Table of Contents

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Chapter 11. Secure Sockets

One of the perennial fears of consumers buying goods over the Internet is that some hacker will steal their credit card number and run up a several-thousand-dollar bill by calling phone sex lines. In reality, it's more likely that a clerk at a department store will read their credit card number from a store receipt than that some hacker will grab it in transit across the Internet. In fact, as of mid-2004, the major online thefts of credit card numbers have been accomplished by stealing the information from poorly secured databases and filesystems *after* the information has been safely transmitted across the Internet. Nonetheless, to make Internet connections more fundamentally secure, sockets can be encrypted. This allows transactions to be confidential, authenticated, and accurate.

However, encryption is a complex subject. Performing it properly requires a detailed understanding not only of the mathematical algorithms used to encrypt data but also of the protocols used to exchange keys and encrypted data. Even a small mistake can open a large hole in your armor and reveal your communications to an eavesdropper. Consequently, writing encryption software is a task best left to experts. Fortunately, nonexperts with only a layperson's understanding of the underlying protocols and algorithms can secure their communications with software designed by experts. Every time you order something from an online store, chances are the transaction is encrypted and authenticated using protocols and algorithms you need to know next to nothing about. As a programmer who wants to write network client software that talks to online stores, you need to know a little more about the protocols and algorithms involved but not a lot more, provided you can use a class library written by experts who do understand the details. If you want to write the server software that runs the online store, then you need to know a little bit more but still not as much as you would if you were designing all this from scratch without reference to other work.

Until recently, such software was subject to the arms control laws of the United States. To some extent it still is. Laws about encryption in other countries range from much stricter than the U.S.'s to nonexistent. This has limited the ability of Sun and other vendors who operate internationally to ship strong encryption software. Consequently, such capabilities were not built into the standard `java.net` classes until Java 1.4. Prior to this, they were available as a standard extension called the Java Secure Sockets Extension (JSSE). Although JSSE is now

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

part of the standard distribution of the JDK, it is still hobbled by design decisions made to support earlier, less liberal export control regulations, and it is therefore less simple and easy to use than it could or should be.

Nonetheless, JSSE can secure network communications using the Secure Sockets Layer (SSL) Version 3 and Transport Layer Security (TLS) protocols and their associated algorithms. SSL is a security protocol that enables web browsers to talk to web servers using various levels of confidentiality and authentication.

11.1. Secure Communications

Confidential communication through an open channel such as the public Internet absolutely requires that data be encrypted. Most encryption schemes that lend themselves to computer implementation are based on the notion of a key, a slightly more general kind of password that's not limited to text. The clear text message is combined with the bits of the key according to a mathematical algorithm to produce the encrypted cipher text. Using keys with more bits makes messages exponentially more difficult to decrypt by brute-force guessing of the key.

In traditional *secret key* (or *symmetric*) encryption, the same key is used for both encrypting and decrypting the data. Both the sender and the receiver have to possess the single key. Imagine Angela wants to send Gus a secret message. She first sends Gus the key they'll use to exchange the secret. But the key can't be encrypted because Gus doesn't have the key yet, so Angela has to send the key unencrypted. Now suppose Edgar is eavesdropping on the connection between Angela and Gus. He will get the key at the same time that Gus does. From that point forward, he can read anything Angela and Gus say to each other using that key.

In *public key* (or *asymmetric*) encryption, different keys are used to encrypt and decrypt the data. One key, called the public key, is used to encrypt the data. This key can be given to anyone. A different key, called the private key, is used to decrypt the data. This must be kept secret but needs to be possessed by only one of the correspondents. If Angela wants to send a message to Gus, she asks Gus for his public key. Gus sends it to her over an unencrypted connection. Angela uses Gus's public key to encrypt her message and sends it to him. If Edgar is eavesdropping when Gus sends Angela his key, Edgar also gets Gus's public key. However, this doesn't allow Edgar to decrypt the message Angela sends Gus, since decryption requires Gus's private key. The message is safe even if the public key is detected in transit.

Asymmetric encryption can also be used for authentication and message integrity checking. For this use, Angela would encrypt a message with her private key before sending it. When

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Gus received it, he'd decrypt it with Angela's public key. If the decryption succeeded, Gus would know that the message came from Angela. After all, no one else could have produced a message that would decrypt properly with her public key. Gus would also know that the message wasn't changed en route, either maliciously by Edgar or unintentionally by buggy software or network noise, since any such change would have screwed up the decryption. With a little more effort, Angela can double-encrypt the message, once with her private key, once with Gus's public key, thus getting all three benefits of privacy, authentication, and integrity.

In practice, public key encryption is much more CPU-intensive and much slower than secret key encryption. Therefore, instead of encrypting the entire transmission with Gus's public key, Angela encrypts a traditional secret key and sends it to Gus. Gus decrypts it with his private key. Now Angela and Gus both know the secret key, but Edgar doesn't. Therefore, Gus and Angela can now use faster secret-key encryption to communicate privately without Edgar listening in.

Edgar still has one good attack on this protocol, however. (Very important: the attack is on the protocol used to send and receive messages, *not* on the encryption algorithms used. This attack does not require Edgar to break Gus and Angela's encryption and is completely independent of key length.) Edgar can not only read Gus's public key when he sends it to Angela, but he can also replace it with his own public key! Then when Angela thinks she's encrypting a message with Gus's public key, she's really using Edgar's. When she sends a message to Gus, Edgar intercepts it, decrypts it using his private key, encrypts it using Gus's public key, and sends it on to Gus. This is called a *man-in-the-middle attack*. Working alone on an insecure channel, Gus and Angela have no easy way to protect against this. The solution used in practice is for both Gus and Angela to store and verify their public keys with a trusted third-party certification authority. Rather than sending each other their public keys, Gus and Angela retrieve each other's public key from the certification authority. This scheme still isn't perfect—Edgar may be able to place himself in between Gus and the certification authority, Angela and the certification authority, and Gus and Angela—but it makes life harder for Edgar.

This discussion has been necessarily brief. Many interesting details have been skimmed over or omitted entirely. If you want to know more, the Crypt Cabal's Cryptography FAQ at http://www.faqs.org/faqs/cryptography-faq/ is a good place to start. For an in-depth analysis of protocols and algorithms for confidentiality, authentication, and message integrity, Bruce Schneier's *Applied Cryptography* (Wiley & Sons)

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

is the standard introductory text. Finally, Jonathan Knudsen's *Java Cryptography* (O'Reilly) and Scott Oak's *Java Security* (O'Reilly) cover the underlying cryptography and authentication packages on which the JSSE rests.

As this example indicates, the theory and practice of encryption and authentication, both algorithms and protocols, is a challenging field that's fraught with mines and pitfalls to surprise the amateur cryptographer. It is much easier to design a bad encryption algorithm or protocol than a good one. And it's not always obvious which algorithms and protocols are good and which aren't. Fortunately, you don't have to be a cryptography expert to use strong cryptography in Java network programs. JSSE shields you from the low-level details of how algorithms are negotiated, keys are exchanged, correspondents are authenticated, and data is encrypted. JSSE allows you to create sockets and server sockets that transparently handle the negotiations and encryption necessary for secure communication. All you have to do is send your data over the same streams and sockets you're familiar with from previous chapters. The Java Secure Socket Extension is divided into four packages:

`javax.net.ssl`
: The abstract classes that define Java's API for secure network communication.

`javax.net`
: The abstract socket factory classes used instead of constructors to create secure sockets.

`javax.security.cert`
: A minimal set of classes for handling public key certificates that's needed for SSL in Java 1.1. (In Java 1.2 and later, the `java.security.cert` package should be used instead.)

`com.sun.net.ssl`
: The concrete classes that implement the encryption algorithms and protocols in Sun's reference implementation of the JSSE. Technically, these are not part of the JSSE standard. Other implementers may replace this package with one of their own; for instance, one that uses native code to speed up the CPU-intensive key generation and encryption process.

None of these are included as a standard part of the JDK prior to Java 1.4. To use these with Java 1.3 and earlier, you have to download the JSSE from http://java.sun.com/products/jsse/ and install it. Third parties have also implemented this API, most notably Casey Marshall, who wrote Jessie (http://www.nongnu.org/jessie/), an open source implementation of JSSE published under the GPL with library exception.

Sun's reference implementation is distributed as a Zip file, which you can unpack and place anywhere on your system. In the *lib* directory of this Zip file, you'll find three JAR archives: *jcert.jar*, *jnet.jar*, and *jsse.jar*. These need to be placed in your class path or *jre/lib/ext* directory.

Next you need to register the cryptography provider by editing your *jre/lib/ext/security/java.security* file. Open this file in a text editor and look for a line like these:

```
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.rsajca.Provider
```

You may have more or fewer providers than this. However many you have, add one more line like this:

```
security.provider.3=com.sun.net.ssl.internal.ssl.Provider
```

You may have to change the "3" to 2 or 4 or 5 or whatever the next number is in the security provider sequence. If you install a third-party JSSE implementation, you'll add another line like this with the class name as specified by your JSSE implementation's documentation.

If you use multiple copies of the JRE, you'll need to repeat this procedure for each one you use. For reasons that have never been completely clear to me, Sun's JDK installer always places multiple copies of the JRE on my Windows box: one for compiling and one for running. You have to make these changes to both copies to get JSSE programs to run.

If you don't get this right, you'll see exceptions like "java.net.SocketException: SSL implementation not available" when you try to run programs that use the JSSE. Alternatively, instead of editing the *java.policy* file, you can add this line to classes that use Sun's implementation of the JSSE:

```
java.security.Security.addProvider(
 new com.sun.net.ssl.internal.ssl.Provider( ));
```

This may be useful if you're writing software to run on someone else's system and don't want to ask them to modify the *java.policy* file.

11.2. Creating Secure Client Sockets

If you don't care very much about the underlying details, using an encrypted SSL socket to talk to an existing secure server is truly straightforward. Rather than constructing a `java.net.Socket` object with a constructor, you get one from a `javax.net.ssl.SSLSocketFactory` using its `createSocket()` method. `SSLSocketFactory` is an abstract class that follows the abstract factory design pattern:

```
public abstract class SSLSocketFactory extends SocketFactory
```

Since the `SSLFactorySocket` class is itself abstract, you get an instance of it by invoking the static `SSLSocketFactory.getDefault()` method:

```
public static SocketFactory getDefault( ) throws InstantiationException
```

This either returns an instance of `SSLSocketFactory` or throws an `InstantiationException` if no concrete subclass can be found. Once you have a reference to the factory, use one of these five overloaded `createSocket( )` methods to build an `SSLSocket`:

```
public abstract Socket createSocket(String host, int port)
 throws IOException, UnknownHostException
public abstract Socket createSocket(InetAddress host, int port)
 throws IOException
public abstract Socket createSocket(String host, int port,
 InetAddress interface, int localPort)
 throws IOException, UnknownHostException
public abstract Socket createSocket(InetAddress host, int port,
 InetAddress interface, int localPort)
 throws IOException, UnknownHostException
public abstract Socket createSocket(Socket proxy, String host, int port,
 boolean autoClose) throws IOException
```

The first two methods create and return a socket that's connected to the specified host and port or throw an `IOException` if they can't connect. The third and fourth methods connect and return a socket that's connected to the specified host and port from the specified local network interface and port. The last `createSocket( )` method, however, is a little different. It begins with an existing `Socket` object that's connected to a proxy server. It returns a `Socket` that tunnels through this proxy server to the specified host and port. The `autoClose` argument determines whether the underlying `proxy` socket should be closed when this socket is closed. If `autoClose` is `true`, the underlying socket will be closed; if `false`, it won't be.

The `Socket` that all these methods return will really be a `javax.net.ssl.SSLSocket`, a subclass of `java.net.Socket`. However, you don't need to know that. Once the secure socket has been created, you use it just like any other socket, through its `getInputStream( ),getOutputStream(),` and other methods. For example, let's suppose there's a server running on *login.ibiblio.org* on port 7,000 that accepts orders. Each order is sent as an ASCII string using a single TCP connection. The server accepts the order and closes the connection. (I'm leaving out a *lot* of details that would be necessary in a real-world system, such as the server sending a response code telling the client whether the order was accepted.) The orders that clients send look like this:

```
Name: John Smith
Product-ID: 67X-89
Address: 1280 Deniston Blvd, NY NY 10003
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
Card number: 4000-1234-5678-9017
Expires: 08/05
```

There's enough information in this message to let someone snooping packets use John Smith's credit card number for nefarious purposes. Consequently, before sending this order, you should encrypt it; the simplest way to do that without burdening either the server or the client with a lot of complicated, error-prone encryption code is to use a secure socket. The following code sends the order over a secure socket:

```
try {

  // This statement is only needed if you didn't add
  // security.provider.3=com.sun.net.ssl.internal.ssl.Provider
  // to your java.security file.
  Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider( ));

  SSLSocketFactory factory
   = (SSLSocketFactory) SSLSocketFactory.getDefault( );
  Socket socket = factory.createSocket("login.metalab.unc.edu", 7000);

  Writer out = new OutputStreamWriter(socket.getOutputStream( ),
   "ASCII");
  out.write("Name: John Smith\r\n");
  out.write("Product-ID: 67X-89\r\n");
  out.write("Address: 1280 Deniston Blvd, NY NY 10003\r\n");
  out.write("Card number: 4000-1234-5678-9017\r\n");
  out.write("Expires: 08/05\r\n");
  out.flush( );
  out.close( );
  socket.close( );

}
catch (IOException ex) {
  ex.printStackTrace( );
}
```

Only the first three statements are noticeably different from what you'd do with an insecure socket. The rest of the code just uses the normal methods of the `Socket`, `OutputStream`, and `Writer` classes.

Reading input is no harder. Example 11-1 is a simple program that connects to a secure HTTP server, sends a simple GET request, and prints out the response.

Example 11-1. HTTPSClient

```
import java.net.*;
import java.io.*;
import java.security.*;
import javax.net.ssl.*;
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
import com.macfaq.io.*;

public class HTTPSClient {

  public static void main(String[] args) {

    if (args.length == 0) {
      System.out.println("Usage: java HTTPSClient2 host");
      return;
    }

    int port = 443; // default https port
    String host = args[0];

    try {
      SSLSocketFactory factory
       = (SSLSocketFactory) SSLSocketFactory.getDefault( );

      SSLSocket socket = (SSLSocket) factory.createSocket(host, port);

      // enable all the suites
      String[] supported = socket.getSupportedCipherSuites( );
      socket.setEnabledCipherSuites(supported);

      Writer out = new OutputStreamWriter(socket.getOutputStream( ));
      // https requires the full URL in the GET line
      out.write("GET http://" + host + "/ HTTP/1.1\r\n");
      out.write("Host: " + host + "\r\n");
      out.write("\r\n");
      out.flush( );

      // read response
      BufferedReader in = new SafeBufferedReader(
        new InputStreamReader(socket.getInputStream( )));

      // read the header
      String s;
      while (!(s = in.readLine( )).equals("")) {
          System.out.println(s);
      }
      System.out.println( );

      // read the length
      String contentLength = in.readLine( );
      int length = Integer.MAX_VALUE;
      try {
        length = Integer.parseInt(contentLength.trim( ), 16);
      }
      catch (NumberFormatException ex) {
        // This server doesn't send the content-length
        // in the first line of the response body
      }
      System.out.println(contentLength);

      int c;
      int i = 0;
      while ((c = in.read( )) != -1 && i++ < length) {
        System.out.write(c);
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
          }

          System.out.println( );
          out.close( );
          in.close( );
          socket.close( );

        }
        catch (IOException ex) {
          System.err.println(ex);
        }

      }

    }
```

Here are the first few lines of output you get when you connect to the U.S. Postal Service's web site:

```
% java HTTPSClient www.usps.com
HTTP/1.1 200 OK
Server: Netscape-Enterprise/6.0
Date: Wed, 28 Jan 2004 18:13:08 GMT
Content-type: text/html
Set-Cookie: WEBTRENDS_ID=216.254.85.72-1075313584.16566; expires=Fri,
            31-Dec-2010 00:00:00 GMT; path=/
Transfer-Encoding: chunked

b6b
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=UTF-8">
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
  <HEAD>

    <link rel="stylesheet" href="/common/stylesheets/styles.css"
                                                    type="text/css">
    <TITLE>USPS - The United States Postal Service (U.S. Postal
                                                   Service)</TITLE>
```

When this program was tested for this edition, it initially refused to connect to www.usps.com because it couldn't verify the identity of the remote server. The problem was that the root certificates shipped with the version of the JDK I was using (1.4.2_02-b3) had expired. Upgrading to the latest minor version (1.4.2_03-b2) fixed the problem. If you see any exception messages like "No trusted certificate found", try upgrading to the latest minor version of either the JDK (if you're using 1.4 or later) or the JSSE (if you're using Java 1.3 or earlier).

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

One thing you may notice when you run this program is that it's slower to respond than you might expect. There's a noticeable amount of both CPU and network overhead involved in generating and exchanging the public keys. Even over a fast connection, it can easily take 10 seconds or more for the connection to be established. Consequently, you probably don't want to serve all your content over HTTPS, only the content that really needs to be private.

11.3. Methods of the SSLSocket Class

Besides the methods we've already discussed and those it inherits from `java.net.Socket`, the `SSLSocket` class has a number of methods for configuring exactly how much and what kind of authentication and encryption is performed. For instance, you can choose weaker or stronger algorithms, require clients to prove their identity, force reauthentication of both sides, and more.

11.3.1. Choosing the Cipher Suites

Different implementations of the JSSE support different combinations of authentication and encryption algorithms. For instance, the implementation Sun bundles with Java 1.4 only supports 128-bit AES encryption, whereas IAIK's iSaSiLk (http://jce.iaik.tugraz.at/products/02_isasilk/) supports 256-bit AES encryption. The `getSupportedCipherSuites()` method tells you which combination of algorithms is available on a given socket:

```
public abstract String[] getSupportedCipherSuites( )
```

However, not all cipher suites that are understood are necessarily allowed on the connection. Some may be too weak and consequently disabled. The `getEnabledCipherSuites()` method tells you which suites this socket is willing to use:

```
public abstract String[] getEnabledCipherSuites( )
```

The actual suite used is negotiated between the client and server at connection time. It's possible that the client and the server won't agree on any suite. It's also possible that although a suite is enabled on both client and server, one or the other or both won't have the keys and certificates needed to use the suite. In either case, the `createSocket( )` method will throw an `SSLException`, a subclass of `IOException`. You can change the suites the client attempts to use via the `setEnabledCipherSuites( )` method:

```
public abstract void setEnabledCipherSuites(String[] suites)
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

The argument to this method should be a list of the suites you want to use. Each name must be one of the suites listed by `getSupportedCipherSuites( )`. Otherwise, an `IllegalArgumentException` will be thrown. Sun's JDK 1.4 supports these 23 cipher suites:

- SSL_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_RC4_128_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_DSS_WITH_AES_128_CBC_SHA
- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_WITH_DES_CBC_SHA
- SSL_DHE_RSA_WITH_DES_CBC_SHA
- SSL_DHE_DSS_WITH_DES_CBC_SHA
- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
- SSL_RSA_WITH_NULL_MD5
- SSL_RSA_WITH_NULL_SHA
- SSL_DH_anon_WITH_RC4_128_MD5
- TLS_DH_anon_WITH_AES_128_CBC_SHA
- SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
- SSL_DH_anon_WITH_DES_CBC_SHA
- SSL_DH_anon_EXPORT_WITH_RC4_40_MD5
- SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA

Each name has an algorithm divided into four parts: protocol, key exchange algorithm, encryption algorithm, and checksum. For example, the name SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA means Secure Sockets Layer Version 3; Diffie-Hellman method for key agreement; no authentication; Data Encryption Standard encryption with 40-bit keys; Cipher Block Chaining, and the Secure Hash Algorithm checksum.

By default, the JDK 1.4 implementation enables all the encrypted authenticated suites (the first 15 members of this list). If you want nonauthenticated transactions or authenticated but unencrypted transactions, you must enable those suites explicitly with the `setEnabledCipherSuites( )` method.

Besides key lengths, there's an important difference between DES/AES and RC4-based ciphers. DES and AES are block ciphers; that is, they encrypt a certain number of bits at a time. DES always encrypts 64 bits. If 64 bits aren't available, the encoder has to pad the input with extra bits. AES can encrypt blocks of 128, 192, or 256 bits, but still has to pad the input if it

doesn't come out to an even multiple of the block size. This isn't a problem for file transfer applications such as secure HTTP and FTP, where more or less all the data is available at once. However, it's problematic for user-centered protocols such as chat and Telnet. RC4 is a stream cipher that can encrypt one byte at a time and is more appropriate for protocols that may need to send a single byte at a time.

For example, let's suppose that Edgar has some fairly powerful parallel computers at his disposal and can quickly break any encryption that's 64 bits or less and that Gus and Angela know this. Furthermore, they suspect that Edgar can blackmail one of their ISPs or the phone company into letting him tap the line, so they want to avoid anonymous connections that are vulnerable to man-in-the-middle attacks. To be safe, Gus and Angela decide to use at least 111-bit, authenticated encryption. It then behooves them to enable only the strongest available algorithms. This code fragment accomplishes that:

```
String[] strongSuites = {"SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA",
 "SSL_RSA_WITH_RC4_128_MD5", "SSL_RSA_WITH_RC4_128_SHA",
 "SSL_RSA_WITH_3DES_EDE_CBC_SHA"};
socket.setEnabledCipherSuites(strongSuites);
```

If the other side of the connection doesn't support strong encryption, the socket will throw an exception when they try to read from or write to it, thus ensuring that no confidential information is accidentally transmitted over a weak channel.

11.3.2. Event Handlers

Network communications are slow compared to the speed of most computers. Authenticated network communications are even slower. The necessary key generation and setup for a secure connection can easily take several seconds. Consequently, you may want to deal with the connection asynchronously. JSSE uses the standard event model introduced in Java 1.1 to notify programs when the handshaking between client and server is complete. The pattern is a familiar one. In order to get notifications of handshake-complete events, simply implement the `HandshakeCompletedListener` interface:

```
public interface HandshakeCompletedListener
 extends java.util.EventListener
```

This interface declares the `handshakeCompleted( )` method:

```
public void handshakeCompleted(HandshakeCompletedEvent event)
```

This method receives as an argument a `HandshakeCompletedEvent`:

```
public class HandshakeCompletedEvent extends java.util.EventObject
```

The `HandshakeCompletedEvent` class provides four methods for getting information about the event:

```
public SSLSession getSession( )
public String getCipherSuite( )
public X509Certificate[] getPeerCertificateChain( )
 throws SSLPeerUnverifiedException
public SSLSocket getSocket( )
```

Particular `HandshakeCompletedListener` objects register their interest in handshake-completed events from a particular `SSLSocket` via its `addHandshakeCompletedListener( )` and `removeHandshakeCompletedListener( )` methods:

```
public abstract void addHandshakeCompletedListener(
 HandshakeCompletedListener listener)
public abstract void removeHandshakeCompletedListener(
 HandshakeCompletedListener listener) throws IllegalArgumentException
```

11.3.3. Session Management

SSL is commonly used on web servers, and for good reason. Web connections tend to be transitory; every page requires a separate socket. For instance, checking out of Amazon.com on its secure server requires seven separate page loads, more if you have to edit an address or choose gift-wrapping. Imagine if every one of those pages took an extra 10 seconds or more to negotiate a secure connection. Because of the high overhead involved in handshaking between two hosts for secure communications, SSL allows *sessions* to be established that extend over multiple sockets. Different sockets within the same session use the same set of public and private keys. If the secure connection to Amazon.com takes seven sockets, all seven will be established within the same session and use the same keys. Only the first socket within that session will have to endure the overhead of key generation and exchange.

As a programmer using JSSE, you don't need to do anything extra to take advantage of sessions. If you open multiple secure sockets to one host on one port within a reasonably short period of time, JSSE will reuse the session's keys automatically. However, in high-security applications, you may want to disallow session-sharing between sockets or force reauthentication of a session. In the JSSE, sessions are represented by instances of the `SSLSession` interface; you can use the methods of this interface to check the times the session was created and last accessed, invalidate the session, and get various information about the session:

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
public byte[] getId( )
public SSLSessionContext getSessionContext( )
public long getCreationTime( )
public long getLastAccessedTime( )
public void invalidate( )
public void putValue(String name, Object value)
public Object getValue(String name)
public void removeValue(String name)
public String[] getValueNames( )
public X509Certificate[] getPeerCertificateChain( )
 throws SSLPeerUnverifiedException
public String getCipherSuite( )
public String getPeerHost( )
```

The `getSession( )` method of `SSLSocket` returns the `Session` this socket belongs to:

```
public abstract SSLSession getSession( )
```

However, sessions are a trade-off between performance and security. It is more secure to renegotiate the key for each and every transaction. If you've got really spectacular hardware and are trying to protect your systems from an equally determined, rich, motivated, and competent adversary, you may want to avoid sessions. To prevent a socket from creating a session that passes false to `setEnableSessionCreation( )`, use:

```
public abstract void setEnableSessionCreation(boolean allowSessions)
```

The `getEnableSessionCreation()` method returns `true` if multisocket sessions are allowed, `false` if they're not:

```
public abstract boolean getEnableSessionCreation( )
```

On rare occasions, you may even want to reauthenticate a connection; that is, throw away all the certificates and keys that have previously been agreed to and start over with a new session. The `startHandshake( )` method does this:

```
public abstract void startHandshake( ) throws IOException
```

11.3.4. Client Mode

It's a rule of thumb that in most secure communications, the server is required to authenticate itself using the appropriate certificate. However, the client is not. That is, when I buy a book from Amazon using its secure server, it has to prove to my browser's satisfaction that it is indeed Amazon and not Joe Random Hacker. However, I do not have to prove to Amazon that I am Elliotte Rusty Harold. For the most part, this is as it should be, since purchasing and

installing the trusted certificates necessary for authentication is a fairly user-hostile experience that readers shouldn't have to go through just to buy the latest Nutshell handbook. However, this asymmetry can lead to credit card fraud. To avoid problems like this, sockets can be required to authenticate themselves. This strategy wouldn't work for a service open to the general public. However, it might be reasonable in certain internal, high-security applications.

The `setUseClientMode()` method determines whether the socket needs to use authentication in its first handshake. The name of the method is a little misleading. It can be used for both client- and server-side sockets. However, when `true` is passed in, it means the socket is in client mode (whether it's on the client side or not) and will not offer to authenticate itself. When `false` is passed, it will try to authenticate itself:

```
public abstract void setUseClientMode(boolean mode)
 throws IllegalArgumentException
```

This property can be set only once for any given socket. Attempting to set it a second time throws an `IllegalArgumentException`.

The `getUseClientMode()` method simply tells you whether this socket will use authentication in its first handshake:

```
public abstract boolean getUseClientMode( )
```

A secure socket on the server side (that is, one returned by the `accept( )` method of an `SSLServerSocket`) uses the `setNeedClientAuth( )` method to require that all clients connecting to it authenticate themselves (or not):

```
public abstract void setNeedClientAuth(boolean needsAuthentication)
 throws IllegalArgumentException
```

This method throws an `IllegalArgumentException` if the socket is not on the server side.

The `getNeedClientAuth()` method returns `true` if the socket requires authentication from the client side, `false` otherwise:

```
public abstract boolean getNeedClientAuth( )
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

11.4. Creating Secure Server Sockets

fSecure client sockets are only half of the equation. The other half is SSL-enabled server sockets. These are instances of the `javax.net.SSLServerSocket` class:

```
public abstract class SSLServerSocket extends ServerSocket
```

Like `SSLSocket`, all the constructors in this class are protected. Like `SSLSocket`, instances of `SSLServerSocket` are created by an abstract factory class, `javax.net.SSLServerSocketFactory`:

```
public abstract class SSLServerSocketFactory
 extends ServerSocketFactory
```

Also like `SSLSocketFactory`, an instance of `SSLServerSocketFactory` is returned by a static `SSLServerSocketFactory.getDefault( )` method:

```
public static ServerSocketFactory getDefault( )
```

And like `SSLSocketFactory`, `SSLServerSocketFactory` has three overloaded `createServerSocket( )` methods that return instances of `SSLServerSocket` and are easily understood by analogy with the `java.net.ServerSocket` constructors:

```
public abstract ServerSocket createServerSocket(int port)
 throws IOException
public abstract ServerSocket createServerSocket(int port,
 int queueLength) throws IOException
public abstract ServerSocket createServerSocket(int port,
 int queueLength, InetAddress interface) throws IOException
```

If that were all there was to creating secure server sockets, they would be quite straightforward and simple to use. Unfortunately, that's not all there is to it. The factory that `SSLServerSocketFactory.getDefault( )` returns generally only supports server authentication. It does not support encryption. To get encryption as well, server-side secure sockets require more initialization and setup. Exactly how this setup is performed is implementation-dependent. In Sun's reference implementation, a `com.sun.net.ssl.SSLContext` object is responsible for creating fully configured and initialized secure server sockets. The details vary from JSSE implementation to JSSE implementation, but to create a secure server socket in the reference implementation, you have to:

- Generate public keys and certificates using *keytool*.

- Pay money to have your certificates authenticated by a trusted third party such as Verisign.
- Create an `SSLContext` for the algorithm you'll use.
- Create a `TrustManagerFactory` for the source of certificate material you'll be using.
- Create a `KeyManagerFactory` for the type of key material you'll be using.
- Create a `KeyStore` object for the key and certificate database. (Sun's default is JKS.)
- Fill the `KeyStore` object with keys and certificates; for instance, by loading them from the filesystem using the pass phrase they're encrypted with.
- Initialize the `KeyManagerFactory` with the `KeyStore` and its pass phrase.
- Initialize the context with the necessary key managers from the `KeyManagerFactory`, trust managers from the `TrustManagerFactory`, and a source of randomness. (The last two can be null if you're willing to accept the defaults.)

Example 11-2 demonstrates this procedure with a complete `SecureOrderTaker` for accepting orders and printing them on `System.out`. Of course, in a real application, you'd do something more interesting with the orders.

Example 11-2. SecureOrderTaker

```
import java.net.*;
import java.io.*;
import java.util.*;
import java.security.*;
import javax.net.ssl.*;
import javax.net.*;

public class SecureOrderTaker {

  public final static int DEFAULT_PORT = 7000;
  public final static String algorithm = "SSL";

  public static void main(String[] args) {

    int port = DEFAULT_PORT;
    if (args.length > 0) {
      try {
        port = Integer.parseInt(args[0]);
        if (port < 0 || port >= 65536) {
          System.out.println("Port must between 0 and 65535");
          return;
        }
      }
      catch (NumberFormatException ex) {}
    }

    try {

      SSLContext context = SSLContext.getInstance(algorithm);

      // The reference implementation only supports X.509 keys
      KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");

      // Sun's default kind of key store
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
            KeyStore ks = KeyStore.getInstance("JKS");

            // For security, every key store is encrypted with a
            // pass phrase that must be provided before we can load
            // it from disk. The pass phrase is stored as a char[] array
            // so it can be wiped from memory quickly rather than
            // waiting for a garbage collector. Of course using a string
            // literal here completely defeats that purpose.
            char[] password = "2andnotafnord".toCharArray( );
            ks.load(new FileInputStream("jnp3e.keys"), password);
            kmf.init(ks, password);

            //
            context.init(kmf.getKeyManagers( ), null, null);

            SSLServerSocketFactory factory
             = context.getServerSocketFactory( );

            SSLServerSocket server
             = (SSLServerSocket) factory.createServerSocket(port);

            String[] supported = server.getSupportedCipherSuites( );
            String[] anonCipherSuitesSupported = new String[supported.length];
            int numAnonCipherSuitesSupported = 0;
            for (int i = 0; i < supported.length; i++) {
              if (supported[i].indexOf("_anon_") > 0) {
                anonCipherSuitesSupported[numAnonCipherSuitesSupported++] =
                                                              supported[i];
              }
            }

            String[] oldEnabled = server.getEnabledCipherSuites( );
            String[] newEnabled = new String[oldEnabled.length
             + numAnonCipherSuitesSupported];
            System.arraycopy(oldEnabled, 0, newEnabled, 0, oldEnabled.length);
            System.arraycopy(anonCipherSuitesSupported, 0, newEnabled,
             oldEnabled.length, numAnonCipherSuitesSupported);

            server.setEnabledCipherSuites(newEnabled);
            // Now all the set up is complete and we can focus
            // on the actual communication.
            try {
              while (true) {
                // This socket will be secure,
                // but there's no indication of that in the code!
                Socket theConnection = server.accept( );
                InputStream in = theConnection.getInputStream( );
                int c;
                while ((c = in.read( )) != -1) {
                  System.out.write(c);
                }
                theConnection.close( );
              }  // end while
            } // end try
            catch (IOException ex) {
              System.err.println(ex);
            } // end catch
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
      }  // end try
      catch (IOException ex) {
        ex.printStackTrace( );
      } // end catch
      catch (KeyManagementException ex) {
        ex.printStackTrace( );
      } // end catch
      catch (KeyStoreException ex) {
        ex.printStackTrace( );
      } // end catch
      catch (NoSuchAlgorithmException ex) {
        ex.printStackTrace( );
      } // end catch
      catch (java.security.cert.CertificateException ex) {
        ex.printStackTrace( );
      } // end catch
      catch (UnrecoverableKeyException ex) {
        ex.printStackTrace( );
      } // end catch

    } // end main

  } // end server
```

This example loads the necessary keys and certificates from a file named *jnp3e.keys* in the current working directory protected with the password "2andnotafnord". What this example doesn't show you is how that file was created. It was built with the *keytool* program that's bundled with the JDK like this:

```
D:\JAVA>keytool -genkey -alias ourstore -keystore jnp3e.keys
Enter keystore password:  2andnotafnord
What is your first and last name?
  [Unknown]:  Elliotte
What is the name of your organizational unit?
  [Unknown]:  Me, Myself, and I
What is the name of your organization?
  [Unknown]:  Cafe au Lait
What is the name of your City or Locality?
  [Unknown]:  Brooklyn
What is the name of your State or Province?
  [Unknown]:  New York
What is the two-letter country code for this unit?
  [Unknown]:  NY
Is <CN=Elliotte, OU="Me, Myself, and I", O=Cafe au Lait, L=Brooklyn,
ST=New York, C=NY> correct?
  [no]:  y

Enter key password for <ourstore>
        (RETURN if same as keystore password):
```

When this is finished, you'll have a file named *jnp3e.keys*, which contains your public keys. However, no one will believe that these are your public keys unless you have them certified by a trusted third party such as Verisign (http://www.verisign.com/). Unfortunately, this certification costs money. The cheapest option is $14.95 per year for a Class 1 Digital ID.

Verisign hides the sign-up form for this kind of ID deep within its web site, apparently to get you to sign up for the much more expensive options that are prominently featured on its home page. At the time of this writing, the sign-up form is at https://www.verisign.com/client/. Verisign has changed this URL several times in the past, making it much harder to find than its more expensive options. In the more expensive options, Verisign goes to greater lengths to guarantee that you are who you say you are. Before signing up for any kind of digital ID, you should be aware that purchasing one has potentially severe legal consequences. In some jurisdictions, poorly thought-out laws make digital ID owners liable for all purchases made and contracts signed using their digital ID, regardless of whether the ID was stolen or forged. If you just want to explore the JSSE before deciding whether to go through the hassle, expense, and liability of purchasing a verified certificate, Sun includes a verified keystore file called *testkeys*, protected with the password "passphrase", that has some JSSE samples (http://java.sun.com/products/jsse/). However, this isn't good enough for real work.

For more information about exactly what's going on and what the various options are, as well as other ways to create key and certificate files, consult the online documentation for the *keytool* utility that came with your JDK, the Java Cryptography Architecture guide at http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html, or the previously mentioned books *Java Cryptography*, by Jonathan Knudsen, or *Java Security*, by Scott Oaks (both from O'Reilly).

Another approach is to use cipher suites that don't require authentication. There are six of these in Sun's JDK 1.4: SSL_DH_anon_WITH_RC4_128_MD5, TLS_DH_anon_WITH_AES_128_CBC_SHA, SSL_DH_anon_WITH_3DES_EDE_CBC_SHA, SSL_DH_anon_WITH_DES_CBC_SHA, SSL_DH_anon_EXPORT_WITH_RC4_40_MD5, and SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA.

These are not enabled by default because they're vulnerable to a man-in-the-middle attack, but at least they allow you to write simple programs without paying Verisign any money.

11.5. Methods of the SSLServerSocket Class

Once you've successfully created and initialized an `SSLServerSocket`, there are a lot of applications you can write using nothing more than the methods inherited from `java.net.ServerSocket`. However, there are times when you need to adjust its behavior a little. Like `SSLSocket`, `SSLServerSocket` provides methods to choose cipher suites, manage sessions, and establish whether clients are required to authenticate themselves. Most of these methods are very similar to the methods of the same name in `SSLSocket`.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

The difference is that they work on the server side and set the defaults for sockets accepted by an SSLServerSocket. In some cases, once an SSLSocket has been accepted, you can still use the methods of SSLSocket to configure that one socket rather than all sockets accepted by this SSLServerSocket.

11.5.1. Choosing the Cipher Suites

The SSLServerSocket class has the same three methods for determining which cipher suites are supported and enabled as SSLSocket does:

```
public abstract String[] getSupportedCipherSuites( )
public abstract String[] getEnabledCipherSuites( )
public abstract void     setEnabledCipherSuites(String[] suites)
```

These methods use the same suite names as the similarly named methods in SSLSocket. The difference is that these methods apply to all sockets accepted by the SSLServerSocket rather than to just one SSLSocket. For example, this code fragment has the effect of enabling anonymous, unauthenticated connections on the SSLServerSocket server. It relies on the names of these suites containing the string "_anon_". This is true for Sun's reference implementations, though there's no guarantee that other implementers will follow this convention:

```
String[] supported = server.getSupportedCipherSuites( );
String[] anonCipherSuitesSupported = new String[supported.length];
int numAnonCipherSuitesSupported = 0;
for (int i = 0; i < supported.length; i++) {
  if (supported[i].indexOf("_anon_") > 0) {
    anonCipherSuitesSupported[numAnonCipherSuitesSupported++]
     = supported[i];
  }
}

String[] oldEnabled = server.getEnabledCipherSuites( );
String[] newEnabled = new String[oldEnabled.length
 + numAnonCipherSuitesSupported];
System.arraycopy(oldEnabled, 0, newEnabled, 0, oldEnabled.length);
System.arraycopy(anonCipherSuitesSupported, 0, newEnabled,
 oldEnabled.length, numAnonCipherSuitesSupported);

server.setEnabledCipherSuites(newEnabled);
```

This fragment retrieves the list of both supported and enabled cipher suites using getSupportedCipherSuites( ) and getEnabledCipherSuites( ). It looks at the name of every supported suite to see whether it contains the substring "_anon_". If the suite name does contain this substring, the suite is added to a list of anonymous cipher suites. Once the list of anonymous cipher suites is built, it's combined in a new array with the

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

previous list of enabled cipher suites. The new array is then passed to `setEnabledCipherSuites( )` so that both the previously enabled and the anonymous cipher suites can now be used.

11.5.2. Session Management

Both client and server must agree to establish a session. The server side uses the `setEnableSessionCreation( )` method to specify whether this will be allowed and the `getEnableSessionCreation()` method to determine whether this is currently allowed:

```
public abstract void setEnableSessionCreation(boolean allowSessions)
public abstract boolean getEnableSessionCreation( )
```

Session creation is enabled by default. If the server disallows session creation, then a client that wants a session will still be able to connect. It just won't get a session and will have to handshake again for every socket. Similarly, if the client refuses sessions but the server allows them, they'll still be able to talk to each other but without sessions.

11.5.3. Client Mode

The `SSLServerSocket` class has two methods for determining and specifying whether client sockets are required to authenticate themselves to the server. By passing `true` to the `setNeedClientAuth( )` method, you specify that only connections in which the client is able to authenticate itself will be accepted. By passing `false`, you specify that authentication is not required of clients. The default is `false`. If for some reason you need to know what the current state of this property is, the `getNeedClientAuth()` method will tell you:

```
public abstract void setNeedClientAuth(boolean flag)
public abstract boolean getNeedClientAuth( )
```

The `setUseClientMode( )` method allows a program to indicate that even though it has created an `SSLServerSocket`, it is and should be treated as a client in the communication with respect to authentication and other negotiations. For example, in an FTP session, the client program opens a server socket to receive data from the server, but that doesn't make it less of a client. The `getUseClientMode( )` method returns `true` if the `SSLServerSocket` is in client mode, `false` otherwise:

```
public abstract void setUseClientMode(boolean flag)
public abstract boolean getUseClientMode( )
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004