Chapter 8

Secure Socket

Contents

1. Secure Communication
2. Creating Secure Client Sockets
3. Event Handlers
4. Session Management
5. Client Mode
6. Creating Secure Server Socket
7. Cofigure SSLServerSocket: Choosing the Cipher Suits, Session Management and Client Mode

Secure Communication

- Secure communication refers to the exchange of information between two or more parties in a way that is protected from interception, eavesdropping, and unauthorized access.
- The need for secure communication arises in situations where sensitive or confidential information needs to be transmitted, such as in business, government, military, healthcare, and financial sectors.
- 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 plain-text message is combined with the bits of the key according to a mathematical algorithm to produce the encrypted ciphertext.

- 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 to encrypt and decrypt the data.
- Both the sender and the receiver have to know the single key.
- In **public key** (or **asymmetric encryption**), different keys are used to encrypt and decrypt the data.
- One key, called the public key, encrypts 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.
- JSSE allows you to create sockets and server sockets that transparently handle the negotiations and encryption necessary for secure communication.

➢ The **Java Secure Socket Extension(JSSE)** 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.
- java.security.cert
  - The classes for handling the public-key certificates needed for SSL.
- 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.

Creating Secure Client Sockets

- Rather than constructing a java.net.Socket object with a constructor, you get one from a javax.net.ssl.SSLSocketFactory using its **createSocket()** method
- You get an instance of it by invoking the static SSLSocketFactory.getDefault() method:
- Example:

```
SocketFactory factory = SSLSocketFactory.getDefault();
Socket socket = factory.createSocket("login.ibiblio.org", 7000);
```

- 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:

1. public abstract Socket createSocket(String host, int port) throws IOException, UnknownHostException
2. public abstract Socket createSocket(InetAddress host, int port) throws IOException
3. public abstract Socket createSocket(String host, int port, InetAddress interface, int localPort) throws IOException, UnknownHostException
4. public abstract Socket createSocket(InetAddress host, int port, InetAddress interface, int localPort) throws IOException, UnknownHostException
5. public abstract Socket createSocket(Socket proxy, String host, int port, boolean autoClose) throws IOException

➢ 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

Q. Secure Client Socket Program (Read from a secure server socket)

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

public class SecureClient {
    public static void main(String[] args) {
        int port = 443; // default https port
        String host = "merojob.com";
        SSLSocketFactory factory = (SSLSocketFactory)
SSLSocketFactory.getDefault();
        SSLSocket socket = null;
        try {
            socket = (SSLSocket) factory.createSocket(host, port);
```

```
// tells you which combination of algorithms is available on a given socket
String[] supported = socket.getSupportedCipherSuites();

// enable all the suites
socket.setEnabledCipherSuites(supported);
Writer out = new OutputStreamWriter(socket.getOutputStream(), "UTF-8");

// 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 BufferedReader(
    new InputStreamReader(socket.getInputStream()));
```

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

        } catch (IOException ex) {
            System.err.println(ex);
        } finally {
            try {
                if (socket != null)
                    socket.close();
            } catch (IOException e) {
            }
        }
    }
```

Choosing the Cipher Suites

- ➢ Different implementations of the JSSE support different combinations of authentication and encryption algorithms.
- ➢ For instance, the implementation that Oracle bundles with Java 7 only supports 128-bit AES encryption, whereas IAIK's iSaSiLk supports 256-bit AES encryption
- ➢ The **getSupportedCipherSuites**() method in SSLSocketFactory 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 of SSLSocketFactory tells you which suites this socket is willing to use:

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

➢ You can change the suites the client attempts to use via the setEnabledCipherSuites() method:

public abstract void setEnabledCipherSuites(String[] suites)

➢ 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()**.

➢ Oracle's JDK 1.7 supports these cipher suites:

I. TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256

II. TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256

III. TLS_RSA_WITH_AES_128_CBC_SHA256

IV. TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256

V. TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256

VI. TLS_DHE_RSA_WITH_AES_128_CBC_SHA256

VII. TLS_DHE_DSS_WITH_AES_128_CBC_SHA256

VIII. TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA

IX. TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA

X. TLS_RSA_WITH_AES_128_CBC_SHA

XI. TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA

XII. TLS_ECDH_RSA_WITH_AES_128_CBC_SHA

XIII. TLS_DHE_RSA_WITH_AES_128_CBC_SHA

XIV. TLS_DHE_DSS_WITH_AES_128_CBC_SHA

XV. TLS_ECDHE_ECDSA_WITH_RC4_128_SHA

XVI. TLS_ECDHE_RSA_WITH_RC4_128_SHA

XVII. SSL_RSA_WITH_RC4_128_SHA

XVIII. TLS_ECDH_ECDSA_WITH_RC4_128_SHA

XIX. TLS_ECDH_RSA_WITH_RC4_128_SHA

XX. TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA

XXI. TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA

XXII. SSL_RSA_WITH_3DES_EDE_CBC_SHA

XXIII. TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA

XXIV. TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA

XXV. SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA

XXVI. SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA

XXVII. SSL_RSA_WITH_RC4_128_MD5

XXVIII. TLS_EMPTY_RENEGOTIATION_INFO_SCSV

XXIX. TLS_DH_anon_WITH_AES_128_CBC_SHA256

XXX. TLS_ECDH_anon_WITH_AES_128_CBC_SHA

XXXI. TLS_DH_anon_WITH_AES_128_CBC_SHA

XXXII. TLS_ECDH_anon_WITH_RC4_128_SHA

XXXIII. SSL_DH_anon_WITH_RC4_128_MD5

XXXIV. TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA

XXXV. SSL_DH_anon_WITH_3DES_EDE_CBC_SHA

- Each name has an algorithm divided into four parts:
    1. protocol,
    2. key exchange algorithm,
    3. encryption algorithm, and
    4. checksum.
- ***For example***, the name SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA means

- Secure Sockets Layer Version 3;
- DiffieHellman method for key agreement;
- no authentication;
- Data Encryption Standard encryption with 40-bit keys;
- Cipher Block Chaining,
- and the Secure Hash Algorithm checksum.

➢ This code fragment limits their connection to that one suite:

```
String[] strongSuites =
{"TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA25
6"};
socket.setEnabledCipherSuites(strongSuites);
```

Event Handler

➢ 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 Java event model 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:

1. public SSLSession getSession()
2. public String getCipherSuite()
3. public X509Certificate[] getPeerCertificateChain() throws SSLPeerUnverifiedException
4. public SSLSocket getSocket()

➢ Particular HandshakeCompletedListener objects register their interest in handshakecompleted 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

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
- 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 highsecurity 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:

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()

➢ 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 (i.e., 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

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.
- ➢ 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.

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 (i.e., 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()
```

Creating Secure Server Sockets

Secure 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 and instances 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 create ServerSocket() 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 straight forward and simple to use.