Assignment-3

Raghuvamsi Mallampalli -A04316093

Koushik Reddy Kambham -A04316083

Introduction

This study showcases the enhanced performance of SSL-encrypted socket communication compared to plain text socket communication. For this experiment, the client-server code was modified to utilize SSL Sockets for network message exchange. Wireshark, a network sniffer, assessed the implementation's effectiveness. The study details the development, compilation, security configurations, and analysis of encrypted packets using Wireshark.

1. Code Explanation

The provided code for this assignment covers the two essential components of SSL/TLS-secured communication: the server and the client.

Server-side Code: The server-side code begins by loading the server's keystore, which contains the server's private key and certificate. This keystore is then initialized within an SSLContext, which is used to create an SSLServerSocket to listen for client connections. Once a connection is established, the server receives a message from the client, prints it, and sends a response back to the client.

Client-side Code: Similarly, the client-side code starts by loading the client's truststore, which contains the certificates of trusted servers. This truststore is used to initialize an SSLContext, which is then used to create an SSLSocket. Through this socket, the client communicates with the server, reads the server's response, and prints it.

Both programs demonstrate how to configure and utilize SSLContexts and SSL sockets, establishing the groundwork for secure, encrypted communication between a client and a server.

SecureServer.java

```
1 import javax.net.ssl.*;
2 import java.io.*;
3 import java.security.KeyStore;
4
5 public class SecureServer {
6     private static final int SERVER_PORT = 8800;  // Server listening port
7     private static final String KEYSTORE_PATH = "/home/seed/Downloads/assign3/keystore.jks";  // Path to the keystore file
8     private static final String KEYSTORE_PASSWORD = "koushik";  // Keystore password
9
10    public static void main(String[] args) {
11        try {
12            // Load the keystore
13            KeyStore serverKeystore = KeyStore.getInstance("JKS");
14            serverKeystore.load(new FileInputStream(KEYSTORE_PATH), KEYSTORE_PASSWORD.toCharArray());
15
16            // Initialize SSL context
17            SSLContext sslContext = SSLContext.getInstance("TLS");
18            KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance("SunX509");
19            keyManagerFactory.init(serverKeystore, KEYSTORE_PASSWORD.toCharArray());
20            sslContext.init(keyManagerFactory.getKeyManagers(), null, null);
21
22            // Create SSL server socket factory
23            SSLServerSocketFactory sslServerSocketFactory = sslContext.getServerSocketFactory();
24
25            // Create SSL server socket
26            SSLServerSocket sslServerSocket = (SSLServerSocket) sslServerSocketFactory.createServerSocket(SERVER_PORT);
27
28            System.out.println("Server started. Listening on port " + SERVER_PORT + "...");
29
30            // Wait for a client connection
31            SSLSocket serverSSLSocket = (SSLSocket) sslServerSocket.accept();
```

```
32
33            // Create reader and writer for the socket
34            BufferedReader reader = new BufferedReader(new InputStreamReader(serverSSLSocket.getInputStream()));
35            PrintWriter writer = new PrintWriter(serverSSLSocket.getOutputStream(), true);
36
37            // Read and print the client's message
38            String clientMessage = reader.readLine();
39            System.out.println("Received message from client: " + clientMessage);
40
41            // Send response back to the client
42            writer.println("Server response: Message received! " + clientMessage);
43
44            // Close the connections
45            reader.close();
46            writer.close();
47            serverSSLSocket.close();
48            sslServerSocket.close();
49        } catch (Exception e) {
50            e.printStackTrace();
51        }
52    }
```

Defining Constants: Values are specified for SERVER_PORT, KEYSTORE_PATH, and KEYSTORE_PASSWORD, which respectively represent the server's listening port, the path to the server's keystore, and the keystore password.

- SERVER_PORT: 8800
- KEYSTORE_PATH: "/home/seed/Downloads/assign3/keystore.jks"
- KEYSTORE_PASSWORD: "koushik"

Loading the Server's Keystore: A KeyStore instance is populated with the contents of the server's keystore file. This keystore contains the server's private key and certificate.

- The keystore is loaded from the specified path using KeyStore.getInstance("JKS") and load(new FileInputStream(KEYSTORE_PATH), KEYSTORE_PASSWORD.toCharArray()).

Setting up SSL Context: An SSLContext instance is created to act as a factory for SSLSocket and SSLSocketFactory. The server's keystore and password are used to configure the KeyManager, which is the initial component of the SSLContext.

- The KeyManagerFactory is initialized with the server's keystore and password using keyManagerFactory.init(serverKeystore, KEYSTORE_PASSWORD.toCharArray()).
- The SSLContext is initialized using sslContext.init(keyManagerFactory.getKeyManagers(), null, null).

Creating an SSLServerSocket: An SSLServerSocket that listens on the specified port is created using the SSLServerSocketFactory.

- The SSLServerSocketFactory is obtained from the SSLContext using sslContext.getServerSocketFactory().
- The SSLServerSocket is created using sslServerSocketFactory.createServerSocket(SERVER_PORT).

Waiting for a Connection: The server waits for a client to connect. When a client connects, an SSLSocket is created for the connection.

- The server listens for connections using sslServerSocket.accept() which returns an SSLSocket when a client connects.

Setting up Communication: Input and output streams are configured on the SSLSocket, enabling the server to send and receive data from the client.

- The input stream is created using new BufferedReader(new InputStreamReader(serverSSLSocket.getInputStream())).
- The output stream is created using new PrintWriter(serverSSLSocket.getOutputStream(), true).

Receiving a Message: The server reads a line from the client and prints it.

- The message from the client is read using reader.readLine() and printed using System.out.println("Received message from client: " + clientMessage).

Sending a Response: The server sends a reply to the client.

- The response is sent using writer.println("Server response: Message received! " + clientMessage).

Closing Connections: All streams and sockets are closed after communication ends.

- The reader, writer, and sockets are closed using reader.close(), writer.close(), serverSSLSocket.close(), and sslServerSocket.close().

SecureClient.java

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

public class SecureClient {
    private static final String SERVER_HOST = "localhost";  // Server host
    private static final int SERVER_PORT = 8800;  // Server port
    private static final String TRUSTSTORE_PATH = "/home/seed/Downloads/assign3/truststore.jks";  // Path to the truststore file
    private static final String TRUSTSTORE_PASSWORD = "koushik";  // Truststore password

    public static void main(String[] args) {
        try {
            // Load the truststore
            KeyStore clientTruststore = KeyStore.getInstance("JKS");
            clientTruststore.load(new FileInputStream(TRUSTSTORE_PATH), TRUSTSTORE_PASSWORD.toCharArray());

            // Initialize SSL context
            SSLContext sslContext = SSLContext.getInstance("TLS");
            TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance("SunX509");
            trustManagerFactory.init(clientTruststore);
            sslContext.init(null, trustManagerFactory.getTrustManagers(), null);

            // Create SSL socket factory
            SSLSocketFactory sslSocketFactory = sslContext.getSocketFactory();

            // Create SSL socket
            SSLSocket sslSocket = (SSLSocket) sslSocketFactory.createSocket(SERVER_HOST, SERVER_PORT);

            // Create reader and writer for the socket
            BufferedReader reader = new BufferedReader(new InputStreamReader(sslSocket.getInputStream()));
```

```
            PrintWriter writer = new PrintWriter(sslSocket.getOutputStream(), true);

            // Get user message and send to the server
            String userMessage = JOptionPane.showInputDialog("Enter message to send to server");
            writer.println(userMessage);

            System.out.println("Sent message to server: " + userMessage);

            // Read and display server's response
            String serverResponse = reader.readLine();
            JOptionPane.showMessageDialog(null, "The server sent: " + serverResponse);

            System.out.println("Server response: " + serverResponse);

            // Close the connections
            reader.close();
            writer.close();
            sslSocket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Defining Constants: There are specified values for the variables SERVER_HOST, SERVER_PORT, TRUSTSTORE_PATH, and TRUSTSTORE_PASSWORD. These represent the server's host address, port, the path to the client's truststore, and the truststore's password.

- SERVER_HOST: "localhost"
- SERVER_PORT: 8800
- TRUSTSTORE_PATH: "/home/seed/Downloads/assign3/truststore.jks"
- TRUSTSTORE_PASSWORD: "koushik"

Loading the Client's Truststore: A KeyStore instance is loaded with the client's truststore, which contains the certificates of trusted servers.

- The truststore is loaded from the specified file using KeyStore.getInstance("JKS") and load(new FileInputStream(TRUSTSTORE_PATH), TRUSTSTORE_PASSWORD.toCharArray()).

Setting up SSL Context: An SSLContext instance is created for generating SSLSocket objects. A TrustManager is configured with the client's truststore and initialized in the SSLContext.

- The TrustManagerFactory is initialized with the client's truststore using trustManagerFactory.init(clientTruststore).
- The SSLContext is set up using sslContext.init(null, trustManagerFactory.getTrustManagers(), null).

Creating an SSLSocketFactory: An SSLSocketFactory is created using the SSLContext, which will generate SSLSocket instances.

- The SSLSocketFactory is obtained from the SSLContext using sslContext.getSocketFactory().

Creating an SSLSocket: The SSLSocketFactory is used to create an SSLSocket that connects to the server at the specified host address and port.

- The SSLSocket is created using sslSocketFactory.createSocket(SERVER_HOST, SERVER_PORT).

Setting up Communication: Input and output streams are configured on the SSLSocket, allowing the client to send and receive data to and from the server.

- The input stream is created using new BufferedReader(new InputStreamReader(sslSocket.getInputStream())).
- The output stream is created using new PrintWriter(sslSocket.getOutputStream(), true).

Sending a Message: The client sends a message entered by the user in a dialog box to the server.

- The user message is obtained using JOptionPane.showInputDialog("Enter message to send to server").
- The message is sent using writer.println(userMessage).
- The sent message is logged using System.out.println("Sent message to server: " + userMessage).

Receiving a Response: The client reads the server's response, which is displayed in a dialog box and logged.

- The server's response is read using reader.readLine().
- The response is displayed using JOptionPane.showMessageDialog(null, "The server sent: " + serverResponse).
- The response is logged using System.out.println("Server response: " + serverResponse).

Closing Connections: All streams and the socket are closed after communication is complete.

- The reader, writer, and socket are closed using reader.close(), writer.close(), and sslSocket.close().

2. Steps to compile and run

Open the terminal in the folder, where all files are stored

Compile and run SecureServer.java to start the server.

```
[07/26/24]seed@VM:~/.../assign3$ javac SecureServer.java
[07/26/24]seed@VM:~/.../assign3$ java SecureServer
Server started. Listening on port 8800...
```

After that server begins and says “Server started Listening on port 8800”.

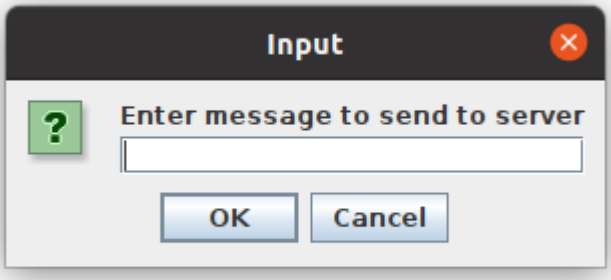
Now open a new terminal window.

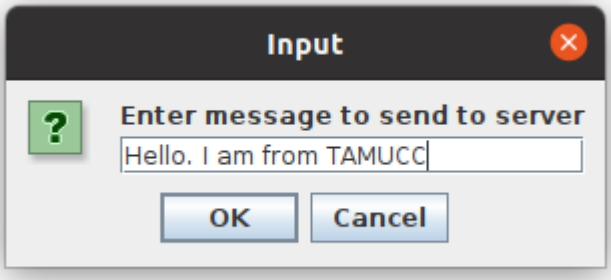
Compile and run SecureClient.java

```
[07/26/24]seed@VM:~/.../assign3$ javac SecureClient.java
[07/26/24]seed@VM:~/.../assign3$ java SecureClient
```

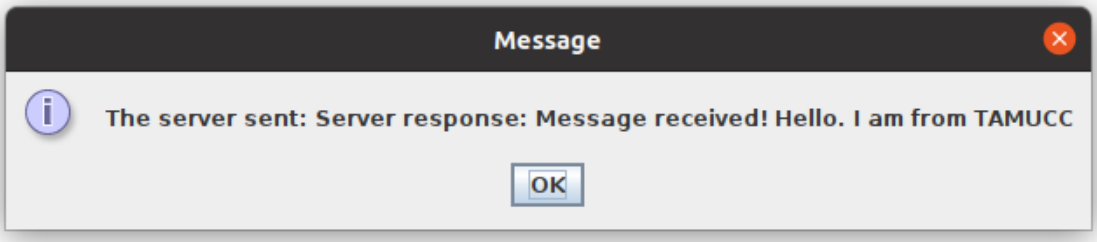
On the client side, prompt the user to input a message through a graphical user interface.

Transmit the entered message to the server via an SSL-secured connection.





Display the server's response message to the client user using a graphical user interface.



3. Security Configurations:

Step 1: Create the server's SSL certificate and keystore.

Open a command prompt and navigate to the directory where you want to create the keystore and certificate files. To generate a self-signed SSL certificate for the server and save it in a keystore, execute the following command:

bash

keytool -genkeypair -alias server -keyalg RSA -keysize 2048 -keystore keystore.jks -validity 365

You will be prompted to provide details such as your name, organization, and location. Enter all the required information. When prompted for a keystore password, make sure to remember it as it will be needed in your server code.

Step 2: Export the server's SSL certificate.

To export the server's SSL certificate to a separate file, use the following command:

bash

keytool -exportcert -alias server -keystore keystore.jks -file serverCertificate.cer

Step 3: Create the client's truststore.

To set up a truststore for the client, use this command:

bash

keytool -importcert -alias server -file serverCertificate.cer -keystore truststore.jks

You will see a confirmation prompt asking if you want to proceed with the import. Type "yes" and press Enter.

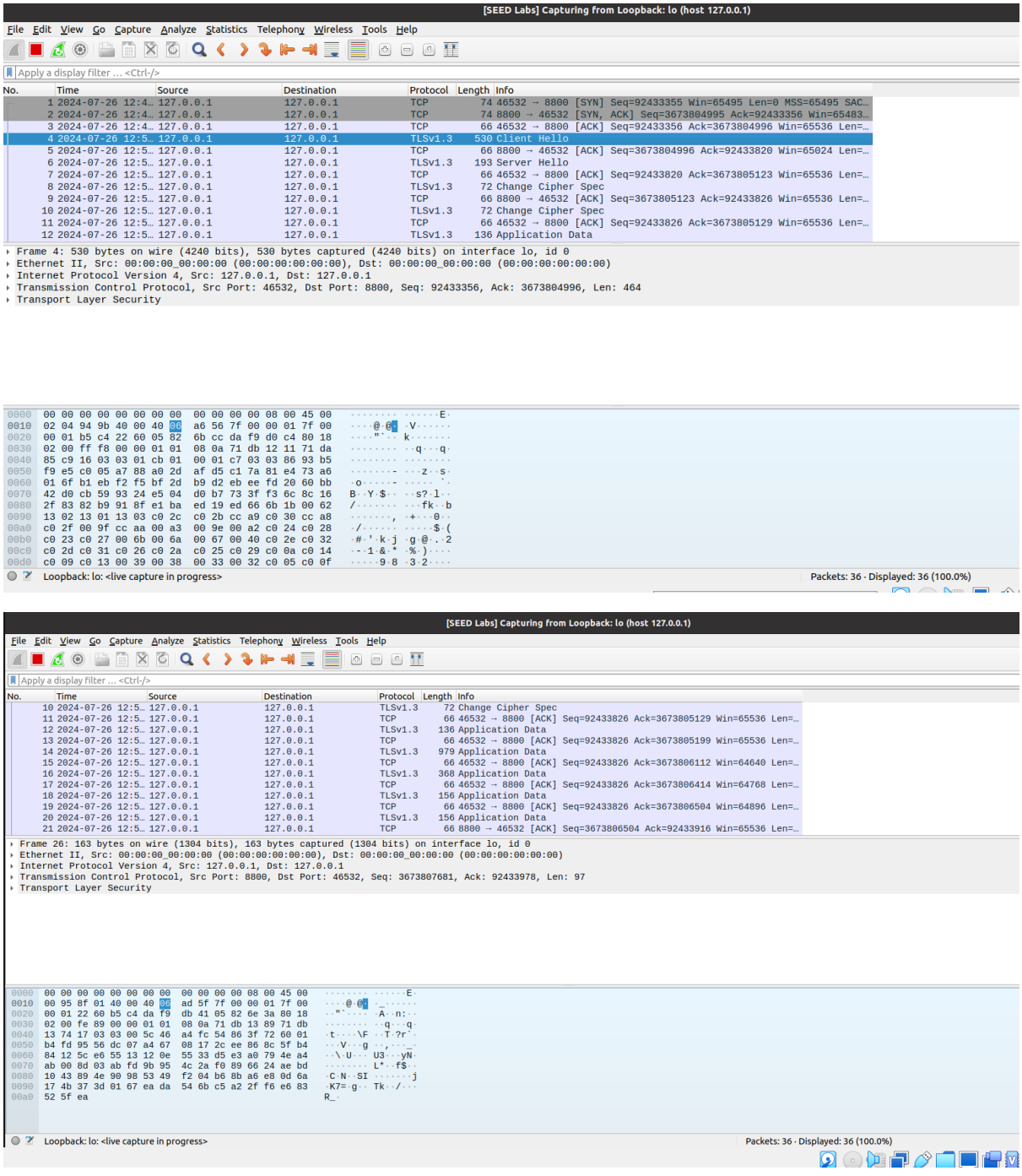
When prompted for a truststore password, remember to note it down as it will be necessary for the client code.

With these steps completed, you will have the client's truststore (`truststore.jks`), the server's keystore (`keystore.jks`), and the server's SSL certificate (`serverCertificate.cer`). Ensure these files are placed in the correct locations as specified in the server and client code.

4. Validation Using Wireshark:

Wireshark now allows us to view the packets in their encrypted state.

The screenshots from Wireshark below demonstrate encrypted transmission and demonstrate that the data is truly encrypted.



5. CONCLUSION:

This assignment successfully demonstrated how to set up and verify SSL encryption in socket communication to improve security. By modifying the Java code for both client and server, secure data transmission and reception over an SSL/TLS channel is achieved. The security of this communication was confirmed with the Wireshark network sniffer, which showed that the data packets were encrypted, highlighting the effectiveness of SSL/TLS in protecting data privacy during transmission. This research emphasizes the crucial role of encryption in network

security and lays a strong foundation for comprehending and applying SSL/TLS for secure socket communication.