CHAPTER 6

Securing the Wire

Raw TCP packets flowing through a data network may be incomprehensible, even invisible, to a normal user fostering a sense of security, but in reality, the data in these packets are very accessible to those with the appropriate tools and know-how. The data networks over which these packets flow were not designed to protect the information from malicious folks and provide little or no security. With the help of programs freely available over the Internet, one can easily view, analyze and filter, on a normal PC, all the data being exchanged by machines on the same LAN. What it means is that a rogue neighbor, subscribing to the same cable or DSL ISP (**Internet Service Provider**) as you, can easily collect your account names and the passwords on different websites, including those from your online broker or bank, without you ever being suspicious. In fact, even if your neighbors are all perfectly honest people, it is possible that someone sitting across the ocean may take control of a machine and snoop over all the traffic. Similarly, a mischievous employee connected to an office LAN can watch all sorts of e-mail communication among coworkers and senior company officials on a normal work PC, without causing any special attention.

Outside the LAN, Internet traffic flows through a number of routers and gateways controlled by different organizations. People who have access to these systems, either legitimately or illegitimately, can collect the data, and in some cases even modify it or route it to different destinations.

Recently introduced wireless LANs offer even less security, as one can catch signals even without being physically connected. Imagine a CEO downloading confidential e-mail messages in a conference hall over a wireless LAN and some crook surreptitiously collecting all this information and benefiting from it in the stock market or in some other way.

To make matters worse, a number of widely used application protocols layered over TCP/IP, such as TELNET, FTP, SMTP and HTTP, make no attempt to protect the application data or

even the account names and passwords. Essentially, communication using these protocols is *not confidential* (can be seen by others), is *vulnerable to tampering* (can be modified), and *does not provide strong authentication* of end points (end-point addresses can be faked).

During the early days of the WWW (**World Wide Web**), these security concerns were a major stumbling block for wide adoption of e-commerce, as it required transmission of sensitive financial information such as bank account numbers and passwords, credit card information and so on. in clear text. In response, Netscape Communications, an early pioneer in this area and now part of AOL Time Warner, developed SSL protocol, a layer over TCP,[1] to secure data exchange between two communicating end points. This protocol has been widely adopted and has become the *de facto* mechanism to secure the exchange of sensitive information over the Internet.

SSL is an important piece of the overall puzzle of system security, providing the much needed network security. Other protocols also exist but none has achieved the same level of adoption. It is also an excellent example of using basic cryptography and PKI to meet higher-level system security needs. This chapter is devoted to the discussion of SSL protocol and the Java API to develop SSL-enabled programs. The example programs can be found in subdirectories of `%JSTK_HOME%\src\jsbook\ch6`. Continuing the tradition of building a more functional and usable tool around the example programs, we present **`ssltool`**, a tool that can run as an SSL client, server or proxy and can be used to explore the Java SSL environment.

Brief Overview of SSL

Early versions of SSL—SSLv1, SSLv2 and SSLv3—were developed by Netscape Communications and made available to other vendors for implementation. As different implementations appeared with their own interpretations of *not-so-well-specified* aspects of the protocol, it became clear that a more formal approach to standardization was needed. In response, IETF formed the TLS working group in May of 1996 to standardize an SSL-like protocol. The result was a protocol specified in RFC 2246, a minor upgrade of SSLv3 and known as TLS, or at times as TLSv1. As the basic principles and mode of operation are the same for both SSLv3 and TLS, we use the term SSL while talking about features and capabilities common to both, reserving the terms SSLv3 and TLSv1 for aspects that are unique to either of these versions.

In simple terms, SSL employs cryptography and PKI to provide message confidentiality, integrity and end-point authentication. Here is a layman's description of the SSL protocol: An SSL session is established between two end points of a TCP connection and encrypted data is exchanged over the established connection. To establish a session, the server presents a X.509 certificate to the client. The client may, on request of the server and after successfully validating the server certificate, present its certificate for mutual authentication. Subsequently, a key exchange algorithm, such as Diffie-Hellman, is used to compute a shared secret key. This secret

1. SSL specification does not mandate TCP but assumes only a reliable, connection-oriented transport. However, in practice, almost all implementations are for TCP.

key is used for encrypting and decrypting all the messages using a symmetric encryption algorithm. For ensuring message integrity, message digest is computed using a digest algorithm and is appended to the data.

The combination of the server authentication algorithm, key exchange algorithm, the encryption algorithm and the digest algorithm is known as a *cipher suite* and is conventionally represented as a string incorporating well-known abbreviations of each algorithm, separated by underscores. For example, cipher suite `TLS_DH_RSA_WITH_DES_CBC_SHA` refers to RSA authentication, Diffie-Hellman key exchange, DES_CBC bulk-encryption algorithm, and SHA digest. SSL uses a cipher-suite as parameter to the protocol. What it means is that newer and more powerful algorithms can be added without changing the basic protocol operation.

SSL is essentially a layer over TCP, meaning it relies on TCP for reliable, connection-oriented, byte stream-based communication, and any TCP-based higher-level protocol can be layered over SSL with minimum changes. Even the APIs provided by SSL libraries, including Java API for SSL, also known as JSSE (**Java Secure Socket Extension**), mimic the popular socket-based API for TCP programming. This design characteristic of SSL has been a key factor in its success as the dominant security protocol.

However, as we soon see, semantically SSL is quite different from TCP, and certain extension of the API and applications are necessary. These differences are primarily due to the need for each end-point to specify its certificate and validate the certificate supplied by the other end-point based on its list of trusted certificates. Recall from the discussion in previous chapters that within the Java platform a certificate with a private key is usually stored in a keystore, and trusted certificates, without private keys, are stored in a truststore.

With this brief introduction to SSL, let us turn our attention to the Java API for SSL. We come back with a more detailed description of the SSL protocol operation after going over the Java API and few example programs.

Java API for SSL

Java API for SSL or JSSE was available as a separate download prior to J2SE v1.4 but is now part of the standard platform. With this API, you can write SSL-enabled client and server programs. As the structure of these programs is similar to those of plain TCP-based programs and the SSL API is closely related to the underlying socket-based networking API, let us first go over a brief overview of TCP-based client-server programs and the Java socket API.

A typical TCP-based networking program plays the role of either a client, the one who initiates the connection, or a server, the one who accepts connection requests, or both, acting as a server for some interactions and a client for others. As the interaction style is different for client and server, it is convenient to talk about the actions of a particular role. In a server role, the program opens a server socket on a particular TCP port, identified by a number between 0 and 65536, and listens for incoming connection requests. In a client role, the program attempts to establish a TCP connection by specifying the machine name or IP address and the port number

of the destination server socket, and optionally, the local TCP port number. On a successful connection request processing, a new socket, bound to an unused TCP port, is created for the server program. The client also gets a socket, bound to either a specified port or an unused port chosen by the socket library.

Once the connection has been established and both client and server programs have their sockets, the connection essentially behaves like a two-way pipe. Data written by one end is available to the other end for reading and vice-versa. At the server program, the same thread that accepted the connection may engage in data exchange (*an iterative server*) or spawn a separate thread for data exchange (*a concurrent server*), allowing the main thread to listen for more connections. The exchange pattern and the syntax and semantics of the data are usually governed by a higher, application-level protocol.

Now, let us talk about using the Java APIs to perform these functions. These API classes and interfaces reside in the packages `java.net` and `javax.net`, and support many different sequences of class instantiations and method invocations to establish a connection. We limit our discussion to one such specific sequence of steps. These steps use the factory classes `ServerSocketFactory` and `SocketFactory` of the package `javax.net`. Concrete instances of default factory classes are obtained by calling the static method `getDefault()`. A server program calls the method `createServerSocket()` on `ServerSocketFactory` to create a `ServerSocket` bound to a specific port and waits for incoming connection requests by calling `accept()` on a `ServerSocket` instance. On getting a request and successfully establishing the connection, `accept()` returns a `java.net.Socket` object ready for read and write. A client program initiates a connection by calling `SocketFactory.createSocket()` passing the machine name or IP address and the port number of the destination socket. Successful execution of this call returns a `java.net.Socket` object. A `Socket` has two I/O streams associated with it: an `InputStream` for reading and an `OutputStream` for writing. You can get these by calling methods `getInputStream()` and `getOutputStream()`, respectively.

You must have noticed the underlying framework in these API classes. The framework essentially supports the notion of server sockets to accept connections and normal sockets for data exchange. The default sockets support TCP communication, but there can be sockets for other kinds of communication and appropriate factories create these sockets, presenting a uniform and consistent interface to programmers. Java API for SSL fits nicely into this framework with derived factory classes `SSLServerSocketFactory` and `SSLSocketFactory` and socket classes `SSLServerSocket` and `SSLSocket`, all under the `javax.net.ssl` package.

Enough theory. Let us now look at the source code of a simple client-server program that uses SSL to communicate. The source files for this program can be found under `src\jsbook\ch6\ex1` subdirectory of JSTK installation. Let us begin with the server program `EchoServer.java`. This program creates a SSL server socket on port 2950, waiting to accept connection requests on this port. Once a connection is established, it reads incoming data from the socket and echoes back the same data to the same socket.

Listing 6-1 Program to accept SSL connections and read/write data-bytes

```
// File: src\jsbook\ch6\ex1\EchoServer.java
import javax.net.ServerSocketFactory;
import javax.net.ssl.SSLServerSocketFactory;
import javax.net.ssl.SSLServerSocket;
import java.net.ServerSocket;
import java.net.Socket;

public class EchoServer {
  public static void main(String[] args) throws Exception {
    ServerSocketFactory ssf = SSLServerSocketFactory.getDefault();
    ServerSocket ss = ssf.createServerSocket(2950);

    // Placeholder for additional code.

    while (true){
      System.out.print("Waiting for connection... ");
      System.out.flush();
      Socket socket = ss.accept();
      System.out.println(" ... connection accepted.");
      SocketUtil.printSocketInfo(socket, " <-- ");

      java.io.InputStream is = socket.getInputStream();
      java.io.OutputStream os = socket.getOutputStream();
      int nread = 0;
      byte[] buf = new byte[1024];

      while ((nread = is.read(buf)) != -1){
        System.out.println("Read " + nread + " bytes.");
        os.write(buf, 0, nread);
        System.out.println("Wrote " + nread + " bytes.");
      } // inner while
    } // while (true)
  } // main()
}
```

Except for the static method `SocketUtil.printSocketInfo()`, which prints information about the socket passed as an argument, this code is fairly straightforward and hardly needs any explanation. Also, notice the comment indicating a placeholder for additional code. We get back to both these points in a short while.

The corresponding client side program, `EchoClient.java`, is given below. After establishing an SSL connection to the server, it prompts the user for a message, sends the message to the server, reads back the response, and displays it on the screen.

Listing 6-2 Program to initiate SSL connections and write/read data-bytes

```
// File: src\jsbook\ch6\ex1\EchoClient.java
import javax.net.SocketFactory;
import javax.net.ssl.SSLSocketFactory;
import java.net.ServerSocket;
import java.net.Socket;

public class EchoClient {
  public static void main(String[] args) throws Exception {
    String hostname = "localhost";
    if (args.length > 0)
      hostname = args[0];
    SocketFactory sf = SSLSocketFactory.getDefault();
    Socket socket = sf.createSocket(hostname, 2950);
    System.out.println("Connection established.");
    SocketUtil.printSocketInfo(socket, " --> ");

    java.io.InputStream is = socket.getInputStream();
    java.io.OutputStream os = socket.getOutputStream();
    byte[] buf = new byte[1024];
    java.io.BufferedReader br = new java.io.BufferedReader(
        new java.io.InputStreamReader(System.in));

    while (true){
      System.out.print("Enter Message (Type \"quit\" to exit): ");
      System.out.flush();
      String inp = br.readLine();
      if (inp.equalsIgnoreCase("quit"))
        break;
      os.write(inp.getBytes());
      int n = is.read(buf);
      System.out.println("Server Returned: " + new String(buf, 0,
n));
    }
    socket.close();
    System.out.println("Connection closed.");
  }
}
```

Those of you familiar with socket programming will notice that this is indeed quite similar to the plain TCP-based socket programs. But wait! What about the certificates for authentication and verification? Where do they come from? Well, by default the Java library examines a number of java system properties to get these values:

- `javax.net.ssl.keyStore`: The keystore file having the private key and the corresponding certificate or certificate chain required for authentication. It needs to be set at the server program. It is also required for the client program if client

authentication is enabled. Note that the EchoServer code presented above does not enable client authentication.

- `javax.net.ssl.keyStoreType`: The type of the keystore specified by system property `javax.net.ssl.keyStore`. Possible values are: `JKS`, `JCEKS` and `PKCS12`. Default value is `JKS`.
- `javax.net.ssl.keyStorePassword`: The password of the keystore specified by the system property `javax.net.ssl.keyStore`. This is required, as the server needs to load the private key. Recall that a keystore allows each entry to be password protected, potentially with different passwords. However, the default behavior relies on the fact that a random entry in the keystore is picked and this entry has the same password as that of the keystore.
- `javax.net.ssl.trustStore`: The truststore file, which is of the same format as a keystore file, having certificate entries for trusted subjects and issuers. This is required at the client program for verifying the certificate presented by the server. It is also required at the server to verify the client's certificate if client authentication is enabled. By default, keystore `jssecacerts`, if present in the *`jre-home`*`\lib\security` directory, is taken as the truststore. This keystore doesn't ship with J2SE v1.4 SDK. In its absence, the `cacerts` file of the same directory is used.
- `javax.net.ssl.trustStoreType`: The type of the truststore specified by system property `javax.net.ssl.trustStore`.
- `javax.net.ssl.trustStorePassword`: The password of the truststore specified through system property `javax.net.ssl.trustStore`. Use to check the integrity of the truststore, if specified. This password may be omitted.

You can pass these properties to the JVM at the command line using syntax "`java -D`*`prop`*`=`*`value`* ... *`classname`*" or programmatically within the code by invoking method `System.setProperty(`*`prop`*`, `*`value`*`)`. We describe this mechanism more concretely in a subsequent section.

The above description and code fragments may give the impression that SSL communication with Java is essentially the same as TCP communication, at least for a programmer. Reality is more complex:

- The above code fragments rely on JVM-wide system properties for specifying the certificate, private key and the set of trusted issuers. As we see later, this mechanism has its limitations and is not adequate for many scenarios.
- By default, the server doesn't negotiate or insist on client authentication. To check with the client if it can furnish a certificate or to insist on a certificate, separate calls, `setWantClientAuthentication(true)` or `setNeedClientAuthentication(true)`, are needed on a `SSLServerSocket` instance, taking us away from using the base `ServerSocket` class. Further examination of the SSL-specific properties

associated with an SSLSocket class would take us even further away from the generic model.

- Java New I/O library, under the package java.nio, with support for asynchronous I/O and direct buffers to minimize in-memory copy, supports TCP sockets, but not SSL sockets.

Let us augment the EchoServer.java file so that it can negotiate or insist on client authentication based on a command line argument. To do so, we would replace the placeholder comment in the *Listing 7-1* with the following code fragment:

```
if (args.length > 0){
  SSLServerSocket sss = (SSLServerSocket)ss;
  if ("-needClientAuth".equalsIgnoreCase(args[0])){
    sss.setNeedClientAuth(true);
  } else if ("-wantClientAuth".equalsIgnoreCase(args[0])){
    sss.setWantClientAuth(true);
  }
}
```

You can see that this requires casting the ServerSocket object into SSLServer-Socket. Retrieving SSL-specific connection information from the socket also requires us to work with an SSLSocket object, as we can see from the code in SocketUtil.java, shown in *Listing 6-3*.

Listing 6-3 Displaying SSL socket information

```
// File: src\jsbook\ch6\ex1\SocketUtil.java
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSession;
import javax.net.ssl.SSLPeerUnverifiedException;
import java.security.cert.Certificate;
import java.security.cert.X509Certificate;
import java.net.Socket;
import java.net.InetSocketAddress;

public class SocketUtil {
  public static void printSocketInfo(Socket socket, String dir) {
    try {
      InetSocketAddress localAddr, remoteAddr;
      LocalAddr = (InetSocketAddress)socket.getLocalSocketAddress();
      remoteAddr = (InetSocketAddress)socket.getRemoteSocketAddress();

      System.out.println("  Connection   : " +
          localAddr.getHostName() + ":" + localAddr.getPort() + dir +
          remoteAddr.getHostName() + ":" + remoteAddr.getPort());

      SSLSession sess = ((SSLSocket)socket).getSession();
      System.out.println("  Protocol     : " + sess.getProtocol());
```

```
      System.out.println("  Cipher Suite : " + sess.getCipherSuite());
      Certificate[] localCerts = sess.getLocalCertificates();
      if (localCerts != null && localCerts.length > 0)
        printCertDNs(localCerts, "  Local Cert");

      Certificate[] remoteCerts = null;
      try {
        remoteCerts = sess.getPeerCertificates();
        printCertDNs(remoteCerts, "  Remote Cert");
      } catch (SSLPeerUnverifiedException exc){
        System.out.println("  Remote Certs: Unverified");
      }
    } catch (Exception exc){
      System.err.println("Could not print Socket Information: " + exc);
    }
  }

  private static void printCertDNs(Certificate[] certs, String label){
    for (int i = 0; i < certs.length; i++){
      System.out.println(label + "[" + i + "]: " +
          ((X509Certificate)certs[i]).getSubjectDN());
    }
  }
}
```

The above listing illustrates how to access SSL-specific information, such as protocol, cipher suite, local and remote certificates from an active `SSLSocket` object. As we discover in the next section, the information displayed by the method `printSocketInfo()` is helpful to us in analyzing the behavior of the `EchoServer` and `EchoClient` programs.

Running Programs `EchoServer` and `EchoClient`

In this section, we go through the steps to compile and run the `EchoServer` and `EchoClient` programs. For simplicity, we run them on the same machine. However, you can certainly try them on two separate machines. The purpose is to get familiar with the operational issues and set the stage for more advanced development and experimentation.

Compiling the programs is simple. Just go to the source directory and issue this command:

```
C:\ch6\ex1>javac *.java
```

To run the programs, we need to set up appropriate keystore and truststore files. For this, we try a number of configurations:

1. Server authentication with a self-signed certificate. `EchoClient` trusts `EchoServer`'s certificate.
2. Mutual authentication with self-signed certificates. `EchoClient` trusts `EchoServer`'s certificate and `EchoServer` trusts `EchoClient`'s certificate.

3. `EchoServer` authentication with a CA signed certificate. `EchoClient` trusts CA's certificate.
4. Mutual authentication with CA signed certificates. Both `EchoClient` and `EchoServer` trust CA's certificate.

These configurations are illustrated in *Figure 6-1*.

With respect to each of these configurations, we are interested in two activities: setting up keystore and truststore files, and specifying system properties to run the server and the client. The following subsections explain the commands used to carry out these activities. The script files for these commands can be found in the same directory as the source files for `EchoServer` and `EchoClient` programs.

Server Authentication with Self-Signed Certificate

For this configuration, let us first create server keystore `server.ks` with a private key and self-signed certificate; export the certificate to a temporary file `temp$.cer`; and then import it

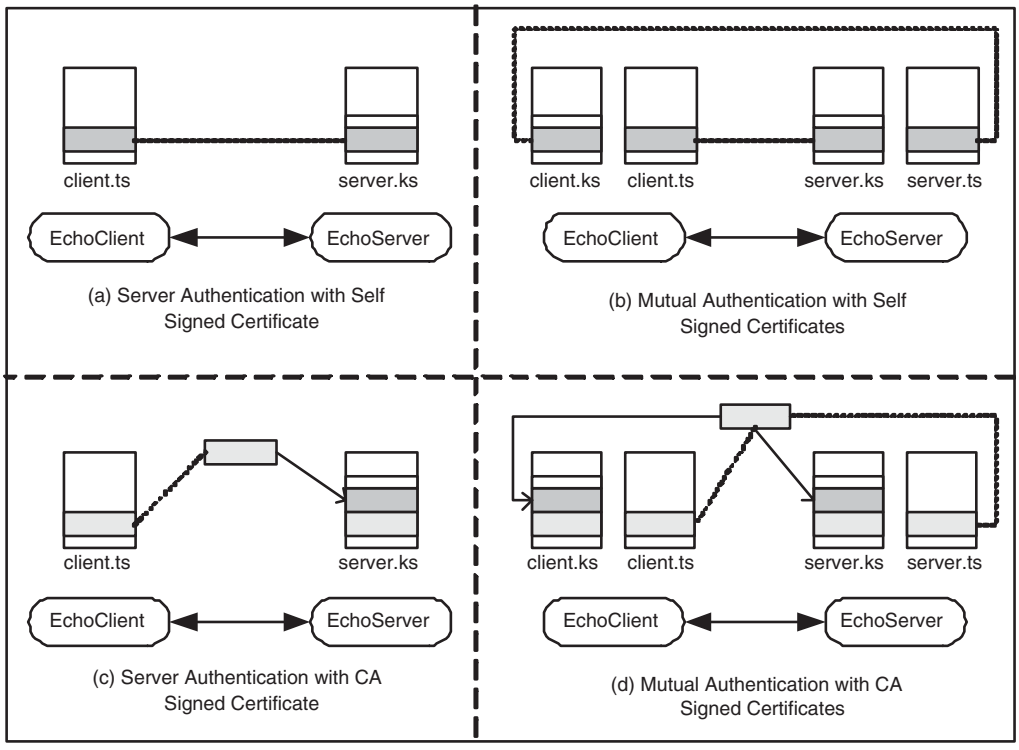

Figure 6-1 Configurations for running EchoServer and EchoClient.

from the temporary file to the client's truststore `client.ts`. Execution of these commands is shown below.

```
C:\ch6\ex1>keytool -genkey -storepass changeit -storetype JCEKS \
-keypass changeit -keystore server.ks -keyalg RSA \
-dname  "CN=Server, OU=X, O=Y, L=Z, S=XY, C=YZ"

C:\ch6\ex1>keytool -export -file temp$.cer -storepass changeit \
-storetype JCEKS -keypass changeit -keystore server.ks
Certificate stored in file <temp$.cer>

C:\ch6\ex1>keytool -import -file temp$.cer -storepass changeit \
-storetype JCEKS -keypass changeit -keystore client.ts -noprompt
Certificate was added to keystore
```

We are now ready to run the programs. Let us first run `EchoServer`, specifying the keystore information through system properties and let it wait for a connection.

```
C:\ch6\ex1>java -Djavax.net.ssl.keyStore=server.ks \
-Djavax.net.ssl.keyStoreType=JCEKS \
-Djavax.net.ssl.keyStorePassword=changeit EchoServer
Waiting for connection...
```

Now let us run the client program `EchoClient`, specifying the truststore information through system properties, in a different command window.

```
C:\ch6\ex1>java -Djavax.net.ssl.trustStore=client.ts \
-Djavax.net.ssl.trustStoreType=JCEKS EchoClient
Connection established.
  Connection   : 127.0.0.1:1296 --> localhost:2950
  Protocol     : TLSv1
  Cipher Suite : SSL_RSA_WITH_RC4_128_MD5
  Remote Certs: [0]CN=Server, OU=X, O=Y, L=Z, ST=XY, C=YZ
Enter Message (Type "quit" to exit): Hello, World!
Server Returned: Hello, World!
Enter Message (Type "quit" to exit): quit
Connection closed.
```

The server program displays the following output:

```
Waiting for connection...  ... connection accepted.
  Connection   : 127.0.0.1:2950 <-- 127.0.0.1:1296
  Protocol     : TLSv1
  Cipher Suite : SSL_RSA_WITH_RC4_128_MD5
  Local Certs : [0]CN=Server, OU=X, O=Y, L=Z, ST=XY, C=YZ
  Remote Certs: Unverified
Read 13 bytes.
Wrote 13 bytes.
Waiting for connection...
```

Note that the client program gets a remote certificate, the one supplied by the server, but the server program gets no certificate. This is expected, as the server has not asked for client authentication. Also, the cipher suite selected for the communication is `SSL_RSA_WITH_RC4_128_MD5`, the strongest with RSA encryption among all enabled cipher suites.

As an exercise, run these programs with a DSA certificate and see what cipher suite is used.

Mutual Authentication with Self-Signed Certificate

This configuration requires a separate keystore, `client.ks`, for the client program and a separate truststore, `server.ts`, for the server program in addition to what we had in the previous section. Let us create these keystore and truststore files with commands similar to the ones we used in the previous section:

```
C:\ch6\ex1>keytool -genkey -storepass changeit -storetype JCEKS \
-keypass changeit -keystore client.ks -keyalg RSA \
-dname "CN=Client, OU=X, O=Y, L=Z, S=XY, C=YZ"

C:\ch6\ex1>keytool -export -file temp$.cer -storepass changeit \
-storetype JCEKS -keypass changeit -keystore client.ks
Certificate stored in file <temp$.cer>

C:\ch6\ex1>keytool -import -file temp$.cer -storepass changeit \
-storetype JCEKS -keypass changeit -keystore server.ts -noprompt
Certificate was added to keystore
```

For running the server for mutual authentication, we need to specify not only the keystore information but also the truststore information, so that the client certificate can be verified, and the command line flag to either *require* or *negotiate* client certificate. The following command, by specifying the command line option `"-wantClientAuth"`, requires client authentication:

```
C:\ch6\ex1>java -Djavax.net.ssl.keyStore=server.ks \
-Djavax.net.ssl.keyStoreType=JCEKS \
-Djavax.net.ssl.keyStorePassword=changeit \
-Djavax.net.ssl.trustStore=server.ts \
-Djavax.net.ssl.trustStoreType=JCEKS EchoServer -wantClientAuth
Waiting for connection...
```

The command to run the client now needs to specify the keystore information as well, in addition to truststore information:

```
C:\ch6\ex1>java -Djavax.net.ssl.trustStore=client.ts \
-Djavax.net.ssl.trustStoreType=JCEKS \
-Djavax.net.ssl.keyStore=client.ks \
-Djavax.net.ssl.keyStoreType=JCEKS \
-Djavax.net.ssl.keyStorePassword=changeit EchoClient
```

```
Connection established.
  Connection   : 127.0.0.1:1297 --> localhost:2950
  Protocol     : TLSv1
  Cipher Suite : SSL_RSA_WITH_RC4_128_MD5
  Local Certs : [0]CN=Client, OU=X, O=Y, L=Z, ST=XY, C=YZ
  Remote Certs: [0]CN=Server, OU=X, O=Y, L=Z, ST=XY, C=YZ
Enter Message (Type "quit" to exit): Hello, Friend!
Server Returned: Hello, Friend!
Enter Message (Type "quit" to exit): quit
Connection closed.
```

Notice that it now shows not only the remote certificate but also the local certificate.

Try running the client program without specifying the keystore and see what happens. Also, try the scenario when the `EchoServer` specifies `"-needClientAuth"` and the `EchoClient` doesn't set keystore information.

Server Authentication with CA Signed Certificate

This configuration requires either getting a CA signed certificate from an established CA or setting up a CA of our own. We use our own JSTK utility **`certtool`**, explained in Chapter 4, *PKI with Java*, to set up a simple CA. To perform this setup, issue this command:

```
C:\ch6\ex1>%JSTK_HOME%\bin\certtool setupca -password changeit
CA setup successful: cadir
```

For the above command to work, the environment variable `JSTK_HOME` must be set to the home directory of the JSTK software.

The next steps are: create server keystore `server.ks` with a private key and self-signed certificate, generate a Certificate Signing Request, issue a CA signed certificate based on the CSR, and then import the signed certificate to the keystore server.ks. We also need to import the CA certificate in the client's trust store `client.ts`. The execution of these commands is presented below.

```
C:\ch6\ex1>keytool -genkey  -storepass changeit -storetype JCEKS \
-keypass changeit -keystore server.ks \
-dname "CN=Server, OU=X, O=Y, L=Z, S=XY, C=YZ"

C:\ch6\ex1>keytool -certreq -file temp$.csr -storepass changeit \
-storetype JCEKS -keypass changeit -keystore server.ks

C:\ch6\ex1>%JSTK_HOME%\bin\certtool issue -csrfile temp$.csr \
-cerfile server.cer -password changeit
Issued Certificate written to file: server.cer

C:\ch6\ex1>keytool -import -file server.cer -storepass changeit \
-storetype JCEKS -keypass changeit -keystore server.ks -noprompt
Certificate reply was installed in keystore
```

```
C:\ch6\ex1>keytool -import -file temp$.cer -storepass changeit \
-storetype JCEKS -keypass changeit -keystore client.ts -noprompt
Certificate was added to keystore
```

Note that these commands use the default key algorithm, i.e., DSA, for key generation.

The steps to run the `EchoServer` and `EchoClient` programs are the same as in the subsection Server Authentication with Self-Signed Certificate, and hence are skipped.

Mutual Authentication with CA Signed Certificate

Similar to mutual authentication with a self-signed certificate, this configuration needs a client keystore, `client.ks`, with a valid CA signed client certificate and a server truststore, `server.ts`, with CA's certificate in it. The commands to accomplish this are similar to those presented in the previous section *Server Authentication with CA Signed Certificate*, and are skipped.

Execute the `EchoServer` and `EchoClient` programs by issuing the same commands as the ones presented in the subsection *Mutual Authentication with Self-Signed Certificate*.

General Notes on Running Java SSL Programs

If you omit the truststore system properties while running either the client or server, the default truststore of the J2SE installation gets used. Validation of the self-signed certificate will not succeed against this truststore. However, it is possible to import additional certificates to this truststore, thus avoiding the need to specify truststore information with every invocation.

In the previous command executions, we chose to create our own certificates using **`keytool`** and **`certtool`** utilities. In most environments, this would not be the case. You would be either using certificates obtained by an established CA or using a full-fledged CA software to generate certificates. This would change the specifics of the each step, but conceptually you would be carrying out the same steps.

There are many more possible combinations to run the client and server programs than we have explained here. JSTK utility **`ssltool`** lets you try many of these combinations by simply changing the command line parameters and observing the result. Sometimes, it is quite instructive to try out combinations that are likely to fail and observe the error messages.

KeyManager and TrustManager APIs

Java SSL library has a flexible mechanism to access externally stored certificates for the purpose of authentication and verification. This mechanism consists of a Key Manager, an instance of a class implementing interface `javax.net.ssl.KeyManager`, to get the certificate for authentication, and Trust Manager, an instance of a class implementing interface `javax.net.ssl.TrustManager`, to get all the certificates for verifying a certificate. Note that the certificate to be used for authentication needs to be accompanied by the corresponding private key whereas certificates for verification have no such requirement.

The SSL library is initialized with default implementations of `KeyManager` and `TrustManager`. As we saw earlier, the default `KeyManager` looks at system properties `javax.net.ssl.keyStore`, `javax.net.ssl.keyStoreType` and `javax.net.ssl.keyStorePassword` to access authentication certificates from an external keystore. If more than one certificate is present in the keystore then one of the certificates matching the signature algorithm of the requested SSL cipher suite, the one found first by the implementation, is used. It is not possible to specify a specific certificate by specifying the alias of the entry in the keystore.

This default behavior is not adequate or appropriate in many real scenarios:

- You may not want the keystore password to be stored as a system property, accessible to all code within that JVM. In sensitive applications, the normal practice is to prompt the user for a password, read it within a character array and overwrite the array after use.
- You may want to get the certificate and the private key from a source different from a Java keystore. Say you want to access the same certificate used by your Apache or Microsoft IIS WebServer, without duplicating it in a Java KeyStore. This could be an important consideration if the Java program must integrate with an existing PKI infrastructure.
- You may want to select the certificate from a list, with input from the user. This is relevant in case of client authentication where the user is prompted to select a certificate from all the available certificates.

The solution is to write your own `KeyManager` class. The steps involved in writing and using a `KeyManager` for X.509 certificates are outlined below:

1. Write a `MyKeyManager` class that implements interface `javax.net.ssl.KeyManager`, and supply implementation of all the methods.
2. Write a provider class for `KeyManagerFactory`. This class extends `java.net.ssl.KeyManagerFactorySpi` and returns an array of `KeyManager` objects, with a single element as the `KeyManager` written in the previous step.
3. Register the `KeyManagerFactory` provider implementation of the previous step with an existing or your own Java Cryptographic Service provider.
4. For using the newly developed `MyKeyManager`, get its instance by invoking static method `getInstance()`, with the appropriate security provider as argument, on class `javax.net.ssl.KeyManagerFactory`; initialize a `javax.net.ssl.SSLContext` instance with the `KeyManager` array returned by the `KeyManagerFactory`; and get the `SocketFactory` or `ServerSocketFactory` from this `SSLContext` object. Use this factory to create sockets for SSL communication.

We have skipped a lot of details here. You can find them in the JSSE Reference Guide and the Javadoc documentation of various classes.

What about Trust Managers? Analogous to the default Key Manager, the default Trust Manager relies on system properties `javax.net.ssl.trustStore`, `javax.net.ssl.trustStoreType` and `javax.net.ssl.trustStorePassword` to access trusted certificates. The verification mechanism used by the default Trust Manager is simple compared to the PKIX validation we talked about in Chapter 4, *PKI with Java* and only looks for the presence of the issuer's certificate in the list of trusted certificates. There are situations when you would want to enhance the verification process. For example, a user-facing application may want to present the reason(s) for unsuccessful verification and let the user decide whether or not to continue. Whatever the reason, the mechanism to change the verification process is to write your own `MyTrustManager`. The steps are similar to those used for a custom `MyKeyManager`, and are skipped.

Understanding SSL Protocol

SSL setup and data exchange over the underlying TCP connection takes place in two different phases: *handshake* and *data transfer*. The handshake phase involves negotiation of the ciphersuite, authentication of end-points and agreement on cryptographic keys for subsequent encryption and decryption of application data. This essentially establishes a *SSL Session* between two end-points. The data transfer phase involves message digest computation, encryption and transmission of the encrypted data blocks at one end and reception, decryption and digest verification at the other end.

Recall that TCP is a byte stream-oriented protocol, meaning there is no grouping of a sequence of bytes in records at the application level. SSL protocol layers a record structure on top of this, exchanging data, both during handshake and data transfer phase, in *SSL Records*. An SSL record consists of a header and a body, the header indicating the size and type of the body. More details on the various SSL Record types and the exact format of SSL messages can be found in RFC 2246.

It is important to keep in mind that even though SSL introduces the notion of records on top of TCP byte stream, applications do not see this packaging and continue to operate on byte streams.

Figure 6-2 depicts the sequence of messages exchanged between a SSL server and a client during the handshake phase. These messages and the corresponding processing at the client and server are explained below:

1. The client sends a **`ClientHello`** message with a list of cipher suites it is willing to support, highest protocol version supported by the client, a random number and a compression method value, possibly NULL. Since a SSL client doesn't know whether it is

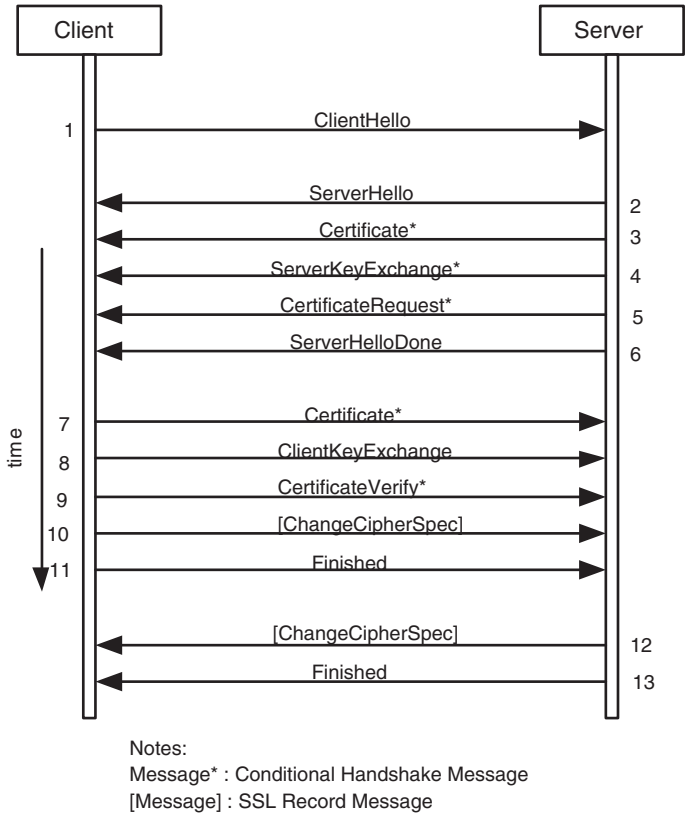

Figure 6-2 SSL Handshake between Client and Server.

talking to a SSLv2, SSLv3 or TLSv1 server, it could use the SSLv2 format for this message. This is the case with the J2SE v1.4 SSL library.

2. Based on the **`ClientHello`** message, the server picks the strongest cipher suite supported by both end points and responds back with a **`ServerHello`** message. This message contains protocol version (3.1 for TLSv1), a 32-byte random value, a 32-byte session ID, the selected cipher suite and agreed upon compression method value.
3. The server sends its certificate or certificate chain in a **`Certificate`** message if the selected cipher suite requires it. Except for a few cipher suites with `anon` authentication, all require the server to send its certificate.
4. The server sends a **`ServerKeyExchange`** message if it has no certificate or the certificate is for signing purposes only.
5. If the server is configured for *client authentication* then it sends a **`CertificateRequest`** message.

6. Finally, the server sends a **`ServerHelloDone`** message. Note that all messages since the **`ServerHello`** message could be packaged in one SSL record. On receiving all these messages, the client performs a number of operations: verifies server's certificate, extracts the public key, creates a secret string called *pre-master secret*, and encrypts it using the server's public key.
7. The client sends its certificate or certificate chain if the server asked for it.
8. The client sends the encrypted pre-master secret in **`ClientKeyExchange`** message.
9. The client sends a **`CertificateVerify`** message having a string signed by the private key corresponding to the public key of the certificate sent earlier. This message is not sent if the client did not send a certificate.
10. The client sends a **`ChangeCipherSpec`** message in a separate SSL record to indicate that now it is switching to the newly negotiated protocol parameters for subsequent communication.
11. The client sends a **`Finished`** message, which is a digest of the negotiated master secret and concatenated handshake messages. The server verifies this to make sure that the handshake has not been tampered with.
12. The server sends a **`ChangeCipherSpec`** message in a separate SSL record to indicate switching to the newly negotiated protocol parameters.
13. The server sends **`Finished`** message, which is a digest of the negotiated master secret and concatenated handshake messages. The client performs a similar verification on this message as the server.

This concludes the creation of a *SSL session* between the client and the server, allowing the end-points to exchange application data securely, using the negotiated parameters for encryption and decryption. It is to be noted that the handshake involves compute-intensive public-key cryptography, adding significant delay to the connection establishment process. We talk more about it in the *Performance Issues* section.

A SSL session can span multiple TCP connections. It is possible for two communicating end-points to use the parameters associated with a SSL session for a subsequent TCP connection using a technique known as *session resumption*. This helps avoid the costly handshake operation between the same client and server.

A large number of cipher suites have been specified in TLS specification and more have been added in subsequent RFCs. The original cipher suites included combinations of authentication and key exchange algorithms RSA, DH_DSS, DH_RSA, DHE_DSS, DHE_RSA; symmetric ciphers RC2, RC4, IDEA, DES and Triple DES; and digest functions MD5 and SHA. RFC 2712 adds cipher suites for Kerberos-based authentication and RFC 3268 adds cipher suites for AES symmetric cipher. You can get supported and enabled ciphers in a Java implementation by querying the `SSLSocketFactory` class. The source code to accomplish this can be found in file `ShowCipherSuites.java`.

Listing 6-4 Displaying cipher suites supported by JSSE

```
// File: src\jsbook\ch6\ex1\ShowCipherSuites.java
import javax.net.ssl.SSLSocketFactory;

public class ShowCipherSuites {
  public static void main(String[] unused) {
    SSLSocketFactory sf =

(SSLSocketFactory)SSLSocketFactory.getDefault();
    String[] supportedCSuites = sf.getSupportedCipherSuites();
    String[] enabledCSuites = sf.getDefaultCipherSuites();

    System.out.println("Supported Cipher Suites:");
    for (int i = 0; i < supportedCSuites.length; i++){
      System.out.println("\t[" + i + "] " + supportedCSuites[i]);
    }
    System.out.println("Enabled Cipher Suites  :");
    for (int i = 0; i < enabledCSuites.length; i++){
      System.out.println("\t[" + i + "] " + enabledCSuites[i]);
    }
  }
}
```

Running the `ShowCipherSuites` program with Sun's J2SE v1.4 produces the following output. The same output may be obtained by running command "`ssltool show -cs`".

```
C:\ch6\ex1>java ShowCipherSuites
Supported Cipher Suites:
        [0] SSL_RSA_WITH_RC4_128_MD5
        [1] SSL_RSA_WITH_RC4_128_SHA
        [2] SSL_RSA_WITH_3DES_EDE_CBC_SHA
        [3] SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
        [4] SSL_RSA_WITH_DES_CBC_SHA
        [5] SSL_DHE_DSS_WITH_DES_CBC_SHA
        [6] SSL_RSA_EXPORT_WITH_RC4_40_MD5
        [7] SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
        [8] SSL_RSA_WITH_NULL_MD5
        [9] SSL_RSA_WITH_NULL_SHA
        [10] SSL_DH_anon_WITH_RC4_128_MD5
        [11] SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
        [12] SSL_DH_anon_WITH_DES_CBC_SHA
        [13] SSL_DH_anon_EXPORT_WITH_RC4_40_MD5
        [14] SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA
Enabled Cipher Suites  :
        [0] SSL_RSA_WITH_RC4_128_MD5
        [1] SSL_RSA_WITH_RC4_128_SHA
        [2] SSL_RSA_WITH_3DES_EDE_CBC_SHA
        [3] SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
```

```
[4] SSL_RSA_WITH_DES_CBC_SHA
[5] SSL_DHE_DSS_WITH_DES_CBC_SHA
[6] SSL_RSA_EXPORT_WITH_RC4_40_MD5
[7] SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
```

This is only a small subset of cipher suites supported by TLSv1. As we see later, the choice of cipher suite has implications on security and performance of the system.

Does a programmer need to know all these gory details of SSL internals? The answer is yes and no. Most of the time everything works fine and you don't need to worry about the internal working of SSL. However, while configuring a system for SSL communication, it is common to encounter compatibility or configuration problems. In cases where you cannot get things working at the first try, you may want to look at the handshake messages to identify the cause of the problem. That is where the information presented in this section comes in handy.

How do you get hold of these messages? You can do so by either enabling debug messages at your Java program or by using a program like **ssldump** or running **ssltool** in proxy mode.

Enabling debug messages related to SSL operation in a Java program is simple. You set the system property `javax.net.debug` to the appropriate value, either at the command line or within the program by calling `System.setProperty()`. To get a list of all the supported values, set this property to string `"help"`. This property is examined only when a SSL related method is invoked, so you must have the program that performs some SSL operation to get the help message. You can find more about this system property and other ways of trouble-shooting in the section *Trouble Shooting*.

Run **ssltool** in proxy mode to analyze SSL messages by issuing command "**ssltool proxy -patype ssl -inport <inport> -host <host> -port <port>**". The client should connect to the proxy by specifying the machine running the proxy and `<inport>` as the target TCP address and the server's machine name and port number should set as arguments `<host>` and `<port>`. In contrast, **ssldump** can be used to listen for TCP packets flowing through a network interface card, without requiring the client's destination host and port to be modified. You can find more about **ssldump** at http://www.rtfm.com/ssldump/.

HTTP over SSL

HTTP has the notion of clients identifying and accessing network resources, files or programs, from an HTTP server through an HTTP URL, a string of form `"http://<machine>:<port>/<path>"`. Underneath, the client program opens a TCP connection to the server identified by machine and port (port 80 is assumed if no port is specified), sends a request, essentially a message consisting of text headers separated by newlines and optionally followed by a binary or text payload. The server gets the request, processes it, and sends back the response.

As is evident, it is fairly straightforward to layer HTTP over SSL, the combination also known as HTTPS. IETF RFCs 2817 and 2818 contain the necessary information to accomplish

this. A client indicates its desire to use SSL by using string `"https"` in the protocol part of the URL in place of `"http"`. A server could either upgrade a TCP connection to SSL at a client's request or open a separate port for all SSL connections. In practice, a separate port is almost always used, the default being port 443.

A client is expected to match the identity of the server by matching the server name in the URL with the CNAME field of the distinguished name corresponding to the certificate presented by the server. This check can be overridden if the client has external information, say in the form of user input, to verify the server's identity. Optionally, the server may also ask the client to authenticate itself by presenting a certificate.

Simple, isn't it? Don't be too hasty. Real world uses of HTTP are more complex than direct communication between a client and server. *HTTP proxies* aggregate outgoing traffic from *behind the firewall* corporate networks and *virtual hosts* allow multiple websites corresponding to different domain names to be hosted on the same physical machine. Their presence poses unique challenges to HTTPS.

HTTP proxies have been used to cache static content, thus reducing the load from the HTTP servers and improving the response time. They are also used to filter or log access to certain sites in some restricted environments. But HTTPS works by instructing the proxy to establish transparent TCP connection, bypassing it completely and negating many of its advantages. Virtual hosts present a different kind of problem—as the information about the target virtual host is available only in an HTTP Header field, which comes encrypted over SSL, there is no way for the HTTP server to identify and present the certificate corresponding to the target virtual host.

More detailed discussion of these issues is beyond the scope of this book.

Java API for HTTP and HTTPS

You can access an HTTP or HTTPS URL by simply constructing a `java.net.URL` object with the URL string, either starting with `http://` or `https://` as argument and reading the `InputStream` obtained by invoking method `openStream()` on the `URL` object. The source code of program `GetURL.java` illustrates this.

```
//File: src\jsbook\ch6\ex2\GetURL.java
import java.net.URL;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class GetURL {
  public static void main(String[] args) throws Exception {
    if (args.length < 1){
      System.out.println("Usage:: java GetURL <url>");
      return;
    }
    String urlString = args[0];
```

```
    URL url = new URL(urlString);
    BufferedReader br =
        new BufferedReader(new
InputStreamReader(url.openStream()));
    String line;
    while ((line = br.readLine()) != null){
      System.out.println(line);
   }
  }
}
```

Let us compile and run this program with a HTTPS URL.

```
C:\ch6\ex2>javac GetURL.java
C:\ch6\ex2>java GetURL https://www.etrade.com
<META HTTP-EQUIV="expires" CONTENT=0>
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">
... more stuff skipped ...
```

How did the program validate the certificate provided by the Web server running at `www.etrade.com`? The short answer is that it validates the server certificate against the default truststore. Recall that JSSE first tries file `jssecacerts`, and if not found then file `cacerts`, in directory *`jre-home`*`\lib\security` as the default truststore. Successful execution of `GetURL` with `https://www.etrade.com` implies that the server certificate is signed, either directly or indirectly, by a CA whose certificate is present in the default truststore.

This is really simple. But then, how do you specify the truststore to validate the server certificate? How do you override the default server identity verification based on matching the name in the certificate and the hostname in the URL? How do you provide the client certificate, if the server asks for it?

All of these are valid questions. And the good news is that you can do all of the above. But let us first understand what happens under the hood when `openStream()` is called.

The method `openStream()` internally invokes `openConnection()` on the `URL` object, and then `getInputStream()` on the returned `java.net.URLConnection` object. The returned `URLConnection` object is of type `java.net.HttpURLConnection` for an HTTP URL and of type `javax.net.ssl.HttpsURLConnection` for an HTTPS URL. Note that `HttpsURLConnection` is a subclass of `HttpURLConnection` and inherits all its public and protected methods.

Class `HttpsURLConnection` is associated with class `SSLSocketFactory` and uses this factory to establish the SSL connection. By default, this association is with the default `SSLSocketFactory` instance obtained by the static method `getDefault()` of `SSLSocketFactory`. Recall that the default `SSLSocketFactory` relies on a number of system properties to locate the truststore and the keystore information, and so does `HttpsURLConnection`. What it means is that you can specify the system properties as options to the JVM launcher at the command line or set them by invoking the `System.setProperty()` method.

You can also replace the default `SSLSocketFactory` with your instance of `SSLSocketFactory`, for all instances of `HttpsURLConnection` by calling the static method `setDefaultSSLSocketFactory()` or for a single instance by calling the method `setSSLSocketFactory()` on that instance.

Custom Hostname Verification

The mechanism to override hostname verification is slightly different. You need to extend the interface `javax.net.ssl.HostnameVerifier` and override the method `verify()` to place your logic there. An instance of this class can be set as the verifier to be called whenever the identification present in the server certificate doesn't match the hostname part of the URL. Similar to `SSLSocketFactory`, a `HostnameVerifier` can be set for all instances of `HttpsURLConnection` by calling the static method `setDefaultHostnameVerifier()` or for a single instance by calling the method `setHostnameVerifier ()` on that instance. The example program `GetVerifiedURL.java`, shown in *Listing 6-5* overrides hostname verification for a single instance of HTTPS connection.

Listing 6-5 Overriding hostname verifier

```
// File: src\jsbook\ch6\ex2\GetVerifiedURL.java
import java.net.URL;
import java.net.URLConnection;
import javax.net.ssl.HttpsURLConnection;
import javax.net.ssl.HostnameVerifier;
import javax.net.ssl.SSLSession;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class GetVerifiedURL {
  public static class CustomHostnameVerifier
                                    implements HostnameVerifier {
    private String hostname;
    public CustomHostnameVerifier(String hostname){
      this.hostname = hostname;
    }
    public boolean verify(String hostname, SSLSession sess){
      try {
        String peerHost = sess.getPeerHost();
        System.out.println("Expected hostname: " + hostname +
                                          ", Found: " + peerHost);
        System.out.print("Proceed(yes/no)?");  // Prompt user
        System.out.flush();
        BufferedReader br = new BufferedReader(
                            new InputStreamReader(System.in));
        String response = br.readLine();
```

```
        return ("yes".equalsIgnoreCase(response.trim()));
      } catch (IOException ioe){
      return false;
      }
    }
  }
  public static void main(String[] args) throws Exception {
    if (args.length < 1){
      System.out.println("Usage:: java GetVerifiedURL <url>");
   return;
    }
    String urlString = args[0];
    URL url = new URL(urlString);
    CustomHostnameVerifier custVerifier =
                         new CustomHostnameVerifier(url.getHost());
    URLConnection con = url.openConnection();
    if (!(con instanceof HttpsURLConnection)){
      System.out.println(urlString + " is not a HTTPS URL.");
   return;
    }
    HttpsURLConnection httpsCon = (HttpsURLConnection)con;
    httpsCon.setHostnameVerifier(custVerifier);
    BufferedReader br = new BufferedReader(
    new InputStreamReader(httpsCon.getInputStream()));
    String line;
    while ((line = br.readLine()) != null){
      System.out.println(line);
    }
  }
}
```

This program sets up a verifier that succeeds or fails based on user response, displaying expected hostname as specified in the URL and the hostname retrieved from the certificate presented by the server. Let us run this program twice, once with the URL `https://www.etrade.com` and then with `https://`*`ip-addr`*, where *`ip-addr`* is the IP address of the host `www.etrade.com` obtained by a DNS lookup tool like **`nslookup`**. What we find is that the first execution runs exactly like `GetURL`, but the second one triggers the execution of the `CustomHostnameVerifier` class, prompting the user to proceed or abort.

```
C:\ch6\ex2>java GetVerifiedURL https://12.153.224.22
Expected hostname: 12.153.224.22, Found: www.etrade.com
Proceed(yes/no)?yes
<META HTTP-EQUIV="expires" CONTENT=0>
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">
... more stuff skipped ...
```

This is expected, as the hostname string found in the certificate is not same as the one specified in the URL.

Tunneling Through Web Proxies

If you are running these programs on a machine behind a corporate firewall to read an external URL, and the firewall allows HTTP and HTTPS connections to be made through an HTTP proxy, then you should set the system properties `http.proxyHost` and `http.proxyPort` for HTTP URLs and `https.proxyHost` and `https.proxyPort` for HTTPS URLs to the host where the proxy is running and the corresponding port. Here is how the execution command looks behind a firewall.

```
C:\ch6\ex2>java -Dhttp.proxyHost=web-proxy -Dhttp.proxyPort=8088 \
GetURL http://www.etrade.com

C:\ch6\ex2>java -Dhttps.proxyHost=web-proxy -Dhttps.proxyPort=8088 \
GetURL https://www.etrade.com
```

To get Web proxy hostname and port values for your network, you could look into your browser setup or check with the network administrator.

RMI Over SSL

Socket-based programming is powerful but quite low-level for developing distributed Java applications. Most often, a programming paradigm based on a client program directly invoking methods on objects within a server program, passing input objects as arguments and getting output objects as return values, is more suitable. Java RMI is one such paradigm.

In its simplest form, an RMI server is a *unicast* server—meaning it supports point-to-point communication as opposed to broadcast or multicast. It lives within a running process and communicates with clients through sockets, the default being TCP sockets. Programmatically, one creates a unicast server class by subclassing the class `java.rmi.server.UnicastRemoteObject`. Such a class gets *exported* at the time of instantiation, or registered to the RMI system, so that method invocations can be dispatched to this object by the RMI system.

By default, the bits encapsulating method invocation, parameters and return values flow over the TCP connection in clear, and hence, suffer from the same security problems. This is not a problem most of the time as RMI is typically used within applications running within an enterprise over a trusted network. However, if you plan to deploy an RMI application over the Internet, or a not-so-trusted portion of an intranet, and are concerned about data security, then RMI over SSL could be a good solution. As we see, this can be accomplished with little programming.

But before we get to the source code to accomplish this, let us review the basic facts: the class `UnicastRemoteObject` has a constructor that takes three arguments—a TCP port number, a `java.rmi.server.RMIClientSocketFactory` object and a `java.rmi.server.RMIServerSocketFactory` object. The server invokes the method `createServerSocket()` on `RMIServerSocketFactory` to create `ServerSocket` object and the client invokes the method `createSocket()` on downloaded `RMIClientSocket-`

Factory (recall that Java byte-code is mobile and can move from one machine to another, in the same way as data) to create a Socket object. The default factories create normal ServerSocket and Socket objects for TCP communication. However, replacing TCP with SSL is a simple matter of supplying the right factories.

Let us look at factory class RMISSLServerSocketFactory to create SSLServerSocket using default SSLServerSocketFactory.

```
public class RMISSLServerSocketFactory
    implements RMIServerSocketFactory, Serializable {
  public ServerSocket createServerSocket(int port)
      throws IOException {
    ServerSocketFactory factory =
      SSLServerSocketFactory.getDefault();
    ServerSocket socket = factory.createServerSocket(port);
    return socket;
  }
}
```

The factory class RMISSLClientSocketFactory for client sockets is very similar:

```
public class RMISSLClientSocketFactory
    implements RMIClientSocketFactory, Serializable {
  public Socket createSocket(String host, int port)
      throws IOException {
    SocketFactory factory = SSLSocketFactory.getDefault();
    Socket socket = factory.createSocket(host, port);
    return socket;
  }
}
```

If you are well-versed in RMI programming then writing a unicast RMI server class using these factories is trivial. If not, look at the sample code provided with J2SE v1.4 SDK documentation under directory docs\guide\security\jsse\samples\rmi and the instructions to compile and run the programs. The JSTK utility **ssltool** also includes these factory classes and uses them for communication between client and server when you specify the protocol as SRMI, for Secure RMI. Actually, there in no such protocol as SRMI, this is a term that I coined to refer to RMI over SSL.

More information on RMI security and working examples can be found in Chapter 8, *RMI Security.*

Recall that the default SSLServerSocketFactory and SSLSocketFactory rely on system properties to locate the certificate and truststore. So an RMI program using these factories would expect the appropriate system properties to be set. You can override this behavior by supplying your own KeyManager and TrustManager. Look at the source files within JSTK for complete working code.

Performance Issues

It should come as no surprise that data transfer with SSL, on account of all the cryptographic processing, is slower than TCP. Security doesn't come free. Given that, you, an implementer and designer, should understand the extent and nature of this slowness and be able to assess its impact on user experience or response time and system capacity or number of concurrent users. As SSL adds no additional functionality on top of TCP, it is fairly straightforward to get an idea of the overhead by simply running the same program twice, once over TCP and then over SSL. We use a simple benchmark program, consisting of a client and server, to observe the overhead of SSL for making connections and for exchanging messages.

However, you must be careful in interpreting and applying these results to your specific application scenario. Micro-benchmarks, like the one we are going to talk about, tend to execute a small set of operations repeatedly, and suffer from a number of known problems, the most notable being that the benchmark may not simulate a real usage scenario. A real world application does more than just transmit messages and hence the overhead of SSL, although significant with respect to communication time, may be a small percentage of overall processing time. Nevertheless, there is value in understanding the performance trade-offs of basic operations for better system design.

Let us first define our benchmark and the measurement conditions: We measure the performance of SSL and TCP for two different types of activities—exchanging data and establishing connections. The performance of exchanging data over an established connection is measured at the client program by sending 8KB blocks of data to a server program, which acts as a sink, in a loop with an iteration count of 2048, and computing the data transfer rate in MB/second. Similarly, the rate of connection establishment is measured at the client program establishing connection with the server program in a loop and is expressed in terms of connections/second. JSTK utility **`ssltool`** was used to simulate client and server programs.

Table 6-1 presents the measured figures for data transfer rate over TCP and SSL connections with different cipher suites under three different scenarios: (I) both client and server programs run on the same machine; (II) client and server program run on different machines connected to a 100 Mbps (Mega *bits* per second) Ethernet LAN; (III) same as II but with 10 Mbps Ethernet LAN. Both machines were equipped with 450 MHz Pentium II CPU, 256 MB RAM, were running Windows 2000 with Service Pack3, and had J2SDK v1.4 from Sun.

A number of observations are in order:

- The measured TCP data transfer rate for scenarios (II) and (III) is close to the raw bandwidth offered by Ethernet wire. In fact, a popular native benchmarking tool for TCP performance on Windows machines, **`pcattcp`**, downloadable from http://www.pcausa.com/Utilities/pcattcp.htm, simulating the same load as our **`ssltool`** based benchmark, reported a bandwidth of 11.4 MB/sec and 1.1 MB/sec, respectively, between these two machines. These numbers are close to our observed values of 11.3 and 1.10, respectively.

Table 6-1 Data transfer rate (MB/second)—TCP and SSL

Connection Protocol/Cipher Suite	(I)	(II)	(III)
TCP	11.50	11.3	1.10
SSL_RSA_WITH_RC4_128_MD5	1.50	3.00	1.00
SSL_RSA_WITH_RC4_128_SHA	1.20	2.30	1.00
SSL_RSA_WITH_NULL_MD5	2.85	5.40	1.00
SSL_RSA_WITH_NULL_SHA	2.80	3.50	1.00
SSL_RSA_WITH_3DES_EDE_CBC_SHA	0.27	0.55	0.48
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA	0.27	0.55	0.48
SSL_RSA_WITH_DES_CBC_SHA	0.55	1.14	0.93
SSL_DHE_DSS_WITH_DES_CBC_SHA	0.55	1.14	0.90

- The SSL data transfer rate is worse when both client and server are running on the same machine than when they are running on different machines for 100 Mbps LAN. This can be explained by the fact that the benchmark simulates a continuous stream of data flow, allowing parallel cryptographic processing at both the machines.
- As expected, the SSL data transfer rate depends on the cipher suite. In most practical cases, SSL slows the data transfer by 90 to 95 percent for 100 Mbps LAN and by 10 to 50 percent for 10 Mbps LAN. How do we explain this? For 100 Mbps LAN, the CPU is the bottleneck whereas for 10 Mbps LAN, the network bandwidth becomes the bottleneck.
- Cipher suites using MD5 for computing MAC are faster than the corresponding ones using SHA. However, keep in mind that MD5, using 128 bits for hash values, is considerably weaker than SHA, which uses 160 bits for hash values.

Now let us look at connection establishment overhead of SSL under the same scenarios. These performance figures for TCP and certain selected cipher suites are presented in *Table 6-2*.

Table 6-2 Connection rate (connections/second)—TCP and SSL

Connection Protocol/Cipher Suite	(I)	(II)	(III)
TCP	330.0	500.0	500.0
SSL_RSA_WITH_RC4_128_MD5	6.5	3.2	3.2
SSL_RSA_WITH_RC4_128_SHA	6.5	3.2	3.2
SSL_RSA_WITH_3DES_EDE_CBC_SHA	6.5	3.2	3.2
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA	1.15	1.14	1.12

The impact of SSL is even more profound on the rate of connection establishment, with DSS-based authentication performing significantly worse than RSA. This is expected, as DSS is much slower than RSA. In fact, now that RSA's patent has expired, there is no valid reason to

use DHE and DSS in place of RSA. Other things to notice: network bandwidth plays no role here as the public-key cryptography dominates the overall connection setup time. Also, unlike the data transfer rate, the connection rate is not better for a two-machine scenario. This is only to be expected as the connection setup involves sequential processing by both machines.

In a real application the data transfer rate or connection time alone is rarely of much concern. What you really care about is the response time (how long does it take to satisfy a request?) and the capacity of the system (how many requests can be processed per unit of time?). The impact of SSL on both of these metrics would depend on the structure of a specific application, average load, type of interactions, network latency, specific J2SE platform and many other things besides CPU speed, network bandwidth and transport protocol. Still, the use of SSL could have a significant adverse impact on response time and system capacity. We will talk some more about it in Chapter 8, *Web Application Security.*

The good news is that this processing can be significantly speeded up by specialized crypto accelerators and separate SSL connections can be processed in parallel on different machines, improving the response time and scaling the capacity.

Trouble Shooting

Developing and testing SSL programs could be quite frustrating for beginners. It would seem the program compiles properly, everything is set up the right way and still nothing works. It takes, even for experience programmers, some amount of detective work to figure out what is wrong. Let us look at what goes wrong.

System properties are not set properly. So much behavior of JSSE APIs is governed by system properties is that even a minor mistake, like a typographical error or use of the wrong keystore or truststore file, may cause things go haywire. So, when your program is having trouble, recheck the system property names and their values.

There is no common cipher suite between the client and the server. This is rare when both ends are Java programs, running with the same JSSE implementation. However, with independent client and server programs, this is a real possibility. I spent many frustrating hours trying to figure out why a Java HTTPS server program was not working with a Netscape 7.0 browser. It turned out that the server certificate was using a DSA algorithm and there was just no common cipher suite between the two ends. Changing the key algorithm to RSA solved the problem.

Non-interoperable implementation. The interoperability has significantly improved with SSLv3 and TLSv1, but still, if you are working with less frequently used implementations, it is quite possible to encounter non-interoperable implementations.

There is a firewall between communicating ends. A firewall may disallow communication using certain port numbers. We talked about how to get around Web-proxy based firewalls, but not all firewalls are so friendly. In such cases, if you have a genuine business need to cross the firewall, you should talk to your network administrator. Going by my personal experience, this could be quite frustrating, especially in large companies.

Certificate validation fails. Even if everything is all right, it may happen that the certificate validation itself fails. Maybe the truststore doesn't have the certificate of the CA that signed the certificate under validation. Or perhaps the certificate is expired. Look at error messages carefully to determine the cause of failure.

The JSSE implementation is different from the one that comes with J2SE v1.4. A number of settings we have covered in this chapter are quite specific to the JSSE provider that comes with J2SE v1.4. If you are working with a different implementation, you should consult the relevant documentation for an appropriate configuration.

What trouble-shooting tools are available to a Java programmer to identify the cause of the errant behavior? At the minimum, you can set the system property `javax.net.debug` to an appropriate value. This generates log messages on the screen that might give important clues to what is going wrong. To get a listing of all the possible values this system property might be set to, run a SSL-aware program with this system property set to `help`. Here is the output shown by running the `ShowCipherSuites` program with the `javax.net.debug` set to `help`:

```
C:\ch6\ex1>java -Djavax.net.debug=help ShowCipherSuites

all             turn on all debugging
ssl             turn on ssl debugging

The following can be used with ssl:
        record       enable per-record tracing
        handshake    print each handshake message
        keygen       print key generation data
        session      print session activity
        defaultctx   print default SSL initialization
        sslctx       print SSLContext tracing
        sessioncache print session cache tracing
        keymanager   print key manager tracing
        trustmanager print trust manager tracing

        handshake debugging can be widened with:
        data         hex dump of each handshake message
        verbose      verbose handshake message printing

        record debugging can be widened with:
        plaintext    hex dump of record plaintext
```

A good way to get comfortable with these options is to deliberately introduce error conditions and browse the output generated.

Summary

SSL is a secure data communication protocol layered over TCP. It inherits the properties of being reliable and connection-oriented from the underlying TCP and adds security capabilities of end-point authentication, data confidentiality and message integrity, making use of cryptography and PKI. SSL is remarkable in its ability to hide the inherent complexity of cryptographic algorithms and PKI abstractions and expose a simple and familiar interface to applications.

SSL API for Java is modeled after socket-based networking API and it is fairly straightforward to modify existing TCP programs to use SSL. Using JCA-compliant API to plug different implementation of cryptographic services and to build and install key managers and trust managers provides an extensible framework to use security components from different sources.

HTTP over SSL, also referred to as HTTPS, has been widely deployed to secure connections between a Web Browser and a Web Server for exchanging sensitive information such as user account names, passwords, credit card information, bank account details, and so on. The popularity of HTTP, and hence HTTPS, for newer uses such as Web Services communication, implies that SSL will continue to be the dominant protocol to secure online connections.

As expected, **SSL communication is slower than plain TCP communication**. The initial handshake required by SSL and subsequent encryption and decryption consume CPU cycles, with the net effect of decreasing communication latency and bandwidth. Fortunately, SSL allows a number of cryptographic parameters to be negotiated to meet the performance and security needs of the application. You can also boost SSL bandwidth by adding more CPU power and speedup SSL latency by special crypto accelerators.

Experimentation and troubleshooting with various configuration parameters and interaction with external components to get an SSL program working can be quite a challenge. This is where the JSTK utility **`ssltool`** can help you do your job better and quicker. Further debugging tips explained in this chapter should also come in handy.

Further Reading

This chapter has covered SSL from the perspective of a Java developer. Further details on the protocol, its evolution, supported cipher suites and so on. can be found in the book *SSL and TLS: Designing and Building Secure Systems*. Its author, Eric Rescorla, has developed **`ssldump`**, a freely available tool based on OpenSSL cryptographic library and `libpcap` packet capture library to analyze SSL traffic by capturing data packets flowing through a network interface card. You can download OpenSSL from http://www.openssl.org, libpcap from http://www.tcpdump.org, and **`ssldump`** from http://www.rtfm.com/ssldump. This tool, especially its display format, has been the inspiration behind the SSL protocol analysis capability of JSTK utility **`ssltool`** when running as a proxy.

The official document specifying the TLSv1 standard is in IETF RFC 2246. Two RFCs, RFC 2817 and RFC2818, provide information on using HTTP over SSL.

The best source for authoritative and up-to-date documentation on Java API for SSL is the official reference guide from Sun Microsystems, *Java Secure Socket Extension (JSSE) Reference Guide for J2SE SDK, v1.4*. This document has good background information on the underlying protocol and contains code samples and operating procedures, in addition to a description of the API classes, interfaces and standard names. Further method-level details can be found in the Javadoc API documentation.