UNIT-8: SECURE SOCKETS

- Secure Sockets in Java can be implemented using the **Java Secure Socket Extension** (JSSE) API. The JSSE API provides a **set of classes** for implementing SSL/TLS in Java applications.
- To use SSL/TLS with JSSE, you first need to obtain **a digital certificate for your server**. The certificate is used to **verify the identity of the server to clients**. You can obtain a certificate from a certificate authority or **create a self-signed certificate using the Java keytool utility**.
- Once you have a certificate, you can use the JSSE API to create **an SSL socket factory**, which can be used to create SSL sockets. Here is an example code snippet for creating an SSL socket:

UNIT-8: SECURE COMMUNICATIONS

- Secure communications refers to the practice of ensuring that communication between two parties is protected from eavesdropping or interception by unauthorized parties. Secure communication is important in a wide range of contexts, including online transactions, military communications, and personal privacy.
- There are several requirements for achieving secure communications:
- Confidentiality: This means that the content of the communication must be protected from unauthorized access. Confidentiality can be achieved through encryption, which scrambles the message in a way that can only be deciphered by the intended recipient.
- Integrity: This means that the content of the communication must not be altered or tampered with during transmission. Integrity can be achieved through the use of cryptographic checksums, which allow the recipient to verify that the message has not been altered.
- Authentication: This means that the identity of the sender and/or recipient must be verified to prevent impersonation or "man-in-the-middle" attacks. Authentication can be achieved through the use of digital signatures or other forms of identity verification.
- Non-repudiation: This means that the sender cannot deny sending a message, and the recipient cannot deny receiving it. Non-repudiation can be achieved through the use of digital signatures, which provide a verifiable record of the message and the sender's identity.
- Availability: This means that the communication must be available to the intended recipient when it is needed. Availability can be achieved through the use of redundant systems or backup communication channels.

UNIT-8: SECURE COMMUNICATIONS

- **Encryption and decryption** are two important concepts in cryptography, which is the practice of secure communication in the presence of third parties. Encryption is the process of converting plain text into an encoded form that is unintelligible to anyone except the intended recipient. Decryption is the process of converting the encoded text back into plain text.
- There are two main types of encryption: symmetric key encryption and asymmetric key encryption.
- **Symmetric key encryption**, also known as secret key encryption, uses the same key for both encryption and decryption. The key is shared between the sender and the recipient of the message, and it must be kept secret from anyone else who might intercept the message. Symmetric key encryption is fast and efficient, but it requires a secure method for distributing the secret key.
- **Asymmetric key encryption**, also known as public key encryption, uses a pair of keys: a public key and a private key. The public key is used to encrypt the message, and the private key is used to decrypt it. The public key can be freely distributed to anyone who wants to send a message to the owner of the private key, but the private key must be kept secret. Asymmetric key encryption is slower than symmetric key encryption, but it provides a more secure method for exchanging keys.
- Symmetric Key Encryption:
  - Uses the **same key** for both encryption and decryption
  - Key is shared between sender and recipient
  - Fast and efficient
  - Requires a secure method for distributing the secret key
- Asymmetric Key Encryption:
  - **Uses a pair of keys**: a public key for encryption and a private key for decryption
  - Public key can be freely distributed, but private key must be kept secret
  - Slower than symmetric key encryption
  - Provides a more secure method for exchanging keys

UNIT-8: THE COMMON CLASSES AND PACKAGES FOR IMPLEMENTING SECURE SOCKETS

The common classes and packages for implementing secure sockets in Java are:

- **javax.net.ssl:** This package provides the classes and interfaces for creating and configuring SSL/TLS sockets and engines. This includes classes for SSL/TLS protocols, cipher suites, key managers, and trust managers.
- **javax.net**: This package provides the abstract socket factory classes that are used to create secure sockets. These classes are used instead of constructors to create sockets that support SSL/TLS protocols.
- **java.security.cert**: This package provides classes for handling X.509 digital certificates, which are commonly used for authentication and encryption in SSL/TLS protocols. These classes include Certificate, X509Certificate, CertificateFactory, CertPath, and CertPathValidator.
- **java.security**: This package provides the general security framework for Java, including cryptographic services and key management. This includes classes for generating and managing keys, as well as classes for secure random number generation and message digests.

CREATING SECURE CLIENT SOCKETS

```
SSLSocketFactory socketFactory = (SSLSocketFactory) SSLSocketFactory.getDefault();
SSLSocket socket = (SSLSocket) socketFactory.createSocket("localhost", 8000);
```

```
import javax.net.ssl.SSLSocketFactory;
import java.io.IOException;
import java.net.Socket;
public class Testtt {
    public static void main(String[] args) {
        Run | Debug
        try {
            // Create the SSL socket factory
            SSLSocketFactory factory = (SSLSocketFactory) SSLSocketFactory.getDefault();
            // Create a secure socket
            Socket socket = factory.createSocket("localhost", 7000);
            // Use the socket for further communication
            // ...

            // Close the socket
            socket.close();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

CREATING SECURE CLIENT SOCKETS

- In the example above, we create an **SSLSocketFactory** using **SSLSocketFactory.getDefault(),** which returns the **default SSL socket factory** provided by the Java runtime. Then, we use this factory to create an SSLSocket by calling factory.createSocket("localhost", port) with the desired host (in this case, "localhost") and port.
- After creating the secure socket, you can use the socket object for further communication, such as reading from and writing to the socket's input and output streams.
- Finally, make sure to handle any potential IOException that may occur during socket creation or communication. The provided example includes a basic try-catch block to catch and print any exception that occurs.

CREATING SECURE CLIENT SOCKETS

```java
import javax.net.ssl.*;
import java.io.*;
public class SSLClient {
    Run | Debug
    public static void main(String[] args) {
        try {
            // Set up SSL context
            System.setProperty(key:"javax.net.ssl.trustStore", value:"server.truststore");
            System.setProperty(key:"javax.net.ssl.trustStorePassword", value:"aaaaaa");
            // Create SSL socket
            SSLSocketFactory socketFactory = (SSLSocketFactory) SSLSocketFactory.getDefault();
            SSLSocket socket = (SSLSocket) socketFactory.createSocket("localhost", 8000);
            System.out.println(x:"Connected to server.");
            // Create input and output streams
            DataInputStream in = new DataInputStream(socket.getInputStream());
            DataOutputStream out = new DataOutputStream(socket.getOutputStream());
            // Send a message to the server
            out.writeUTF(str:"Hello, server!");
            // Read the server's response
            String response = in.readUTF();
            System.out.println("Received response from server: " + response);
            // Close the socket
            socket.close();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

CREATING SECURE SERVER SOCKETS

- To create a secure server socket in Java, you can use the **javax.net.ssl.SSLServerSocket** class, which provides a way to listen for secure incoming connections over the network.

```
// Set up SSL server socket
SSLServerSocketFactory serverSocketFactory = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
SSLServerSocket serverSocket = (SSLServerSocket) serverSocketFactory.createServerSocket(8000);
```

```
/*
 * to generate the key:
 * keytool -keystore "C:\Program Files\Java\jre-1.8\lib\cacerts"
 * -import -alias https://www.facebook.com
 * -file "C:\Users\acer\Downloads\cert\demo.crt"
 */
```

CREATING SECURE SERVER SOCKETS

```
import javax.net.ssl.*;
import java.io.*;
public class SSLServer {
    Run | Debug
    public static void main(String[] args) {
        try {
            // Set up SSL server socket
            SSLServerSocketFactory serverSocketFactory = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault()
            SSLServerSocket serverSocket = (SSLServerSocket) serverSocketFactory.createServerSocket(8000);

            // Set up truststore for server
            System.setProperty(key:"javax.net.ssl.trustStore", value:"server.truststore");
            System.setProperty(key:"javax.net.ssl.trustStorePassword", value:"aaaaaa");

            System.out.println(x:"Server started on port 8000.");
```

```
/*
* to generate the key:
* keytool -keystore "C:\Program Files\Java\jre-1.8\lib\cacerts"
* -import -alias https://www.facebook.com
* -file "C:\Users\acer\Downloads\cert\demo.crt"
*/
```

CREATING SECURE SERVER SOCKETS

```
            System.out.println(x:"Server started on port 8000.");
            while (true) {
                // Wait for a client to connect
                SSLSocket socket = (SSLSocket) serverSocket.accept();
                System.out.println("Client connected: " + socket.getInetAddress());
                // Create input and output streams
                DataInputStream in = new DataInputStream(socket.getInputStream());
                DataOutputStream out = new DataOutputStream(socket.getOutputStream());
                // Read a message from the client
                String message = in.readUTF();
                System.out.println("Received message from client: " + message);
                // Send a response to the client
                out.writeUTF(str:"Hello, client!");
                // Close the socket and server socket
                socket.close();
            }
            // serverSocket.close();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

EVENT HANDLERS

- To handle events in a secure socket connection in Java, you can use event handlers provided by the **javax.net.ssl.SSLSocket class**. Here are a few common event handlers that you can use:

- **HandshakeCompletedListener**: This event handler is triggered when the SSL handshake process is completed successfully. You can use it to perform actions after the handshake, such as verifying the client's certificate or establishing application-specific protocols.

```
SSLSocket socket = (SSLSocket) serverSocket.accept();
socket.addHandshakeCompletedListener(new HandshakeCompletedListener() {
    @Override
    public void handshakeCompleted(HandshakeCompletedEvent event) {
        // Handshake completed, perform post-handshake actions
        // Access the negotiated session parameters, peer's certificate, etc. using event.get* methods
    }
});
```

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()
```

SESSION MANAGEMENT

- SSL session management in Java involves managing and utilizing the SSL sessions created during SSL/TLS connections.
- SSL sessions represent the ongoing secure connection between the client and the server and provide various information about the connection.

1. **getId**(): This method returns a byte array that represents the unique identifier for the SSL session.
2. **getSessionContext**(): This method returns the SSLSessionContext object associated with the SSL session. The SSLSessionContext represents the context in which SSL sessions are created and managed.
3. **getCreationTime**(): This method returns the time (in milliseconds since the epoch) when the SSL session was created.
4. **getLastAccessedTime**(): This method returns the time (in milliseconds since the epoch) when the SSL session was last accessed.
5. **invalidate**(): This method invalidates the SSL session, making it no longer usable. The session will be removed from the session cache and subsequent requests will require a new SSL handshake.
6. **putValue**(String name, Object value): This method associates a value with a specified name in the SSL session. The value can be retrieved later using the same name.
7. **getValue**(String name): This method retrieves the value associated with the specified name in the SSL session.
8. **removeValue**(String name): This method removes the value associated with the specified name from the SSL session.
9. **getValueNames**(): This method returns an array of names associated with values in the SSL session.
10. **getPeerCertificateChain**() throws SSLPeerUnverifiedException: This method returns the peer's certificate chain as an array of X509Certificate objects. It throws an SSLPeerUnverifiedException if the peer's identity has not been verified.
11. **getCipherSuite**(): This method returns the name of the cipher suite used in the SSL session.
12. **getPeerHost**(): This method returns the host name of the peer to which the SSL session is connected.

CLIENT MODE

- In SSL/TLS communication, the client mode refers to the role that a particular SSL socket or SSL context plays. In Java, you can configure an SSL socket or SSL context to operate in client mode using the setUseClientMode(boolean mode) method. Here's how it works:

SSLSocket socket = (SSLSocket) serverSocket.accept();

- setUseClientMode(boolean mode): This method is used to set the mode of an SSL socket or SSL context. When mode is true, **it sets the SSL socket or context to client mode**. When mode is false, it sets the SSL socket or **context to server mode**.
  ex. **socket.setUseClientMode(true);** // Set the socket to client mode
- getUseClientMode(): This method returns a boolean value indicating whether the SSL socket is set to client mode (true) or server mode (false).

Ex **boolean isClientMode = socket.getUseClientMode();** // Check if socket is in client mode

- setNeedClientAuth(boolean needsAuthentication): This method sets whether the SSL socket requires client authentication. If needsAuthentication is true, the **server requests client authentication**. If needsAuthentication is false, client authentication is optional.

ex. socket.setNeedClientAuth(true); // Require client authentication

- **getNeedClientAuth():** This method returns a boolean value indicating whether client authentication is required (true) or optional (false) for the SSL socket.

Ex. boolean needsClientAuth = socket.getNeedClientAuth(); // Check if client authentication is required

CONFIGURING SSLSERVERSOCKETS

SSLServerSocket is a subclass of ServerSocket in Java that provides secure communication over SSL/TLS protocols. It implements the ServerSocket interface and adds support for SSL/**TLS encryption and authentication**. With SSLServerSocket, you can establish secure server sockets that allow clients to connect over SSL/TLS and protect sensitive data in transit.

```
SSLServerSocketFactory serverSocketFactory = SSLServerSocketFactory.getDefault();
SSLServerSocket serverSocket = (SSLServerSocket) serverSocketFactory.createServerSocket("localhost",8000);
serverSocket.setEnabledProtocols(new String[] {"TLSv1.2", "TLSv1.3"});
String[] cipherSuites = serverSocket.getSupportedCipherSuites();
serverSocket.setEnabledCipherSuites(cipherSuites);
SSLSocket clientSocket = (SSLSocket) serverSocket.accept();
```

CHOOSING THE CIPHER SUITES

In Java, the available cipher suites for SSL/TLS connections depend on the version of Java and the underlying cryptographic providers available. However, I can provide you with a list of commonly supported cipher suites in Java.

Here are some examples of cipher suites commonly used in Java for SSL/TLS connections:

TLS_RSA_WITH_AES_128_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_AES_128_GCM_SHA256
TLS_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

```java
import javax.net.ssl.SSLServerSocketFactory;
public class CipherSuit {
    Run | Debug
    public static void main(String[] args) {
        String[] cipherSuites = ((SSLServerSocketFactory)
        SSLServerSocketFactory.getDefault()).getSupportedCipherSuites();
        for (String suite : cipherSuites) {
            System.out.println(suite);
        }
    }
}
```

Program to display all cipher suites available in your system

CHOOSING THE CIPHER SUITES

- **getSupportedCipherSuites**(): This method returns an array of cipher suites supported by the SSL/TLS implementation. These cipher suites represent the available encryption algorithms, key exchange methods, and message authentication codes (MACs). You can use this method to obtain the list of supported cipher suites.

```
String[] supportedCipherSuites = sslSocket.getSupportedCipherSuites();
```

- **getEnabledCipherSuites**(): This method returns an array of cipher suites that are currently enabled on the SSL/TLS socket. Initially, all supported cipher suites are enabled by default. You can use this method to check the currently enabled cipher suites.

```
String[] enabledCipherSuites = sslSocket.getEnabledCipherSuites();
```

- **setEnabledCipherSuites**(String[] suites): This method allows you to set the list of enabled cipher suites on the SSL/TLS socket. You can provide an array of cipher suite names to enable specific suites. The provided cipher suites should be supported by the SSL/TLS implementation. If any of the specified cipher suites are not supported, they will be ignored.

```
String[] cipherSuites = {
   "TLS_RSA_WITH_AES_128_CBC_SHA",
   "TLS_RSA_WITH_AES_256_CBC_SHA"
};
sslSocket.setEnabledCipherSuites(cipherSuites);
```