MD5 and SHA-1. Therefore, the algorithm names are HMAC-SHA1, HMAC-MD5, and so on, as listed in Table 9.1.

Digital Signature

The asymmetric digital signature technology is referred to as *digital signature* 📖. As shown in Figure 9.4, Alice signs the plaintext with her private key. *Signing* here means creating a signature value that's sent with the original plaintext. Bob can verify the integrity of the incoming message by generating the signature value from the plaintext with Alice's public key; he can compare this value with the signature value that accompanies the incoming plaintext.

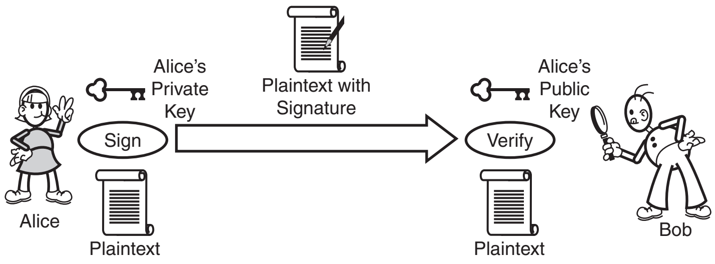

Figure 9.4 Digital signature

Like MAC algorithms, digital signature algorithms are also combined with hashing functions such as SHA-1. Table 9.1 shows an example: in RSA-SHA1, a digest is calculated with SHA-1, and a signature value on the digest is created with a private key.

You can use the digital signature technology to ensure nonrepudiation as well as integrity. In Figure 9.4, Bob can make sure that the incoming plaintext is signed by Alice, because he uses Alice's public key. However, how can he know that Alice is the holder of the public key? Public Key Infrastructure (PKI) provides a solution: An authority issues digital certificates, each of which binds a party to a public key. PKI and X.509 digital certificates are reviewed later in this chapter.

Asymmetric Versus Symmetric Technologies

Asymmetric keys may seem more useful than symmetric ones because the former can solve the issue of key distribution—symmetric keys must be transmitted and managed carefully so that attackers can't steal them. However, asymmetric keys have some limitations. One of their practical problems is performance. Asymmetric operations with private keys (decryption and signing) are a great deal slower than symmetric key operations. Even asymmetric public key operations such as encryption and signature verification are much slower than symmetric key operations. Based on such performance characteristics, it's best to combine asymmetric and symmetric key operations to take advantage of both benefits.

Authentication

Password authentication is the most commonly used authentication method on the Internet. A client shows its ID or username and password, and the server checks the ID/password pair by referring to a user registry that manages a collection of such pairs.

Password authentication to access Web servers over HTTP is called HTTP Basic Authentication (BASIC-AUTH); it's defined in RFC 2617. The specification defines an interaction protocol between Web browser and Web server in addition to how to encode the ID/password into the HTTP header (see the sidebar "HTTP Basic Authentication").

HTTP Basic Authentication

You've probably experienced being required to enter a user ID and password while visiting a Web site. This process is based on HTTP Basic Authentication (BASIC-AUTH), which is defined in RFC 2617. The typical BASIC-AUTH interaction between a Web browser and a Web server is illustrated in Figure 9.5.

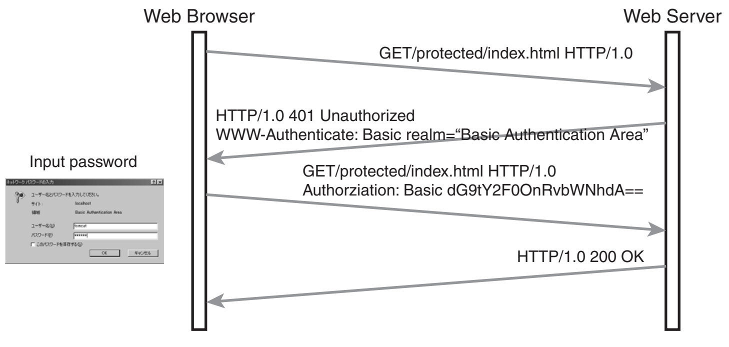
Web Browser
Web Server
GET/protected/index.html HTTP/1.0
HTTP/1.0 401 Unauthorized
WWW-Authenticate: Basic realm="Basic Authentication Area"
Input password
GET/protected/index.html HTTP/1.0
Authorziation: Basic dG9tY2F0OnRvbWNhdA==
HTTP/1.0 200 OK

Figure 9.5 Interaction protocol for HTTP Basic Authentication

When the Web browser sends an HTTP request to access a protected Web resource, the Web server returns an HTTP response that includes the error code "401 Unauthorized" and the following HTTP header:

```
WWW-Authenticate: Basic realm="Realm Name"
```

Realm is a name given to a set of Web resources; it's a unit to be protected. *Basic* in front of *realm* indicates a type of authentication—in this case, `BASIC-AUTH`. Based on this information, the Web browser shows a login dialog to the user. Then, the Web browser sends an HTTP request again, including the following HTTP header:

```
Authorization: Basic credential
```

Although the credential looks like encrypted text, it's logically plaintext because its format is *UserName*:*Password* encoded with *Base64* —for example, `dG9tY2F0OnRvbWNhdA` in Figure 9.5 can be decoded to `SkateboardWarehouse:wsbookexample`. Because BASIC-AUTH isn't secure alone, it's combined with another security mechanism such as SSL.

The Web server authenticates the user with the user ID and password included in the credential. If the given user ID and password are wrong, "401 Unauthorized" is returned. Moreover, the Web server has an *access control list* (ACL) that specifies who can access what and checks whether the authenticated user can access the Web resource. If the check succeeds, then "200 OK" is returned; otherwise, "401 Unauthorized" is returned.

The digital signature technology can also be used for authentication. As we mentioned in the previous section, a message's signature can be bound to a digital certificate. Because the certificate is bound to a particular entity such as a user or a trading partner, you can identify a holder of the certificate.

Authentication with a digital signature is in principle more convenient than password authentication because you can assume that a certificate authority manages certificates. However, client certificates aren't widely used, because it isn't easy for Web browser users to install certificates. On the other hand, server certificates are commonly used when Web browsers authenticate servers.

Security Protocols

Symmetric and asymmetric keys each have pros and cons, so combining them is a good idea. There has been a great deal of research in the area of network security to define *security protocols*, with which symmetric keys can be shared between two parties in a secure manner. In a simple protocol, two parties only agree on a key, exchanging random numbers. However, commonly used protocols also perform authentications during the key agreement process.

The Secure Socket Layer (SSL) defined by Netscape for Web browsers is the most widely used protocol on the Internet. With SSL, two parties can share a symmetric key, and authentication is also performed. The protocol works as follows:

1. The client accesses a server.
2. The server returns its certificate.

3. The client prepares a random number that is a seed for generating a symmetric key, encrypts the seed number with a public key contained in the server certificate, and sends the encrypted data to the server.
4. The server decrypts the received data to extract the seed number.
5. Both the client and the server have the same seed number, so they can generate a symmetric key from it.

Note that authentication isn't based on digital signature technology, but on encryption. After the negotiation has been completed, the client sends its application data, encrypting it with the symmetric key. If the server's response is properly encrypted, the client can authenticate the server.

The server can authenticate the client two ways. First, when the client sends a request after the SSL negotiation, the server can return an HTTP "401 Unauthorized" message to the client (see the sidebar "HTTP Basic Authentication"). Then the client attaches the username and password to be authenticated by the server. Second, the server can use another variation of the SSL negotiation protocol, where the client is required to decrypt a random number encrypted with its public key. If the decryption is performed properly, the server can authenticate the client.

Security Infrastructures

It's difficult for developers to combine security technologies properly, so *security infrastructures* 🕮 have been developed and are used today in real systems. Essentially, a security infrastructure is a basis on which applications can interact with each other securely. As illustrated in Figure 9.6, applications and the communication between them can be protected.

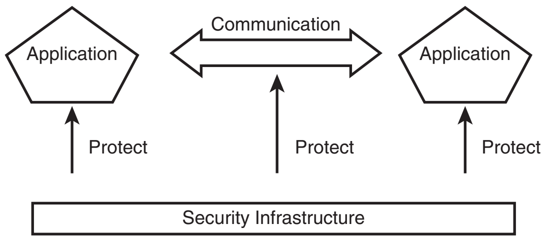

Figure 9.6 Security infrastructure

Each security infrastructure has different design requirements. Although we listed five items in the "Security Protocols" section of this chapter, not all of them have to be addressed in each infrastructure. Security infrastructures vary in terms of their design and architecture. In this section we'll review three security infrastructures that are commonly used in real-world systems: user registries, PKI, and Kerberos.

User Registries

One of the most basic security infrastructures is the *user registry* 🕮. User registries ordinarily manage user IDs and their associated passwords, which are used for authentication. As shown in Figure 9.7, an authentication module sitting in front of the applications checks each ID/password pair with the user registry. Only authenticated users can access the applications.

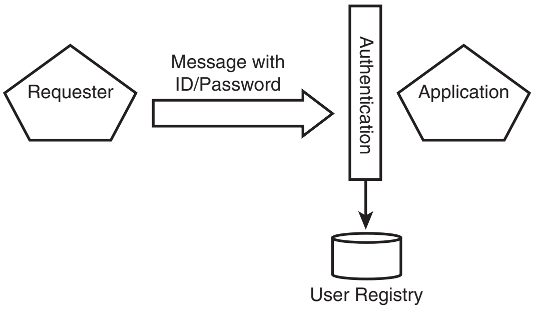

Figure 9.7 User registry for password authentication

One of the advantages of this mechanism is its simplicity. Most commonly used systems such as operating systems, database management systems, and HTTP servers incorporate user registries. Furthermore, although only authentication is involved, this simple system can be combined with other infrastructures to meet additional security requirements, if necessary.

On the other hand, user registries may be brittle. Once a password is stolen, an attacker can easily access a system using the ID and the password. In spite of such low-security level, password authentication is often chosen because its development and management are much cheaper than for sophisticated mechanisms. This decision is reasonable when the cost of any potential damage is much lower than the cost of implementing the user registry.

Public Key Infrastructure

As we reviewed in the "Cryptography" section, asymmetric key operations offer an advantage. Because public keys can be shown to anyone, you don't have to worry about key delivery. One remaining issue is how to bind a public key to a particular party. *Public Key Infrastructure (PKI)* 🕮 provides a basis to certify holders of public keys. The key constructs of PKI are the certificate and the certificate authority*:*

- A *certificate* 🕮 is a proof of identity. With a certificate, you can relate an entity such as a trading party to its public key. Because the certificate is digitally signed, you can trust its contents as long as you can trust the certificate's issuer. Although there are alternative certificate formats such as Pretty Good Privacy (PGP) and X.509, we mainly use X.509 in our examples.

- A *certificate authority* (*CA*) is an entity that issues certificates. If you can trust the CA, you can trust certificates issued by the CA. PKI assumes a fairly small number of CAs (such as VeriSign) and allows a CA to issue certificates for other CAs. As a result, certificates are organized into a hierarchy, as shown in Figure 9.8. The root is called a *root CA*, intermediary nodes are called *subordinate CAs*, and terminal nodes are called *end entities*. The path for an end entity to a root CA is called a *certificate path*. If two end entities have a common CA, they can establish trust with each other.

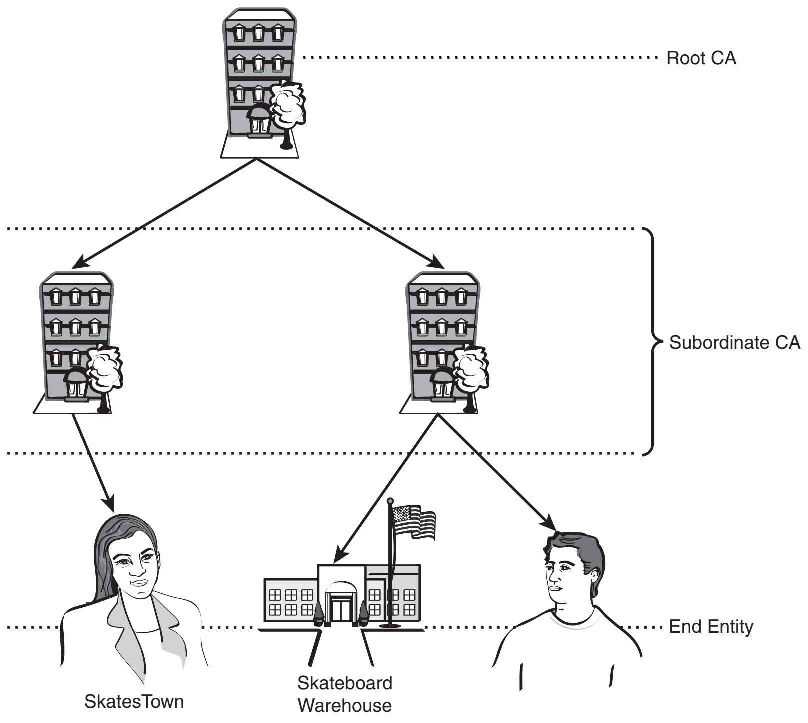

Figure 9.8 PKI trust model

Figure 9.9 illustrates how to use a certificate. First, a requestor registers its public key with the CA, and a certificate is issued. The requestor signs a message with the private key and sends the signed messages to a service provider, attaching the certificate. To verify the signature, the provider performs a cryptographic operation using a public key included in the certificate. To verify the certificate, the provider checks whether the certificate is signed by the CA, potentially traversing subordinate CAs in the process.

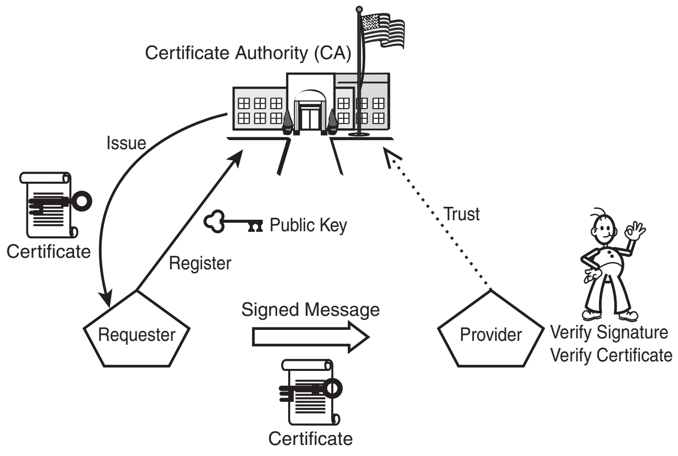

Figure 9.9 Using a certificate for a digital signature

Kerberos

Kerberos 🕮 was initially developed for workstation users who wanted to access a network. One of the key requirements is *single sign-on (SSO)* 🕮: A user provides an ID and a password only once to access various applications within a certain interval. Another requirement is no use of public key cryptography. With a symmetric key operation, you can achieve much higher performance.

The Kerberos architecture is illustrated in Figure 9.10. The requestor (Alice) first requests and receives a *ticket-granting ticket (TGT)* 🕮 through password authentication by accessing the *Key Distribution Center (KDC)* 🕮. Next, Alice requests and receives a service ticket (ST) for a provider (Bob), showing the TGT to the KDC. Alice can now access Bob by including the ST in a request message. The TGT contains Alice's ID, her session key, and TGT expiration time. Therefore, as long as the TGT is valid, Alice can get various STs without giving her ID and password. In this way, SSO is achieved.

The KDC is a core of the Kerberos system: It authenticates users to issue tickets and, more importantly, manages all participants' IDs and secret keys (called *master keys* 🕮). For example, when issuing an ST for Bob, the KDC encrypts Alice's information with Bob's key. As a result, only Bob can decrypt Alice's ID in the ST, and therefore he can authenticate her. The ST contains a session key between Alice and Bob, so they can securely exchange messages with encryption and digital signatures.

Security Domains

Before moving on to Web services security, let's define the term *security domain* 🕮. As you've seen, each security infrastructure has a scope of management in terms of participants and resources. A user registry or a Kerberos KDC has explicit participant databases. On the other hand, the root CA implicitly prescribes a set of participants in PKI. We call such scope a security domain. Note that security domains can be different even if they're based on the same security infrastructure. For example, there are multiple root CAs in PKI, and certificates that have different root CAs can't trust each other.

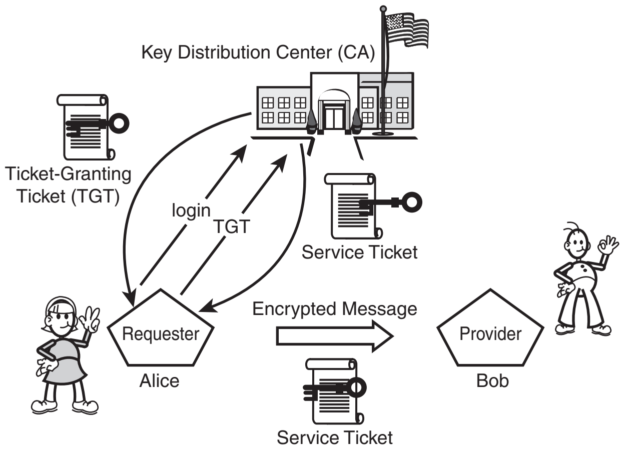

Figure 9.10 Kerberos architecture

As you can imagine, numerous security domains exist in the real world, and there is no point in considering a single security infrastructure to integrate them. Web services security—which is a main topic of this chapter—addresses how to integrate security domains that are often based on different security infrastructures.

Web Services Security

Because applications are integrated across business boundaries according to the Web services concept, Web services have specialized security requirements. As discussed in the "Security Domains" section of this chapter, numerous security domains already exist on the Internet and in intranets. Although applications and business entities belong to one or more security domains, they have to interact with one another beyond the security domain boundary. Thus, the *federation* 📖 of different security domains is extremely important, although the issue hasn't been addressed in existing security technologies. The Web services security architecture discussed in the roadmap document addresses the issue of integration of security domains.

This section examines the overall architecture of Web services security. In subsequent sections, we'll take a closer look at the security specifications.

Security Model for Web Services

Each business has its own security infrastructure and mechanism, such as PKI or Kerberos. In the context of Web services, these security systems need to interoperate over different security domains. The Web services security architecture defines an abstract model for that purpose.

As shown in Figure 9.11, three parties are identified: the requestor, the Web service, and the security token service. Each has its own claims, security token, and policy. The roadmap document defines several terms to discuss the security model, as follows:

- *Subject*—A principal (for example, a person, an application, or a business entity) about which the claims expressed in the security token apply. The subject, as the owner of the security token, possesses information necessary to prove ownership of the security token.
- *Claim*—A statement about a subject either by the subject or by a relying party that associates the subject with the claim. This specification doesn't attempt to limit the types of claims that can be made, nor does it attempt to limit how these claims may be expressed. Claims can be about keys that may be used to sign or encrypt messages. Claims can be statements the security token conveys. For example, a claim may be used to assert the sender's identity or an authorized role.
- *Security token*—A representation of security-related information (X.509 certificate, Kerberos ticket and authenticator, mobile device security token from a SIM card, username, and so on).
- *Web service endpoint policy*—Collectively, the claims and related information that Web services require in order to process messages (the Web Services have complete flexibility in specifying these claims). Endpoint policies may be expressed in XML and can be used to indicate requirements related to authentication (for example, proof of user or group identity), authorization (such as proof of execution capabilities), or other requirements.

Based on this terminology, the highly abstracted security model is designed to fit many diverse situations.

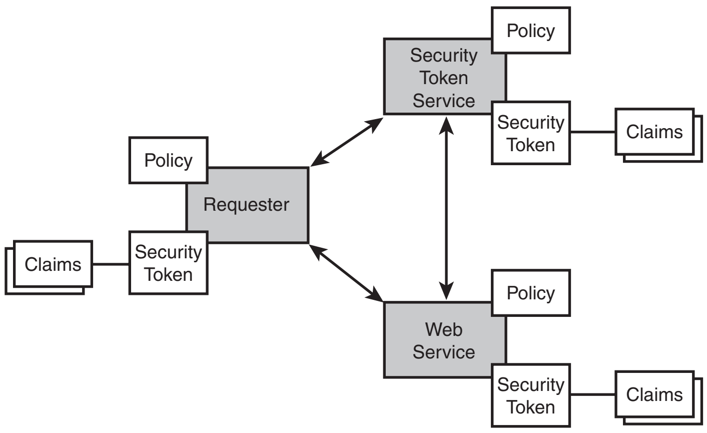

Figure 9.11 Security model for Web services

Let's examine the security model with some example scenarios. Assume that a requestor wants to invoke a Web service. The requestor has claims such as its identity and its privileges. On the other hand, the invoked Web service has a policy that requires encryption of messages and authentication of the requestor. Note that the word *policy* isn't a general term, but has a specific meaning such as "security requirements to access the Web service." Therefore, the requestor has to send messages that meet the security policy.

When you're sending security claims, you have to consider how to represent them in messages. The Web services security model suggests that all claims are included in the security token that's attached to request messages. For example, identity via a password or X.509 certificate is a security claim; therefore it's represented as a security token.

Although the requestor creates a security token, some security tokens (such as X.509 certificates) must be issued by a third party. Such a third party is called a *Security Token Service* (STS) in the security model (see Figure 9.11). The certificate authority in PKI and the Key Distribution Center in Kerberos are good examples. Because the STS is a Web service, it has security policies, claims, and security tokens. Most security systems include security servers, each of which manages its security domain. The STS is an abstraction of such security servers.

The Web services security model attempts to define an abstraction of existing security mechanisms, aiming at a generic security model. An opposite approach would be to define a generic security *mechanism*. There are security mechanisms running already, and we don't want to replace all of them. Therefore, instead of the mechanism, we use an abstract model with which we can unite different mechanisms without changing them.

Web Services Security Specifications

No Web services security specifications have been finalized yet during the writing of this book. Rather, draft specifications have been published, and some of them are being standardized in OASIS. However, a diagram of the specification release is summarized in the roadmap document, as shown in Figure 9.12:

- *WS-Security* defines how to include security tokens in SOAP messages and how to protect messages with digital signatures and encryption.
- *WS-Policy* provides a framework for describing Web services meta-information. Based on the framework, domain-specific languages can be defined, such as WS-SecurityPolicy (described later).
- *WS-Trust* prescribes an interaction protocol to access Security Token Services.
- *WS-SecureConversation* defines a security context with which parties can share a secret key to sign and encrypt parts of messages efficiently.
- *WS-Federation* provides a framework for federating multiple security domains. For example, it defines how to get a temporary identity to access a Web service in another security domain.
- *WS-Privacy* provides a framework for describing the privacy policy of Web services.

- *WS-Authorization* defines how to exchange authorization information among parties. The authorization is defined as a security token.

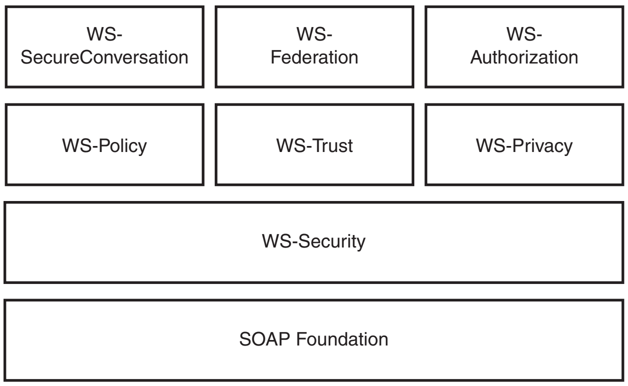

Figure 9.12 Roadmap of Web services security specifications

As of the time of this writing, draft specifications of WS-Authorization and WS-Privacy haven't been released. Other specifications are discussed in the remainder of this chapter.

Extended SkatesTown Security Scenario

Although SkatesTown's CTO, Dean Caroll, has an overview of the Web services security model, he doesn't realize how these technologies can help his business. Therefore, he asked Al Rosen of Silver Bullet Consulting to show him some scenarios that employ Web services security.

Al Rosen envisioned a buyer/seller/supplier scenario that might happen in the near future. Figure 9.13 shows the addition of a credit card company. A typical transaction is carried out as follows:

1. Buyer submits a purchase order to SkatesTown, specifying a product, a delivery date, and a credit card number.
2. SkatesTown allocates stock to Part Supplier to confirm the order.
3. If the stock is allocated, SkatesTown checks with Credit to determine whether the buyer's purchase is possible.
4. SkatesTown returns an invoice to Buyer.

Let's consider how Web services security helps protect each process in the transaction. When accepting the purchase order message, SkatesTown considers three security requirements: nonrepudiation, authentication, and confidentiality. Note that integrity is not a requirement, but nonrepudiation ensures integrity. WS-Security can be used to satisfy these requirements.

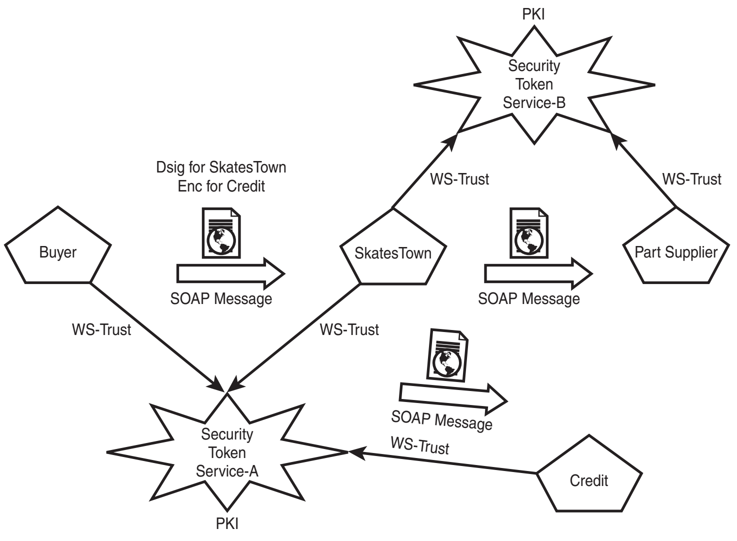

Figure 9.13 Extended SkatesTown scenario for security

A digital signature is applied to Buyer's order information as part of WS-Security. During order submission, SkatesTown wants to ensure nonrepudiation so that Buyer can't deny their request. If Buyer signs the order with its private key, SkatesTown can verify the signature with Buyer's public key.

SkatesTown also can verify Buyer's identity with Buyer's certificate. Through the signature verification, SkatesTown already has proof of possession on the certificate. Thus, authentication can be performed during the signature verification.

The card information involves a special requirement: SkatesTown can't see Buyer's number, but Credit can see it. In order to achieve this requirement, they can use the encryption feature provided in WS-Security. Specifically, the card number is encrypted with Credit's public key by Buyer and then decrypted with Credit's private key by Credit.

WS-Trust can be viewed as an API to access an STS. In this scenario, when SkatesTown receives a signed purchase order, it verifies not only the signature, but also the certificate used for the signature. In this case, SkatesTown may access an STS (STS-A in Figure 9.13) with WS-Trust in order to validate the certificate.

With WS-Policy and WS-SecurityPolicy, administrators of Web services can specify security requirements to access their services. SkatesTown requires a digital signature for purchase order submission. If such a security policy is published, buyers will be prepared to insert digital signatures into their messages.

WS-SecureConversation is used to establish a security context; it fits into the message exchange between SkatesTown and Supplier. SkatesTown interacts with many buyers but only a small number of suppliers, but the number of messages it exchanges with suppliers is much larger. Although you can apply WS-Security directly using PKI, such a solution isn't efficient because public key operations are expensive. In this case, it's a good idea to use WS-SecureConversation to establish a security context, and then communicate with partners, utilizing efficient symmetric key operations.

WS-Federation is required for authentication across security boundaries. Assume that Buyer and SkatesTown belong to security domain A, and Credit and SkatesTown belong to security domain B. In this case, Credit wants to authenticate Buyer, although Buyer can show a security token issued in domain A. WS-Federation provides a framework for resolving such federation issues as described later.

No draft specification has been published for WS-Privacy or WS-Authorization, but we'll briefly describe how they fit into this scenario. With WS-Privacy, SkatesTown could define a privacy policy dictating that purchase order information must not be used for other purposes (such as direct mail). Under such a policy, Buyer would decide whether to submit an order. WS-Privacy is provided as a specific language of WS-Policy, like WS-SecurityPolicy. This way, buyers can get the privacy policy in advance when they submit a purchase order.

WS-Authorization would define a security token for authorization. In our scenario, Supplier would have access control on its stock allocation service and would require a particular authorization token. The authorization token would prove a rank for the requestor, such as AAA or BB. Supplier would then specify the maximum amount of the transaction according to the ranking. For example, $10,000 total allocation might be allowed for an AAA requestor, but only $3,000 might be allowed for a BB requestor.

In the following sections, we'll review the specifications in more detail.

WS-Security

The WS-Security specification defines a format to include security tokens and mechanisms to protect SOAP messages. Digital signatures serve as integrity and/or nonrepudiation checks to ensure message protection, and encryption guarantees confidentiality. In addition, WS-Security provides a flexible mechanism to include various claims in SOAP messages with security tokens. With message protection and security tokens, WS-Security can provide a basis for other specifications in the roadmap.

Listing 9.1 illustrates the syntax of WS-Security. A `Security` element is defined and included in a SOAP header. Under the `Security` element can appear a `Signature` element (defined in the XML Digital Signature specification), an encryption-related element such as `EncryptedKey` (defined in the XML Encryption specification), and security tokens such as `UsernameToken` (defined in WS-Security). The following sections review digital signatures, encryption, and security tokens.

Listing 9.1 **Basic Syntax of WS-Security**

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" >
   <S:Header>
      <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/06/secext" >
         <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
         </Signature>
```

Listing 9.1 **Continued**

```
         <EncryptedKey xmlns="http://www.w3.org/2001/04/enc-enc-enc#">
         </EncryptedKey>
         <wsse:UsernameToken xmlns="http://schemas.xmlsoap.org/ws/2003/06/secext">
         </wsse:UsernameToken>
      </wsse:Security>
  </S:Header>
   <S:Body>
    ...
   </S:Body>
</S:Envelope>
```

Digital Signatures

Returning to the example scenario, Dean Caroll now understands an immediate problem: Without nonrepudiation, the buyer can deny their purchase order even if the buyer sent the order request, or the buyer can claim that the number of products ordered is wrong. With respect to the exchange of messages between two parties, a digital signature provides a means to prove that the sending party created the message. Al Rosen emphasized that WS-Security incorporates the *World Wide Web Consortium (W3C)* 🕮/IETF standard, the XML Digital Signature specification.

The XML Digital Signature specification defines how to sign part of an XML document in a flexible manner. Whereas in the "Cryptography" section we assumed that data is signed, we are now concerned with how to create the signed data from an XML document. The specification defines ways to specify parts of the document and accompanying canonicalization methods, as we'll review later. The specification also permits signature algorithms. As we discussed earlier, digital signatures and Message Authentication Code (MAC) are similar—the difference only involves their use of asymmetric and symmetric keys. Based on this similarity, both digital signatures and MAC are handled in an integrated manner in the specification. For example, you can specify HMAC-SHA1 (see Table 9.1) as a signature algorithm. In that case, you can only ensure integrity—that is, you can only ensure that the message hasn't been modified during transmission.

In our extended example, Buyer sends a purchase order document and receives an invoice document. In practice, these two documents should be signed; otherwise, one of the parties can repudiate that it sent the document. Listing 9.2 shows a digitally signed purchase order document. The `Signature` element includes a signature on the purchase order and specifies a collection of parameters to create the signature.

Listing 9.2 **Digital Signature Sample**

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC=http://schemas.xmlsoap.org/soap/encoding/
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
```

Listing 9.2 **Continued**

```
   <SOAP-ENV:Header>
      <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/06/secext">
         <wsse:BinarySecurityToken xmlns:wsu=
         "http://schemas.xmlsoap.org/ws/2003/06/utility"
         EncodingType="wsse:Base64Binary" ValueType="wsse:X509v3" wsu:Id="bst_id">
        MIIDQTCCAqqgAwIBAgICAQQwDQYJKoZIhvcNAQEFBQAwTjELMAkGA1UEBhMCSlAxETAP
        BgNVBAgTCEthbmFnYXdhMQwwCgYDVDQQKEwNJQk0xDDAKBgNVBAsTA1RSTDEQMA4GA1UE
        ...
         </wsse:BinarySecurityToken>
         <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
           <SignedInfo>
             <CanonicalizationMethod Algorithm=
            "http://www.w3.org/2001/10/xml-exc-c14n#">
               </CanonicalizationMethod>
            <SignatureMethod Algorithm=
           "http://www.w3.org/2000/09/xmldsig#rsa-sha1">
             </SignatureMethod>
             <Reference URI="#body_id">
               <Transforms>
                <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
                </Transform>
               </Transforms>
               <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1">
               </DigestMethod>
               <DigestValue>U2BIJSk6OL0W0mGXXiGVn5XPV54=</DigestValue>
             </Reference>
           </SignedInfo>
           <SignatureValue>
          Ojjw8nkT3jJoNN/AxsdOwTqWnhfZewubBWp0Sa0vJTTjQBrnKR18brODc8byuwVf2v
          iFdvMY4mT7Iumk/ZRLRNF1tEBCFRki2++W2LIXBIXVtmwo1riS98kmFZo6dBvhFOnX
          wKE1ag6C8x/UgAMVU+YzYd11KqNXtpwvi9Ydoq4=
           </SignatureValue>
           <KeyInfo>
              <wsse:SecurityTokenReference>
                <wsse:Reference URI="#bst_id"></wsse:Reference>
              </wsse:SecurityTokenReference>
           </KeyInfo>
         </Signature>
      </wsse:Security>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body xmlns:wsu="http://schemas.xmlsoap.org/ws/2003/06/utility"
   wsu:Id=" body_id">
      <po xmlns="http://www.skatestown.com/ns/po" id="43871" submitted=
      "2004-01-05" customerId="73852">
         <billTo>
```

Listing 9.2 **Continued**

```
            <company>The Skateboard Warehouse</company>
            <street>One Warehouse Park</street>
            <street>Building 17</street>
            <city>Boston</city>
            <state>MA</state>
            <postalCode>01775</postalCode>
         </billTo>
         <shipTo>
            <company>The Skateboard Warehouse</company>
            <street>One Warehouse Park</street>
            <street>Building 17</street>
            <city>Boston</city>
            <state>MA</state>
            <postalCode>01775</postalCode>
         </shipTo>
         <order>
            <item sku="318-BP" quantity="5">
               <description>Skateboard backpack; five pockets</description>
            </item>
            <item sku="947-TI" quantity="12">
               <description>Street-style titanium skateboard.</description>
            </item>
            <item sku="008-PR" quantity="1000">
            </item>
         </order>
      </po>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In the XML Digital Signature specification (XML Signature), an element `Signature` is defined with its descendants under the namespace `http://www.w3.org/2000/09/xmldsig#`. The WS-Security specification defines how to embed the `Signature` element in SOAP messages as a header entry. You can sign all or part of the message. In our example, the body part is signed. The digest value of the body part is calculated, and the value is signed and included in the `Signature` element.

Let's review how to get the digest value of the target. The target is specified by `Reference` under the `SignedInfo` element. Its `URI` attribute indicates the target in such a way that the `id` attribute of the `body` element is referenced. The target is transformed by `EXC-C14N`—that is, an exclusive *canonicalization* method for XML. (Exclusive canonicalization is a W3C Recommendation to generate a canonical form for physically different but logically equivalent XML documents.) With EXC-C14N, you can check whether XML documents are semantically equivalent using a standardized code set, the order of attributes, tab processing, and so on.

Let's look at a canonicalization example. The following two documents appear quite different at a glance:

```
<?xml version="1.0" encoding="us-ascii"?>
<foo
   b="b"
   a="a"
></foo>

<?xml version="1.0" encoding="us-ascii"?>
<foo a="a" b="b"/>
```

However, with canonicalization, they're both translated into the following document:

```
<?xml version="1.0" encoding="us-ascii"?>
<foo a="a" b="b"></foo>
```

The rules applied here are as follows:

- White spaces and new line feeds in a begin tag are normalized to a single white space.
- Attributes in a begin tag are sorted in alphabetical order.
- An empty element is converted to a start-end tag pair.
- Characters are encoded with UTF-8 (although you can't see it in the printed text).

There are two specifications for XML canonicalization: XML-C14N and EXC-C14N. The key difference is how to handle namespaces. Specifically, the outer element namespace can affect the canonicalization of the inner elements with XML-C14N. However, this behavior isn't good when you want to insert a signed part into another XML document, such as a SOAP envelope. For this reason, in WS-Security, EXC-C14N is recommended rather than XML-C14N.

After translation with EXC-C14N, a digest value is calculated with an algorithm specified by the `DigestMethod` element. Here *SHA1* is used. The calculated value is inserted in the `DigestValue` element, represented in Base64 format.

The value of the target isn't signed directly. Rather, the `SignedInfo` element is signed. An algorithm specified by the `CanonicalizationMethod` element—that is, XML-C14N—canonicalizes `SignedInfo`. The canonicalized `SignedInfo` is signed with an algorithm specified by `SignatureMethod`: RSA-SHA1. This algorithm calculates a digest value of the `SignedInfo` subtree and then signs it with an RSA private key. The calculated value is inserted into the `SignatureValue` element, represented in Base64 format.

Optionally, the signer can include a `KeyInfo` element to attach key information. More specifically, the example includes a reference (via `SecurityTokenReference` and `Reference` elements) to a `BinarySecurityToken` element that contains an X.509 certificate. (Security tokens are discussed in more detail in the "Security Tokens" section.)

So far, we've reviewed the XML Signature syntax in the signature-processing process. Verification is carried out in the same manner. First, you check the value of the `DigestValue` element according to EXC-C14N and SHA1. Next, you calculate a digest value for the `SignedInfo` subtree to compare it with the value in the `SignatureValue` element. More precisely, the signature value is decrypted with the public key and then compared to the calculated value.

Encryption

In our scenario, credit card information should be encrypted, because SkatesTown doesn't have to know the card number. The XML Encryption specification defines a means to encrypt portions of XML documents; this selective encryption feature is incorporated into WS-Security.

We can now update the purchase order document to include credit card information, as shown in Listing 9.3. Instead of a `billTo` element, we insert `cardInfo` so that the service requestor can pay with the card. When it receives the document from Buyer, SkatesTown doesn't have to know the credit card information. Rather, SkatesTown wants verify with Credit that SkatesTown can charge the purchase on the card. In order to achieve this scenario, the credit card information must be encrypted in such a way that only Credit can decrypt it.

Listing 9.3 **Purchase Order that Includes Card Information**

```
<po xmlns="http://www.skatestown.com/ns/po-with-card" id="43871"
submitted="2004-01-05" customerId="73852">
   <cardInfo>
      <name>The Skateboard Warehouse</name>
      <company>VISA</company>
      <expiration>02/2005</expiration>
      <number>1234123412341234</number>
   </cardInfo>
   <shipTo>
      <company>The Skateboard Warehouse</company>
      <street>One Warehouse Park</street>
      <street>Building 17</street>
      <city>Boston</city>
      <state>MA</state>
      <postalCode>01775</postalCode>
   </shipTo>
   <order>
      <item sku="318-BP" quantity="5">
         <description>Skateboard backpack; five pockets</description>
      </item>
      <item sku="947-TI" quantity="12">
         <description>Street-style titanium skateboard.</description>
      </item>
```

Listing 9.3 **Continued**

```
      <item sku="008-PR" quantity="1000">
      </item>
   </order>
</po>
```

XML Encryption Example

Let's look at a simple XML Encryption sample first. Listing 9.4 is an encrypted version of the purchase order document. In this example, we assume that two parties share a common symmetric key indicated by a key name. Note that the namespace `ds` is a prefix for the XML Digital Signature namespace; the XML Encryption specification reuses elements from the XML Digital Signature namespace as much as possible.

Listing 9.4 **Encrypted Purchase Order Document**

```
<po xmlns="http://www.skatestown.com/ns/po-with-card"
id="43871" submitted="2004-01-05" customerId="73852""">
   <enc:EncryptedData
      Type="http://www.w3.org/2001/04/xmlenc#Element">
      <enc:EncryptionMethod
         Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
      <ds:KeyInfo>
         <ds:KeyName>Shared key</ds:KeyName>
      </ds:KeyInfo>
      <enc:CipherData>abCdeF...</enc:CipherData>
   </enc:EncryptedData>
   <shipTo>
      <company>The Skateboard Warehouse</company>
      <street>One Warehouse Park</street>
      <street>Building 17</street>
      <city>Boston</city>
      <state>MA</state>
      <postalCode>01775</postalCode>
   </shipTo>
   <order>
      <item sku="318-BP" quantity="5">
         <description>Skateboard backpack; five pockets</description>
      </item>
      <item sku="947-TI" quantity="12">
         <description>Street-style titanium skateboard.</description>
      </item>
      <item sku="008-PR" quantity="1000">
      </item>
   </order>
</po>
```

The `EncryptedData` element is a root element for the encrypted part, and its `Type` attribute indicates that the encrypted data is an XML element. The `EncryptionMethod` element specifies an encryption algorithm, and `KeyInfo` specifies a secret key. Based on the secret key and the algorithm, the credit card information is encrypted and stored in the `CipherData` element. The data to be encrypted must be a portion of the XML document encoded with UTF-8.

Listing 9.5 shows encryption with a public key.

Listing 9.5 **Encryption with a Public Key**

```
<po xmlns="http://www.skatestown.com/ns/po-with-card" id="43871"
submitted="2004-01-05" customerId="73852">
   <enc:EncryptedData
      Type="http://www.w3.org/2001/04/xmlenc#Element">
      <enc:EncryptionMethod
         Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
      <ds:KeyInfo>
         <enc:EncryptedKey>
            <enc:EncryptionMethod
               Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
            <ds:KeyInfo>
               <ds:KeyName>Receiver's key</ds:KeyName>
            </ds:KeyInfo>
            <enc:CipherData>ghIjkL...</enc:CipherData>
         </enc:EncryptedKey>
      </ds:KeyInfo>
      <enc:CipherData>abCdeF...</enc:CipherData>
   </enc:EncryptedData>
   <shipTo>
      <company>The Skateboard Warehouse</company>
      <street>One Warehouse Park</street>
      <street>Building 17</street>
      <city>Boston</city>
      <state>MA</state>
      <postalCode>01775</postalCode>
   </shipTo>
   <order>
      <item sku="318-BP" quantity="5">
         <description>Skateboard backpack; five pockets</description>
      </item>
      <item sku="947-TI" quantity="12">
         <description>Street-style titanium skateboard.</description>
      </item>
      <item sku="008-PR" quantity="1000">
      </item>
   </order>
</po>
```

The idea here is that a random symmetric key is generated to encrypt the data, and the symmetric key itself is encrypted with the receiver's public key. Let's first look at how the symmetric key is encrypted. `EncryptedKey` includes the encrypted symmetric key, specifying how it's encrypted. `EncryptionMethod` specifies the encryption algorithm, and the inner `KeyInfo` specifies the receiver's public key. Based on these elements, the encrypted key is stored in `CipherData`. The outer `CipherData` comes from an encryption based on the symmetric key and an encryption algorithm specified by the outer `EncryptionAlgorithm`.

WS-Security Example

Let's move on to encryption in WS-Security. Listing 9.6 shows a WS-Security example for encryption. In addition to the encrypted data in the body, the `EncryptedKey` element is located under the `Security` element. This indicates that the header element is used as an instruction to process the body.

`EncryptedKey` contains an encrypted symmetric key and a reference to the encrypted body. `CipherData` contains a symmetric key encrypted with the RSA-1.5 algorithm and a public key identified in the `KeyIdentifier` element. When you specify a reference to an X.509 certificate with `KeyIdentifier`, you have to use the `SubjectKeyIdentifier` attribute in the X.509 certificate.

The decrypted symmetric key is used to process the `EncryptedData` element in the body. `DataReference` in `ReferenceList` has a reference to the `EncryptedData` element.

Listing 9.6 **WS-Security Encryption** Ex**ample**

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Header>
      <wsse:Security xmlns:wsse=
      "http://schemas.xmlsoap.org/ws/2003/06/secext"
      SOAP-ENV:mustUnderstand="1">
         <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">
            <EncryptionMethod Algorithm=
            "http://www.w3.org/2001/04/xmlenc#rsa-1_5"></EncryptionMethod>
            <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
               <wsse:SecurityTokenReference>
                  <wsse:KeyIdentifier>u3AA1M+DMOAlbX/vWJWnFtOKBck=
                  </wsse:KeyIdentifier>
            </wsse:SecurityTokenReference>
           </KeyInfo>
           <CipherData>

<CipherValue>cdck0cWh94oF5xBoEm9x/LjjJfmfnVn3SmhryPr5Rui/Y5tJQz8hQq
➥729vPHETtKWwwRBkpkp6wqFlHztCw2h
```

Listing 9.6 **Continued**

```
KMBMubZzPTODzzgAU0ZvbHtjRKtqPnNuq3ZDYDGQ9RBIfyjPyVdwrwlPaR9eaXtmbLK/G3e3iGaxAW4jh
➥Lq+wM=</CipherValue>
           </CipherData>
           <ReferenceList>
              <DataReference URI=
              "#wssecurity_encryption_id_519136303015631520_1045115597786">
              </DataReference>
           </ReferenceList>
         </EncryptedKey>
      </wsse:Security>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
      <po xmlns="http://www.skatestown.com/ns/po-with-card" id="43871"
      submitted="2004-01-05" customerId="73852">
         <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
         Id="wssecurity_encryption_id_519136303015631520_1045115597786"
         Type="http://www.w3.org/2001/04/xmlenc#Content">
            <EncryptionMethod Algorithm=
            "http://www.w3.org/2001/04/xmlenc#tripledes-cbc">
            </EncryptionMethod>
            <CipherData>
              <CipherValue>Ew7Zggr8z3/uFGzKVNP69SPSij+Y65L/jyk5sggKcKjkBv1hip5npg
              ==</CipherValue>
            </CipherData>
         </EncryptedData>
         <shipTo>
            <company>The Skateboard Warehouse</company>
            <street>One Warehouse Park</street>
            <street>Building 17</street>
            <city>Boston</city>
            <state>MA</state>
            <postalCode>01775</postalCode>
         </shipTo>
         <order>
            <item sku="318-BP" quantity="5">
               <description>Skateboard backpack; five pockets</description>
            </item>
            <item sku="947-TI" quantity="12">
               <description>Street-style titanium skateboard.</description>
            </item>
            <item sku="008-PR" quantity="1000">
            </item>
         </order>
      </po>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The `EncryptedKey` in the header is intended to be a directive to message recipients. The recipients can know in advance which portions are encrypted and how to decrypt them. In this scenario, the recipient isn't SkatesTown but Credit. Therefore, Buyer should use Credit's public key so that SkatesTown can't decrypt the data. Furthermore, the `Security` element that contains `EncryptedKey` should have an `actorURI` to specify a particular recipient.

Java Cryptography Extension

To implement digital signatures and encryption for WS-Security, you need to use *Java Cryptography Extension (JCE)*. Let's review JCE to help you understand Java's cryptography architecture.

JCE provides a framework for accessing and developing core cryptographic functions. Implementation and algorithm independence are addressed so that applications are insulated from cryptographic details. In other words, you can change the cryptographic implementation and algorithms without modifying applications.

Implementation independence is achieved through the security provider architecture. If you look at the file `java-home>\jre\lib\security\java.security`, you'll find the following format:

```
security.provider.<n>=<Security Provider Class>
```

The number indicates a priority, and the right side specifies a security provider class. Each provider class is a subclass of `java.security.Provider` and supports some or all Java security algorithms, such as DSA, RSA, MD5, and SHA-1. For example:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.rsajca.Provider
security.provider.3=com.sun.net.ssl.internal.ssl.Provider
```

Note that this example also specifies an SSL provider with priority 3. When a security algorithm is required, providers are asked whether they support the particular algorithm according to the priority. For example, if `SHA1withRSA` is required, the second provider is chosen. Although the second and third providers both support it, the second one has higher priority. If SSL is required, the third provider is chosen because only it supports SSL.

Even if you want to use another provider, such as IBM JCE, you don't have to change your program. Rather, you only have to change the security configuration file.

Security Tokens

In addition to message protection with digital signatures and encryption, WS-Security defines security tokens, which can contain various requestors' claims (such as a username and optional password, an X.506 certificate, or a Kerberos ticket). As we reviewed earlier in this chapter, a key purpose of the Web services security model is to integrate various security infrastructures and domains. In these security infrastructures, security claims are represented in different ways; therefore, you need an abstraction to integrate these differences.

A security token is an XML representation that can contain security claims. Because the security token can contain any mechanism-specific security data, it serves as the integration of different security infrastructures. Let's take a closer look at two types of tokens: `UsernameToken` and `BinarySecurityToken`. In addition, we'll review other tokens that can be embedded in the WS-Security `Security` element.

UsernameToken

The simplest security token is `UsernameToken`, which contains a mandatory `Username` and an optional `Password`. Listing 9.7 shows an example.

Listing 9.7 `UsernameToken` **Example**

```
<wsse:UsernameToken>
   <wsse:Username>testName</wsse:Username>
   <wsse:Password>testPassword</wsse:Password>
</wsse:UsernameToken>
```

`UsernameToken` is used for password authentication, such as HTTP Basic Authentication (see the sidebar "HTTP Basic Authentication"). From a security point of view, this plaintext representation is extremely insecure. Therefore, BASIC-AUTH needs to be used with security protection methods such as WS-Security encryption and SSL/TLS (see the sidebar).

Without a password, `UsernameToken` can be viewed as ID assertion. If you have a secured intranet, ID assertion is enough. If you have a gateway server that maps external IDs to internal IDs, the downstream server only needs the internal ID represented with `UsernameToken`.

BinarySecurityToken

Unlike `UsernameToken`, some tokens such as X.509 certificates and Kerberos tickets are represented as binary data. `BinarySecurityToken` is defined to contain such binary data. Look at the format in Listing 9.8. The `Id` attribute with the `wsu` prefix is used for referencing from another place in the SOAP message. The `ValueType` attribute specifies the kind of data. In this example, `X509v3` indicates an X.509 v3 digital certificate; you can also specify `Kerberos5TGT` for a Kerberos ticket-granting ticket and `Kerberos5ST` for a Kerberos service ticket. The `EncodingType` attribute specifies the encoding format of the binary data. Because `Base64Binary` is specified, a Base64 representation of an X.509 certificate is included in the `BinarySecurityToken` element.

Listing 9.8 `BinarySecurityToken` **Example**

```
<wsse:BinarySecurityToken xmlns:wsu=
"http://schemas.xmlsoap.org/ws/2003/06//utility"
EncodingType="wsse:Base64Binary" ValueType="wsse:X509v3"
wsu:Id="wssecurity_binary_security_token_id_2343669525027134511_1045057262242">
```

Listing 9.8 **Continued**

```
      MIIDQTCCAqqgAwIBAgICAQQwDQYJKoZIhvcNAQEFBQAwTjELMAkGA1UEBhMCSlAxETAP
      BgNVBAgTCEthbmFnYXdhMQwwCgYDVQQKEwNJQk0xDDAKBgNVBAsTA1RSTDEQMA4GA1UE
     ...
</wsse:BinarySecurityToken>
```

As in Listing 9.2, X.509 certificates are often combined with digital signatures. Because any third party can take a certificate and include it in their messages, the certificate alone can't serve as *proof of possession* 📖. Therefore, the X.509 certificate in a binary security token should be used for authentication only when combined with an XML Signature.

Other Security Tokens

WS-Security only provides a framework to include security tokens, mentioning how to use them with signatures and encryption; defining concrete tokens is outside the scope of the specification. Rather, security tokens are defined as separate profiles. As of December 2003, profiles for username token and X.509 have been published by OASIS. For example, "Web Services Security: X509 Token Profile" defines the value type for the `ValueType` attribute of the `BinarySecurityToken` element. As you saw in the previous section, X509v3 is defined to indicate an X.509 v3 digital certificate.

In addition, "Web Services Security: Kerberos Binding" is also being discussed. It defines Kerberos5TGT for a Kerberos ticket-granting ticket and Kerberos5ST for a Kerberos service ticket. Like the X.509 sample, a Kerberos ticket can be contained in `BinarySecurityToken` as a Base64 representation, specifying either of the Kerberos value types in `ValueType`.

The Secure Assertion Markup Language (SAML) is another candidate that will be embedded in WS-Security as a security token. SAML lets you represent security assertions in XML format. The assertions are similar to claims in WS-Security, and they can relate to authentication, authorization, or attributes of entities (either human or computer). "Web Services Security: SAML Binding" defines how to represent such assertions as security tokens in accordance with WS-Security.

In order to understand security tokens from a broader perspective, let's consider a security context defined in WS-SecureConversation. As we'll review later, WS-SecureConversation defines `SecurityContextToken` and key derivation mechanisms. The security context is like that in SSL and is established through a security handshake. It's important to note that the context is considered a security token in the specification. This abstraction contributes to simplifying digital signatures and encryption with the security context, as you'll see in the section on WS-SecureConversation.

Security tokens are a key concept of abstraction in WS-Security. Like the security context, you can define various types of security tokens. If you define new tokens, you can use them in WS-Security—that is, you can use a new token for digital signatures and encryption. This way, you can extend WS-Security by defining your own tokens for future use.

WS-Trust

WS-Trust defines how to request and issue security tokens and how to establish trust relationships. At its heart, WS-Trust assumes a security token service that issues security tokens and manages a security domain. In Figure 9.14, we assume that STS-A is a certificate authority (CA) in PKI and STS-B is a Key Distribution Center (KDC) in Kerberos. In this section, we'll review PKI and Kerberos as examples, mapping them onto WS-Trust. Through these examples, you'll see how the STS concept abstracts existing security infrastructures. We'll also review the XML Key Management Specification (XKMS), relating it to WS-Trust.

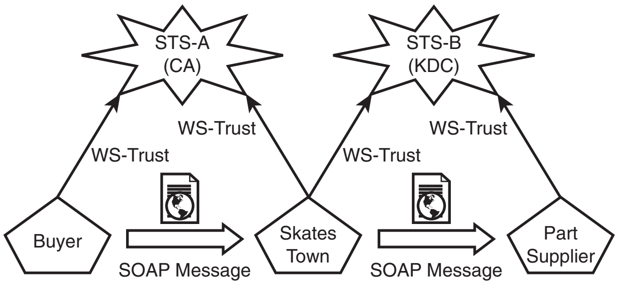

Figure 9.14 Using PKI and Kerberos with WS-Trust

Public Key Infrastructure

In terms of WS-Trust, a CA can be considered a Security Token Service (STS). Of course, in that case, the CA must implement WS-Trust. Let's review some WS-Trust examples, assuming that STS-A in Figure 9.14 is a CA. In our scenario, Buyer requires a Credit public key to encrypt credit card information. In this case, Buyer can get a Credit X.509 certificate from STS-A through WS-Trust.

As shown in Listing 9.9, the WS-Trust request is contained in the SOAP body. The `RequestSecurityToken` element is used for requesting tokens, and its child elements specify details. `wsse:X509v3` in `TokenType` indicates that an X.509 certificate is involved. `wsse:ReqIssue` in `RequestType` indicates that the required action is to issue a token. The `Base` element references a base token that is used to validate the authenticity of a request. In this example, `UsernameToken` is referred to in the header; thus the ID and password are used for authentication. Furthermore, because we need to get a Credit certificate, we want to specify it within the request. WS-Trust doesn't define how to specify a target party, so we use `AppliesTo`, which is defined in WS-Policy.

Listing 9.9 **Request to Issue an X.509 Certificate**

```
<S:Envelope xmlns:S="..." xmlns="../secext" xmlns:wsu="../utility>
   <S:Header wsu:Id="req">
```

Listing 9.9 **Continued**

```
      <wsse:Security>
         <wsse:UsernameToken wsu:Id='Me' >
            <wsse:Username>Buyer</wsse:Username>
            <wsse:Password>buyerPW</wsse:Password>
         </wsse:UsernameToken>
       </wsse:Security>
    </S:Header>
    <S:Body>
       <wst:RequestSecurityToken>
          <wst:TokenType>wsse:X509v3</wst:TokenType>
          <wst:RequestType>wsse:ReqIssue</wst:RequestType>
          <wst:Base>
             <wsse:Reference URI='#Me'
                   ValueType='wsse:UsernameToken' />
          </wst:Base>
          <wsp:AppliesTo
             xmlns:wsp='http://schemas.xmlsoap.org/ws/2002/12/policy' >
             <wsa:EndpointReference>
                <wsa:Address>http://credit.com/service</wsa:Address>
             </wsa:EndpointReference>
          </wsp:AppliesTo>
       </wst:RequestSecurityToken>
    </S:Body>
</S:Envelope>
```

Listing 9.10 shows a response example. `RequestedSecurityToken` in `RequestSecurityTokenResponse` contains a token. Because a Credit X.509 certificate is returned, `BinarySecurityToken` is used to contain it in a Base64 representation.

Listing 9.10 **Response for an X.509 Certificate Request**

```
<S:Envelope xmlns:S="..." xmlns=".../secext" xmlns:wsu=".../utility>
   <S:Body wsu:Id="req">
       <RequestSecurityTokenResponse>
          <RequestedSecurityToken>
            <BinarySecurityToken ValueType="wsse:X509v3"
                                 EncodingType="wsse:Base64Binary">
                    MIIEZzCCA9CgAwIBAgIQEmtJZc0...
            </BinarySecurityToken>
         </RequestedSecurityToken>
       </RequestSecurityTokenResponse>
   </S:Body>
</S:Envelope>
```

Kerberos

As shown in Figure 9.14, SkatesTown and Part Supplier are managed by a Kerberos security domain in our scenario. As we discussed earlier, users first get a ticket-granting ticket (TGT) via password authentication, next get a Service Ticket (ST) for a particular service, and then can access the service with the ST (see Figure 9.10). Once the user gets a TGT, they can get other STs for other services without further authentication. In this way, single sign-on (SSO) is achieved.

Requests from a client to KDC can be represented in a WS-Trust request. Listing 9.11 shows a request to issue a Kerberos TGT. Password authentication is required, so `UsernameToken` is included and referenced from the `Base` element. `wsse:Kerberos5TGT` is specified in `TokenType`.

Listing 9.11 **Request for Issuing a Kerberos TGT**

```
<S:Envelope ......>
   <S:Header>
      <Security>
          <UsernameToken wsu:Id="myToken">
             <Username>SkatesTown</Username>
              <Password>pwd</Password>
          </UsernameToken>
      </Security>
    </S:Header>
    <S:Body>
       <RequestSecurityToken>
           <TokenType>wsse:Kerberos5TGT</TokenType>
           <RequestType>wsse:ReqIssue</RequestType>
           <Base>
            <Reference URI='#myToken'
                   ValueType='wsse:UsernameToken'/>
           </Base>
       </RequestSecurityToken>
    </S:Body>
</S:Envelope>
```

The client can sign and/or encrypt request messages based on the ST. Listing 9.12 shows an example of a signature with a Kerberos ST. Notice that there isn't a significant difference between this and the X.509 signature example (Listing 9.2); the `ValueType` attribute in `BinarySecurityToken` is the only change. However, unlike PKI, a Kerberos ticket contains a shared secret; therefore HMAC-SHA1 is used for the signature processing instead of RSA-SHA1.

Listing 9.12 **Signature with Kerberos Service Ticket**

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
```

Listing 9.12 **Continued**

```
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Header>
      <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/06/secext"
      SOAP-ENV:mustUnderstand="1">
         <wsse:BinarySecurityToken xmlns:wsu=
         "http://schemas.xmlsoap.org/ws/2003/06//utility"
         EncodingType="wsse:Base64Binary" ValueType="wsse:Kerberos5ST"
         wsu:Id=
         "wssecurity_binary_security_token_id_2343669525027134511_1045057262242">
        MIIDQTCCAqqgAwIBAgICAQQwDQYJKoZIhvcNAQEFBQAwTjELMAkGA1UEBhMCSlAxETAP
        BgNVBAgTCEthbmFnYXdhMQwwCgYDVQQKEwNJQk0xDDAKBgNVBAsTA1RSTDEQMA4GA1UE
        AxMHSW50IENBMjAeFw0wMTEwMDEwOTU0MDZaFw0xMTEwMDEwOTU0MDZaMFQxCzAJBgNV
        BAYTAkpQMREwDwYDVQQIEwhLYW5hZ2F3YTEMMAoGA1UEChMDSUJNMQwwCgYDVQQLEwNU
        UkwxFjAUBgNVBAMTDVNPQVBSZXF1ZXN0ZXIwgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJ
        AoGBAMy3PfZ1mPhrEsBvYiOuIlPV3Uis5Yy6hmxo2YwYC2nNDBPzKslWUi/Q+fK+DNdY
        6KEHmuDrcVcEma48J9X1a5avRlksQfKptKoVn4eBys2i/wkwyzQhDaFji79/MvnTRW8E
        Vy99FNKw4PFnhOoe1tlDcNBuIH/fIuGOz9ElTV+fAgMBAAGjggEmMIIBIjAJBgNVHRME
        AjAAMAsGA1UdDwQEAwIF4DAsBglghkgBhvhCAQ0EHxYdT3BlblNTTCBHZW5lcmF0ZWQg
        Q2VydGlmaWNhdGUwHQYDVR0OBBYEFIW3FD1cXie4j4zw1gAp4cuOAZ4lMIG6BgNVHSME
        gbIwga+AFL35INU4+WRy09vaf9zOsP7QvO9voYGSpIGPMIGMMQswCQYDVQQGEwJKUDER
        MA8GA1UECBMIS2FuYWdhd2ExDzANBgNVBAcTBllhbWF0bzEMMAoGA1UEChMDSUJNMQww
        CgYDVQQLEwNUUkwxGTAXBgNVBAMTEFNPQVAgMi4xIFRlc3QgQ0ExIjAgBgkqhkiG9w0B
        CQEWE21hcnV5YW1hQGpwLmlibS5jb22CAgEBMA0GCSqGSIb3DQEBBQUAA4GBAHkthdGD
        gCvdIL9/vXUo74xpfOQd/rr1owBmMdb1TWdOyzwbOHC7lkUlnKrkI7SofwSLSDUP571i
        iMXUx3tRdmAVCoDMMFuDXh9V7212luXccx0s1S5KN0D3xW97LLNegQC0/b+aFD8XKw2U
        5ZtwbnFTRgs097dmz09RosDKkLlM
         </wsse:BinarySecurityToken>
         <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
            <SignedInfo>
               <CanonicalizationMethod Algorithm=
               "http://www.w3.org/2001/10/xml-exc-c14n#"></CanonicalizationMethod>
               <SignatureMethod Algorithm=
               "http://www.w3.org/2000/09/xmldsig#hmac-sha1"></SignatureMethod>
               <Reference URI=
               "#wssecurity_body_id_2934309014555244973_1045057262232">
                  <Transforms>
                     <Transform Algorithm=
                     "http://www.w3.org/2001/10/xml-exc-c14n#"></Transform>
                  </Transforms>
                  <DigestMethod Algorithm=
                  "http://www.w3.org/2000/09/xmldsig#sha1"></DigestMethod>
                  <DigestValue>U2BIJSk6OL0W0mGXXiGVn5XPV54=</DigestValue>
               </Reference>
            </SignedInfo>
            <SignatureValue>
```

Listing 9.12 **Continued**

```
          Ojjw8nkT3jJoNN/AxsdOwTqWnhfZewubBWp0Sa0vJTTjQBrnKR18brODc8byuwVf2v
          iFdvMY4mT7Iumk/ZRLRNF1tEBCFRki2++W2LIXBIXVtmwo1riS98kmFZo6dBvhFOnX
          wKE1ag6C8x/UgAMVU+YzYd11KqNXtpwvi9Ydoq4=
            </SignatureValue>
            <KeyInfo>
               <wsse:SecurityTokenReference>
                  <wsse:Reference URI="#wssecurity_binary_security_token_id_
➥2343669525027134511_1045057262242"></wsse:Reference>
               </wsse:SecurityTokenReference>
            </KeyInfo>
         </Signature>
      </wsse:Security>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body xmlns:wsu="http://schemas.xmlsoap.org/ws/2003/06/utility"
   wsu:Id="wssecurity_body_id_2934309014555244973_1045057262232">
      <po xmlns="http://www.skatestown.com/ns/po" id="43871"
      submitted="2004-01-05" customerId="73852">
         <billTo>
            <company>The Skateboard Warehouse</company>
            <street>One Warehouse Park</street>
            <street>Building 17</street>
            <city>Boston</city>
            <state>MA</state>
            <postalCode>01775</postalCode>
         </billTo>
         <shipTo>
            <company>The Skateboard Warehouse</company>
            <street>One Warehouse Park</street>
            <street>Building 17</street>
            <city>Boston</city>
            <state>MA</state>
            <postalCode>01775</postalCode>
         </shipTo>
         <order>
            <item sku="318-BP" quantity="5">
               <description>Skateboard backpack; five pockets</description>
            </item>
            <item sku="947-TI" quantity="12">
               <description>Street-style titanium skateboard.</description>
            </item>
            <item sku="008-PR" quantity="1000">
            </item>
         </order>
      </po>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

XML Key Management Specification

Although WS-Trust provides a framework to access security token services, it doesn't encompass all aspects of key management: retrieval, registration, backup, revocation, and recovery. The XML Key Management Specification (XKMS) defines concrete operations for key managements, addressing PKI. In this section, we'll review XKMS and discuss how it relates to WS-Trust.

With XKMS, applications can interact using PKI without focusing on the myriad of fine details in PKI, such as ASN.1. There is some overlap between XKMS and WS-Trust, although they're being developed independently.

XKMS was published as a W3C Note in March 2001 by VeriSign, Microsoft, and WebMethods. XKMS has two major components: the XML Key Information Service Specification (X-KISS) and the XML Key Registration Service Specification (X-KRSS). One of the main goals of XKMS is to complement emerging W3C standards, such as XML Digital Signature and XML Encryption.

Let's go back to our example scenario, where Buyer requires a Credit X.509 certificate for encrypting credit card information. The WS-Trust request in Listing 9.9 can be represented as a XKMS request as shown in Listing 9.13. The key value of the public key can be retrieved via X-KISS, as in Listing 9.14.

Listing 9.13 **Query for Retrieving a Key Value**

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlso ap.org/soap/envelope/">
   <SOAP-ENV:Body>
      <Locate xmlns="http://www.xkms.org/schema/xkms-2001-01-20">
         <Query>
            <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
               <KeyName>
           ➥CN=Purchase Order Client, OU=Purchase Department,
           ➥O=SkateboardWarehouse, L=..., S=NY, C=US</KeyName>
            </KeyInfo>
         </Query>
         <Respond>
            <string>KeyValue</string>
         </Respond>
      </Locate>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The `Locate` element includes a query on `KeyInfo` and a format for the response. This message requests a public key for the Distinguished Name (DN) specified by the `KeyName` element. Note that WS-Trust doesn't define how to specify the DN. Listing 9.14 shows its response.

Listing 9.14 **Response to the Key Value Query**

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" >
   <SOAP-ENV:Body>
      <LocateResult xmlns="http://www.xkms.org/schema/xkms-2001-01-20">
         <Result>Success</Result>
         <Answer>
            <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
               <KeyValue>
                  <DSAKeyValue>
                     <P>
                        /X9TgR11EilS30qcLuzk5/YRt1I870QAwx4/gLZRJmlFXUAiUftZPY1
                ➥Y+r/F9bow9sbVWzXgTuAHTRv8mZgt2uZUKWkn5/oBHsQIsJPu6nX/rf
                ➥GG/g7V+fGqKYVDwT7g/bTxR7DAjVUE1oWkTL2dfOuK2HXKu/yIgMZnd
                ➥FIAcc=
                     </P>
                     <Q>l2BQjxUjC8yykrmCouuEC/BYHPU=</Q>
                     <G>
                        9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+Z
                ➥xBxCBgLRJFn
                        Ej6EwoFhO3zwkyjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+
                ➥2J6ASQ7zKTx
                        vqhRkImog9/hWuWfBpKLZl6Ae1UlZAFMO/7PSSo=

                     </G>
                     <Y>
                        Q9N/x1cj2LSaV9ZdKPl0Sl9HhqbBdloc/AvxvY41sQREau9s/HmPwFd
                ➥Tgn6iRCdXrg
                        Y2HaiQYOlBdt09UW+q2XjvY1vdrWhXlxy8VdSFEdMCla926o38igZjF
                ➥qXF0LOlBKTK
                        LQTsCzWWxDB6sK8LkvaUikUFpudYa/rWP562GUI=
                     </Y>
                  </DSAKeyValue>
               </KeyValue>
            </KeyInfo>
         </Answer>
      </LocateResult>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The value of the public key is included in the `KeyValue` element. With this value, you can verify the signature of the initial SOAP message.

As shown in the example, X-KISS helps applications obtain cryptographic key information. In addition to key value retrieval, it can be used to validate a binding between a key name and a key value. It's also useful for getting key information from an X.509 certificate. In a complementary way, X-KRSS provides key registration, revocation, and recovery services.

As you've seen, there is overlap between WS-Trust and XKMS. Conceptually, WS-Trust is a superset of X-KISS because it can cover other security mechanisms such as Kerberos. On the other hand, XKMS defines retrieval operations more concretely than WS-Trust, addressing PKI. Furthermore, X-KRSS operations such as key registration and revocation aren't considered in WS-Trust. At this moment, we don't know whether these specifications will converge, although duplicated functions should ideally be converged.

WS-SecurityPolicy

When Buyer accesses SkatesTown services, it has to know the security requirements (the security policy) in advance. As we reviewed in Chapter 4, "Describing Web Services," WS-Policy and WS-PolicyAttachment provide a framework to describe policies and to associate them with particular services. WS-SecurityPolicy is a domain-specific language to represent policies for WS-Security. For example, you can describe your desired policy in such a way that a signature is required on a particular element and that a particular element must be encrypted.

In our scenario, SkatesTown requires a signature on Buyer's purchase order and therefore wants to represent this requirement as a policy. As in Listing 9.2, we assume that a signature is required on the SOAP `Body` element. Listing 9.15 shows a sample policy for `Signature`.

Listing 9.15 **WS-SecurityPolicy Sample for Signature**

```
<wsp:Policy xmlns:wsp="..." xmlns:wsse="...">
  <wsp:All wsp:Preference="100">
    <wsse:Integrity wsp:Usage="wsp:Required">
      <wsse:Algorithm  Type="wsse:AlgCanonicalization"
                       URI="http://www.w3.org/Signature/Drafts/xml-exc-c14n"/>
        <wsse:Algorithm Type="wsse:AlgSignature"
                       URI=" http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
        <MessageParts
                 Dialect="http://schemas.xmlsoap.org/2002/12/wsse#soap">
            S:Body
        </MessageParts>
    </wsse:Integrity>
    <wsse:SecurityToken>
       <wsse:TokenType>wsse:X509v3</wsse:TokenType>
    </wsse:SecurityToken>
  </wsp:All>
</wsp:Policy>
```

Although XML Signature can be used for integrity and nonrepudiation, statements on signatures are represented with the `Integrity` element. Specifically, when you use PKI-based signatures, you can ensure the nonrepudiation of the signed message. On the other hand, if you use shared-secret-based signatures, you can't ensure nonrepudiation—only integrity.

In addition, WS-SecurityPolicy provides statements for confidentiality and security tokens. Listing 9.15 contains a `SecurityToken` statement for an X.509 certificate.

From an implementation point of view, one of the difficult aspects is verifying that necessary parts are signed or encrypted. WS-SecurityPolicy provides two mechanisms. You can use XPath to specify any parts of the message. Although this is a generic mechanism, its computational cost is often high. Therefore, if the part selection can be represented by the message part selection functions in WS-PolicyAssertion, it's recommended that you use those functions. `S:Body` in Listing 9.15 is an example of such a function.

WS-SecureConversation

SkatesTown and Part Supplier interact with each other frequently. Although you can use public-key-based signatures and encryption for such interactions, you can easily get into a performance problem. With WS-SecureConversation, the two parties can have a shared secret, making possible more effective signature and encryption algorithms.

Let's change our scenario a little, defining a security domain with PKI for SkatesTown and Part Supplier. Based on X.509 certificates, the two organizations establish a security context that contains shared secrets. With the shared secrets, they can interact with each other, ensuring integrity and confidentiality (see Figure 9.15).

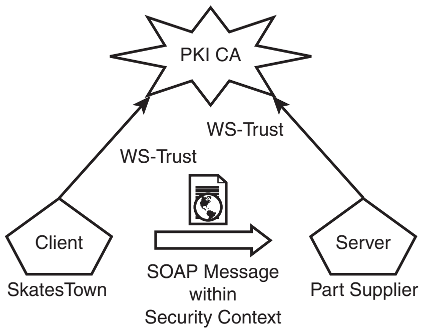

Figure 9.15 Secure conversation with PKI

The security context idea is similar to that in SSL/TLS: first establish a security context via handshake protocols, and then have a secure interaction with the security context. In this section, after we examine WS-SecureConversation, we'll review the SSL protocol in detail. Then, we'll consider a concrete authentication protocol based on WS-SecureConversation.

WS-SecureConversation Overview

WS-SecureConversation defines a format for security context mechanisms to establish a security context, and for mechanisms to derive session keys from security contexts. Let's look at an example, shown in Listing 9.16.

Listing 9.16 **Signature Based on** `SecurityContextToken`

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd=
http://www.w3.org/2001/XMLSchema
xmlns:wsu=http://schemas.xmlsoap.org/ws/2003/06/utility
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Header>
      <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/06/secext"
      SOAP-ENV:mustUnderstand="1">
         <wsse:SecurityContextToken wsu:Id="SecContext">
            <wsu:Identifier>uuid:...</wsu:Identifier>
         </wsse:SecurityContextToken>
         <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
            <SignedInfo>
               <CanonicalizationMethod Algorithm=
               "http://www.w3.org/2001/10/xml-exc-c14n#"></CanonicalizationMethod>
               <SignatureMethod Algorithm=
               "http://www.w3.org/2000/09/xm]ldsig#rsa-sha1"></SignatureMethod>
               <Reference URI=
               "#wssecurity_body_id_2934309014555244973_ 1045057262232">
                  <Transforms>
                     <Transform Algorithm=
                     "http://www.w3.org/2001/10/xml-exc-c14n#"></Transform>
                  </Transforms>
                  <DigestMethod Algorithm=
                  "http://www.w3.org/2000/09/xmldsig#sha1"></DigestMethod>
                  <DigestValue>U2BIJSk6OL0W0mGXXiGVn5XPV54=</DigestValue>
               </Reference>
            </SignedInfo>
            <SignatureValue>
          Ojjw8nkT3jJoNN/AxsdOwTqWnhfZewubBWp0Sa0vJTTjQBrnKR18brODc8byuwVf2v
          iFdvMY4mT7Iumk/ZRLRNF1tEBCFRki2++W2LIXBIXVtmwo1riS98kmFZo6dBvhFOnX
          wKE1ag6C8x/UgAMVU+YzYd11KqNXtpwvi9Ydoq4=
            </SignatureValue>
            <KeyInfo>
               <wsse:SecurityTokenReference>
                  <wsse:Reference URI="#SecContext"></wsse:Reference>
               </wsse:SecurityTokenReference>
            </KeyInfo>
         </Signature>
```

Listing 9.16 **Continued**

```
      </wsse:Security>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body wsu:Id="wssecurity_body_id_2934309014555244973_1045057262232">
      <orderSupplies xmlns=
      "http://www.wheelsandboards.com/services/orderSupplies">
         <item sku="318-BP" quantity="5"/>
         <item sku="947-TI" quantity="12"/>
         <item sku="008-PR" quantity="1000"/>
      </orderSupplies>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A security context is represented as a `SecurityContextToken` element. An `Identifier` element contains an ID for a shared secret, which is represented as a UUID or URI. Because this message is sent from a client to a server after the establishment of the security context, both parties can retrieve the secret from the ID. Note that the security context is considered a security token. As a result, the context element can be referred to from the `Reference` element within `KeyInfo` (boldface in the listing).

Although this example illustrates how to use a security context, we have to establish the context in advance. WS-SecureConversation defines three ways to do this:

- A security token service creates a `SecurityContextToken`.
- One of the communicating parties (particularly the initiating party) creates a `SecurityContextToken`.
- A `SecurityContextToken` is created through negotiation between the parties.

The first and second ways should be obvious. We'll discuss the third approach in more detail later in this section, using a sample negotiation protocol.

In Listing 9.16, the shared secret is used as a key for XML Signature. However, the specification recommends using keys derived from the shared secret. Listing 9.17 shows a sample for the derived key. `DerivedKeyToken` indicates how to derive a key from a particular shared secret. In this case, `wsse:PSHA1` specified in the `wsse:Algorithm` attribute indicates a P_SHA-1 function defined for the TLS specification. With this function, a derived key is generated from a secret, a label, and a seed. The secret is shared already, and the label and seed are included in the `Properties` element. Because the P_SHA-1 function takes two parameters, the secret, label, and seed are used as follows in the function:

```
P_SHA1 (secret, label + seed)
```

Although this function generates a new secret, you can generate other secrets, applying the function to the generated secret repeatedly. You can specify how many times you need to apply the function with the `Generation` element, as in the example.

Listing 9.17 **Using a Derived Key from** `SecurityContextToken`

```
<SOAP-ENV:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Header>
      <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/06/secext"
      SOAP-ENV:mustUnderstand="1">
         <wsse:SecurityContextToken wsu:Id="SecContext"
            <wsu:Identifier>uuid:...</wsu:Identifier>
         </wsse:SecurityContextToken>
         <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
            <SignedInfo>
               <CanonicalizationMethod Algorithm=
               "http://www.w3.org/2001/10/xml-exc-c14n#"></CanonicalizationMethod>
               <SignatureMethod Algorithm=
               "http://www.w3.org/2000/09/xmldsig#rsa-sha1"></SignatureMethod>
               <Reference URI=
               "#wssecurity_body_id_2934309014555244973_1045057262232">
                  <Transforms>
                     <Transform Algorithm=
                     "http://www.w3.org/2001/10/xml-exc-c14n#"></Transform>
                  </Transforms>
                  <DigestMethod Algorithm=
                  "http://www.w3.org/2000/09/xmldsig#sha1"></DigestMethod>
                  <DigestValue>U2BIJSk6OL0W0mGXXiGVn5XPV54=</DigestValue>
               </Reference>
            </SignedInfo>
            <SignatureValue>
          Ojjw8nkT3jJoNN/AxsdOwTqWnhfZewubBWp0Sa0vJTTjQBrnKR18brODc8byuwVf2v
          iFdvMY4mT7Iumk/ZRLRNF1tEBCFRki2++W2LIXBIXVtmwo1riS98kmFZo6dBvhFOnX
          wKE1ag6C8x/UgAMVU+YzYd11KqNXtpwvi9Ydoq4=
            </SignatureValue>
            <KeyInfo>
               <wsse:DerivedKeyToken wsse:Algorithm="wsse:PSHA1">
                  <wsse:SecurityTokenReference>
                     <wsse:Reference URI="#SecContext"></wsse:Reference>
                  </wsse:SecurityTokenReference>
                  <Properties>
                  <Label>NewLabel</Label>
                  <Nonce>FHFE...</Nonce>
                  </Properties>
                  <wsse:Generation>2</wsse:Generation>
               </wsse:DerivedKeyToken>
            </KeyInfo>
         </Signature>
```

Listing 9.17 **Continued**

```
      </wsse:Security>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body wsu:Id="wssecurity_body_id_2934309014555244973_1045057262232">
      <orderSupplies xmlns=
      "http://www.wheelsandboards.com/services/orderSupplies">
         <item sku="318-BP" quantity="5"/>
         <item sku="947-TI" quantity="12"/>
         <item sku="008-PR" quantity="1000"/>
      </orderSupplies>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In the WS-SecureConversation specification, the description of the establishment of the security context is a little abstract. In particular, no concrete negotiation protocol is mentioned. In the following section, we'll review the SSL protocol as an example of negotiation, and then we'll represent a protocol with WS-SecureConversation.

The SSL Protocol

SSL was proposed by Netscape Communications and has been used since the widespread adoption of the World Wide Web, because it's supported by Netscape Navigator and Microsoft Internet Explorer. The latest version of SSL, 3.0, has been presented to the *Internet Engineering Task Force* (IETF) for standardization.

Note

Another closely related protocol called *Transport Layer Security* (TLS) is currently on version 1.0, published as RFC 2246. There are no major differences between SSL and TLS. TLS hasn't yet become widely used, so SSL 3.0 is still dominant.

Figure 9.16 illustrates how a security *handshake* establishes a secure connection between the client and server. Once the handshake completes, the server and client have a common secret key with which data is encrypted and decrypted. In other words, SSL uses the public key(s) to encrypt exchange messages for the sole purpose of generating the shared secret key.

Despite the advantages of using public key encryption alone, SSL combines public and shared key encryption. It does so because the public key encryption system takes more time to encrypt and decrypt messages than the secret key encryption system. Thus, the combination used by SSL takes advantage of both the easy maintenance of public key encryption and the quicker operating speed of secret key encryption.

Let's take a closer look at the SSL handshake protocol, again referring to Figure 9.16. At phase I, the client starts the handshake and then sends a random number, a list of supported ciphers, and compression algorithms. At phase II, the server selects a cipher and a compression algorithm and notifies the client. Then it sends another random number and a server certificate (which includes a public key). At phase III, the client sends a premaster secret to the server, encrypting it with the server public key. Finally, the client might send a client certificate. Now the handshake is completed.

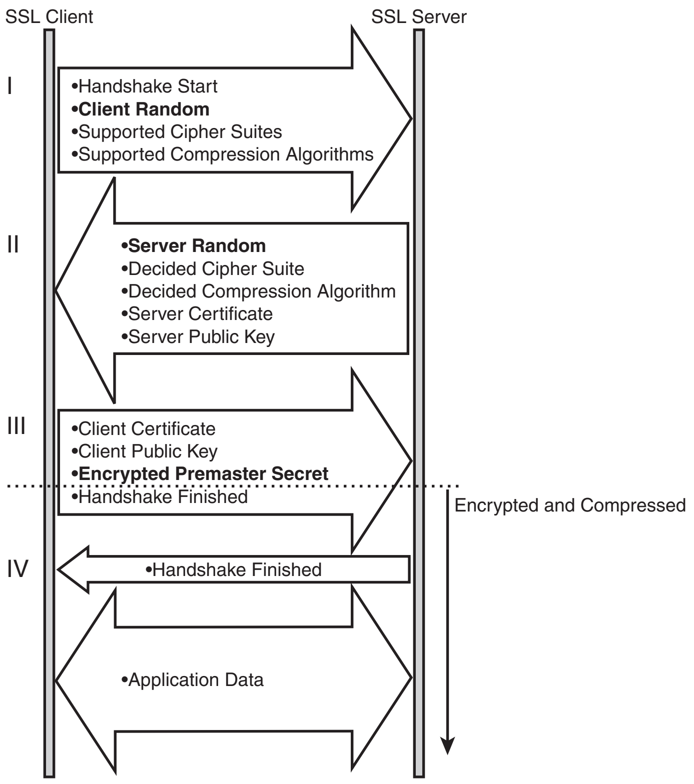

Figure 9.16 SSL security handshake protocol

The server and the client each generate a master secret by combining the random number the server sent, the random number the client sent, and the premaster secret. Several secret keys are created from the master secret. For example, one is used for encrypting transmitted data, and another is used for calculating the digest value of the date for integrity.

SSL ensures authentication (by verifying the certificates), confidentiality (by encrypting the data with a secret key), and integrity (by digesting the data). However, nonrepudiation isn't ensured with SSL because the MAC value of the transmitted data is calculated with a common secret key.

Negotiation Protocol Example

The security handshake protocol as in SSL can be represented with WS-Trust and WS-SecureConversation. WS-Trust defines a framework for challenge-response protocols, and

WS-SecureConversation defines a format for the security context token and a key derivation from a shared secret.

The idea discussed here seems similar to one in SSL. However, you don't stick to point-to-point security as in SSL. This protocol can be applied to a situation where you have a SOAP intermediary node. You can use the protocol even if two participants belong to different security domains. Thus handshake protocols in Web services security offer great advantages, especially when services and requestors are deployed on different security infrastructures and domains.

Figure 9.17 gives an overview of the protocol discussed here. The challenge-response protocol is defined with an initial `RequestSecurityToken` message and subsequent `RequestSecurityTokenResponse` messages. Note that a response message is also sent from an initial sender (Alice) to the service provider (Bob) at step 3. The protocol is performed as follows:

1. Alice sends a `RequestSecurityToken` to Bob to initiate the negotiation.
2. Bob prepares three numbers, *p, g*, and *r1*, and returns them to Alice. Simultaneously, he generates a secret *Rb* that is never exposed.
3. Alice prepares a random number *r2* and a secret *Ra*, and calculates *X*. Then she sends *X*, *r1*, and *r2* to Bob, signing them with her PKI private key. Note that $[X]_{Alice}$ indicates that *X* is signed with Alice's key.
4. Bob calculates *Y*, and sends *Y*, *r1*, and *r2*, signing them with his PKI private key. At the same time, a security context token containing an ID for a shared secret is sent.

The shared secrets are calculated with $K=X^{Y} \bmod p$ and $K=Y^{X} \bmod p$ respectively, and the calculated numbers are the same. Therefore, with the shared secret key *K*, Alice and Bob can interact with each other securely. (Note that the protocol here is prepared only for this demonstration; it isn't meant for use in a real application.)

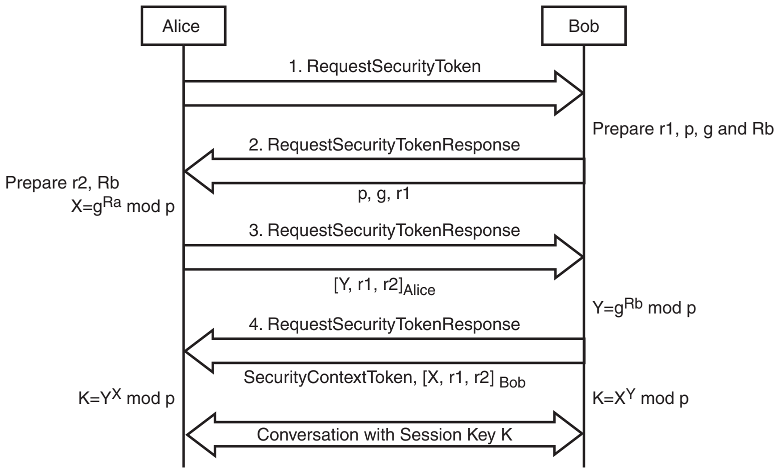
Alice
Bob
1. RequestSecurityToken
Prepare r1, p, g and Rb
2. RequestSecurityTokenResponse
Prepare r2, Rb
X=g^Ra mod p
p, g, r1
3. RequestSecurityTokenResponse
[Y, r1, r2]Alice
Y=g^Rb mod p
4. RequestSecurityTokenResponse
SecurityContextToken, [X, r1, r2] Bob
K=Y^X mod p
K=X^Y mod p
Conversation with Session Key K

Figure 9.17 Security handshake protocol

Let's review the four messages in this scenario. Listing 9.18 shows the first message. This is a request represented in WS-Trust, and `SecurityContextToken` is specified in the `TokenType` element.

Listing 9.18 **Message 1**

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body  xmlns:wsu="http://schemas.xmlsoap.org/ws/2003/06/utility">
      <wst:RequestSecurityToken xmlns:wst=
      "http://schemas.xmlsoap.org/ws/2002/12/secext">
         <wst:TokenType>wsse:SecurityContextToken</wst:TokenType>
         <wst:RequestType>wsse:ReqIssue</wst:RequestType>
      </wst:RequestSecurityToken>
   </S:Body>
</soapenv:Envelope>
```

The second message, shown in Listing 9.19, is a first response from Bob. Because this response is considered a challenge from Bob, the `SignChallenge` element is included. We define an element `ExValue` that can contain exchanged values between Alice and Bob. In this case, *p*, *g*, and *r1* are included in `val:P`, `val:G`, and `val:R1`, respectively.

Listing 9.19 **Message 2**

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <wst:RequestSecurityTokenResponse xmlns:wst=
      "http://schemas.xmlsoap.org/ws/2002/12/secext">
         <wst:SignChallenge>
            <wst:Challenge>
               <val:ExValue xmlns:val="http://dumml.org/exvalue">
                  <val:P>PoBmL7mRw/GI...3JSQygyzhDpRlY=</val:P>
                  <val:G>dNCglMprbcYT...GjxOCrLNczvAm=</val:G>
                  <val:R1>abcFCD...klkDDlfSGREdsfaDkF=</val:R1>
               </val:ExValue>
            </wst:Challenge>
         </wst:SignChallenge>
     </wst:RequestSecurityTokenResponse>
  </S:Body>
</S:Envelope>
```

Next, Alice returns *X*, *r1*, and *r2*, signing them with her PKI private key (see Listing 9.20). These values are included within the `ExValue` element. The `Signature` element is also added in the header to sign on the previous value. The signature targets the SOAP body element indicated by `Id="BODY"`.

Listing 9.20 **Message 3**

```
<S:Envelope xmlns:S=http://schemas.xmlsoap.org/soap/envelope/
xmlns:wsu="http://schemas.xmlsoap.org/ws/2003/06/utility">
   <S:Header>
      <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/06/secext"
      soapenv:mustUnderstand="1">
         <wsse:BinarySecurityToken
         wsu:Id=
         "wssecurity_binary_security_token_id_2148556043341261473_1054010042109">
         A1UEBxMKWWFtYXRvLXNoaTEMMAoGA1UEChMDSUJi9NI0I=.....
         </wsse:BinarySecurityToken>
         <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
            <Sig:SignedInfo xmlns:Sig="http://www.w3.org/2000/09/xmldsig#">
               <Sig:CanonicalizationMethod Algorithm=
               "http://www.w3.org/2001/10/xml-exc-c14n#">
               </Sig:CanonicalizationMethod>
               <Sig:SignatureMethod Algorithm=
               "http://www.w3.org/2000/09/xmldsig#rsa-sha1"></Sig:SignatureMethod>
               <Sig:Reference URI="#BODY">
                  <Sig:Transforms>
                     <Sig:Transform Algorithm=
                     "http://www.w3.org/2001/10/xml-exc-c14n#"></Sig:Transform>
                  </Sig:Transforms>
                  <Sig:DigestMethod Algorithm=
                  "http://www.w3.org/2000/09/xmldsig#sha1"></Sig:DigestMethod>
                  <Sig:DigestValue>FZEkQjaigph/bqYNlGJYpMQ6/ds=</Sig:DigestValue>
               </Sig:Reference>
            </Sig:SignedInfo>
            <ds:SignatureValue>BvwwClOX5FdF/E7tE5iSBo9htu12521ktEMkBw=.........
            </ds:SignatureValue>
            <ds:KeyInfo>
               <wsse:SecurityTokenReference>
                  <wsse:Reference URI="#wssecurity_binary_security_token_id_21485
➥56043341261473_1054010042109"></wsse:Reference>
               </wsse:SecurityTokenReference>
            </ds:KeyInfo>
         </ds:Signature>
      </wsse:Security>
   </S:Header>
   <S:Body wsu:Id="BODY">
      <wst:RequestSecurityTokenResponse xmlns:wst=
      "http://schemas.xmlsoap.org/ws/2002/12/secext">
         <wst:SignChallengeResponse>
            <val:ExValue xmlns:val="http://dumml.org/exvalue">
               <val:X>alksfdcYT...dzv112rism=</val:X>
               <val:R1>abcFCD...klkDDlfSGREdsfaDkF=</val:R1>
```

Listing 9.20 **Continued**

```
               <val:R2>dNCglMprbcYT...GjxOCrLNczvAm=</val:R2>
            </val:ExValue>
         </wst:SignChallengeResponse>
      </wst:RequestSecurityTokenResponse>
   </S:Body>
</soapenv:Envelope>
```

Finally, Bob sends *Y*, *r1*, and *r2* to Alice, signing them (see Listing 9.21). The message also includes `SecurityContextToken`, which contains the ID of the shared secret ($K=X^Y \bmod p$ and $K=Y^X \bmod p$). From now on, the secret is shared and can be retrieved through the ID.

Listing 9.21 **Message 4**

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:wsu="http://schemas.xmlsoap.org/ws/2003/06/utility">
   <soapenv:Header>
      <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/06/secext"
      soapenv:mustUnderstand="1">
         <wsse:BinarySecurityToken ....>
   1tYUdGOvZUtV93k9oarQ/wDy6ac0gc0z+ixDGx1VRbhN........
         </wsse:BinarySecurityToken>
         <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
            <Sig:SignedInfo xmlns:Sig="http://www.w3.org/2000/09/xmldsig#">
               <Sig:CanonicalizationMethod Algorithm=
               "http://www.w3.org/2001/10/xml-exc-c14n#">
               </Sig:CanonicalizationMethod>
               <Sig:SignatureMethod Algorithm=
               "http://www.w3.org/2000/09/xmldsig#rsa-sha1"></Sig:SignatureMethod>
               <Sig:Reference URI="#BODY">
                  <Sig:Transforms>
                     <Sig:Transform Algorithm=
                     "http://www.w3.org/2001/10/xml-exc-c14n#"></Sig:Transform>
                  </Sig:Transforms>
                  <Sig:DigestMethod Algorithm=
                  "http://www.w3.org/2000/09/xmldsig#sha1"></Sig:DigestMethod>
                  <Sig:DigestValue>FZEkQjaigph/bqYNlGJYpMQ6/ds=</Sig:DigestValue>
               </Sig:Reference>
            </Sig:SignedInfo>
            <ds:SignatureValue>7tE5iSBo9htu12521ktEMkBw=.........
            </ds:SignatureValue>
            <ds:KeyInfo>
               <wsse:SecurityTokenReference>
                  <wsse:Reference URI="#wssecurity_binary_security_token_id_21485
➥56043341261473_1054010042109"></wsse:Reference>
```

Listing 9.21 **Continued**

```
              </wsse:SecurityTokenReference>
           </ds:KeyInfo>
        </ds:Signature>
     </wsse:Security>
  </soapenv:Header>
  <S:Body xmlns:S="http://schemas.xmlsoap.org/soap/envelope/" wsu:Id="BODY">
     <wst:RequestSecurityTokenResponse xmlns:wst=
     "http://schemas.xmlsoap.org/ws/2002/12/secext">
        <wst:RequestedSecurityToken>
           <wsse:SecurityContextToken xmlns:wsse=
           "http://schemas.xmlsoap.org/ws/2003/06/secext">
              <wsu:Identifier>uuid...</wsu:Identifier>
           </wsse:SecurityContextToken>
           <val:ExValue xmlns:val="http://dumml.org/exvalue">
              <val:Y>alksfdcYT...dzv112rism=</val:Y>
              <val:R1>abcFCD...klkDDlfSGREdsfaDkF=</val:R1>
              <val:R2>dNCglMprbcYT...GjxOCrLNczvAm=</val:R2>
           </val:ExValue>      </wst:RequestedSecurityToken>
        </wst:RequestedSecurityToken>
     </wst:RequestSecurityTokenResponse>
  </S:Body>
</soapenv:Envelope>
```

The protocol shown here is based on the Diffie-Hellman protocol. Although we defined a proprietary element `ExValue` for our explanation, you can use the `DHKeyValue` element defined in the XML Encryption specification in a real case.

If you look at WS-Trust and WS-SecureConversation, concrete negotiation protocols aren't included. Our example should give you a better idea of how negotiation protocols can be represented. We expect that profiles will emerge that define concrete protocols like the one we've presented here.

WS-Federation

One of the important challenges in Web services security is to federate different security domains—that is, a party in a security domain accesses another party in another domain. This process is very useful to provide users with SSO over multiple security domains. Extending the STS concept, WS-Federation defines how STS brokers security tokens such as identities, authentication, and security attributes.

Let's change our scenario a little as in Figure 9.18, where Buyer and SkatesTown belong to different security domains. In this scenario, Buyer has an ID managed by STS-A; Buyer gets a temporary ID from STS-B and accesses SkatesTown showing the temporary ID.

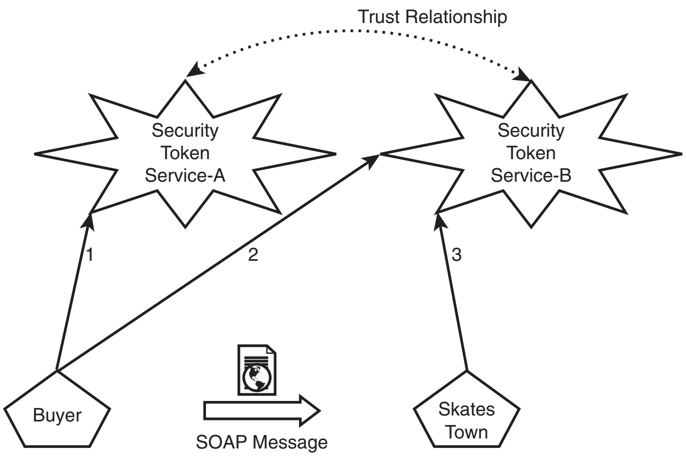

Figure 9.18 Getting a temporary ticket with federation

In message 2, a WS-Trust `RequestSecurityToken` is sent. Listing 9.22 shows a sample of the request. A Kerberos5 services ticket is required, containing an X.509 certificate issued by STS-A. The way STS-B validates the received certificate is defined by a policy or an out-of-band mechanism (Trust Relationship in the figure). After the validation, STS-B returns a temporary Kerberos ticket. Then Buyer can access SkatesTown services, including the ticket in its request messages. The request messages are no different from ones that contain Kerberos tickets issued from the same STS.

Listing 9.22 **Requesting a Kerberos Ticket to Another STS**

```
<S:Envelope xmlns:S="..." xmlns=".../secext" xmlns:wsu=".../utility>
   <S:Header>
      <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/06/secext"
      SOAP-ENV:mustUnderstand="1">
         <wsse:BinarySecurityToken xmlns:wsu=
         "http://schemas.xmlsoap.org/ws/2003/06//utility"
         EncodingType="wsse:Base64Binary" ValueType="wsse:X509v3"
         wsu:Id="myToken">
        MIIDQTCCAqqgAwIBAgICAQQwDQYJKoZIhvcNAQEFBQAwTjELMAkGA1UEBhMCSlAxETAP
        BgNVBAgTCEthbmFnYXdhMQwwCgYDVQQKEwNJQk0xDDAKBgNVBAsTA1RSTDEQMA4GA1UE
        ........
         </wsse:BinarySecurityToken>
         <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
            <SignedInfo>
               <CanonicalizationMethod Algorithm=
               "http://www.w3.org/2001/10/xml-exc-c14n#"></CanonicalizationMethod>
               <SignatureMethod Algorithm=
               "http://www.w3.org/2000/09/xmldsig#rsa-sha1"></SignatureMethod>
```

Listing 9.22 **Continued**

```
               <Reference URI="#body_id">
                  <Transforms>
                     <Transform Algorithm=
                     "http://www.w3.org/2001/10/xml-exc-c14n#"></Transform>
                  </Transforms>
                  <DigestMethod Algorithm=
                  "http://www.w3.org/2000/09/xmldsig#sha1"></DigestMethod>
                  <DigestValue>U2BIJSk6OL0W0mGXXiGVn5XPV54=</DigestValue>
               </Reference>
            </SignedInfo>
            <SignatureValue>
          Ojjw8nkT3jJoNN/AxsdOwTqWnhfZewubBWp0Sa0vJTTjQBrnKR18brODc8byuwVf2v
          iFdvMY4mT7Iumk/ZRLRNF1tEBCFRki2++W2LIXBIXVtmwo1riS98kmFZo6dBvhFOnX
          wKE1ag6C8x/UgAMVU+YzYd11KqNXtpwvi9Ydoq4=
            </SignatureValue>
            <KeyInfo>
               <wsse:SecurityTokenReference>
                  <wsse:Reference URI="#myToken"></wsse:Reference>
               </wsse:SecurityTokenReference>
            </KeyInfo>
         </Signature>
      </Security>
   </S:Header>
   <S:Body wsu:Id="req">
      <RequestSecurityToken>
         <TokenType>wsse:Kerberos5ST</TokenType>
         <RequestType>wsse:ReqIssue</RequestType>
         <Base>
            <Reference URI="#myToken"/>
         </Base>
        <wsp:AppliesTo
             xmlns:wsp='http://schemas.xmlsoap.org/ws/2002/12/policy' >
          <wsa:EndpointReference>
             <wsa:Address>http://skatestown.com/service</wsa:Address>
          </wsa:EndpointReference>
        </wsp:AppliesTo>
       </RequestSecurityToken>
    </S:Body>
</S:Envelope>
```

In addition to the federated identity model, WS-Federation defines policy assertions, extensions of WSDL, and a UDDI profile. Let's review the policy assertions. In our example, Buyer has to know that STS-B can issue a temporary ID. Listing 9.23 shows a policy assertion for our example. `RelatedServices` specifies services that should be related to a policy subject. Because the policy is attached to a SkatesTown service, Buyer can send the `RequestSecurityToken` message in Listing 9.22 to an STS-B port specified in the `Address` element.

Listing 9.23 **Policy Assertion Sample for Federation**

```
<wsp:Policy>
   <wsse:RelatedService wsse:ServiceType="wsse:ServiceSTS">
      <wsa:EndpointReference>
         <wsa:Address>http://www.sts_b.com/tempIdentity</wsa:Address>
      </wsa:EndpointReference>
   </wsse:RelatedService>
</wsp:Policy>
```

Although the related service concept is generic, WS-Federation defines four service types: ServiceIP (identify provider), ServiceSTS (STS), ServiceAS (attribute service), and ServicePS (pseudonym service). In our example, Buyer requires a `BinarySecurityToken` that contains a Kerberos ticket; the ServiceSTS for it is specified in Listing 9.23.

Enterprise Security

SkatesTown's CTO, Dean Caroll, now has a good understanding of the Web services security model and has some ideas for each specification. WS-Security provides a message format to include signature, encryption and security tokens. With WS-Trust, WS-SecureConversation, and WS-Federation, you can obtain security tokens from Security Token Services (STS) in various ways.

However, it's still difficult for Dean to envision how to combine the specifications in real situations. He especially wants to see how he can extend the SkatesTown computing environment with Web services security. Addressing Dean's requirements, Al Rosen of Silver Bullet Consulting discusses the overall security architecture of SkatesTown's system.

In this section, we'll review the J2EE security model, addressing its authentication and authorization mechanisms. Then, we'll discuss how the Web services security model can be incorporated with J2EE, showing a hypothetical overall architecture.

J2EE Security

Figure 9.19 depicts the J2EE security architecture, which is based on role-based access control (RBAC). The HTTP server or Web container authenticates a requestor, referring to a user registry (an operating system [OS] user registry or a Lightweight Directory Access Protocol [LDAP] server). Within the Web container, access to Web resources is authorized based on a URL permission list. Within the EJB container, access to EJB objects is authorized based on a method permission list. Permission lists are mappings between roles and target objects: Web resources and EJB objects.

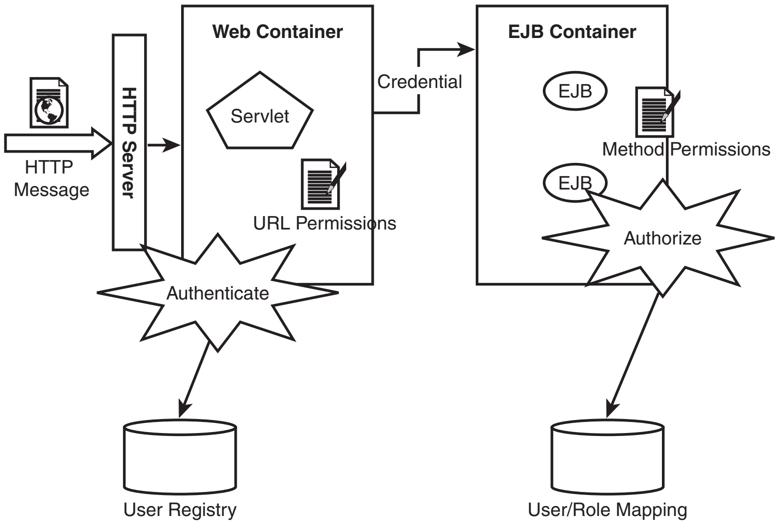

Figure 9.19 J2EE security architecture

RBAC is flexible because role assignment rules and permission lists can be independently defined. There is a trick here: A credential containing a user ID travels along the method invocation path, which can span multiple containers, and roles are assigned to the requestor at each container for authorization. This concept is especially useful when the system configuration is extremely complex.

J2EE requires compliant platforms to support the following three authentication methods:

- HTTP Basic Authentication
- SSL Client Authentication
- Form-Based Authentication

In our discussion, we'll focus on HTTP Basic Authentication (BASIC-AUTH).

Authorization in J2EE

Using BASIC-AUTH, let's review how user IDs and roles are defined in J2EE. Listing 9.24 is an excerpt from `application.xml`, which is a deployment descriptor for the overall J2EE application.

Listing 9.24 **J2EE Deployment Descriptor**

```
<application>
   ...
   <security-role>
      <role-name>GoodCustomer</role-name>
```

Listing 9.24 **Continued**

```
   </security-role>
</application>
```

The `GoodCustomer` role is used in the J2EE application. J2EE doesn't prescribe any particular means for user definition and user-role mapping. Listing 9.25 is a platform-dependent format extracted from Sun's J2EE Reference Implementation.

Listing 9.25 **A Sample of User-Role Mapping**

```
<j2ee-ri-specific-information>
   <server-name></server-name>
   <rolemapping>
      <role name="GoodCustomer">
         <principals>
            <principal>
               <name>ABCRetailer</name>
            </principal>
         </principals>
      </role>
   </rolemapping>
   ......
</j2ee-ri-specific-information>
```

With this format, you can enumerate user IDs within the `role` element. A principal indicates a user or a user group. In this example, user `ABCRetailer` can have a role `GoodCustomer`.

Let's look at a method permission definition for EJB objects. Listing 9.26 is an excerpt from `ejb.xml`, which is a deployment descriptor for EJBs.

Listing 9.26 **A Sample of Method Permission**

```
<ejb-jar>
   <display-name>OrderEjb</display-name>
   <enterprise-beans>
   </enterprise-beans>
   <assembly-descriptor>
      <security-role>
         <role-name>GoodCustomer</role-name>
      </security-role>
      <method-permission>
         <role-name>GoodCustomer</role-name>
         <method>
            <ejb-name>POProcess</ejb-name>
            <method-intf>Remote</method-intf>
            <method-name>order</method-name>
```

Listing 9.26 **Continued**

```
            <method-params>
               <method-param>java.lang.String</method-param>
               <method-param>java.lang.String</method-param>
               <method-param>java.lang.String</method-param>
               <method-param>int</method-param>
            </method-params>
         </method>
      </method-permission>
   </assembly-descriptor>
</ejb-jar>
```

The `security-role` element includes a collection of roles that are referenced somewhere in this file. `method-permission` indicates who can access the target method with `role`. `role-name` specifies the role name for this method permission—in this case, `GoodCustomer`.

J2EE and Web Services Security

How can you extend J2EE servers in order to support Web services security specifications? Figure 9.20 illustrates a possible extension of the J2EE architecture. Incoming SOAP messages with WS-Security are processed, potentially invoking the user registry and key/certificate registry. The registries access resources on the intranet, such as an LDAP server, and optionally invoke resources on the Internet, such as STSs and a CA. Within the extended J2EE, the security policy must be checked against incoming WS-Security messages. A security context may be established when WS-SecureConversation is used. We'll discuss in detail how each specification is processed in this architecture.

Let's look at WS-Security first. When a SOAP message uses WS-Security, you have to process digital signatures, encryption, and security tokens. When the signature is verified, a certificate (or key) should be required; therefore the key/certificate registry is invoked. In the simplest case, certificates are stored in files (such as keystore files). Certificates can also be managed by LDAP servers typically located on the intranet or external servers such STSs and CAs. For decryption, you need a private (or shared) key; therefore the key should be stored in an internal resource such as a file or an LDAP server.

The specification doesn't define a processing rule for a security token in WS-Security; only its format is defined. Security tokens are typically used for proof of identity. For example, when `UsernameToken` is used with a password, you may want to authenticate the ID/password. The authentication is equivalent to that in BASIC-AUTH. Therefore, you can reuse the module for BASIC-AUTH for this purpose. If `UsernameToken` contains only an identity represented in the Distinguished Name (DN), the LDAP server is accessed. Note that this solution is applicable only when you can trust the requestor node—that is, the requestor node is a gateway server of the business. An X.509 certificate can also be used as a proof of identity. In that case, *proof of position* should be ensured by means of a digital signature with the certificate.

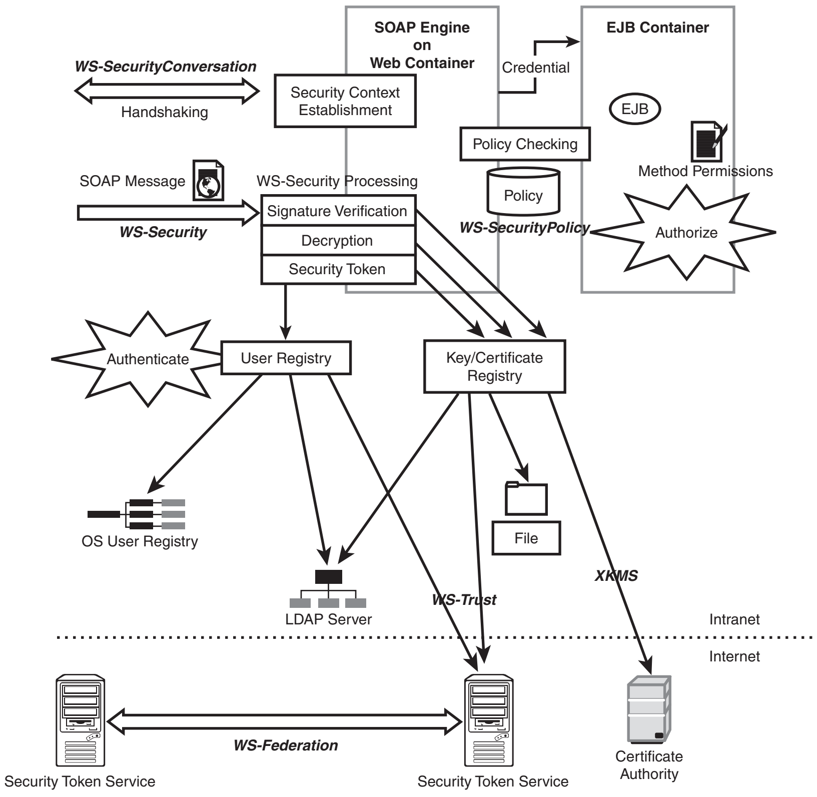

Figure 9.20 Possible integration of Web services security with J2EE

Once the requestor ID is authenticated, a credential is generated to travel along the downstream method invocation. This indicates that authorization in the downstream is exactly the same as for HTTP BASIC-AUTH. In this way, authentication with security tokens can be integrated with J2EE servers without affecting J2EE's core implementation.

The security policy described with WS-SecurityPolicy must be verified during WS-Security processing. You may have a policy that requires a signature on the message body. The signature verification only verifies whether the signature is valid; it doesn't check to see if the required part is signed. The policy-checking module in the figure needs to check such required parts, processing XPath or using the message part selection function in WS-PolicyAssertion.

In some cases, security handshaking is required to establish a security context. In that case, WS-SecureConversation performs the negotiation. Once a security context is established, a security context token is created, and its shared secret is cached for future use. When a SOAP message with the security context token is received, WS-Security processing is invoked. To process the security context token, Key/Certificate registry is invoked, and then the cached shared secret is retrieved.

There are several implementations for the key/certificate registry. When you need to retrieve an external key or certificate, you may want to use WS-Trust and XKMS. XKMS is specialized for PKI; therefore the specification is concrete. CAs such as VeriSign provide key management services with XKMS.

WS-Trust is more generic in the sense that it can be used beyond PKI. On the other hand, its specification is abstract; therefore you need further specifications (profiles). Once a collection of profiles is provided, WS-Trust becomes very useful, because you can get various security tokens with a single API.

WS-Federation defines how to integrate multiple STSs. As we reviewed in the section on WS-Federation, a temporary identity can be issued to access Web services in another security domain, for example. Thus you can achieve the ultimate goal: secure communication over multiple security domains.

Here we have envisioned how a J2EE server would be extended to support Web services security specifications. This kind of architecture discussion should help you understand Web services security more precisely, because you can imagine concrete behaviors within the J2EE architecture. Furthermore, integrating all of the specifications into a single architecture gives you an opportunity to consider the relationships among the specifications.

Security Services

XKMS suggests that key management can be outsourced to an independent third party; it could even be a Web service. In the future, a more comprehensive collection of security services may emerge. Examples include a key management service, an identity service, a signature service, an encryption service, a timer service, a rating service, and so on. Here we'll discuss a notary service as an example.

With BASIC-AUTH/SSL and/or WS-Security, you can generally fulfill the four basic security requirements: confidentiality, authentication, integrity, and nonrepudiation. However, there is a further requirement: *nonrepudiation of message receipt*. At the client side, for example, if SkatesTown receives a purchase order document from Skateboard Warehouse, it should return an invoice document. Then, it begins processing the order—that is, shipping products in the order. However, Skateboard Warehouse might not receive the invoice because of network trouble, or it might claim that the invoice wasn't delivered. Or, if the invoice delivery takes more than 10 days, Skateboard Warehouse can, by policy, consider the order unplaced or misplaced.

We want to ensure that the message has been received at the destination, and to determine when it was received. In the context of SOAP, it might be a good idea to include a notary service as a SOAP intermediary between the trading parties, as shown in Figure 9.21. The notary service provider is trusted by both SkatesTown and its customers. When there is a disagreement over message delivery between two parties, the notary service can arbitrate the problem on the basis of its log database.

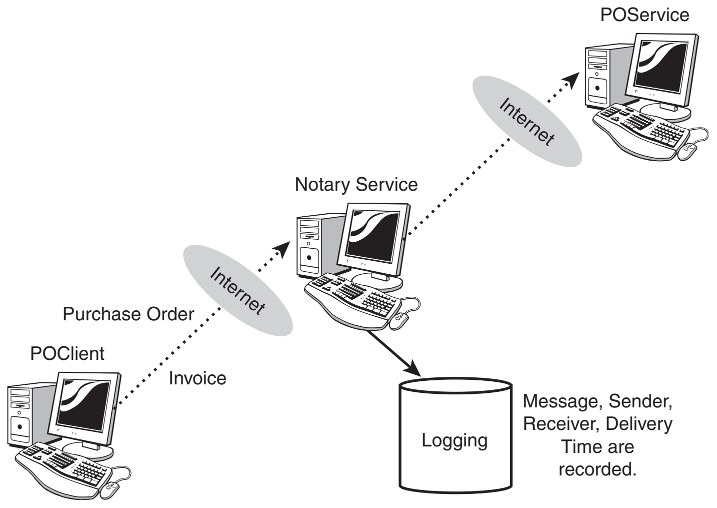

Figure 9.21 Notary service as a SOAP intermediary

From a business perspective, this structure is beneficial for all three parties. Trading parties can perform business transactions safely by contracting with the notary service. The notary service can earn money according to the transaction volume. The only problem is whether the notary service is a *real* trusted party. A number of notary services have emerged already, such as VeriSign. Most of them aim solely at trusted storage of data such as medical records. However, once they're trusted widely, they could play the role of a notary in a business structure like the one in our example.

The notary service is just one example of security services. Recently, componentization of business processes has become a trend because businesses must improve their efficiency and sharpen their competency. Considering this trend, it's possible that security services will be outsourced. After STS and XKMS become popular, we will probably see more interesting security services like our example notary service.

Three Steps to Ensure Security

You can follow a concrete process to protect your computing environment. This process consists of the following three steps:

1. *Evaluate risk* Identify resources that you need to protect, and evaluate the value of each resource. Then, predict how much money you would lose if the resource was attacked.
2. *Decide on a policy* Decide on an appropriate policy, referring to the risk evaluation as your context. If the amount of money that could potentially be lost is low and protection from the risk requires a huge budget, you may choose to accept the risk.
3. *Choose a protection method* Once you decide to protect a resource, you need to specify a protection method. The protection is a process that may include system administrators, system developers, and application users. Technologies should be integrated in the process.

As you can see, technologies like Web services security are just a piece of the complete protection method. Before introducing technologies, you must consider the balance between your investment and your predicted results.

Summary

In this chapter, we've reviewed the Web services security model and looked more closely at its specifications. We've also discussed how Web services security specifications can be implemented, extending the J2EE architecture. As a future trend, we have reviewed a notary service as an example of security services.

Web services security addresses the federation of different security domains, which will be required for application integration both in intranets and on the Internet. In order to satisfy this requirement, an abstract security model is defined; as a result, we can use and integrate existing security infrastructures through the abstract model.

Let's briefly review the specifications. WS-Security is the basis in that it defines how to include security tokens, signatures, and encryption. WS-Trust defines a means to access Security Token Services (STSs), and thus defines how to obtain security tokens. WS-SecureConversation and WS-Federation define other ways to obtain security tokens. The former addresses the security context and key derivation from the security context. The latter defines federation, assuming that there are multiple STSs. In contrast to the other specifications, WS-SecurityPolicy defines security requirements that are necessary to access particular Web services.

Resources

- *AES*—NIST FIPS 197, "Advanced Encryption Standard (AES)" (NIST, November 2001), `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`
- *ASN.1*—ITU-T Recommendation X.680, "Information Technology—Abstract Syntax Notation One (ASN.1): Specification of Basic Notation" (ITU-T, July 2002), `http://www.itu.int/rec/recommendation.asp?type=folders&lang=e&parent=T-REC-X.680`
- *Base64*—RFC 2045, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies" (IETF, November 1996), `http://www.ietf.org/rfc/rfc2045.txt`
- *BASIC-AUTH*—RFC 2617, "HTTP Authentication: Basic and Digest Access Authentication" (IETF, June 1999), `http://www.ietf.org/rfc/rfc2617.txt`
- *C14N*—"Canonical XML Version 1.0" (W3C, March 2001), `http://www.w3.org/TR/2001/REC-xml-c14n-20010315`

- *DES/3DES*—NIST FIPS 46-3, "Data Encryption Standard (DES)" (NIST, October 1999), `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`
- *Diffie-Hellman*—RFC 2631, "Diffie-Hellman Key Agreement Method" (IETF, 1999), `http://www.ietf.org/rfc/rfc2631.txt`
- *EXC-C14N*—"Exclusive XML Canonicalization, Version 1.0" (W3C, July 2002), `http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/`
- *HMAC*—RFC 2104, "HMAC: Keyed-Hashing for Message Authentication" (IETF, Feb 1997), `http://www.ietf.org/rfc/rfc2104.txt`
- *J2EE*—"Java 2 Platform, Enterprise Edition (J2EE)" (Sun Microsystems, November 2003), `http://java.sun.com/j2ee/`
- *JAAS*—"Java Authentication and Authorization Service (JAAS)" (Sun Microsystems), `http://java.sun.com/products/jaas/`
- *JCE*—"Java Cryptography Extension (JCE)" (Sun Microsystems), `http://java.sun.com/products/jce/`
- *JSSE*—"Java Secure Socket Extension (JSSE)" (Sun Microsystems), `http://java.sun.com/products/jsse/`
- *Kerberos*—RFC 1510, "The Kerberos Network Authentication Service (V5)" (IETF, September 1993), `http://www.ietf.org/rfc/rfc1510.txt`
- *LDAP*—RFC 3377, "Lightweight Directory Access Protocol (v3): Technical Specification" (IETF, September 2002), `ftp://ftp.rfc-editor.org/in-notes/rfc3377.txt`
- *OASIS* (Organization for the Advancement of Structured Information Standards)—`http://www.oasis-open.org/`
- *PGP*—RFC 1991, "PGP Message Exchange Formats" (IETF, August 1996), `http://www.ietf.org/rfc/rfc1991.txt`
- *PKCS*—"Public-Key Cryptography Standards" (RSA Laboratories), `http://www.rsasecurity.com/rsalabs/pkcs/index.html`
- *PKI/X.509*—RFC 2510, "Internet X.509 Public Key Infrastructure Certificate Management Protocols" (IETF, March 1999), `http://www.ietf.org/rfc/rfc2510.txt`
- *RFC 2437*—"PKCS #1: RSA Cryptography Specifications Version 2.0" (IETF, October 1998), `http://www.ietf.org/rfc/rfc2437.txt`
- *Roadmap document*—"Security in a Web Services World: A Proposed Architecture and Roadmap" (IBM and Microsoft, April 2002), `http://www-106.ibm.com/developerworks/webservices/library/ws-secmap/`

- *SAML*—"Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V1.1" (OASIS, September 2003), `http://www.oasis-open.org/committees/download.php/3406/oasis-sstc-saml-core-1.1.pdf`
- *SHA-1*—FIPS 180-1, "Secure Hash Standard" (U.S. Dept. of Commerce/National Institute of Standards and Technology, April 1995), `http://csrc.nist.gov/publications/fips/fips180-1/fip180-1.txt`
- *SSL*—"The SSL Protocol Version 3.0" (Netscape Communications, November 1996), `http://home.netscape.com/eng/ssl3/draft302.txt`
- *TLS*—RFC 2246, "The TLS Protocol Version 1.0" (IETF, January 1999), `http://www.ietf.org/rfc/rfc2246.txt`
- *VeriSign Inc.*—`http://www.verisign.com/`
- *WS-Federation*—"Web Services Federation Language (WS-Federation)" (IBM and Microsoft, July 2003), `http://www-106.ibm.com/developerworks/library/ws-fed/`
- *WS-SecureConversation*—"Web Services Secure Conversation (WS-SecureConversation)" (IBM and Microsoft, December 2002), `http://www-106.ibm.com/developerworks/library/ws-secon/`
- *WS-Security*—"Web Services Security: SOAP Message Security " (OASIS, August 2003), `http://www.oasis-open.org/committees/download.php/3281/WSS-SOAPMessageSecurity-17-082703-merged.pdf`
- *WS-Security Kerberos Binding*—"Web Services Security Kerberos Binding" (IBM and Microsoft, December 2003), `http://msdn.microsoft.com/webservices/understanding/specs/default.aspx?pull=/library/en-us/dnglobspec/html/ws-security-kerberos.asp`
- *WS-Security SAML Profile*—"Web Services Security: SAML Token Profile" (OASIS, December 2003), `http://www.oasis-open.org/committees/download.php/4534/WSS-SAML-08.pdf`
- *WS-Security UsernameToken Profile*—"Web Services Security: UsernameToken Profile" (OASIS, August 2003), `http://www.oasis-open.org/committees/download.php/3154/WSS-Username-04-081103-merged.pdf`
- *WS-Security X.509 Profile*—"Web Services Security: X.509 Certificate Token Profile" (OASIS, August 2003), `http://www.oasis-open.org/committees/download.php/3214/WSS-X509%20draft%2010.pdf`
- *WS-SecurityPolicy*—"Web Services Security Policy (WS-SecurityPolicy)" (IBM and Microsoft, December 2002), `http://www-106.ibm.com/developerworks/webservices/library/ws-secpol/`
- *WS-Trust*—"Web Services Trust Language (WS-Trust)" (IBM and Microsoft, December 2002), `http://www-106.ibm.com/developerworks/webservices/library/ws-trust/`

- *X.500 Distinguished Name*—ITU-T Recommendation X.500, "Information Technology—Open Systems Interconnection—The Directory: Overview of Concepts, Models and Services" (ITU-T, February 2001), `http://www.itu.int/rec/recommendation.asp?type=folders&lang=e&parent=T-REC-X.500`
- *XKMS*—"XML Key Management Specification (XKMS)" (W3C, March 2001), `http://www.w3.org/TR/2001/NOTE-xkms-20010330/`
- *XKMS2*—"XML Key Management Specification (XKMS) Version 2.0" (W3C, April 2003), `http://www.w3.org/TR/2003/WD-xkms2-20030418/`
- *XML Encryption*—"XML Encryption Syntax and Processing" (W3C, December 2002), `http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/`
- *XML Signature*—"XML-Signature Syntax and Processing" (W3C, February 2002), `http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/`