NETWORK PROGRAMMING

CHAPTER 8: SECURE SOCKETS

CHANDAN GUPTA BHAGAT

https://me.chandanbhagat.com.np

CONTENT

- Secure Communication
- Creating Secure Client Sockets
- Event Handlers
- Session Management
- Client Mode
- Creating Secure Server Socket
- Configure SSLServerSockets : Choosing the Cipher Suits, Sesion Management, Client Mode

Secure Sockets

- 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, 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

Secure Sockets

- The Java Secure Sockets Extension (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 and other TCP clients to talk to HTTP and other TCP servers using various levels of confidentiality and authentication.

Secure Communication

Secure Communication

- 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.
- *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.*

Secure Communication

- 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.
- *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, because 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 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, because 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.*

Secure Communication

- 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. (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.*

Secure Communication

- 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.
- 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. 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.

Creating Secure Client Sockets

Creating Secure Client Sockets

- If you don’t care very much about the underlying details, using an encrypted SSL socketto 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. You get an instance of it by invoking the static SSLSocketFactory.getDefault() method:

```
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:

Creating Secure Client Sockets

```
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
```

Creating Secure Client Sockets

- 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 getInput Stream(), getOutputStream(), and other methods. For example, suppose a server that accepts orders is listening on port 7000 of *login.ibiblio.org*. 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.)

Creating Secure Client Sockets

- The orders that clients send look like this:

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

```
SSLSocketFactory factory = (SSLSocketFactory) SSLSocketFactory.getDefault();
Socket socket = factory.createSocket("login.ibiblio.org", 7000);
Writer out = new OutputStreamWriter(socket.getOutputStream(), "US-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();
```

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 getEnabledCipher Suites() method of SSLSocketFactory 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)
```

- 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.

Choosing the Cipher Suites

- Oracle's JDK 1.7 supports these cipher suites:
  - TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
  - TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
  - TLS_RSA_WITH_AES_128_CBC_SHA25
  - TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256
  - TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256
  - TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
  - TLS_DHE_DSS_WITH_AES_128_CBC_SHA256
  - TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
  - TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
  - TLS_RSA_WITH_AES_128_CBC_SHA
  - TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA
  - TLS_ECDH_RSA_WITH_AES_128_CBC_SHA
  - TLS_DHE_RSA_WITH_AES_128_CBC_SHA
  - TLS_DHE_DSS_WITH_AES_128_CBC_SHA
  - TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
  - TLS_ECDHE_RSA_WITH_RC4_128_SHA
  - SSL_RSA_WITH_RC4_128_SHA
  - TLS_ECDH_ECDSA_WITH_RC4_128_SHA
  - TLS_ECDH_RSA_WITH_RC4_128_SHA
  - TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
  - TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
  - SSL_RSA_WITH_3DES_EDE_CBC_SHA
  - TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA
  - TLS_ECDH_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_RC4_128_MD5
  - TLS_EMPTY_RENEGOTIATION_INFO_SCSV
  - TLS_DH_anon_WITH_AES_128_CBC_SHA256
  - TLS_ECDH_anon_WITH_AES_128_CBC_SHA
  - TLS_DH_anon_WITH_AES_128_CBC_SHA
  - TLS_ECDH_anon_WITH_RC4_128_SHA
  - SSL_DH_anon_WITH_RC4_128_MD5
  - TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA
  - SSL_DH_anon_WITH_3DES_EDE_CBC_SHA

TLS_RSA_WITH_NULL_SHA256

  - TLS_ECDHE_ECDSA_WITH_NULL_SHA
  - TLS_ECDHE_RSA_WITH_NULL_SHA
  - SSL_RSA_WITH_NULL_SHA
  - TLS_ECDH_ECDSA_WITH_NULL_SHA
  - TLS_ECDH_RSA_WITH_NULL_SHA
  - TLS_ECDH_anon_WITH_NULL_SHA
  - SSL_RSA_WITH_NULL_MD5
  - SSL_RSA_WITH_DES_CBC_SHA
  - SSL_DHE_RSA_WITH_DES_CBC_SHA
  - SSL_DHE_DSS_WITH_DES_CBC_SHA
  - SSL_DH_anon_WITH_DES_CBC_SHA
  - SSL_RSA_EXPORT_WITH_RC4_40_MD5
  - SSL_DH_anon_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_DH_anon_EXPORT_WITH_DES40_CBC_SHA
  - TLS_KRB5_WITH_RC4_128_SHA
  - TLS_KRB5_WITH_RC4_128_MD5
  - TLS_KRB5_WITH_3DES_EDE_CBC_SHA
  - TLS_KRB5_WITH_3DES_EDE_CBC_MD5
  - TLS_KRB5_WITH_DES_CBC_SHA
  - TLS_KRB5_WITH_DES_CBC_MD5
  - TLS_KRB5_EXPORT_WITH_RC4_40_SHA
  - TLS_KRB5_EXPORT_WITH_RC4_40_MD5
  - TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA
  - TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5

Choosing the Cipher Suites

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

```
String[] strongSuites = {"TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256"};
socket.setEnabledCipherSuites(strongSuites);
```

- If the other side of the connection doesn’t support this encryption protocol, 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

Event Handlers

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 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
```

Event Handlers

- 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 handshakecompleted events from a particular SSLSocket via its addHandshakeCompletedListen er() and removeHandshakeCompletedListener() methods:

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

Session Management

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.

Session Management

- 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()
```

Session Management

- 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 (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

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. To avoid problems like credit card fraud, 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.

Client Mode

- 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 Socket

Creating Secure Server Socket

- 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
```

Creating Secure Server Socket

- 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:
  1. Generate public keys and certificates using *keytool*.
  2. Pay money to have your certificates authenticated by a trusted third party such as Comodo.
  3. Create an SSLContext for the algorithm you'll use.
  4. Create a TrustManagerFactory for the source of certificate material you'll be using.
  5. Create a KeyManagerFactory for the type of key material you'll be using.

Creating Secure Server Socket

6. Create a KeyStore object for the key and certificate database. (Oracle's default is JKS.)

7. Fill the KeyStore object with keys and certificates; for instance, by loading them from the filesystem using the passphrase they're encrypted with.

8. Initialize the KeyManagerFactory with the KeyStore and its passphrase.

9. 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.)

Creating Secure Server Socket

- Another approach is to use cipher suites that don't require authentication. There are several of these in the JDK, including:
  - SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA
  - SSL_DH_anon_EXPORT_WITH_RC4_40_MD5
  - SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
  - SSL_DH_anon_WITH_DES_CBC_SHA
  - SSL_DH_anon_WITH_RC4_128_MD5
  - TLS_DH_anon_WITH_AES_128_CBC_SHA
  - TLS_DH_anon_WITH_AES_128_CBC_SHA256
  - TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA
  - TLS_ECDH_anon_WITH_AES_128_CBC_SHA
  - TLS_ECDH_anon_WITH_NULL_SHA
  - TLS_ECDH_anon_WITH_RC4_128_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 money.

Configure SSLServerSockets

Configure SSLServerSockets

- 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 similar to the methods of the same name in SSLSocket. 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.

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 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, the following 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 Oracle's reference implementations, though there's no guarantee that other implementers will follow this convention:

Choosing the Cipher Suites

```
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);
```

Choosing the Cipher Suites

- This fragment retrieves the list of both supported and enabled cipher suites using get SupportedCipherSuites() 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 previous list of enabled cipher suites.
- The new array is then passed to setEnabledCipher Suites() so that both the previously enabled and the anonymous cipher suites can now be used.

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.

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()
```

THANK YOU

YOU CAN FIND THIS SLIDES/NOTES IN

https://chandanbhagat.com.np/docs/network-programming/

AND CODES IN

https://github.com/chandan-g-bhagat/network-programming