CHAPTER 8

XML-Security

In the last few chapters, you have seen the extensive use of XML for data transport. When such data is transported over public networks, is it secured? Could the data be read or even tampered with by a malicious third party? Further, will the receiver of the data be assured about the authenticity of the sender and integrity of the data? This chapter gives answers to all these questions.

I will discuss the need for security and cover important security concepts. Cryptography plays an important role in securing your applications and data, and in this chapter you will study both symmetric- and asymmetric-key cryptography. You will learn to create digital signatures and certificates and use them in your applications. XML signature specifications allow you to sign documents partially, and you will also learn about these specifications. Finally, you will learn the techniques of encrypting and decrypting data for protecting its contents from prying eyes.

Why Security?

Throughout this book, we have been working on a stock brokerage case study. Our stock brokerage accepts trade orders from its customers by using a web interface. As the orders pass through a public network, it is conceivable that somebody could monitor the network traffic, and read and modify the order, before sending the modified order to the brokerage. The stock brokerage, unaware of the modifications to the order, might execute it on the stock exchange and cause no end of grief for the customer.

When the brokerage receives the customer's order, how does it know that the order did indeed originate from the appropriate individual? In the case of trade losses, the customer might refuse to confirm the order. Similarly, whenever the customer places an order or makes an online payment to the broker, is the customer sure he's really talking to the broker and not some malicious third party masquerading as the organization?

The concepts of secure data transmission and the authentication of sender and receiver are not restricted to stock exchange operations. Data transmissions need to be secure in several other situations such as banking transactions, e-commerce shopping, and government operations. Further, should that data be intercepted, we need to ensure that it cannot be read by an intruder.

Before we look into the answers to the preceding questions and the implementations of security in this digital world, let's look at some of the important terms used in security.

Important Terms in Security

The answers to the questions presented in the previous section lie in the definitions of several important security terms: authentication, authorization, nonrepudiation, message integrity, and message confidentiality. Each of these terms is defined in this section.

Authentication

Authentication is a process of determining whether someone is in fact whosoever he or she claims to be.

When the customer places an order with the stock brokerage, the brokerage needs to authenticate itself to the customer. The customer can then be sure of sending the order to the appropriate destination. Similarly, the customer may have to authenticate himself to the broker. This is called *mutual authentication*, whereby both parties confirm the identity of the other party involved in the conversation.

Authorization

Authorization is a process of granting an authenticated user the authority to access certain resources, based on the access rights of that user.

After a user is authenticated by a web application, he will be allowed to perform various tasks. The scope of these tasks is decided by the user's privileges as defined by the system administrator. This process of allowing the user to perform certain operations is called *authorization*.

Nonrepudiation

Nonrepudiation is a process followed by two parties involved in a communication that will prevent either user from denying the conversation in the future.

As mentioned earlier, a customer might deny having placed an order that resulted in trade losses. In such cases, the stock brokerage should be able to prove to the customer that it has a valid order from the customer in question, that the order did indeed originate from that customer, and that the order was not tampered with on its way. This process is called *nonrepudiation*, whereby the customer cannot deny having placed the order.

Message Integrity

Message integrity is a process by which the message receiver is guaranteed that the received message has not been tampered with on its way.

The stock brokerage needs to ensure that the various trade requests received from its customers have not been modified on their way. An intruder could intercept a message, modify its contents, and forward it to the ultimate receiver. The integrity of the message is guaranteed by generating message digests and creating digital signatures on digests. This is discussed in depth later in the chapter.

Message Confidentiality

Message confidentiality is a process that ensures both message sender and receiver that even if the message is intercepted by an intruder on its way, it would not make any tangible sense to the interceptor.

Whenever a customer sends a trade request to the stock brokerage, hiding the trade order details from an interceptor may not be required. However, when the customer sends credit card details to the brokerage, the content must be hidden from spying eyes. Message confidentiality is achieved by encrypting the message content.

Brief Introduction to Cryptography

Cryptography is the process of transforming data from one form to another in order to hide the message content from an interceptor and to prevent undetected modifications.

The original form of data is called *plaintext* in cryptography, and the transformed form is called *ciphertext*. The term *plaintext* refers to not only human-readable data, but also binary data such as images and database content. Ciphertext is the transformed human-unreadable data.

The cryptography process involves converting plaintext to ciphertext, and converting ciphertext back to its original form of plaintext. Transforming plaintext into ciphertext is known as *encryption*. This is illustrated in Figure 8-1.

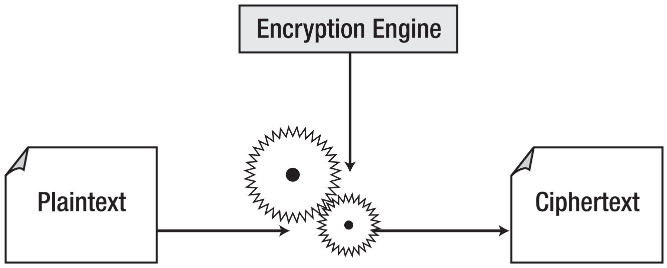

Figure 8-1. *Encryption transforms plaintext to ciphertext.*

Transforming ciphertext to plaintext is called *decryption*. This is illustrated in Figure 8-2.

Encryption ensures that the data is hidden from illegitimate users. But how does it prevent undetected modification of the data? In cryptography, we sign the data by creating a checksum on the data. This checksum may be generated by a simple algorithm, such as adding binary representations of each character in the message text. This checksum is called a *message digest*. At the receiver end, the receiver re-creates the checksum (message digest) by using the same algorithm as the sender. If the two digests match, the receiver can be sure that the data has not been tampered with on its way. It also ensures message authenticity—that the message did originate from the sender. This process is illustrated in Figure 8-3.

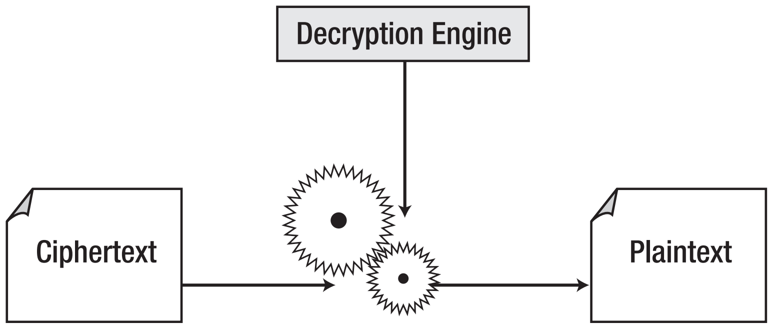

Figure 8-2. *Decryption transforms ciphertext to plaintext.*

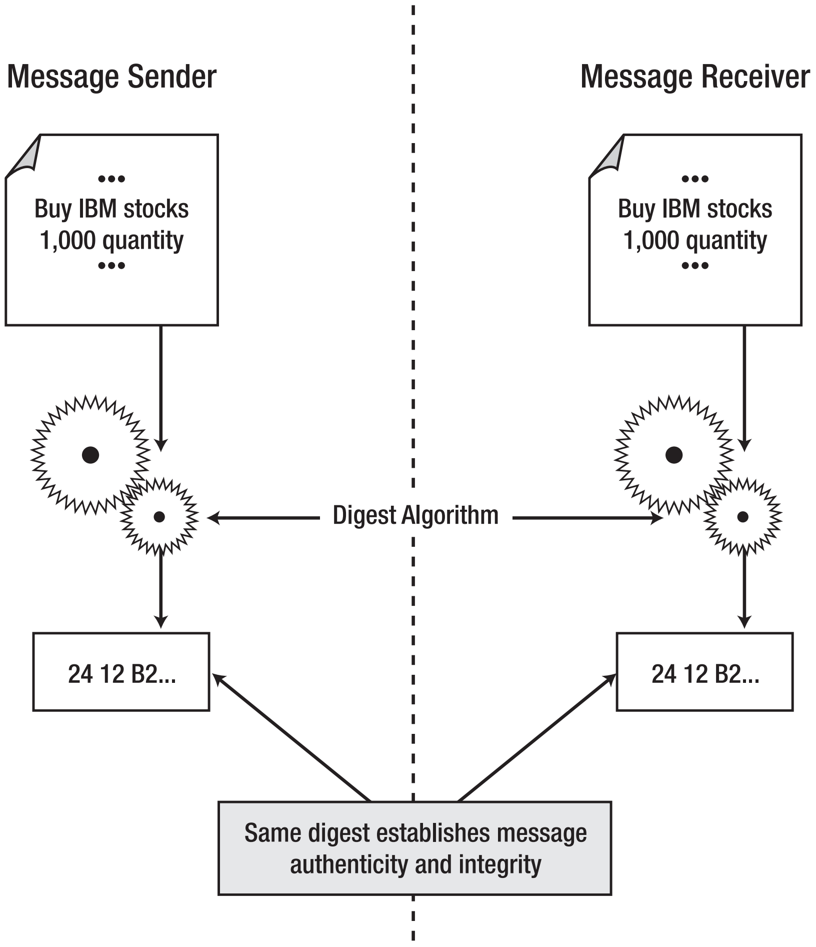

Figure 8-3. *Using a message digest for message authenticity and integrity checks*

In cryptography, we use a key for encryption and decryption. A *key* is a predefined character sequence. The encryption process that uses a key is shown in Figure 8-4.

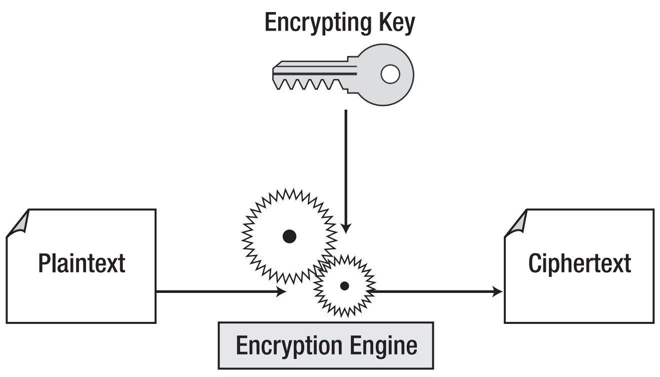

Figure 8-4. *Encryption algorithm that uses a key*

We can use one key or a matching pair of keys for encryption and decryption. We decrypt the encrypted data by using either the same key used for encryption or a matching key provided by the message sender. Because the one key or matching keys used for decryption are privately held by the two involved parties, the process ensures message confidentiality.

A key is also used for generating the message digest. The sender sends both the message and its generated digest to the intended receiver. The signing key or a matching key is shared with the receiver. The receiver uses the supplied key to generate a message digest on the received message. If the two message digests match, the message authenticity and integrity are established.

Thus, these cryptography techniques answer all the requirements of security:

- Hide information content
- Prevent undetected modification
- Prevent unauthorized use

Depending on whether a shared key or a matching pair of keys is used, two types of cryptography techniques are available: symmetric and asymmetric cryptography. You will learn about both in this section. You will also learn about the use of digital signatures and certificates.

Symmetric Cryptography

Symmetric cryptography uses the same key for both encryption and decryption, or for signing and verifying. (Its name reflects the symmetry of using the same key at both ends.) Figure 8-5 illustrates encryption and decryption using symmetric-key cryptography.

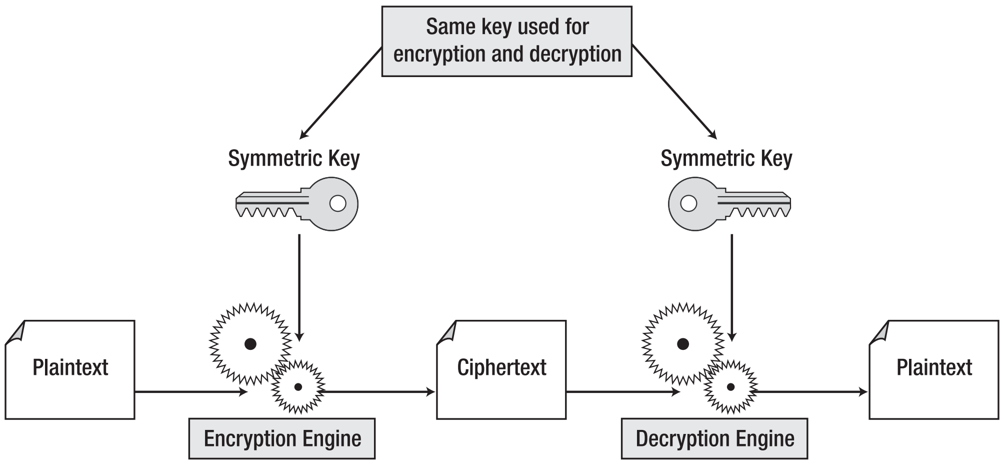

Figure 8-5. *Symmetric cryptography uses the same key for both encryption and decryption.*

Either party can generate the key for the conversation. This key should not be transmitted over unsecured channels such as the Internet and must be made available to the other party even before the conversation begins.

Symmetric cryptography has its roots deep into the early stages of computing. In World War II, symmetric cryptography was used extensively by many countries. The Germans confidently believed that messages sent via Enigma, an electromechanical rotor-based cipher system, could not be cracked. But the British cracked some of the Enigma codes early in the war and continued to decrypt important messages until the Allies' final victory.

Sharing the key with the legitimate user has posed the greatest challenge in symmetric-key cryptography and ultimately led to the invention of asymmetric cryptography. However, although asymmetric cryptography offers a solution to key sharing, it is slower than symmetric cryptography. Thus, in practice, and as you will see later in this chapter, we use a combination of the two techniques.

Asymmetric Cryptography

In *asymmetric cryptography*, the sender creates a pair of matching keys. It is assumed that the techniques used for creating the pair ensure that there is one and only one matching key. The two keys are called the *public key* and the *private key*. As its name indicates, the public key is truly public and is made available to everybody who asks for it. Likewise, the private key is totally private to the key generator and should never be lost or disclosed to anybody; doing so would compromise security.

It is also assumed in asymmetric cryptography that by knowing the public key, nobody can deduce its private key. This eliminates any risk that may be involved in making the key truly public.

The sender uses his private key to sign or encrypt the document. The receiver uses the matching public key obtained from the sender to verify the signature or to decrypt the *gibberish* (this is another term used in cryptography to refer to ciphertext) received from the sender. This process is illustrated in Figure 8-6.

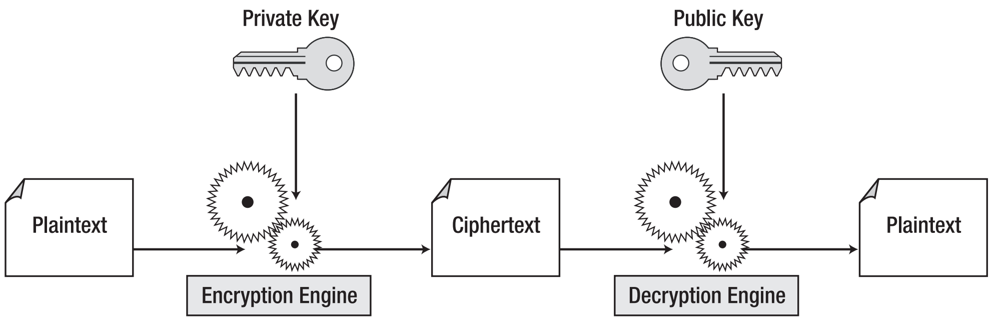

Figure 8-6. *Asymmetric cryptography uses different keys for encryption and decryption.*

Note If encryption is carried out with one of the keys of the key pair, decryption is possible *only* with the matching key of the same key pair. The encrypted message cannot be decrypted with the same key.

Digital Signatures

A *digital signature* is the equivalent of a paper signature in the digital world. Just as you sign paper documents to establish their authenticity, you sign digital documents by creating and adding a digital signature. Signatures on paper cannot be forged easily; also, any changes made to the content of a document on physical paper can be detected easily.

In digital documents, we create a digital signature by creating a checksum based on the document contents. In bookkeeping, you typically take a total of all debit transactions and another total of all credit transactions. These are the two checksums. When they match, you know that your accounts tally. Similarly, in the digital world, a checksum can be generated with a simple algorithm such as adding binary representations of each character in the document.[1] This checksum is also known as a *message digest*. In fact, we create another checksum based on the message digest, which is then called the digital signature.Thus, a digital signature is a checksum of a checksum (message digest).

We use a mix of both symmetric and asymmetric cryptography while generating digital signatures. The process of signing and verifying the documents is as follows:

1. A sender creates a message digest on a document by using a message digest engine. A message digest algorithm might be seeded with a symmetric key if message authenticity is desired. The sender then uses asymmetric-key cryptography to encrypt the message digest along with the symmetric key (also called a secret) used for creating the digest. That means that two values, namely the message digest and the symmetric key, are encrypted by using the private key of the public/private key pair. This encrypted data (signature), along with the original document, is sent to the receiver. This process is illustrated in Figure 8-7.

1. The checksum algorithms used in practice are much more complicated and thereby more secure than the one suggested here.

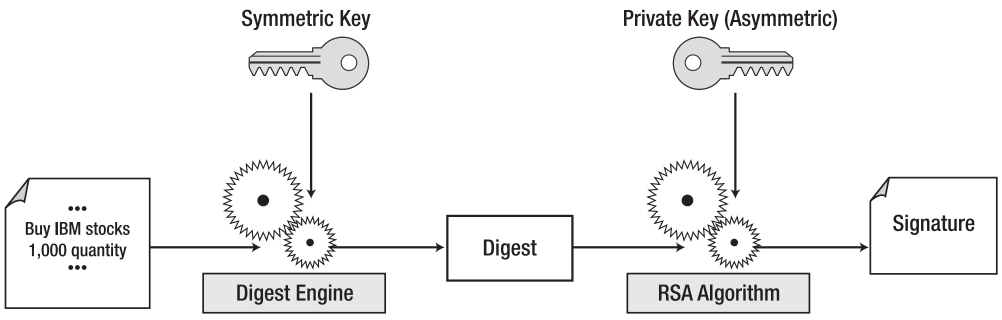

Figure 8-7. *Generating a digital signature*

2. After receiving the document along with its signature, the receiver uses the public key supplied by the sender to decrypt and thus obtain the message digest and the secret (the symmetric key used for encryption). The receiver re-creates the message digest by using the obtained secret. If the generated message digest matches the received message digest, the document origin is authenticated and the message integrity is assured. This process is illustrated in Figure 8-8.

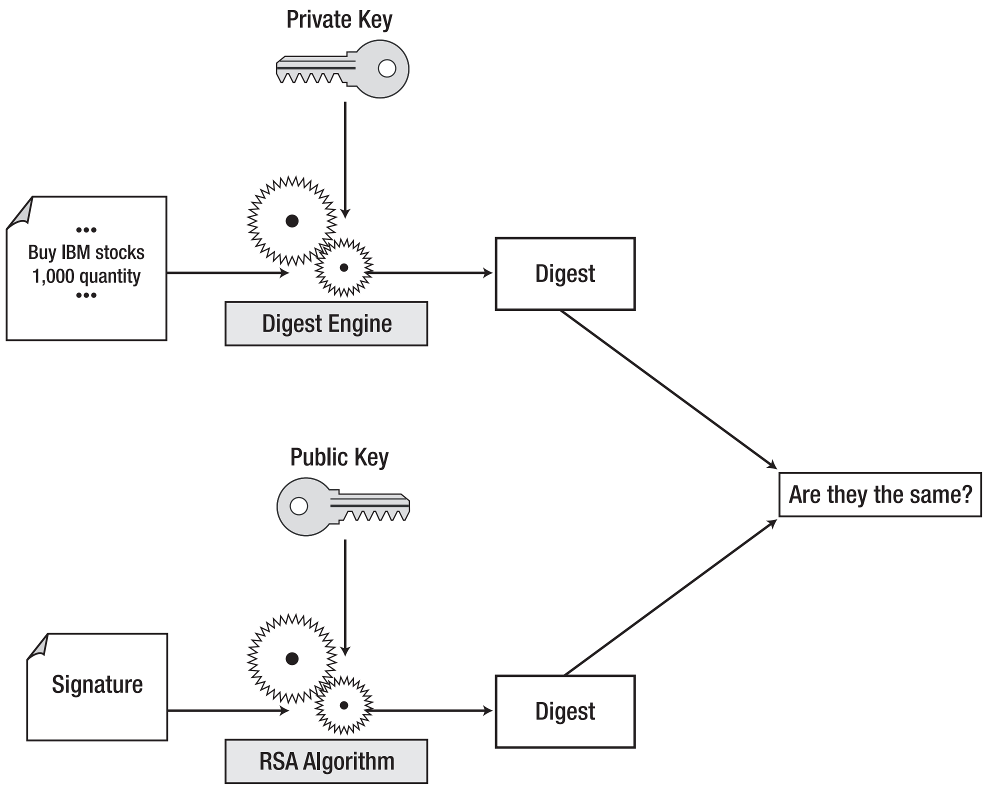

Figure 8-8. *Message verification using a public key*

This technique is ideally suited for large documents. For small documents, you can use asymmetric-key cryptography to sign the entire document.

Digital Certificates

There are many utilities available for generating a matching public/private key pair. Because public/private key pairs can be generated with ease, how can anyone be assured about the authenticity of the public key? It is possible for someone to create a public/private key pair and distribute the public key to everyone while claiming to be someone else. This is where digital certificates come into the picture.

After creating a public/private key pair, the creator should send the public key to a certification authority (CA) such as VeriSign.[2] After establishing the sender's identity, VeriSign creates a digital certificate for that sender. This certificate contains the sender's public key, name, and various other details. Itself a digital document, the certificate is signed by the CA by using its own private key. Figure 8-9 shows the structure of a digital certificate.

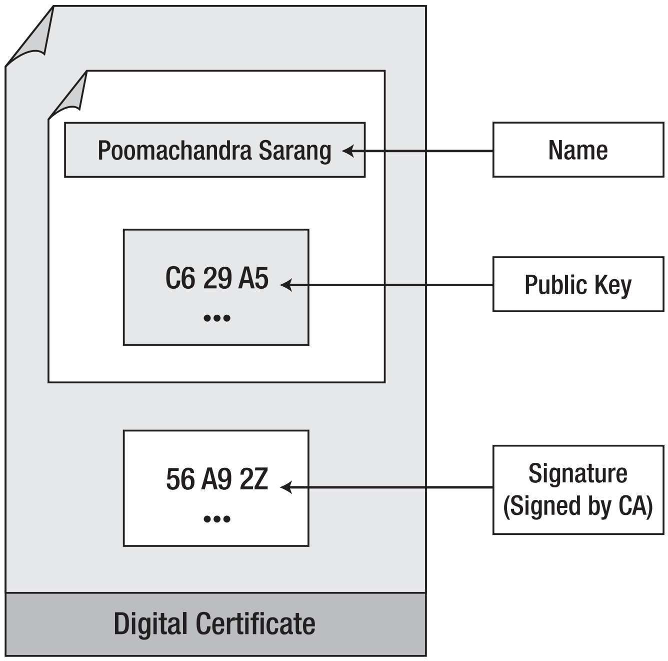

Figure 8-9. *A digital certificate*

The public key of the CA is available to the public, like any other public key. The document receiver now can verify the authenticity of the sender's public key by obtaining the certificate issued by the CA. In short, the CA vouches for the correctness of the sender's public key.

2. http://www.verisign.com

A widely used standard for defining digital certificates is X.509. This International Telecommunication Union (ITU) recommendation[3] is known as ITU-T X.509 (formerly CCITT[4] X.509) and as ISO/IEC/ITU 9594-8, which defines a standard certificate format for public-key certificates and certification validation.

Using XML Signatures

Digital signatures sign the entire document. XML signature is a specification that allows you to sign the document partially. It is similar to digital signatures.

XML signature technology embodies the concepts discussed in the previous sections. The only difference is that by using XML signatures, you can sign either the full document or only a specified portion of it. When using other digital signatures, you sign the entire document.

The XML signatures are not restricted to XML documents, but can be applied to any other type of digital data such as images or even HTML documents.

The XML signature technology was developed by the XML-DSig charter (`http://www.w3.org/1999/05/XML-DSig-charter-990521.html`). This technology was developed in response to the Electronic Signatures in Global and National Commerce Act (ESIGN), which in 2000 made digital signatures legally binding in the United States.

In this section, you will study the advantages of XML signatures over digital signatures. You will learn about the various types of XML signatures. You will also learn about the importance of canonical XML. Finally, you will study the full structure of XML signatures.

Advantages

XML signatures offer few advantages over digital signatures. An XML signature itself is written in XML and is thus somewhat user readable. Digital signatures are typically binary documents consisting of both printable and nonprintable characters, whereas XML signatures are always encoded so as to consist of only printable characters. Thus, the content of an XML signature, although it might not make sense to a human reader, is at least readable. Listing 8-1 illustrates a typical XML signature.

Listing 8-1. *A Typical XML Signature*

```
<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <ds:SignedInfo>
    <ds:CanonicalizationMethod Algorithm=
      "http://www.w3.org/TR/2001/REC-xml-c14n-20010315">
    </ds:CanonicalizationMethod>
    <ds:SignatureMethod Algorithm=
      "http://www.w3.org/2000/09/xmldsig#rsa-sha1">
    </ds:SignatureMethod>
```

3. http://www.itu.int/ITU-T/asn1/database/itu-t/x/x509/1997/

4. CCITT stands for Comite Consultatif Internationale de Telegraphie et Telephonie.

```
    <ds:Reference URI="">
      <ds:Transforms>
        <ds:Transform Algorithm=
          "http://www.w3.org/2000/09/xmldsig#enveloped-signature">
        </ds:Transform>
      </ds:Transforms>
      <ds:DigestMethod Algorithm=
        "http://www.w3.org/2000/09/xmldsig#sha1">
      </ds:DigestMethod>
      <ds:DigestValue>/EzketjAlFVxtuJG8Dg1bUYoKCE=</ds:DigestValue>
    </ds:Reference>
  </ds:SignedInfo>
  <ds:SignatureValue>
BHq3QeByVI7oAjLZOZsGDiCLjOLstBB2Z7O2jxPC88QGhQUKPxWhXOrGHDcNuS9mZYvbeO2HCnccTrC
Su1Uwys9v8GanL6akAMvdxO4tYTMwbNm+YXQgf3gBaWP/XXe6WibJzR2v2+aOIeWZnoR2gkHsIPnpL1
JeqiqYMxjMkR4=
  </ds:SignatureValue>
  <ds:KeyInfo>
    <ds:KeyValue>
      <ds:RSAKeyValue>
        <ds:Modulus>
ijqXBjmSWNbJmD7zZoauMRBYDh/1LeKhVljz/FCLhJofDQhXj+ZMY48/1J+KAGOtSa3U9UHOsAKeprX
24/tTkWiWTyIxQRHgCl/Z3B5fh/lylfvSN47WSPAgIZ6JpNfOxOa3XlVht/aCs+QbdKTk2qOs
          FlJjRu8N6Kw5pNa9slU=
        </ds:Modulus>
        <ds:Exponent>AQAB</ds:Exponent>
      </ds:RSAKeyValue>
    </ds:KeyValue>
  </ds:KeyInfo>
</ds:Signature>
```

As you can see in Listing 8-1, the signature consists entirely of only printable characters. A signature is an XML document. Because XML is text-based, so is the XML signature. The entire signature is enclosed in an opening and closing `Signature` tag. Unlike digital signatures, XML signatures include the name of the algorithm that was used to generate the signature. Look at the `SignatureMethod` and `DigestMethod` elements in Listing 8-1 for the algorithm they use. For digital signatures, there has to be a prior agreement between the sender and the receiver on the algorithms they are going to use for signing and creating message digests. Also, look at the `DigestValue` and `SignatureValue` element contents. They contain only printable characters.

Another major difference between digital signatures and XML signatures is that XML signatures can be used to selectively sign a part of the document. This offers an advantage of not breaking the original signature when the document is processed by multiple parties on its way to its final destination. The customer purchase order for our stock brokerage may pass through multiple stages of approval to its final destination. If the intermediaries modify the document, the original signature will break. However, with the XML signature's ability to sign only a partial document, the original signature and all the intermediary signatures can be retained to their final destination.

Signature Types

XML signatures are classified into three types, depending on their association with the document that is signed:

- *Enveloped*: The XML signature is embedded in the document that is being signed. That means that the `Signature` element will appear as one of the child elements in the original XML document.
- *Enveloping*: The entire document that is to be signed appears as one of the child elements of the `Signature` element.
- *Detached*: The XML signature appears in a document that is separate from the signed document. The `Signature` stores the reference to the signed document. This type of signature is useful for signing non-XML documents.

Canonical XML

You have seen in Listing 8-1 that the `Signature` contains a digest value specified by the `DigestValue` element. This digest is very sensitive to changes in the document. As a matter of fact, even the slightest change such as adding one white-space character to the document will change the digest drastically. As you might imagine, a digest algorithm should not generate a digest that is similar to the old one whenever the original document undergoes even a very minor change.

Unfortunately, different XML applications can generate XML documents that contain identical content but potentially have slight variations such as the amount of white space found within it. Also, there are no strict guidelines on how the entire document is structured as long as it is well formed. Thus, for each document, the generated digest will vary drastically and the signature verification on a signed document could fail due to minor differences in the two versions of the same logical document. To overcome this limitation, canonical XML was introduced.

Canonical XML is a normalized representation of any physical XML document. It is a standard for signature processing. Before a document is signed, it is transformed into a canonical XML document. Similarly, before the signature is verified, the received document is transformed again to canonical XML to regenerate the digest for verification.

XML Signature Structure

Listing 8-2 shows the general structure of an XML `Signature` element.

Listing 8-2. *XML Signature Structure*

```
<Signature ID="">
  <SignedInfo>
    <CanonicalizationMethod Algorithm=""/>
    <SignatureMethod Algorithm=""/>
    <Reference URI="">
      <DigestMethod Algorithm=""/>
      <DigestValue> ... </DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue> ... </SignatureValue>
  <KeyInfo> ... </KeyInfo>
</Signature>
```

As seen in Listing 8-2, the `Signature` contains child elements called `CanonicalizationMethod` and `SignatureMethod`. This is where you would specify the algorithms used for creating canonical XML and for signing the document, respectively. Similarly, the `DigestMethod` element allows you to specify the digest algorithm in its attribute value (`Algorithm`). The `Signature` element also contains elements such as `DigestValue`, `SignatureValue`, and `KeyInfo`, which contain the information that their names suggest.

Downloading and Installing Software

The Apache XML-Security project is available in source and binary distribution for both Java and C++ libraries. If you use C++ for development, you will need to download C++ libraries. I will use Java libraries in this book. You can download Java libraries from the following URL:

```
http://xml.apache.org/security/dist/java-library/
```

The download site contains both binary and source distributions:

- `xml-security-bin-1_3_0.zip` (binary distribution)
- `xml-security-src-1_3_0.zip` (source distribution)

You can download the source distribution and build it by using Apache Ant[5] to create the libraries. Ant is a Java-based build tool that is similar to other build tools such as make, gnumake, nmake, and others.

Alternatively, if you download the binary distribution, which is much larger than the source distribution, simply unzip the file to a desired folder for installing the software. After unzipping the binary, include the various JAR files from the `libs` folder of your installation in your classpath and you are ready to go.

Signing XML Documents

In this section, we will start our programming exercises. Imagine that a customer wants to place a trade order with our stock brokerage. Because the brokerage needs to authenticate the customer, the customer must sign the purchase order (PO). The brokerage would verify the purchase order before executing it.

To sign the PO, the customer generates a public and private key for his own use and then signs the PO by using the generated private key. The customer sends the original document along with the signature and the public key to the brokerage. The brokerage retrieves the public key from the received document and then verifies the authenticity and the integrity of the

5. http://ant.apache.org/

document. The brokerage performs this verification by using the algorithms specified in the `Signature` element of the received document to re-create and match the re-created message digest and the signature with those obtained as a part of the `Signature` element.

Note that this technique has one fallacy: the brokerage cannot be sure about the authenticity of the customer's public key itself. To authenticate the public key, the brokerage needs a digital certificate issued by a CA. We will use digital certificates in our future examples. To keep matters simple in this example, we will generate a public/private key pair dynamically and use it while signing and verifying the document.

Generating and Signing an XML Document

Listing 8-3 illustrates how to sign a dynamically created XML document.

Listing 8-3. *Signing Dynamically Generated XML Document (*`Ch08\src\SignedPO.java`*)*

```
/*
 * SignedPO.java
 */
package apress.ApacheXML.ch08;

// Import required classes
import java.io.File;
import java.io.FileOutputStream;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import org.apache.xml.security.algorithms.MessageDigestAlgorithm;
import org.apache.xml.security.signature.XMLSignature;
import org.apache.xml.security.transforms.Transforms;
import org.apache.xml.security.utils.XMLUtils;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import javax.xml.parsers.DocumentBuilder;

public class SignedPO {

  static Document createDocument() throws Exception {
    // Obtain an instance of Docuement Builder Factory
    javax.xml.parsers.DocumentBuilderFactory dbf =
        javax.xml.parsers.DocumentBuilderFactory.newInstance();
    dbf.setNamespaceAware(true);
    javax.xml.parsers.DocumentBuilder db = dbf.newDocumentBuilder();

    // Create a new document
    Document doc = db.newDocument();
    // Create elements
    Element root = doc.createElementNS(null, "PurchaseOrder");
    Element contents = doc.createElementNS(null, "signedContents");

    doc.appendChild(root);
    root.appendChild(contents);
    contents.appendChild(doc.createTextNode(
            "\nWe request that you EXECUTE the following trades\n"));

    // Add Trade details
    Element stock1 = doc.createElementNS(null, "stock");
    contents.appendChild(stock1);
    stock1.appendChild(doc.createTextNode("GFW"));
    Element quantity1 = doc.createElementNS(null, "quantity");
    contents.appendChild(quantity1);
    quantity1.appendChild(doc.createTextNode("50"));
    Element price1 = doc.createElementNS(null, "price");
    contents.appendChild(price1);
    price1.appendChild(doc.createTextNode("25.35"));
    Element type1 = doc.createElementNS(null, "type");
    contents.appendChild(type1);
    type1.appendChild(doc.createTextNode("B"));

    // Add one more Trade
    // ...

    return doc;
  }

  public static void main(String unused[]) throws Exception {

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

    // Generate a public/private key pair for temporary use
    KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
    KeyPair keyPair = kpg.generateKeyPair();

    // Obtain reference to generated public/private keys
    PrivateKey privateKey = keyPair.getPrivate();
    PublicKey pubkey = keyPair.getPublic();

    // Create a po document
    Document doc = createDocument();

    // Obtain the root element
    Element root = doc.getDocumentElement();
    // Create file for writing output
    File f = new File("po.xml");
    // Create a XMLSignature instance that uses RSA_SHA1 algorithm
    XMLSignature signature = new XMLSignature(doc, f.toURL().toString(),
            XMLSignature.ALGO_ID_SIGNATURE_RSA_SHA1);

    // Create canonical XML
    Transforms transforms = new Transforms(doc);
    transforms.addTransform(Transforms.TRANSFORM_ENVELOPED_SIGNATURE);

    // Add canonicalized document to signature
    signature.addDocument("", transforms,
            MessageDigestAlgorithm.ALGO_ID_DIGEST_SHA1);

    // Add the public key information to signature
    signature.addKeyInfo(pubkey);

    // Add signature itself to the PO document
    root.appendChild(signature.getElement());

    // Sign the document
    signature.sign(privateKey);

    // Create an output stream
    FileOutputStream fos = new FileOutputStream(f);
    // Output the memory document using XMLUtils.
    XMLUtils.outputDOMc14nWithComments(doc, fos);
  }
}
```

We have to initialize the security libraries before invoking any of the Apache XML-Security library services. The `main` function first initializes the security libraries with the following code:

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

We then generate a public/private key pair for the current session:

```
// Generate a public/private key pair for temporary use
KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
KeyPair keyPair = kpg.generateKeyPair();
```

We are using RSA algorithms for key generation.[6] This is the most popular implementation of the Public Key Infrastructure (PKI). The other well-known implementations are Diffie-Hellman (DH)[7] and Elliptic Curve Diffie-Hellman (ECDH).[8]

6. The RSA algorithm was invented by Ron Rivest, Adi Shamir, and Len Adleman. The name RSA was coined after the initials of the inventors.
7. Refer to http://www.rsasecurity.com/rsalabs/node.asp?id=2126 for details.
8. Refer to http://www.rsasecurity.com/rsalabs/node.asp?id=2013 for details.

We retrieve the public and private keys from the generated key pair:

```
// Obtain reference to generated public/private keys
PrivateKey privateKey = keyPair.getPrivate();
PublicKey pubkey = keyPair.getPublic();
```

We then create a purchase order XML document by calling the `createDocument` method. The `createDocument` method uses the DOM API discussed in Chapter 2 to generate the PO document dynamically. It returns the reference to the `Document` object to the caller. We will add a `Signature` element to this document.

We construct the `Signature` element by using the following statement:

```
// Create a XMLSignature instance that uses RSA_SHA1 algorithm
XMLSignature signature = new XMLSignature(doc, f.toURL().toString(),
        XMLSignature.ALGO_ID_SIGNATURE_RSA_SHA1);
```

Note that we have specified the algorithm to be used for signature generation as one of the parameters. The `XMLSignature` class defines several IDs for the use of different algorithms. For example, you can use the `ALGO_ID_SIGNATURE_RSA_SHA256` ID to create a 256-bit signature or you can use `ALGO_ID_SIGNATURE_RSA_SHA512` to generate a 512-bit signature.

■ **Note** The `ALGO_ID_SIGNATURE_RSA_SHA1` generates a 172-bit signature and not a 1-bit signature as you might conclude from the previous statement.

■ **Tip** Try these different algorithms to see their effect on the generated signature value in the signed document.

The first parameter to the `XMLSignature` constructor is the `Document` object that is to be signed, and the second argument is a reference to the output file where the signed document will later be serialized.

After constructing the `Signature`, the program transforms the input document to canonical XML:

```
// Create canonical XML
Transforms transforms = new Transforms(doc);
transforms.addTransform(Transforms.TRANSFORM_ENVELOPED_SIGNATURE);
```

We then add the document to the `Signature` element:

```
// Add canonicalized document to signature
signature.addDocument("", transforms,
        MessageDigestAlgorithm.ALGO_ID_DIGEST_SHA1);
```

We specify the digest algorithm to be used while adding the document.

■ **Tip** I again urge you to try the different digest algorithms to see their effect on the digest value.

We add the key info to the `signature` so that the receiver obtains the customer's public key as a part of the document:

```
// Add the key information to signature
signature.addKeyInfo(pubkey);
```

The `signature` node is then appended to the `root` of the document:

```
// Add signature itself to the PO document
root.appendChild(signature.getElement());
```

After this, the document itself is signed by calling the `sign` method of the `XMLSignature` class:

```
// Sign the document
signature.sign(privateKey);
```

During execution of this `sign` method, the signature is generated and inserted into the `Signature` element of the document. Our document is now signed. We need to serialize the document constructed in memory to a physical file. We do this by creating an output stream on the desired physical file and by calling the `outputDOMc14nWithComments` method of the `XMLUtils` class to perform the serialization:

```
// Create an output stream
FileOutputStream fos = new FileOutputStream(f);
// Output the memory document using XMLUtils.
XMLUtils.outputDOMc14nWithComments(doc, fos);
```

Running the Application

Compile the application by using the following command line:

```
C:\<working folder>\ch08\src>javac -d . SignedPO.java
```

Run the application by using the following command line:

```
C:\<working folder>\ch08\src>java apress.ApacheXML.ch08.SignedPO
```

On a successful run, you will find a `po.xml` file created in your working folder.

Examining the Signed Document

If you open the po.xml file in your browser or your console, you will see the following output:

```
<PurchaseOrder>
  <signedContents>
    We request that you EXECUTE the following trades
    <stock>GFW</stock>
    <quantity>50</quantity>
    <price>25.35</price>
    <type>B</type>
    <stock>ABNPRF</stock>
    <quantity>100</quantity>
    <price>24.83</price>
    <type>S</type>
  </signedContents>
  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo>
      <ds:CanonicalizationMethod Algorithm=
          "http://www.w3.org/TR/2001/REC-xml-c14n-20010315">
      </ds:CanonicalizationMethod>
      <ds:SignatureMethod Algorithm=
          "http://www.w3.org/2001/04/xmldsig-more#rsa-ha256">
      </ds:SignatureMethod>
      <ds:Reference URI="">
        <ds:Transforms>
          <ds:Transform Algorithm=
              "http://www.w3.org/2000/09/xmldsig#enveloped-signature">
          </ds:Transform>
        </ds:Transforms>
        <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256">
        </ds:DigestMethod>
        <ds:DigestValue>
          nkXWyyMrZ5WEfpXHUuWNqDUnF2xkmW5WkwoIkSn2nFk=
        </ds:DigestValue>
      </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>
Fzi6hsoz3bkO2Bd2GfumTKzwqu5+i14Mfw1MW3Vd8u4bTyK3K5bJqmV7mz1DibcysdbSa3bxycVdCR9
XvuWC5XtRs9xABrU7+eNRK/IeRG1dsrQIIYgZ4XN97pLVU/iu5BJDERop/5CxqwnTjRWlvLselwdKnp
PqmGaFy6mrBks=
    </ds:SignatureValue>
    <ds:KeyInfo>
      <ds:KeyValue>
        <ds:RSAKeyValue>
          <ds:Modulus>
```

```
fL1AftPy9xQo6kEmE1Pw1Swe6HeCkNKWYdP7OfkFZu8PDJ43RzgflV5VYipD1u8J5YGqJg71XDIzMOQ
2IPcWU6NbCkcKrzeM4CMlw4d7Z8lQPPeIdQhVo/6+jCtqPemtwdpSZmiyfZhJyf/bXfezgBtPKM/5MP
KdepeUjOKVbhO=
          </ds:Modulus>
          <ds:Exponent>AQAB</ds:Exponent>
        </ds:RSAKeyValue>
      </ds:KeyValue>
    </ds:KeyInfo>
  </ds:Signature>
</PurchaseOrder>
```

The root element of the generated document is PurchaseOrder. The PurchaseOrder element contains a child element called signedContents, which in turn encapsulates the two trade orders. The element of interest to us is the ds:Signature element. This contains subelements that describe the algorithms used for canonicalization, signing, and creating the message digest. It also contains elements that describe the value for the message digest and the signature. The document also contains the key information specified by the KeyInfo element. This contains the public key of the signing customer. We will retrieve this key in our next application to verify the document.

Verifying the Purchase Order

In the previous section, you learned how the customer generates a purchase order dynamically, signs it by using its own private key, and sends the signed document to the stock brokerage. The brokerage now needs to verify the document and its integrity. In this section, we will develop an application that retrieves the public key of the customer from the received document and uses it to verify the document.

Developing an Application for Verifying the Document

Listing 8-4 illustrates how to verify a given signed document.

Listing 8-4. *Program to Verify a Signed Document* (Ch08\src\VerifyPO.java)

```
/*
 * VerifyPO.java
 */

package apress.ApacheXML.ch08;

// Import required classes
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.security.PublicKey;
import org.apache.xml.security.keys.KeyInfo;
import org.apache.xml.security.samples.utils.resolver.OfflineResolver;
import org.apache.xml.security.signature.XMLSignature;
import org.apache.xml.security.utils.Constants;
import org.apache.xml.security.utils.XMLUtils;
import org.apache.xpath.XPathAPI;
import org.w3c.dom.Element;

public class VerifyPO {

public static void main(String unused[]) {
  // Initialize security
  org.apache.xml.security.Init.init();

  // Obtain a builder factory instance
  javax.xml.parsers.DocumentBuilderFactory dbf =
          javax.xml.parsers.DocumentBuilderFactory.newInstance();
  dbf.setNamespaceAware(true);
  dbf.setAttribute("http://xml.org/sax/features/namespaces", Boolean.TRUE);

  try {
    // Open the file to be verified
    File f = new File("po.xml");
    System.out.println("Trying to verify " + f.toURL().toString());

    // Create a document builder
    javax.xml.parsers.DocumentBuilder db = dbf.newDocumentBuilder();
    db.setErrorHandler(
            new org.apache.xml.security.utils.IgnoreAllErrorHandler());

    // parse the input document
    org.w3c.dom.Document doc = db.parse(new java.io.FileInputStream(f));

    // Look for the Signature element in the required namespace
    Element nscontext = XMLUtils.createDSctx(doc, "ds",
          Constants.SignatureSpecNS);
    Element sigElement = (Element) XPathAPI.selectSingleNode(doc,
          "//ds:Signature[1]", nscontext);

    // Create signature element
    XMLSignature signature = new XMLSignature(sigElement,
          f.toURL().toString());

    // Add a resource resolver to enable the retrieval of resources
    signature.addResourceResolver(new OfflineResolver());

    // Retrieve the key information
    KeyInfo ki = signature.getKeyInfo();
    if (ki != null) {
      // Retrieve the public key from key information
      PublicKey pk = signature.getKeyInfo().getPublicKey();
      if (pk != null) {
        boolean result = signature.checkSignatureValue(pk);
        String str = null;
        if (result)
          str = "The document " + f.toURL().toString() + " is valid!";
        else
          str = "The document " + f.toURL().toString() + " is invalid!";
        System.out.println(str);
      } else {
        System.out.println("No public key found for document verification");
            }
      } else {
        System.out.println("Missing KeyInfo");
      }
    } catch (Exception ex) {
        ex.printStackTrace();
    }
  }
}
```

The main function first initializes the security libraries and like the earlier program obtains the document builder factory instance, creates a document builder, and parses the specified XML document by using DOM APIs.

After the document is parsed, the Signature element is located by using the XPath APIs:

```
// Look for the Signature element in the required namespace
Element nscontext = XMLUtils.createDSctx(doc, "ds",
      Constants.SignatureSpecNS);
Element sigElement = (Element) XPathAPI.selectSingleNode(doc,
      "//ds:Signature[1]", nscontext);
```

We construct the Signature element by using the located instance of the Signature in the input document:

```
// Create signature element
XMLSignature signature = new XMLSignature(sigElement,
      f.toURL().toString());
```

We now add a resource resolver to the constructed signature to retrieve the resources:

```
// Add a resource resolver to enable the retrieval of resources
signature.addResourceResolver(new OfflineResolver());
```

The key information stored in the input document is retrieved by calling the getKeyInfo method of the XMLSignature class:

```
// Retrieve the key information
KeyInfo ki = signature.getKeyInfo();
```

After the key information is retrieved, we retrieve a public key from it. Note that in our case, the sender has sent his public key as part of the document:

```
// Retrieve the public key from key information
PublicKey pk = signature.getKeyInfo().getPublicKey();
```

Using the public key, the document is verified:

```
    boolean result = signature.checkSignatureValue(pk);
```

The `checkSignatureValue` method returns a `boolean` value indicating the success or failure of the document verification. Note that the `Signature` element itself contains the digest value, signature value, the algorithms used, and so on. The `checkSignatureValue` method uses this information to validate the document.

If there are errors, the program prints the appropriate exception messages at the appropriate points in the program.

Running the Application

Compile the code by using the following command line:

```
C:\<working folder>\ch08\src>javac -d . VerifyPO.java
```

Run the application by using the following command line:

```
C:\<working folder>\ch08\src>java apress.ApacheXML.ch08.VerifyPO
```

When you run the program, you will see output similar to the following:

```
Trying to verify file:/C:/apress/ch08/src/po.xml
Feb 1, 2006 10:35:30 AM org.apache.xml.security.signature.Reference verify
INFO: Verification successful for URI ""
The document file:/C:/apress/ch08/src/po.xml is valid!
```

If the verification fails, you will get an appropriate failure message.

■Tip Make a minor modification to `po.xml` (for example, adding a white-space character) and save it. Now rerun the application to verify that the document verification fails this time.

Using Digital Certificates

The example in the previous section used a dynamically generated key pair. This key pair is valid only during the current session, when the application is running. Thus, there is nobody to vouch for the public key of the sender. In fact, the sender himself will not remember this public key unless he stores it to secondary storage in the same session as when the key is created. In practice, each sender who wants to sign the documents must send the public key to a CA and obtain a digital certificate that authenticates the public key.

In this section, I will describe the entire procedure for using digital certificates in your application. The procedure consists of the following steps:

1. Creating a public/private key
2. Exporting the public key to a physical file
3. Requesting a certificate from a CA
4. Importing the CA's certificate into your database

To complete these steps, we will use the Keytool utility provided as a part of your JDK installation.

Creating a Key Pair

We will use the Keytool utility to create a key pair. The utility stores the generated keys in a local database. This is a flat file known as `keystore`. You can give any name and extension to this file. We will call our keystore file `keystore.jks` and create it in the root folder of your current drive.

To generate a key pair for yourself, go to the `C:\` root folder (assuming you are running Windows) and run the Keytool utility at the command prompt with the `-genkey` option. You will also need to specify the alias name for your key and the path to the keystore file. When you run this utility, you will be asked a series of questions to gather information about the key signer, that is, you. Note that you will later send this information to a CA for ultimately including it in a digital certificate. The following transcript of the Keytool session was output when I ran it on my machine:

```
C:\>keytool -genkey -v -alias XMLBook -keystore c:\\keystore.jks
Enter keystore password:  sanjay
What is your first and last name?
  [Unknown]:  Poornachandra Sarang
What is the name of your organizational unit?
  [Unknown]:  Authoring
What is the name of your organization?
  [Unknown]:  ABCOM
What is the name of your City or Locality?
  [Unknown]:  Mumbai
What is the name of your State or Province?
  [Unknown]:  Maharashtra
What is the two-letter country code for this unit?
  [Unknown]:  IN
Is CN=Poornachandra Sarang, OU=Authoring, O=ABCOM, L=Mumbai, ST=Maharashtra,
C=I N correct?
  [no]:  y
```

```
Generating 1,024 bit DSA key pair and self-signed certificate (SHA1WithDSA)
        for: CN=Poornachandra Sarang, OU=Authoring, O=ABCOM, L=Mumbai,
             ST=Maharashtra, C=IN
Enter key password for <XMLBook>
        (RETURN if same as keystore password):  sanjay
[Storing c:\\keystore.jks]
```

You can input the information specific to you while generating the key. We will need the alias name to identify the key to be used while signing the document.

Listing Keys in Your Keystore

You can make a list of the keys in your keystore anytime by using the `-list` option on the Keytool utility. If you have created the keystore and added an `XMLBook` key entry to it as described in the previous section, you will get output similar to the following when you run Keytool with the `-list` option:

```
C:\>keytool -list -keystore c:\keystore.jks
Enter keystore password:  sanjay

Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry

xmlbook, Dec 28, 2005, keyEntry,
Certificate fingerprint (MD5): 36:79:EB:27:66:17:17:8B:47:79:FC:AB:FF:E4:5D:08
```

Note that every time you open the keystore, you will be asked to enter the password.

Exporting the Certificate

After creating a public and private key pair for your use, you can send this pair to a CA for obtaining a CA's certificate that authenticates the validity of your public key. To do this, you export your public key to a physical file and send that to a CA. To export the public key, you use the `-export` option on the Keytool utility. The transcript of the session run on my machine is as follows:

```
C:\>keytool -export -alias XMLBook -file xmlbook.cer -keystore c:\keystore.jks
Enter keystore password:  sanjay
Certificate stored in file <xmlbook.cer>
```

Note that you need to specify the alias for which you want to extract and export the public key. The certificate containing your public key is stored in the file specified by the `-file` option, and the `-keystore` option specifies the path to your keystore.

■Note You can create more than one keystore for use on the same machine.

Requesting a Certificate from a CA

After your public key is exported to a physical file, you send this to a CA for verification. In this step, you do not have to do anything except send your exported certificate to a CA and wait for its verification. After the CA is satisfied with your credentials, it will issue you another certificate (of course, they may charge some money for it). This certificate contains your public key and your company's information. It also contains information about the CA who issues the certificate. The certificate itself is signed using the private key of the CA.

We will now request a test certificate from a CA for the `XMLBook` alias that we created in the previous section. Export the `XMLBook` key to a text file and save that file as `xml.txt` by using the following command:

```
C:\>keytool -certreq -keystore c:\keystore.jks -file xml.txt -alias xmlcer
```

Thawte, a certification authority, provides free Secure Sockets Layer (SSL) digital certificates for testing purposes. You can get a test certificate from their website: `http://www.thawte.com/ssl-digital-certificates/ssl123/index.html`. Click on Free SSL Trial and follow the instructions. You have to paste the contents of the `xml.txt` file in the space provided for a certificate signing request. After submitting this request, your key will be processed and a certificate will be displayed on the next page. Copy this content and save it as `xml.cert`. In the next step, you will import this certificate in your keystore database.

Importing Certificates

After you receive a certificate from the CA, you can distribute it along with your signed documents to prove your identity. Just as you distribute your certificate, you can also receive certificates from other senders who want to prove their identities to you. You can import such certificates into your keystore database for your future use. To import a certificate, you use the `-import` option on the Keytool utility as shown in the following command:

```
C:\>keytool -import -file xml.cert -alias xmlcert -trustcacerts -keystore ➥
 c:\keystore.jks
```

The following is a transcript of importing a certificate into my database:

```
Enter keystore password:  sanjay
Owner: CN=Poornachandra Sarang, OU=Authoring, O=ABCOM, L=Mumbai,
         ST=Maharashtra, C=IN
Issuer: CN=Thawte Test CA Root, OU=TEST TEST TEST, O=Thawte Certification,
         ST=FOR TESTING PURPOSES ONLY, C=ZA
Serial number: d2d9c50eb3e45f46
Valid from: Thu Feb 02 18:32:17 GMT+05:30 2006 until: Thu Feb 23 18:32:17 GMT+0
         5:30 2006
Certificate fingerprints:
         MD5:  89:50:B8:1B:11:DF:B6:47:DA:97:06:72:DC:F2:0F:C5
         SHA1: C1:3F:8E:4F:E6:2C:5E:CE:62:34:25:5D:0D:8E:9F:DF:5F:0C:9F:45
Trust this certificate? [no]:  y
Certificate was added to keystore
```

If you list the certificates in your database, you will find two entries: one for `XMLBook` that you created earlier and one that you imported (alias `xmlcert`) in this step.

Now that we are ready with our certificates and keystore, let's look at how to use these certificates for signing and verifying our documents.

Using Digital Certificates for Document Signing

In this section, you will learn how to sign a document and attach a digital certificate to a signed document for establishing your identity to the client. We will use the same dynamically created PO document as in the earlier example in Listing 8-3. Listing 8-5 illustrates how to sign the document and attach a digital certificate to the signed document.

Listing 8-5. *Signing and Attaching a Digital Certificate* (`Ch08\src\CertifiedPO.java`)

```
/*
 * CertifiedPO.java
 */
package apress.ApacheXML.ch08;

// Import Required Classes

public class CertifiedPO {

  static Document createDocument() throws Exception {
        // Same as in Listing 8-3 - SignedPO.java
    return doc;
  }

  public static void main(String unused[]) throws Exception {
```

```
org.apache.xml.security.Init.init();
Constants.setSignatureSpecNSprefix("");

String keystoreType = "JKS";
String keystoreFile = "c:\\keystore.jks";
String keystorePass = "sanjay";
String privateKeyAlias = "XMLBook";
String privateKeyPass = "sanjay";
String certificateAlias = "XMLBook";

KeyStore ks = KeyStore.getInstance(keystoreType);
FileInputStream fis = new FileInputStream(keystoreFile);

//load the keystore
ks.load(fis, keystorePass.toCharArray());

//get the private key for signing.
PrivateKey privateKey = (PrivateKey) ks.getKey(privateKeyAlias,
        privateKeyPass.toCharArray());

// Create a PO document
Document doc = createDocument();

// Obtain the root element
Element root = doc.getDocumentElement();

// Create file for writing output
File f = new File("PO-certified.xml");

// Create a XMLSignature instance that uses RSA_SHA1 algorithm
XMLSignature signature = new XMLSignature(doc, f.toURL().toString(),
        XMLSignature.ALGO_ID_SIGNATURE_DSA);

// Create canonical XML
Transforms transforms = new Transforms(doc);
transforms.addTransform(Transforms.TRANSFORM_ENVELOPED_SIGNATURE);

// Create canonicalized document to signature
 signature.addDocument("", transforms,
         MessageDigestAlgorithm.ALGO_ID_DIGEST_SHA1);

//Add in the KeyInfo for the certificate that we used the private key of
X509Certificate cert =
        (X509Certificate) ks.getCertificate(certificateAlias);
```

```
    // Add the information to signature
    signature.addKeyInfo(cert);
    signature.addKeyInfo(cert.getPublicKey());

    // Add signature itself to the PO document
    root.appendChild(signature.getElement());

    // Sign the document
    signature.sign(privateKey);

    // Create an output stream
    FileOutputStream fos = new FileOutputStream(f);

    // Output the memory document using XMLUtils.
    XMLUtils.outputDOMc14nWithComments(doc, fos);
  }
}
```

After initializing the security system, the main function creates a few variables that specify the details of our keystore and the key that we intend to use for signing the document:

```
String keystoreType = "JKS";
String keystoreFile = "c:\\keystore.jks";
String keystorePass = "sanjay";
String privateKeyAlias = "XMLBook";
String privateKeyPass = "sanjay";
String certificateAlias = "XMLBook";
```

The getInstance method of the KeyStore class obtains a reference to the KeyStore:

```
KeyStore ks = KeyStore.getInstance(keystoreType);
```

We open the keystore file by creating an input stream on it and then initialize the KeyStore instance with the data read from the file:

```
FileInputStream fis = new FileInputStream(keystoreFile);
//load the keystore
ks.load(fis, keystorePass.toCharArray());
```

For signing, we need the private key. We obtain this from the keystore by calling its getKey method. The getKey method requires the alias for the key and its password. The method returns an instance of PrivateKey to the caller:

```
//get the private key for signing.
PrivateKey privateKey = (PrivateKey) ks.getKey(privateKeyAlias,
        privateKeyPass.toCharArray());
```

The program then creates a document as in the earlier case (refer to Listing 8-3), obtains its root element for later adding the signature, and opens a file for writing the signed document. These steps are the same as the ones in Listing 8-3.

Next, we create an instance of `XMLSignature` by using the Digital Signature Algorithm (DSA):

```
// Create a XMLSignature instance that uses RSA_SHA1 algorithm
XMLSignature signature = new XMLSignature(doc, f.toURL().toString(),
        XMLSignature.ALGO_ID_SIGNATURE_DSA);
```

We perform canonicalization as in the earlier case and add the canonicalized document to the created signature:

```
// Create canonical XML
Transforms transforms = new Transforms(doc);
transforms.addTransform(Transforms.TRANSFORM_ENVELOPED_SIGNATURE);

// Create canonicalized document to signature
 signature.addDocument("", transforms,
         MessageDigestAlgorithm.ALGO_ID_DIGEST_SHA1);
```

We retrieve the certificate from the keystore by calling its `getCertificate` method. We will attach this certificate to the signed document:

```
//Add in the KeyInfo for the certificate that we used the private key of
X509Certificate cert =
      (X509Certificate) ks.getCertificate(certificateAlias);
```

The `getCertificate` method takes the certificate alias as its parameter and returns an instance of `X509Certificate`. We add this certificate and the public key associated with it to the `signature` object:

```
// Add the information to signature
signature.addKeyInfo(cert);
signature.addKeyInfo(cert.getPublicKey());
```

Finally, we add the `signature` object to the document root as in the earlier example:

```
// Add signature itself to the PO document
root.appendChild(signature.getElement());
```

The document is signed by using the `sign` method of the `XMLSignature` class:

```
// Sign the document
signature.sign(privateKey);
```

The signed document is serialized to an output file by using the `XMLUtils` class as in the earlier example:

```
// Create an output stream
FileOutputStream fos = new FileOutputStream(f);

// Output the memory document using XMLUtils.
XMLUtils.outputDOMc14nWithComments(doc, fos);
```

Running the CertifiedPO Application

Compile the source shown in Listing 8-5 by using the following command line:

```
C:\<working folder>\ch08\src>javac -d . CertifiedPO.java
```

Run the application by using the following command line:

```
C:\<working folder>\ch08\src>java apress.ApacheXML.ch08.CertifiedPO
```

On a successful run of the application, you will find the PO-certified.xml file created in your working folder. Outputting this file to your console or opening it in your browser would produce a screen output similar to the one shown here:

```
<PurchaseOrder>
  <signedContents>
    We request that you EXECUTE the following trades
    <stock>GFW</stock>
    <quantity>50</quantity>
    <price>25.35</price>
    <type>B</type>
    <stock>ABNPRF</stock>
    <quantity>100</quantity>
    <price>24.83</price>
    <type>S</type>
  </signedContents>
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod
        Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315">
      </CanonicalizationMethod>
      <SignatureMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1">
      </SignatureMethod>
      <Reference URI="">
        <Transforms>
          <Transform
            Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature">
          </Transform>
        </Transforms>
        <DigestMethod
          Algorithm="http://www.w3.org/2000/09/xmldsig#sha1">
        </DigestMethod>
        <DigestValue>KhvCkxATMSfEBakaOMscs1AaDmw=</DigestValue>
      </Reference>
    </SignedInfo>
    <SignatureValue>d9Z4Q3E7T+gJgQHBMCCtJm4hG6QzhU39B/bvbWZnizVFOMH3wc49vQ==
    </SignatureValue>
    <KeyInfo>
      <X509Data>
        <X509Certificate>
MIIDIzCCAuECBEPgThQwCwYHKoZIzjgEAwUAMHcxCzAJBgNVBAYTAklOMRQwEgYDVQQIEwtNYWhhcmF
zaHRyYTEPMA0GA1UEBxMGTXVtYmFpMQ4wDAYDVQQKEwVBQkNPTTESMBAGA1UECxMJQXV0aG9yaW5nMR
OwGwYDVQQDExRQb29ybmFjaGFuZHJhIFNhcmFuZzAeFw0wNjAyMDEwNTU4NDRaFw0wNjA1MDIwNTU4N
DRaMHcxCzAJBgNVBAYTAklOMRQwEgYDVQQIEwtNYWhhcmFzaHRyYTEPMA0GA1UEBxMGTXVtYmFpMQ4w
DAYDVQQKEwVBQkNPTTESMBAGA1UECxMJQXV0aG9yaW5nMROwGwYDVQQDExRQb29ybmFjaGFuZHJhIFN
hcmFuZzCCAbgwggEsBgcqhkjOOAQBMIIBHwKBgQD9f1OBHXUSKVLfSpwu7OTn9hG3UjzvRADDHj+Atl
EmaUVdQCJR+1k9jVj6v8X1ujD2y5tVbNeBO4AdNG/yZmC3a5lQpaSfn+gEexAiwk+7qdf+t8Yb+DtX5
8aophUPBPuD9tPFHsMCNVQTWhaRMvZ1864rYdcq7/IiAxmdOUgBxwIVAJdgUI8VIwvMspK5gqLrhAvw
WBz1AoGBAPfhoIXWmz3ey7yrXDa4V7l5lK+7+jrqgvlXTAs9B4JnUVlXjrrUWU/mcQcQgYC0SRZxI+h
MKBYTt88JMozIpuE8FnqLVHyNKOCjrh4rs6Z1kW6jfwv6ITVi8ftiegEkO8yk8b6oUZCJqIPf4Vrlnw
aSi2ZegHtVJWQBTDv+z0kqA4GFAAKBgQDfCQe9UlgB4/1k9gC9QqdwqTnJAzKQV+sCYkWWckmSL1LvT
jcX37pvOTO6azdSWDfdpWAH99TkrbTmX2wOopuKSTGCDNrf+bbmiWZeLg/36Vnm4F3lFLXzKk25sWx4
5DzkpW8cEu7T5G/3uwAgRVmkrHTqVOD7ezbwytDOEADOMDALBgcqhkjOOAQDBQADLwAwLAIUMKHnmQm
z4pJT1T37I7mUb7KjhfYCFCU+b7suEeRGHQ1k7ZNeSllb2xs7
        </X509Certificate>
      </X509Data>
      <KeyValue>
        <DSAKeyValue>
          <P>
/X9TgR11EilS30qcLuzk5/YRt1I870QAwx4/gLZRJmlFXUAiUftZPY1Y+r/F9bow9subVWzXgTuAHTR
v8mZgt2uZUKWkn5/oBHsQIsJPu6nX/rfGG/g7V+fGqKYVDwT7g/bTxR7DAjVUE1oWkTL2dfOuK2HXKu
/yIgMZndFIAcc=
          </P>
          <Q>l2BQjxUjC8yykrmCouuEC/BYHPU=</Q>
          <G>
9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJFnEj6EwoFhO3zwk
yjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hWuWfBpKLZl6Ae1
UlZAFMO/7PSSo=
          </G>
          <Y>
3wkHvVJYAeP9ZPYAvUKncKk5yQMykFfrAmJFlnJJki9S7O43F9+6b9Ezums3Ulg33aVgB/fU5K205l9
sNKKbikkxggza3/m25olmXi4P9+lZ5uBd5RS18ypNubFseOQ85KVvHBLuO+Rv97sAIEVZpKxO6lTg+3
s28MrQ9BAA9DA=
          </Y>
        </DSAKeyValue>
      </KeyValue>
    </KeyInfo>
  </Signature>
</PurchaseOrder>
```

Note that the output document now contains an `X509Certificate` element that contains the encoded data for the certificate. The receiver will use this certificate to establish trust in the sender-provided public key.

We will now develop an application for verifying this document.

Verifying Documents Containing Digital Certificates

Like the earlier example of Listing 8-3, the document verification process using a digital certificate is easy. The entire application for document verification is available in the `VerifyCertifiedPO.java` file in your downloaded code. I will discuss only the relevant code, shown here:

```
// Retrieve the key information
KeyInfo ki = signature.getKeyInfo();

if (ki != null) {
    X509Certificate cert = signature.getKeyInfo().getX509Certificate();

    if (cert != null) {
      boolean result = signature.checkSignatureValue(cert);
      String str = null;
      if (result)
        str = "The document " + f.toURL().toString() + " is valid!";
      else
        str = "The document " + f.toURL().toString() + " is invalid!";
          System.out.println(str);
    } else {
      System.out.println("Did not find a Certificate");
    }

  } else {
    System.out.println("Missing KeyInfo");
  }
```

As in the earlier case, we retrieve the `Signature` element from the received document and obtain the key information from it. From the key information, we retrieve the embedded certificate by calling its `getX509Certificate` method:

```
X509Certificate cert = signature.getKeyInfo().getX509Certificate();
```

We verify the document by calling the `checkSignatureValue` method on the `signature`. The method takes the preceding `cert` as its parameter. The method returns a `boolean` value that indicates the success or failure of the verification:

```
boolean result = signature.checkSignatureValue(cert);
```

■Note The sender need not attach a certificate to every signed document, if such a certificate is made available to the intended receiver in advance. The receiver can store the certificate in his own keystore and refer to it whenever the authenticity of the sender's public key is to be established.

Running the VerifyCertifiedPO Application

Compile the provided VerifyCertifiedPO.java file by using the following command line:

```
C:\<working folder>\ch08\src>javac -d . VerifyCertifiedPO.java
```

Run the application by using the following command line:

```
C:\<working folder>\ch08\src>java apress.ApacheXML.ch08.VerifyCertifiedPO
```

You will see output similar to the following:

```
Trying to verify file:/C:/apress/ch08/src/PO-certified.xml
Feb 4, 2006 4:07:43 PM org.apache.xml.security.signature.Reference verify
INFO: Verification successful for URI ""
The document file:/C:/apress/ch08/src/PO-certified.xml is valid!
```

So far you have seen how to sign a document and how a receiver of the document establishes the document's authenticity and integrity. However, in all of these cases, the document content always remains readable to an interceptor. You will now study the techniques of encryption and decryption so that the document content would make sense only to a legitimate intended receiver and not to an interceptor.

Using XML Encryption/Decryption

As stated earlier in this chapter, encryption transforms the document content known as plaintext to gibberish, or ciphertext. The encrypted document makes little or no sense to anybody receiving it. A legitimate receiver has to decrypt the document by using a shared secret key to retrieve the document in its original format.

In this section, I will illustrate encryption and decryption techniques by using an example. We will use the dynamic PO document from Listing 8-3 as a sample document for encryption. We will encrypt the content of this document and output it to a file. Later, the receiver will decrypt this document to read the original content.

Developing an Application for Encrypting the PO

As mentioned earlier, we will use a symmetric key for encryption because of its quicker processing time as compared to asymmetric-key cryptography. We will create a symmetric key during the application session. This symmetric key, which is valid only during the application session, must be given to the receiver. Thus, we will serialize the key to a file and distribute it along with the encrypted file. We will encrypt the symmetric key by using another key called the key encryption key (KEK) so that even if an interceptor gets his hands on the symmetric key, it would not be useful to him unless he decrypts the received key. The entire process is illustrated in Figure 8-10.

The key generator uses salt[9] to randomize the password.

9. Refer to *RSA Security's Official Guide to Cryptography* by Steve Burnett and Stephen Paine (McGraw-Hill, 2001).

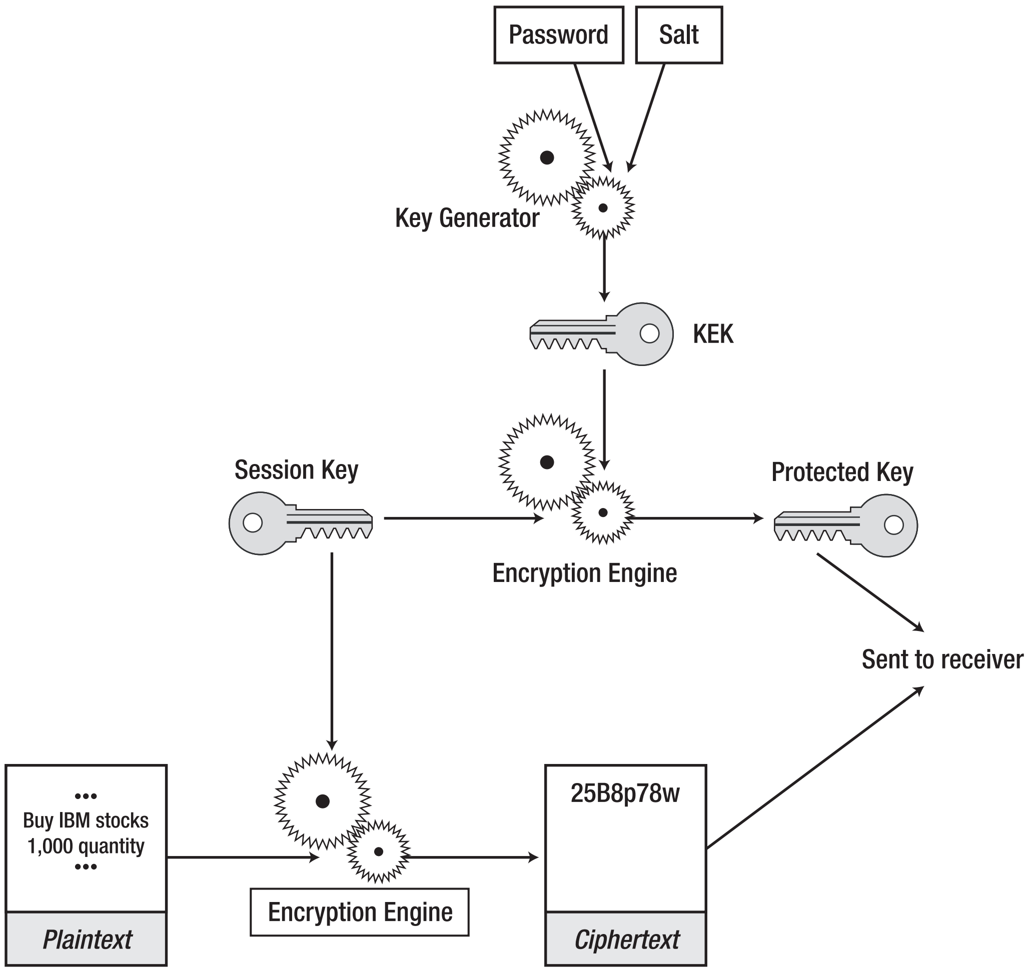

Figure 8-10. *Encryption based on combination of symmetric and asymmetric cryptography*

So let's now look at the application that encrypts our PO. Listing 8-6 illustrates how to encrypt a source XML document.

Listing 8-6. *Application for Encrypting the PO* (`Ch08\src\EncryptPO.java`)

```
/*
 * EncryptPO.java
 */
package apress.ApacheXML.ch08;

// import required classes

public class EncryptPO{
```

```
static Document createDocument() throws Exception {
// Same as in Listing 8-3
}

public static void main(String unused[]) throws Exception {
  org.apache.xml.security.Init.init();

  // Create a PO document that we intend to encrypt
  Document document = createDocument();

  // Generate a 128 bit AES symmetric key for encryption
  KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
  keyGenerator.init(128);
  Key symmetricKey = keyGenerator.generateKey();

  // Generate a key (KEK) for encrypting the above symmetric key
  keyGenerator = KeyGenerator.getInstance("DESede");
  Key kek = keyGenerator.generateKey();

  // Store the KEK to a file
  File kekFile = new File("SecretKEK");
  FileOutputStream f = new FileOutputStream(kekFile);
  f.write(kek.getEncoded());
  f.close();
  System.out.println("Key encryption key (KEK) stored in " +
         kekFile.toURL().toString());

  // Get a Cipher instance
  XMLCipher keyCipher = XMLCipher.getInstance(XMLCipher.TRIPLEDES_KeyWrap);

  // Initialize Cipher for wrapping KEK
  keyCipher.init(XMLCipher.WRAP_MODE, kek);

  // Encrypt symmetric key with KEK
  EncryptedKey encryptedKey = keyCipher.encryptKey(document, symmetricKey);

  // Obtain document root element reference for encypting the document
  Element rootElement = document.getDocumentElement();

  // Create and initalize cipher for encyption using our symmetric key
  XMLCipher xmlCipher = XMLCipher.getInstance(XMLCipher.AES_128);
  xmlCipher.init(XMLCipher.ENCRYPT_MODE, symmetricKey);

  // Add the document to be signed and the encryption key into
  // a KeyInfo instance
  KeyInfo keyInfo = new KeyInfo(document);
  keyInfo.add(encryptedKey);
```

```
        // Add the key information to cipher
        EncryptedData encryptedData = xmlCipher.getEncryptedData();
        encryptedData.setKeyInfo(keyInfo);

        // This is where actual encryption takes place
        xmlCipher.doFinal(document, rootElement, true);

        // Open file for storing encrypted document
        File encryptionFile = new File("encryptedPO.xml");
        f = new FileOutputStream(encryptionFile);

        // Create transformer for outputting encrypted document to a stream
        Transformer transformer =
              TransformerFactory.newInstance().newTransformer();
        transformer.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "yes");

        // Perform the transformation
        DOMSource source = new DOMSource(document);
        StreamResult result = new StreamResult(f);
        transformer.transform(source, result);

        f.close();
        System.out.println("Wrote encrypted document to " +
              encryptionFile.toURL().toString());
    }
}
```

After initializing the security system, the `main` function creates the PO document as in our earlier examples.

Next, we generate a 128-bit symmetric key by using the Advanced Encryption Standard (AES) algorithm:

```
        // Generate a 128 bit AES symmetric key for encryption
        KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
        keyGenerator.init(128);
        Key symmetricKey = keyGenerator.generateKey();
```

■ **Note** The discussion of AES or any other algorithm for key generation is beyond the scope of this book. You can refer to a local copy of the Java Cryptography Extension (JCE) reference guide (`<JDK installation folder>\docs\guide\security\JCERefGuide.html`) or its online version (at `http://java.sun.com/products/jce/reference/docs/index.html`) for more general information on cryptography.

Next, we will generate a key for encrypting this symmetric key so that if it landed in the wrong hands en route to the intended receiver, the interceptor would not be able to use it to decrypt the original document (unless the interceptor figured out how to decrypt the received key to retrieve the original symmetric key). As noted earlier in this chapter, a key that is used for encrypting another key (symmetric key) is generally called a KEK—a key that encrypts another key.

We generate the KEK like the earlier key generation, by using the `generateKey` method of the `KeyGenerator` class, except that we use a different algorithm for key generation. This is an additional safety provided for securing the document contents. The point is that even if somebody breaks the algorithm used for encryption, he will have to now break two such algorithms:

```
// Generate a key (KEK) for encrypting the above symmetric key
keyGenerator = KeyGenerator.getInstance("DESede");
Key kek = keyGenerator.generateKey();
```

After the KEK is generated, we store it to a file for transporting it to the intended receiver:

```
// Store the KEK to a file
File kekFile = new File("SecretKEK");
FileOutputStream f = new FileOutputStream(kekFile);
f.write(kek.getEncoded());
f.close();
System.out.println("Key encryption key (KEK) stored in " +
       kekFile.toURL().toString());
```

We will now look into how to encrypt our document by using these keys. As mentioned earlier, the encrypted data is called ciphertext. Apache defines a class called `XMLCipher` for representing this ciphertext. We create an instance of this class by calling its `getInstance` method. The `getInstance` method receives the algorithm that we intend to use for encryption as a parameter:

```
// Get a Cipher instance
XMLCipher keyCipher = XMLCipher.getInstance(XMLCipher.TRIPLEDES_KeyWrap);
```

We use a Triple DES (Digital Encryption Standard) algorithm in the current case.

Note Many more algorithms are available in the Apache implementation of XML signatures. Refer to the documentation of the `XMLCipher` class for a list of implemented algorithms.

Next, we initialize the cipher for wrapping the KEK:

```
// Initialize Cipher for wrapping KEK
keyCipher.init(XMLCipher.WRAP_MODE, kek);
```

The first parameter to the `init` method specifies whether the cipher is used for encryption and decryption, or for key wrapping and unwrapping. We will use the encryption mode later in our code when we create a cipher for encrypting our document.

We now encrypt the symmetric key using our KEK by calling the encryptKey method on the cipher:

```
// Encrypt symmetric key using KEK
EncryptedKey encryptedKey = keyCipher.encryptKey(document, symmetricKey);
```

After encrypting the key, we encrypt the document. We first obtain the reference to the root node:

```
// Obtain document root element reference for encypting the document
Element rootElement = document.getDocumentElement();
```

We create another cipher for encrypting our document, initialize it for the appropriate mode, and specify the use of the previously generated symmetric key:

```
// Create and initalize cipher for encyption using our symmetric key
XMLCipher xmlCipher = XMLCipher.getInstance(XMLCipher.AES_128);
xmlCipher.init(XMLCipher.ENCRYPT_MODE, symmetricKey);
```

Before we perform the actual encryption, we need to inform the cipher about the whereabouts of our document to be encrypted and the encryption key. For this, we construct a KeyInfo object that refers to the document to be signed. We use its add method to add the encryption key to it:

```
// Add the document to be signed and the encryption key into
// a KeyInfo instance
KeyInfo keyInfo = new KeyInfo(document);
keyInfo.add(encryptedKey);
```

Now, we add the keyInfo itself to the cipher. This is done by obtaining a reference to its EncryptedData object and setting the KeyInfo on it:

```
// Add the key information to cipher
EncryptedData encryptedData = xmlCipher.getEncryptedData();
encryptedData.setKeyInfo(keyInfo);
```

In the next step, when we perform the actual encryption, this encryptedData element will contain the ciphertext corresponding to the plaintext from our document.

We perform the actual encryption by calling the doFinal method on the cipher object:

```
// This is where actual encryption takes place
xmlCipher.doFinal(document, rootElement, true);
```

The first parameter to the doFinal method refers to the context document to be encrypted; in our case, it is the program-generated PO document. The second parameter indicates the node whose contents are to be encrypted, and the third parameter indicates that we want to encrypt the contents of the node and not the node itself.

After successful execution, the doFinal method replaces the contents of the EncryptedData element with the generated ciphertext. At this stage, you have an in-memory encrypted document corresponding to our PO. We will serialize this to a physical file for transporting it to an intended receiver. The rest of the code in the main function does the job of serialization of the in-memory DOM structure by using XSLT transformations (discussed in Chapter 5).

Running the EncryptPO Application

Compile the `EncryptPO.java` file by using the following command line:

```
C:\<working folder>\ch08\src>javac -d . EncryptPO.java
```

Run the application by using the following command:

```
C:\<working folder>\ch08\src>java apress.ApacheXML.ch08.EncryptPO
```

The console output from running this application is as follows:

```
Key encryption key (KEK) stored in file:/C:/apress/ch08/src/SecretKEK
Wrote encrypted document to file:/C:/apress/ch08/src/encryptedPO.xml
```

The application has created two files, the `SecretKEK` that contains the key for decrypting the symmetric session key used for document encryption, and the encrypted document itself in the `encryptedPO.xml` file. The receiver must have access to both files to recover the original document contents.

Examining the Encrypted Document

If you open the generated `encryptedPO.xml` file in your browser, you will see output similar to the following:

```
<PurchaseOrder>
  <xenc:EncryptedData
    xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
    Type="http://www.w3.org/2001/04/xmlenc#Content">
    <xenc:EncryptionMethod
      Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"
      xmlns:xenc="http://www.w3.org/2001/04/xmlenc#" />
    <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <xenc:EncryptedKey
        xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
        <xenc:EncryptionMethod
          Algorithm="http://www.w3.org/2001/04/xmlenc#kw-tripledes"
          xmlns:xenc="http://www.w3.org/2001/04/xmlenc#" />
        <xenc:CipherData
          xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
          <xenc:CipherValue
```

```
            xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
            IgeLdkOmi9KOOq8bbsKAeImpXA8/FEnNKAE+SU3UOtQ=
            </xenc:CipherValue>
        </xenc:CipherData>
      </xenc:EncryptedKey>
    </ds:KeyInfo>
    <xenc:CipherData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
      <xenc:CipherValue
        xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
2zqKbOKWiytZSOIwHQ5Zdb9o6stHPH8uhb2We9dQCo33a/r7JmQIe5TBdEFjl8N+ik1XZ6dTg8b7z1M
MVuN8kVZYrzQNcG/qpcuhGzbeDLYoEU7zjbMoJ1IX4fNaHltedLVJDVnPOk6ZzuFD8m6qj6ww9KtN7l
nBXGgqochuZngIBZiJV2TV41mk5SEL8HbcBHf4M3+/JJkDpgOFOtnQaA+FOX4cvd3GuIFpqy5elasGr
nVaS1DUuHfesBtPnaAJRx/CSeFD+l2+v6/vNLvbQERHIwL88hkSybTwSuXmG6B4OQvwsAeS8cO6koDJ
DGWIWAF4+gzODc9nONemvjojyA==
      </xenc:CipherValue>
    </xenc:CipherData>
  </xenc:EncryptedData>
</PurchaseOrder>
```

Note the presence of the `CipherValue` and `CipherData` elements. These encapsulate the generated ciphertext. Our receiver will have to decrypt this data by using the procedures described in the next section to retrieve the original document contents. Any interceptor receiving this encrypted file will not gain any information about the contents in the original document.

Now, we will develop an application that decrypts the PO: `encryptedPO.xml`.

Developing an Application for Decrypting the PO

The application that decrypts our encrypted PO must have access to two files: the encrypted document file itself, and the file containing the KEK. The application will have to read the KEK, use it to decrypt the symmetric session key used for encryption, and then decrypt the encrypted document. This process is depicted in Figure 8-11.

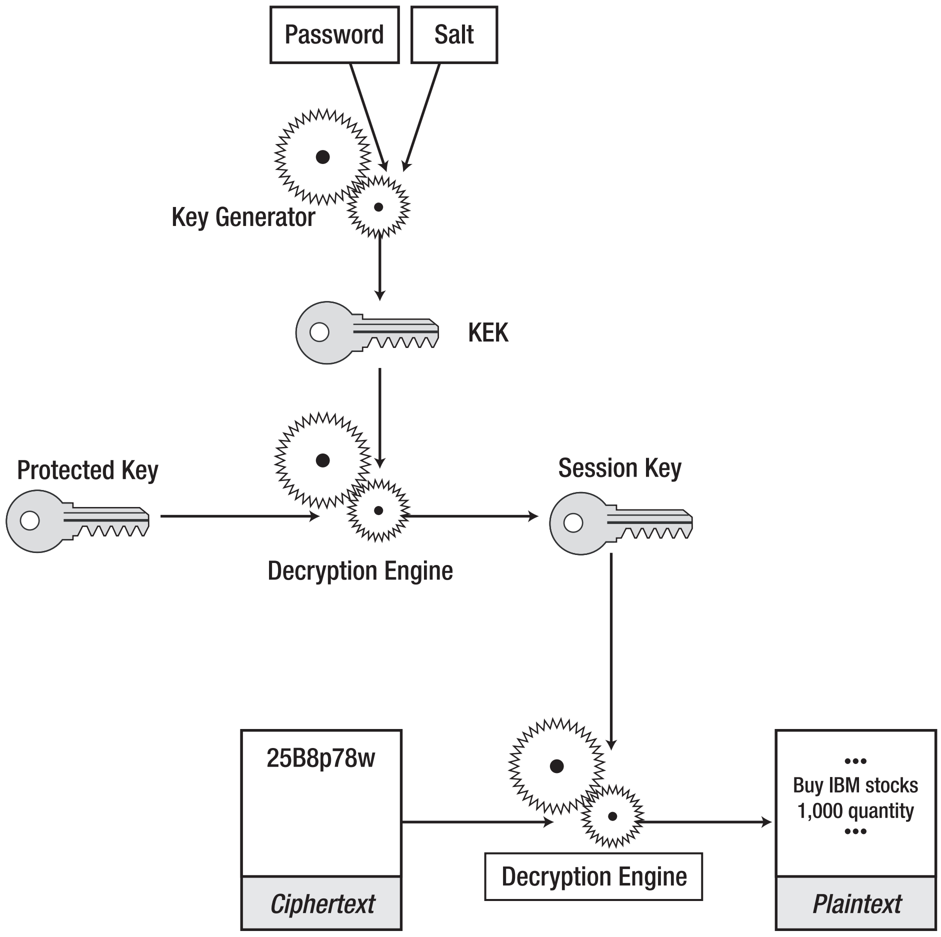

Figure 8-11. *Decryption based on both symmetric and asymmetric cryptography*

Listing 8-7 illustrates how the decryption is performed.

Listing 8-7. *Application for Decrypting Encrypted Purchase Order Document (*`Ch08\src\DecryptPO.java`*)*

```
/*
 * DecryptPO.java
 */
package apress.ApacheXML.ch08;

// import required classes

public class DecryptPO {
```

```
public static void main(String unused[]) throws Exception {
  org.apache.xml.security.Init.init();

  // Open the encrypted document and build a DOM tree from it
  File f = new File("encryptedPO.xml");
  javax.xml.parsers.DocumentBuilderFactory dbf =
          javax.xml.parsers.DocumentBuilderFactory.newInstance();
  dbf.setNamespaceAware(true);
  javax.xml.parsers.DocumentBuilder db = dbf.newDocumentBuilder();
  Document document = db.parse(f);
  System.out.println("Encrypted PO loaded from " +f.toURL().toString());

  // Read encrypted data element
  Element encryptedDataElement =(Element) document.getElementsByTagNameNS(
          EncryptionConstants.EncryptionSpecNS,
          EncryptionConstants._TAG_ENCRYPTEDDATA).item(0);

  // Load KEK
  String fileName = "SecretKEK";
  File kekFile = new File(fileName);

  // Construct the DES key specs from the file contents
  DESedeKeySpec keySpec = new DESedeKeySpec(
          JavaUtils.getBytesFromFile(fileName));

  // Create a key factory instance
  SecretKeyFactory skf = SecretKeyFactory.getInstance("DESede");

  // Generate the key from the specs
  Key kek = skf.generateSecret(keySpec);
  System.out.println("Key encryption key loaded from " +
          kekFile.toURL().toString());

  // Get cipher instance
  XMLCipher xmlCipher = XMLCipher.getInstance();

  // Initialize cipher for decryption
  xmlCipher.init(XMLCipher.DECRYPT_MODE, null);

  // Set the KEK that contains the key for decryption
  xmlCipher.setKEK(kek);

  // Perform the actual decryption
  xmlCipher.doFinal(document, encryptedDataElement);
```

```
    // Open file for writing in-memory decrypted document
    File decryptedFile = new File("decryptedPO.xml");
    FileOutputStream fo = new FileOutputStream(decryptedFile);

    // Serialize DOM to file using transformations
    TransformerFactory factory = TransformerFactory.newInstance();
    Transformer transformer = factory.newTransformer();
    transformer.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "yes");
    DOMSource source = new DOMSource(document);
    StreamResult result = new StreamResult(fo);
    transformer.transform(source, result);

    fo.close();
    System.out.println("Wrote decrypted PO to " +
            decryptedFile.toURL().toString());
  }
}
```

After initializing the system, the `main` function loads the encrypted document and constructs an in-memory DOM for further processing. This code is similar to the code in Listing 8-4 that described how to decrypt a signed document.

The program then extracts the encrypted data element for the in-memory document:

```
// Read encrypted data element
Element encryptedDataElement =(Element) document.getElementsByTagNameNS(
        EncryptionConstants.EncryptionSpecNS,
        EncryptionConstants._TAG_ENCRYPTEDDATA).item(0);
```

Note that this encrypted data element contains the ciphertext and also a reference to the keys used for encryption. Before we decrypt the data, we must obtain the KEK and thus the symmetric key used for encryption. We construct the `File` object for reading the KEK:

```
// Load KEK
String fileName = "SecretKEK";
File kekFile = new File(fileName);
```

We construct the DES key specs from the data read from the KEK file:

```
// Construct the DES key specs from the file contents
DESedeKeySpec keySpec = new DESedeKeySpec(
        JavaUtils.getBytesFromFile(fileName));
```

We create an instance of the secret key factory:

```
// Create a key factory instance
SecretKeyFactory skf = SecretKeyFactory.getInstance("DESede");
```

We now generate the key itself by using the factory and the key specifications read from the KEK file:

```
// Generate the key from the specs
Key kek = skf.generateSecret(keySpec);
```

Now that we have obtained the encrypted data and the key used for encryption (in fact, only the KEK and not the symmetric key), we will proceed with the decryption. For this, we first construct the instance of the `XMLCipher` class. Remember, this class represents the ciphertext:

```
// Get cipher instance
XMLCipher xmlCipher = XMLCipher.getInstance();
```

We initialize the cipher for decrypt mode:

```
// Initialize cipher for decryption
xmlCipher.init(XMLCipher.DECRYPT_MODE, null);
```

We set the KEK on the cipher. Note that the KEK encapsulates the symmetric key:

```
// Set the KEK that contains the key for decryption
xmlCipher.setKEK(kek);
```

We now perform the actual decryption by calling the `doFinal` method on the cipher object:

```
// Perform the acutal decryption
xmlCipher.doFinal(document, encryptedDataElement);
```

At this stage, the content of the encrypted data element is replaced by the decrypted data, that is, the plaintext. We now simply need to serialize this in-memory DOM data to a file. The rest of the code in the `main` function does this. This uses transformations and is similar to the code described in earlier examples.

Running the `DecryptPO` Application

Compile the provided `DecryptPO.java` file by using the following command line:

```
C:\<working folder>\ch08\src>javac -d . DecryptPO.java
```

Run the application by using the following command:

```
C:\<working folder>\ch08\src>java apress.ApacheXML.ch08.DecryptPO
```

On a successful run of the application, you will find a file called `decryptedPO.xml` generated in your work folder. The content of this file is as follows:

```
<PurchaseOrder>
  <signedContents>
    We request that you EXECUTE the following trades
    <stock>GFW</stock>
    <quantity>50</quantity>
    <price>25.35</price>
    <type>B</type>
    <stock>ABNPRF</stock>
    <quantity>100</quantity>
    <price>24.83</price>
    <type>S</type>
   </signedContents>
</PurchaseOrder>
```

Note that this is the content generated by our sender.

Summary

In this chapter, you studied an important aspect of XML programming: how to implement security in your XML applications. XML is widely used for data transport, but if the data is not securely transported, these XML applications will find little use in practical life. The XML signature provides this security.

The chapter started by describing important security terms such as authentication, authorization, and nonrepudiation. Authentication signifies that you are allowed to use the application. Within the application, what you are allowed to do is decided by the authorization. Nonrepudiation guarantees that the parties involved in the secured communication cannot deny at a later time that they performed a particular operation during their secured communication. You were also introduced to other security terms such as message integrity and message confidentiality.

These requirements of security are implemented by using two cryptography techniques: symmetric and asymmetric. As their names suggest, symmetric cryptography is based on a single shared key used by both sender and receiver during their secured communication. Asymmetric cryptography uses a matching key pair consisting of a public and a private key. The public key is given to everybody, while the private key is held only by the person generating the key pair. The sender signs and/or encrypts the document by using her private key, and the receiver verifies and/or decrypts the document by using the public key supplied by the sender.

Asymmetric cryptography solves the problem of sharing and distributing the key. However, asymmetric cryptography is much slower than its counterpart. Thus, in practice we use a combination of both. We create a message digest on the input document by using a symmetric key, and encrypt this symmetric key by using asymmetric cryptography. The symmetric key is encrypted by using the private key of the sender. The receiver retrieves the symmetric key by using the sender's public key. The encrypted session key must itself be transported over a secured channel. This is what is done in HTTPS protocol, where the session key is transported after establishing the authentication. In HTTPS, the session key is also changed periodically to add further security against hackers who may intercept network traffic and perform a brute-force or any other attack on a session key.

The XML signature uses these cryptography techniques for implementing security in XML applications. Unlike other digital signatures, an XML signature allows you to sign only a part of the input document. This gives us an advantage of being able to pass the XML document through a series of approvers, where each approver can process the document and add some information to it. Note that when using a digital signature, because the entire document as a whole is signed, any changes made to the document invalidate its signature.

This chapter also described the three types of XML signatures: The signature can be included in a document that is signed, or the `Signature` element can include the document, or the signature can be totally detached from the document.

Because of XML's less-restrictive structuring, two XML documents may differ physically but carry the same logical content. Because digital signatures are sensitive to the physical content of a signed document, creating identical signatures for the two logically same documents is impossible. This problem is solved by creating canonical XML, whereby predefined transformation rules transform the input document to standard XML format.

This entire theory was explained further through examples. We considered a case of signing and verifying a purchase order by using both runtime-generated key pairs and the key pair taken from a digital certificate.

Signing the document helps in establishing the sender's identity and the integrity of the signed document's content. It does not solve the problem of hiding the content from the hacker's eyes. Encryption techniques solve this problem by converting the plaintext to gibberish that does not make any sense to the interceptor. The legitimate user of the data decrypts the data to look up the original content. The chapter described the Apache implementation of encryption/decryption techniques with the help of practical examples.