CS6903: Network Security

Assignment 6: Secure chat using openssl and MITM attacks
Report and Readme

PLAGIARISM STATEMENT

I certify that this assignment/report is my own work, based on my personal study and/or research and that I have acknowledged all materials and sources used in its preparation, whether they be books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication. I also certify that this assignment/report has not previously been submitted for assessment in any other course, except where specific permission has been granted from all course instructors involved, or at any other time in this course, and that I have not copied in part or whole or otherwise plagiarised the work of other students and/or persons. I pledge to uphold the principles of honesty and responsibility at CSE@IITH. In addition, I understand my responsibility to report honor violations by other students if I become aware of them.

Name: Manisha Mahapatra, Reisha Ali
Date: 02/04/2023
Signature: CS22MTECH14009, CS23RESCH01001

Description:

In this programming assignment, we have implemented a secure peer-to-peer chat application using openssl in C/C++ and demonstrated how Alice and Bob could chat with each other using it. We have also implemented evil Trudy who is trying to intercept the chat messages between Alice and Bob by launching various MITM attacks.

Setup:
Accessing assigned VM : ssh ubuntu@10.200.13.113

Task – 1: Generate keys and certificates

OpenSSL commands used:

For RootCA

- Create rootCA key
  openssl ecparam -name secp521r1 -genkey -out rootCA.pem
- View rootCA.pem key
  cat rootCA.pem
- Create ExtCA file
  cat >> ExtCA.ext
  basicConstraints = CA:true
  keyUsage = digitalSignature, nonRepudiation, keyEncipherment, keyCertSign, cRLSign
- Create root CSR and gennerate v3 x509 self-signed certificate
  openssl req -new -key rootCA.pem -out rootCAcsr.pem
  openssl x509 -req -days 365 -in rootCAcsr.pem -signkey rootCA.pem -out rootCA.crt -extfile ExtCA.ext

For IntermediateCA

- Create intermediateCA key
  openssl genrsa -out interCA.pem 2048
- View interCA.pem key
  cat interCA.pem
- Generate public key
  openssl rsa -in interCA.pem -pubout -out interCA_p.pem
- View interCA_p.pem
  cat interCA_p.pem
- Create intermediate CSR
  openssl req -new -key interCA.pem -out interCAcsr.pem
- Create v3 x509 certificate signed by rootCA
  openssl x509 -req -in interCAcsr.pem -CA rootCA.crt -extfile ExtCA.ext -CAkey rootCA.pem -CAcreateserial -out interCA.crt -days 365 -sha256

For Alice

- Create Alice key
  openssl genrsa -out alice.pem 1024
- View alice.pem key
  cat alice.pem
- Generate Alice public key
  openssl rsa -in alice.pem -pubout -out alice_p.pem
- View alice_p.pem
  cat alice_p.pem
- Create Alice CSR
  openssl req -new -key alice.pem -out alicecsr.pem
- Generate digest and sign to generate hash
  openssl dgst -sha256 -sign alice.pem -out alice_sign.pem alicecsr.pem
  At RootCA:
- Verify that CSR request sent by Alice
  openssl dgst -sha256 -verify alice_p.pem -signature alice_sign.pem alicecsr.pem
- Create v3 x509 certificate signed by interCA
  openssl x509 -req -in alicecsr.pem -CA interCA.crt -extfile ExtCA.ext -CAkey interCA.pem -CAcreateserial -out alice.crt -days 365 -sha256
- Generate digest and sign to generate hash
  openssl dgst -sha256 -sign interCA.pem -out interCA_sign.pem alice.crt

At Alice end:

- Verify that certificate sent by Intermediate CA
  openssl dgst -sha256 -verify interCA_p.pem -signature interCA_sign.pem alice.crt
- Verify Alice certificate
  openssl verify -verbose -CAfile rootCA.crt alice.crt
  In .pem format
  openssl verify -verbose -CAfile rootCA.crt alicecert.pem

For Bob

- Create Bob key
  openssl ecparam -name prime256v1 -genkey -out bob.pem
- View bob.pem key
  cat bob.pem
- Generate Bob public key
  openssl rsa -in bob.pem -pubout -out bob_p.pem
- View bob_p.pem
  cat bob_p.pem
- Create Bob CSR
  openssl req -new -key bob.pem -out bobcsr.pem
- Generate digest and sign to generate hash
  openssl dgst -sha256 -sign bob.pem -out bob_sign.pem bobcsr.pem
  At RootCA:
- Verify that CSR request sent by Bob
  openssl dgst -sha256 -verify bob_p.pem -signature bob_sign.pem bobcsr.pem
- Create v3 x509 certificate signed by interCA
  openssl x509 -req -in bobcsr.pem -CA interCA.crt -extfile ExtCA.ext -CAkey interCA.pem -CAcreateserial -out bob.crt -days 365 -sha256
- Generate digest and sign to generate hash
  openssl dgst -sha256 -sign interCA.pem -out interCA_sign.pem bob.crt
  At Alice end:
- Verify that certificate sent by Intermediate CA
  openssl dgst -sha256 -verify interCA_p.pem -signature interCA_sign.pem bob.crt
- Verify Bob certificate
  openssl verify -verbose -CAfile rootCA.crt bob.crt
  In .pem format
  openssl verify -verbose -CAfile rootCA.crt bobcert.pem

Process followed for checking integrity and authenticity:

- Alice and Bob created their private key and CSR.
- Sending .csr files to interCA along with signed hash (digital signature) to prove that those cert req are indeed sent by them.
- Then interCA verifies the digital signature and issues the certificate to Alice/Bob.
- Along with the .crt file interCA will now send a signed digest of .crt to Alice/Bob to prove that the certificate was indeed issued by interCA.
- Since interCA's public key is available to Alice/Bob they can verify in the same manner.

Task – 2: Secure Chat App

Setup and message flow for secure_chat:
In secure_chat we have secure_chat_app.cpp which takes command line arguments.
For server and client we created two functions in secure_chat_app
Commands:
For compiling the code-
secure_chat/home/ubuntu/secure_chat_alice/secure_chat# g++ secure_chat_app.cpp -lssl -lcrypto -o secure_chat_app
For server:
./secure_chat_app -s <hostname> <portNo>
For client:
./secure_chat_app -c <hostname> <portNo>
For collecting traffic data and saving in .pcap file we use tcpdump-
tcpdump -i eth0 -w TASK2.pcap

At Server():

- Socket creation for the server & binding it to the server port then set it in listen mode (waiting for the client).
- After getting connected with client address, a chat session is initiated with ‘chat_request’ and ‘start_tls’.
- Server waits for client until it sends the chat_request message and once received it sends ‘chat_reply’ message
- The server waits for the client’s ‘start_tls’ message and once received, the server acknowledges it by sending ‘start_tls_ack’ message to the client.
- ‘start_comm_flag’ and ‘start_tls_flag’ flag variables are used to ensure that the TCP hello and TLS handshake occurs for only one time.
- Later the server loads its keys and certificate into the SSL Context and verifies the certificate received from the client into the context.
- Wraps the client socket with SSL Context and performs TLS handshake with the client thereby a secure TLS connection is established.
- Then for the further communication, the new TLS socket is used for auto encrypted and decrypted data.

At Client():

- Socket creation for the client.
- From serverhostname, the server ip address is resolved using gethostbyname() and connect() is used for connecting to server.
- Once connected, client sends the ‘chat_request’ to server and waits for the server’s reply
- After receiving ‘chat_reply’ from server, client sends ‘start_tls’ to the server.
- Once receiving start_tls_ack then loads its keys and certificate into SSL Context for making the secure TLS connection.
- Wraps the client socket with SSL Context and performs TLS handshake with the server.
- Now Secure TLS connection is established, client starts messaging the server.
- If the reply for start_tls is not an acknowledgement , then the client continues sending the messages in the TCP socket without TLS handshake
- When a client receives or sends ‘term’ then the TLS & TCP connections are closed and the program terminates.

Screenshots:

With Alice-

```
root@alice1:~/Alice# ./secure_chat_app -c bob1 8001

IP address of Client:
172.31.0.3

Client Socket Created


Enter message for Server:
chat_request

Received From Server: chat_reply

Enter message for Server:
start_tls

Received From Server: start_tls_ack

Client ctx created
C = IN, ST = OD, L = RKL, O = RSP, OU = NS, CN = Alice1.com
C = EU, ST = London, L = RoseBay, O = EY, OU = NS, CN = Bob1.com
Server Certificate Valid

Enter message for Server on TLS:
Hello Alice

Server received on tls: Hello Bob

Enter message for Server on TLS:
GoodBye.

Server received on tls: Ok Bye.

Enter message for Server on TLS:
term

Connection Terminated
```

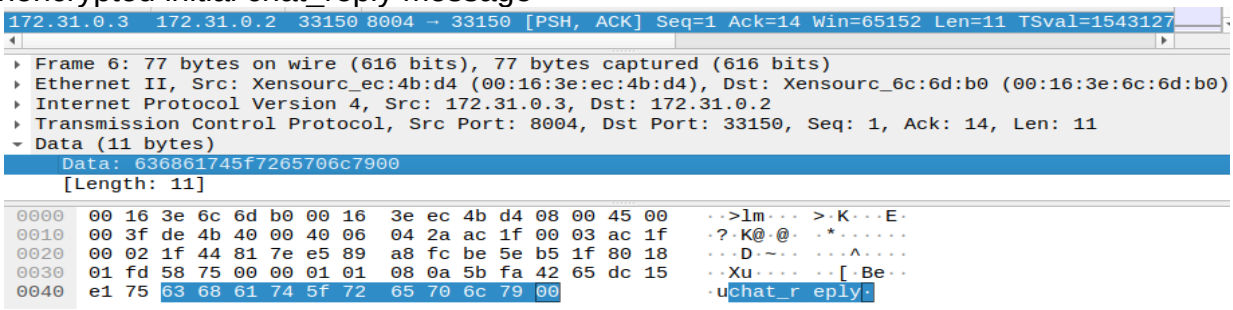
With Bob-

```
root@bob1:~/Bob# ./secure_chat_app -s 8001
......Waiting for client connection........

Client request accepted from IP: 172.31.0.2

Received From Client: chat_request

Enter message for Client:
chat_reply

Received From Client: start_tls

Enter message for Client:
start_tls_ack

Server ctx created
C = EU, ST = London, L = RoseBay, O = EY, OU = NS, CN = Bob1.com 4here in this
C = IN, ST = OD, L = RKL, O = RSP, OU = NS, CN = Alice1.com
Client Certificate Valid
Received from Client on TLS: Hello Alice

Enter message for Client on TLS:
Hello Bob
Received from Client on TLS: GoodBye.

Enter message for Client on TLS:
Ok Bye.

Connection terminated from Client
```

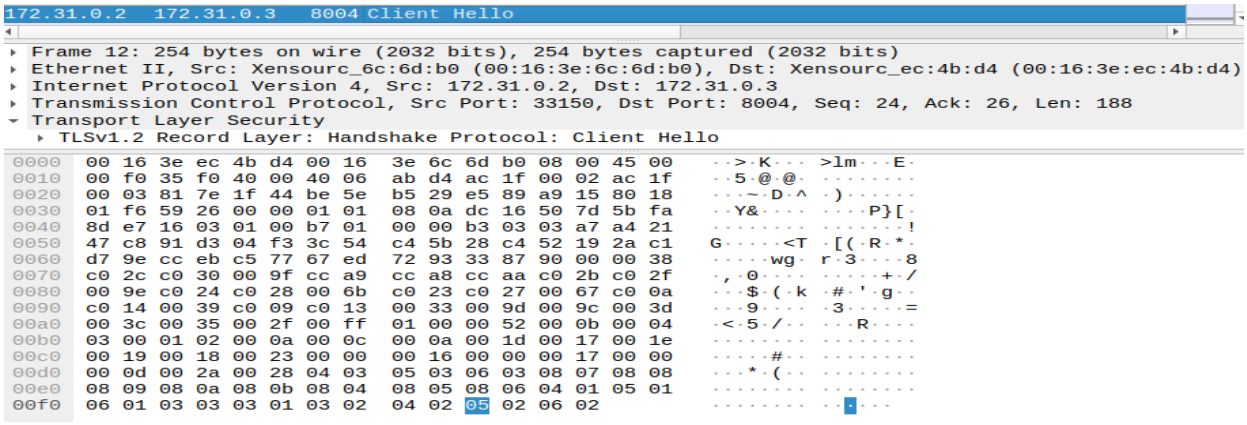
Unencrypted initial chat_request message-

```
2   172.31.0.3    8004 33150 → 8004 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=13 TSval=369242149
▸ Ethernet II, Src: Xensourc_6c:6d:b0 (00:16:3e:6c:6d:b0), Dst: Xensourc_ec:4b:d4 (00:16:
▸ Internet Protocol Version 4, Src: 172.31.0.2, Dst: 172.31.0.3
▾ Transmission Control Protocol, Src Port: 33150, Dst Port: 8004, Seq: 1, Ack: 1, Len: 13
    Source Port: 33150
    Destination Port: 8004
    [Stream index: 0]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 13]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 3193877778
    [Next Sequence Number: 14    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 3851004156
    1000 .... = Header Length: 32 bytes (8)
  ▸ Flags: 0x018 (PSH, ACK)
    Window: 502
    [Calculated window size: 64256]
    [Window size scaling factor: 128]
    Checksum: 0x5877 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  ▸ Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
  ▸ [Timestamps]
  ▸ [SEQ/ACK analysis]
    TCP payload (13 bytes)
▾ Data (13 bytes)
    Data: 636861745f7265517565737400
    [Length: 13]

0000  00 16 3e ec 4b d4 00 16  3e 6c 6d b0 08 00 45 00   ··>·K··· >lm···E·
0010  00 41 35 ec 40 00 40 06  ac 87 ac 1f 00 02 ac 1f   ·A5·@·@· ········
0020  00 03 81 7e 1f 44 be 5e  b5 12 e5 89 a8 fc 80 18   ···~·D·^ ········
0030  01 f6 58 77 00 00 01 01  08 0a dc 15 e1 75 5b f9   ··Xw···· ·····u[·
0040  ff 0f 63 68 61 74 5f 72  65 71 75 65 73 74 00      ··chat_r equest·
```

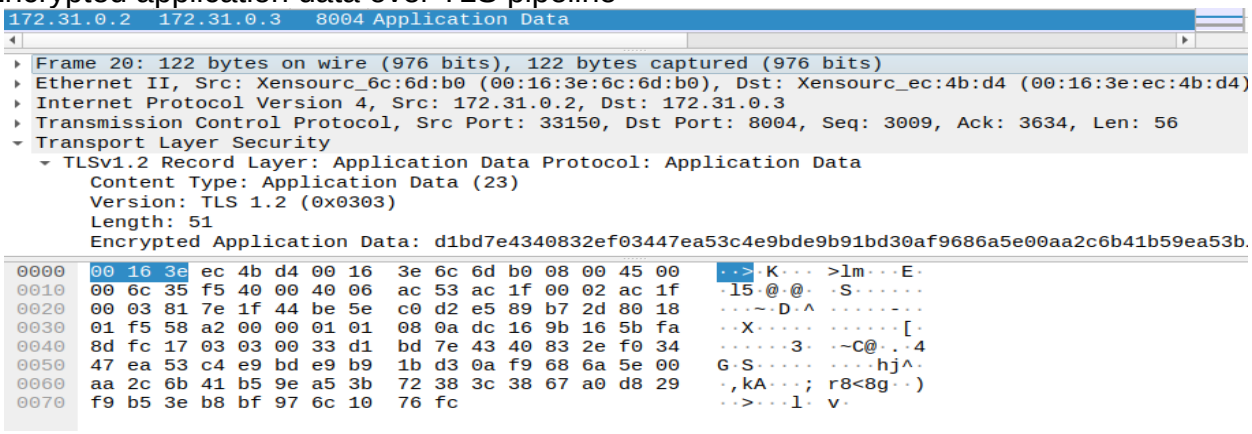
Unencrypted initial chat_reply message-



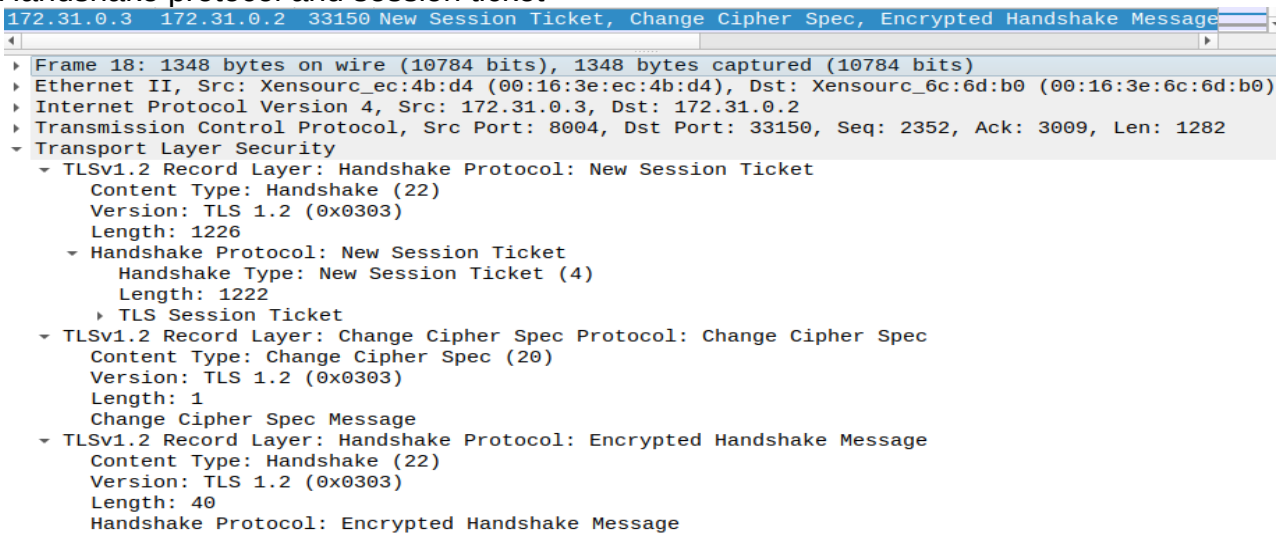
TLSv1.2 Handshake of Server and Client-



Encrypted application data over TLS pipeline-



Handshake protocol and session ticket-

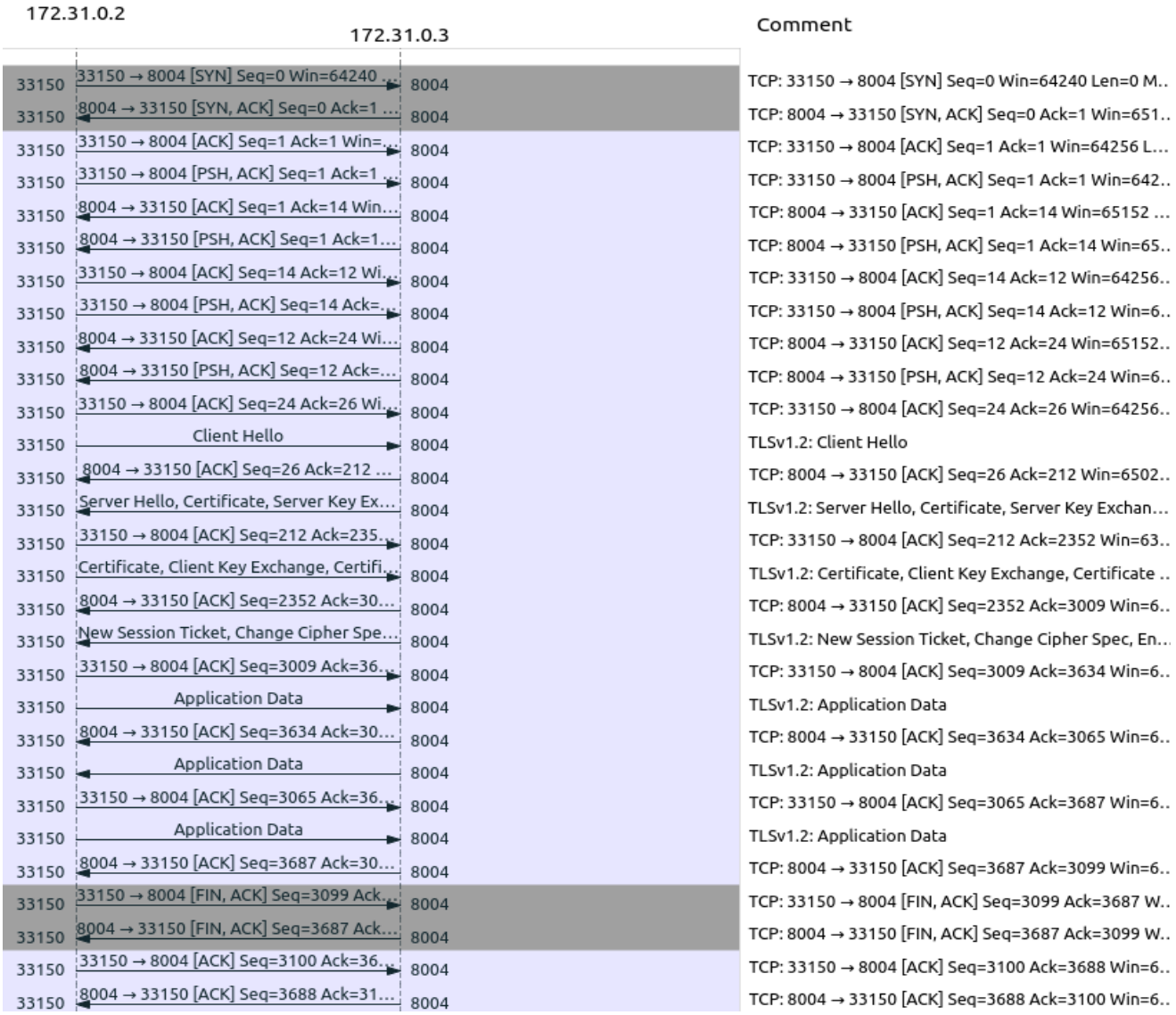


Supported ciphersuites as specified in the program:

```
//Initialize list of cipher suites
const char* str = "ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES256-GCM-SHA384";
```

For other ciphersuited, we not be able to perform verification and thus the connection will fail.

Flow Graph-



Task – 3: START_SSL downgrade attack #1 for eavesdropping

Setup and message flow for secure_chat_interceptor:
In secure_chat_interceptor we have secure_chat_ interceptor.cpp which takes command line arguments.
For server and client we created two functions in secure_chat_app as above.
In addition we have functions for performing the specific attacks as mentioned.
Commands:
To poison the etc/hosts file of Alice, Bob containers-
bash poison-dns-alice1-bob1.sh
For compiling the code (Client, Server)-
secure_chat/home/ubuntu/secure_chat_alice/secure_chat# g++ secure_chat_app.cpp -lssl -lcrypto -o secure_chat_app
For compiling the code (Trudy)-
secure_chat/home/ubuntu/secure_chat_alice/secure_chat# g++
secure_chat_interceptor .cpp -lssl -lcrypto -o secure_chat_ interceptor
For server:
./secure_chat_app -s <hostname> <portNo>
For client:
./secure_chat_app -c <hostname> <portNo>
For Trudy:
./secure_chat_ interceptor -d alice1 bob1 <portNo>
For collecting traffic data and saving in .pcap file we use tcpdump-
tcpdump -i eth0 -w TASK3.pcap

TLS protocol downgrade attack for eavesdropping:

- The function fake_connection(clienthostname, serverhostname) allows to create fake connections for the downgrade attack.
- In the fake_connection() function a fake server socket and a fake client socket are created which is then connected with Alice and Bob (Client and server).
- Once the ‘chat_hello’ is received from Client, Trudy forwards it to Server. Again when it receives ‘chat_reply’ from Server, it forwards to Client.
- When Trudy receives the ‘start_tls’ from Client, it blocks the message from reaching Server and sends fake ‘start_tls not supported’ message to Client.
- Then Client will not initiate TLS session and proceed with unsecure chat over soclet. And the connection between Trudy and server proceeds as usual with TCP.
- Hence downgrade attack by Trudy will be Successful.
- After receiving ‘term’ message from either Client or Server, the connection is terminated after closing the tcp and tls sockets properly.

Screenshots:
With Alice-

```
root@alice1:~/Alice# ./secure_chat_app -c bob1 8001

IP address of Client:
172.31.0.4

Client Socket Created


Enter message for Server:
chat_request

Received From Server: chat_reply

Enter message for Server:
start_tls

Received From Server: start_tls_not_supported

Enter message for Server on socket:
Hello Alice

Server received on socket: Hello Bob

Enter message for Server on socket:
GoodBye.

Server received on socket: Ok Bye.

Enter message for Server on socket:
term
Connection Terminated
```

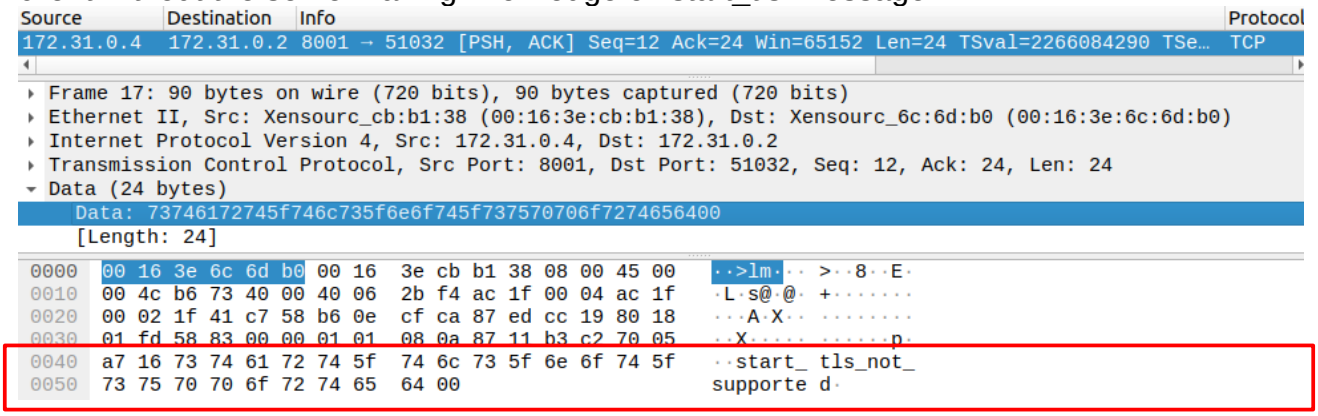
With Bob-

```
root@bob1:~/Bob# ./secure_chat_app -s 8001
......Waiting for client connection........

Client request accepted from IP: 172.31.0.4

Received From Client: chat_request

Enter message for Client:
chat_reply

Received From Client: Hello Alice
Enter message for Client:
Hello Bob
Received from Client on socket: GoodBye.

Enter message for Client on socket:
Ok Bye.

Connection terminated from Client
```

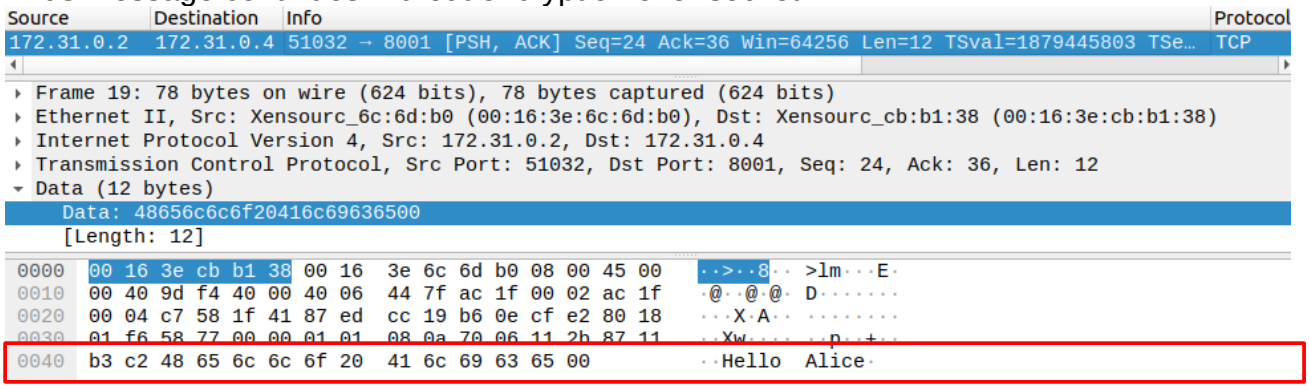
With Trudy-

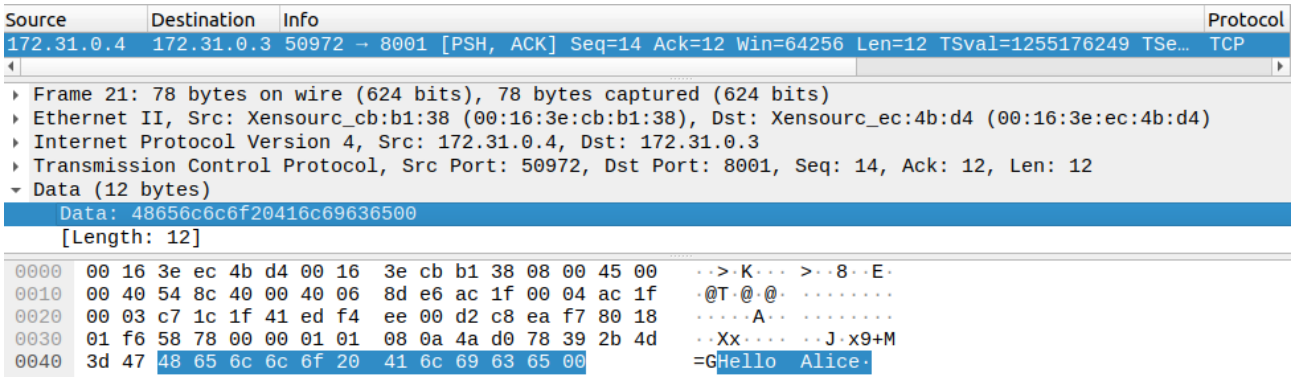
```
root@trudy1:~/Trudy# ./secure_chat_interseptor -d alice1 bob1 8001
......Waiting for client connection........


Client request accepted from ip: 172.31.0.2

Client socket created


Received message from Client on socket: chat_request

Received message from Server on socket: chat_reply

Received message from Client on socket: start_tls

Enter Fake TLS not supported message for Client: start_tls_not_supported

Received message from Client on socket: Hello Alice

Received message from Server on socket: Hello Bob

Received message from Client on socket: GoodBye.

Received message from Server on socket: Ok Bye.

Received message from Client on socket: term

Connection terminated from Client.
```

Trudy successfully performed downgrade attack by sending ‘start_tls_not_supported’ to client without the server having knowledge of ‘start_tls’ message-

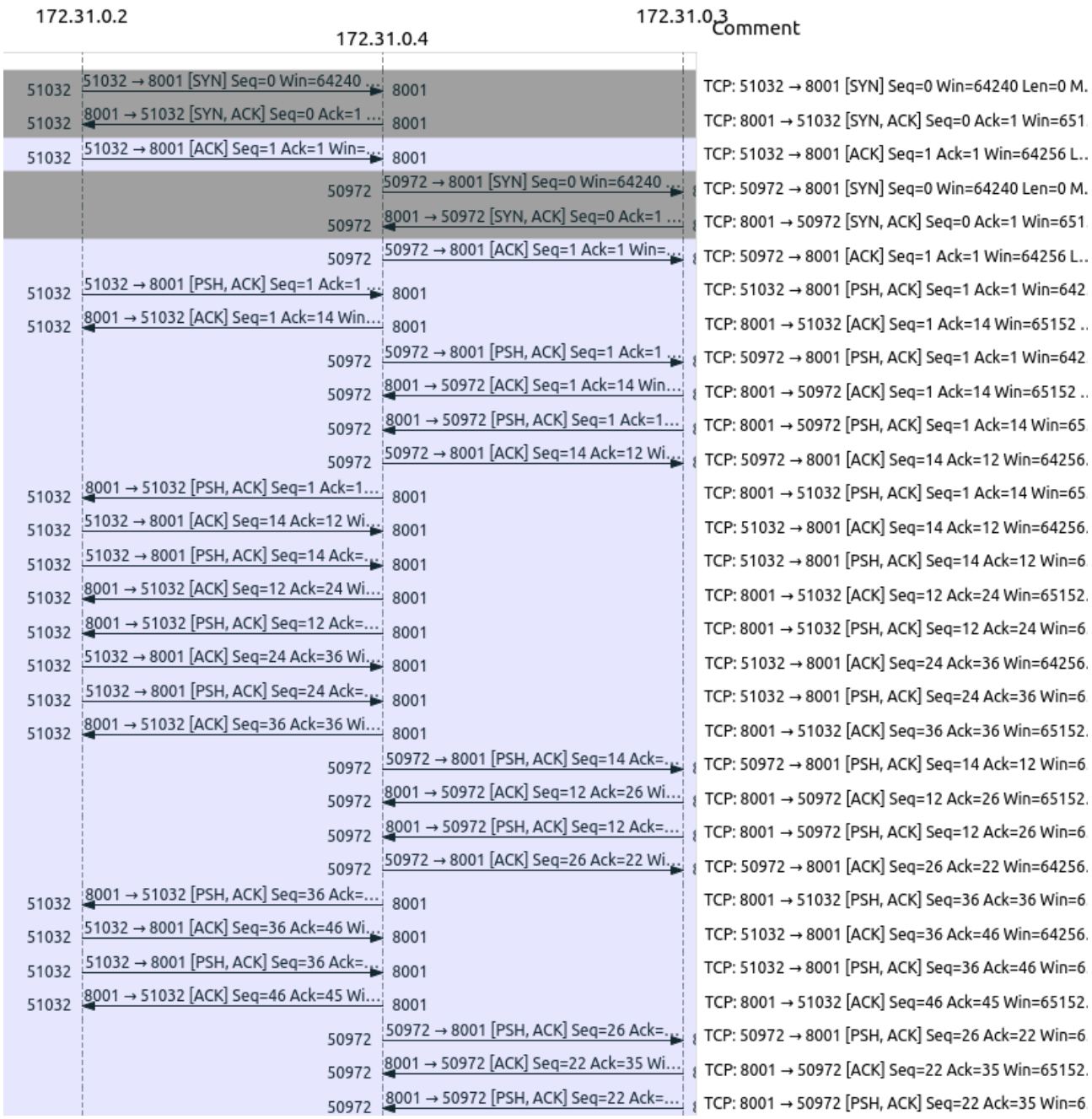


Thus message continues without encryption over socket-





Flow Graph-


172.31.0.2
172.31.0.4
172.31.0.3
Comment
51032 51032 → 8001 [SYN] Seq=0 Win=64240 ... 8001 TCP: 51032 → 8001 [SYN] Seq=0 Win=64240 Len=0 M.
51032 8001 → 51032 [SYN, ACK] Seq=0 Ack=1 ... 8001 TCP: 8001 → 51032 [SYN, ACK] Seq=0 Ack=1 Win=651
51032 51032 → 8001 [ACK] Seq=1 Ack=1 Win=... 8001 TCP: 51032 → 8001 [ACK] Seq=1 Ack=1 Win=64256 L...
50972 50972 → 8001 [SYN] Seq=0 Win=64240 ... TCP: 50972 → 8001 [SYN] Seq=0 Win=64240 Len=0 M.
50972 8001 → 50972 [SYN, ACK] Seq=0 Ack=1 ... TCP: 8001 → 50972 [SYN, ACK] Seq=0 Ack=1 Win=651
50972 50972 → 8001 [ACK] Seq=1 Ack=1 Win=... TCP: 50972 → 8001 [ACK] Seq=1 Ack=1 Win=64256 L...
51032 51032 → 8001 [PSH, ACK] Seq=1 Ack=1 ... 8001 TCP: 51032 → 8001 [PSH, ACK] Seq=1 Ack=1 Win=642
51032 8001 → 51032 [ACK] Seq=1 Ack=14 Win... 8001 TCP: 8001 → 51032 [ACK] Seq=1 Ack=14 Win=65152 ...
50972 50972 → 8001 [PSH, ACK] Seq=1 Ack=1 ... TCP: 50972 → 8001 [PSH, ACK] Seq=1 Ack=1 Win=642
50972 8001 → 50972 [ACK] Seq=1 Ack=14 Win... TCP: 8001 → 50972 [ACK] Seq=1 Ack=14 Win=65152 ...
50972 8001 → 50972 [PSH, ACK] Seq=1 Ack=1... TCP: 8001 → 50972 [PSH, ACK] Seq=1 Ack=14 Win=65
50972 50972 → 8001 [ACK] Seq=14 Ack=12 Wi... TCP: 50972 → 8001 [ACK] Seq=14 Ack=12 Win=64256.
51032 8001 → 51032 [PSH, ACK] Seq=1 Ack=1... 8001 TCP: 8001 → 51032 [PSH, ACK] Seq=1 Ack=14 Win=65
51032 51032 → 8001 [ACK] Seq=14 Ack=12 Wi... 8001 TCP: 51032 → 8001 [ACK] Seq=14 Ack=12 Win=64256.
51032 51032 → 8001 [PSH, ACK] Seq=14 Ack=... 8001 TCP: 51032 → 8001 [PSH, ACK] Seq=14 Ack=12 Win=6
51032 8001 → 51032 [ACK] Seq=12 Ack=24 Wi... 8001 TCP: 8001 → 51032 [ACK] Seq=12 Ack=24 Win=65152.
51032 8001 → 51032 [PSH, ACK] Seq=12 Ack=... 8001 TCP: 8001 → 51032 [PSH, ACK] Seq=12 Ack=24 Win=6
51032 51032 → 8001 [ACK] Seq=24 Ack=36 Wi... 8001 TCP: 51032 → 8001 [ACK] Seq=24 Ack=36 Win=64256.
51032 51032 → 8001 [PSH, ACK] Seq=24 Ack=... 8001 TCP: 51032 → 8001 [PSH, ACK] Seq=24 Ack=36 Win=6
51032 8001 → 51032 [ACK] Seq=36 Ack=36 Wi... 8001 TCP: 8001 → 51032 [ACK] Seq=36 Ack=36 Win=65152.
50972 50972 → 8001 [PSH, ACK] Seq=14 Ack=... TCP: 50972 → 8001 [PSH, ACK] Seq=14 Ack=12 Win=6
50972 8001 → 50972 [ACK] Seq=12 Ack=26 Wi... TCP: 8001 → 50972 [ACK] Seq=12 Ack=26 Win=65152.
50972 8001 → 50972 [PSH, ACK] Seq=12 Ack=... TCP: 8001 → 50972 [PSH, ACK] Seq=12 Ack=26 Win=6
50972 50972 → 8001 [ACK] Seq=26 Ack=22 Wi... TCP: 50972 → 8001 [ACK] Seq=26 Ack=22 Win=64256.
51032 8001 → 51032 [PSH, ACK] Seq=36 Ack=... 8001 TCP: 8001 → 51032 [PSH, ACK] Seq=36 Ack=36 Win=6
51032 51032 → 8001 [ACK] Seq=36 Ack=46 Wi... 8001 TCP: 51032 → 8001 [ACK] Seq=36 Ack=46 Win=64256.
51032 51032 → 8001 [PSH, ACK] Seq=36 Ack=... 8001 TCP: 51032 → 8001 [PSH, ACK] Seq=36 Ack=46 Win=6
51032 8001 → 51032 [ACK] Seq=46 Ack=45 Wi... 8001 TCP: 8001 → 51032 [ACK] Seq=46 Ack=45 Win=65152.
50972 50972 → 8001 [PSH, ACK] Seq=26 Ack=... TCP: 50972 → 8001 [PSH, ACK] Seq=26 Ack=22 Win=6
50972 8001 → 50972 [ACK] Seq=22 Ack=35 Wi... TCP: 8001 → 50972 [ACK] Seq=22 Ack=35 Win=65152.
50972 8001 → 50972 [PSH, ACK] Seq=22 Ack=... TCP: 8001 → 50972 [PSH, ACK] Seq=22 Ack=35 Win=6

Task – 4: START_SSL downgrade attack #2 for tampering

Setup and message flow for secure_chat_interceptor:
In secure_chat_interceptor we have secure_chat_ interceptor.cpp which takes command line arguments.
For server and client we created two functions in secure_chat_app as above.
In addition we have functions for performing the specific attacks as mentioned.
Commands:
For compiling the code (Client, Server)-
secure_chat/home/ubuntu/secure_chat_alice/secure_chat# g++ secure_chat_app.cpp -lssl -lcrypto -o secure_chat_app
For compiling the code (Trudy)-
secure_chat/home/ubuntu/secure_chat_alice/secure_chat# g++ secure_chat_interceptor .cpp -lssl -lcrypto -o secure_chat_ interceptor
For server:
./secure_chat_app -s <hostname> <portNo>
For client:
./secure_chat_app -c <hostname> <portNo>
For Trudy:
./secure_chat_ interceptor -m alice1 bob1 <portNo>
For collecting traffic data and saving in .pcap file we use tcpdump-
tcpdump -i eth0 -w TASK4.pcap

TLS protocol downgrade attack for tampering:

- The function fake_connection(clienthostname, serverhostname) allows to create fake connections for the downgrade attack.
- In the fake_connection() function a fake server socket and a fake client socket are created which is then connected with Alice and Bob (Client and server).
- Once the ‘chat_hello’ is received from Client, Trudy forwards it to Server. Again when it receives ‘chat_reply’ from Server, it forwards to Client.
- Once Trudy receives the ‘start_tls’ from Client for the first time(which is checked using a ‘start_tls_flag’ to avoid repeated ssl wrapping), it sends it to the Server and receives the reply, and sends it back to Client.
- If Server sends ‘start_tls_ack’ then the Trudy creates two TLS pipes (one for Server and Trudy and other for Client and Trudy) and does the handshake.
- Further chat continues using the secured socket of TLS which genrate encrypted application data.
- We have used ‘term’ as the termination message for all the programs.
- To depict the MITM functionality for tampering messages we have included the feature which allows Trudy to tamper the sent messages. This seriously affects meaninful communication and thus cause loss of information leading to miscommunication, misrepresentation and various losses.
- After receiving ‘term’ message from either Client or Server, the program is terminated tafter closing the TCP and TLS sockets properly.

Screenshots:
With Alice-

```
root@alice1:~/Alice# ./secure_chat_app -c bob1 8001

IP address of Client:
172.31.0.4

Client Socket Created


Enter message for Server:
chat_request

Received From Server: chat_reply

Enter message for Server:
start_tls

Received From Server: start_tls_ack

Client ctx created
C = IN, ST = OD, L = RKL, O = RSP, OU = NS, CN = Alice1.com
C = AK, ST = OT, L = BayBeach, O = GenP, OU = Dev, CN = Bob1.com
Server Certificate Valid

Enter message for Server on TLS:
Hello Alice. This is Bob. Transfer Rs200 to G.

Server received on tls: Hello Bob. Please confirm for Transfer Rs200 for G.

Enter message for Server on TLS:
Confirm.

Server received on tls: Done.

Enter message for Server on TLS:
term

Connection Terminated
```

With Bob-

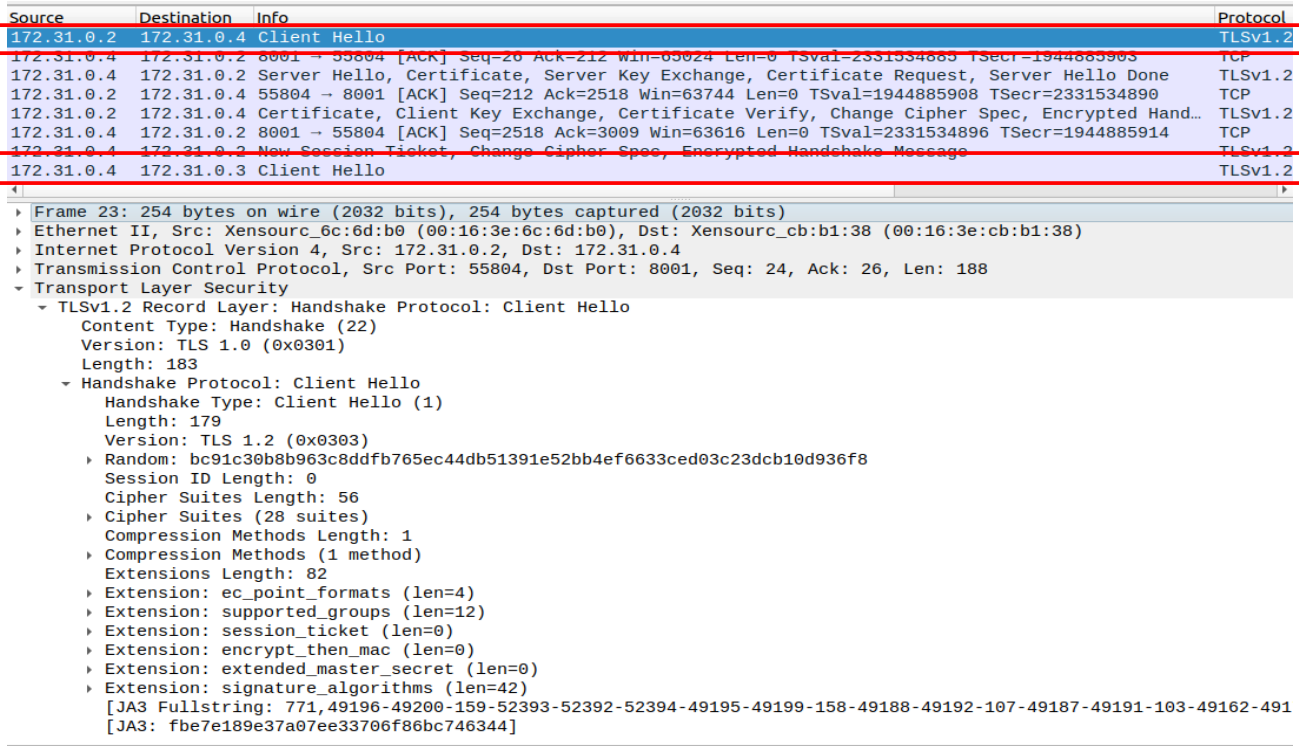
```
root@bob1:~/Bob# ./secure_chat_app -s 8001
......Waiting for client connection........

Client request accepted from IP: 172.31.0.4

Received From Client: chat_request

Enter message for Client:
chat_reply

Received From Client: start_tls

Enter message for Client:
start_tls_ack

Server ctx created
C = EU, ST = London, L = RoseBay, O = EY, OU = NS, CN = Bob1.com 4here in this
C = IN, ST = TE, L = HYD, O = IIT, OU = ACN, CN = Alice1.com
Client Certificate Valid
Received from Client on TLS: Hello Alice. This is Bob. Transfer Rs20000 to Trudy.

Enter message for Client on TLS:
Hello Bob. Please confirm for Transfer Rs20000 to Trudy.
Received from Client on TLS: Confirm.

Enter message for Client on TLS:
Done.

Connection terminated from Client
root@bob1:~/Bob#
```

With Trudy-

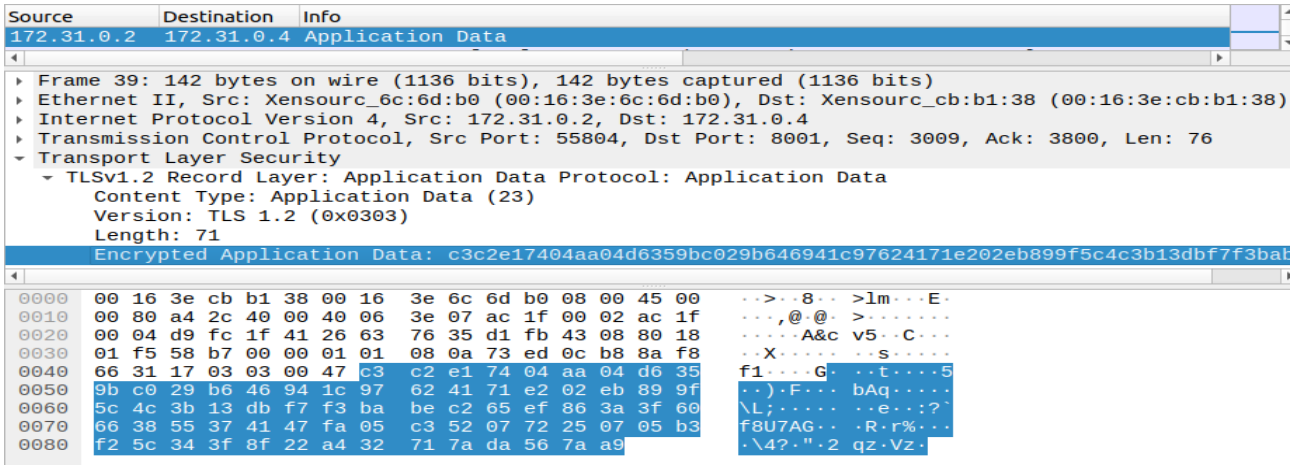
```
root@trudy1:~/Trudy# ./secure_chat_interceptor -m alice1 bob1 8001
......Waiting for client connection........


Client request accepted from ip: 172.31.0.2

Client socket created


From Alice on socket: chat_request

From Client on socket: chat_reply

From Alice on socket: start_tls

From Client on socket: start_tls_ack

Fake Server ctx created

Client Certificate Valid.

Fake Client ctx created

Server Certificate Valid.

From Client on TLS: Hello Alice. This is Bob. Transfer Rs200 to G.

Enter 1 to tamper:
1

Enter tampered data to send to Server:
Hello Alice. This is Bob. Transfer Rs20000 to Trudy.

From Server on TLS: Hello Bob. Please confirm for Transfer Rs20000 to Trudy.

 Enter 1 to tamper:
1

Enter tampered data to send to Client:
Hello Bob. Please confirm for Transfer Rs200 for G.
```

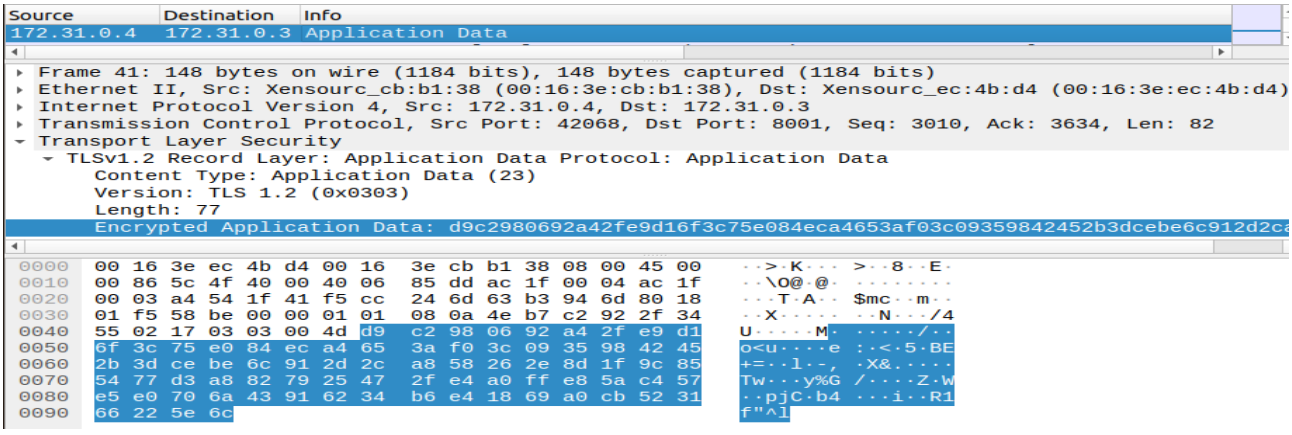
```
From Client on TLS: Confirm.

Enter 1 to tamper:
0

From Server on TLS: Done.

 Enter 1 to tamper:
0

From Client on TLS: term

Enter 1 to tamper:
0

Connection terminated.

Aborted (core dumped)
root@trudy1:~/Trudy#
```

As a result of message tampering by Trudy, messages were changed. This leads to loss of proper meaning and causes confusion, misscommunication between Client and Server.
Creation of two TLSv1.2 pipes (2 handshakes)-

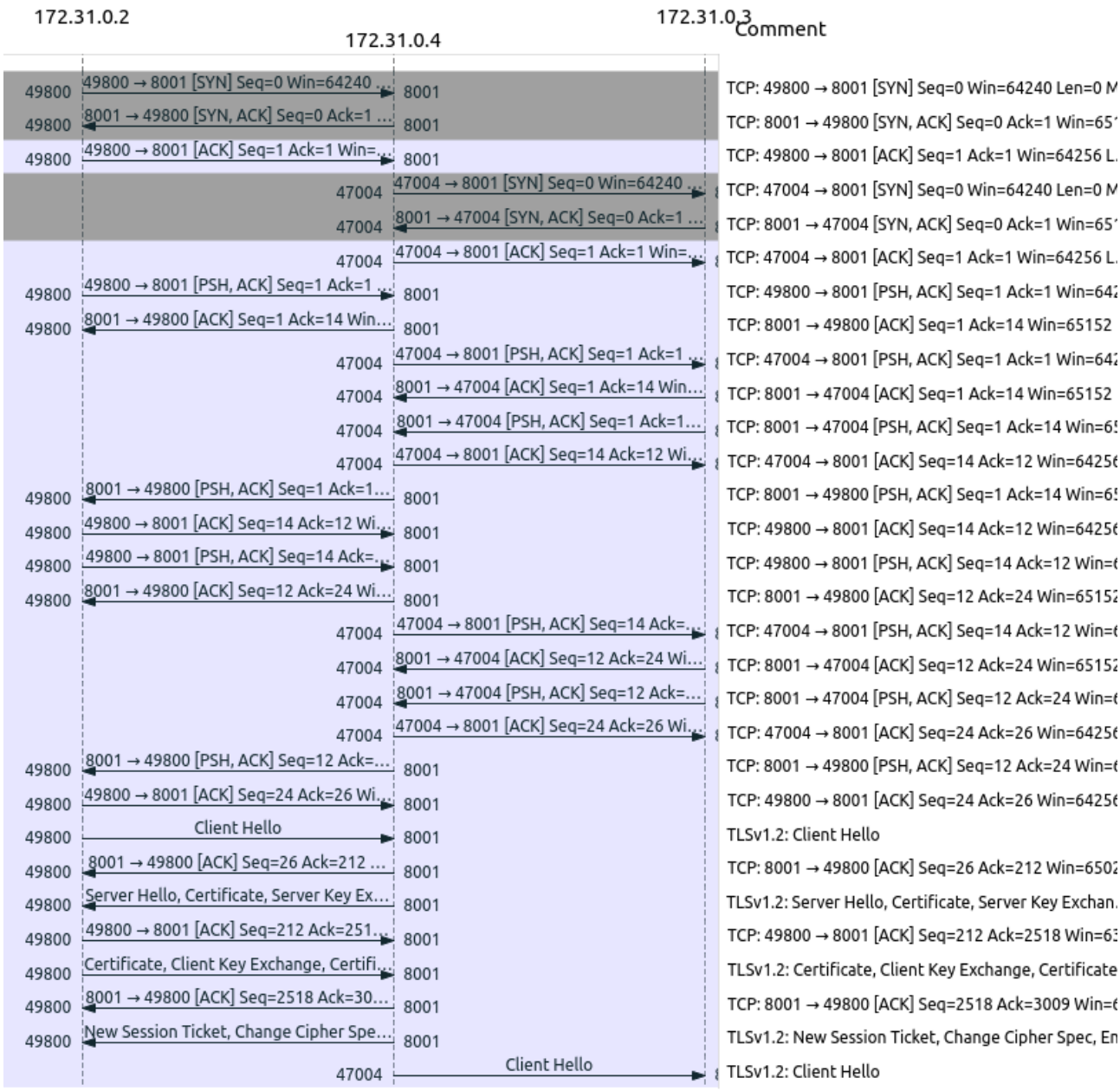


Encrypted messages between Trudy and Client/Server (Bob/Alice)-





Flow Graph-


172.31.0.2
172.31.0.4
172.31.0.3
Comment
49800 → 8001 [SYN] Seq=0 Win=64240 ...
8001 → 49800 [SYN, ACK] Seq=0 Ack=1 ...
49800 → 8001 [ACK] Seq=1 Ack=1 Win=...
47004 → 8001 [SYN] Seq=0 Win=64240 ...
8001 → 47004 [SYN, ACK] Seq=0 Ack=1 ...
47004 → 8001 [ACK] Seq=1 Ack=1 Win=...
49800 → 8001 [PSH, ACK] Seq=1 Ack=1 ...
8001 → 49800 [ACK] Seq=1 Ack=14 Win...
47004 → 8001 [PSH, ACK] Seq=1 Ack=1 ...
8001 → 47004 [ACK] Seq=1 Ack=14 Win...
8001 → 47004 [PSH, ACK] Seq=1 Ack=1...
47004 → 8001 [ACK] Seq=14 Ack=12 Wi...
8001 → 49800 [PSH, ACK] Seq=1 Ack=1...
49800 → 8001 [ACK] Seq=14 Ack=12 Wi...
49800 → 8001 [PSH, ACK] Seq=14 Ack=...
8001 → 49800 [ACK] Seq=12 Ack=24 Wi...
47004 → 8001 [PSH, ACK] Seq=14 Ack=...
8001 → 47004 [ACK] Seq=12 Ack=24 Wi...
8001 → 47004 [PSH, ACK] Seq=12 Ack=...
47004 → 8001 [ACK] Seq=24 Ack=26 Wi...
8001 → 49800 [PSH, ACK] Seq=12 Ack=...
49800 → 8001 [ACK] Seq=24 Ack=26 Wi...
Client Hello
8001 → 49800 [ACK] Seq=26 Ack=212 ...
Server Hello, Certificate, Server Key Ex...
49800 → 8001 [ACK] Seq=212 Ack=251...
Certificate, Client Key Exchange, Certifi...
8001 → 49800 [ACK] Seq=2518 Ack=30...
New Session Ticket, Change Cipher Spe...
Client Hello
TCP: 49800 → 8001 [SYN] Seq=0 Win=64240 Len=0 M
TCP: 8001 → 49800 [SYN, ACK] Seq=0 Ack=1 Win=65
TCP: 49800 → 8001 [ACK] Seq=1 Ack=1 Win=64256 L
TCP: 47004 → 8001 [SYN] Seq=0 Win=64240 Len=0 M
TCP: 8001 → 47004 [SYN, ACK] Seq=0 Ack=1 Win=65
TCP: 47004 → 8001 [ACK] Seq=1 Ack=1 Win=64256 L
TCP: 49800 → 8001 [PSH, ACK] Seq=1 Ack=1 Win=642
TCP: 8001 → 49800 [ACK] Seq=1 Ack=14 Win=65152
TCP: 47004 → 8001 [PSH, ACK] Seq=1 Ack=1 Win=642
TCP: 8001 → 47004 [ACK] Seq=1 Ack=14 Win=65152
TCP: 8001 → 47004 [PSH, ACK] Seq=1 Ack=14 Win=65
TCP: 47004 → 8001 [ACK] Seq=14 Ack=12 Win=64256
TCP: 8001 → 49800 [PSH, ACK] Seq=1 Ack=14 Win=65
TCP: 49800 → 8001 [ACK] Seq=14 Ack=12 Win=64256
TCP: 49800 → 8001 [PSH, ACK] Seq=14 Ack=12 Win=6
TCP: 8001 → 49800 [ACK] Seq=12 Ack=24 Win=65152
TCP: 47004 → 8001 [PSH, ACK] Seq=14 Ack=12 Win=6
TCP: 8001 → 47004 [ACK] Seq=12 Ack=24 Win=65152
TCP: 8001 → 47004 [PSH, ACK] Seq=12 Ack=24 Win=6
TCP: 47004 → 8001 [ACK] Seq=24 Ack=26 Win=64256
TCP: 8001 → 49800 [PSH, ACK] Seq=12 Ack=24 Win=6
TCP: 49800 → 8001 [ACK] Seq=24 Ack=26 Win=64256
TLSv1.2: Client Hello
TCP: 8001 → 49800 [ACK] Seq=26 Ack=212 Win=6502
TLSv1.2: Server Hello, Certificate, Server Key Exchan
TCP: 49800 → 8001 [ACK] Seq=212 Ack=2518 Win=63
TLSv1.2: Certificate, Client Key Exchange, Certificate
TCP: 8001 → 49800 [ACK] Seq=2518 Ack=3009 Win=6
TLSv1.2: New Session Ticket, Change Cipher Spec, En
TLSv1.2: Client Hello

Files submitted

- CS22MTECH14009|CS23RESCH01001.tgz
  - Alice
    - alice.crt
    - alice.pem
    - alicecert.pem
    - alicecsr.pem
  - Bob
    - bob.crt
    - bob.pem
    - bobcert.pem
    - bobcsr.pem
  - Trudy
    - fake_alice
      - fake_alice.crt
      - fake_alicecsr.pem
    - fake_bob
      - fake_bob.crt
      - fake_bobcsr.pem
    - fake_alice.pem
    - fake_alicecert.pem
    - fake_bob.pem
    - fake_bobcert.pem
  - Root
    - Intermediate
      - interCA.crt
      - interCA.pem
      - interCAcsr.pem
    - Root
      - root.crt
      - rootCA.pem
      - rootCAcsr.pem
    - ExtCA.ext
    - rootCA.crt
  - pcap files
    - TASK2.pcap
    - TASK3.pcap
    - TASK4.pcap
  - secure_chat_app.cpp
  - secure_chat_interceptor.cpp
  - Report.pdf
  - Makefile

Steps to run (README):

a) Download and extract CS22MTECH14009|CS23RESCH01001.tgz
b) Open Terminal in present folder with MakeFile.
c) Below command to transfer the files/floders to the vm
   For folders
   scp -r <foldername> ubuntu@10.200.13.113:
   For files
   scp <filename> ubuntu@10.200.13.113:

```
manisha@manisha-HP-Laptop-15-bs1xx:~/Desktop$ scp -r CS22MTECH14009|CS23RESCH01001 ubuntu@10.200.13.113:
```

d) Connect to assigned vm with below command
   ssh ubuntu@10.200.13.113
e) Once logged in we move to the directory CS22MTECH14009| CS23RESCH01001
f) Here we will push the files including certificate, keys and .cpp codes to the respective containers with Makefile lxc commands
g) In each container the rootCA.crt needs to be transfered to the trust store.
   cp rootCA.crt /usr/local/share/ca-certificates/

Task2:

a) Start the server by running secure_chat_app in server mode
   ./secure_chat_app -s 8001

```
root@bob1:~/Bob# ./secure_chat_app -s 8001
......Waiting for client connection........
```

b) Start the client by running secure_chat_a in client mode
   ./secure_chat_app -c bob1 8001

```
root@alice1:~/Alice# ./secure_chat_app -c bob1 8001
```

Task3:

a) First we poison the DNS with below command
   bash poison-dns-alice1-bob1.sh

```
ubuntu@ns00-gold:~$ bash poison-dns-alice1-bob1.sh
```

b) Start the server by running secure_chat_app in server mode
   ./secure_chat_app -s 8001

```
root@bob1:~/Bob# ./secure_chat_app -s 8001
......Waiting for client connection........
```

c) Start the interceptor by running secure_chat_interceptor in downgrade attack mode
   ./secure_chat_ interceptor -d alice1 bob1 8001

```
root@trudy1:~/Trudy# ./secure_chat_interceptor -m alice1 bob1 8001
......Waiting for client connection........
```

d) Start the client by running secure_chat_a in client mode
   ./secure_chat_app -c bob1 8001

```
root@alice1:~/Alice# ./secure_chat_app -c bob1 8001
```

e) After execution is completed, we unpoison the DNS
bash unpoison-dns-alice1-bob1.sh

```
ubuntu@ns00-gold:~$ bash unpoison-dns-alice1-bob1.sh
```

Task4:

a) First we poison the DNS with below command
bash poison-dns-alice1-bob1.sh

```
ubuntu@ns00-gold:~$ bash poison-dns-alice1-bob1.sh
```

b) Start the server by running secure_chat_app in server mode
./secure_chat_app -s 8001

```
root@bob1:~/Bob# ./secure_chat_app -s 8001
......Waiting for client connection........
```

c) Start the interceptor by running secure_chat_interceptor in downgrade attack mode
./secure_chat_ interceptor -m alice1 bob1 8001

```
root@trudy1:~/Trudy# ./secure_chat_interceptor -m alice1 bob1 8001
......Waiting for client connection........
```

d) Start the client by running secure_chat_a in client mode
./secure_chat_app -c bob1 8001

```
root@alice1:~/Alice# ./secure_chat_app -c bob1 8001
```

e) After execution is completed, we unpoison the DNS
bash unpoison-dns-alice1-bob1.sh

```
ubuntu@ns00-gold:~$ bash unpoison-dns-alice1-bob1.sh
```

Important code snippets:
Task2-

```
//-------------------------------------------------------------------------
//Funtion for certificate setup
//-------------------------------------------------------------------------
void configureCertificates(SSL_CTX* ctx, char* CertFile, char* KeyFile)
{
    if ( SSL_CTX_use_certificate_file(ctx, CertFile, SSL_FILETYPE_PEM) <= 0 )
    {
        cout<<"\nCertificate file not valid"<<endl;
        ERR_print_errors_fp(stderr);
        abort();
    }
    if ( SSL_CTX_use_PrivateKey_file(ctx, KeyFile, SSL_FILETYPE_PEM) <= 0 )
    {
        cout<<"\nKey file not valid"<<endl;
        ERR_print_errors_fp(stderr);
        abort();
    }
    if ( !SSL_CTX_check_private_key(ctx) )
    {
        cout<<"\nKey not match with certificate file"<<endl;
        abort();
    }
}
//-------------------------------------------------------------------------
```

The function takes 3 argument- an SSL_CTX pointer called ctx and two character pointers called CertFile and KeyFile.

- SSL_CTX* **ctx**: a pointer to an SSL_CTX object, which is a data structure that holds the configuration information for SSL/TLS connections.
- char* **CertFile**: a pointer to a string that represents the path to the PEM-encoded certificate file.
- char* **KeyFile**: a pointer to a string that represents the path to the PEM-encoded private key file.

The first conditional statement verifies if the certificate file at the location specified by **CertFile** is valid. If it is not valid, an error message is displayed and the program is aborted.

The second conditional statement verifies if the private key file at the location specified by **KeyFile** is valid. If it is not valid, an error message is displayed and the program is aborted.

The third conditional statement checks if the private key matches the certificate file. If it doesn't match, an error message is displayed and the program is aborted.

```
//-------------------------------------------------------------------------
//Function to verify certificate
//-------------------------------------------------------------------------
int verify_the_certificate(SSL *ssl)
{
    int result;
    X509 *cert = SSL_get_peer_certificate(ssl);
    if (cert == nullptr)
    {
        ERR_print_errors_fp(stderr);
        cout<<"\nCertificate Not Given by Peer"<<endl;
        abort();
    }
    int err = SSL_get_verify_result(ssl);
    if (err != X509_V_OK)
    {
        ERR_print_errors_fp(stderr);
        const char *err_string = X509_verify_cert_error_string(err);
        printf("\nCertificate Not Valid : %s\n", err_string);
        abort();
    }
    result = err;
    return result;
}
```

- Inside the function, ssl pointer retrieves the peer's certificate using the **SSL_get_peer_certificate** function, which returns an X509 certificate pointer. If the peer's certificate is not provided, the function displays an error message using **ERR_print_errors_fp** function, then aborts the program.
- Next, the function retrieves the verification result of the certificate using **SSL_get_verify_result** function. If the certificate is not valid, an error message is displayed using **ERR_print_errors_fp** function and the **X509_verify_cert_error_string** function is called to obtain a human-readable error string. Then, the program is aborted.
- Finally, the function sets the verification result to a variable called **result** and returns it.

```
//----------------------------------------
//Client function
//----------------------------------------
int client(const char *hostname, int port)
{
    int client_sd;
    X509* cert;
    X509* peer_cert;
    SSL_CTX *ctx;
    int start_tls_flag =0;
    int start_comm_flag =0;
    SSL *ssl;
    char send_buffer[MAX];
    char receive_buffer[MAX];
    struct hostent *host;
    struct sockaddr_in addr;
```

- The client function takes two arguments: a hostname (represented as a C-style string) and a port number. It returns an integer value indicating the success or failure of the communication process.
- The function first creates a TCP socket using the **socket**() system call and initializes a **sockaddr_in** structure with the server's IP address and port number. It then connects to the server using the **connect**() system call. If the connection fails, it prints an error message and aborts the program.
- After establishing the connection, the function prompts the user to enter a message to send to the server. It reads the user's input, copies it into a buffer, and sends it to the server using the **send**() system call. It then waits for a response from the server using the **recv**() system call and prints the received message to the console.
- If the received message is "**start_tls_ack**", the function initiates a TLS handshake with the server. It uses the OpenSSL library to create a TLS/SSL context object and configure it with the client's X.509 certificate and private key. It then sets up a new SSL connection using the SSL_new() function and attaches it to the existing TCP socket using the **SSL_set_fd**() function. Finally, it performs the TLS handshake using the **SSL_connect**() function.
- If the TLS handshake succeeds, the function enters a loop that allows the user to send messages to the server over the encrypted channel. It reads the user's input, sends it to the server using the **SSL_write**() function, and waits for a response using the **SSL_read**() function. If the received message is "term", the function terminates the loop and closes the connection.
- If the TLS handshake fails, the function enters a similar loop that sends messages to the server over the original unencrypted channel.

```c
int server(int port)
{
    X509 *cert;
    X509 *peer_cert;
    int start_tls_flag = 0;
    int start_comm_flag = 0;
    char send_buffer[MAX];
    char receive_buffer[MAX];
    SSL *ssl;
    int server_sd;
    SSL_CTX *ctx;
    int connection;
    struct sockaddr_in addr, client_addr;
    server_sd = socket(AF_INET, SOCK_STREAM, 0);
    bzero(&addr, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);
    addr.sin_addr.s_addr = INADDR_ANY;
```

- The function **server**() takes an integer port as input and returns an integer.
- The function initializes various variables and structures, including cert and **peer_cert** of type X509, **send_buffer** and **receive_buffer** of type char, **ssl** of type SSL, **server_sd** of type int, **ctx** of type SSL_CTX, and **addr** and **client_addr** of type struct sockaddr_in.
- The function creates a socket using the **socket**() function, sets up the addr structure, binds the socket to the specified port using **bind**(), and starts listening for incoming connections using **listen**().
- When a client connects to the server, the function accepts the connection using **accept**(), receives a message from the client using **recv**(), and checks the message for a specific string. If the string is "**term**", the function closes the connection and aborts. If the string is "**chat_request**", the function sets a flag to indicate that a chat session has started.
- The function prompts the user to enter a message to send to the client using **getline**(), copies the message into **send_buffer**, and sends the message to the client using **send**().
- If the client sends a message "**start_tls**", the function sets up a TLS connection with the client by initializing the SSL library using **SSL_library_init**(), creating a new SSL context using **InitCTX**(), configuring the context with server certificates using **configureCertificates**(), setting the context to verify the client's certificate using **SSL_CTX_set_verify**(), creating a new SSL object using **SSL_new**(), setting the SSL object to use the established connection using **SSL_set_fd**(), and performing the SSL handshake using **SSL_accept**().
- If the SSL handshake is successful, the function verifies the client's certificate using **verify_the_certificate**(), sets a flag to indicate that TLS has started, and enters a loop to receive messages from the client and print them to the console.
- If the client does not send a "**start_tls**" message, the function simply prompts the user to enter another message to send to the client and repeats the process.

Task3-

```
int mitm_attack_1(int connection, int client_sd)
{
    char sendMessageAlice[MAX];
    char receiveMessageAlice[MAX];
    char receiveMessageBob[MAX];
    char sendMessageBob[MAX];
    int tls_flag = 0;
    while(true)
    {
        int alice_msg = recv(connection , receiveMessageAlice , MAX , 0);
        if(alice_msg < 0)
        {
            cout << "Message not recieved from Client." << endl;
            return 0;
        }
```

- It takes two parameters, "**connection**" and "**client_sd**"(connection to the client that the attacker is impersonating), which are integer values representing sockets.
- The function begins by declaring four character arrays: "**sendMessageAlice**", "**receiveMessageAlice**", "**receiveMessageBob**", and "**sendMessageBob**". It also sets a variable "**tls_flag**" to 0.
- The function uses a while loop to continuously intercept and manipulate messages until the connection is terminated. The loop starts by receiving a message from the client using **recv**() function and storing it in **receiveMessageAlice**. If there is an error in receiving the message, the function returns 0.
- If the received message from the client starts with the string "**start_tls**", the attacker sends a fake message to the client indicating that TLS is not supported. The attacker sets the **tls_flag** to 1 to indicate that the client should not try to use TLS.
- The attacker then forwards the message from the client to the server by copying the content of **receiveMessageAlice** to **sendMessageBob**. If the received message from the client starts with "**term**", the attacker forwards the message to the server and terminates the connection by breaking out of the loop.
- The attacker then receives a message from the server using **recv**() function and stores it in **receiveMessageBob**. If there is an error in receiving the message, the loop is terminated.
- The attacker then forwards the message from the server to the client by copying the content of **receiveMessageBob** to **sendMessageAlice**. If the received message from the server starts with "**term**", the attacker forwards the message to the client and terminates the connection by breaking out of the loop.
- Once the loop is terminated, the connections to the client and server are closed using **close**() function.

Task4-

```
int mitm_attack_2(int connection, int client_sd)
{
    char sendMessageAlice[MAX];//client
    char receiveMessageAlice[MAX];
    char receiveMessageBob[MAX];//server
    char sendMessageBob[MAX];
    X509* cert;
    X509* peer_cert;
    SSL_CTX *ctx_client;
    SSL_CTX *ctx_server;
    SSL *ssl_client;
    SSL *ssl_server;
    int client_ver = 0;
    int server_ver = 0;
    int start_tls_flag = 0;
    int start_comm_flag = 1;
    int count =2;
    string s;
```

- The **mitm_attack_2** function takes two integer arguments **connection** and **client_sd** which are socket descriptors for the connection.
- The function sets up two SSL contexts, one for the fake server and the other for the fake client, and two SSL objects for the SSL connections with Client and Server. It initializes these SSL objects with fake certificates and establishes a TLS connection with both parties.
- The function then enters into a loop to intercept and modify messages between Client and Server.
- If it receives the message "**start_tls_ack**" from Server and if the start_tls_flag is not set, it initializes the fake TLS connections with Client and Server.
- The function then enters into another loop to receive messages over the fake TLS connection. Onc a message is received (from Client/Server), then it outputs the message to the console(Trudy) and prompts to enter "1" to tamper with the message. If the attacker enters "1", the function modifies the message (as per input by attacker) and sends the modified message to the Server/Client(other party).
- The function continues this loop until the connection is terminated by either party or until there is an error in the connection.