CS-302

Computer Networks Lab Report

Assignment - 5

BTech 5th Semester 2021

Department of Computer Science and Engineering,

National Institute of Technology Karnataka, Surathkal

October 2021

Submitted By:
Deepta Devkota - 191CS117
Akanksha More - 191CS106

Q1. Develop a code to illustrate a secure socket connection between client and server.

Secure Socket Layer(SSL) is a security protocol to encrypt the data being transmitted. It is used for establishing authenticated and encrypted links between networked devices. Thus it encrypts the established link hence ensuring that the data is transferred without any attack.

Working of SSL:

1. The client sends a request to the server.
2. The server sends a public key along with an SSL certificate which is digitally signed by a third party i.e. CA (Certificate Authority).
3. The client then verifies the digital signature using CA's public key.
4. It sends an encrypted symmetric key to the server which is encrypted using the server's public key.
5. The server then decrypts the encrypted symmetric key using its own private key.
6. This symmetric key is thereafter used for encryption and decryption of data being transferred.

Certificate Authority (CA):
It is a trusted organization that issues digital certificates. It validates the identities of entities such as websites, email addresses, companies, or individuals and binds them to the cryptographic keys thus issuing it as a digital certificate. Generated certificates provide authentication, encryption, and integrity.

SSL Certificate:
SSL certificate consists of the following details:

1. Domain Name
2. Name of person or organization, for which it was issued
3. Name of CA which issued the certificate
4. Digital signature of CA
5. Associated subdomains
6. Date of issue of certificate
7. The expiry date of the certificate
8. Public key

SSL Server-side implementation:
For the given question for illustrating secure socket connection between client and server, we have used **SOCKET and SSL library** in python.

SSL certificate generation:
We use the following commands for certificate generation.

1. CA

Generate Private key:

```
openssl genrsa -out ca_key.pem 2048
```

Generate Certificate:

```
openssl x509 -out ca_cert.pem -req -signkey ca_key.pem -days 365
```

2. Server

Generate Private key:

```
openssl genrsa -out server_key.pem 2048
```

Certificate Signing Request:

```
openssl req -new -key server_key.pem -out server.pem
```

Generate Certificate:

```
openssl x509 -req -in server.pem -CA ca_cert.pem -CAkey
ca_key.pem -CAcreateserial -out server_cert.pem -days 365
-sha256
```

3. Client

Generate Private key:

```
openssl genrsa -out client_key.pem 2048
```

Certificate Signing Request:

```
openssl req -new -key client_key.pem -out client.pem
```

Generate Certificate:

```
openssl x509 -req -in client.pem -CA ca_cert.pem -CAkey
ca_key.pem -CAcreateserial -out client_cert.pem -days 365
-sha256
```

Server-side Code:

```python
import socket
import ssl


server = socket.socket()

server.bind(("localhost",9999))
server.listen(5)

print("Server is listening....")

contex = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)

while True:
    client, address = server.accept()

    secure_socket = ssl.wrap_socket(client,
                                    server_side=True,
                                    ssl_version=ssl.PROTOCOL_TLS,
                                    ca_certs="certificate/ca_cert.pem",
                                    certfile="certificate/server_cert.pem",
                                    keyfile="certificate/server_key.pem",
                                    cert_reqs=ssl.CERT_REQUIRED)

    client_certificate = secure_socket.getpeercert()

    if not client_certificate:
        raise Exception("\nUnable to get certificate from client")
    else:
        print("Certificate of client ", address, " received")
        secure_socket.send(bytes('You now have a secured connection!','utf-8'))

    secure_socket.close()
```

Code Explanation:

1. **The Bind()** method assigns the address and port number to the server whereas the **listen** method listens for incoming clients.
2. **SSLContext()** creates an SSL context that holds data regarding SSL connections, certificates, and private keys. In the case of server-side sockets, it helps in managing the cache of SSL sessions.
3. After accepting the client, the **wrap_socket()** method wraps the socket in an SSL context, thus it takes the socket, server key, server certificate, and CA(Certificate Authority) certificate as the parameters.
4. **getpeercert()** method returns None in case of no certificate received from the other endpoint else it raises value error in case handshake protocol is not performed. In the case of a server socket, the client provides a certificate only if the server asks for it.
5. Finally, the connection is closed.

Server certificate:


```
-----BEGIN CERTIFICATE-----
MIIDjjCCAnYCFFE07rDIL9Fjhv1DWocu/M6cQXmNMA0GCSqGSIb3DQEBCwUAMHsx
CzAJBgNVBAYTAklOMRQwEgYDVQQIDAtNYWhhcmFzaHlyYTEPMA0GA1UEBwwGTXVt
YmFpMQswCQYDVQQKDAJjYTELMAkGA1UECwwCY2ExDzANBgNVBAMMBmNhLmNvbTEa
MBgGCSqGSIb3DQEJARYLY2FAdGVzdC5jb20wHhcNMjExMDE2MDkzNjUzWhcNMjIx
MDE2MDkzNjUzWjCBizELMAkGA1UEBhMCSW4xFDASBgNVBAgMC01haGFyYXNodHJh
MQ8wDQYDVQQHDAZNdW1iYWkxDzANBgNVBAoMBnNlcnZlcjEPMA0GA1UECwwGc2Vy
dmVyMRMwEQYDVQQDDApzZXJ2ZXIuY29tMR4wHAYJKoZIhvcNAQkBFg9zZXJ2ZXJA
dGVzdC5jb20wggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQDU/0L0NBtw
b04Il7vg1gTcwQmxsNVGWJeo/N49pt1V8Um7qiNM9VkeMbPRJCVehKDt8MPUZ4x7
pewk1Z/zehQjbChxdMvvlfUxiUFL90hH4wtYKfjYoYQh3WoQg0jm2sPc1ZffYMTf
Jb7WkEnz3E+qKw/aYvWw5nsfSq27748Lgo6BpaAOb2MtxwyQyJiWpuPuO210knHD
Uy8+D+yUJmJyIoZ0Ccymsf4B0bF3EM9KyEbC5UrJvKCLPfQJfMx5Dv5IWxnAU9Ns
gMVPN3Ffg+RkvhRCVrKtI6NydNxlYV66ggaNN9pLz/Feqw91Mp82K2GT0L7iKiVd
Rl44oGxYfYHjAgMBAAEwDQYJKoZIhvcNAQELBQADggEBAKKpjPiAu497z5lN/LQk
bWVIuxkAMDyW8PPy55c59m0QkbLTc7ZjUNLga/pAQDE1S3gnk+Of6BwZPrJVZb0L
RT6gVXcXe+hKWcPfMhuZIC5S19ear1Qj5LiJyPJOZxwBCH8dYZqN/Jdq7CE3Ay49
EAS91sFA5vVnw7ULT/3/+bKbG0f6ksGL49rQi4BAs56isnQzZtnlQYoUxk2BWr7b
NjlKkqrhAVxPiBREdYITY10BanueCVuT07n6PQzewbn41HxlpOaxaM49JTMk38U7
ZBH/xNnU/23N9yrM2/IpcUiZw1ZpAbIGM4A+ekYJQADlMWHsn4/SGvRBiTCnxO85
2bc=
-----END CERTIFICATE-----
```


Server Private Key:


```
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEA1P9C9DQbcG9OCJe74NYE3MEJsbDVRliXqPzePabdVfFJu6oj
TPVZHjGz0SQlXoSg7fDD1GeMe6XsJNWf83oUI2wocXTL75X1MYlBS/dIR+MLWCn4
2KGEId1qEINI5trD3NWX32DE3yW+1pBJ89xPqisP2mL1sOZ7H0qtu++PC4KOgaWg
Dm9jLccMkMiYlqbj7jttdJJxw1MvPg/slCZiciKGdAnMprH+AdGxdxDPSshGwuVK
ybygiz30CXzMeQ7+SFsZwFPTbIDFTzdxX4PkZL4UQlayrSOjcnTcZWFeuoIGjTfa
S8/xXqsPdTKfNithk9C+4iolXUZeOKBsWH2B4wIDAQABAoIBAQCM93vqf2oBmci9
ax7PCRYivCecUHp2mj+VefXkO8GVFaJE7695SY/3qdFmg1OwIHsZvxT8SKPzWbbW
EgRQceVOJFEH9dLa+BZU+JAcpMZTvXJOoUiupwb6Gr4Nu7XOA89JIpvnHPZu8V5C
RzCKkq6u4t0VHhWZRJEL/rLJBR1Y8ZQj8E2pyxchyznSj8LKqPX7j08gH/pcH6hE
EGVWybivmq2SfCLWDYgfu7IBpnVD4no3nCp8pSZ9RKbQWW+Bw6vCnlhDiLgyZq4H
9wXXHcgxRNxMnanGY7Edt/GSF2whigcvgLWN+t48UHNatHp9tDC7f9ivzDZqdaGM
+Id68GIBAoGBAOmSedxInh/DtBuIpTj16lHjoYa8VmjIJvjf2Dm+Jdrm2A+ysjL1
sL1iJk7scYHoDmhUKm0DQf3M5Zm6BrPrP9NW10bhEPO9ZHyIXz0Loe4X9f24n/vM
S1F6uIdfW15K92NRYrs2iOAPJtZzw3yeKj53BBZxJeAJgAz5CxJ+madjAoGBAOlz
Bb7TfoJ6w173cALL2/RPJyjKrZn5DD53qeuqrIuApas38RiECkUXDy4ZTFYSVBRp
whD9SeCScCxiv4H7oStOwaxO44UpeABOaSmVvboagzxuz9rwrX9ApQXXDFniEOVC
Cs0ZH5Gy9rbwXbd6/EVt8FdfILrPwUnnDPYbngOBAoGAWUCqpj1thu/OnBLcJ+b4
kjAsogLH9iQ9mC5ZPr9lUb+SYnVDWOu/jK67ur7pJN3a240F5UAwcF5ighquwwOr
EQQHIUORqcH+awPleLtANm4bhseNArfvNAR0DO6oSGDXJbGMy8MsDIpdqeb0lWMO
+22ZN0YnVfPrZYmo7HlKF8cCgYBEJYKI2P42IpYDk+GLv2y+3ZhFzRxuNQhA9/eP
RuxFcNFrsNHdYTYKQxf4UgQ5n/IRX2n6QaYeO0Epg835qLW8ian5nbHG/nr8Q+do
70BKIlErBAIDHHzZB8rkcIk3d8mM0wGKK4pOotpoPrRsyib4MYaWSruoLNt0slDT
Su6OgQKBgFW8o8Jd7wKXA+hyEnT1gGxxa2YlvoD49mx7ucOAYY/mJ4nrid74IwqH
hfkSLzR5N8hXwKU8Twmq0Cb+PyYCIO7xk0ngqgBJi8kPyP5Qj89ZWDWk36mrW+4A
3f0I5pE1pl0jmfWDn+gRuoo56X+cUc2NQm4/AA0E4lmNtGqhsiab
-----END RSA PRIVATE KEY-----
```

SSL Client-side implementation:

We have used the socket and the SSL library of python for implementing the client-side connection.

```
import socket
import ssl

client = socket.socket()

context = ssl.SSLContext()
context.verify_mode = ssl.CERT_REQUIRED

context.load_verify_locations("certificate/ca_cert.pem")

context.load_cert_chain(certfile="certificate/client_cert.pem", keyfile="certificate/client_key.pem")

secure_socket = context.wrap_socket(client)
secure_socket.connect(("localhost",9999))

server_certficate = secure_socket.getpeercert()

if not server_certficate:
    raise Exception("\nUnable to get certificate from server")
else:
    print("\nConnected to server")
    print("Receiving..")
    print(secure_socket.recv(1024).decode())

secure_socket.close()
client.close()

print("Connection closed!\n")
```

Code functions explanation:

1. **SSLcontext():** SSLContext acts as a placeholder where the policies and artifacts related to the secure communication of a client or a server can be stored.
   It is the first step of an SSL connection.
1. **verify_mode():** This policy on the certificate requirements of a host expected out of its peer is defined through the SSLContext.verify_mode attribute. For the successful validation of the server's certificate at the client-side, we assign CERT_REQUIRED to the attribute SSLContext.verify_mode
2. **load_verify_location():** This SSLContext class loads a set of CA certificates used for verifying the certificate of the peer.
3. **load_cert_chain():** This method load_cert_chain() loads an X.509 certificate and its private key into the SSLContext object. We pass the paths to the certificate file and the key file in this function as the parameters.
4. **wrap_socket():** This method adds the SSL layer to the socket.
5. **connect():** It connects a **TCP-based client socket** to a **TCP-based server socket.**
6. **getpeercert():** This method retrieves the digital certificate available if any, from the other end of the communication.

After calling the method get_peer_cert() we check whether we received the digital certificate from the server-side, if yes then the connection is established and the full-duplex communication can occur otherwise the connection is not established as the requirement is not fulfilled.

Client certificate:


```
-----BEGIN CERTIFICATE-----
MIIDjjCCAnYCFFE07rDIL9Fjhv1DWocu/M6cQXmOMA0GCSqGSIb3DQEBCwUAMHsx
CzAJBgNVBAYTAklOMRQwEgYDVQQIDAtNYWhhcmFzaHlyYTEPMA0GA1UEBwwGTXVt
YmFpMQswCQYDVQQKDAJjYTELMAkGA1UECwwCY2ExDzANBgNVBAMMBmNhLmNvbTEa
MBgGCSqGSIb3DQEJARYLY2FAdGVzdC5jb20wHhcNMjExMDE2MDkzNzE0WhcNMjIx
MDE2MDkzNzE0WjCBizELMAkGA1UEBhMCSU4xFDASBgNVBAgMC01haGFyYXNodHJh
MQ8wDQYDVQQHDAZNdW1iYWkxDzANBgNVBAoMBmNsaWVudDEPMA0GA1UECwwGY2xp
ZW50MRMwEQYDVQQDDApjbGllbnQuY29tMR4wHAYJKoZIhvcNAQkBFg9jbGllbnRA
dGVzdC5jb20wggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQDAD9L16KNd
uJ8DbBTuN9vbgogj9dPBIDQA4WjoRxKwYnvvWNSB667BfY8hZeK9dEZKpZmYzpcQ
U8VjehTb+5gPZ0qnrdbtBMan2cY9PEjIUaJ/BYiFnq6ebQSkE4emahqd18NmDcEX
VeDok7VCx2jfHrYbJXDr75N4XRljbIuSJqfXpMhY157SQdSd/zPdDGLtdW8ettO9
DkNn6ROQd0ZotbhTwWjevz7EfaCi6pLpbIUVRLjryZ39uS8KrDyyzgSpgYGUmhUX
exR/B6Mvu3uC0RvmkNY7XJZbWrIkDrhUhiEerIcREsL2mrEcdhQThKtrOcWXvvhx
21De6RstXZZDAgMBAAEwDQYJKoZIhvcNAQELBQADggEBAD+rObc11de3vloYuTON
qEhCuzcZddpKRWiDhp7AdU5uz24JOQkdirneOUlwRIshiYPbPRqIPJpDzS1BNokr
ueC/K8eyeEZj/5DM8ADbK+lh7RiFd41yDF6rYd97nknnSi1RXuoquyABnR/TwqSF
qvWSdSS1cZ0gaOsRANQrOZZWjHxVzC/HhKkor0/bPDkiIns4o2LYFbsEYr2T1WKk
0t7BVvv+AyLJrj2SJ7ICrxmyNS5UnLEHazA/nierq7Y2xPU/1qbH8L6XWZk1NxwV
0NoIeNlOqnE0faJ2sJudPMwAMnfip1rEFqFo7EUvKguo+RwXfQFNhPRWeq5jjx8Q
hZ8=
-----END CERTIFICATE-----
```


Client private key:


```
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAwA/S9eijXbifA2wU7jfb24KII/XTwSA0AOFo6EcSsGJ771jU
geuuwX2PIWXivXRGSqWZmM6XEFPFY3oU2/uYD2dKp63W7QTGp9nGPTxIyFGifwWI
hZ6unm0EpBOHpmoandfDZg3BF1Xg6JO1Qsdo3x62GyVw6++TeF0ZY2yLkian16TI
WNee0kHUnf8z3Qxi7XVvHrbTvQ5DZ+kTkHdGaLW4U8Fo3r8+xH2gouqS6WyFFUS4
68md/bkvCqw8ss4EqYGBlJoVF3sUfwejL7t7gtEb5pDWO1yWW1qyJA64VIYhHqyH
ERLC9pqxHHYUE4SraznFl774cdtQ3ukbLV2WQwIDAQABAoIBAFkHPMgGtaudCADD
GhSuyfg/0rGwox7Tuadq1YFOaMhgboitSZyLzm8cr7tQikE949N4XfGqWfYUjtwk
3AR45KzHOaDugUFPRHLMDQuXpTlONh/hRb6CpvkDGfKgpvGqH3ppaMcmuwoBIgdN
aOlynBdJGisNOQH1BQv3Q+yFhVESUk44V8MoLl6c7clwvCc3kBPVf2bJFA5KoomX
Weny7ja5qweuSBA0upAJYcepZzO+4zHCgc+xutIExqFQ7zTZaNvtvBz9tzkKGtZ+
PnlOHIasMhiT4SwjaJBixwmhK1ticqBd/sU+caWq3PbgBdR1Wvad4bZmHb+lP3XR
H/TBdikCgYEA6KGaLofMY+aGnaGpCw4O+Mu+KI1Ahs9YWITvYEb2Zu4iIUeZI7Wl
/u91hOmlw8Aa2ouTcNhn/TGQdSnn65RA+z7zfM+4LJcw9TUuEBpzjziEl0Wa8Ni8
rvQ3x/qzCsoAHC6mJbkVr9Z4qd7f8yq2FZfxeVybLYqQF8O+taqfj5cCgYEA01rt
xcz3Oo9pkmW10PecHYuAhjbI745A65WKsoJi2L1+zZEBHNV2smfpnOyWP2f2sKzz
+96qrSUd124l8BPI3JLwlDjYmmcmQANvKy7+rZRKCLr3UFjv77r49CcDgfPgwFgq
2+kbIDhOKbVs1LfNmqvHTgpZgz5NJLzhfTdthDUCgYBczgN1ZqWKgS/Y+OI4T853
QMjG4rsITPWgsr/Qd97a50tkXWzhixkC4ELQ2GlR83SDFUWnsh2iK8DGjQZBvC5E
TTPT6gY+e76DREjxEaxiSZjTxGfwh3aWkUUjmYcN2dI7a/zKddEKChSvKAPNvY9Q
hAJIeUJK48liQav2S3BGVwKBgQCdCxtRthKdeKJBHUITm2hsq5M8Jsj29wRWCC+e
pDM+SM9Hf08MVbB0r7Ft4H1jb6Rlcp13sOOw87tQr4+Q66J8AtKvz+1iDPLm7aZU
t/6Ui3LXOdU55luiDZOeFr8MeGGvidOOw45cSnoJk7zh8hDbtHfLDPWmB0za113U
5LKDYQKBgCon+Fkz2M0TkkS+BkNyuRgwnZWzdkkSvVMogIn987H+tlM/6fexgRIo
t9LMb+uW5UPzxRziD+XvdU+L1W3jHw47KAc4wkh1gNxM8OvpAxO4lwe68bkzlldd
KLCzh5K3DUD/5Pwk+9NbYv5yHLHqAkQCKrKLUpL5zzLnH6yQxn4W
-----END RSA PRIVATE KEY-----
```

Server Output:

```
(base) PS F:\sem5\cn_lab_git\Computer-Network-Lab\Week_5\p1> python server.py
Server is listening....

Certificate of client  ('127.0.0.1', 62488)  received
Client name:  c1

Certificate of client  ('127.0.0.1', 62489)  received
Client name:  c2
```

Client Output:

```
(base) PS F:\sem5\cn_lab_git\Computer-Network-Lab\Week_5\p1> python client.py

Enter name: c1

Connected to server
Receiving..
You now have a secured connection!
Connection closed!

(base) PS F:\sem5\cn_lab_git\Computer-Network-Lab\Week_5\p1> python client.py

Enter name: c2

Connected to server
Receiving..
You now have a secured connection!
Connection closed!

(base) PS F:\sem5\cn_lab_git\Computer-Network-Lab\Week_5\p1>
```

Q2. a. Capture TCP Packets and analyze the three-way handshake during the establishment of the communication.

TCP three-way handshake is a process that is used in a TCP/IP network to make a **reliable connection** between the server and client. The **connection is full-duplex,** and both sides synchronize (SYN) and acknowledge (ACK) each other. The TCP three-way handshake is performed in three steps namely:

1. **SYN,**
2. **SYN-ACK, and**
3. **ACK**.

For analyzing the three-way handshake during the establishment **WireShark** was used and the connection was initiated to **www.google.com** on **port number 80** using the command **telnet www.google.com 80** on the **command prompt.**

The following TCP packets were observed on WireShark:

Time	Source	Destination	Protocol	Length	Info
1.625712750	2400:1a00:b050:21bb…	2404:6800:4009:824:…	TCP	94	35258 → 80 [SYN] Seq=0 Win=64800 Len=0 MSS=1440 SACK_PERM=1 TSval=2572397198 TSecr=0 WS=128
1.673658441	2404:6800:4009:824:…	2400:1a00:b050:21bb…	TCP	94	80 → 35258 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1412 SACK_PERM=1 TSval=812833860 TSecr=25723
1.673717263	2400:1a00:b050:21bb…	2404:6800:4009:824:…	TCP	86	35258 → 80 [ACK] Seq=1 Ack=1 Win=64896 Len=0 TSval=2572397246 TSecr=812833860
5.398913175	2400:1a00:b050:21bb…	2404:6800:4009:824:…	TCP	86	35258 → 80 [FIN, ACK] Seq=1 Ack=1 Win=64896 Len=0 TSval=2572400972 TSecr=812833860
5.445461665	2404:6800:4009:824:…	2400:1a00:b050:21bb…	TCP	86	80 → 35258 [FIN, ACK] Seq=1 Ack=2 Win=65536 Len=0 TSval=812837632 TSecr=2572400972
5.445483089	2400:1a00:b050:21bb…	2404:6800:4009:824:…	TCP	86	35258 → 80 [ACK] Seq=2 Ack=2 Win=64896 Len=0 TSval=2572401018 TSecr=812837632

The **first segment is the TCP three-way handshake is SYN**, we will start off by analyzing the very first TCP packets.

TCP SYN

The **TCP SYN packet** has the following fields:

```
▸ Frame 26: 94 bytes on wire (752 bits), 94 bytes captured (752 bits) on interface wlp3s0, id 0
▸ Ethernet II, Src: Chongqin_90:7c:6d (ac:d5:64:90:7c:6d), Dst: Shenzhen_31:47:20 (68:d4:82:31:47:20)
▸ Internet Protocol Version 6, Src: 2400:1a00:b050:21bb:2f61:9b05:b8a5:b78c, Dst: 2404:6800:4009:824::2004
▸ Transmission Control Protocol, Src Port: 35258, Dst Port: 80, Seq: 0, Len: 0
```

We will observe the **Ethernet II** field, it has the following values:

```
▸ Frame 26: 94 bytes on wire (752 bits), 94 bytes captured (752 bits) on interface wlp3s0, id 0
▾ Ethernet II, Src: Chongqin_90:7c:6d (ac:d5:64:90:7c:6d), Dst: Shenzhen_31:47:20 (68:d4:82:31:47:20)
  ▸ Destination: Shenzhen_31:47:20 (68:d4:82:31:47:20)
  ▸ Source: Chongqin_90:7c:6d (ac:d5:64:90:7c:6d)
    Type: IPv6 (0x86dd)
▸ Internet Protocol Version 6, Src: 2400:1a00:b050:21bb:2f61:9b05:b8a5:b78c, Dst: 2404:6800:4009:824::2004
▸ Transmission Control Protocol, Src Port: 35258, Dst Port: 80, Seq: 0, Len: 0
```

Above the destination is my default gateway's MAC address and the source should be my MAC address. This was verified by using the ipconfig/all command. The IP address version can also be found here, as it mentions **IPv6.**

Observing the **IPv6** field:

```
▾ Internet Protocol Version 6, Src: 2400:1a00:b050:21bb:2f61:9b05:b8a5:b78c, Dst: 2404:6800:4009:824::2004
    0110 .... = Version: 6
  ▸ .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... .... .... 1001 1010 1100 1100 1011 = Flow Label: 0x9accb
    Payload Length: 40
    Next Header: TCP (6)
    Hop Limit: 64
    Source Address: 2400:1a00:b050:21bb:2f61:9b05:b8a5:b78c
    Destination Address: 2404:6800:4009:824::2004
```

Here the **Source address is my IP address** and the **destination address is the IP address of one of Google's web servers.**

Observing the **TCP field**:

```
▾ Transmission Control Protocol, Src Port: 35258, Dst Port: 80, Seq: 0, Len: 0
    Source Port: 35258
    Destination Port: 80
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence Number: 0    (relative sequence number)
    Sequence Number (raw): 53136432
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 0
    Acknowledgment number (raw): 0
    1010 .... = Header Length: 40 bytes (10)
  ▸ Flags: 0x002 (SYN)
    Window: 64800
    [Calculated window size: 64800]
    Checksum: 0x3f09 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  ▸ Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-Operation (NOP), Window scale
  ▸ [Timestamps]
```

Above we can observe the **source port** and the **destination port**, the destination port is **80** as we had established an HTTP connection. The **relative sequence** and **acknowledgment number** are 0 as this is the first packet.

Expanding the **Flags** field:

```
▾ Flags: 0x002 (SYN)
    000. .... .... = Reserved: Not set
    ...0 .... .... = Nonce: Not set
    .... 0... .... = Congestion Window Reduced (CWR): Not set
    .... .0.. .... = ECN-Echo: Not set
    .... ..0. .... = Urgent: Not set
    .... ...0 .... = Acknowledgment: Not set
    .... .... 0... = Push: Not set
    .... .... .0.. = Reset: Not set
  ▸ .... .... ..1. = Syn: Set
```

Here only the **Syn flag** is set to **one** which signifies that the packet is the first segment of the TCP three-way handshake.

The **second packet of the TCP three-way handshake is SYN-ACK**.

TCP SYN-ACK

We will observe the **Ethernet II** field, it has the following values:

```
▸ Frame 27: 94 bytes on wire (752 bits), 94 bytes captured (752 bits) on interface wlp3s0, id 0
▾ Ethernet II, Src: Shenzhen_31:47:20 (68:d4:82:31:47:20), Dst: Chongqin_90:7c:6d (ac:d5:64:90:7c:6d)
  ▸ Destination: Chongqin_90:7c:6d (ac:d5:64:90:7c:6d)
  ▸ Source: Shenzhen_31:47:20 (68:d4:82:31:47:20)
    Type: IPv6 (0x86dd)
▸ Internet Protocol Version 6, Src: 2404:6800:4009:824::2004, Dst: 2400:1a00:b050:21bb:2f61:9b05:b8a5:b78c
▸ Transmission Control Protocol, Src Port: 80, Dst Port: 35258, Seq: 0, Ack: 1, Len: 0
```

Above the **destination is my MAC address** and the **source should be my default gateway MAC address,** the IP version is observed to be **IPv6.**

Observing the **IPv6** field:

```
▾ Internet Protocol Version 6, Src: 2404:6800:4009:824::2004, Dst: 2400:1a00:b050:21bb:2f61:9b05:b8a5:b78c
    0110 .... = Version: 6
  ▸ .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... .... .... 1110 1010 0110 0010 0110 = Flow Label: 0xea626
    Payload Length: 40
    Next Header: TCP (6)
    Hop Limit: 55
    Source Address: 2404:6800:4009:824::2004
    Destination Address: 2400:1a00:b050:21bb:2f61:9b05:b8a5:b78c
```

Here the **Source address is the IP address of one of Google's web servers** and the **destination IP address is the same dynamic port selected for the connection** as observed in the SYN packet.

Observing the **TCP field**:

```
▾ Transmission Control Protocol, Src Port: 80, Dst Port: 35258, Seq: 0, Ack: 1, Len: 0
    Source Port: 80
    Destination Port: 35258
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence Number: 0    (relative sequence number)
    Sequence Number (raw): 3088572129
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 53136433
    1010 .... = Header Length: 40 bytes (10)
  ▸ Flags: 0x012 (SYN, ACK)
    Window: 65535
    [Calculated window size: 65535]
    Checksum: 0xc84b [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  ▸ Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-Operation (NOP), Window scale
  ▸ [SEQ/ACK analysis]
  ▸ [Timestamps]
```

Above we can observe the source port is **80**. The **relative sequence** number is **0** and the **acknowledgment number** is **1** as this is the second TCP packet. The acknowledgment number is 1 as the sequence number in the previous packer was 0, this is because of **cumulative acknowledgment**.

Expanding the **Flags** field:

```
▾ Flags: 0x012 (SYN, ACK)
    000. .... .... = Reserved: Not set
    ...0 .... .... = Nonce: Not set
    .... 0... .... = Congestion Window Reduced (CWR): Not set
    .... .0.. .... = ECN-Echo: Not set
    .... ..0. .... = Urgent: Not set
    .... ...1 .... = Acknowledgment: Set
    .... .... 0... = Push: Not set
    .... .... .0.. = Reset: Not set
  ▸ .... .... ..1. = Syn: Set
    .... .... ...0 = Fin: Not set
    [TCP Flags: ·······A··S·]
```

Here the only the **SYN** and **Acknowledgement flag is set to one** this signifies that the packet is the **second segment** of the TCP three-way handshake.

The **second packet of the TCP three-way handshake is ACK**.

TCP ACK

We will observe the **Ethernet II** field, it has the following values:

```
▸ Frame 28: 86 bytes on wire (688 bits), 86 bytes captured (688 bits) on interface wlp3s0, id 0
▾ Ethernet II, Src: Chongqin_90:7c:6d (ac:d5:64:90:7c:6d), Dst: Shenzhen_31:47:20 (68:d4:82:31:47:20)
  ▸ Destination: Shenzhen_31:47:20 (68:d4:82:31:47:20)
  ▸ Source: Chongqin_90:7c:6d (ac:d5:64:90:7c:6d)
    Type: IPv6 (0x86dd)
▸ Internet Protocol Version 6, Src: 2400:1a00:b050:21bb:2f61:9b05:b8a5:b78c, Dst: 2404:6800:4009:824::2004
▸ Transmission Control Protocol, Src Port: 35258, Dst Port: 80, Seq: 1, Ack: 1, Len: 0
```

Above the **destination is my default gateway's MAC address** and the **source should be my MAC address**. This was verified by using the ipconfig/all command. The IP version is shown to be IPv6.

Observing the **IPv6** field:

```
▾ Internet Protocol Version 6, Src: 2400:1a00:b050:21bb:2f61:9b05:b8a5:b78c, Dst: 2404:6800:4009:824::2004
    0110 .... = Version: 6
  ▸ .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... .... .... 1001 1010 1100 1100 1011 = Flow Label: 0x9accb
    Payload Length: 32
    Next Header: TCP (6)
    Hop Limit: 64
    Source Address: 2400:1a00:b050:21bb:2f61:9b05:b8a5:b78c
    Destination Address: 2404:6800:4009:824::2004
```

Here the **Source address is my IP address** and the **destination address is the IP**

address of one of Google's web servers.

Observing the **TCP field**:

```
Transmission Control Protocol, Src Port: 35258, Dst Port: 80, Seq: 1, Ack: 1, Len: 0
   Source Port: 35258
   Destination Port: 80
   [Stream index: 0]
   [TCP Segment Len: 0]
   Sequence Number: 1    (relative sequence number)
   Sequence Number (raw): 53136433
   [Next Sequence Number: 1    (relative sequence number)]
   Acknowledgment Number: 1    (relative ack number)
   Acknowledgment number (raw): 3088572130
   1000 .... = Header Length: 32 bytes (8)
 > Flags: 0x010 (ACK)
   Window: 507
   [Calculated window size: 64896]
   [Window size scaling factor: 128]
   Checksum: 0x3f01 [unverified]
   [Checksum Status: Unverified]
   Urgent Pointer: 0
 > Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
 > [SEQ/ACK analysis]
 > [Timestamps]
```

Above we can observe the source port and the destination port, the destination port is **80**. The acknowledgment number is 1 as the sequence number in the previous packer was 0, this is because of **cumulative acknowledgment**.

Expanding the Flags field:

```
v Flags: 0x010 (ACK)
    000. .... .... = Reserved: Not set
    ...0 .... .... = Nonce: Not set
    .... 0... .... = Congestion Window Reduced (CWR): Not set
    .... .0.. .... = ECN-Echo: Not set
    .... ..0. .... = Urgent: Not set
    .... ...1 .... = Acknowledgment: Set
    .... .... 0... = Push: Not set
    .... .... .0.. = Reset: Not set
    .... .... ..0. = Syn: Not set
    .... .... ...0 = Fin: Not set
    [TCP Flags: ·······A····]
```

Here the **only the acknowledgment flag is set to one** this signifies that the packet is the **last segment** of the TCP three-way handshake.

Hence, once the TCP three-way handshake is established the client and the server can connect with and a **full-duplex connection** is ensured. The three-way handshake always occurs before sending/receiving the HTTP requests.

Q2. b. Capture TCP Packets and identify if there are any retransmitted segments.

TCP retransmission involves resending the packets over the network that were either lost or damaged.

How retransmission works:

1. The sender sends messages/data to the receiver.
2. In case of successful transmission, the receiver sends an acknowledgment to the sender.
3. In case the sender does not receive the acknowledgment within a certain time, it retransmits the message/data.

For analyzing TCP retransmission we can analyze the following packets:

tcp.stream eq 2

No.	Time	Source	Destination	Protocol	Length	DNS Time	Info
3144	0.000000	192.168.43.174	172.217.160.195	TCP	54		64843 → 443 [FIN, ACK] Seq=1 Ack=1 Win=256 Len=0
3159	0.307586	192.168.43.174	172.217.160.195	TCP	54		[TCP Retransmission] 64843 → 443 [FIN, ACK] Seq=1 Ack=1 Win=256 Len=0
3166	0.610353	192.168.43.174	172.217.160.195	TCP	54		[TCP Retransmission] 64843 → 443 [FIN, ACK] Seq=1 Ack=1 Win=256 Len=0
3167	0.050597	172.217.160.…	192.168.43.174	TCP	54		443 → 64843 [FIN, ACK] Seq=1 Ack=2 Win=271 Len=0
3168	0.000152	192.168.43.174	172.217.160.195	TCP	54		64843 → 443 [ACK] Seq=2 Ack=2 Win=256 Len=0

Above we can view the traffic for a single TCP communication as per the applied filter i.e. **tcp.stream eq 46.**
In the Time field, we can observe the time interval between the current and previous packet.

We can observe that the first **[FIN, ACK]** is retransmitted after nearly 0.3s whereas the second one is retransmitted **0.6s**. Thereafter, we can see the [FIN, ACK] is received on an interval of nearly **0.05s.** Thus, if we sum up the time, it took around **0.95s** to send the [FIN, ACK] packet.

We further observe the **TCP Analysis Flags** subfield:

```
[SEQ/ACK analysis]
  [TCP Analysis Flags]
    [Expert Info (Note/Sequence): This frame is a (suspected) retransmission]
      [This frame is a (suspected) retransmission]
      [Severity level: Note]
      [Group: Sequence]
    [The RTO for this segment was: 0.307586000 seconds]
    [RTO based on delta from frame: 3144]
```

RTO(Retransmission Timeout): It is used by TCP to **retransmit lost segments.** Whenever TCP sends a segment, a timer starts and runs until ACK is received. In case the timer exceeds the timeout, then the segment is retransmitted. The value of RTO is based on the smoothed round-trip time and its deviation.

For the above packet, **RTO** = **0.307s**

Thus, if we observe the RTO for the **second retransmission** we get **RTO=0.917s**:

```
[SEQ/ACK analysis]
  [TCP Analysis Flags]
    [Expert Info (Note/Sequence): This frame is a (suspected) retransmission]
      [This frame is a (suspected) retransmission]
      [Severity level: Note]
      [Group: Sequence]
  [The RTO for this segment was: 0.917939000 seconds]
  [RTO based on delta from frame: 3144]
```

Reasons for **packet retransmission**:

1. Network congestions: In this case, the packets may be dropped.
2. Tight Router rules: Gives preferential treatment to certain protocols.
3. Receiving TCP segments out of order.

TCP retransmission ensures **reliable end-to-end data transfer** and helps in **troubleshooting the data loss.**
