SAMPLE CHAPTER

SOA SECURITY

Ramarao Kanneganti
Prasad Chodavarapu

MANNING

SOA Security

by Ramarao Kanneganti

Prasad Chodavarapu

Sample Chapter 6

brief contents

6 Protecting confidentiality of messages using encryption

This chapter covers

- Public key infrastructure
- JCE and Apache XML security
- Certificate authorities

In the preceding chapters, we've seen how to extend SOAP via headers. In particular, we saw how to add user credentials so that the application can determine whether the request came from a genuine user. We introduced various techniques to secure credentials so that they cannot be misused by any party listening over the wire or by the service providers themselves.

There is more to security than mere authentication. Imagine that you are requesting a brokerage firm to buy some shares using the funds you have in a bank account. The firm requires you to authenticate with username and password. Suppose you use the digest mechanism so that password is not available to the eavesdropper. Is that enough? Wouldn't you also want to safeguard the bank account information you are providing?

What this scenario points out is that we need a way to encrypt the message so that only the intended recipient can understand it. Traditionally, this task is accomplished by encrypting the whole message. As we mentioned in chapter 1, straightforward encryption of the whole message is not good enough to meet the requirements of SOA. Instead, a mechanism is needed to encrypt different parts of a message differently.

There are other advantages to being able to encrypt parts of a message. Encryption and decryption are computationally intensive operations. By encrypting only the confidential parts of a message, we can enhance the performance of a solution without compromising on security. Selective encryption also helps if parts of a message need to be kept in plain text for reasons beyond our control. For example, a critical legacy application may depend on a part of the message and we may not be allowed to modify the application in any way.

In this chapter, we will describe how the web services standards support selective encryption of messages. Encryption in web services is built on the solid foundations of encryption technology that is widely used on the web. The technology is well understood, with good implementations and support structure. To understand how encryption can be used in SOAP messages, we need to understand the basics of encryption technology first. In the first three sections of this chapter (6.1-6.3), we will cover these in detail. In particular:

- The first section will introduce a tool that helps you snoop on the network to see messages in transit. This tool will help you see the need for encryption.
- The second section will introduce you to symmetric key encryption, asymmetric or public key encryption, and hybrid encryption. You will also learn about how PKI and digital certificates help in the adoption of encryption on a large scale.

- The third section will show you how to program with digital certificates. We will show you how to create a digital certificate and how to use it for point-to-point encryption with SSL/TLS. We will introduce Java Cryptographic Extension (JCE) and other Java APIs you can use for implementing encryption with Java.

If you are familiar with encryption, particularly PKI and SSL, feel free to jump to the start of section 6.4. This section will describe the standards for encryption in web services, specifically how WS-Security supports the XML Encryption standard and how you can implement the same using the Apache XML Security library. Section 6.5 will discuss a few practical issues you are likely to face when using encryption in the real world.

We will start our discussion by showing you how you can snoop on the network to intercept messages without the consent of the sender or the receiver. We will show you how encryption protects the confidentiality of a message even when it is intercepted by a man in the middle. This example will help you see the need for encryption before we get into the nitty gritty of encryption.

6.1 Encryption in action: an example

Recall how we depicted the message exchanges in the preceding chapters using `tcpmon`. We set up a proxy, over which we sent our requests. This proxy prints the request and response so that we can understand the messages between the client and the server. We employed `tcpmon` as simply a pedagogical tool, which required active consent from the sender and receiver, in terms of configuration.

Now we are going to introduce you to a different tool called `ethereal`, which allows us to watch what is really happening over the wire. As it happens, this tool lets us examine the flow of messages *without the consent of the participants.* In this section, we will use this tool to intercept HTTP and HTTPS traffic.

Table 6.1 shows how to run this example:

Table 6.1 Steps to use `ethereal` for intercepting HTTP and HTTPS traffic. Watch steps 5 and 9 to understand what kind of traffic is really going over the wire.

Step	Action	How To
1	Install ethereal.	Download WinPcap (assuming you are on Windows) and ethereal installers from http://www.ethereal.com/. Install WinPcap followed by ethereal.

continued on next page

Table 6.1 Steps to use `ethereal` for intercepting HTTP and HTTPS traffic. Watch steps 5 and 9 to understand what kind of traffic is really going over the wire. *(continued)*

Step	Action	How To
2	Open a web page that contains a login form but does not use HTTPS.	Go to http://www.manning-sandbox.com/login!withRedirect.jspa but do not log in just yet. Leave the browser window open.
3	Set up ethereal to capture HTTP traffic.	Start `ethereal`. Go to "Capture -> Interfaces…" and "Prepare" the interface over which you are connecting to the Internet. In the resulting Capture Options dialog, enter `tcp port 80` as the capture filter. This is to instruct ethereal that all TCP packets originating from port 80 or destined to port 80 on any network node should be captured. In addition, uncheck the "Capture packets in promiscuous mode" check box, as we are only interested in snooping on packets originating from/destined to our box. Select the "Update list of packets in real time" option. Click OK to start capturing. This will bring up a dialog showing the number of packets captured. You may not find any packets captured at this time, as you have yet to do any HTTP activity.
4	Carry out HTTP activity using the login form opened in step 2.	Go back to the browser window you used to reach the login page, type in your username and password, and click the Login button
5	Stop packet capture and analyze captured traffic.	Once your login succeeds or fails, stop the capture process in ethereal. You should now see a list of packets captured as shown in figure 6.1. Right-click the first HTTP packet and choose to "Follow tcp stream." You can now see the HTTP conversation that took place between your browser and the login server. In particular, look at the first line after the end of the HTTP headers in the POST request. You will notice that your username and password are visible in clear-text.
6	Open a web page that contains a login form and uses HTTPS.	Log out of manning sandbox in case your login was successful. Now go to http://www.manning-sandbox.com/login!withRedirect.jspa but do not log in just yet. Leave the browser window open.
7	Set up ethereal to capture HTTPS traffic this time.	Use step 2's instructions but replace port 80 with port 443.
8	Carry out HTTPS activity using the login form opened in Step 7.	Go back to the browser window, type in your username and password, and login.
9	Stop packet capture and analyze captured traffic.	Once your login succeeds or fails, stop the capture process in ethereal. You should once again see a list of packets captured. Right-click the first TLS packet and choose to "Follow tcp stream." You will see that you can no longer decipher the conversation that took place between your browser and the login server.

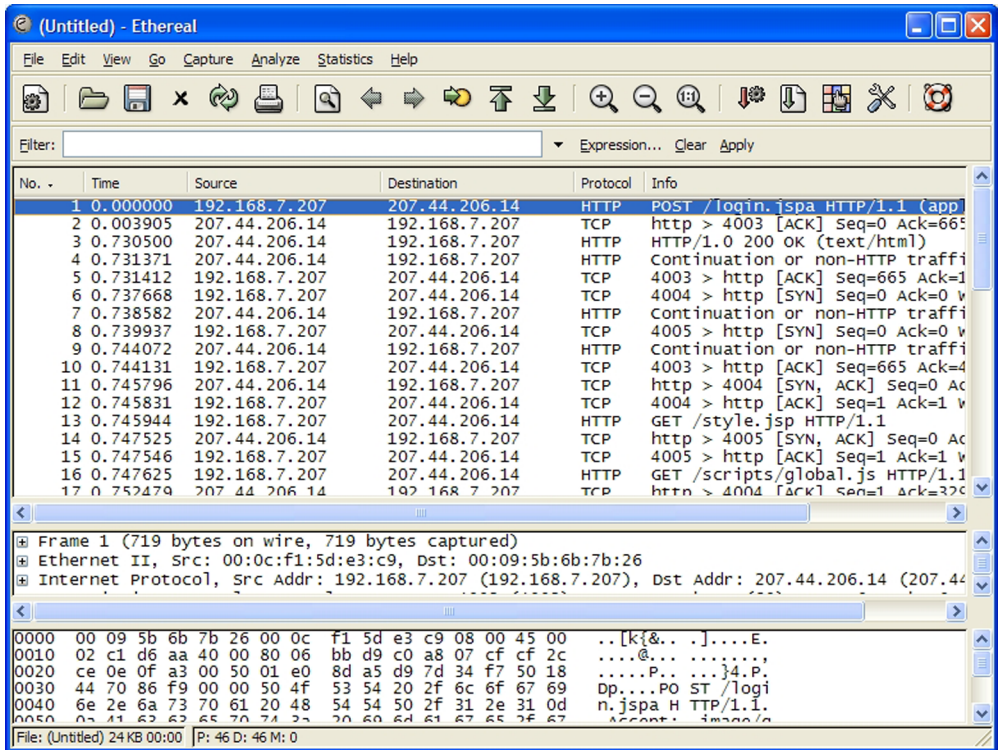

Figure 6.1 Screen shot of `ethereal` illustrating the capture of network data without the consent or cooperation of the participants.

Figure 6.1 shows a screenshot of ethereal after step 5, which shows us the packet details that are going over the wire. In fact, you can see the protocol and understand the actual contents of each packet. After step 9, the same application will show that packet contents are not in the clear; i.e., they appear as gibberish.

We can learn the following from this example:

1. When using HTTP, the message is transmitted in a clear-text protocol. We can easily see the username and password.
2. When using HTTPS, the message is transmitted in encrypted form. There is no way a man in the middle can understand any part of the message, including the username and password. Of course, we have yet to discuss the details of encryption technology behind HTTPS, so, it is premature to say if the encryption we see here is good enough.

We will now describe the technology behind encryption in order for you to better understand how you can guard the confidentiality of network data.

6.2 The basics of encryption

To understand how encryption works, consider the case where one party sends a message to another party. How can the sender make sure that only the intended receiver can read the message? If he sends the data as is, as you saw earlier, anybody who happens to be in the path can read the message. To make sure that it cannot be read, he needs to send it in a form that nobody (including perhaps even the sender) other than the receiver can understand. That is, the sender transforms the data in the message into *cipher* data, and the receiver undoes the transformation so that he can get back the original data from the cipher data.

These transformations are called *encryption* and *decryption* respectively. Mathematically speaking, these operations are carried out by two functions `E` and `D`, where `E` is the encryption function and `D` is the decryption function. To encrypt a message `m`, you apply the function `E` to `m`, thereby producing the cipher data `E(m)`. If you are the intended receiver, you would apply the function `D` to get the original message back. To state in mathematical terms: `D(E(m)) = m`.

Why is such a scheme secure? Actually, the security does not lie in the scheme—it lies in the algorithms. If it is computationally difficult to guess `m` from `E(m)` then we can say that only the party that possesses `D` can decrypt the message. There are several other features that are required of these algorithms if they have to be used repeatedly. In the rest of this section, we will examine different kinds of encryption algorithms and describe the situations in which they are appropriate. In particular, we will describe two kinds of encryption algorithms: symmetric key and asymmetric key (or public key) algorithms, which are required for you to understand PKI. We will also see why and how these two kinds of algorithms are often combined to create what is known as *hybrid encryption*.

6.2.1 Types of encryption algorithms

As we mentioned previously, the effectiveness of encryption in protecting the confidentiality of a message depends on the encryption algorithm used. Let's examine a sample encryption algorithm so that you can see what we mean.

One of the earliest known encryption algorithms is Caesar shift. It shifts each letter by certain positions to create the cipher text. For example, with a shift of 1, "a" becomes "b" and "b" becomes "c." If the sender encrypts a message using such a shift, the receiver can decrypt by applying the reverse shift to the received

cipher data. The shift amount is a required parameter, or the *key* to the encryption and decryption functions.

It is easy to see that a Caesar shift algorithm is weak. All we need to do is try all 25 shifts (assuming we only use the English alphabet in a message) to forcefully crack the encryption. If we want to refine Caesar shift to make decryption more challenging, we can apply arbitrary substitutions, where each letter may get transformed into a different letter. Decryption now requires the knowledge of a whole substitution table, not just a mere numeric key indicating the magnitude of shift. Such an algorithm is not strong either; all we need to do is to look at the patterns in the text. If we know that the word "the" occurs frequently, we can easily look for such frequent words in the cipher text too.

If you are beginning to get the feeling that devising secure encryption functions is difficult, you are right. Consider some of the popular technical attacks you have to guard against:

- *Brute force method* Under this method, the attacker tries all the possible keys. If the number of possible keys is small, this method is easy to apply. In fact, with advanced computing power these days, even large numbers of keys, up to billions, can be tried in hours. Thus, we need to make the possible key universe as large as possible.
- *Frequency analysis* In this method, the attacker takes a look at several cipher texts and attempts to break the cipher by comparing the frequency distribution of letters/words/phrases in the cipher text with the frequency distribution he may know of letters/words/phrases in the domain. Note that not all algorithms are susceptible to this attack. For example, if the encryption scrambles the message randomly then frequency analysis does not work.
- *Known text attack* Some messages always contain the same information in known places. For example, every legal letter may end with a standard disclaimer. Using such knowledge, it may be possible to understand the algorithm. That is, the attacker knows the `E(m,k)` for some specific messages and may try to deduce `k` from it.
- *Chosen plain text attack* Under this model, the attacker tricks the sender into encrypting known messages. That is, he gets `E(m,k)` for any message he desires. Depending on the algorithm, it may be possible to deduce `k` from the pairs `(m, E(m,k))`.

Unfortunately, given a function, it is difficult to prove that it is cryptographically strong. It takes a lot of theoretical analysis, peer review, and public usage before

we can develop confidence in the cryptographic strength of any algorithm. The actual development of algorithms should be left to professionals. Fortunately, there are several standard algorithms both in public and private domains for us to choose from. In fact, several of these algorithms have been coded, tested, and incorporated into standards for us to use.

Now, we will look at some standard algorithms. We will first describe a class of algorithms known as *symmetric-key algorithms*.

Symmetric-key encryption algorithms

Caesar shift is a simple but good example of a class of encryption algorithms known as symmetric-key algorithms. A symmetric key algorithm is one in which the encryption and decryption functions are parameterized by the same key.[1] To understand this definition, let's re-examine how text that is encrypted using Caesar shift is decrypted. To decrypt, we apply the reverse shift to the cipher text. If decryption is to succeed, the amount of reverse shift during decryption has to match the amount of shift carried out during encryption. If we call the amount of shift and reverse shift the *key* for encryption and decryption functions then Caesar shift is a symmetric-key algorithm because we use the same key during encryption and decryption.

To explain it mathematically, if we denote the message with `m`, the key with `k`, the encryption function with `E`, and the decryption function with `D`, the following is true for symmetric-key encryption: `D(E(m,k),k)=m`.

Symmetric-key algorithms have been widely investigated and are well understood. Some are block-oriented algorithms, which take a block of text and encrypt it (think of a transformation that reverses the text in each block); some are stream-oriented algorithms, where each byte is encoded as it comes out using a varying encryption scheme (think of the earlier shifting algorithm modified to increment the shift by 1 after every letter). Here is a list of some popular symmetric-key algorithms. Most algorithms have public-domain implementations ready to be used in our applications.

DES: This is a block-oriented algorithm that uses 56-bit keys. Given the computational power available to hackers these days, 56-bit keys are considered easy to crack; DES is no longer recommended.

Triple DES (3DES): In this scheme, DES is applied three times to the message, with three different keys, thus increasing the key length to 168 bits. Most

[1] Even if the encryption and decryption keys are different, as long as the keys can be inferred from one another, the algorithm can be considered a symmetric-key algorithm.

implementations do decryption as the second step rather than encryption. That is, the cipher text is computed as `E(D(E(m,k1),k2),k3)`.[2] 3DES is quite slow compared to other symmetric encryption algorithms, such as AES.

RC2 and RC4: This algorithm can theoretically use up to 2048 bits.

RC5: This algorithm is more recent than RC2. It also features a variable key length of up to 2040, with a block length of 32, 64, or 128 bits.

AES: This algorithm is a recent standard from NIST (National Institute of Science and Technology). It is a fast and compact algorithm, suitable for implementation in hardware or software with a key length of 128, 192, or 256 bits. Encryption standards in web services often recognize AES as an algorithm that all compliant implementations must support.

Symmetric-key algorithms suffer from one drawback. To understand what this is, let us look at an end-to-end scenario. The sender and receiver need to establish an algorithm and the key for the algorithm before they can start communicating securely. Negotiating which algorithm to use is easy; the sender can send a list of algorithms it is capable of using, in the order of preference, and the receiver can pick one out of them. Next, they need to pick a key. How do they decide on a key *securely*?

The two parties can negotiate the algorithm using messages in clear-text, but they cannot communicate a key to each other in clear-text. There is no danger if a snooper knows the algorithm—after all, only a handful of algorithms are used in practice—but if he knows the key, the secrecy of the messages is compromised.

A key can be agreed upon out-of-band. For example, one party can call up the other party on the phone and agree upon a key to use. But that would restrict the number of parties one can work with. One cannot possibly call everyone else on Earth to agree upon a key with each person. One may want to use dynamically generated keys, but now there is a bootstrapping problem. To secure a message exchange, one needs a key, but the key itself needs to be securely communicated first.

Is there a possibility of establishing keys up front in a central store? That is, for every pair of communicating parties, can we create a central key store that contains a key for encryption? The answer is no, because that would be a logistical nightmare! First of all, it means we would need around n^2 keys for n parties,

[2] There is a good reason for using decryption instead of encryption in the second pass of 3DES implementation. EDE—that is encryption, decryption, followed by encryption—provides backward compatibility with simple DES, when all three keys are chosen to be the same; that is, when k1=k2=k3.

making the key store scale poorly. Second, if we want to add a new party, we will need to set up n additional keys.

A symmetric-key algorithm is of limited use unless we find a way to securely share the key between the sender and receiver. This chicken-and-egg problem can be solved using an alternative class of encryption algorithms known as asymmetric-key or public-key encryption algorithms. Let us study them next.

Public-key encryption algorithms

If the need to share the same key between the sender and receiver is what limits the applicability of symmetric-key algorithms, can't we use different keys for encryption and decryption? Yes, we can. When the sender wants to send a message securely, he can encrypt it using `k1` and the receiver can decrypt it using `k2`. This is the basic idea behind public-key encryption. The equation that makes it all work is `D(E(m,k1),k2) = m`.

At first glace, the situation does not seem to have improved; we are simply using two keys instead of one, where each application gets two keys. The difference is that one of the keys (`k1`) is published publicly so that everyone is able to see it, and the other (`k2`) is kept private so that only the key's owner knows it. In the end, for each application, there is only one private key that needs to be guarded.

Here is how the keys are used: if an application wants to send a confidential message `m` to another application, it will send the cipher text `E(m,k1)`, which can only be decrypted by an application that knows the corresponding key `k2`, which means only the intended recipient can decrypt it.

What if the receiver wants to send a response message back to the sender? It needs to follow the same mechanism of using the other application's public key. Figure 6.2 shows how public-key encryption happens in either direction.

For public-key encryption to work, every party needs to publish its public key someplace. Unlike in the symmetric-key situation, where the number of keys needed by n parties is n^2, in public-key encryption, we only need to keep track of the same number of keys as the parties; that is n. In fact, in section 6.2.2 where we describe PKI, you can see how we can optimize the public-key distribution mechanism even further.

The most popular public-key encryption algorithm used today is RSA. It can be implemented with any key length. In later sections, we will show how to use RSA in Java. For now, we will discuss a few important properties of public-key encryption algorithms in general that are useful for PKI.

Because public-key encryption algorithms are computationally more expensive than symmetric-key algorithms, they are never used in practice to encrypt

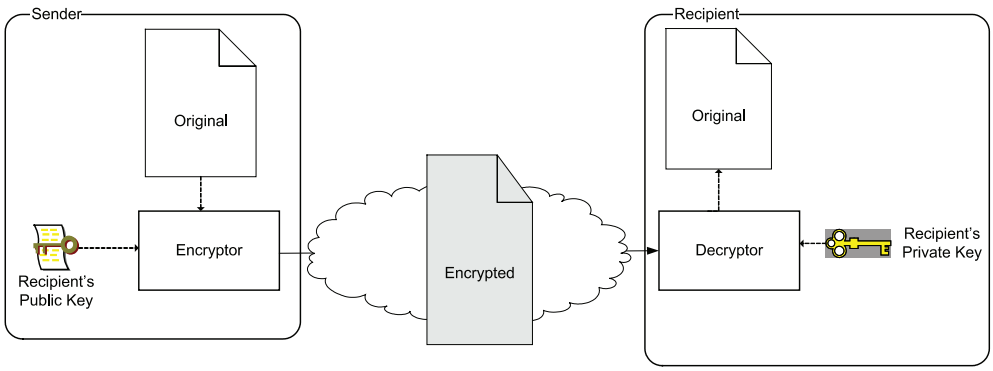

Figure 6.2 Public-key encryption in action: The original message encrypted using the recipient's public key can be decrypted using the recipient's private key.

arbitrarily large messages. Instead, they are limited to encryption of small messages. Still, public-key encryption algorithms are very useful for two purposes: digital signatures and hybrid encryption. We will first describe digital signatures and take up the topic of hybrid encryption after that.

We have previously defined public-key encryption algorithms as those that satisfy the equation `D(E(m,k1),k2) = m`. There is a second equation that public-key encryption algorithms satisfy: `D(E(m,k2),k1) = m`. Comparing the two equations, we see that encryption and decryption keys can be swapped. *A message encrypted using one of the two keys can always be decrypted using the other key.*

We used the property expressed in the first equation as follows: A party X can openly distribute its public key to anyone who wishes to communicate with it. Any messages encrypted with that public key can only be decrypted by the associated private key that is only known to X. Now, here's what the second equation means: If X encrypts a message with its private key everyone who knows its public key can decrypt it. How is this second property of public key encryption useful?

In practice, this is useful to solve a problem that is different than what we are focusing on in this chapter, but related. We have been looking only at how we can use encryption to keep sensitive parts of a message confidential, but we haven't yet thought of how we can guarantee message integrity. That is, if a man in the middle modifies a part of the message, we would not be able to detect it. We can use the second property of public-key encryption that we just presented to tackle this problem. Although preserving message integrity is the topic of the next chapter, we need to briefly discuss it here, as some of the lessons involved are necessary to understand the rest of this chapter.

How can a recipient possibly check that the received message has not been tampered with en route? The sender can help the recipient in this process by providing a value computed from the message along with the message. The receiver can recompute the same value and check whether the result matches the value provided by the sender. For example, the sender can provide a checksum,[3] the sum of all the bytes in the message. The receiver can also sum up all the bytes and check whether the result matches the sender-provided checksum. Unfortunately, checksums do not offer good enough protection against data tampering. An attacker can simply swap the bytes in the message without damaging the checksum. For example, if you asked to transfer $1001 from a bank account, an attacker might be able to change the amount to $1100 without changing the checksum. So, we need a function that makes it difficult to produce an input that produces the given output. Cryptographic hash functions such as SHA-1 (introduced in chapter 4 when we discussed password digests) fulfill this requirement.

Suppose that the sender attaches `SHA1(m)` to message `m` when communicating it to the receiver. Does this digest suffice for guaranteeing message integrity? What if the attacker replaces the whole of the message `m` with `m'` and attaches `SHA1(m')`? To rule out this possibility, we need a way of making sure that `SHA1` value was computed by none other than the sender.

If the sender encrypts `SHA1(m)` as `E(SHA1(m))` using his private key and attaches it to the message, the receiver can verify that the message has not been tampered with along the way by decrypting `E(SHA1(m))` using the sender's public key and comparing it with `SHA1(m)`, which the receiver computed independently. As we know that public-key encryption is expensive, we also have to make sure that the computed value, `SHA1(m)` in this example, is short enough no matter how long `m` is. Otherwise, the cost of computing `E(SHA1(m))` may be too high, making this technique impractical for use. Fortunately, cryptographic hash functions such as SHA-1 produce short fixed-length output no matter how long the input is. That is why they are often referred to as *message digest algorithms* and the values they compute are referred to as *message digests*.

The act of computing the message digest and encrypting the result with the sender's private key is known as *signing*. That is, the sender digitally affixes a signature to a message to prove that he is indeed the sender of the message and that

[3] A checksum is the result obtained by "adding" all the bits in a message. The algorithm used for computing checksum is irrelevant for our discussion here. Think of it as addition of bits in some manner to create a number that can be represented in a fixed number of bits.

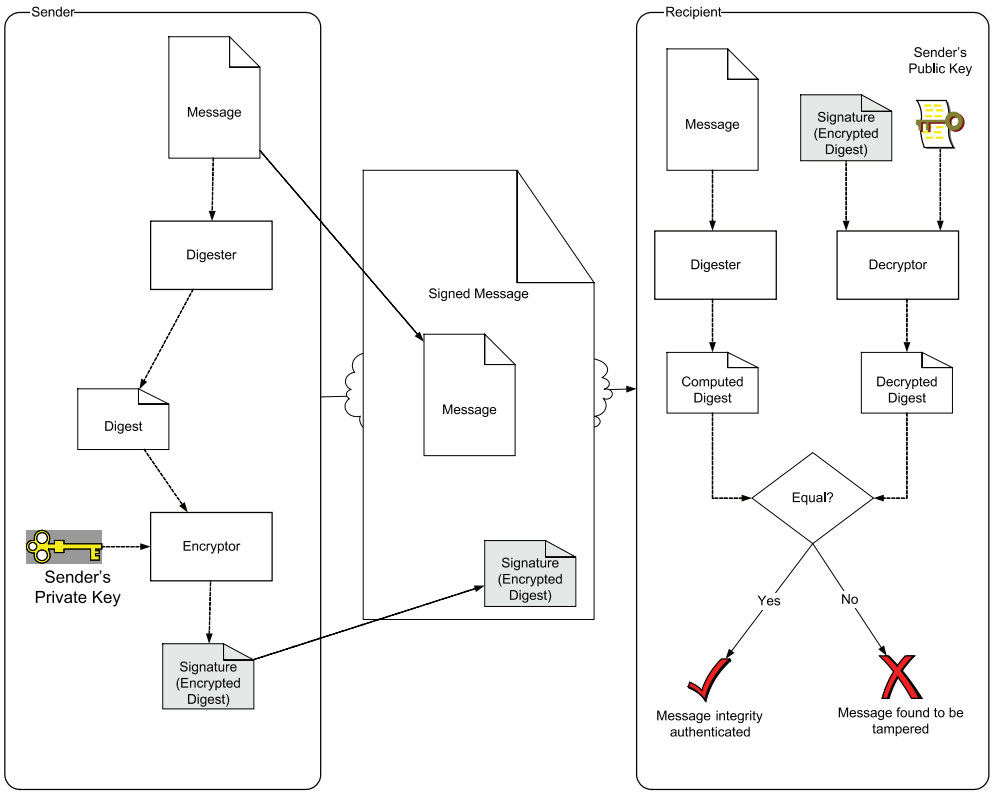

Figure 6.3 How signatures work: The sender attaches a message digest that is encrypted using his private key. The recipient uses the sender's public key to decrypt the encrypted message digest. If the result of this decryption matches the digest value independently computed by the recipient, the message is proven to be intact; otherwise, a man in the middle must have tampered with the message en route.

the message is genuine. Figure 6.3 illustrates this idea. We'll focus more on signatures in the next chapter. For now, you've learned enough to understand PKI.

Let's recap what you have learned so far about encryption algorithms. Let's say you want to receive confidential messages from your partners. You can do so using symmetric-key encryption algorithms if each of your partners can securely share with you the secret key he will use for encryption. But you have a bootstrapping problem. To secure your message exchange, you need a key, but the key itself needs to be securely communicated first. If you use public-key encryption algorithms instead of symmetric-key algorithms, you do not have this problem. You can publish your public key openly. Anyone who needs to securely communicate

with you can use your public key to encrypt his messages, and only you can decrypt those messages, as you are the only one in possession of your private key. But public-key encryption algorithms are too expensive computationally, and can only be used for small messages. In other words, neither symmetric-key algorithms nor public-key algorithms provide a perfect solution for all encryption needs.

Fortunately, there is a way to take care of these problems by combining the best of both worlds using a hybrid encryption scheme. We will describe this in the next subsection.

Hybrid encryption

Hybrid encryption combines the efficiency of symmetric-key encryption with the relative ease of setting up public-key encryption. Recall the big hurdle in using symmetric-key encryption: difficulty establishing the initial key securely. If we use public-key encryption to securely communicate a randomly chosen symmetric key at the start of a conversation, we can cross that hurdle. For all the subsequent communications, we can use that symmetric key. Figure 6.4 illustrates how this can happen.

There are two benefits to this hybrid scheme. First, since it uses symmetric-key encryption except for the initial key establishment, it is simple and fast. Second, we do not need to distribute keys to all parties. We only need to publish public keys for server applications. Since clients initiate the contact, they only need to generate a random key for symmetric encryption and communicate it using public key of the server application.

Now you know how different types of encryption algorithms work in theory. We can take for granted that all these algorithms work well in theory. From now on, we will turn to practical aspects. For example, how are public keys distributed? How do applications negotiate the algorithms to use? The answer to all these questions is PKI.

6.2.2 PKI: A framework for encryption

While introducing you to the basics of encryption, we have described various types of encryption algorithms. We discussed symmetric-key encryption algorithms, public-key encryption algorithms, and a hybrid scheme that combines the two. All through this discussion, we have repeatedly hinted that one also needs to look into an algorithm's key distribution requirements before deciding whether it is appropriate for use. PKI provides a cost-effective and proven framework that addresses the key distribution requirements in public-key encryption. Of course, hybrid encryption can also use this framework, as its key distribution requirements are the same as those of public-key encryption.

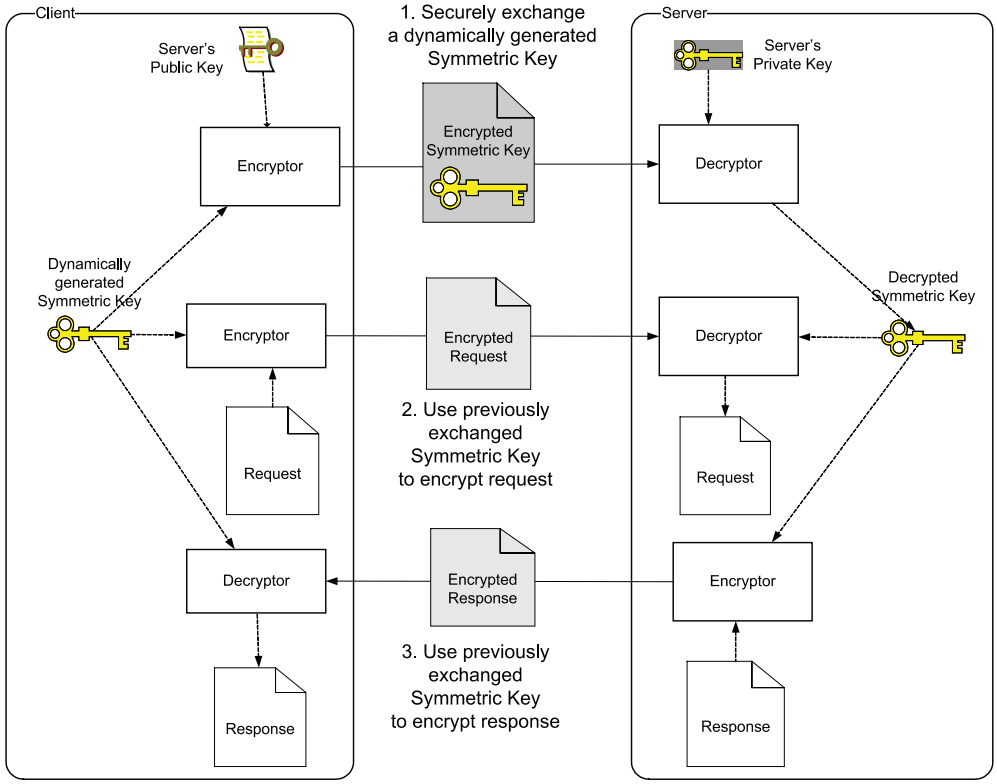

Figure 6.4 How hybrid encryption works: The key needed for symmetric-key encryption is first exchanged securely using public-key encryption. All subsequent message exchanges use symmetric-key encryption.

Consider the issue of publishing keys. One easy choice for publishing these keys would be some central stores. Say application A wants to send a message to B. A can get the public key for B from a central store S. But what guarantee is there that this is the key for B? A man in the middle can easily pose as S and give A his public key instead. Such an attack is easily detected if S adds B's "name" (we put the name in the quotes here because a name can be any general-purpose name—more details in the next section) along with its key and signs using its own private key. A can verify S's signature to make sure that the message came untampered from S. Of course, we are assuming that everyone knows the central store's public key in order to verify its signatures.

This scheme will work; however, every application needs to communicate with the central store for any public key. As such, the central store will become the bot-

tleneck in any implementation. But, there is no need to get this information from a central store. The same information, *signed by the central store S*, can be obtained from any source, and the confidence in that information does not change. Just as a photo ID with a government seal can be presented by the person to whom the ID was issued, the owner of a public key (B in our example) can itself present a copy of its public key that is signed by the central store S. In a sense, B is acting as a proxy for its own public key information.

It's time we call all these entities by their proper names. A central store trusted by the application is called a *Certificate Authority* or *CA*. The information signed by a central store—a public key, the name of the entity who owns the key, and other details such as expiry date—is called a *(digital) certificate*. In other words, a certificate not only provides the public key of the certified party, but also provides the certified party's identification information.

Now put it all together as in figure 6.5.

Let us see what the terms used in this figure mean:

- *Subject info* This information specifies who the certificate is issued to in X.500 DN format. It contains the name, organization, organizational unit, and a serial number. The serial number allows for more than one certificate in the same name. (Note: X.500 is a set of directory services standards. A directory [think of your corporate LDAP directory] is a collection of entries. Each directory entry is a collection of multivalued named attributes and is uniquely identified by what is known as a distinguished name (DN). A DN is structured in a way that allows the directory to be viewed as a tree of entries. For example, an entry with DN `CN=Prasad Chodavarapu, OU=Authors, O=Manning, L=Bangalore, ST=Karnataka, C=IN` can be seen as a node named `CN=Prasad Chodavarapu` with `OU=Authors` as its parent, `O=Manning` as its grandparent, and so on. Public-key certificates were first standardized by the X.509 specification, which belongs to the X.500 family of standards. X.509 adopted X.500 DNs when it needed a way of identifying who a certificate belongs to.)
- *CA info* Since the client needs to know whose public key can be used to verify the certificate, the name of the issuer is also kept in the certificate. Clients normally accept only the CAs that they trust. If you recall, the CA signs a certificate by encrypting a digest of the information in the certificate with its private key. The CA information should also contain the encryption algorithm and the digest algorithm used by the CA.

- *Validity dates* These dates specify the period in which the certificate is valid.
- *Public key* This is the key provided to the subject. This field also needs to provide which algorithm is to be used with this key.

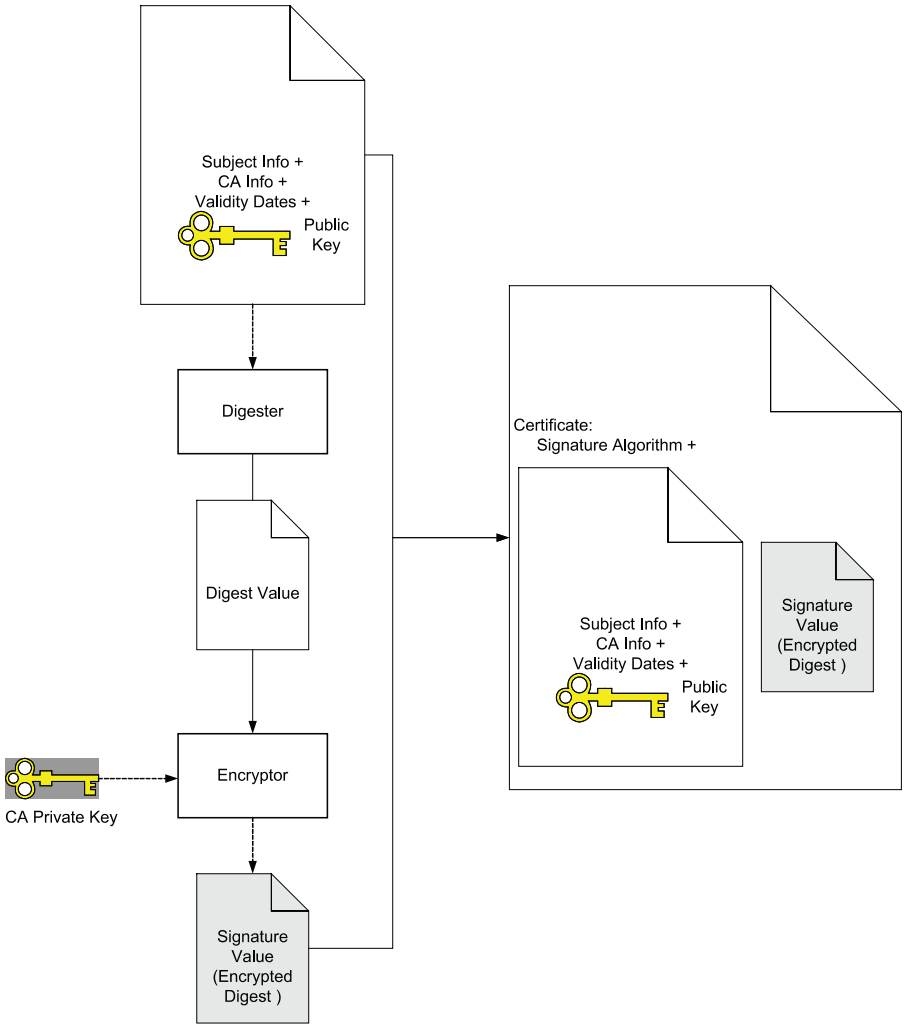

Figure 6.5 How the certificate is generated. Only a part of the certificate, the digest, needs to be encrypted (as a signature) by the CA. The certificate which contains the public key of the application is guaranteed by the CA. This signing is a one-time activity—the application can use a signed certificate as long as the contents of the certificate do not change.

- *Signature value* This signature is produced by taking the digest of the entire certificate, sans the digest and the digest algorithm, and encrypting it. This means that the digest algorithm name is repeated here as well.

Certificates also allow extensions that can be used to carry additional information. For example, the Certificate Revocation List (CRL) Distributions Points extension specifies the parties one can contact to acquire the list of certificates revoked by the issuing authority. The Certificate Key Usage extension can be used to restrict the uses a certificate may be put to. For example, one may specify that the certificate must be used for signing-only or to identify a CA itself.

The last use we'll mention for certificates is interesting. In the discussion so far, we have assumed that all parties trust a single CA. That assumption is not really required, and can be easily relaxed using the concept of *certificate chain*s. We will discuss this concept next.

Certificate chains

Let's recap what we've described about PKI so far. PKI is a framework that addresses public key distribution. Central to PKI are the concepts of certificate and CA. CAs issue certificates. A certificate is a digital document that ties a public key to a particular identity. Every certificate is digitally signed by the issuing CA using its public key. This eliminates the possibility of fake certificates, unless, of course, the CA makes the mistake of issuing a certificate to an impostor.[4]

For PKI to work, applications need to trust a CA. Furthermore, the CA's identity information and public key should be known a priori to all applications. But what if not every application trusts the same CA? In fact, this is a very common occurrence. If you compare a certificate to a passport and a CA to a national government, it is obvious that a single CA cannot hope to issue certificates for everyone, just as a single national government cannot possibly be the passport-issuing authority for the entire world. PKI accommodates multiple CAs by introducing a concept known as *certificate chaining*. Let's discuss this concept using an example.

Let's say the ACME brokerage company (from our running example introduced in chapter 1) sets up its own CA to avoid the cost of paying an external CA every time it requires a certificate for one of its applications or employees. Now, what if ACME's trade execution application needs to present its certificate to an external application, such as a stock exchange application? The stock exchange application is of course not going to accept certificates issued by ACME's CA.

[4] How CAs guard against granting certificates to imposters is beyond the scope of discussion here.

The stock exchange application may only accept certificates issued by a well-known CA such as, say, VeriSign. There are two options to make ACME's trade execution application work with the stock exchange application. The ACME trade execution application can get a second certificate from VeriSign and present it instead. Or, the ACME trade execution application can present along with its certificate (issued by ACME CA) another certificate issued by VeriSign that documents the identity and public key of the ACME CA. In other words, a certificate issued by ACME CA can be made more credible by chaining it to the certificate issued by VeriSign to ACME CA.

In fact, a certificate chain can consist of more than two certificates. Each certificate in the chain is issued by a CA that can in turn be certified by another CA. Given a certificate chain, an application can validate it by traversing through the chain and checking if there is at least one certificate issued by a CA that the application trusts.

This completes the journey through the basics of encryption. You now know about symmetric encryption, public-key encryption, hybrid encryption, and PKI, perhaps more than you need to know for SOA security.

Returning to SOA security, we will devote the next section to using PKI to secure messages in SOA. To start we will show you how you can create a digital certificate for point-to-point encryption with SSL/TLS. Next we will show how to encrypt in Java using JCE and other APIs. See the callout for other uses of PKI such as authentication.

NOTE *PKI-based authentication* PKI certificates, when combined with digital signatures, provide an alternative to the authentication mechanisms we have discussed in the previous two chapters, namely password-based schemes and Kerberos. Using PKI, each party can prove its identity to the other by signing the message with its private key and attaching its certificate to the message. The other party can:

1. Validate the sender's public key and identity information by checking the CA's signature in the certificate.
2. Verify that the request indeed came from the party identified in the certificate by checking the signature provided.

Notice that this method can be used for mutual authentication as well. Clients can authenticate services and services can authenticate clients. This mechanism is also much more secure than username and password-based authentication, given the problems with passwords we detailed in

the previous two chapters. Because managing the certificates is difficult this scheme is not widely used.

6.3 Programming with digital certificates

So far you saw the theoretical side of encryption. You now know what symmetric encryption is and how to exchange a symmetric key using public-key encryption. You also know how PKI provides a cost-effective framework for public key distribution using digital certificate chains.

Before proceeding to encryption in SOA, let us understand how to create and use digital certificates. This knowledge is required for implementing encryption in SOA. The first step in working with certificates is to create one. In the next section, we will create a *self-signed certificate*—that is, a certificate signed by ourselves acting as our own CA.

6.3.1 Creating digital certificates

JDK supports use of digital certificates through its tools and APIs. We depend on these in our examples throughout the book. This subsection describes these tools and APIs.

A tool named `keytool` is packaged with the JDK to let us manage a key store. A *key store* may contain the certificates for the entities that we're dealing with and our own key pairs with associated certificate chains, if any. Each entry in a key store, be it a certificate or a key pair, is identified by a unique *alias*. Passwords can be used to protect the whole store and each of the private keys we save in the store.

Listing 6.1 shows how to create a key pair and store it in a key store using `keytool`.

Listing 6.1 Creating a key pair and storing it in a key store using Java `keytool`

```
$keytool -keystore example4.keystore -genkey -alias \
  http://manning.com/xmlns/samples/soasecimpl/cop -keyalg RSA \
  -keypass goodpass
Enter keystore password:  goodpass
What is your first and last name?
  [Unknown]:  Prasad Chodavarapu
What is the name of your organizational unit?
  [Unknown]:  Authors
What is the name of your organization?
  [Unknown]:  Manning
What is the name of your City or Locality?
  [Unknown]:  Bangalore
What is the name of your State or Province?
  [Unknown]:  Karnataka
```

```
What is the two-letter country code for this unit?
  [Unknown]:  IN
Is CN=Prasad Chodavarapu, OU=Authors, O=Manning, L=Bangalore,
ST=Karnataka, C=IN  correct?
[no]:  yes
```

Let us examine each of the options shown in listing 6.1.

- `-keystore` is used to set the path to the key store file. If a file does not already exist at the given path, `keytool` creates it.
- `-genkey` indicates to `keytool` that we wish to generate a key pair.
- `-alias` indicates the identifier we wish to assign to the generated key pair. Any string can be used as an alias. In this example, we are using a URI as an alias for reasons we will mention later.
- `-keyalg` is used to choose the key-generation algorithm. In this example, we are generating an RSA key pair.
- `-keypass` is used to specify the password with which we want to protect the generated private key. No one will be able to retrieve the generated private key from the key store unless they have both the store password and the key password.

The `keytool` will prompt us for the store password. If the key store is being created as a result of this command, the given password is set as the store's password. If not, the given password is checked against the store's password.

The `keytool` also generates a self-signed certificate after generating a key pair. A self-signed certificate is one in which the public key and the identity of its owner are signed using the owner's private key instead of a CA's private key. In other words, the certificate owner and the signing CA are one and the same. To create the certificate, `keytool` needs to know who the certificate belongs to. For this reason, it prompts you for the information needed to create a "name" (an X.500 distinguished name to be precise).

Once the certificate is generated, you can inspect and confirm the contents of key store using `keytool` itself, as shown in listing 6.2.

Listing 6.2 Inspecting the contents of a key store using Java `keytool`

```
$ keytool -keystore example4.keystore -list -v
Enter keystore password:  goodpass

Keystore type: jks
Keystore provider: SUN
```

```
Your keystore contains 1 entry

Alias name: http://manning.com/xmlns/samples/soasecimpl/cop
Creation date: Jul 10, 2005
Entry type: keyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Prasad Chodavarapu, OU=Authors, O=Manning, L=Bangalore,
ST=Karnataka,C=IN
Issuer: CN=Prasad Chodavarapu, OU=Authors, O=Manning, L=Bangalore,
ST=Karnataka,
 C=IN
Serial number: 42d04632
Valid from: Sun Jul 10 03:18:34 IST 2005 until: Sat Oct 08 03:18:34
IST 2005
Certificate fingerprints:
  MD5:  2C:11:43:C0:25:FB:02:3B:72:5C:71:79:1F:B7:05:F0
  SHA1: 2B:64:85:3A:35:28:FA:B3:E9:0B:BC:9E:64:F3:8C:DE:C7:7D:01:61
```

Notice that the "names" of the owner and issuer are the same here because this is a self-signed certificate. Now, if you want to get our public key certified by a CA such as VeriSign, you need to prepare what is known as a Certificate Signing Request (CSR). `keytool`'s `-certreq` option comes in handy for generating a CSR, as shown in listing 6.3.

Listing 6.3 Generating a CSR with Java `keytool`

```
$keytool -keystore example4.keystore -certreq -alias \ http://manning.com/
   xmlns/samples/soasecimpl/cop
Enter keystore password:  goodpass
-----BEGIN NEW CERTIFICATE REQUEST-----
MIIBtjCCAR8CAQAwdjELMAkGA... [7 lines deleted]
-----END NEW CERTIFICATE REQUEST-----
```

If you send your CSR to a CA, the CA will return a signed certificate after verifying, often through physical documentation, that the identity you claim in the CSR is genuine. Once the CA returns a signed certificate (or certificate chain in case the CA includes its own certificate signed by some other CA), you can import it into the key store to replace your self-signed certificate. See the official Sun JDK documentation for `keytool`'s `-import` option to understand how you can import the certificate.

6.3.2 Point to point encryption with digital certificates (SSL/TLS)

We saw how to create certificates and store them in a key store. The next task is to use these certificates to encrypt messages. We will start by encrypting username and password, for which there are two choices:

1. We can use HTTPS instead of HTTP as the transport for SOAP. This has the benefit of keeping changes to our application code, both on the client and service side, minimal. We will end up encrypting every bit that is passed on the wire—not just the username and password. Such blanket encryption may or may not be appropriate, as explained the beginning of the chapter.
2. We can change our client and server to only use encryption for username and password.

In this subsection, we will implement the first of these two possibilities. We will implement the second in section 6.4.

HTTPS uses SSL or TLS to encrypt all data transmission over the wire. Both SSL and TLS use hybrid encryption and/or similar techniques to safeguard data confidentiality. As a bonus, we also get free server authentication, as both SSL and TLS require servers to identify themselves using digital certificates.

To use HTTPS transport with SOAP, we need to follow these steps:

1. Create a self-signed certificate to identify the host on which Tomcat is running.
2. Configure Tomcat to use the certificate.
3. Configure the client to trust the certificate.
4. Invoke the web service from the client code via HTTPS.

You have already seen how you can create a self-signed certificate, which is the first step. When creating a certificate for a machine as opposed to a user, you need to provide the machine's fully qualified domain name (FQDN) when `keytool` prompts you for first and last name. Tomcat assumes that its key pair is always stored in the key store using `tomcat` as the alias; so don't forget to use that as the alias when you create the self-signed certificate for the Tomcat host.

The second step of the process is to configure the HTTPS connector in Tomcat to make use of the certificate you created. Tomcat comes with its SSL/TLS connector commented by default. Edit `conf/server.xml` in your Tomcat server's home directory, and uncomment the SSL/TLS connector configuration and set it up as

shown in listing 6.4. You will need to replace the values shown here for `keystoreFile` and `keystorePass` with values that are appropriate for your key store.

Listing 6.4 SSL/TLS connector configuration in Tomcat

```
<Connector port="8443"
  keystoreFile="d:/work/eclipse/soas_code/conf/example4.keystore"
  keystorePass="goodpass"
  ...
  scheme="https" secure="true"
  clientAuth="false" sslProtocol="TLS" />
```

Restart Tomcat. You should now be able to reach the Tomcat homepage on port 8443 of your Tomcat host.

As the third step, we now need to set up our test client to trust Tomcat's certificate. This step is analogous to the step in browsers where we are asked to accept a certificate certified by an unknown CA. In case of browsers, we have GUI wizards to import the certificate. Let us do the same manually, as in listing 6.5, for our test client.

Listing 6.5 Exporting Tomcat's digital certificate from its keystore and importing it into a client keystore

```
$ keytool -keystore example4.keystore -export -alias tomcat \
  -file tomcat.cer
Enter keystore password:  goodpass
Certificate stored in file <tomcat.cer>

$ keytool -keystore client.keystore -import -alias tomcat\
  -file tomcat.cer
Enter keystore password:  goodpass
Owner: CN=localhost, OU=Authors, O=Manning, L=Bangalore,
ST=Karnataka, C=India
Issuer: CN=localhost, OU=Authors, O=Manning, L=Bangalore,
ST=Karnataka, C=India
Serial number: 42dc0c56
Valid from: Tue Jul 19 01:38:54 IST 2005 until: Mon Oct 17 01:38:54
IST 2005
Certificate fingerprints:
   MD5:  C0:AE:CE:9E:A7:1A:51:34:32:72:E6:49:F9:02:6C:7C
   SHA1: CA:43:5E:1E:64:27:CF:66:0B:D5:99:E6:94:71:BE:9E:37:BC:7C:0F
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

Now that we have the client's key store set up to trust Tomcat's certificate, we tell the client to use this key store by setting the `javax.net.ssl.trustStore` system property to the path of the client's key store. In the samples you downloaded in chapter 2, you can do this simply by changing the value for `ssl.truststore` in `build.properties`.

As the final step, we have to modify the client code to use HTTPS. The exact mechanism of these modifications depends on whether our client uses a pregenerated stub. Since we generate the stubs from WSDL, we need to modify WSDL with the right service address to invoke the service via HTTPS. Of course, we have to regenerate the stubs for our client code again. This is the line in WSDL that you have to modify:

Before:

```
<wsdlsoap:address
 location="http://localhost:8080/axis/services/example4"/>
```

After:

```
<wsdlsoap:address
 location="https://localhost:8443/axis/services/example4"/>
```

If we're using a dynamic proxy or DII, we need to make sure that the WSDL fetched by the client at runtime provides the right service address. When serving WSDL for currently deployed services, Axis dynamically sets the protocol and host/port information in the service address based on the address used by the client to fetch WSDL. For example, if the client uses HTTPS to fetch WSDL, Axis provides an HTTPS URI as the service address. So, if we are using dynamic proxy or DII, all we need to do is change the location from which we fetch WSDL at runtime. Here is the relevant code from `example4/BrokerageServiceTestCase.java`.

```
if (sslOrEncryption == USE_SSL) {
    wsdlLocation = "https://" +
        System.getProperty("axis.host","localhost") + ":" +
        System.getProperty("axis.ssl.port", "8443") +
        "/axis/services/example4?WSDL";
} else {
    wsdlLocation = "http://" +
        System.getProperty("axis.host","localhost") + ":" +
        System.getProperty("proxy.port", "8080") +
        "/axis/services/example4?WSDL";
}
```

As you can see, the only difference is the protocol (HTTP vs. HTTPS) and the port number (8080 vs. 8443). Our code has a conditional to test which port we are using, as we use the same code for HTTP and HTTPS. The rest of the code

remains unchanged. In fact, if you examine the code generated by WSDL, you will find it similar to this code—there you will find the service address is hard-coded from WSDL.

We are now ready to run the test (assuming that you have done an `ant deploy` when running previous examples in this book; if not, refer to chapter 2). Edit `build.properties` once again on the client-side and make sure that `axis.host`, `axis.ssl.port`, and `ssl.truststore` are all set to the right values. Run `ant demo -example.id=4`. It results in two test runs. We are only interested in the first one here. The second will error out, as we don't have the setup needed to run that one successfully. We will run it again in section 6.4.

Unlike the earlier cases, we cannot demonstrate what transpires over the net using `tcpmon`. Since the data is encrypted, `tcpmon` cannot act as the proxy. If you would like to look at the traffic, you can use `ethereal`, which we showed earlier. Even `ethereal`, for a purely unrelated technical reason, does not work if the client and the server are on the same machine, if you are using Windows. If you run the client and server on different machines, and capture the packets going to and from the port 8443 on either machine, you will see the traffic going over in HTTPS. Of course, you cannot make sense of it, since it is encrypted.

Limiting SSL/TLS to point-to-point exchanges

The steps we described work—they let the client and server communicate over a secure channel. Is it really safe? Consider the case of a commerce site where you submit your credit card information over SSL. The browser shows you a lock icon, giving you the confidence that nobody in the middle can view your information.

Notice that you have no control over how the data is handled after it reaches the commerce site. The site may in turn submit your confidential data to another agency, without putting in enough safeguards to protect your data. The site's server may also have been compromised by a Trojan that captures the information submitted by clients to send it to rogue machines.

In June 2005, the data from 40 million credit cards was compromised in a similar situation. The data was flowing through a company that verifies cardholder credit for merchants. Even though the company was not supposed to retain the data, the company held it in a database for future analysis. That system was cracked, and consequently all the data was compromised.

The lesson here is that if there are multiple parties in an exchange, SSL/TLS does not provide the end-to-end security that the user needs.

There is one more reason why transport-level encryption is inadequate for web services. HTTPS forces blanket encryption of the complete package, which is

unnecessary in some cases. Services routinely exchange large amounts of data. Encrypting all of that data is costly.

For these reasons, we like to encrypt messages selectively, targeted toward the final intended recipients. We will look into selective encryption in section 6.4. To get to selective encryption, we need to understand how to encrypt using certificates in Java, which is the topic of the next section.

6.3.3 Java APIs for encryption

As we argued, selective encryption is important for SOA security. In this subsection, you will learn how you to use Java APIs to encrypt (and decrypt) arbitrary data. In the next section, you will learn more about selective encryption of SOAP messages, in particular.

The first API you need to get familiar with is the `java.security.KeyStore` class. Given that each security tool vendor may choose a different format for a key store, the `KeyStore` API focuses on presenting a uniform API that can be used with any underlying store. Here is an example showing how you can load a key store.

```
KeyStore keyStore =
  KeyStore.getInstance(KeyStore.getDefaultType());          ❶
keyStore.load
  (keyStoreInputStream, keyStorePassword.toCharArray());    ❷
```

As shown in the code, you first need to create a `KeyStore` instance using the `getInstance` factory method ❶. You need to specify the type of store you will be loading into the instance as an argument. The type name will be provided by your key store vendor. Sun JDK 1.5 ships with the capability to handle two types of stores: JKS and JCEKS. Previous versions of Sun JDK supported only JKS stores. In any case, you can defer this choice by using the value returned by `KeyStore.getDefaultType()`. The default type is JKS on Sun JRE unless you override it using the `keystore.type` property in `<JRE_HOME>/lib/security/java.security`.

Once you have a `KeyStore` instance, loading the contents of a key store into the instance is simple. Provide the password for the key store and an input stream from which the contents can be read to `KeyStore`'s `load` method ❷.

Once you have loaded a `KeyStore` instance, you can retrieve a key from it, be it your own private key or the other party's public key, and use it for encryption or decryption. The java APIs that support you in this are defined by JCE in the `javax.crypto` package. This package provides a uniform API for encryption, key generation, and other cryptographic activities, abstracting away the differences between different algorithms. Listing 6.6 illustrates the use of some of the main classes in this API. This listing shows the first step in the hybrid encryption

scheme illustrated by figure 6.4. A key needed for using a symmetric-key encryption algorithm is dynamically generated by the sender and secured for transmission to the receiver using a public-key encryption scheme that relies on the receiver's public key. A certificate stating the receiver's public key is assumed to be available with the sender in its key store.

Listing 6.6 Dynamically generating a symmetric key and encrypting it using intended recipient's public key

```
SecretKey symmetricKey =
  KeyGenerator.getInstance("DESede").generateKey();

X509Certificate recipientCert = (X509Certificate)
  keyStore.getCertificate(recipientCertAlias);

Cipher keyTransportCipher =
  Cipher.getInstance("RSA");
keyTransportCipher.init
  (Cipher.ENCRYPT_MODE, recipientCert);
byte[] encryptedKeyBytes =   keyTransportCipher.doFinal
  (symmetricKey.getEncoded());
```

❶ Generates symmetric key usable with Triple DES

❷ Looks up recipient's certificate in key store

❸ Creates a cipher based on RSA algorithm

❹ Sets up cipher to use recipient's public key

❺ Cipher encrypts symmetric key

In this example, we first generate a symmetric key ❶ for use with the Triple DES algorithm. The `ede` suffix in the algorithm name, `DESede`, is used to indicate that the cipher text is to be computed as `E(D(E(m,k1),k2),k3)`, where `E` is DES encryption algorithm, `D` is DES decryption algorithm, and `k1`, `k2`, and `k3` are three keys.

Next, we retrieve the intended recipient's certificate from a key store ❷. How we figure out the recipient certificate alias is an interesting issue we have to deal with. The easiest way to answer this riddle is to assume that all certificate aliases in our key store are identical to the recipient URIs.

To secure the symmetric key generated in ❶, we instantiate a `javax.crypto.Cipher` object ❸. The `Cipher` class provides an algorithm-independent API to encrypt and decrypt data. You create a `Cipher` instance by calling the `getInstance` factory method with an argument that describes how you want your data to be encrypted. This argument can simply be the name of an algorithm, but if you understand encryption techniques really well, you can also specify advanced options. For example, when using block encryption, you can specify how data should be split up into blocks and how data should be padded if is shorter than a block. In the listing here, we simply create a cipher that uses the RSA algorithm.

A `Cipher` instance can work in two modes: encrypt and decrypt. It also needs the key required by the encryption/decryption algorithm. We set up either of these in ❹.

The final step is to use the `Cipher` instance to encrypt our symmetric key's bytes. If you want to encrypt something other than a symmetric key, the code is exactly the same. Simply pass the bytes you wish to encrypt to the `doFinal` method ❺.

You now know how to encrypt arbitrary data using Java APIs. To use it to encrypt SOAP messages, we need to understand how the client can select only portions of a SOAP message for encryption and how the server can decrypt it. We will go into those details next.

6.4 Encrypting SOAP messages

Until now, we have discussed encryption in a generic setting. Almost all of the encryption technology we saw can be applied to any data exchanged over the wire. Of course, that means it applies to SOAP messages as well. Encrypting SOAP messages poses additional challenges:

- *Need for selective encryption* SOAP messages can pass through multiple SOAP processing nodes, as we will see in part III of this book. For example, almost all purchases on the web will involve the purchaser, merchant, and at least one financial institution that channels funds for the purchase. Typically, that means different parts of the message may be encrypted for different clients.
- *Need for syntax respecting encryption* SOAP envelopes have to be well-formed XML documents even after we apply selective encryption.
- *Need for including metadata along with encryption* As we may not be encrypting the whole message, we need a standard way to communicate what parts have been encrypted and how.

In fact, some of these challenges are applicable to all XML-based messaging protocols, not just SOAP. A standard named *XML Encryption* provides solutions to these challenges. WS-Security makes extensive use of XML Encryption in specifying how SOAP messages can be encrypted. Now, we will show an example to understand the use of XML Encryption in WS-Security.

6.4.1 Example: Sending user credentials with selective encryption

We will use the same example that we considered earlier when demonstrating point-to-point encryption with SSL/TLS. We want to safeguard the secrecy of username and clear-text password as they are passed over the network as part of a SOAP request. In chapter 4, you saw how username and password are exchanged as part of a WS-Security header. See listing 6.7 to refresh your memory.

Listing 6.7 Header of a sample SOAP message using username/password–based authentication

```
<soapenv:Header>
   <wsse:Security ...>
     <wsse:UsernameToken>
         <wsse:Username>chap</wsse:Username>
         <wsse:Password>goodpass</wsse:Password>
     </wsse:UsernameToken>
   </wsse:Security>
</soapenv:Header>
```

To understand what a selectively encrypted message looks like, let's run an example. Table 6.2 provides instructions for running the example.[5]

Table 6.2 Steps to run the example that illustrates how WS-Security supports selective encryption

Step	Action	How To
1	Set up your environment.	If you have not already set up the environment required to run the examples in this book, please refer to chapter 2 to do so. `ant deploy` installs all the examples.
2	If it is not already running, start TCP monitor.	Run `ant tcpmon` so that you can observe the conversation. Check the "XML Format" check box to allow `tcpmon` to format shown requests and responses.
3	Run the example.	Run `ant demo -Dexample.id=4`. You should be able to view the request-response pairs going through the tcpmon console.

After running the example, examine the web service requests and responses captured by `tcpmon`. Observe how username and password are encrypted in the request:

[5] One or more known issues in Apache Axis 1.x prevent this example from running successfully. See appendix A for a description of these issues.

Listing 6.8 Overview of SOAP header contents after encrypting the `UsernameToken` element shown in listing 6.7

```
<soapenv:Header>
  <wsse:Security ...>
    <xenc:EncryptedKey ...>...</xenc:EncryptedKey>
    <xenc:EncryptedData ...>...</xenc:EncryptedData>
  </wsse:Security>
</soapenv:Header>
```

Compare the header in listing 6.8 with that in listing 6.7. You can see that the `UsernameToken` element is now gone, and instead, two different elements, `EncryptedKey` and `EncryptedData`, appear.

Overview of the EncryptedData element

Take a look at listing 6.9, which shows the simpler of these two: `EncryptedData`. This element is used to replace whatever XML fragment we are encrypting with its encrypted form. In our case, we are replacing the entirety of `UsernameToken` element.

Listing 6.9 A sample `EncryptedData` element

```
<xenc:EncryptedData
  xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
  Type="http://www.w3.org/2001/04/xmlenc#Element"
  Id="EncryptedData-26882784-0" >

  <xenc:EncryptionMethod
    Algorithm=".../xmlenc#tripledes-cbc"/>

  <xenc:CipherData>
    <xenc:CipherValue>...</xenc:CipherValue>
  </xenc:CipherData>

</xenc:EncryptedData>
```

❶ Namespace declared by XML Encryption spec

❷ Indicates that the whole element is encrypted

❸ An identifier for this element

❹ The encryption algorithm used

❺ Encrypted bytes in base64 encoding

The `xenc` namespace prefix used in this example is bound to a URI defined by the XML Encryption standard ❶. Elements such as `EncryptedData` and `EncryptedKey` that are defined by the XML Encryption standard belong to the namespace identified by this URI.

The `Type` attribute specifies what this `EncryptedData` element is substituting for. The XML Encryption spec defines standard values for two of the frequently replaced items: whole element or content within an element. Here (in ❷), we

indicate that the whole element has been encrypted. If we had encrypted only the content within an element, we would have indicated that using `http://www.w3.org/2001/04/xmlenc#Content` as the value of the `Type` attribute.

We defined an identifier for this `EncryptedData` element using an attribute named `Id` (in ❸). The identifier is needed later to associate this element with the key that was used for this encryption.

The `EncryptionMethod` element ❹ specifies the algorithm used for encryption. The XML Encryption spec defines standard URIs to use for popular algorithms such as 3DES, AES, and RSA v1.5. In this example, we are using the standard URI defined for 3DES.

The `CipherData` element ❺ provides the encrypted bytes in one of two ways. As shown here, you can use a `CipherValue` element to provide the encrypted bytes as base64-encoded text. Or, you can use a `CipherReference` element to refer to an external location where the encrypted bytes are available. We will discuss how references work in the next chapter, as XML Signatures make use of them a lot more than XML Encryption does in practice.

Next, we will look into the second new element introduced in listing 6.8, `EncryptedKey`.

Overview of the EncryptedKey element

To understand what goes into `EncryptedKey`, we need to ask ourselves one question: *How can we help the intended recipient locate and decrypt the encrypted parts*? This question is answered by WS-Security as follows: *Prepend* a header entry to a WS-Security header giving out all the information required to help the intended recipient in decryption.

Why prepend? If a particular fragment is encrypted more than once for whatever reason (it is not difficult to imagine situations where a fragment we submit to one of the involved parties is in turn submitted to another as part of a larger encrypted block) then decryptions have to happen in the right order. If all the encryptors prepend header entries describing what has been encrypted then the decryptors can simply process each of these header entries sequentially and do decryption in the right order.

What information should be in the header entries in order to help the decryptor?

- The decryptor needs to know which encrypted nodes it should decrypt. Why is this necessary? For example, can't the decryptor look for all `EncryptedData` elements in the envelope and decrypt them? Think about it—not all encrypted elements in the message may be intended for a single

recipient. Each recipient needs to somehow know which of the encrypted elements are for its consumption and which are not. So, what we need is a `ReferenceList` of encrypted elements.

- Each `EncryptedData` element is already giving out the algorithm used. What it is not giving out is the key used. If the key used is somehow shared a priori between the two parties we really don't need to give it out. For example, if the key can be computed from a password, we don't need to add the key information (except when we are encrypting the username and password themselves, as in our example). Instead, if we are using a dynamically generated key like in hybrid encryption, we need to attach an `EncryptedKey`. In addition, if more than one key has been used in encrypting different parts of the document, we need to specify which encrypted elements can be decrypted by each key. That is, we also need a `ReferenceList` of elements encrypted by the key as part of an `EncryptedKey` element.

To summarize, we need to prepend to the WS-Security header:

- A `ReferenceList` if the encryption key is some how agreed upon a priori.
- One or more `EncryptedKey` entries if the key used for data encryption itself needs to be transported to the other party. Each `EncryptedKey` element should in turn consist of a `ReferenceList` of elements encrypted by that key.

As a `ReferenceList` is also needed as part of `EncryptedKey` elements, we can take a look at what is in an `EncryptedKey` element, and, in the process, learn about `ReferenceList` as well. Listing 6.10 shows a sample `EncryptedKey` element.

Listing 6.10 A sample `EncryptedKey` element

```
<xenc:EncryptedKey
  xmlns:wsse="...wssecurity-secext-1.0.xsd"
  xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
  xmlns:ds="http://www.w3.org/2000/09/xmldsig#">

  <xenc:EncryptionMethod
    Algorithm=
    "http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>

  <ds:KeyInfo>
    <wsse:SecurityTokenReference>
      <ds:X509IssuerSerial>
        <ds:X509IssuerName>
          CN=Prasad Chodavarapu,OU=Authors,O=Manning,...
        </ds:X509IssuerName>
```

❶ **Binds namespace prefixes to URIs**

Identifies algorithm used for encrypting the key

❷ **Identifies a different key used for encrypting the key**

```
        <ds:X509SerialNumber>
          1120945714
        </ds:X509SerialNumber>
      </ds:X509IssuerSerial>
    </wsse:SecurityTokenReference>
  </ds:KeyInfo>

  <xenc:CipherData>
    <xenc:CipherValue>...</xenc:CipherValue>
  </xenc:CipherData>

  <xenc:ReferenceList>
    <xenc:DataReference URI="#EncryptedData-26882784-0"/>
  </xenc:ReferenceList>
</xenc:EncryptedKey>
```

❷ **Identifies a different key used for encrypting the key**

❸ **Provides encrypted key bytes in base64 encoding**

❹ **Points to elements encrypted by the key**

You have previously seen the `xenc` and `wsse` prefixes being bound to namespaces defined by the XML Encryption and WS-Security specs, respectively. The new one you see in ❶, `ds`, is bound to the namespace URI defined by the XML Signature spec that we will discuss in the next chapter.

The `EncryptionMethod` element identifies the algorithm used for encrypting the key. Here, we are using RSA v1.5 to do hybrid encryption. That is, we are using public-key encryption to encrypt the key that is used for symmetric encryption.

We said we are using RSA v1.5, but how do we say which of the recipient's public keys (if it possesses more than one) is used for symmetric-key encryption? The `KeyInfo` element is used to convey this information ❷. In this particular case, we are using the recipient certificate's issuer name and serial number to identify the public key used. There are quite a few other ways we could have done this. We could have embedded the entirety of the certificate as base64-encoded data in a `<wsse:BinarySecurityToken>` element (just like we embedded a Kerberos service ticket in the previous chapter) with the `ValueType` attribute set to `wsse:X509v3`. To keep the discussion simple, we will not list all the possible ways of specifying `KeyInfo`. You can refer to the WS-Security X.509 Certificate Token Profile spec for a discussion of all the possibilities.

Just as we did in `EncryptedData` (see listing 6.9), the `CipherData` element provides encrypted bytes using a `CipherValue` ❸ element. This time, of course, what we are encrypting is the dynamically generated key that needs to be transmitted securely.

As discussed previously, we need to identify exactly which `EncryptedData` elements can be decrypted with the key described by the `EncryptedKey` element. This

identification is done using a `ReferenceList` element ❹. How can we identify a particular `EncryptedData` element in a document? If an `Id` is set on the element (see callout for more details on identifiers), we can refer to it using the `#Id` syntax defined in the URI spec to identify fragments of a resource. That is exactly what we do here in the `DataReference` element.

NOTE *All you need to know about identifiers in XML* In this chapter and the next, we often need to identify elements that are encrypted, digested, or signed. How can we unambiguously identify an element in an XML document? A specification named *XPointer* answers this question for elements as well as other information items in a XML document. Without going into too much detail on the XPointer specification, we can simply say that the two most obvious methods to identify elements are XPath expressions and identifiers. Identifiers are often preferred over XPath for reasons of simplicity and performance. Because of historical reasons, there is considerable confusion over what constitutes an identifier in XML.

The XML 1.0 specification does talk of identifiers, but leaves the choice of the attribute that will hold the identifier value to the DTD or schema. An attribute declared to be of type ID in the DTD/schema provides the identifier value for the element it is defined on. Of course, an identifier value cannot appear more than once in a document, even if different elements use differently named attributes of type ID.

The problem with schema-determined identifiers is that an application needs to be in possession of the DTD/schema to know which attribute values constitute the identifiers. SOAP handlers often do not have the DTD/schema for the XML they are acting on. For this reason, WS-Security defined a special attribute named `wsu:Id` (where `wsu` is a namespace prefix that is bound to the WS-Security–Utility namespace) for holding the identifier value, except in elements defined by the XML Encryption and XML Signature specifications that already have an `Id` (in no namespace) attribute defined on them. In this book, we stick to this definition of an identifier.

At the time of this writing, a special attribute named `xml:id` is being proposed along the same lines as `wsu:Id` for use in all XML-based applications. The example code provided with this book does not understand `xml:id`-based identifiers.

Let's recap what you have learned in this section so far.

- WS-Security supports selective encryption of elements in a SOAP message.
- Each encrypted element (or its content, depending on what is encrypted) is replaced by an `EncrytedData` element that specifies the encryption algorithm and the encrypted bytes.
- If the key used for encryption has been prearranged, a `ReferenceList` element can be used in the `Security` header entry to identify which elements in the message are encrypted.
- If the encryption key is a dynamically generated one that needs to be securely shared with the other party, an `EncryptedKey` element can be used in the `Security` header entry to transport the generated key securely. In this case, the list of the message elements encrypted by the dynamically generated key is identified using a `ReferenceList` element nested inside the `EncryptedKey` element.

Now that you understand how WS-Security supports selective encryption in SOAP messages, you are ready to look into the code that implements all these details. Figure 6.6 provides an overview of the implementation strategy we adopted in this example.

Assuming that you have read at least one the past three chapters, figure 6.6 should look very similar to what you have seen before. On the client-side, we have an `EncryptionHandler` that is responsible for encryption, and on the server-side, we have a `DecryptionHandler` that is responsible for decryption. In this example implementation, the subject of encryption and decryption happens to be the `UsernameToken` element created on the client-side by the `ClientSideWSSecurityHandler` and consumed on the server-side by the `WSSecurityUsernameHandler`.

Just as in the previous chapters, we will dig into the details of the client- and server-sides separately. To emphasize that `EncryptionHandler` and `DecryptionHandler` will switch sides if we discuss encryption of response messages (messages originating from the server-side and addressed to the client-side), we will use the terms *encrypting side* and *decrypting side* instead of client-side and server-side. Let's start with the encrypting side.

6.4.2 Encrypting-side implementation

In this subsection, we will show how to implement the encrypting side.

In figure 6.6, we can see that we have two handlers on the client-side. The first one is `ClientsideWSSecurityHandler`, which we discussed in the previous chapter.

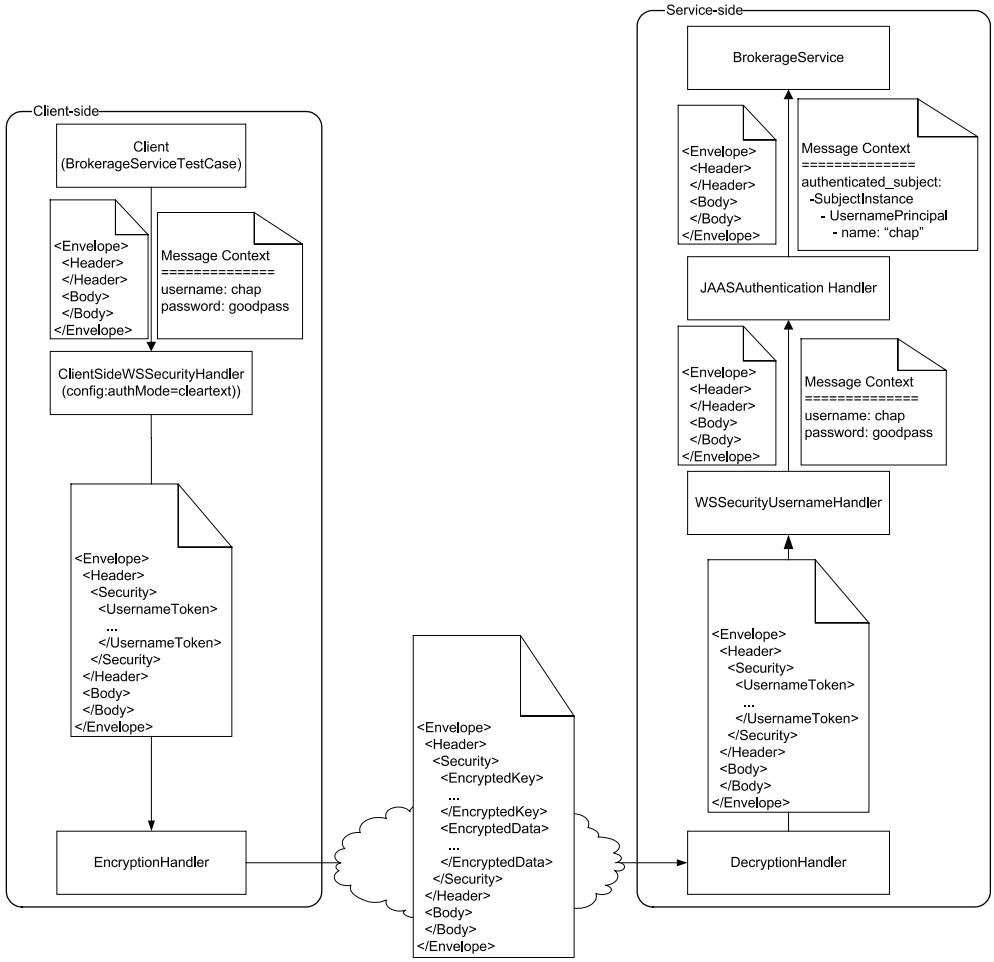

Figure 6.6 Sample implementation of SOAP encryption and decryption. In this example, `UsernameToken` is encrypted by a client-side `EncryptionHandler` and recovered by `DecryptionHandler` on the service-side.

It is responsible for inserting the username and password in the headers. The second handler, called `EncryptionHandler`, implements encryption using the hybrid encryption scheme shown in figure 6.4.

To selectively encrypt parts of a message, the `EncryptionHandler` needs to carry out the steps identified in table 6.3.

Table 6.3 Steps to selectively encrypt parts of a message

Step	Action
1	Identify the portions of the message to encrypt.
2	Pick an algorithm and a key to encrypt the parts of the message.
3	Extract portions of the message identified in step 1 and replace them with encrypted data.
4	Add information on the encryption carried out to the `Security` header. As we will be using a hybrid encryption scheme in this example, this task is again divided into two. First, encrypt the key used in step 2 and add it to the `Security` header along with list of nodes encrypted by the key. Next, add to the `Security` header information (such as the public key used) on how the key is encrypted.

All the code for these steps is in the file `EncryptionHandler.java`. This class heavily depends on the functionality of `org.apache.xml.security.encryption.XMLCipher` from Apache's XML Security library. `XMLCipher` provides a higher-level API than `javax.crypto.Cipher` (introduced in section 6.3.3) for encrypting XML. It subsumes the functionality of `Cipher` and provides facilities to parse/serialize and replace encrypted/decrypted elements in a DOM.

Before we explain the implementation of steps in table 6.3, we need to tell you how `EncryptionHandler` is initialized. We will do that next.

Initial setup

There is some setup that needs to be done before `EncryptionHandler` can do the four steps in table 6.3. We need to initialize Apache's XML Security library. Since this needs to be done only once, we implement this as a static call in the `EncryptionHandler` class:

```
org.apache.xml.security.Init.init();
```

In addition, when the `EncryptionHandler` is initialized (that is, when its `init()` method is called), we need to configure it with answers to the following questions:

1. *In which direction is this handler being used: request or response?* The answer to this question is needed to determine which of the two methods—`handleRequest` or `handleResponse`—should do encryption.
2. *Which of the message elements needs to be encrypted?* We need to specify the XPath for these elements.
3. *Should we encrypt entire elements or just their content?* By default, entire elements are encrypted.

4. *Which symmetric-key encryption algorithm should we use?* By default, the handler uses 3DES. Of course, we can configure the handler to use any symmetric-key algorithm.
5. *Which public key encryption algorithm should we use?* By default, the handler uses RSA v1.5.
6. *Where is the key store?* Here is where the handler looks for the certificate of the intended receiver.
7. *Which certificate in the key store belongs to the intended receiver?* The handler needs to know the certificate's alias in the key store, in order to retrieve the certificate.

By the end of the initialization, the code knows answers to all these questions from the configuration file. Most of the initialization code is straightforward except for the code that compiles XPath expressions. The `javax.xml.xpath` package in Sun JDK 1.5 is very well documented. Take a look at the javadocs for that package (reference 5 in the "Suggestions for further reading" section at the end of this chapter) to understand how to compile and evaluate XPaths in java.

Now that you know how `EncryptionHandler` is initialized, let's look at how it carries out each of the four steps we identified in table 6.3.

Step 1: Identifying the nodes to encrypt

This is the first step that needs to be done when the handler is called. In our example, the `soapenv:Header/wsse:Security/wsse:UsernameToken` element that needs to be encrypted. It is possible to encrypt any element in the complete envelope, including already-encrypted data. Here is how we get the list of the nodes[6] to encrypt:

```
NodeList nodesToEncrypt = (NodeList)
                    xpathToEncrypt.evaluate
                    (soapEnvelope, XPathConstants.NODESET);
```

The second argument to the function makes it return all the nodes that match the XPath, which is exactly what we want. Later on, we will encrypt these nodes and replace the originals with the encrypted nodes. For now, let's look at how `EncryptionHandler` implements step 2.

---

[6] Although we use the word "node" here, what we really mean is an *element*. The XML Encryption spec supports encryption at three granularities: document, element, and element content. In all three cases, what needs to be encrypted can be identified by an element.

Step 2: Picking an algorithm and the key for encryption

Once the `EncryptionHandler` identifies the list of nodes to encrypt, it needs to pick an algorithm and the key for encryption. It already knows which algorithm it should use; the handler got that information during initialization. Now, it needs to generate a symmetric key.

In section 6.3.3, we introduced you to the JCE APIs, which are useful when implementing encryption. Listing 6.6 showed you to how to use the JCE `KeyGenerator` API to generate a key. We use the exact same code in `EncryptionHandler`, but we need to do some additional work here.

`KeyGenerator` needs a JCE-given name for the algorithm to generate a key. Apache XML Security library, on the other hand, needs the name of the algorithm as a URI standardized by the XML Encryption spec. We use the URI form when configuring the handler. So, we need to convert the algorithm name from URI form to JCE-given name as shown in the following code.

```
String algorithmJCEName= JCEMapper.
               getJCEKeyAlgorithmFromURI(dataEncryptionAlgo);
KeyGenerator keyGenerator = KeyGenerator.getInstance
               (algorithmJCEName);
SecretKey symmetricKey =
                keyGenerator.generateKey();
```

So, we know the nodes to encrypt and the encryption algorithm. Let's replace the original ones with the encrypted nodes—that is, selectively encrypt the part of the message.

Step 3: Replacing original nodes with encrypted nodes

Now that the `EncryptionHandler` has generated a key, it can extract the nodes identified in step 1 and replace each of them with an `EncryptedData` element, like the one shown in listing 6.9. When doing this step, the `EncryptionHandler` also needs to assign a unique identifier to each of the `EncryptedData` elements it creates. These identifiers will come in handy in step 4, when the `EncryptionHandler` adds a `ReferenceList` of encrypted elements to the `Security` header. Listing 6.11 shows the code for implementing this logic.

Listing 6.11 Replacing nodes to encrypt with `EncryptedData` elements

```
String prefixForEncryptedNodeIds =                    ❶ Generates a prefix for
    "EncryptedData-" + hashCode();                      EncryptedData identifiers
for (int i = 0; i < numNodesToEncrypt; ++i) {
    Node ithNodeToEncrypt = nodesToEncrypt.item(i);
    . . .
```

```
XMLCipher xmlDataCipher =
    XMLCipher.getInstance(dataEncryptionAlgo);
xmlDataCipher.init
    (XMLCipher.ENCRYPT_MODE, symmetricKey);
xmlDataCipher.getEncryptedData().setId
    (prefixForEncryptedNodeIds + "-" + i);
xmlDataCipher.doFinal
    (soapDoc,
     (Element) ithNodeToEncrypt,
     onlyEncryptElementContent);
}
```

❷ **Instantiates XMLCipher**

❸ **Sets XMLCipher to encrypt using given key**

❹ **Sets identifier of the EncryptedData element**

❺ **Replaces original node with EncryptedData element**

The primary goal of this listing is to show you how to use the XMLCipher API for encrypting elements in a SOAP message.

You can instantiate the `XMLCipher` class using the `getInstance` factory method ❷. This method takes as an argument the algorithm you intend to use for encryption. The algorithm must be in the form of a URI. For all the available URIs, you can refer to the class `org.xml.security.utils.EncryptionConstants`.

Just like a JCE Cipher instance, XMLCipher can work in two modes: encrypt and decrypt. As this listing is part of `EncryptionHandler`, the `XMLCipher` instance is set to work in the encrypt mode using the `init` method ❸. The `init` method also takes as an argument the key to use for encryption.

Post initialization, an `XMLCipher` instance knows what algorithm and key to use for encryption. The only thing it does not know yet is what to encrypt. `XMLCipher`'s `doFinal` method ❺ gets this information through its arguments and replaces the identified element in an XML document with an `EncryptedData` element. The arguments to `doFinal` are the document to which the node to encrypt belongs, the node to encrypt, and a Boolean value to specify whether to encrypt the entirety of the given node or just its content.

There is one important detail in listing 6.11 that we have yet to describe. As we noted earlier, the `EncryptionHandler` needs to assign a unique identifier ❹ to each of the `EncryptedData` elements it creates so that it can refer to them later when creating a `ReferenceList` of `EncryptedData` elements. The `EncryptionHandler` can use "EncryptedData-*i*" as the unique identifier for the *i*-th `EncryptedData` element it generates. The uniqueness of these identifiers may be violated if more than one instance of an `EncryptionHandler` is used on the encrypting side (possibly to encrypt different parts of the same message differently). The strategy used here incorporates the `hashcode` of the `EncryptionHandler` instance ❶ in the unique identifiers generated for `EncryptedData` elements.

We are done looking at how `EncryptionHandler` implements three of the four steps we identified earlier. The only step that's left now is to add encryption information to the `Security` header entry. Let's see how to implement that next.

Step 4: Adding info on encryption to the Security header

In step 2, the `EncryptionHandler` generated a key and used it in step 3 for encryption. As the decrypting-side will need the same key for decryption, the `EncryptionHandler` needs to communicate this key securely to the decrypting-side. As shown in listing 6.10, WS-Security provides an `EncryptedKey` element for this purpose. In this section, we will show you the code for creating an `EncryptedKey` element and adding it to the `Security` header. We will also show you how to add information on the public key used for generating the `EncryptedKey` element.

To produce the `EncryptedKey` element, we turn once again to the Apache XML Security library. This library provides us with classes such as `EncryptedKey` and `ReferenceList`. We fill in instances of these classes with data, and at the end use their serialization facilities to convert the data into XML form.

The activity in this step can be subdivided into the parts shown in table 6.4.

Table 6.4 Steps to selectively encrypt parts of a message

Step	Action
4a	Look up the decrypting side's certificate in the key store specified by the handler's configuration.
4b	Encrypt the symmetric key used for encryption in step 3. For this encryption, use the public-key algorithm specified by the handler's configuration and the certificate retrieved in step 4a.
4c	Construct an instance of the `EncryptedKey` class using the result of step 4b.
4d	Construct an instance of the `ReferenceList` class, populate it with the identifiers of `EncryptedData` elements generated in step 3, and add it to the `EncryptedKey` instance constructed in step 4c.
4e	Add, a `SecurityTokenReference` to `EncryptedKey` to refer to the recipient's certificate used in step 4b.
4f	Serialize the `EncryptedKey` instance as XML and add it to `Security` header entry.

We will walk you through the code for each of these steps next, starting with step 4a, which is shown in listing 6.12.

Listing 6.12 (Step 4a) Looking up the decrypting side's certificate in the key store

```
X509Certificate recipientCert = (X509Certificate)
     keyStore.getCertificate(recipientCertAlias);
```

```
if (recipientCert == null) {
    throw new RuntimeException(
            "Did not find a certificate in the keystore for: "
                    + recipientCertAlias);
}
```

In this listing, we simply use the Java `KeyStore` API introduced in section 6.3.3 to retrieve the decrypting side's certificate from a key store. The location of the key store and the recipient certificate's alias come from the handler configuration, as we described earlier.

Next we need to encrypt the symmetric key generated in step 2 with the public key in the recipient's certificate. The `XMLCipher` class we used in step 3 for encryption does not support public-key encryption. Since we need to encrypt the key with public-key encryption, we fall back to JCE APIs, as shown in listing 6.13.

Listing 6.13 (Step 4b) Encrypting the symmetric key used for encryption in step 3

```
Cipher keyTransportCipher = Cipher.getInstance
    (JCEMapper.translateURItoJCEID(keyEncryptionAlgo));
keyTransportCipher.init(Cipher.ENCRYPT_MODE, recipientCert);
byte[] encryptedKeyBytes =
    keyTransportCipher.doFinal(symmetricKey.getEncoded());
```

Listing 6.13 is very similar to listing 6.6 but for one minor detail. As JCE does not recognize the URIs used in `EncryptionHandler` configuration to identify encryption algorithms, we first have to convert the key encryption algorithm URI into a name that JCE recognizes. Once we do that, the rest of the code is the same as in listing 6.6. By the end of this snippet of the code, we get the encrypted bytes. Now we need to put these bytes in `XMLCipher` as shown in listing 6.14.

Listing 6.14 (Step 4c) Instantiating the `EncryptedKey` class using the result of step 4b

```
XMLCipher xmlKeyCipher =
    XMLCipher.getInstance(keyEncryptionAlgo);
EncryptedKey encryptedKey =
    xmlKeyCipher.createEncryptedKey
        (CipherData.VALUE_TYPE,
         new String
         (Base64.encodeBase64(encryptedKeyBytes),
           "US-ASCII"));
```

To construct an `EncryptedKey`, we first need to create an `XMLCipher` instance. Once we do that, we can use the `createEncryptedKey` method in `XMLCipher` to construct an `EncryptedKey`. The first argument we pass to this method, `CipherData.VALUE_TYPE`, indicates that the `CipherData` element in the `EncryptedKey` will contain a `CipherValue` (the encrypted key bytes) as opposed to a `CipherReference` (a reference to another element). The second argument provides the base64-encoded bytes in the encrypted key.

The next step is to add a `ReferenceList` containing the identifiers of the `EncryptedData` elements generated in step 3 to the `EncryptedKey` instance.

Listing 6.15 (Step 4d) Adding `ReferenceList` to `EncryptedKey`

```
ReferenceList encryptedDataRefList =
     xmlKeyCipher.createReferenceList
     (ReferenceList.DATA_REFERENCE);
for (int i=0; i < numNodesToEncrypt; ++i) {
     encryptedDataRefList.add
          (encryptedDataRefList.newDataReference
          ("#" + prefixForEncryptedNodeIds + "-" + i));
}
encryptedKey.setReferenceList(encryptedDataRefList);
```

We construct a `ReferenceList` instance using a method in `XMLCipher` named `createReferenceList`. The argument we pass to this method, `ReferenceList.DATA_REFERENCE`, indicates that the constructed `ReferenceList` will refer to `EncryptedData` elements (as opposed to `EncryptedKey` elements). Once we have a `ReferenceList` instance, we add to it references to each `EncryptedData` element created in step 2. Observe that we are computing the identifiers for `EncryptedData` elements the same way we computed earlier, thus making sure that the references work.

We also need to add information about the recipient's certificate that was used to encrypt the key to `EncryptedKey`. As shown in listing 6.10, this is done using a `KeyInfo` element that wraps a `SecurityTokenReference`. Listing 6.16 shows the code for this step.

Listing 6.16 (Step 4e) Adding a reference to the recipient's certificate to `EncryptedKey`

```
Element securityTokenReference = soapDoc.createElementNS
     (Constants.WS_SECURITY_NS_URI,
      Constants.WS_SECURITY_TOKEN_REF_TAG);
```

❶

```
securityTokenReference.appendChild
     (new XMLX509IssuerSerial(soapDoc, recipientCert).
             getElement());                                    ❷
KeyInfo keyTransportKeyInfo = new KeyInfo(soapDoc);            ❸
keyTransportKeyInfo.addUnknownElement
     (securityTokenReference);
encryptedKey.setKeyInfo(keyTransportKeyInfo);                  ❹
```

The `KeyInfo` ❸ class in the Apache XML Security library does not know what a `SecurityTokenReference` element is. Fortunately, the class allows addition of an unknown DOM `Element` to `KeyInfo` ❹. We fall back on this facility here. We first create a `SecurityTokenReference` element ❶ and add to it the information on the recipient's certificate used in key encryption. The certificate information is added as an instance of `XMLX509IssuerSerial` ❷, another class defined by the Apache XML Security library.

Now that we have filled out all the information required in an `EncryptedKey` instance, we can serialize it as an XML element and prepend it to the `Security` header entry, as shown in listing 6.17.

Listing 6.17 (Step 4f) Serializing the `EncryptedKey` instance as XML and prepending it to the `Security` header

```
Element encryptedKeyElement = xmlKeyCipher.martial
                (soapDoc, encryptedKey);
securityElement.insertBefore
     (encryptedKeyElement, securityElement.getFirstChild());
```

This completes the encryption logic on the encrypting side. Let us now look at the decryption logic on the decrypting-side.

6.4.3 Decrypting-side implementation

Since we already studied the encrypting-side code carefully, the decrypting-side code is easier to understand.

In the example illustrated by figure 6.6, we see three handlers on the server-side. Of these, you have already seen the `WSSecurityUsernameHandler` and `JAASAuthenticationHandler` in previous chapters. We will restrict our description here to the only new handler on the server-side, `DecryptionHandler`.

The logic in `DecryptionHandler` can be divided into the steps shown in table 6.5.

Table 6.5 Steps to decrypt parts of a SOAP message that are encrypted using a hybrid encryption scheme

Step	Action
1	Identify the `Security` header entry that is applicable to the decryptor. You will see in chapter 8 that there may be multiple `Security` header entries in a SOAP message. The decryptor has to identify the `Security` header entry that it should look into, based on the `actor` attribute we will be describing in chapter 8.
2	Look for `EncryptedKey` elements in the `Security` header entry identified by step 1. For each element found, carry out the rest of the steps.
3	Identify the certificate used to encrypt the encryption key. Information on the certificate will be available in the `KeyInfo` child of the `EncryptedKey` element retrieved in step 2.
4	Look up the key store to find the private key corresponding to the public key in the certificate identified by step 3.
5	Decrypt the encryption key using the private key identified in step 4.
6	Using the encryption key retrieved in step 5, decrypt each of the `EncryptedData` elements referred to in the `ReferenceList` child of `EncryptedKey`. Replace each of the `EncryptedData` elements with the decrypted data.
7	Remove the processed `EncryptedKey` element from the `Security` header entry.

Before any of these steps are done, the `DecryptionHandler` is to be initialized of course. Because the initialization code for `DecryptionHandler` is similar to that of `EncryptionHandler`, we will not be showing it here.

We will also not show you the full code for all of the steps listed in table 6.5. We will only describe the structure and flow of control in the `DecryptionHandler`. You can look up the rest of the source code from the `DecryptionHandler.java` file under the `java/com/manning/samples/soasecimpl/example4` folder in the examples archive you downloaded in chapter 2.

The code for step 1 is in the function `getSecurityElementForThisTarget`, an extract of which is shown in listing 6.18.

Listing 6.18 (Step 1) Locating the relevant `Security` header entry

```
soapEnvelope = message.getSOAPPart().getEnvelope();
soapHeader = soapEnvelope.getHeader();
securityElement = Utils.getHeaderByNameAndActor
 (soapEnvelope, Constants.WS_SECURITY_SECURITY_QNAME,
  roleSet, true);
```

In step 2, we take the `Security` element located by code in listing 6.18, iterate through its children and look for `EncryptedKey` elements in it. For each `EncryptedKey` element found, we need to do steps 3-7, shown in listing 6.19. The high-level code for these tasks is in the function `processEncryptedKey`:

Listing 6.19 (Steps 3-7) Code for processing an `EncryptedKey` element

```
XMLCipher xmlKeyCipher = XMLCipher.getInstance();                    ❶
xmlKeyCipher.init(XMLCipher.DECRYPT_MODE, null);

EncryptedKey encryptedKey = xmlKeyCipher.loadEncryptedKey            ❷
    (soapDoc, encryptedKeyElement);
String keyEncryptionAlgo =                                           ❸
    getKeyTransportAlgorithm(encryptedKey);
byte[] decryptedKeyBytes = getDecryptedKey(encryptedKey,             ❹
                              keyEncryptionAlgo);

ReferenceList refsToEncryptedData =                                  ❺
    encryptedKey.getReferenceList();
decryptData
   (soapEnvelope, refsToEncryptedData, decryptedKeyBytes);           ❻
```

In this code, we first initialize an `XMLCipher` instance to work in decrypt mode ❶. We pass to it the `EncryptedKey` element so that it can parse the element and create an object form of it ❷. The `getKeyTransportAlgorithm` method ❸ of `DecryptionHandler` looks up the `EncryptionMethod` child in the `EncryptedKey` element to find the public-key encryption algorithm used for key encryption. If the `EncryptionMethod` child is not found, the algorithm is assumed to be RSA v1.5. The `getDecryptedKey` method ❹ of `DecryptionHandler` uses JCE APIs to decrypt the key using the algorithm identified in ❸.

Once the decrypted key is available, the `decryptData` method of `DecryptionHandler` iterates through the `EncryptedData` elements identified by `ReferenceList` ❺ and replaces them with the decrypted data ❻.

This completes the high-level walk-through of code on the decrypting side. We have not explained much of the decryption code here, as you can easily follow from our discussion of the code on the encrypting side. As we said before, you can read the entire source code in `DecryptionHandler.java`.

You probably now know all that you need to know about encryption of SOAP messages. We will conclude this chapter with a discussion of the practical issues in encryption.

6.5 Practical issues with encryption

If encryption is so good, why not use it everywhere? Are there any guidelines on where to use it? Are there any drawbacks in using it? Encryption poses interesting choices and questions such as what to encrypt, how to encrypt, and what kind of tools we should use.

Ideally you would like to encrypt every bit of information, but that certainly has drawbacks. For example, if we encrypt all the credit card transactions, including the total amount to be paid, we may not able to write applications that can audit or generate alerts based on the amount. Moreover, sometimes it may be legally necessary for us to leave some of the information in clear text.

When evaluating encryption strategies, we need to look at the infrastructure requirements for each strategy. For example, if we choose PKI and run our own CA, the life cycle processes for a certificate—allocation, management, renewal, and revocation—can get costly and may require special-purpose software.

Coming to the actual implementation of encryption, there are two issues: one is interoperability with existing encryption schemes and the other is inadequacy of implementations. Interoperability is an issue when picking an encryption algorithm, for instance. For example, if an existing application or platform cannot deal with AES, we may have to use a different algorithm.

NOTE *Addressing interoperability problems that may arise out of encryption* SOA places strong emphasis on interoperability, so it is important to design generic solutions to the interoperability problems that may arise from the different capabilities and requirements of different parties. For example, if a client implementation knows how to use 3DES as well as AES, and a service implementation only supports 3DES, how does the client figure out that it should stick to 3DES when talking with that service? We will address this question in chapter 9, where we introduce the ideas of security policy declaration and intersection.

One shortcoming we notice in most implementations, despite the use of mature tools and libraries, is that they do not handle certification revocation well. They may not verify the revocation lists often enough to identify technically valid, yet canceled, certificates.

Even with a good implementation, we are not ensured of the confidentiality of the message. Ultimately, the security system can only be as good as its weakest link. The human element often turns out to be the weakest link in security. We

should have comprehensive logging, auditing, and password management policies in place when designing a secure application.

The question of the right choice of encryption toolkits to use does not have a single answer. As you can guess, there is no single toolkit that can solve all needs. For example, we presented a highly useful toolkit from Apache in this chapter. The `XMLCipher` class, which provides so much support for encrypting XML, does not support PKI when it comes to key transport. Since the APIs[7] are not yet standard, we may get locked into an inferior toolkit. Until good solutions emerge, we may have to build our own toolkits on top of existing ones to fill in the missing functionality.

When proposing a security solution, one always has to watch out for the impact of security enforcement on the overall performance of the system. Encryption and decryption add a lot of computational overhead. This is the primary reason behind the popularity of hybrid encryption, in which we get the convenience of PKI without losing out on the efficiency offered by symmetric-key encryption. In addition, these days hardware-based PKI systems offer an efficient solution for managing security. Even without special hardware, there is one particular optimization we can make in the way we use hybrid encryption schemes in SOAP message exchanges (see callout).

NOTE *More efficient use of the hybrid encryption scheme with WS-SecureConversation* Take a look back at figure 6.4, which pictorially describes the concept of hybrid encryption. In this picture, a symmetric key is established up front using public-key encryption and used for encryption in all further exchanges. If you look at the way we implemented encryption of SOAP messages using WS-Security header entries, you will see that there is no reuse of previously established symmetric keys. The `EncryptionHandler` is generating a new symmetric key for every message and adding it to the WS-Security header as an `EncryptedKey`. This kind of implementation will mean additional per-message CPU cost for key generation, key encryption, and key decryption, as well as network costs for transmitting the encrypted key every time. If we can somehow reuse the symmetric key established in the first message for all subsequent exchanges, we will eliminate the per-message cost and simply pay a fixed cost up front for key establishment. WS-SecureConversation makes this possible. WS-SecureConversation is described in appendix B.

[7] JSR-106 is charged with standardizing these APIs.

In summary, one needs to consider a range of issues when implementing encryption in real-world enterprises. In this section, we sought to provide you with a sampling of issues you are likely to see in practice.

6.6 Summary

In this chapter, we studied the ways we can protect the confidentiality of messages. We started out by looking at the theory behind encryption. Despite the complex mathematics of the actual encryption algorithms, the basics of encryption are simple. We hope you understood the various concepts such as symmetric and asymmetric algorithms and how they come together to solve the needs of encryption in real-world applications.

We also told you about the infrastructure needed for using encryption in practice. In particular, we explained the need for PKI. We introduced you to the ideas behind PKI —digital certificates, digital signatures, CAs, and certificate chaining. Finally, we also showed you the actual details in a certificate.

Even though all this material is not directly related to SOA security, we needed to cover this ground for two reasons. We believe that the practitioners of SOA security should understand the mechanisms of encryption that are used everywhere. We also believe that you may have to build on top of inadequate toolkits or vendor offerings which require you to understand this material.

Once we understood the theory, we looked at the JCE classes that are useful for implementing encryption schemes in Java. We showed how to create and examine digital certificates. Through an example, we showed how to use Java APIs for encrypting point-to-point communications using digital certificates.

There are special requirements for using encryption with SOAP. In particular, for selective encryption of SOAP messages, we need to understand how to use XML Encryption. As usual, WS-Security standards help us communicate the details in the header elements so that the receiving party can understand how to decrypt the elements in SOAP message. We walked through the code for two JAX-RPC handlers that use the Apache XML Security library and JCE for encrypting and decrypting SOAP messages.

If you followed the examples provided in this chapter, you gained enough knowledge to send and receive encrypted SOAP messages. Even though we illustrated encryption in header elements, you can use the same code to encrypt any element in the message.

As we briefly described in the section on signatures, encryption does not protect against data tampering. To protect against such threats, we will use digital

signatures. You have seen the basics of signatures in this chapter. However, using them in conjunction with SOAP and WS standards poses additional challenges. These challenges and the corresponding solutions are the subject of the next chapter.

Suggestions for further reading

- For a quick but more-detailed explanation of cryptography basics and SSL/TLS, see *Web Security, Privacy and Commerce, 2nd Edition*, written by Simson Garfinkel and published by O'Reilly Media Inc. in January 2002. (ISBN is 0596000456.)
- The reference manual for `keytool` is available at http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/keytool.html.
- "XML Encryption Syntax and Processing," a W3C Recommendation, is available at http://www.w3.org/TR/xmlenc-core/.
- Apache XML Security libraries can be downloaded from http://xml.apache.org/security/Java/index.html.
- Javadocs for `javax.xml.xpath` package in Sun JDK 1.5 can be found at http://java.sun.com/j2se/1.5.0/docs/api/javax/xml/xpath/package-summary.html.
- JSR-106, Java Specification Request for the standardization of APIs for XML Encryption, can be found at http://www.jcp.org/en/jsr/detail?id=106. A draft of the API is available for public review at http://jcp.org/aboutJava/communityprocess/pr/jsr106/.

SOA

SOA SECURITY

Ramarao Kanneganti
and Prasad Chodavarapu

"All the security your SOA needs."

—Patrick Steger
Software Architect & Security Engineer
Zühlke Engineering AG

To secure an SOA solution you cannot just secure the participating applications—you must also ensure security of the services that underlie the whole thing. But, how do you do that while keeping services open and easy to use by both internal and external applications? This is one of the many questions answered in this book.

SOA Security is written for architects, designers, developers, and IT managers. Part 1 covers the basics of SOA—best practices, toolkits, and techniques. You will find it useful, even if you are familiar with SOA. Part 2 deals with the building blocks of SOA security: authentication, authorization, nonrepudiation, and more. Part 3 shows how to build industrial-strength solutions, introducing you to real-life problems and frameworks to solve them. It is accessible to IT managers and particularly useful to architects.

This book is down-to-earth. It covers many kinds of security situations, from casual to complex. It combines concepts and practice. Its examples are in Java and the Apache Axis toolkit, but .NET and other users should have no trouble getting the message. And for those using commercial security solutions, it provides valuable theoretical background for the practical issues they encounter, including why and when they may need to implement a missing functionality.

Dr. Ramarao Kanneganti is CTO at HCL EAI Services. He has worked at Bell Labs in databases and large programming systems, and currently advises enterprise clients on SOA strategies. Rama works out of Grosse Pointe Woods, Michigan.

Prasad Chodavarapu is a manager at HCL EAI Services, Bangalore, India. He leads teams designing and deploying integration solutions at enterprises across the globe.

For more information, code samples, and to purchase an ebook
visit www.manning.com/SOASecurity

ISBN-13: 978-1932394689
ISBN-10: 1932394680
55999
9 781932 394689

MANNING

$59.99 / Can $59.99