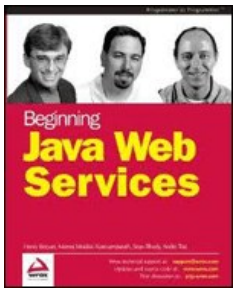


Beginning Java Web Services

by Henry Bequet

Chapter 9: Securing Web Services

Overview

One of the critical aspects of enterprise applications that needs to be addressed with utmost care is security. Security covers a wide variety of enterprise application behavior including authentication, authorization, integrity, confidentiality, non-repudiation, auditing, and single sign-on.

Most of these security concerns existed long before the advent of web services. Over the years powerful security solutions have evolved to address these security issues. These solutions include message digests, digital signatures, symmetric/asymmetric key cryptography, digital certificates, SSL, and so on. Most of these technologies are relevant in the web services world for providing transport-level security.

However, the traditional security solutions deal with messages as a whole. For example, it is pretty straightforward to generate the digital signature or a message digest of an entire XML document and transmit it over wire. But the power of XML lies in the way it can represent hierarchical data. In hierarchical data we may want to encrypt only certain parts of the document and not the whole document. For example, in an XML document representing a purchase order, we may want to encrypt only sensitive information such as a credit card number. This is extremely important as web services normally work in federated environments and the propagation and routing of web services through various intermediaries depend on the information present in the header blocks of the SOAP envelopes representing those web services. In such cases we may want to encrypt only the payload data instead of encrypting the whole message.

The more recent XML and web services security specifications such as XML Signature, XML Encryption, WS-Security, and SAML are build on top of the concepts introduced by the traditional security solutions to provide secure means of transmitting XML data. XML Signature and Encryption are W3C initiatives to provide encryption and digital signature functionality to parts of XML documents.

WS-Security forms the foundation of a set of specifications initiated by IBM, Microsoft, and Verisign, to provide security solutions for web services. SAML is an OASIS specification for propagating security contexts such as authentication and authorization information.

In this chapter we will cover the various security issues relevant to writing web services and the security solutions that address these issues. Namely:

- Security issues
- Traditional security solutions
- HTTP authentication methods
- Web services security scenarios
- XML security specifications
- WS-Security and related specifications
- Security Assertion Markup Language

Setting Up the StockCore Client

In this chapter we will be using the stock core example we developed in Chapter 2 to illustrate the various aspects of security. We will use this example to illustrate:

- How web services can be accessed over SSL
- How we can provide authenticated access to web services using both standard HTTP authentication methods and custom authentication methods
- How we can use WS-Security to sign parts of a SOAP envelope representing the web service request from the client that is verified by the server

In most of the examples we will be using Axis and Tomcat. However, for the WS-Security example we will be using WSTK and Tomcat. WSTK uses Axis as its web services framework. Please refer to earlier chapters on how to install and configure Axis and WSTK.

Try It Out: Stock Quote Service Client

1. The source code for the client class `GetQuote.java` is shown below:

```
package com.wrox.jws.stockcore;
```

Import the required Axis classes:

```
import org.apache.axis.AxisFault;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import org.apache.axis.utils.Options;
```

Import the required JAX-RPC classes:

```
import javax.xml.rpc.ParameterMode;
import javax.xml.namespace.QName;

import java.net.URL;

public class GetQuote {

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

The program expects the symbol and endpoint URL as mandatory command-line arguments. We can optionally pass the user name and password as well:

```
    if(args == null || args.length < 2) {
      System.out.println("Usage: java GetQuote symbol url user password");
      System.exit(0);
    }

    String symbol = args[0];

    URL url = new URL(args[1]);
```

Create the service and call:

```
    Service service = new Service();

    Call call = (Call) service.createCall();
```

Set the endpoint URL and the operation name:

```
    call.setTargetEndpointAddress( url );
    call.setOperationName(new QName("StockQuoteService", "getQuote"));
```

Set the in and out parameters:

```
    call.addParameter("symbol", XMLType.XSD_STRING, ParameterMode.IN );
    call.setReturnType(XMLType.XSD_STRING);
```

Set the user name and password if specified:

```
    if(args.length == 4) {
      call.setUsername(args[2]);
      call.setPassword(args[3]);
    }
```

Invoke the stock quote service and print the results:

```
    System.out.println(call.invoke( new Object[] {symbol}));

  }
}
```

2. Ensure that we have all the JAR files present in the `\lib` directory of the Axis installation in the classpath as well as a JAXP-compliant parser such as Xerces.

3. Compile the class using the following command (assuming you're in a `\src` directory):

```
javac -d ..\classes GetQuote.java.
```

Note You may compile the class to wherever you choose, but please bear in mind that you will need to modify later command-line arguments in order to locate it. We have stored it in `com\wrox\jws\stockcore\GetQuote.class`, so this is what our command line will reflect.

This is the class we'll use in most of our examples without making any changes to the code. We will provide the various security services to the web service and client by altering the deployment information and command-line arguments.

How It Works

The Axis `Service` class implements the JAX-RPC `Service` interface and provides a dynamic interface for invoking web services. This class provides a variety of constructors for passing the WSDL location and service name. However, we have used the empty constructor and set the required information manually rather than using the WSDL location. We use the `service` object to create the call object:

```
Service service = new Service();
Call call = service.createCall();
```

The Axis `Call` class implements the JAX-RPC `Call` interface. We use the methods defined in the `Call` interface for setting the web service endpoint URL, operation name, user name, and password, and register the names and types of the web service arguments and the return value. The endpoint URL is normally mapped to the Axis servlet on the web container:

```
call.setTargetEndpointAddress( url );
call.setOperationName(new QName("StockQuoteService", "getQuote"));
call.addParameter("symbol", XMLType.XSD_STRING, ParameterMode.IN );
call.setReturnType(XMLType.XSD_STRING);

if(args.length == 4) {
  call.setUsername(args[2]);
  call.setPassword(args[3]);
}
```

Finally we use the `call` object to invoke the web service dynamically and print the result:

```
System.out.println(call.invoke( new Object[] {symbol}));
```

Setting Up Axis and WSTK

Before we can install our stock core service on Axis and WSTK we need to make the required classes and data available on the web container in which we run our web service. This section assumes that you have installed and configured Axis and WSTK on Tomcat as explained in the earlier chapters.

Recall from Chapter 3, that the files we need are:

```
stockcore.jar
stock_quote.xml
```

For WSTK, copy the JAR to the `%CATALINA_HOME%\webapps\wstk\WEB-INF\1ib` directory the XML file to the `%CATALINA_HOME%\webapps\wstk\WEB-INF\classes\` directory.

Now that everything is set up as we need it, the subsequent sections will look at how we deploy the web service in varying configurations to provide various security services such as authorization, integrity, confidentiality, and so on.

Security Issues

In enterprise application development, the term security is used in a wide variety of contexts. Writing secure applications involves addressing a whole host of application behavior related to security and privacy. In this section we will have a look at the various aspects of enterprise application behavior that are related to security.

Authentication

Authentication is the process by which the claims of an entity's identity are verified, when it participates in communication with another entity. The entity that is authenticated is called the **principal** and the information used to identify the principal is the **credentials.** Credentials can be passwords, digital certificates, or security cards.

The credentials provided by the principal are normally compared against a store that contains principals and credentials. These stores include databases, LDAP directory services, and so on. If the credentials provided by the principal matches those supplied in the store, the principal is given access to the system. In very simple terms a user accessing the system will provide their user ID and password to the system. The system will compare this information to the data in a user table stored in the database. If the user ID and password pair provided by the user matches one in the database, the user is granted access to the system.

Once the principal is authenticated, the next step is to verify whether the principal has enough authority to access the system. This is verified by a process called **authorization.**

Authorization

Authorization defines the access rights of an authenticated principal. For example, a customer logging on to an online shopping system would be able to buy things using the system. However, an administrator would have more rights than the customer, in order to perform administrative tasks. Authorization relies on the rights defined for authenticated principals.

Data Integrity

When two parties are involved in exchanging data electronically, it is of utmost importance that the data is not modified in any way while in transit. The commonest way for data integrity to be compromised is by what, in security terms, is known as the "Man-in-the-Middle-Attack". In this case someone intercepts the data sent by the sender, alters it, and resends it to the receiver. The receiver would have no knowledge that the data from the original sender was tampered with. This concept is extremely important in the web services world where data is exchanged over the Internet. Data integrity is normally ensured using message digests and digital signatures.

Data Confidentiality

Data confidentiality is another security aspect as important as data integrity. Where data integrity makes it extremely difficult for an attacker to tamper with the data, data confidentiality makes sure that even if the data is intercepted, it is difficult for the intercepting party to decipher the original content. This is important in exchanging confidential information such as credit card numbers and financial transactions. Data confidentiality is generally achieved using cryptography where the sender encrypts the data before sending it and the receiver decrypts it using some shared key information.

Non-repudiation

Security solutions that provide non-repudiation services prove the occurrence of a transaction. Non-repudiation leaves digital trails identifying the originator and recipient of a transaction and the information that was exchanged. This makes originating entities accountable for transactions. Non-repudiation is normally achieved using digital signatures, signature chains, and certificate authorities. All the previous concepts are covered in detail in later sections.

Auditing

It is impossible to write a system that is perfectly secure. We have to strike the balance between cost and security by making it more difficult for attackers to tamper with the security of the system than is worth their while. However, things can go wrong and in the event of security being compromised, there should always be enough auditing information describing when, where, and how the security was compromised so that the security hole can be detected and appropriate measures taken. Auditing provides a log of all important events within the application so that it is possible to use the audit trails to analyze when, where and how security was breached.

Single Sign-on

Single sign-on allows us to authenticate ourselves at one server and thereafter our security credentials (authorization/authentication information and so on) are automatically carried over to other web sites we visit. Single sign-on/sign-in is a security requirement that is very relevant in a federated system of web services. However, one major disadvantage of using single sign-on is that if the security at the authenticating service is breached, the intruder gains access to all the other services as well. This means single sign-on provides a single point of failure.

Traditional Security Solutions

We have seen the key security issues in developing enterprise-level web services. In this section we will provide an overview of the traditional security solutions that are relevant for web services. Please note that these solutions are classified as traditional because they existed before the advent of web services. In the following sections we will cover the shortcomings of traditional security solutions and look at the latest developments in providing security in the XML/web services world.

Traditional security solutions that address most of the above security issues, mainly in point-to-point scenarios include:

- Hashing
- Cryptography
- Digital Signatures
- Public Key Certificates
- Secure Socket Layer

Hashing

Hashing is the process of transforming a set of data into a fixed length digest, so that it is practically impossible for digests generated from two different sets of data (even if they only differ by a single character, as illustrated in the example below) to be identical:

```
M1 = "Cat on a hot tin roof"
M2 = "Cat on a rot tin roof"
H1 = h (M1)
H2 = h (M2)
H1 != H2
```

`H1` is the hash of `M1` and `H2` is the hash of `M2`. Even though `M1` and `M2` are very similar, it is practically impossible for `H1` and `H2` to be the same. This means that when we send our data and then the message digest to someone, it is impossible for anyone else to intercept the data in the middle and alter it, so that the message digest of the altered message generated using the same hashing function is same as that of the old one. Message digests are often used with digital signatures to provide data integrity and authentication.

There are industry-standard algorithms for performing hashing functions such as Secure Hash Algorithm (SHA), RSA-MD5-Message Digest Algorithm.

Bulk/Symmetric Encryption

Encryption is the process of generating a cipher of a block of data, typically via the use of secret encryption keys. Decryption is the reverse process; generating the original clear text from the cipher. In cryptographic terms clear text is the original data and cipher text is the encrypted

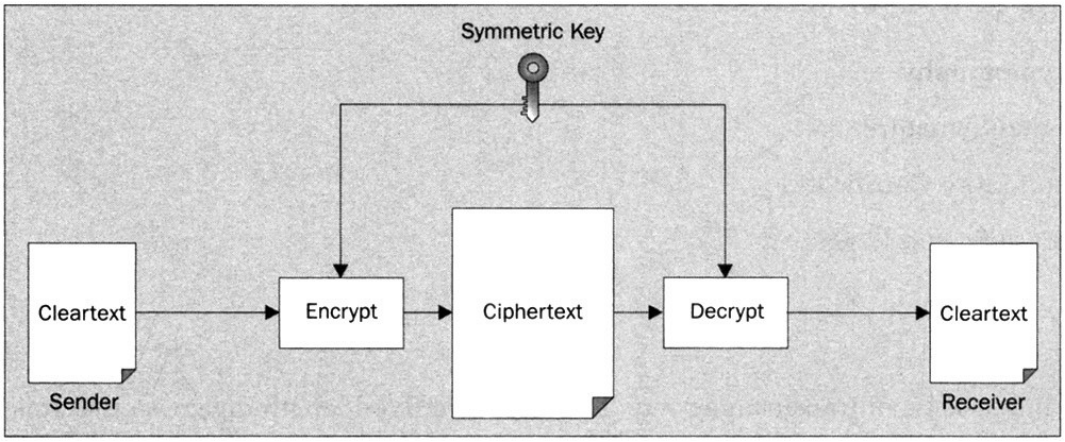
data. The secrecy of data is ensured by the secrecy of the key and not the secrecy of the algorithm used to encrypt/decrypt data. There are also industry-standard algorithms for encrypting/decrypting data using a variety of techniques.

Symmetric encryption is a very efficient mechanism of encrypting/decrypting data. In symmetric key encryption, the sender and receiver share the same secret key. The sender encrypts the data using the key to generate the cipher text and the receiver decrypts the cipher text using the same key to get the original data. This is shown below:

$$C = f(M)_K$$

$$M = F^{-1}(C)_K$$

In the above snippet `f` is the encryption/decryption algorithm, `M` is the clear text, `C` is the cipher text and `K` is the key used for encryption. This is illustrated in the following diagram:



Even though, symmetric encryption is fast and efficient, there is a security concern regarding the exchange of secret keys. Generally, first the secret key has to be exchanged using more secure methods such as asymmetric encryption, since if the key is somehow discovered then transmissions are no longer secure. The key is then used for symmetric encryption to exchange data. This is the technique used in HTTPS, which is a secure version of HTTP that uses SSL. HTTPS and SSL are explained in detail in a later section.

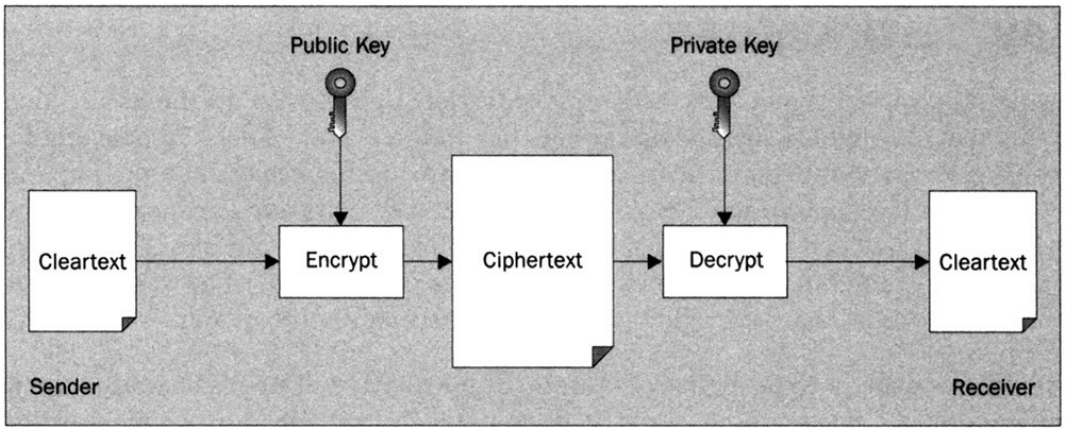
Asymmetric Encryption

Asymmetric Encryption also known as Public Key Cryptography is a more secure way of encrypting/decrypting data. Here instead of using a single key, a pair of keys is used. The owner of the pair keeps one of the keys and distributes the other key to the people with whom they wish to exchange data. The first key is called the private key and the second one is called the public key. When someone sends data to the owner of the key pair, they encrypt it with the public key. Now the encrypted data can be decrypted only using the owner's private key. It is of utmost importance that the owner of the key pair should ensure that the private key is stored securely as the integrity of the private key is critical. This is illustrated below:

$$C = f(M)_{\text{PUBLIC KEY}}$$

$$M = f^{-1}(C)_{\text{PRIVATE KEY}}$$

In the above snippet `f` is the encryption/decryption algorithm, `M` is the clear text and `C` is the cipher text. Even though there exists a mathematical relation between the keys, it is computationally impractical to deduce one from the other. This is illustrated in the following diagram:

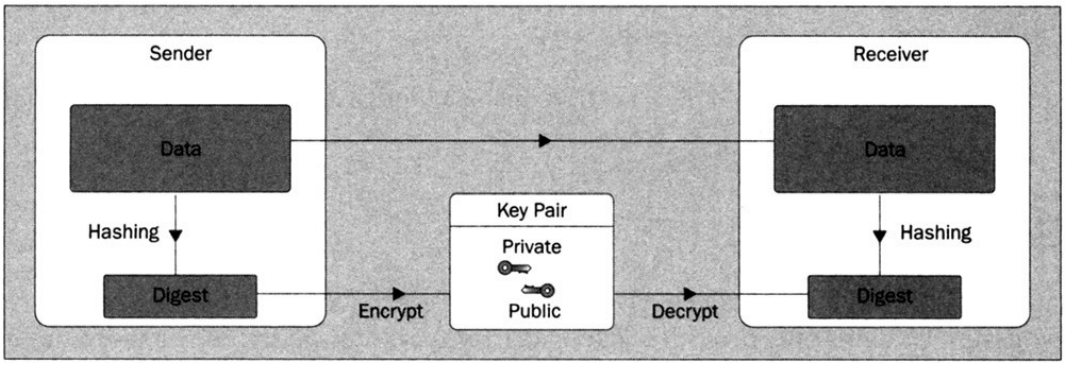


Asymmetric encryption though more secure is slow and computationally expensive. Hence it is normally used for exchanging the secret key used for bulk encryption.

The RSA (Rivest, Shamir, and Adleman) algorithm is a popularly used algorithm for public key encryption.

Digital Signatures

Digital signatures use public key cryptography for ensuring data integrity. However, here the private key is used for encrypting the data and the public key for decrypting it. However, as encrypting the whole data using the private key is slow and computationally expensive, first a message digest is created using hashing and then the message hash is encrypted using the private key. This is illustrated in the diagram shown below:



The sender creates the message digest, encrypts it with the private key, and sends the data and the encrypted digest to the receiver. The receiver decrypts the encrypted digest using the public key and compares it against the digest generated from the data that is received. Someone who sits in the middle can't tamper with encrypted digest unless they have the private key of the sender. If they tamper with the data, the receiver can easily identify it by comparing the digests.

Please note that digital signatures ensure data integrity and not confidentiality.

Public Key Certificates

In the last section on digital signatures, the owner of the key pair distributes the public key to the individuals that would originate secure correspondence to the owner of the key pair. Public key certificates are used to ensure the authenticity of the public keys. These certificates include the public key, information about the owner, and the expiration date of the certificate. Key pairs and certificates can be generated by tools such as `keytool`, which comes with the JDK. Certificates generated like this are called self-signed certificates as they are generated by the owner and there is no information to warranty that the certificate received by a receiver is in fact the original certificate generated by the owner.

We will explain this using a hypothetical example. Suppose Meeraj needs to send a document to Sean securely. He generates a message digest of the document, signs it with his private key and sends the document, signature and public key certificate to Sean. However, Henry intercepts the data in the middle, amends the document, generates a message digest, signs it with his private key, and sends the new document, signature and *his* public key certificate to Sean claiming that it is from Meeraj. Sean will verify the amended data successfully using the new public key certificate, unaware of what happened in the middle.

This is where certificate authorities such as Verisign come into the picture. So our hypothetical situation changes. Let's say that Sean already has the public key of Andre whom he trusts. So Meeraj gets his certificate digitally signed by Andre before sending it to Sean. Now Sean can verify that the certificate originally belongs to Meeraj as it is signed by Andre. Here Andre takes the role of a Certificate Authority (CA).

Normally recipients keep a list of root CA certificates for verifying public key certificates. For example, JDK comes with a file called `cacerts`, which contains five Verisign root CA certificates. Normally, instead of having one CA, there will be a hierarchy of CAs. For example, Craig may sign the public key of Andre and Nicola may sign Craig's public key.

The international standard for public key certificates is called X.509 and public key certificates are often referred to as X.509 Certificates.

Secure Socket Layer

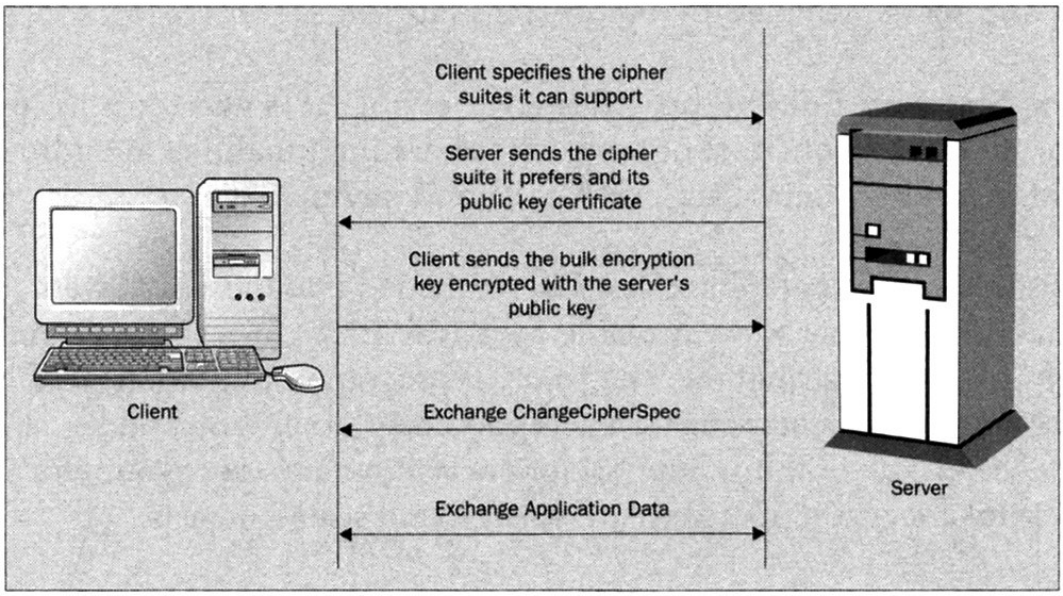
Secure Socket Layer, or SSL, is a transport-dependent protocol originally proposed by Netscape for the secure transmission of data. SSL provides data integrity, confidentiality, authentication, and non-repudiation. Data confidentiality is ensured using encryption, and integrity by using hashing functions. Clients can authenticate the server and the server can optionally authenticate the client. Non-repudiation is provided using digital signatures and public key certificates.

SSL contains a record protocol and a handshake protocol. The record protocol defines the format of the messages exchanged between the client and the server. The communication between the client and the server is dictated by a set of parameters called a cipher suite. The information defined by the cipher suite includes following details:

- The key exchange algorithm
- The encryption algorithm
- The digest algorithm

When the connection is first established the cipher suite will be `SSL_NULL_WITH_NULL_NULL`. For example, `SSL_RSA_WITH_RC4_128_MD5` means it will use RSA key exchange algorithm, MD5 message digest algorithm and an RC4 key encryption algorithm. Please note that some of the strong encryption algorithms are subject to export regulations in the United States.

During the handshake protocol depicted in the following diagram, the server and client exchange a series of messages to decide on the cipher suite and a key that will be used for bulk encryption prior to the actual data exchange. Please note that SSL uses public key encryption only to exchange a key that will be used to bulk encrypt/decrypt the original data to improve efficiency:



In the above diagram we can see that:

- The client connects to the server and sends the cipher suites it can support.
- The server picks the strongest cipher suite that it can support and sends it back to the client along with the server's public key certificate. This public key may be signed by a root CA. The client verifies the certificate against its root CA database.
- Then it generates a key that will be used for bulk encryption, signs it with the server's public key, and sends it to the server. If the key is not already in use, the client and server exchange the `ChangeCipherSpec` message confirming the cipher suite. If the key generated by the client is already in use, the server informs the client and the client will generate a different key.
- The client and server then exchange data using bulk encryption using the generated secret key.

HTTPS is secure way of using HTTP over SSL.

SSL and Java

Normally when using HTTP between a server and client browser such as Internet Explorer, the browser software will take care of the intricacies of SSL such as verifying the public key certificate of the server against the root CA database, generating the bulk encryption key, etc. However, if we are writing Java applications that use HTTPS to connect to secure servers, we can't use the plain sockets available with the JDK. For this we should install **JSSE (Java Secure Socket Extension)** from Sun, which is an implementation of SSL. Java sockets use a factory-based approach, and their behavior can be controlled by setting system properties without making a lot of changes to the client code. JDK 1.4, however, is bundled with JSSE so it is not necessary to install JSSE separately.

In a later section we will look at using JSSE for making our stock quote service available through HTTP and enabling the `GetQuote` client to access it without making any changes.

Java and Cryptography

Most of the web services security specification relies on XML security specifications such as XML digital signature and encryption. These in turn depend on traditional cryptographic concepts. Most of the Java implementations of these specifications therefore rely on the cryptographic tools and API provided by the Java 2 Platform, Standard Edition. Hence it would be a good idea to have an overview of the security API and tools provided by the Java 2 Platform.

The Java 2 Platform, Standard Edition, provides a variety of APIs and tools to support cryptography. The security API provides a whole host of cryptographic functionalities including hashing, encryption/decryption, digital signatures, certificate, and key management.

In pre-JDK 1.4 releases, the cryptographic classes were divided into those that were available with the standard distribution and those that were available with **JCE 1.2.1 (Java Cryptographic Extension)**. JDK 1.4 bundles JCE with the standard distribution. The Java Cryptography Extension (JCE) is a set of packages that provide a framework and implementations for encryption, key generation and key agreement, and Message Authentication Code (MAC) algorithms. Support for encryption includes symmetric, asymmetric, block, and stream ciphers. The software also supports secure streams and sealed objects.

Cryptographic classes in the core distribution provide the following functionalities:

- **Message digests**
  Used for generating hashes of specified data
- **Key pair generation**
  Used to generate private-public key pairs
- **Digital signatures**
  Used to generate and verify digital signatures
- **Key factories**
  Used for key creation from key specifications
- **Certificate factories**
  Used to create public key certificates
- **Keystore management**
  Used for managing keystores containing private keys, certificates, and so on
- **Secure random/pseudo-random number generation**
  Used to generate random and pseudo-random numbers

The JCE classes provide the following functionalities:

- **Asymmetric and symmetric ciphers**
  Used for public key and bulk encryption
- **Secret key factories**
  Factories for secret keys
- **MAC algorithms**
  Used for message authentication codes
- **Key agreement protocol**
  Provides functionality for key agreement protocol

Two tools that come with J2SE standard installation that are very useful in providing cryptographic functionalities are `keytool` and `jarsigner`, which are available in the `\bin` directory of the JDK installation. For example, `keytool` can be used for:

- Creating key pairs
- Creating self-signed certificates
- Issuing CSR (Certificate Sign Requests) to CAs
- Importing public key certificates for verification

- Importing trusted certificates from root CAs to `cacerts`

Java Cryptography Provider Architecture

The **Java Cryptographic Architecture (JCA)** meets the following objectives:

- Provider independence and interoperability
- Algorithm independence

As we have already seen, there are several industry-standard algorithms to provide the same cryptographic functionality. For example, we can use either SHA or MD5 for generating message digests. Similarly there can be a number of implementations that support the MD5 algorithm. For example, the JCA provides an API that is independent of both algorithm and implementation.

JCA allows us to plug in new implementations without having to modify our code; all that we need to do is to edit the `java.security` file in the `\lib\security` directory of the JRE installation. In the JCA vocabulary, these implementations are called providers. The standard JDK distribution from Sun supports the following algorithms:

- Digital Signature Algorithm (DSA)
- Implementations of MD5 and SHA-1 message digest algorithms
- DSA key pair generator DSA algorithm parameter generator
- Certificate factory for X.509 certificates

The Sun provider is registered in the file `java.security` as shown below:

```
    security.provider.1=sun.security.provider.Sun
```

The literal `1` after `security.provider` indicates the.order in which the providers are searched for a specific algorithm implementation. For example, if we have the following providers registered:

```
    security.provider.1=sun.security.provider.Sun
    security.provider.2=com.ibm.crypto.provider.IBMJCE
```

and we ask for the MD5 algorithm for message digests, first the Sun provider is asked for an implementation and if it doesn't support it, the IBM JCE provider is asked. Please note that whenever we install a new provider we should make the provider classes available to the classloader. Generally having the classes as installed extensions does this.

We can also manage the providers dynamically by using the `addProvider()` and `insertProvidetAt()` methods on the class `java.security.Security`. Please note that providers added in this manner are not persistent. The method `getProviders()` can be used to get a list of all installed providers.

Engine Classes

In the last section we have seen how JCA provides implementation independence using the provider architecture. In this section we will have a look at how algorithm independence is achieved for various cryptographic functionalities. Algorithm independence is achieved in JCA using engine classes that provide specific cryptographic functionalities. The core engine classes bundled with the JDK standard API are:

- `java.security.MessageDigest`
  Used for providing hashing functionality
- `java.security.Signature`
  Used for signing and verifying digital signatures
- `java.security.KeyPairGenerator`
  Used for generating public and private keys
- `java.security.KeyFactory`
  Used for converting opaque cryptographic keys into key specifications
- `java.security.cert.CertificateFactory`
  Used for creating public key certificates
- `java.security.KeyStore`
  Used for managing keystores, which are used in Java for storing certificates, key pairs, and so on
- `java.security.AlgorithmParameters`
  Used for managing parameters to various cryptographic algorithms
- `java.security.AlgorithmParameterGenerator`
  Used for generating parameters specific to various algorithms

- `java.security.SecureRandom`
  Used for generating random numbers

The engine classes that come with JCE are:

- `javax.crypto.Cipher`
  Used to provide encryption and decryption functionality
- `javax.crypto.KeyAgreement`
  Used to provide functionality for key exchange protocol
- `javax.crypto.KeyGenerator`
  Used for symmetric key generation
- `javax.crypto.Mac`
  Used to provide MAC algorithm
- `javax:crypto.SecretkeyFactory`
  Used as a factory for secret keys

These classes provide static `getInstance()` methods to instantiate their objects, and take the algorithm name as their arguments. For example if we want to perform a hashing function using the MD5 algorithm, we will use the `MessageDigest` engine class as follows:

```
    MessageDigest md = MessageDigest.getInstance ("MD5");
```

This will ask each of the installed providers for a message digest engine class that supports the MD5 hashing algorithm in order of preference and return an appropriate instance. If a provider that supports the required algorithm is not found an exception is thrown.

Java Cryptographic Tools and Files

In this section we will have a look at the various files and tools used by the Java 2 SDK for providing cryptographic functionality. The main files used by the Java 2 SDK for providing cryptographic functionalities are listed below:

- `keystore`
  As we mentioned earlier, this is a database that stores private keys and certificate chains used to authenticate their public keys. Keystores can be of different formats, depending on the providers we use. The default Sun provider uses a flat-file format called JKS (Java Keystore). Keystores are protected by passwords and each private key in the keystore is protected by its own password.

  Keystores can be manipulated using the engine class `java.security.KeyStore` that is available in JCA.

  It can also be manipulated using the `Keytool` utility that comes with JDK. By default the `keystore` is stored in the user's `home` directory. The implementation and location of the keystores are specified in the `java.security` and `java.policy` files in the `\lib\security` directory of the JRE respectively.

- `cacerts`
  This is a `keystore` file that contains root CA certificates that can be used for authenticating certificate chains. JDK ships with five Verisign root CA certificates in the `cacerts keystore`. This is located in the `\lib\security` directory of the JRE.

- `security.properties`
  The security properties, such as installed providers and keystore type, are defined in the file `java.security` stored in the `\lib\security` directory of the JRE.

In this section we have seen a high-level overview of the cryptographic functionalities provided by the Java 2 Platform. In the subsequent sections we will see how these things fit into the broader world of web services security.

Accessing a Web Service Over HTTPS

In this section we will have a look at how to make the stock quote service accessible through HTTPS. We will be using Tomcat and Axis to deploy the web service, but before we can make our web service accessible through HTTPS we need to enable HTTPS on Tomcat.

Enabling HTTPS on Tomcat

We need to perform the following steps to enable HTTPS on Tomcat:

- Install JSSE (Please note that you don't need to do this if you are using JDK 1.4)
- Create the key pair
- Make configuration changes on Tomcat

Try It Out: Install JSSE

JSSE can be downloaded from the Javasoft web site. The latest version is 1.0.3. Please follow the following steps to install JSSE:

1. Unpack the downloaded ZIP file to your local drive. This will create a directory called `jssel.0.3` that will contain a `lib` directory. The `lib` directory will contain the files `jcert.jar, jnet.jar` and `jsse.jar`.

2. We will install these files as installed extensions. To do this, copy these files to the `lib\ext` directory of your JRE.

3. Now we will register the JSSE provider in the `java.security` file in the `lib\security` directory of the JRE by adding the following entry. Depending on the number of registered providers the numeric following the string `security.provider` will vary:
   `security.provider.3=com.sun.net.ssl.internal.ssl.Provider`

Create Certificates and Key Stores

In the section on SSL, we saw that the servers use public key encryption for key exchange. Before we can use Tomcat with HTTPS, we need to generate the keystore and certificate for Tomcat in order to use SSL. By default Tomcat looks for the key by alias `tomcat` in the default keystore called `.keystore` in the user's home directory. The command below will generate the key pair and public key certificate after you have answered the questions it asks in order to generate your certificate (note that you will have to add a store password and a key password when prompted):

```
keytool -genkey -alias tomcat
```

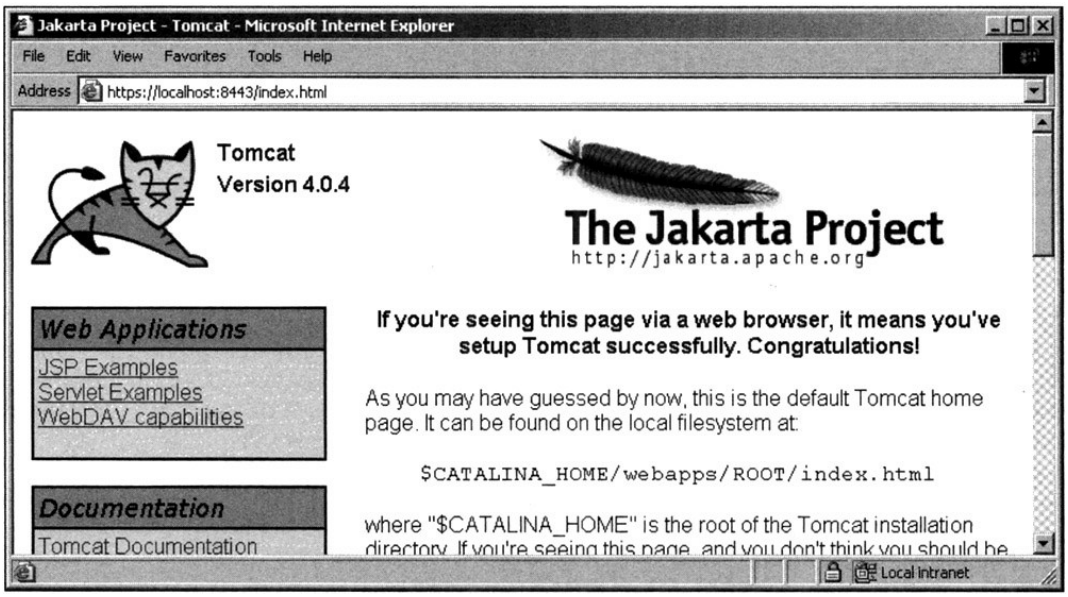
This will generate a key called `tomcat` in the default keystore with a self-signed certificate. Normally we would want get our public key certificates signed by a CA and import them to our keystore. However, here we will use the self-signed certificate. This means that a client browser will pop a message saying the certificate is untrusted; we can click OK and continue as normal.

Try It Out: Configure Tomcat

1. To enable HTTPS on Tomcat you need to add the following entry into the `server.xml` file in the `conf` directory. This should be added under the Tomcat standalone service in the configuration file. Normally this entry is commented in the configuration file and we just need to uncomment it. However, we should specify the keystore password:

```
    <Connector
      className="org.apache.catalina.connector.http.HttpConnector"
      port="8443"
      minProcessors="5"
      maxProcessors="75"
      enableLookups="true"
      acceptCount="10"
    debug="0"
    scheme="https"
    secure="true">
    <Factory
      className="org.apache.catalina.net.SSLServerSocketFactory"
      clientAuth="false"
      protocol="TLS"
      keystorePass="password"/>
</Connector>
```

2. To test whether SSL is configured properly we can access the URL https://localhost:8443. Please note that we use https instead of http and the port number is 8443. The browser will display the Tomcat home page as shown below:



The padlock sign on the right shows that the site is accessed using HTTPS.

Try It Out: Web Services and HTTPS

In this section we will deploy our web and access it using HTTPS.

1. To deploy the web service, store the contents shown below in a file called `ServerDeploy.wsdd`:

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

  <service name="StockQuoteService" provider="java:RPC">
    <parameter name="className" value="com.wrox.jws.stockcore.StockCore"/>
    <parameter name="allowedMethods" value="getQuote"/>
  </service>

</deployment>
```

   Make sure that you have all the Axis JAR files and a JAXP parser in the classpath as explained in the earlier section and run the following command to deploy the web service:

```
java org.apache.axis.client.AdminClient
     -1http://localhost:8080/axis/services/AdminService ServerDeploy.wsdd
```

2. Since we have already compiled the client class `GetQuote.java` in an earlier section of this chapter, we can access the web service by running the following command. Please make sure that you have Axis JAR file, a JAXP parser, and the current directory in the classpath:

```
java -Djava.protocol.handler.pkgs=com.sun.net.ssl.internal.www.protocol
     -Djavax.net.ssl.trustStore=c:\Beginning_JWS_Examples\Chp09\.keystore
     -Djavax.net.ssl.trustStorePassword=password
  com.wrox.jws.stockcore.GetQuote IBM https://localhost:8443/axis/services/
```

This will access the stock quote service and print the quote. Bear in mind that you will have to alter this command to suit your setup. For example, your keystore may be held in `C:\Documents and Settings\<username>` if you are not the only user on your machine; the password will also need to be changed, and the path to `GetQuote` if it is not the same.

How It Works

Deploying the web service is the same routine stuff we have seen in earlier chapters. The main difference here is how we access the web service. When we invoke the client class, we pass the symbol and the endpoint URL. This URL is mapped to the Axis servlet in the Axis `web.xml` file. However, the main difference is that the endpoint URL now uses HTTPS instead of HTTP.

If we simply use HTTPS, our client will throw an exception stating HTTPS is an unrecognised protocol. The system property `java.protocol.handler.pkgs` is used to force the use of secure sockets instead of normal sockets.

What is the use of `javax.net.ssl.trustStore` and `javax.net.ssl.trustStorePassword` system properties you may ask? As mentioned earlier, Tomcat is using a self-signed certificate instead of one signed by a root CA. However, when the client-side JSSE framework receives the public key certificate from Tomcat it will look at the `cacerts` file for verification. It won't find anything in the `cacerts` file to verify the certificate and will throw an exception. Hence we are using the aforementioned properties to look into the default keystore for the trusted certificate.

Important In this example both the client and server use the same keystore. In a real-life scenario, you will have to export the server's public key and import it to the client's keystore. In this case the trust store should be set to the client's keystore.

HTTP Authentication Methods

Even though the web services specification doesn't mandate HTTP as the transport protocol, HTTP is still the most popularly used transport protocol in the web services world. In this section we will have a look at the various methods available within HTTP to provide transport level point-to-point security and how the J2EE servlet specification supports these authentication methods. HTTP provides the following authentication methods for a client to authenticate a server.

Basic Authentication

This is the method specified in the HTTP 1.1 specification and is based on a user name and password. In this method, the HTTP server requests the client to authenticate itself. As part of this challenge, the server sends a string identifying the realm to which the client is to be authenticated. In J2EE web applications this realm name is defined in the web deployment descriptor.

In basic authentication the user name and password are transmitted from the client to the server in simple base64 encoding, which is fundamentally insecure as it doesn't use any strong encryption schemes and it doesn't provide any mechanism for the client to authenticate the server.

Digest Authentication

This is very similar to basic authentication and the only difference is that the credential information (the user ID and password) is transmitted in an encrypted manner and hence is more secure.

HTTPS Client Authentication

This is a highly secure authentication scheme using HTTPS. This requires the client to possess the public key certificate corresponding to the private key owned by the server.

Servlet Support for HTTP Authentication Methods

The servlet specification supports all of the aforementioned authentication methods. It adds in a new form-based authentication method to control the look and feel of the screen used by the client browser for obtaining the users' security credentials.

The servlet specification defines role-based security where a collection of web resources is associated with specific roles. These roles are mapped to physical users using a servlet container-specific technique.

The security policies for the web applications that act as web services can be controlled using the web deployment descriptor for the web application. The example below shows the deployment descriptor excerpt that allows normal users to view the WSDL file and only special users to invoke the web service by defining different security policies for the WSDL and endpoint URLs:

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

<security-role>
  <role-name>normal</role-name>
</security-role>

<security-constraint>

  <web-resource-collection>
    <web-resource-name>WSDL URL</web-resource-name>
    <url-pattern>/MyWebService/ws.wsdl</url-pattern>
  </web-resource-collection>

  <auth-constraint>
    <role-name>normal</role-name>
    <role-name>special</role-name>
  </auth-constraint>

</security-constraint>

<security-constraint>

  <web-resource-collection>
    <web-resource-name>Endpoint URL</web-resource-name>
    <url-pattern>/MyWebService/Service</url-pattern>
  </web-resource-collection>

  <auth-constraint>
    <role-name>special</role-name>
  </auth-constraint>

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

</security-constraint>

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

In the above example, the WSDL URL is accessible to users defined with special and normal roles whereas only special users can invoke the web service. Further to this, the transport for web service invocation should guarantee data confidentiality. The possible values for the `transport-guarantee` element are `NONE`, `INTEGRAL`, and `CONFIDENTIAL`. The authentication method used is `CLIENT-CERT`.

Component-Level Security

Even if we rely on transport-level security, it is always better to put belt and braces on our web services' security by securing the back-end components that implement the web services. In the J2EE world, if we are using EJBs to implement the web services, we can use the method

permission element in the EJB deployment descriptor to provide role-based access to the EJB methods as shown below:

```
<assembly-descriptor>

  <security-role>
    <role-name>special</role-name>
  </security-role>

  <method-permission>
    <role-name>special</role-name>
    <method>
      <ejb-name>WebServiceEJB</ejb-name>
      <method-name>webServiceMethod</method-name>
    </method>
  </method-permission>

</assembly-descriptor>
```

The deployment descriptor excerpt shown above states that the `webServiceMethod()` on `WebServiceEJB` can be invoked only by authenticated principals with the role `special`.

Try It Out: An Authenticated StockQuote Service

In this section we will have a look at how to allow access to our stock quote web service to only authenticated users.

As we saw in the earlier example, the web service is accessed through the URL https://localhost:8443/axis/services/. The non-HTTPS equivalent of this is http://localhost:8080/axis/services. If we look at the deployment descriptor for the axis web application we see that this URL is mapped to the Axis servlet. So one obvious way to restrict access to the web service is to restrict this URI. However, this will restrict access to all other web services as well as the admin service for deploying web service. Hence we will create a new URI for accessing the stock quote service and restrict that one only.

1. First we will modify the Axis web deployment descriptor to apply our new security policies. This file can be found in the `%axisDirectory%/webapps/axis/WEB-INF` directory and is called `web.xml`. Add the following changes to the deployment descriptor:

   ```
   <servlet-mapping>
     <servlet-name>AxisServlet</servlet-name>
     <url-pattern>/services/secure</url-pattern>
   </servlet-mapping>

   <security-constraint>
     <web-resource-collection>
       <web-resource-name>Secure resources</web-resource-name>
       <url-pattern>/services/secure</url-pattern>
       <http-method>GET</http-method>
       <http-method>POST</http-method>
     </web-resource-collection>

     <auth-constraint>
       <role-name>ws_user</role-name>
     </auth-constraint>

   </security-constraint>

   <login-config>
     <auth-method>BASIC</auth-method>
   </login-config>
   ```

   The listing above states that the `BASIC` authentication will be used for authenticating users and for accessing the URI `/services/secure` we should have the role `ws_user`.

2. Now that we have defined our security policy, we need to create users and roles that map to our security policy. By default Tomcat uses a memory realm, which is loaded from the file `tomcat-users.xml` available in the `conf` directory. Add the following entry to create a new user with the required role:

   ```
   <user name="meeraj" password="chelsea" roles="ws_user" />
   ```

   Now restart Tomcat for our new security policies to take effect.

3. To access the web service we can use the same client as in the HTTPS example. However we will pass different command-line arguments as shown below:

   ```
   java com.wrox.jws.stockcore.GetQuote IBM
       http://localhost:8080/axis/services/secure meeraj chelsea
   ```

Please note that here we can use either the HTTP or HTTPS URLs. If we are using the HTTP URL we don't need to set all the system properties specific to HTTPS. However, for both cases we should specify the user name and password. Also note that now we use the secure URL instead of the earlier URL. This will access the web service and print the quote.

If you pass an incorrect password or user name, you will get a SOAP fault with an HTTP code 401 stating unauthorized access.

How It Works

The first thing we did was to use standard J2EE web application techniques to create a secure URI that is mapped to the Axis servlet. We also specified that only users with `ws_user` role might access this web service as shown in the following deployment descriptor:

```
<security-constraint>

  <web-resource-collection>
    <web-resource-name>Secure resources</web-resource-name>
    <url-pattern>/services/secure</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>

  <auth-constraint>
    <role-name>ws_user</role-name>
  </auth-constraint>

</security-constraint>
```

Then we created a user with the required role in the Tomcat memory realm by editing the `tomcat-users.xml` file. So, when we invoked the client we specified the restricted URL and also specified the user name and password.

The client code uses the `call` object for setting the user name and password:

```
if(args.length == 4) {
    call.setUsername(args[2]);
    call.setPassword(args[3]);
}
```

The Axis client-side framework will actually set this information in the HTTP header so that the servlet container can extract it and use it for authentication and authorization purposes.

Custom Authentication Methods

In the last example we saw how to use standard HTTP authentication methods for providing restricted access to our web service. However, there may be scenarios where we want to implement more powerful authentication logic. For example, we may want to let a particular user access a particular web service only a fixed number of times a day. There is no way we can achieve this using standard HTTP authentication techniques.

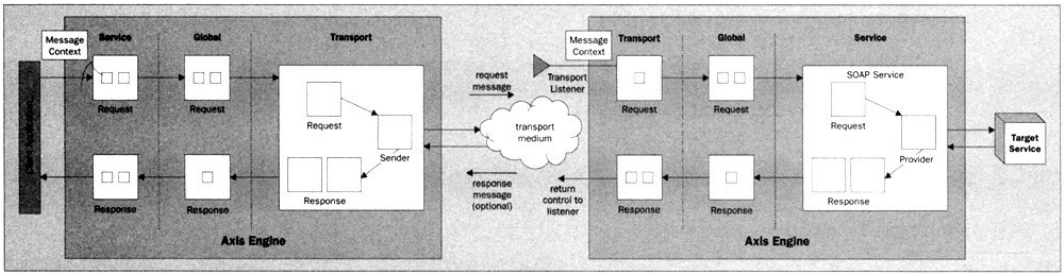
Most of the web services platforms such as Axis, WSTK, WLS, and so on provide powerful functionality called handlers for pre and post processing SOAP messages during the web services invocation. These components use the Decorator pattern and work in a similar way to servlet filters. These components can be used for a variety of purposes such as metering, auditing, logging, encryption, signatures, and authentication.

Axis Handler Architecture

Axis handlers can be used for processing the following:

- A SOAP request before it leaves the client
- A SOAP request before it reaches the server
- A SOAP response before it leaves the server
- A SOAP response before it reaches the client

The diagram below taken from the Axis documentation shows a view of the Axis handler architecture:



Handlers can be configured both on the client and/or the server side. They can be configured for a transport, globally for all the services, or for a specific service. The handlers implement the interface `org.apache.axis.Handler`, which defines a variety of methods. The main method is the `invoke` method that passes a reference to the message context. The message context can be used for a variety of purposes such as accessing the SOAP envelope, or the security credentials for the current thread.

Important A detailed coverage of Axis handler architecture is beyond the scope of this chapter. Please refer to the Axis architecture documentation available with the Axis installation bundle for more details.

Axis comes with two handler implementations called `SimpleAuthenticationHandler` and `SimpleAuthorizationHandler` that illustrate the use of handlers for implementing custom authentication logic. These handlers use two files called `users.1st` and `perms.1st`, available in the `WEB-INF` directory of the Axis web application. In the following example, we will use these handlers to provide restricted access to the stock quote service.

Try It Out: Custom Authentication

1. To deploy the web service, store the contents shown below in a file called `ServerDeploy.wsdd`:

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

  <service name="StockQuoteService" provider="java:RPC">
    <parameter name="className" value="com.wrox.jws.stockcore.StockCore"/>
    <parameter name="allowedMethods" value="getQuote"/>
    <parameter name="allowedRoles" value="meeraj"/>

    <requestFlow name="checks">
      <handler
        type="java:org.apache.axis.handlers.SimpleAuthenticationHandler"/>
      <handler
        type="java:org.apache.axis.handlers.SimpleAuthorizationHandler"/>
    </requestFlow>
  </service>

</deployment>
```

2. To deploy the web service, make sure that you have all the Axis JAR files and a JAXP parser in the classpath as explained in the earlier section and run the following command to deploy the web service:

```
java org.apache.axis.client.AdminClient
    -lhttp://localhost:8080/axis/services/AdminService ServerDeploy.wsdd
```

3. We need to edit the `users.1st` file in the `WEB-INF` directory of the Axis web application and add the following entry (or your own name and password depending on how you set things up in the `ServerDeploy.wsdd` file):

```
user 1 pass1
user2
user3 pass3
meeraj chelsea
```

4. To access the web service we can use the same client as in the HTTPS example. However we will pass different command-line arguments as shown below:

```
java com.wrox.jws.stockcore.GetQuote IBM
     http://localhost:8080/axis/services/ meeraj chelsea
```

Please note that here we can use the normal URL for accessing the service. This will access the web service and print the quote. If we pass an incorrect password or user name, we get a SOAP fault stating unauthorized access.

How It Works

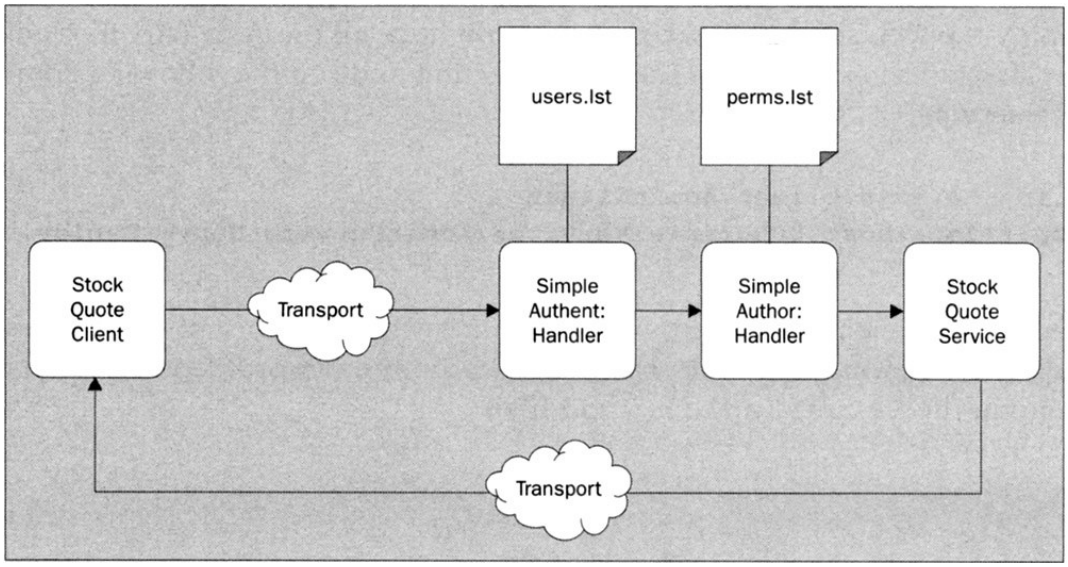
The above example is very similar to the one that uses basic authentication but has a few differences. The main difference is in the deployment descriptor where we specify only the user `meeraj` can access the stock quote web service:

```
<parameter name="allowedRoles" value="meeraj"/>
```

Then we configure a set of handlers that are invoked before the web service as shown below:

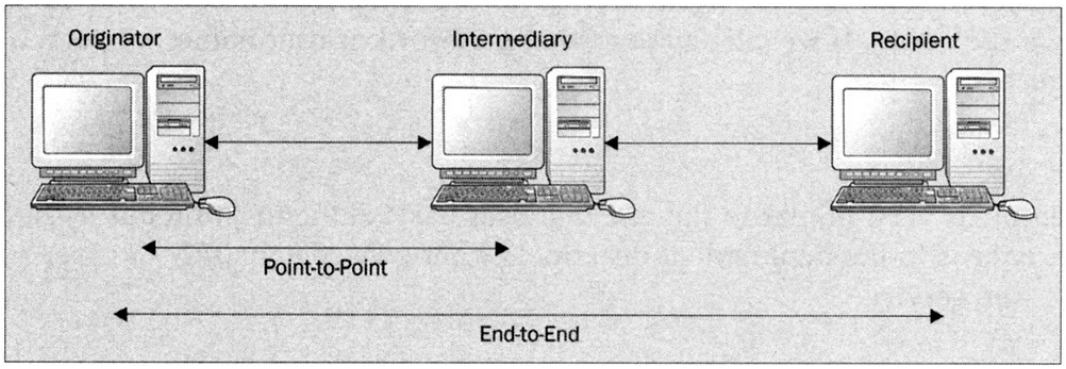
```
<requestFlow name="checks">
  <handler
    type="java:org.apache.axis.handlers.SimpleAuthenticationHandler"/>
  <handler
    type="java:org.apache.axis.handlers.SimpleAuthorizationHandler"/>
</requestFlow>
```

The snippet above states that the authentication and authorization handlers should be invoked before accessing the web service. These handlers will extract the user name and password set by the client from the message context, and compare them against the information stored in the `users.1st` and `perms.1st` files. If they don't match, a SOAP fault authentication failure is thrown. This is illustrated in the diagram shown overleaf:



Shortcomings of Traditional Security Methods

The traditional security solutions we have discussed so far in this chapter provide an efficient point-to-point security at transport level in exchanging information. However, in the web services world we are really talking about end-to-end security. A SOAP envelope that represents a web service request may go through a number of intermediaries before it reaches the ultimate recipient:



As mentioned in the beginning of this chapter, one of the most important aspects of XML is its ability to represent hierarchical data structures and to navigate to arbitrary data elements using DOM, XPath, and so on. Once the whole document is encrypted, this powerful aspect of XML is lost as the whole XML structure is now converted into a binary chunk of data.

Several organizations have been working actively on extending the traditional security solutions, to meet the various aspects of XML security requirements. Due to their efforts, a number of specifications have evolved over the last couple of years. These include:

- Security Assertion Markup Language from OASIS (http://www.oasis.org)
- XML Digital Signatures from the W3C (http://www.w3c.org)
- XML Encryption from W3C the (http://www.w3c.org)

- XML Key Management from the W3C (http://www.w3c.org)
- WS-Security from IBM, Microsoft, and Verisign

In the next section we will have a look at the various XML security specifications from W3C and the XML security specifications implementation from IBM – XML Security Suite. Most of these specifications build on the traditional security techniques such as digital signatures, message digests, and asymmetric and bulk encryption.

XML Security Specifications

In this section we will mainly cover XML Signature and XML Encryption standards. Please note that the more specific web services security specifications such as WS-Security and the security assertion specifications such as SAML utilize the functionality offered by the XML security specifications.

Canonical XML

Before we delve into the details of XML signature and encryption, we will have a quick overview of **canonical XML**. As we have seen in previous sections, signing the message digest of the data with the owner's private key generates digital signatures; mainly for checking data integrity and non-repudiation.

Similarly public key encryption involves encrypting the data with the recipient's public key and the message recipient, in turn, using the private key to decrypt the encrypted message. We have also seen that a small change in the original message will cause the message digest algorithm to generate an entirely different hash value.

In the context of XML documents, two documents can have the same content even if their textual representations are not the same. A simple example would be where the second document is formatted differently from the first. Hence if we generate message digests for the two documents, they will result in different hash values.

The canonical XML specification defines a method of generating XML documents in a form that identifies logically identical documents as having the same content despite the variations in physical formatting. Both XML signature, and encryption, work on the canonical form of the documents, so that the generated hash values are the same for two documents with identical logical structure, even if they vary in physical structure.

XML Signature

XML signature is a W3C recommendation describing the creation and representation of digital signatures in XML. The signatures may represent any arbitrary XML and non-XML data. They can be detached from the data that is signed, or in the case of XML, data being signed can be part of the XML document. If the signature is not detached it can either envelope the data that is signed or can be enveloped by the data that is signed. Basically, XML signature defines a way of associating keys with data that is signed.

XML Signature Syntax

XML digital signatures use references to link signature values to signed data. The data that is signed is first hashed and the hash value is placed in an element. This element is then digested and cryptographically signed.

XML digital signatures use the `Signature` element for representing signatures. The basic structure of the `Signature` element is shown below:

```
<Signature ID>
  <SignedInfo>
    <CanonicalizationMethod/>
    <SignatureMethod/>
    <Reference URI>
      <Transforms/>
      <DigestMethod/>
      <DigestValue/>
    </Reference>
  </SignedInfo>
  <KeyInfo>
  <Object ID/>
</Signature>
```

The `URI` attribute in the `Reference` element is used to link the signature to the signed data. If the signed data is available in the same document as the signature, the `URI` points to the `ID` attribute of the data that is signed.

- The `SignedInfo` element encapsulates all the information related to the signature.
- The `CanonicalizationMethod` element defines the algorithm used to canonicalize the XML data that is signed.
- The `SignatureMethod` identifies the algorithm used to sign the hash of the data that is signed. The algorithms supported include RSA-SHA1, and DSA-SHA1.

- The `Reference` element points to the data that is signed using the `URI` attribute and contains the digest algorithm that is used for hashing the data defined by the `DigestMethod` element and the digest value included in the `DigestValue` attribute. It may also contain one or more `Transform` elements identifying the processing applied to the data before it was digested.
- The optional `KeyInfo` element may be used to specify the key that should be used to validate the signature. The contents of this element vary depending on the signature algorithm that is used.

An XML Signature Example

The listing below shows the XML Signature that is generated from an XML document. The programmatic signing of the document to generate the signature and then the verification of the signature is covered in a later section:

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

The signature method that is used is DSA-SHA1:

```
    <emp:SignatureMethod
      Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1">
    </emp:SignatureMethod>
```

The `URI #1` points to the data that is signed:

```
    <emp:Reference URI="#1">
      <emp:Transforms>
        <emp:Transform
         Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315">
        </emp:Transform>
      </emp:Transforms>
```

The digest algorithm that is used is SHA1:

```
      <emp:DigestMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#shal">
      </emp:DigestMethod>
```

This is the digest value:

```
        <emp:DigestValue>ew6TRCmxglhpUeXvo6LgSy3cZlk=</emp:DigestValue>
      </emp:Reference>
    </emp:SignedInfo>
```

This is the signature value:

```
    <emp:SignatureValue>
      c4lNWEbbm/OB6YfNRi61I7jHiNMzM0gVQWSAggN9ForXMg4aoEusqw==
    </emp:SignatureValue>
    <emp:KeyInfo>
```

This element defines the public key value:

```
    <emp:KeyValue>
      <emp:DSAKeyValue>
        <emp:P>
          /X9TgR11EilS30qcLuzk5/YRt1I870QAwx4/gLZRJmlFXUAiUftZPY1Y+r/F9bow9s
          ubVWzXgTuAHTRv8mZgt2uZUKWkn5/oBHsQIsJPu6nX/rfGG/g7V+fGqKYVDwT7g/bT
          xR7DAjVUE1oWkTL2dfOuK2HXKu/yIgMZndFIAcc=
        </emp:P>

        <emp:Q>
          12BQjxUjC8yykrmCouuEC/BYHPU=
        </emp:Q>

        <emp:G>
          9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBxCBgLRJFn
          Ej6EwoFhO3zwkyjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTx
          vqhRkImog9/hWuWfBpKLZ16Ae1U1ZAFMO/7PSSo=
        </emp:G>

        <emp:Y>
          CrgJnP+zC6Zc5Aj1TA26vdJ+Ui/qR5x1OiYCpV6MSXuweMhWYH5DVr9te1O9hiWDG9
          vwHSkMvoo9Ub7nQ8xqBoLn/YIesho+7Vmua7EBepew5Crq1qgMOWZiay/1Jgs/BFg5
```

```
            AaKbXBMakMmDFzHaqrjID7eFDbVARouW+6XHnAg=
          </emp:Y>

        </emp:DSAKeyValue>
      </emp:KeyValue>
```

This contains the X.509 certificate information that contains the public key:

```
      <emp:X509Data>
        <emp:X509IssuerSerial>
          <emp:X509IssuerName>
            CN=Kunnumpurath\, Meeraj, OU=E-Solutions, O=EDS, L=Milton
            Keynes, ST=Bucks, C=UK
          </emp:X509IssuerName>
          <emp:X509SerialNumber>
            1026516428
          </emp:X509SerialNumber>
        </emp:X509IssuerSerial>

        <emp:X509SubjectName>
          CN=Kunnumpurath\, Meeraj, OU=E-Solutions, O=EDS, L=Milton
          Keynes, ST=Bucks, C=UK
        </emp:X509SubjectName>
        <emp:X509Certificate>
  MIIDJDCCAuICBD0vZcwwCwYHKoZIzjgEAwUAMHgxCzAJBgNVBAYTA1VLMQ4wDAYDVQQIEwVCdWNr
  czEWMBQGA1UEBxMNTW1sdG9uIEt1eW51czEMMAoGA1UEChMDRURTMRQwEgYDVQQLEwtFLVNvbHV0

  ...

  /YIesho+7Vmua7EBepew5Crq1qgMOWZiay/1Jgs/BFg5AaKbXBMakMmDFzHaqrjID7eFDbVARouW
  +6XHnAgwCwYHKoZIzjgEAwUAAy8AMCwCFBhQ48ng00bCmtgEDUb0ga2tT/QsAhQ9SWHM+3/WIzQ7
  8WXo7nnBWuSC/Q==
        </emp:X509Certificate>
      </emp:X509Data>
    </emp:KeyInfo>
```

This is the data that is signed, and referenced by the `Reference` element's `URI` attribute. The signed data is enclosed in the `Object` element:

```
      <dsig:Object xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" Id="1">
        <employee>
          <name>Claudio Ranieri</name>
          <department>Football Club</department>
          <company>Chelsea Village</company>
          <position>Manager</position>
          <salary>100000000</salary>
        </employee>
      </dsig:Object>
    </emp:Signature>
```

Reference Verification

The steps involved in reference verification are listed below:

- The contents of the `SignedInfo` element are canonicalized using the specified algorithm
- For each reference the data is obtained using the `URI` attribute
- This data is digested using the specified digest algorithm
- The digest is compared against the digest value specified in the `SignedInfo` element

The public key can be retrieved from a variety of sources:

- From an external source such as a `keystore`
- From the information present in the `KeyInfo` element

XML Encryption

XML encryption specifies a means of encrypting data and representing it in XML format. The data that is encrypted may be XML documents, elements, element content, or any arbitrary data. The encrypted information is enclosed in the `EncryptedData` element, which will replace

the element, or the element content, in the encrypted XML document. If the whole XML document is encrypted this element will become the document element.

The basic structure of the `EncryptedData` element is shown below:

```
<EncryptedData Id Type>
  <EncryptionMethod/>
  <ds:KeyInfo>
    <EncryptedKey/>
    <AgreementMethod/>
    <ds:KeyName/>
    <ds:RetrievalMethod/>
    <ds:*>
  </ds:KeyInfo>
  <CipherData>
    <CipherValue/>
    <CipherReference URI/>
  </CipherData>
  <EncryptionProperties/>
</EncryptedData>
```

- The `Type` attribute of `EncryptedData` element identifies whether the element, content, or any arbitrary data is encrypted.
- The `EncryptionMethod` element identifies the algorithm used to encrypt the data.
- The `KeyInfo` element can be used to hint which decrypting program to use to obtain the key to decrypt the data. This can be elements such as the plain key value (most unlikely), the encrypted key using `EncryptedKey` element, or the alias name identifying the private key of the owner in the case of asymmetric encryption.
- The `CipherData` element will contain the encrypted data.

An XML Encryption Example

The listing below shows an encrypted XML document, with a particular element encrypted:

```
<?xml version="1.0" encoding="UTF-8"?>
<employee>
  <name>Claudio Ranieri</name>
  <department>Football Club</department>
  <company>Chelsea Village</company>
  <position>Manager</position>
```

The encrypted element identifies the encrypted content as an XML element:

```
  <EncryptedData Id="ed2"
    Type="http://www.w3.org/2001/04/xmlenc#Element"
    xmlns="http://www.w3.org/2001/04/xmlenc#">
```

The algorithm used is RSA:

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

The `KeyInfo` element identifies the private key alias used to decrypt the data:

```
      <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
        <KeyName>MyKey</KeyName>
      </KeyInfo>
```

The `CipherData` element contains the encrypted data:

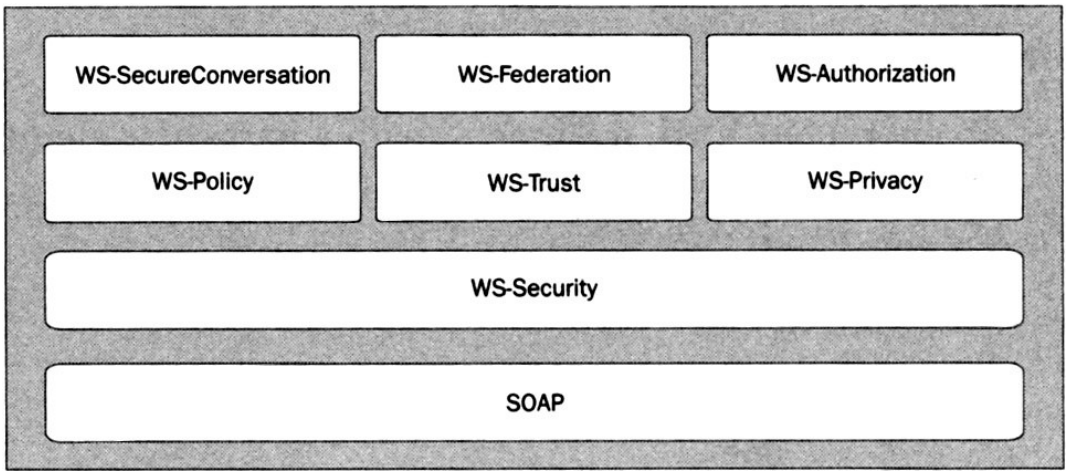
```
      <CipherData>
        <CipherValue>BQYjtLafz8a15C1aCZ1iFi9UY7EwZTPHUBe9VA+jT90kAo3nV
          NPUtVG0o1C0Ro6mowsj1EohGnM11qUHUZRShPr1+SVOr8vOIGFcYvE447Jifd
          1TWtvRfeQTezmt6gt7fuBNu3xieB9PtcKv0pdQ7zDCitjDz9YWhRLfRMn1jLI=
        </CipherValue>
      </CipherData>
    </EncryptedData>
  </employee>
```

WS-Security

WS-Security is the first among a set of security specifications initiated by IBM, Microsoft, and Verisign, which address specific security issues in the context of web services. These include authentication, authorization, privacy, trust, integrity, federation, secure communication channels, and auditing.

These specifications are built on top of the web services technologies such as SOAP and WSDL as well as the XML security specifications such as XML Signature and Encryption. These specifications address end-to-end security solutions in a federated web services world, rather than the point-to-point to transport-level security provided by traditional security solutions such as SSL.

The diagram below depicts the web services security specifications stack:



The only specification that is currently available is WS-Security; so let's look at the various blocks shown in the diagram above:

- **WS-Security**
  Defines a secure messaging model. This extends SOAP to add security-related headers to SOAP envelopes. This security-related information includes signed and unsigned security tokens such as PK certificates, Kerberos tickets, user name/password, as well as digital signatures and encryption headers defined in accordance with the XML Signature and Encryption specifications.
- **WS-Policy**
  Concentrates on the security policies and constraints defined by endpoint and intermediate players.
- **WS-Trust**
  Enables web services to interoperate by defining a framework for trust models.
- **WS-Privacy**
  Defines how web services and service consumers define privacy preferences.
- **WS-SecureConversation**
  Defines how to manage and authenticate message exchanges.
- **WS-Federation**
  Deals with trust relationships in a heterogeneous federated environment.
- **WS-Authorization**
  Deals with authorization information and policies.

As we mentioned, WS-Security extends the SOAP messaging structure to provide message integrity, confidentiality, and authentication. It also provides an extensible mechanism of attaching security tokens with SOAP messages. As well as this, it provides facilities to encode binary security tokens such as X.509 certificates and Kerberos tickets. The listing below shows a simple example of sending a user name and password in a SOAP envelope as defined by the WS-Security specification:

```
<s:Envelope xmlns:s="http://www.w3c.org/2000/12/soap-envelope">
  <s:Header>
    <wsse:Security
      xmlns:wsse="http://schemas.xml.soap.org/ws/2002/04/secext">
      <wsse:UserNameToken>
        <wsse:UserName>Meeraj</wsse:UserName>
        <wsse:Password>wroxauthor528</wsse:Password>
      </wsse:UserNameToken>
    </wsse:Security>
  </s:Header>
  <s:Body>
    ...
  </s:Body>
</s:Header>
```

Security Header Block

As we can see the `Security` element in the http://schemas.xml.soap.org/ws/2002/04/secext namespace is used to pass security tokens inside the SOAP header. The contents of this header include security claims and signatures by which the sender of the message asserts the knowledge of a key. The contents of the security header block may be targeted to any receiver. The intended recipient of the header block can be specified using the standard SOAP actor attribute. A message may have more than one security header block.

However, no two security header blocks can have the same actors defined. Only one security header block may be defined without an actor attribute, and any intermediary, or the endpoint recipient, can consume this. This header block should not be removed before the message reaches the ultimate recipient.

The general structure of the security header block is shown below:

```
<s:Envelope xmlns:s="http://www.w3c.org/2000/12/soap-envelope">
  <s:Header>
    <wsse:Security
      xmlns:wsse="http://schemas.xml.soap.org/ws/2002/04/secext"
      S:actor="..."
      S:mustUnderstand="...">
      ...
    </wsse:Security>
  </s:Header>
  ...
</s:Header>
```

The contents of the `Security` element are not designed to provide extensible types of security information. Instead, it allows additional attributes to provide the extensibility mechanism. The WS-Security specification defines the following set of elements that can be used within the security header block:

- `UserNameToken`
  This is used for providing a user name and optional password information. The password can be sent either as clear text or a hash value.
- `BinarySecurityToken`
  This is used for encoding binary security tokens such as X.509 certificates, and Kerberos tickets.
- `SecurityTokenReference`
  This is used to provide reference to other services from which security tokens can be retrieved.
- `ds:KeyInfo`
  This belongs to the XML Signature namespace and is used to provide information regarding the keys for X.509 certificates among others.
- `ds:Signature`
  This is also from the XML Signature namespace and is used for providing digital signatures.
- `xenc:ReferenceList, xenc:EncryptedKey,` and `xenc:EncryptedData`
  These belong to the XML Encryption namespace and are used to provide encrypted information.

Let's now look at an example that demonstrates some of the topics we have covered.

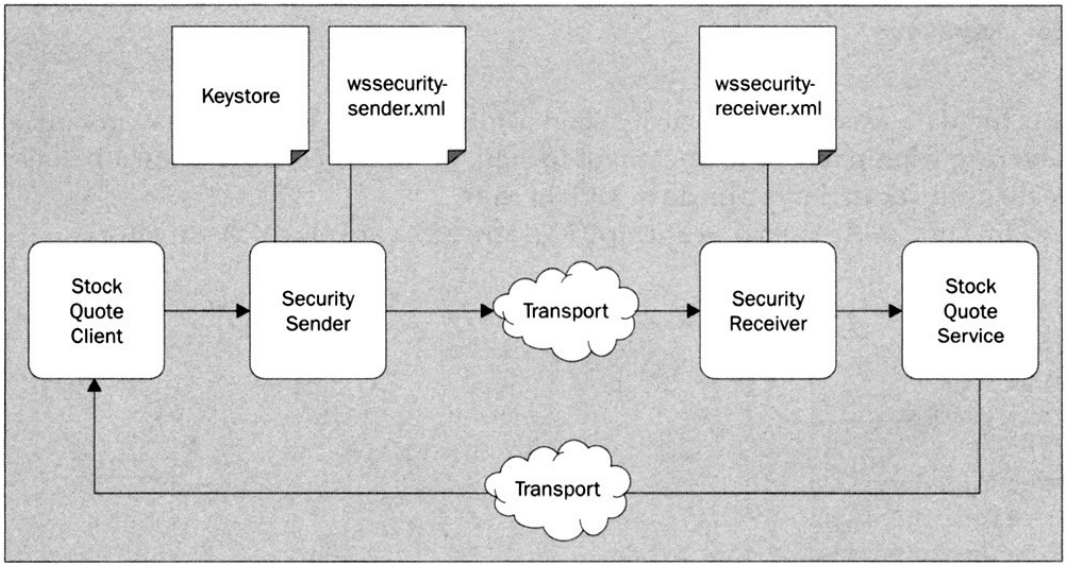
WS-Security in the StockQuote Service

In this section we will see how to sign the body part of the SOAP envelope representing the stock quote request on the client side. We will also look at how we send the signed information and the public key certificate of the signer, as a header block of the same SOAP envelope, to the stock quote service, in accordance with the WS-Security requirements for passing security tokens. The stock quote service will verify the signature using the certificate present in the SOAP envelope.

This is all achieved using the WSTK implementation of WS-Security. WSTK 3.2 provides two Axis handlers that can be used for processing binary security tokens according to the WS-Security requirements:

- `com.ibm.wstk.axis.handlers.SecurityReceiver`
  This can be used for verifying XML signatures and decrypting XML encrypted data
- `com.ibm.wstk.axis.handlers.SecuritySender`
  This can be used for generating XML signatures and creating XML encrypted data

The diagram below depicts the high-level architecture of using XML signature to create a digital signature of the stock request body part on the client side, and how it is verified on the server:



It is explained as follows:

- The client invokes the web service through the client-side security sender handler.
- The security sender first reads the `wssecurity-sender.xml` file to obtain information about which part of the document to sign and the key information required to sign the data.
- Then it accesses the keystore specified in the XML file and gets the private key and corresponding public key certificate for the key alias specified in the `config` file.
- The private key is used to sign the specified part of the SOAP request, and the signature and public key are added to that as a SOAP header block, as specified by the WS-Security specification.
- This envelope is then sent to the server.
- On the server the security receiver handler is invoked before the web service.
- This handler will access the `wssecurity-receiver.xml` file to find out how to verify the signature.
- If the signature is verified successfully, an information message is printed on the server and the service is invoked and the data is sent back to the client.

We will print the SOAP envelope before and after signing on the client side to illustrate the above process.

Try It Out: Deploying and Accessing the Web Service

This section assumes that you have installed and configured WSTK successfully with Tomcat. Please refer to the earlier chapters for details on installing and configuring WSTK. Deploying and accessing the web service will involve the following steps:

1. The command below will generate the key pair and public key certificate for you with the alias `meeraj`:

   ```
   keytool -genkey -dname "CN=Meeraj Kunnumpurath, OU=I Solutions, O=EDS, L=Milton
   Keynes, S=Bucks, C=UK" -alias meeraj -storepass -password -keypass password -
   keystore .keystore
   ```

2. This is the data used by the security sender for obtaining keystore information and discovering what part of the document to sign. Store the contents shown below and store in a file called `wssecurity-sender.xml` in the `\Beginning_JWS_Examples\Chp09\SignedStockQuote\` directory:

   ```
   <?xml version="1.0"?>
   <clientbinding>
     <service-ref>
       <port-qname-binding>

         <SenderServiceConfig>
           <SigningParts>
             <Reference part="body"/>
           </SigningParts>
         </SenderServiceConfig>

         <SenderBindingConfig>
           <SigningKey>
   ```

```
            <KeyStore type="jks"
              path="c:/Beginning_JWS_Examples/Chp09/
                       SignedStockQuote/.keystore"
              storepass="password"/>
            <PrivateKey alias="meeraj" keypass="password"/>
          </SigningKey>
        </SenderBindingConfig>

      </port-qname-binding>
    </service-ref>
  </clientbinding>
```

This configuration information states that the body part of the SOAP document should be signed using the private key identified by the alias `meeraj` in the keystore `\Beginning_JWS_Examples\Chp09\SignedStockQuote\.keystore`. Please note that the absolute path of this XML file should be specified in the client deployment descriptor, as we will see in a later section.

3. Now to create the server handler configuration data. This is the data used by the security receiver for obtaining information on how to verify the signature. Store the contents shown below in a file called `wssecurity-receiver.xml` in the `\Beginning_JWS_Examples\Chp09\SignedStockQuote\` directory:

```
  <?xml version="1.0" encoding="UTF-8"?>
  <wsbinding>
    <ws-desc-binding>
      <pc-binding>

        <ReceiverServiceConfig>
          <RequiredSignedParts>
            <Reference part="body"/>
          </RequiredSignedParts>
        </ReceiverServiceConfig>

        <ReceiverBindingConfig>
          <TrustAnchorList>
          <TrustAnyCertificate/>
          </TrustAnchorList>
        </ReceiverBindingConfig>

      </pc-binding>
    </ws-desc-binding>
  </wsbinding>
```

The configuration information above states that the body part of the SOAP document should be verified using the public key certificate present in the document. Please note that the absolute path of this file should be specified in the client deployment descriptor, as we will see in a later section.

4. To deploy the service first store the contents shown below in a file called `ServerDeploy.wsdd`:

```
  <deployment xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
```

Define the security receiver handler and specify the absolute path to the configuration file:

```
  <handler name="wssecurity-receiver"
    type="java:com.ibm.wstk.axis.handlers.SecurityReceiver">
    <parameter
      value="C:/Beginning_JWS_Examples/Chp09/
                SignedStockQuote/wssecurity-receiver.xml"
    name="configPath"/>
  </handler>
```

Define the service with the handler to be invoked before the service:

```
  <service name="StockQuoteService" provider="java:RPC">
    <requestFlow>
      <handler type="wssecurity-receiver"/>
      </requestFlow>
      <parameter value="com.wrox.jws.stockcore.StockCore" name="className"/>
      <parameter value="getQuote" name="allowedMethods"/>
    </service>

  </deployment>
```

5. Now run the `wstkenv` batch file present in the `bin` directory of WSTK to set up the required environment variables. Run the command

shown below to deploy the service:

```
java -classpath %WSTK_CP% org.apache.axis.client.AdminClient
     -lhttp://localhost:8080/wstk/common/services/AdminService
      ServerDeploy.wsdd
```

6. To deploy the client chain, first store the contents shown below in a file called `ClientDeploy.wsdd`:

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
```

Define the handler to log the request and response to a file:

```
  handler name="log" type="java:org.apache.axis.handlers.LogHandler"/>
```

Define the security sender handler and specify the absolute path to the configuration file. This will also print the SOAP envelope before and after signing:

```
  <handler
    name="wssecurity-sender"
    type="java:com.ibm.wstk.axis.handlers.SecuritySender">
    <parameter
      value="C:/Beginning_JWS_Examples/Chp09/
                SignedStockQuote/wssecurity-sender.xml"
      name="configPath"/>
    <parameter value="true" name="printBefore"/>
    <parameter value="true" name="printAfter"/>
  </handler>
```

Define the service along with the request and response handler chain:

```
  <service name="urn:StockQuoteServiceClient">
    <requestFlow>
      <handler type="log"/>
      <handler type="wssecurity-sender"/>
    </requestFlow>
    <responseFlow>
      <handler type="log"/>
    </responseFlow>
  </service>

</deployment>
```

7. Now run the `wstkenv` batch file present in the `bin` directory of WSTK to set up the required environment variables. Run the command shown below to deploy the service:

```
java -classpath "%WSTK_CP%" org.apache.axis.utils.Admin client
     ClientUndeploy.wsdd
```

8. The client class in this case is slightly different, as the client now invokes the web service through the client handler instead of accessing it directly:

```
package com.wrox.jws.stockcore;

import org.apache.axis.AxisFault;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import org.apache.axis.utils.Options;

import javax.xml.rpc.ParameterMode;
import javax.xml.namespace.QName;

import java.net.URL;

public class GetQuote {

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

    if (args == null || args.length <2) {
      System.out.println ("Usage: java GetQuote symbol url user password");
      System.exit (0);
    }

    String symbol = args[0] ;
```

```
        URL url = new URL (args [1] );

        Service service=new Service();

        Call call = (Call) service.createCall();

        call.setTargetEndpointAddress (url);
```

Now the client accesses the handler first and the handler will access the web service depending on the endpoint URL:

```
        call.setOperationName (new QName ("urn:StockQuoteServiceClient",
          "getQuote"));
        call.addParameter ("symbol", XMLType.XSD_STRING, ParameterMode.IN );
        call.setReturnType(XMLType.XSD_STRING);

        if (args.length == 4) {
          call.setUsername (args[2]);
          call.setPassword (args[3]);
        }

        System.out.println (call.invoke (new Object [] {symbol}));

      }
    }
```

9. Compile the client class as usual. To run the client class, run the `wstkenv` batch file present in the `bin` directory of WSTK to set up the required environment variables. Run the command shown below to deploy the service:

```
java -classpath %WSTK_CP% com.wrox.jws.stockcore.GetQuote IBM
     http://localhost:8080/wstk/common/services/StockQuoteService
```

This will invoke the web service, print the request SOAP envelope before and after signing, and print the result:

On the server side, the security receiver will verify the message and print a success message to the Tomcat console. Additionally, all the request and response messages will be stored into a file called `axis.log` in the current directory.

How It Works

The best way to demonstrate what is happening here is by looking at the contents of the `axis.log` file, as it will show us what is happening throughout the procedure. The contents of this file are shown below:

```
============================================================
= Elapsed: 38060 milliseconds
```

This is the request message:

```
= In message: <?xml version="1.0" encoding="UTF-8"?>
<soapenv: Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <soapenv:Header>
```

The WS-Security security header is used to store the binary security token containing the public key certificate of the private key used to sign the data:

```
    <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext"
      soapenv:mustUnderstand="1">
      <wsse:BinarySecurityToken EncodingType="wsse:Base64Binary"
    Id="wssecurity_binary_security_token_id_5557806051649257235_1028127480620"
        ValueType="wsse:X509v3">
        MIIDDTCCAssCBD1H6a8wCwYHKoZIzjgEAwUAMGwxEDAOBgNVBAYTB1Vua25vd24xEDAO
        BgNVBAgTB1Vua25vd24xEDAOBgNVBAcTB1Vua25vd24xEDAOBgNVBAoTB1Vua25vd24x
        EDAOBgNVBAsTB1Vua25vd24xEDAOBgNVBAMTB1Vua25vd24wHhcNMDIwNzMxMTM0NDE1
        ...
        kUgABeC/3gFgaH5fGpdcFs9IpFc26EbRKud8joNToqSE7BRAreBOGNPv1CYqcZA2r5dG
        Fg4hIeUwZpxp9finV7cAgKfh3zHNFXnJoprQscP9x2fJOXdddw7804L+uoK41DYKI4cX
        1WaEj47JIAoBMAsGByqGSM44BAMFAAMvADAsAhQVL09r6nmjkZHHtqDQUb+RHUcQXwIU
        Qiqe54Ckrd2pHjmYmQkbUdkS4aY=
      </wsse:BinarySecurityToken>
```

This element uses the XML Signature semantics to store the signature information:

```
      Signature xmlns="http://www.w3.org/2000/09/xmldsig#"
        Id="wssecurity_signature_id_2868516446223760635_1028127480510">
        <SignedInfo>
          <CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-
            exc-c14n#"/>
            <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-
            sha1"/>
```

The reference URI points to the ID attribute of the SOAP body part that is actually signed:

```
              <Reference
                URI="#wssecurity_body_1374006965804613176_1028127480510">
                <Transforms>
                  <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-
                  c14n#"/>
              </Transforms>
```

This is the digest method that is used:

```
              <DigestMethod
                Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
              <DigestValue>VmwBN8J936LQP601/E/IK4cbN4c=</DigestValue>
            </Reference>
            <Reference
              Type="http://www.w3.org/2000/09/xmldsig#SignatureProperties"
    URI="#wssecurity_signatureproperty_id_7160021005214925315_1028127480510">
              <Transforms>
                <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-
                  c14n#"/>
                </Transforms>
                <DigestMethod
                  Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
                <DigestValue>dxKPmsgw3pjHcJqi+WwmqawzoEA=</DigestValue>
              </Reference>
            </SignedInfo>
```

This is the actual digital signature:

```
            <SignatureValue>
              a5mYeRAsjnw3OoLp2PIjjMWU26AJTY3XtrU9MvuxmEKfJc8h7UWEpQ==
            </SignatureValue>
```

```
            <Object>
              <SignatureProperties>
                <SignatureProperty
          Id="wssecurity_signatureproperty_id_7160021005214925315_1028127480510"
          Target="#wssecurity_signature_id_2868516446223760635_1028127480510">
                  <ValueOfTimeStamp xmlns=" ">2002-07-
                    31T02:58:OOZ</ValueOfTimeStamp>
                </SignatureProperty>
              </SignatureProperties>
            </Object>
            <KeyInfo>
              <wsse:SecurityTokenReference>
                <wsse:Reference
URI="#wssecurity_binary_security_token_id_5557806051649257235_1028127480620"
                />
              </wsse:SecurityTokenReference>
            </KeyInfo>
          </Signature>
        </wsse:Security>
      </soapenv:Header>
```

This is the information that is signed and is identified by the `Id` attribute:

```
      <soapenv:Body xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext"
        wsse:Id="wssecurity_body_1374006965804613176_1028127480510">
      <ns1:getQuote xmlns:nsl="urn:StockQuoteServiceClient">
        <symbol xsi:type="xsd:string">IBM</symbol>
      </nsl:getQuote>
      </soapenv:Body>
    </soapenv:Envelope>
```

This is the response from the stock quote web service:

```
= Out message: <?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <nsl:getQuoteResponse
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:nsl="urn:StockQuoteServiceClient">
      <getQuoteReturn xsi:type="xsd:string">69.01</getQuoteReturn>
    </ns1:getQuoteResponse>
  </soapenv:Body>
</soapenv:Envelope>
============================================================
```

SAML

SAML, or **Security Assertion Markup Language** is an XML-based framework for exchanging security information, mainly in a federated environment. It is an OASIS specification addressing the following issues:

- Single sign-on
- Authorization services
- Back office transactions

Security information is expressed in terms of assertions about subjects. Subjects can be any entity that has an identity in a security domain. Assertions state whether a subject has been authenticated, or is authorized to access a resource, or has a specified set of attributes. The main types of assertions are:

- **Authentication Assertion**
  For example, an issuing authority asserts that subject, Fred Flintstone, was authenticated at 10:00 in the morning using a rock swipe card. An example of an SAML excerpt asserting this authentication is shown below:

```
   <saml:Assertion
     MajorVersion="1"
     MinorVersion="0"
     AssertionID="123"
     Issuer="Acme.com"
     IssueInstant="2001-12-12T:10:00:00Z">
```

```
    <saml:Conditions
      NotBefore="2001-12-12T:10:00:00Z"
      NotAfter="2001-12-12T:10:15:00Z"/>
    <saml:AuthenticationStatement
      AuthenticationMethod="RockSwipe"
      AuthenticationInstant="2001-12-12T:10:04:00Z">
      <saml:Subject>
        <saml:NameIdentifier
          SecurityDomain="Acme.com"
          Name="FredFlintstone"/>
      </saml:Subject>
    </saml:AuthenticationStatement>
  </saml:Assertion>
```

- **Attribute Assertion**
  For example, an issuing authority asserts that Fred Flintstone's (the subject's), `department` is `Heavy Lifting`. Here the `department` is the attribute and the value is `Heavy Lifting`. An example of SAML attribute assertion is shown below:

```
  <saml:Assertion
    MajorVersion="1"
    MinorVersion="0"
    AssertionID="123"
    Issuer="Acme.com"
    IssueInstant="2001-12-12T:10:00:00Z">
    <saml:Conditions
      NotBefore="2001-12-12T:10:00:00Z"
      NotAfter="2001-12-12T:10:15:00Z"/>
    <saml:AttributeStatement>
      <saml:Subject>
        <saml:NameIdentifier
          SecurityDomain="Acme.com"
          Name="FredFlintstone"/>
      </saml:Subject>
      <saml:Attribute
        AttributeName="Department"
        AttributeNamespace="Acme.com">
        <saml:AttributeValue>Heavy Lifting</saml:AttributeValue>
      </saml:Attribute>
    </saml:AttributeStatement >
  </saml:Assertion>
```

- **Authorization Decision Assertion**
  This asserts on the authority of a subject to perform an action on a resource. For example the subject Fred Flintstone is given access to fiddle the payroll information:

```
  <saml:Assertion
    MajorVersion="1"
    MinorVersion="0"
    AssertionID="123"
    Issuer="Acme.com"
    IssueInstant="2001-12-12T:10:00:00Z">
    <saml:Conditions
      NotBefore="2001-12-12T:10:00:00Z"
      NotAfter="2001-12-12T:10:15:00Z"/>
    <saml:AuthorizationDecisionStatement
      Decision="Permit"
      Resource="Payroll">
      <saml:Actions Namespace="Acme.com/Payroll">
        <saml:Action>Fiddle</saml:Action>
      </saml:Actions>
      <saml:Subject>
        <saml:NameIdentifier
          SecurityDomain="Acme.com"
          Name="FredFlintstone"/>
      </saml:Subject>
    </saml:AuthorizationDecisionStatement>
  </saml:Assertion>
```

Assertion Exchange Protocol

SAML uses a request-response protocol for acquiring assertions. Assertions queries are sent using the assertion request messages and the responses are received as assertion statements. The excerpt below shows an authentication assertion request:

```
<samlp:Request
  MajorVerion="1"
  ...>
  <samlp:AuthenticationQuery>
    <saml:Subject>
      <saml:NameIdentifier
        SecurityDomain="Acme.com"
        Name="FredFlintstone"/>
    </saml:Subject>
  <samlp:AuthenticationQuery>
</samlp:Request>
```

The response to this request will contain an authorization assertion statement as well as status messages indicating the state of the assertion. Similar types exist for attribute and authorization decision assertions.

Protocol Bindings

The baseline binding for encoding SAML encoding requests and responses is SOAP over HTTP.Other bindings, including raw HTTP, will follow in course of time. Vendors have started supporting SAML in their products for providing security service such as single sign-on, authentication, back-office transactions, and do on. JSAML is a Java implementation of SAML that is available for free from Netigrity (http://www.netigrity.com).

Summary

In this chapter we have covered the various security issues in the context of web services. First we started by discussing basic security concerns and had a look at the traditional security solutions that addressed these issues. Then we had a look at the Java support for traditional security solutions. We looked at the shortcomings of traditional security solutions in solving the issues in the web services world.

After that we looked at the various XML specifications that form the foundations for the latest web services security specification. We concluded the chapter by looking at WS-Security and SAML, the two most dominant security specifications currently targeting the web services world.