Faculty of Engineering & Technology

Electrical & Computer Engineering Department

ENCS5322

Crypto Lab - TLS Lab

Prepared by: Diana Naseer.

Std. ID: 1210363.

Section: 1.

Instructor : Dr. Ahmad Alsadeh.

Date: 1/8/2024

Abstract:

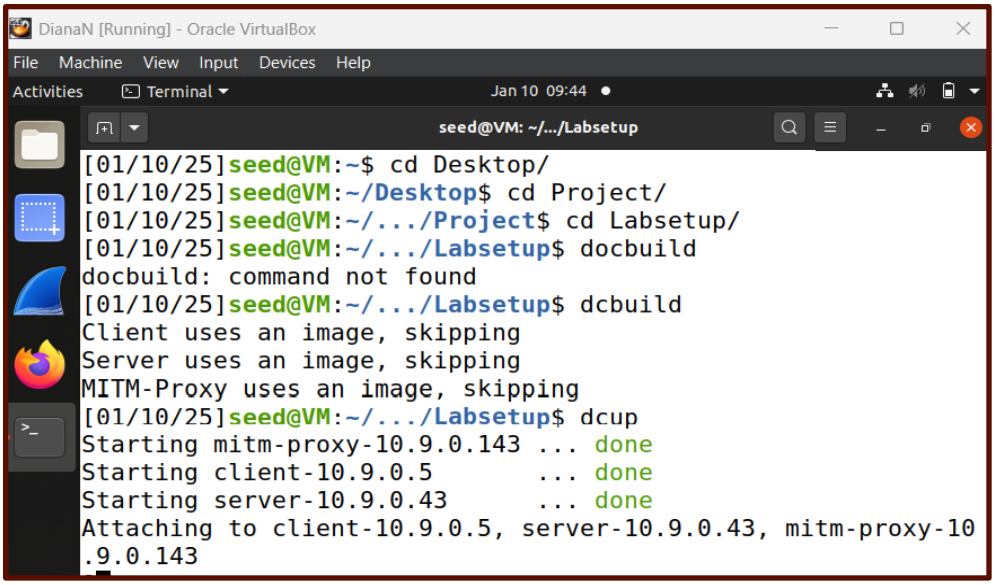
This lab explores the fundamentals of Transport Layer Security (TLS), focusing on its role in ensuring secure communication. Participants begin by implementing a TLS client to understand the handshake process, certificate validation, and secure data exchange. The lab extends to creating a TLS server and managing server certificates, including multi-domain setups with Subject Alternative Names (SANs). Finally, a basic HTTPS proxy demonstrates vulnerabilities like Man-In-The-Middle (MITM) attacks. Through hands-on tasks, this lab provides practical insights into TLS, PKI, and HTTPS, equipping participants to secure communication in real-world applications.

Contents

Task1: TLS Client

In this task, we will incrementally build a simple TLS client program. Through the process, students will understand the essential elements and security considerations in TLS programming. We will run this client program on the client container..

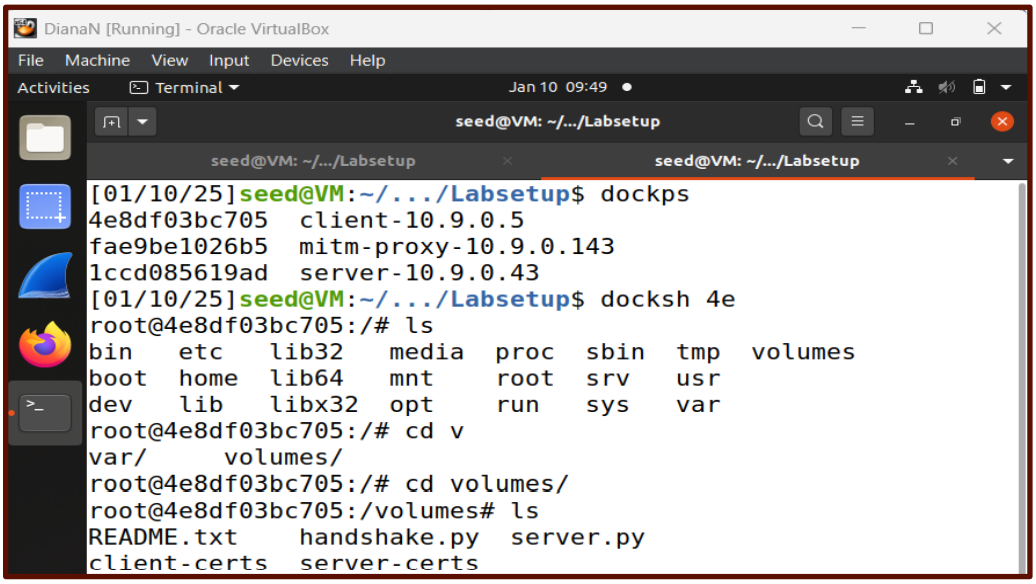
First we setup the containers:



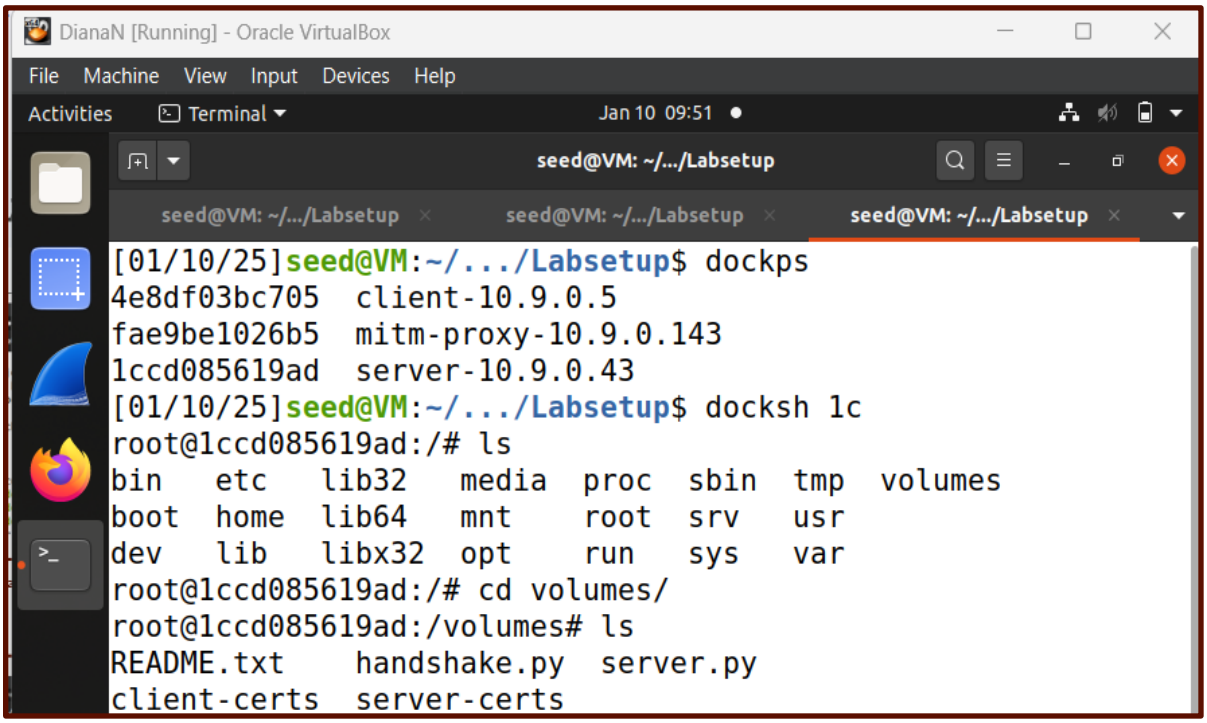
Now, All the containers will be running in the background.

- To start a shell on that container.

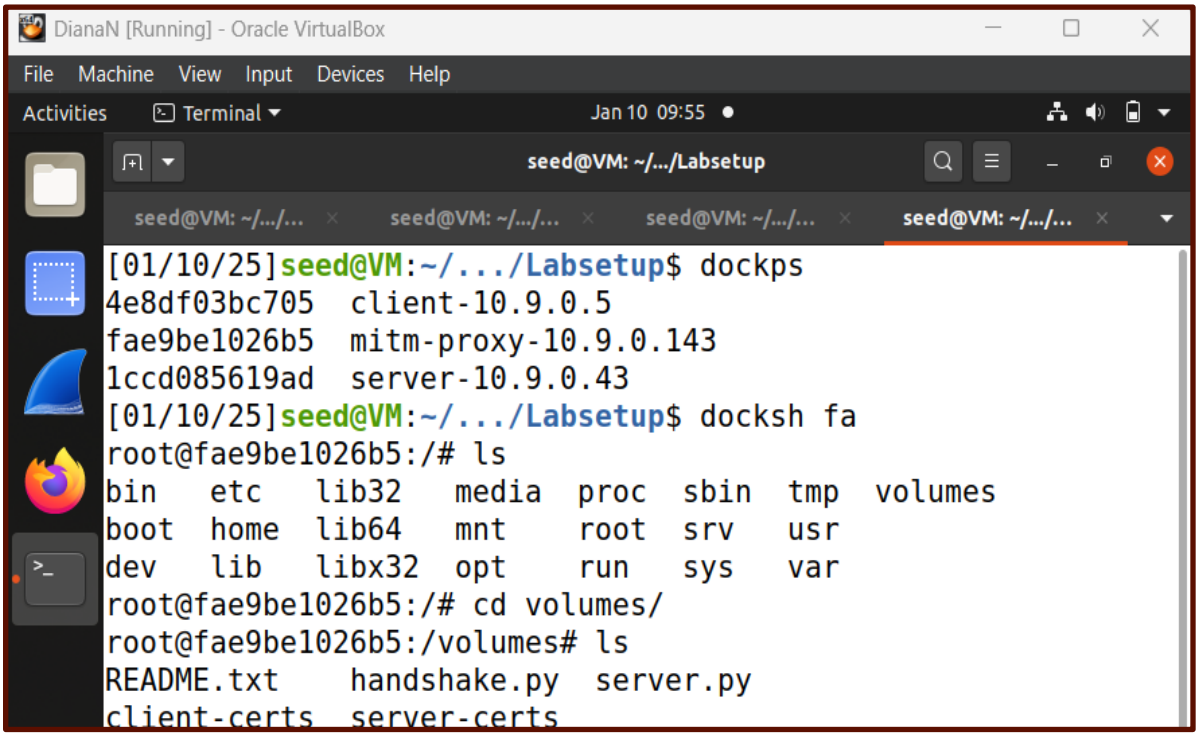
  For Client:



For Server:



For Man in the middle proxy:

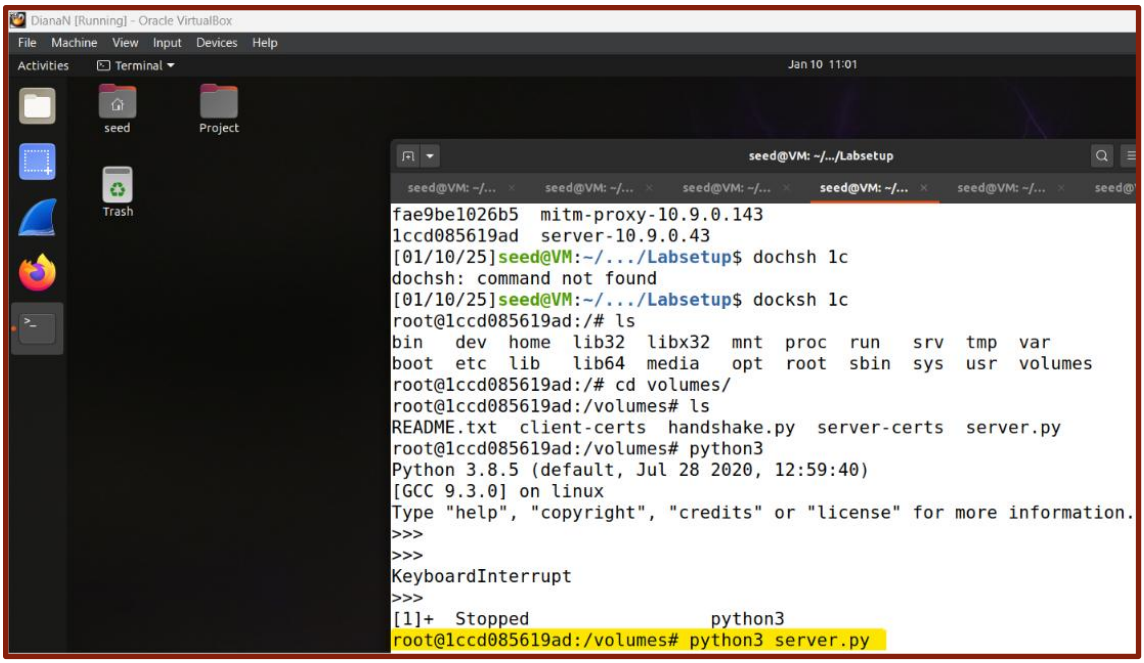


Task 1.a: TLS Handshake

Before a client and a server can communicate securely, several things need to be set up first, including what encryption algorithm and key will be used, what MAC algorithm will be used, what algorithm should be used for the key exchange, etc. These cryptographic parameters need to be agreed upon by the client and the server. That is the primary purpose of the TLS Handshake Protocol. In this task, we focus on the TLS handshake protocol.

In server container the user attempts to execute the server.py script using:

python3 server.py

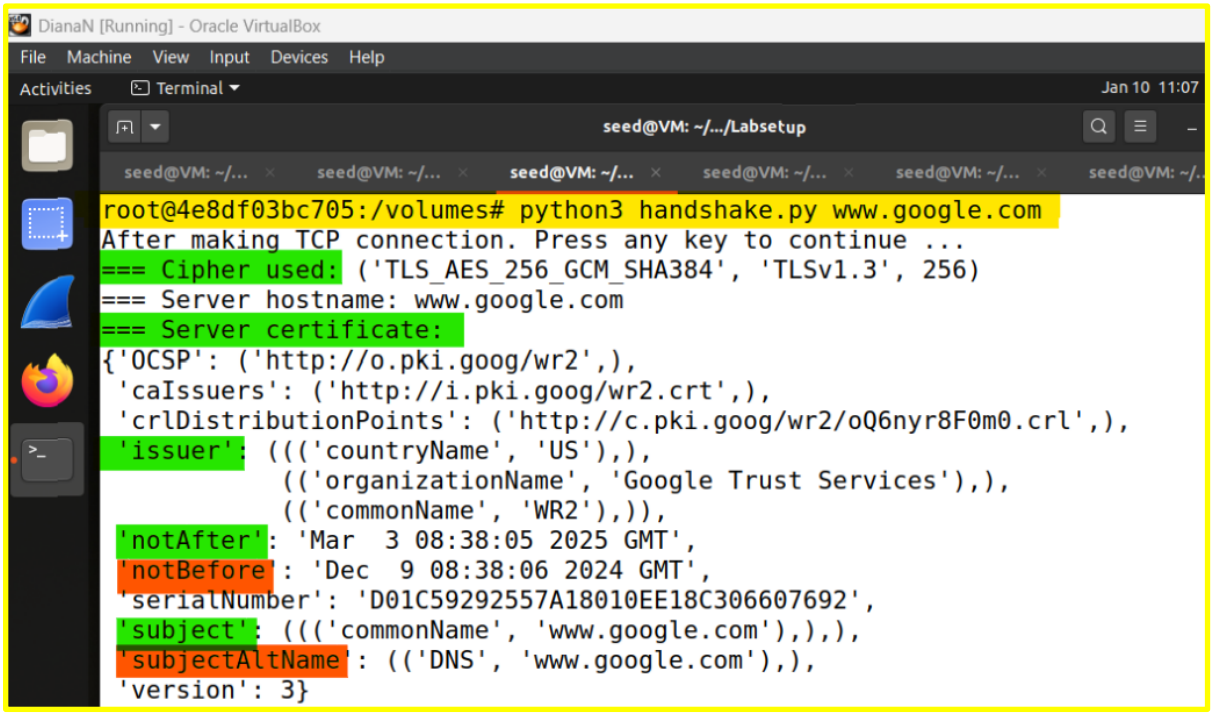


This action likely initializes the TLS server, allowing it to listen for client connections on a predefined port (likely 443, as per the lab instructions). The script facilitates the lab's experimentation with secure server-client communications.

Run the Client Program:

- Navigate to the directory where the handshake.py script is located.
- Run the script against a real server, such as www.google.com

1.a.1: Part 1 & 2 (cipher,server certificate):



The command python3 handshake.py www.google.com was executed from the /volumes directory inside a Docker container, initiating a TLS handshake with Google's server.

- **The cipher** suite provides robust encryption (AES-256), fast and secure data integrity (GCM), and strong hashing (SHA384), ensuring a highly secure communication channel between the client and the server. =>"TLS_AES_256_GCM_SHA384 "
- **the server certificate in the program:**

Issuer:

Country Name (C): US **Organization Name (O):** Google Trust Services

Common Name (CN): WR2

This means the certificate was issued by Google Trust Services, specifically through their intermediate certificate authority "WR2."

Not Before (Start Date): December 9, 2024, at 08:38:06 GMT.

Not After (Expiration Date): March 3, 2025, at 08:38:05 GMT.

The certificate is valid during this period, ensuring it hasn't expired.

Subject (Server Information):

Common Name (CN): www.google.com

Subject Alternative Name (SAN):

Lists additional DNS names the certificate is valid for.

In this case, it's also www.google.com.

==========

1.a.2: Part 3 (/etc/ssl/certs.):

The purpose of /etc/ssl/certs**:**

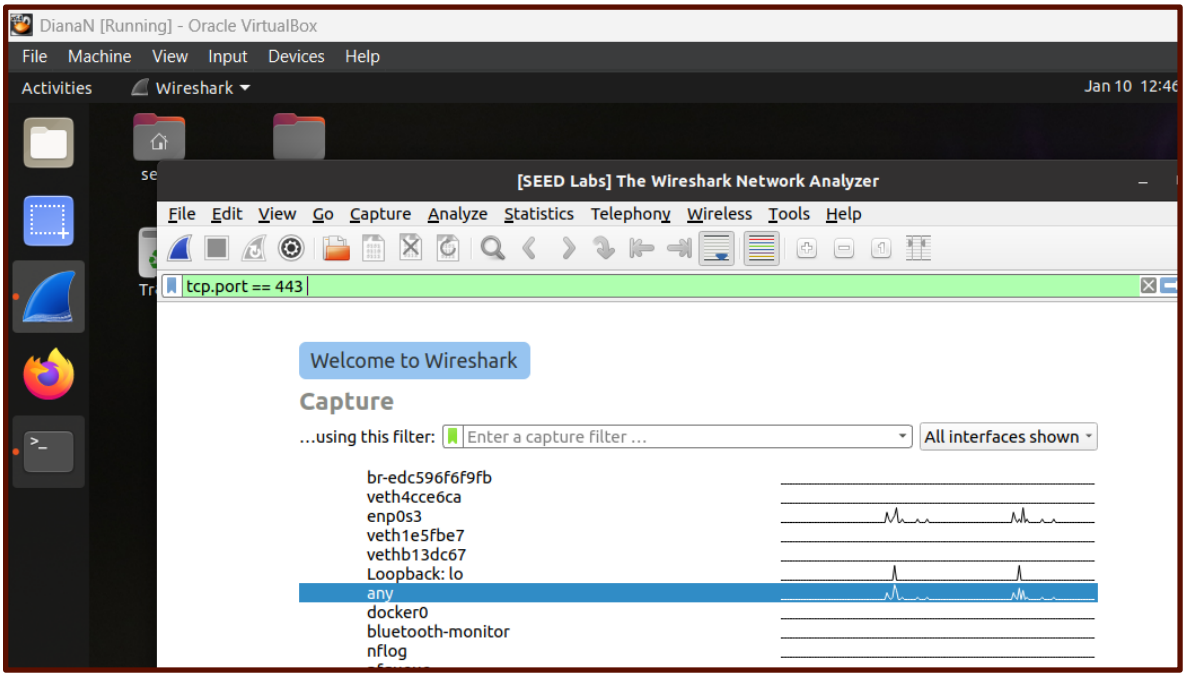
During the TLS handshake, it presents its certificate to the client in order to establish a secure connection. The client then verifies this certificate against trusted Certificate Authority (CA) certificates stored in /etc/ssl/certs. For the client to validate the server's certificate, it needs to find a matching CA certificate by looking at its issuer details. If a valid and matching CA certificate is found, the server certificate is accepted as trustworthy. It relies on the chain of trust, meaning that the server certificate must go back to some trusted CA within /etc/ssl/certs. In case the chain is partially missing or at least one of its parts is an invalid intermediate or root CA certificate, the server certificate is refused, and the connection is terminated.

==========

1.a.3 Part 4 (Wireshark):

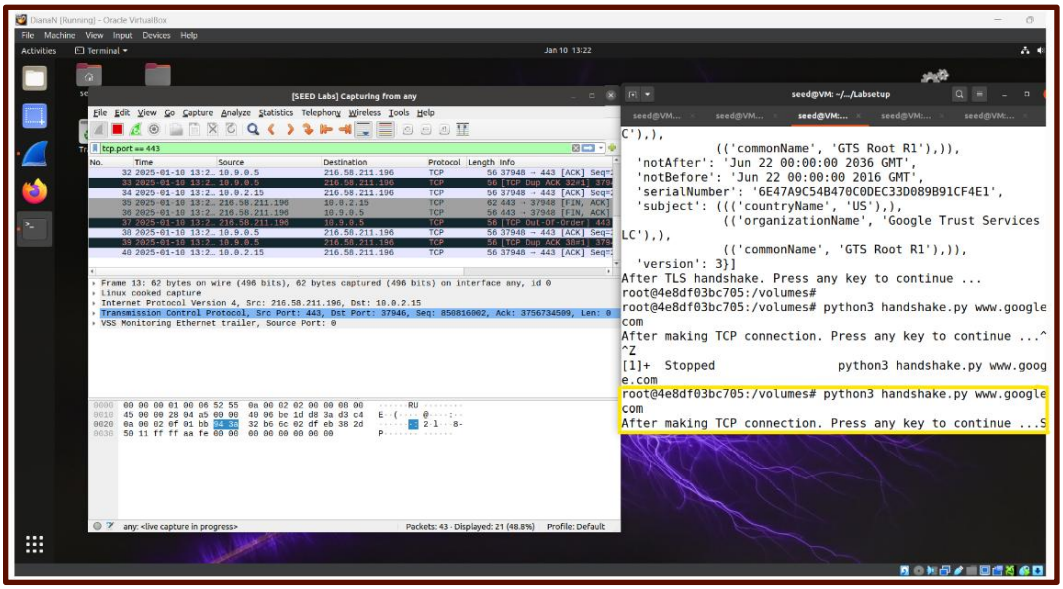
To capture the network traffic during the execution of the program.

I Set a capture filter for HTTPS traffic on port 443:



After this; Execute the Python script inside the client container, then capture the wireshark network traffic in TCP connection.

1. TCP Handshake



- **Packets Observed in Wireshark:**

  SYN (Client → Server): Initiates the connection to the server on port 443.

  SYN-ACK (Server → Client): Acknowledges the connection request.

  ACK (Client → Server): Completes the handshake and establishes a reliable connection.

- **Triggered By:**

  The **TCP handshake** begins when the socket.connect() function is invoked in the handshake.py program. This function ensures that a reliable connection is set up before any higher-layer protocols are involved.

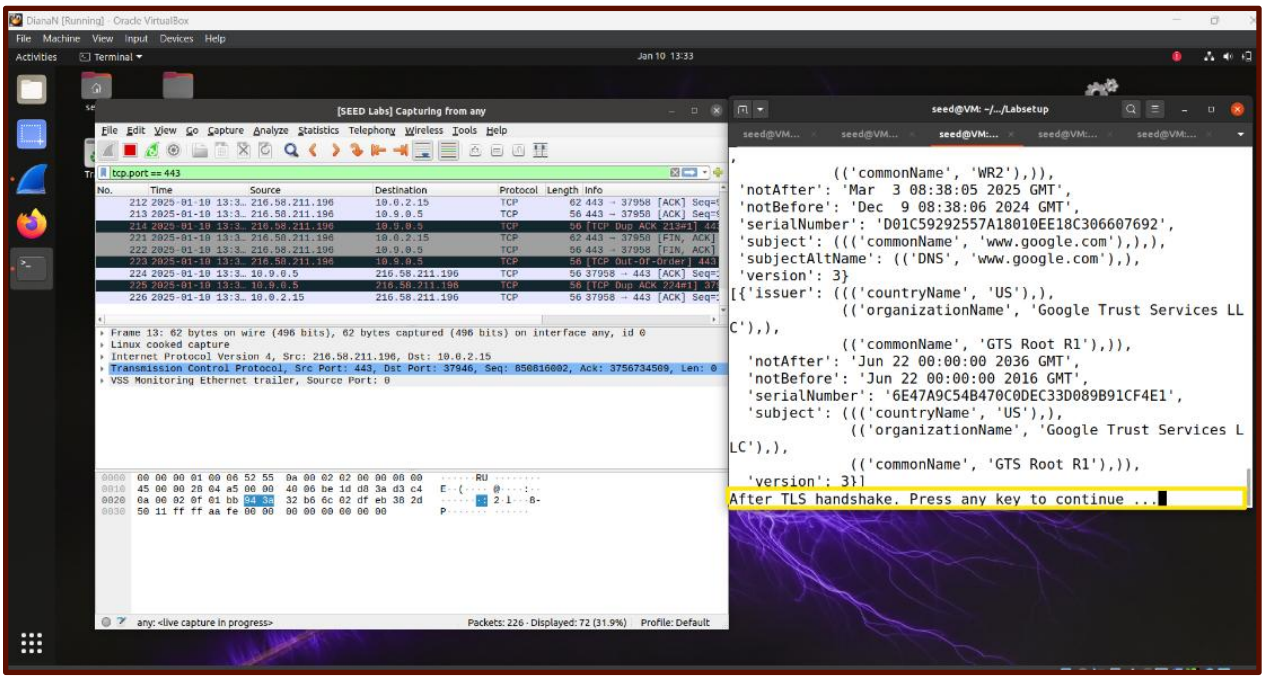
- **Purpose of the TCP Handshake:**

  Establishes a reliable and bidirectional communication channel between the client and server, ensuring readiness for data exchange.

==========

After press another time; Execute the Python script inside the client container, then capture the wireshark network traffic in TLS connection.

2. TLS Handshake



- **Packets Observed in Wireshark:**

  - **ClientHello (Client → Server):**

    Starts the TLS handshake, providing supported protocols ,cipher suites, and random data.

  - **ServerHello (Server → Client):**

    Responds with the selected TLS version, cipher suite, and server's random data.

  - **Certificate (Server → Client):**

    Server sends its certificate for client validation.

- **Key Exchange (Both Directions):**

  Key material is exchanged to establish encryption parameters.

- **Finished (Both Directions):**

  Final confirmation from both sides indicating the secure connection is ready.

- **Triggered By:**
  - The **TLS handshake** is initiated when the ssock.do_handshake() function is executed in the handshake.py program. This occurs after the TCP connection has been established.

- **Purpose of the TLS Handshake:**
  - Secures the connection by:

    Negotiating encryption parameters (cipher suite and keys).

    Authenticating the server's identity using its certificate.

---

3. Relationship Between TCP and TLS Handshakes

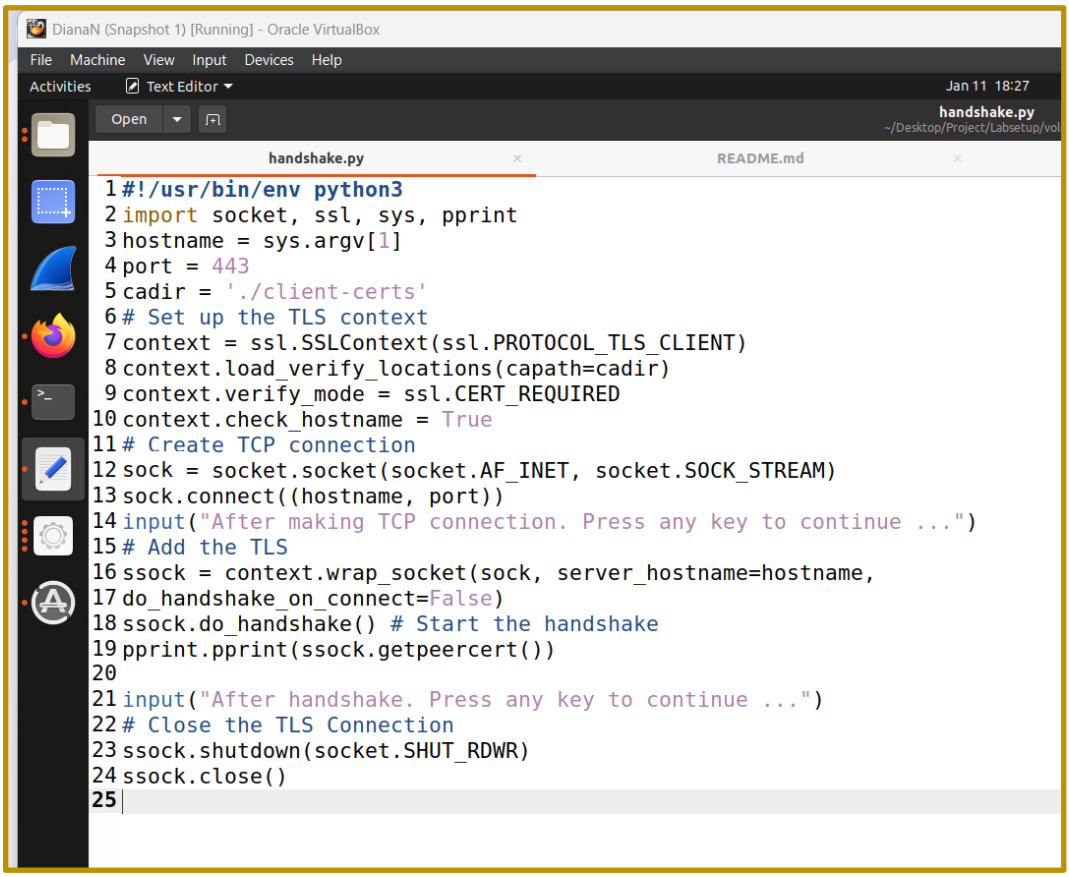
- The TLS handshake relies on the TCP handshake to have a reliable channel of communication; without TCP, TLS cannot proceed. While the TCP handshake works at the transport layer, which is Layer 4 of the OSI model, creating the basic communication channel, the TLS handshake adds encryption and authentication at the application layer, Layer 7. The TCP handshake provides reliability in establishing a connection, while the TLS handshake makes sure the connection is secure by encrypting data and verifying the identity of the server.

Task 1.b: CA's Certifications.

The objective is to set up the client program so that it validates server certificates using a specified custom directory for Certificate Authority (CA) certificates. This configuration ensures that the client only establishes secure connections with trusted servers by referencing the CA certificates in the designated directory.

The Python script (handshake.py) is designed to:

- Establish a TCP connection with a specified server (www.example.com or www.google.com).
- Perform a TLS handshake to validate the server's certificate against a set of trusted CA certificates located in a custom directory (./client-certs).



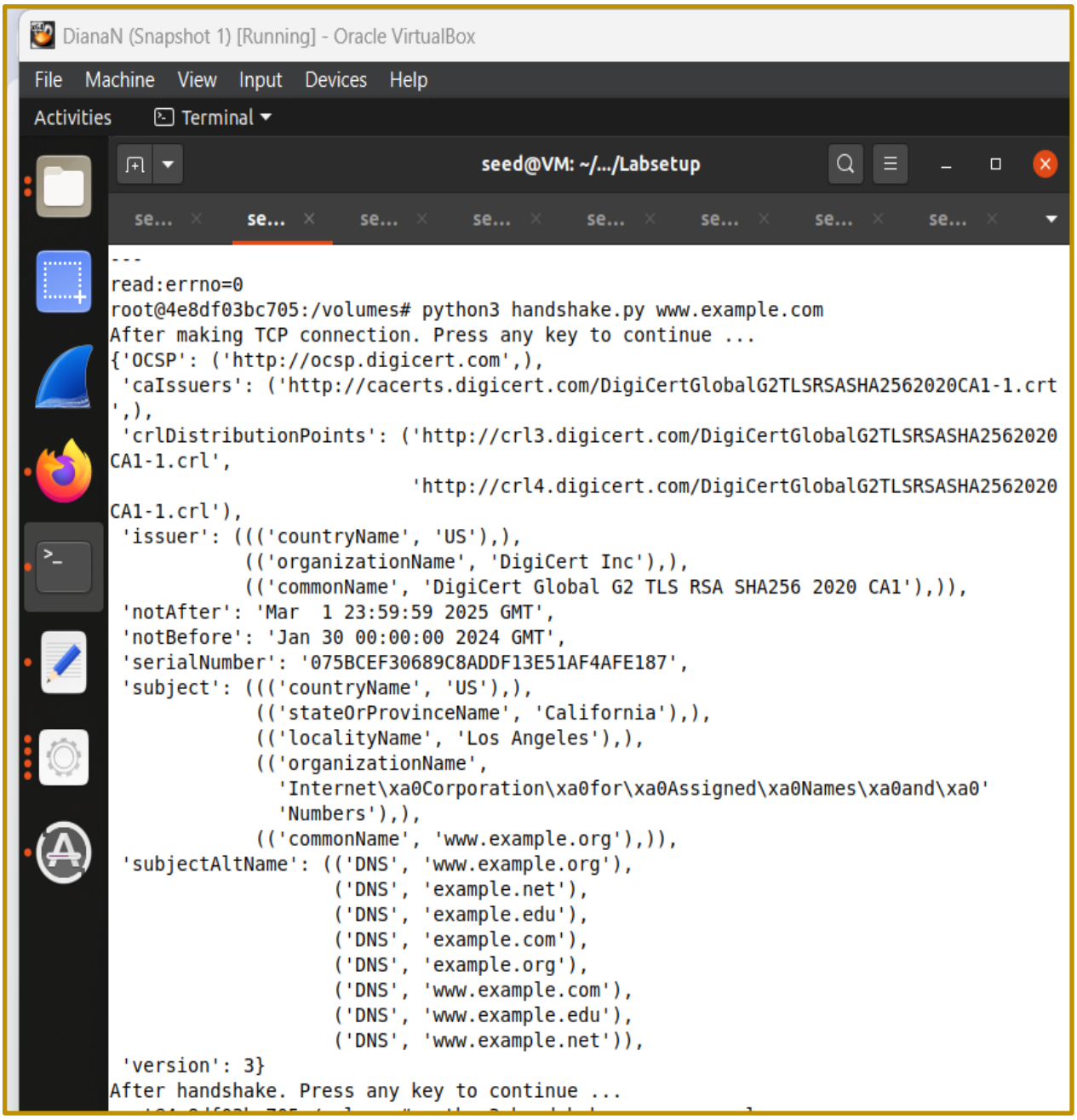
The test case describes the successful process of establishing a secure connection to **www.example.com** and validating its TLS certificate:

Certificate Details:

- **Issuer**: DigiCert Global G2 TLS RSA SHA256 2020 CA1
- **Validity**:
    - Start Date: January 30, 2024
    - Expiration Date: March 1, 2025
- **Subject**: www.example.org (including SANs such as example.com, example.net, etc.)

Results:

- **Handshake Outcome**: Successful
- **Certificate Validation**: Performed using CA certificates sourced from a specified custom directory.



==========

The test case describes a successful connection to www.google.com with insights into the certificate validation process and key observations:

Certificate Details

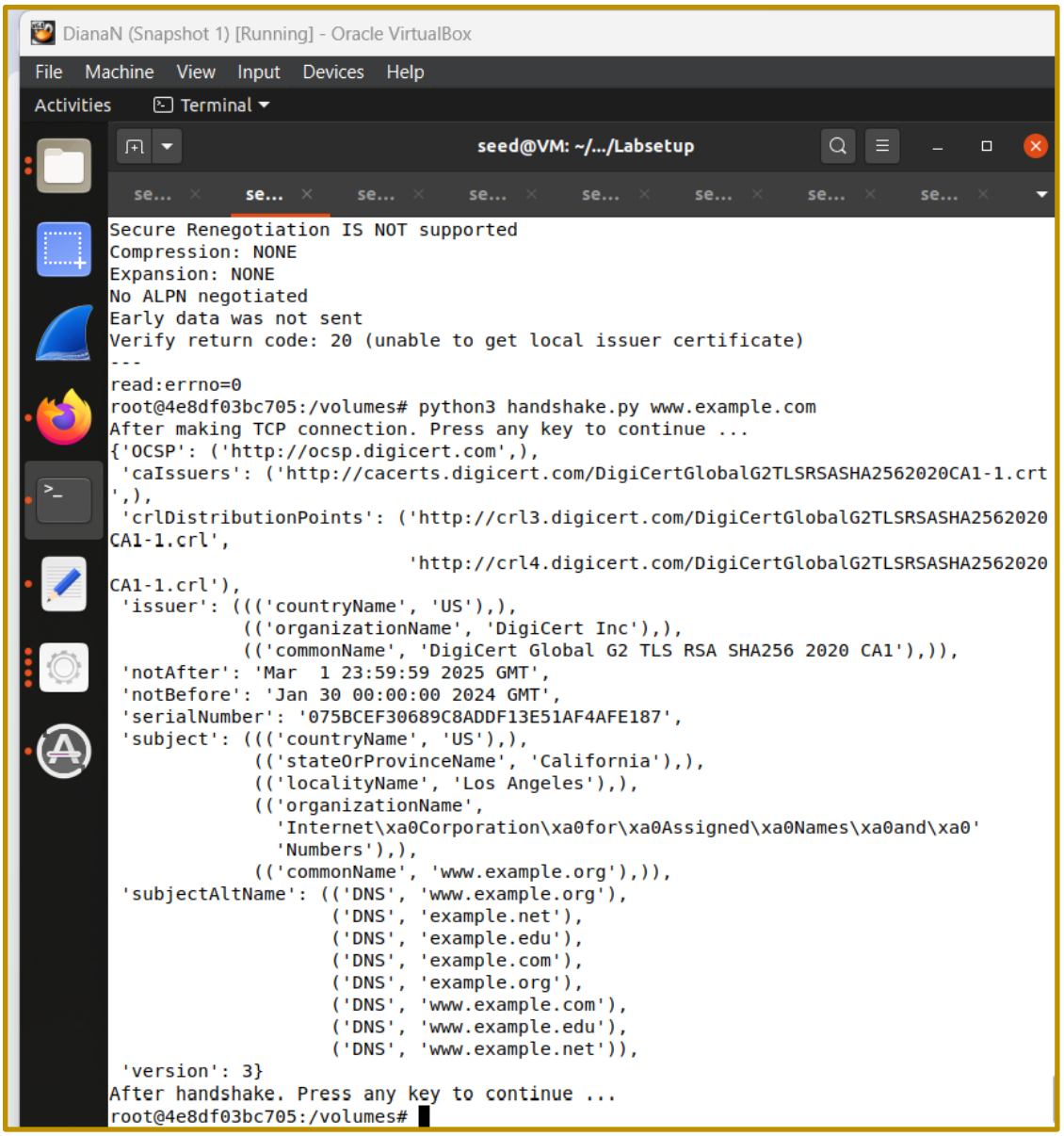
- **Issuer**: Google Trust Services (GTS)
- **Validity**:
    - Start Date: December 9, 2024
    - Expiration Date: March 3, 2025
- **Subject**: www.google.com

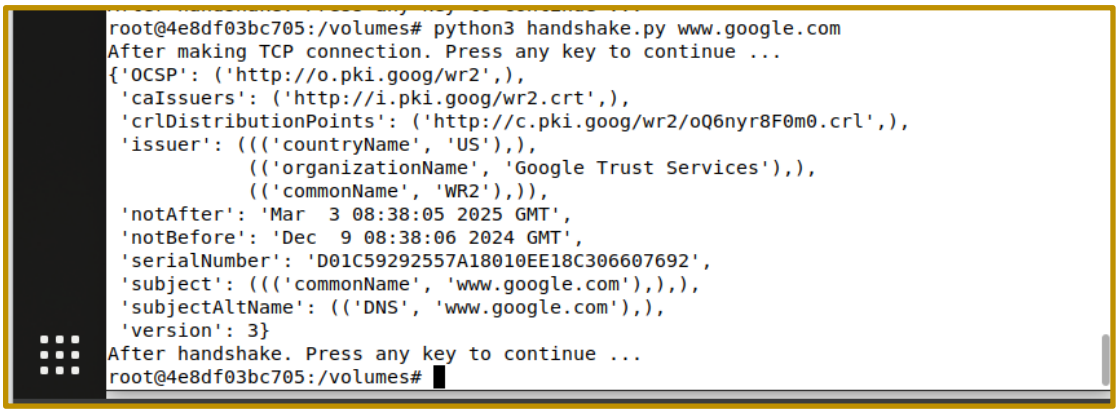
Results

- **Handshake Outcome**: Successful
- **Certificate Validation**: Completed using CA certificates from a custom directory (./client-certs).

Key Observations

1. **Certificate Chain**:
    - The server's certificate and the intermediate CA certificate were retrieved.
    - Validation required both the root and intermediate CA certificates to be present in the custom directory (./client-certs).
2. **Custom CA Directory**:
    - The script loaded CA certificates from the custom directory, bypassing the system-wide directory (/etc/ssl/certs).
    - This approach allowed for more granular control over trusted CAs.
3. **Validation Errors**:
    - Missing or misconfigured CA certificates in the custom directory would result in a **CERTIFICATE_VERIFY_FAILED** error, halting the process.


DianaN (Snapshot 1) [Running] - Oracle VirtualBox
File Machine View Input Devices Help
Activities Terminal
seed@VM: ~/.../Labsetup
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 20 (unable to get local issuer certificate)
---
read:errno=0
root@4e8df03bc705:/volumes# python3 handshake.py www.example.com
After making TCP connection. Press any key to continue ...
{'OCSP': ('http://ocsp.digicert.com',),
'caIssuers': ('http://cacerts.digicert.com/DigiCertGlobalG2TLSRSASHA2562020CA1-1.crt',),
'crlDistributionPoints': ('http://crl3.digicert.com/DigiCertGlobalG2TLSRSASHA2562020CA1-1.crl',
'http://crl4.digicert.com/DigiCertGlobalG2TLSRSASHA2562020CA1-1.crl'),
'issuer': ((('countryName', 'US'),),
(('organizationName', 'DigiCert Inc'),),
(('commonName', 'DigiCert Global G2 TLS RSA SHA256 2020 CA1'),)),
'notAfter': 'Mar 1 23:59:59 2025 GMT',
'notBefore': 'Jan 30 00:00:00 2024 GMT',
'serialNumber': '075BCEF30689C8ADDF13E51AF4AFE187',
'subject': ((('countryName', 'US'),),
(('stateOrProvinceName', 'California'),),
(('localityName', 'Los Angeles'),),
(('organizationName',
'Internet\xa0Corporation\xa0for\xa0Assigned\xa0Names\xa0and\xa0'
'Numbers'),),
(('commonName', 'www.example.org'),)),
'subjectAltName': (('DNS', 'www.example.org'),
('DNS', 'example.net'),
('DNS', 'example.edu'),
('DNS', 'example.com'),
('DNS', 'example.org'),
('DNS', 'www.example.com'),
('DNS', 'www.example.edu'),
('DNS', 'www.example.net')),
'version': 3}
After handshake. Press any key to continue ...
root@4e8df03bc705:/volumes#

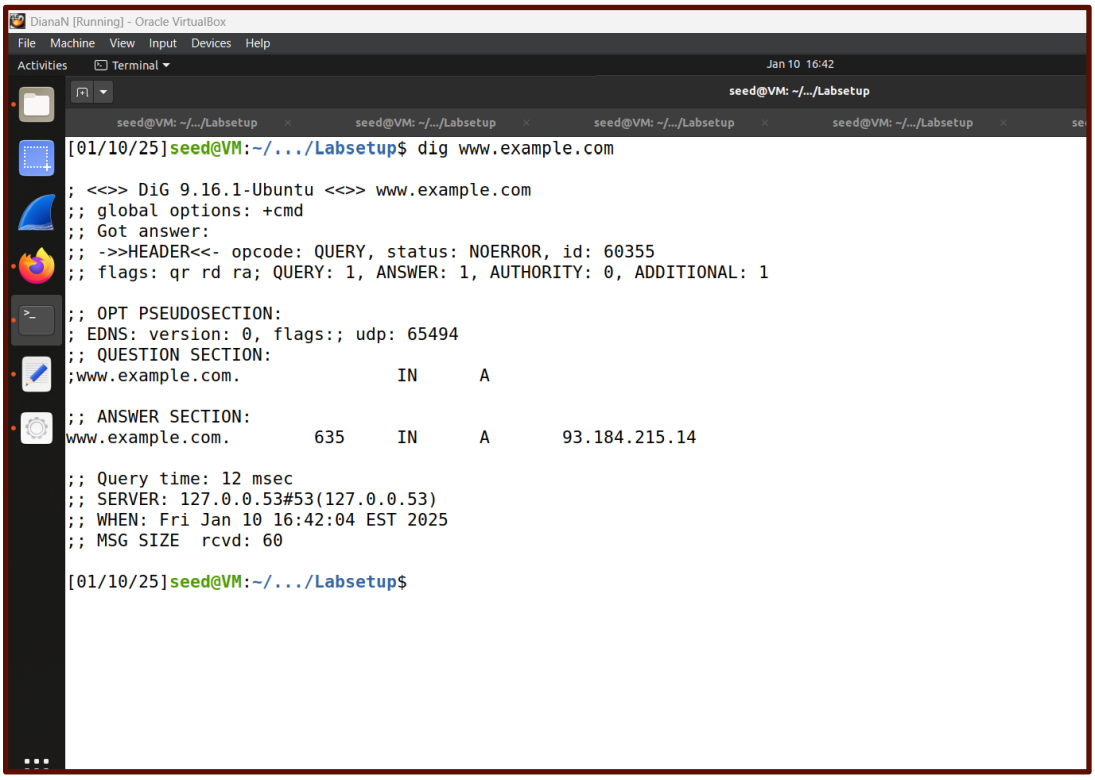

root@4e8df03bc705:/volumes# python3 handshake.py www.google.com
After making TCP connection. Press any key to continue ...
{'OCSP': ('http://o.pki.goog/wr2',),
'caIssuers': ('http://i.pki.goog/wr2.crt',),
'crlDistributionPoints': ('http://c.pki.goog/wr2/oQ6nyr8F0m0.crl',),
'issuer': ((('countryName', 'US'),),
(('organizationName', 'Google Trust Services'),),
(('commonName', 'WR2'),)),
'notAfter': 'Mar 3 08:38:05 2025 GMT',
'notBefore': 'Dec 9 08:38:06 2024 GMT',
'serialNumber': 'D01C59292557A18010EE18C306607692',
'subject': ((('commonName', 'www.google.com'),),),
'subjectAltName': (('DNS', 'www.google.com'),),
'version': 3}
After handshake. Press any key to continue ...
root@4e8df03bc705:/volumes#

Advantages of Custom CA Configuration

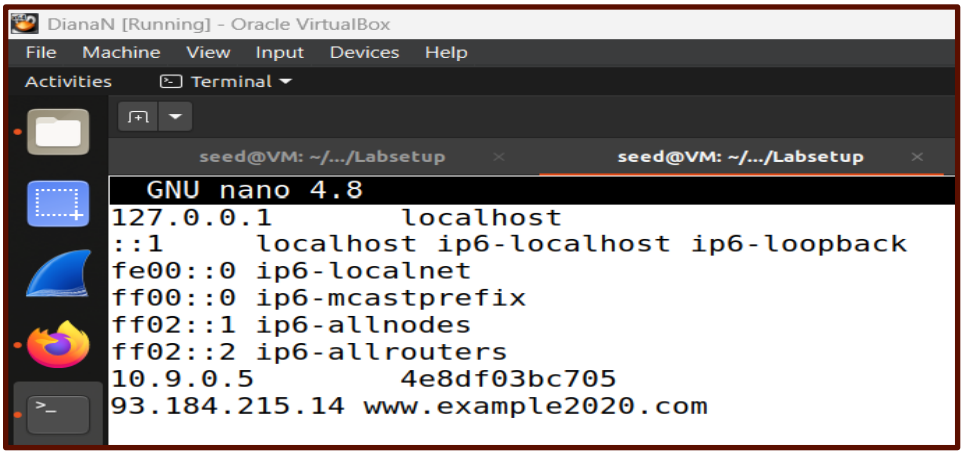
1. **Enhanced Security**:
    - Restricts trust to explicitly added CAs in the custom directory, reducing exposure to potentially untrusted CAs.
2. **Custom Validation**:
    - Suitable for environments where only specific CAs are permitted, such as private networks or specialized applications.
3. **Testing and Debugging**:
    - Enables isolated certificate validation, making it easier to troubleshoot and validate configurations in controlled scenarios.

Task 1.c: Experiment with the hostname check

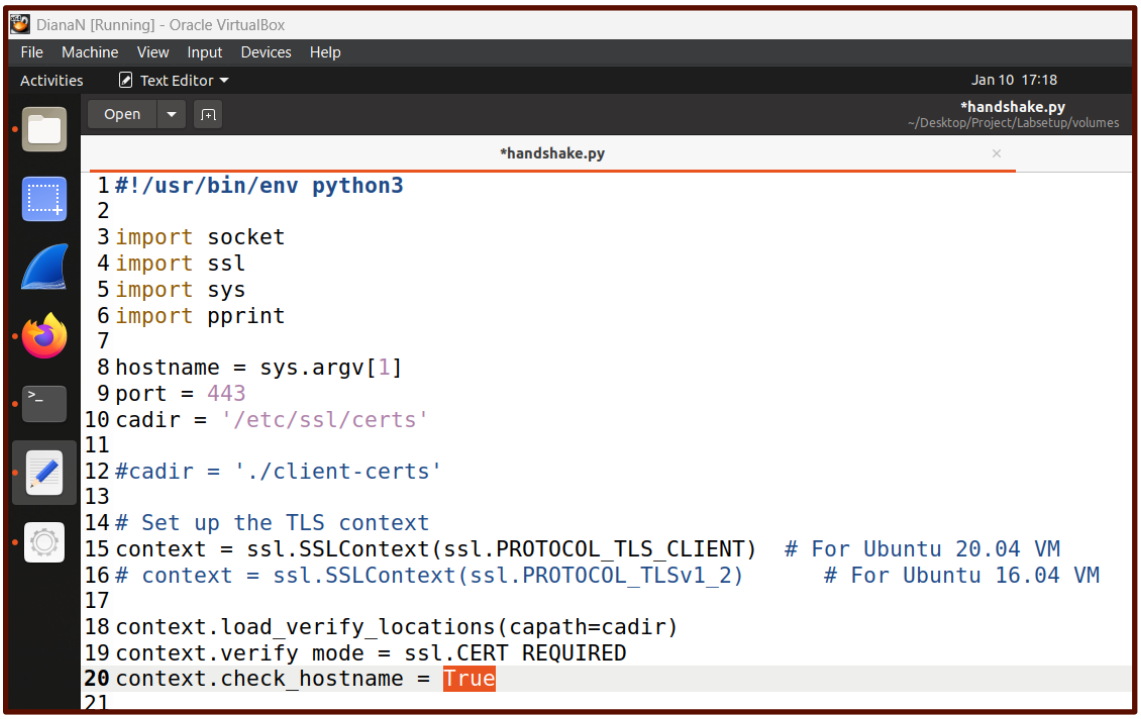
The dig command, work in this setup, and does not work in containers typically configured without extensive network utilities.

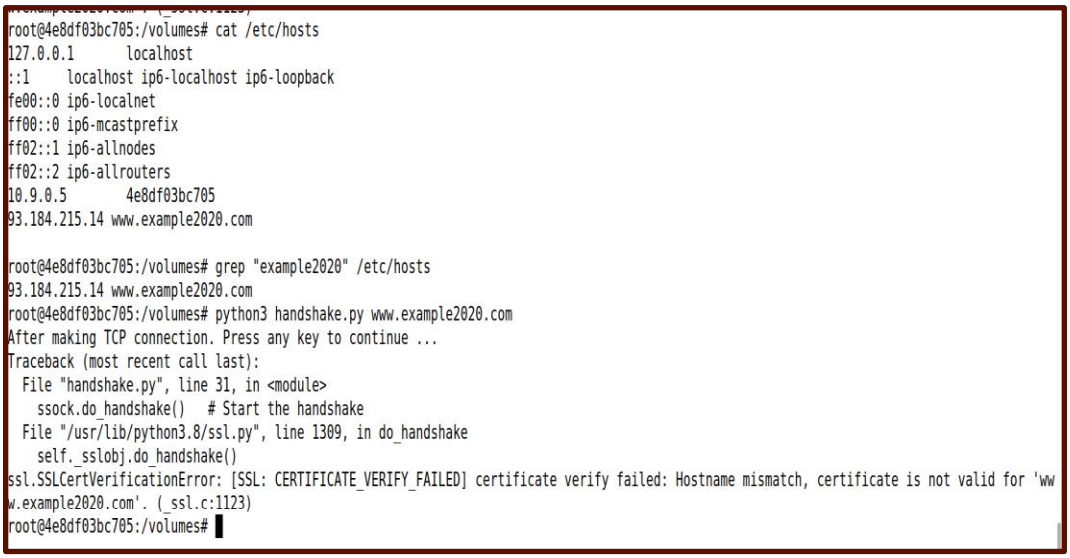


Modify the /etc/hosts file (inside the container), add the following entry at the end of the file (the IP address is what you get from the dig command). 93.184.216.34 www.example2020.com



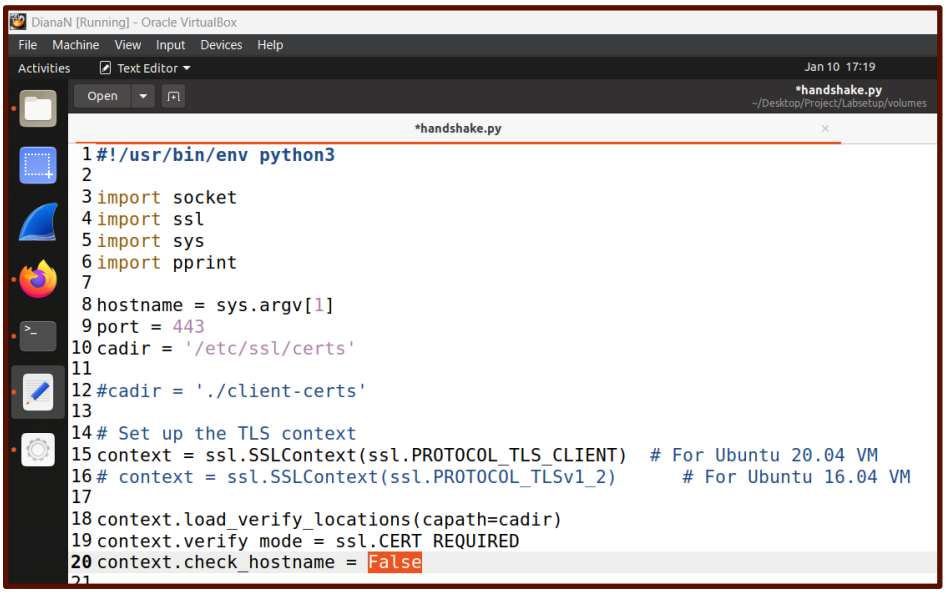
When we put the “check_hostname = True” i get this when i try to run www.example2020.com:

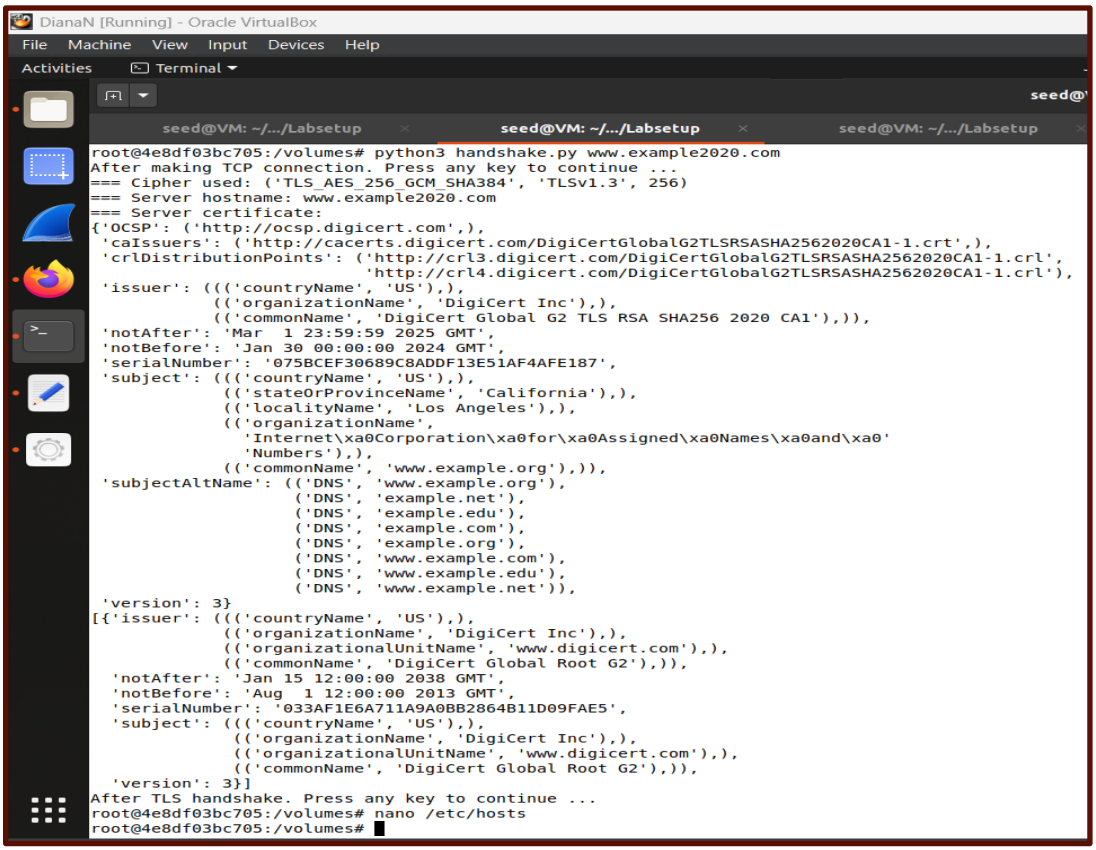




The error message indicates that the hostname validation failed. This occurred because the server's certificate was issued for www.example.com, but your client is attempting to connect to www.example2020.com.

When we put the “check_hostname = False” i get this when i try to run www.example2020.com:





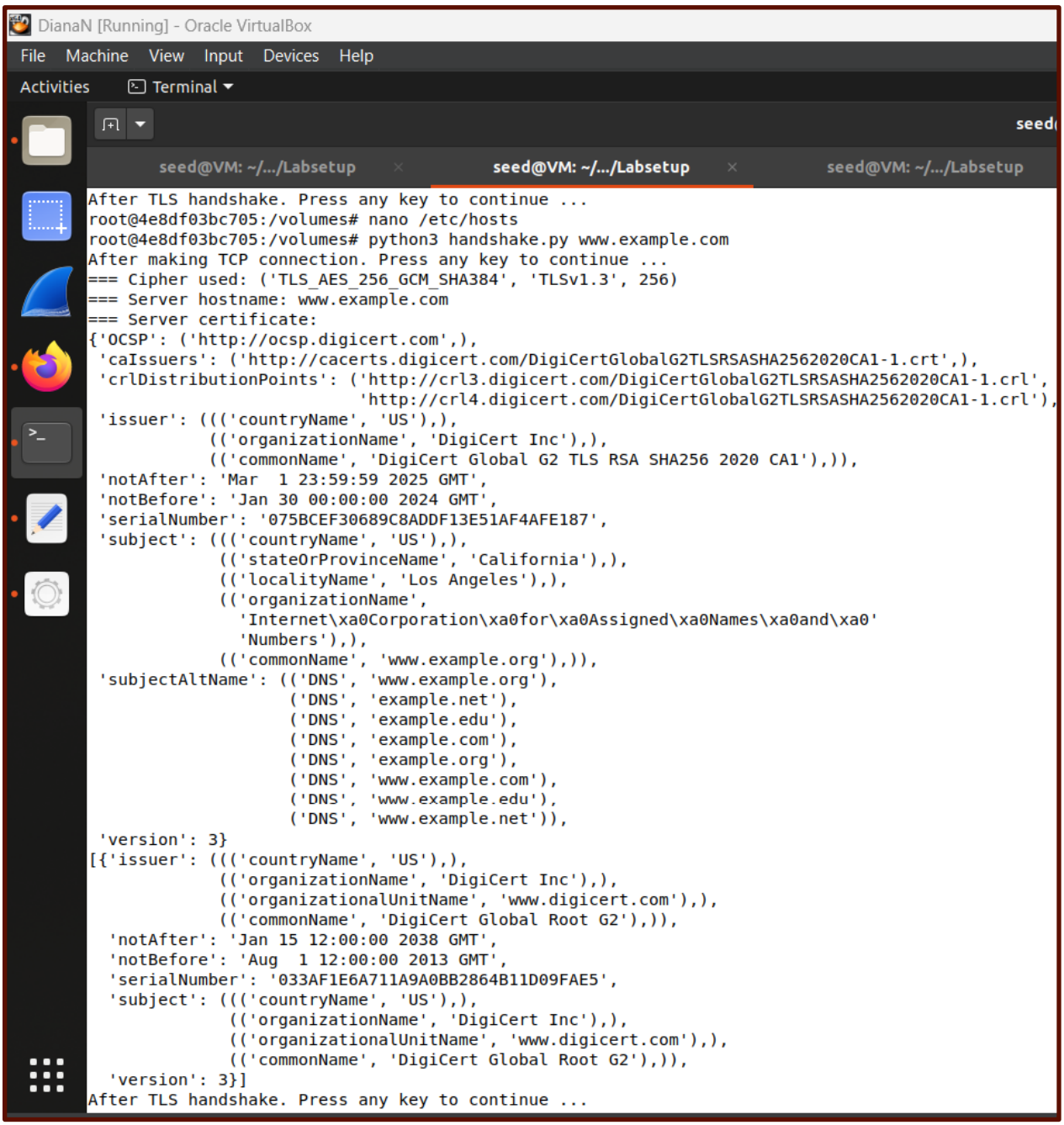
This time, the hostname check will be bypassed, and the connection should succeed.

When check_hostname = True (default setting), the client verifies that the hostname in the certificate matches the requested hostname (www.example2020.com). This is why the connection fails.
By setting check_hostname = False, the client skips the hostname verification step, allowing the connection to succeed. However, this introduces a security risk, as the client cannot confirm the server's identity.

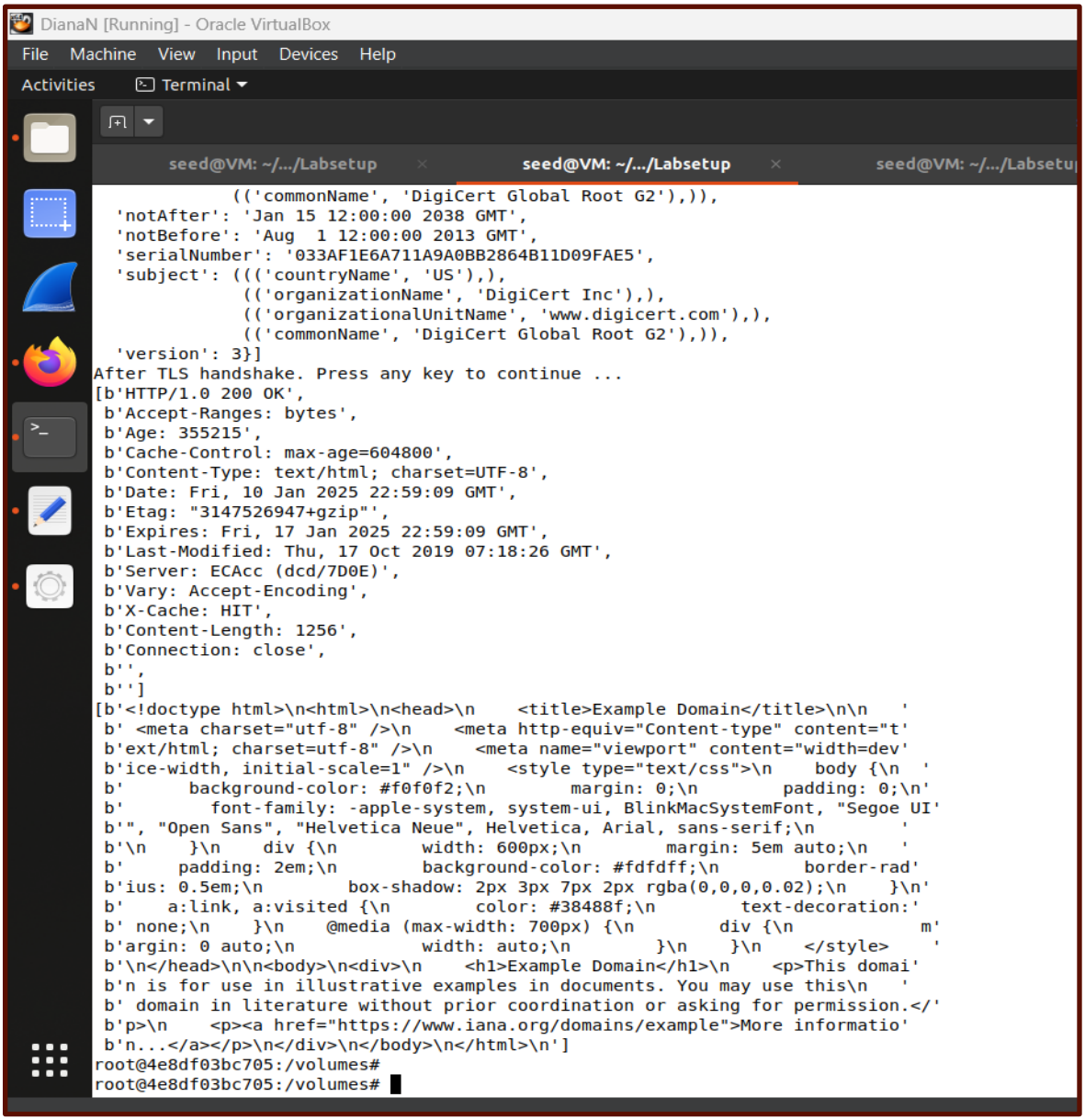
Checks of hostnames should be performed to ensure that the server it connects to is the one that was identified by its SSL/TLS certificate. This makes sure that it communicates with the right server and not some impersonator. If it does not perform host name checking, it introduces a rather critical security weakness whereby an attacker can set up malicious servers with valid but unrelated certificates, which they can use to conduct Man-in-the-Middle or spoofing of servers. It provides means for unauthorized access to sensitive information, theft of personal or financial data, or the compromise of communication security as a whole. Hence, checks on hostnames form one of those steps which become imperative to create trusted, secured connections over the Internet for protection of the users and their systems against specific kinds of cyber threats.

Task 1.d: Sending and getting Data

TLS handshake and server certificate are displayed. It confirms that the handshake was successful, specifying the cipher suite used (TLS_AES_256_GCM_SHA384) and providing server certificate details, including the issuer (DigiCert Inc.) and its validity period. This ensures that the server's identity is verified before data exchange occurs.

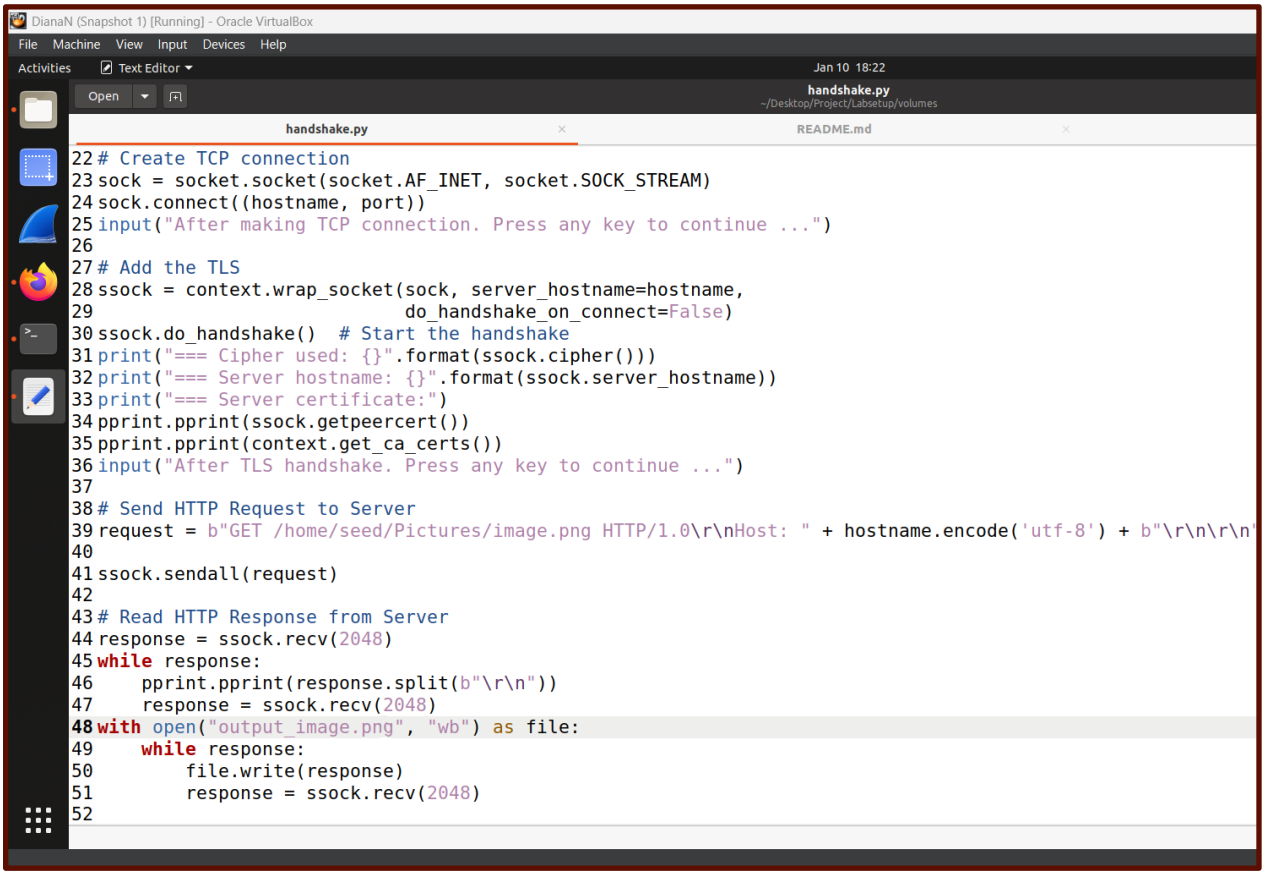


A successful HTTP GET request sent to the server would look something like the following, such as to www.example.com over a secure TLS connection. It would include the HTTP status code from the server, such as 200 OK, headers including Content-Type and Date, and the actual content of the response. This confirms that the client program is capable of establishing a secure TLS connection, sending a request, and receiving a response.

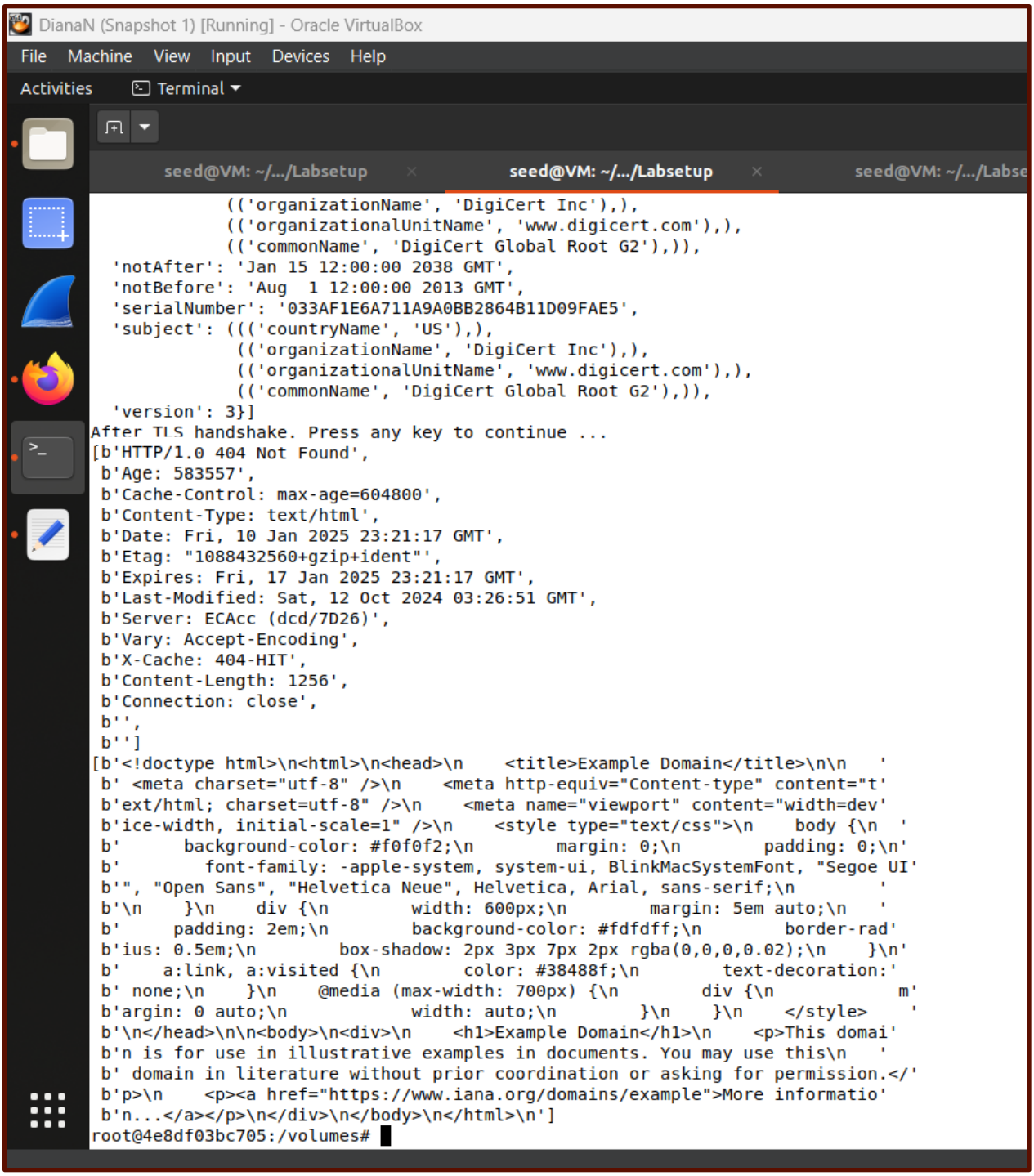


Now, when i put image request:

Also i add while loop to This saves the image data as output_image.jpg in the current directory

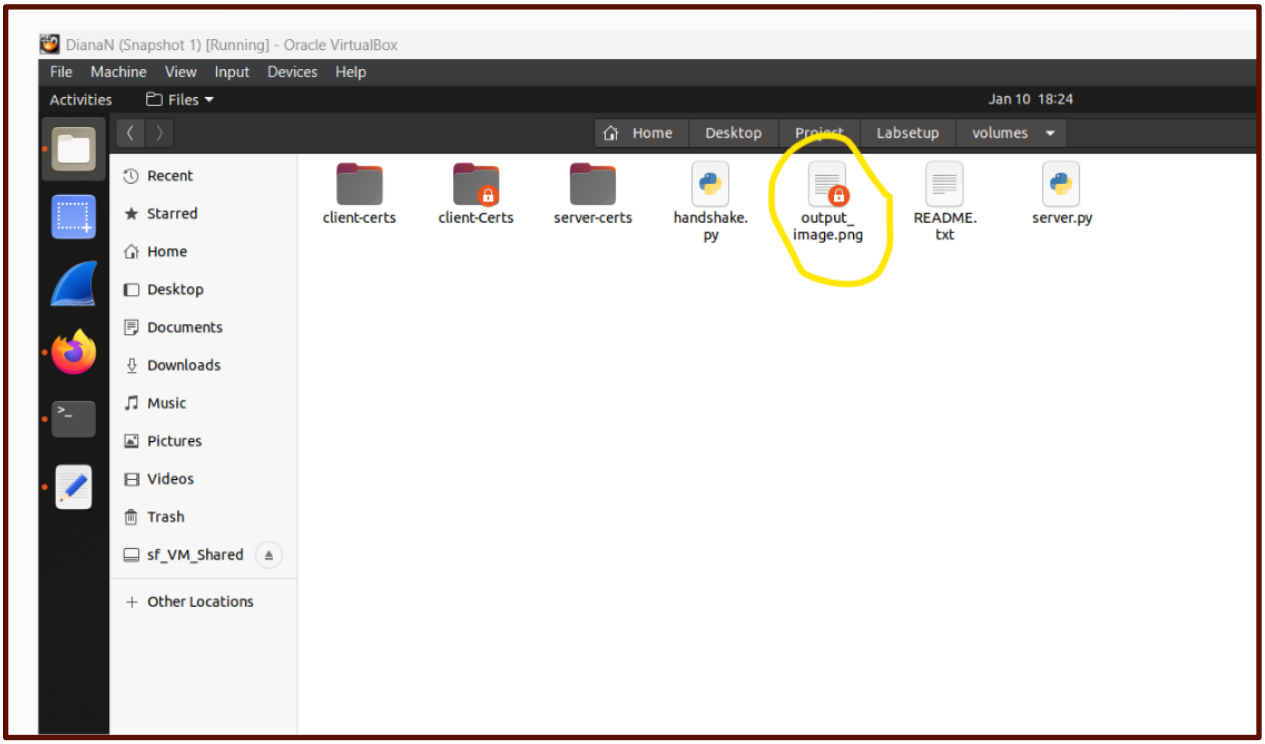


A successful HTTP GET request sent to the server would look something like the following, such as to www.example.com over a secure TLS connection. It would include the HTTP status code from the server, such as 200 OK, headers including Content-Type and Date, and the actual content of the response. This confirms that the client program is capable of establishing a secure TLS connection, sending a request, and receiving a response.


DianaN (Snapshot 1) [Running] - Oracle VirtualBox
File Machine View Input Devices Help
Activities Terminal
seed@VM: ~/.../Labsetup
seed@VM: ~/.../Labsetup
seed@VM: ~/.../Labse
(('organizationName', 'DigiCert Inc'),),
(('organizationalUnitName', 'www.digicert.com'),),
(('commonName', 'DigiCert Global Root G2'),)),
'notAfter': 'Jan 15 12:00:00 2038 GMT',
'notBefore': 'Aug 1 12:00:00 2013 GMT',
'serialNumber': '033AF1E6A711A9A0BB2864B11D09FAE5',
'subject': ((('countryName', 'US'),),
(('organizationName', 'DigiCert Inc'),),
(('organizationalUnitName', 'www.digicert.com'),),
(('commonName', 'DigiCert Global Root G2'),)),
'version': 3}]
After TLS handshake. Press any key to continue ...
[b'HTTP/1.0 404 Not Found',
b'Age: 583557',
b'Cache-Control: max-age=604800',
b'Content-Type: text/html',
b'Date: Fri, 10 Jan 2025 23:21:17 GMT',
b'Etag: "1088432560+gzip+ident"',
b'Expires: Fri, 17 Jan 2025 23:21:17 GMT',
b'Last-Modified: Sat, 12 Oct 2024 03:26:51 GMT',
b'Server: ECAcc (dcd/7D26)',
b'Vary: Accept-Encoding',
b'X-Cache: 404-HIT',
b'Content-Length: 1256',
b'Connection: close',
b'',
b'']
[b'<!doctype html>\n<html>\n<head>\n <title>Example Domain</title>\n\n '
b' <meta charset="utf-8" />\n <meta http-equiv="Content-type" content="t'
b'ext/html; charset=utf-8" />\n <meta name="viewport" content="width=dev'
b'ice-width, initial-scale=1" />\n <style type="text/css">\n body {\n '
b' background-color: #f0f0f2;\n margin: 0;\n padding: 0;\n'
b' font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI'
b'", "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;\n '
b'\n }\n div {\n width: 600px;\n margin: 5em auto;\n '
b' padding: 2em;\n background-color: #fdfdff;\n border-rad'
b'ius: 0.5em;\n box-shadow: 2px 3px 7px 2px rgba(0,0,0,0.02);\n }\n'
b' a:link, a:visited {\n color: #38488f;\n text-decoration:'
b' none;\n }\n @media (max-width: 700px) {\n div {\n m'
b'argin: 0 auto;\n width: auto;\n }\n }\n </style> '
b'\n</head>\n\n<body>\n \n <h1>Example Domain</h1>\n This domai'
b'n is for use in illustrative examples in documents. You may use this\n '
b' domain in literature without prior coordination or asking for permission.</'
b'p>\n More informatio'
b'n... \n \n</body>\n</html>\n']
root@4e8df03bc705:/volumes#

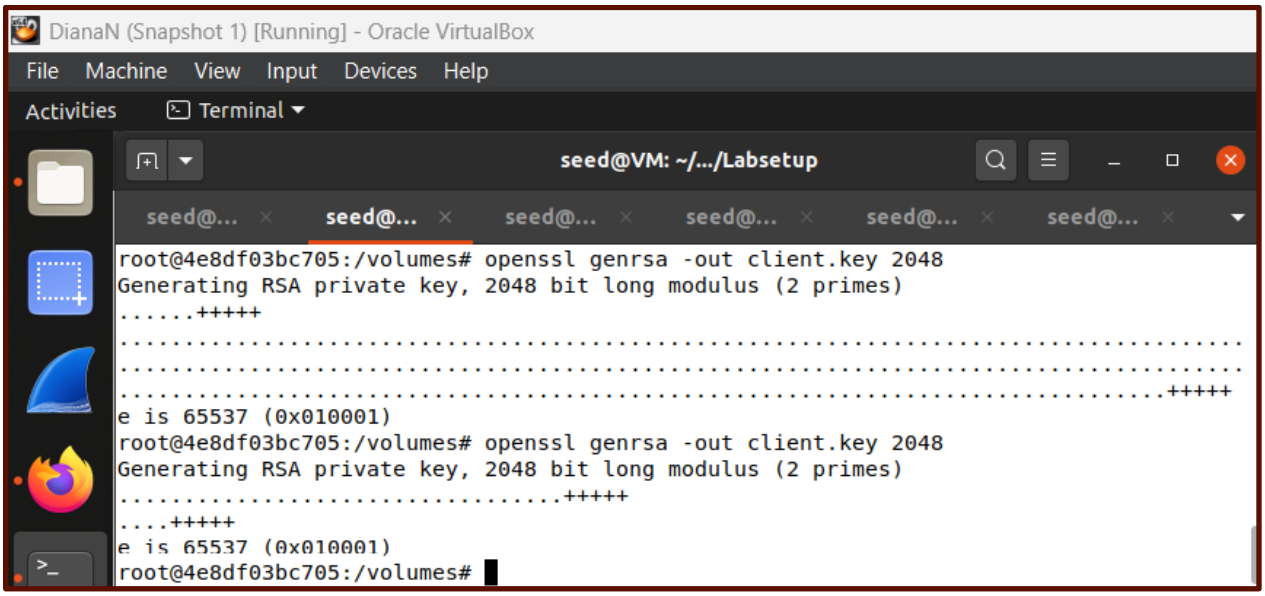
And to be sure that it work:

We see that the image upload here after run the command , and that is mean it work successfully!



Task2:TLS Server.

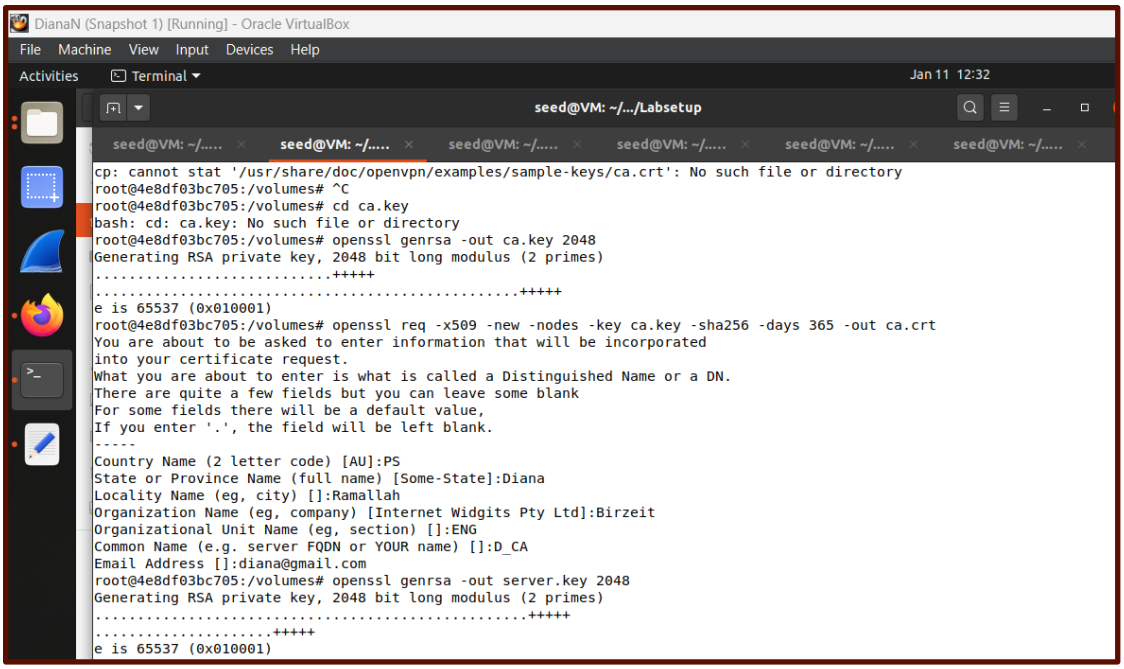
This command creates the private key client.key for the client.



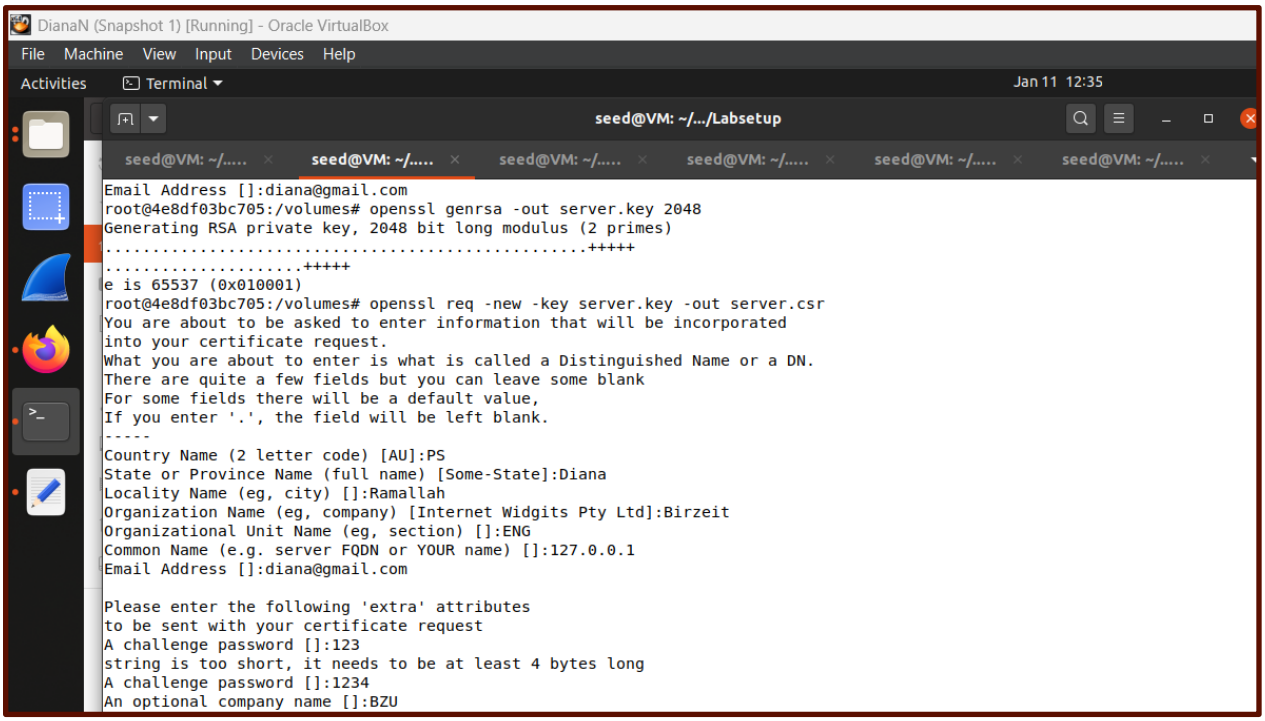
then , to generate a Certificate Signing Request (CSR).

The first step: **Generate a CA Certificate (ca.crt):**

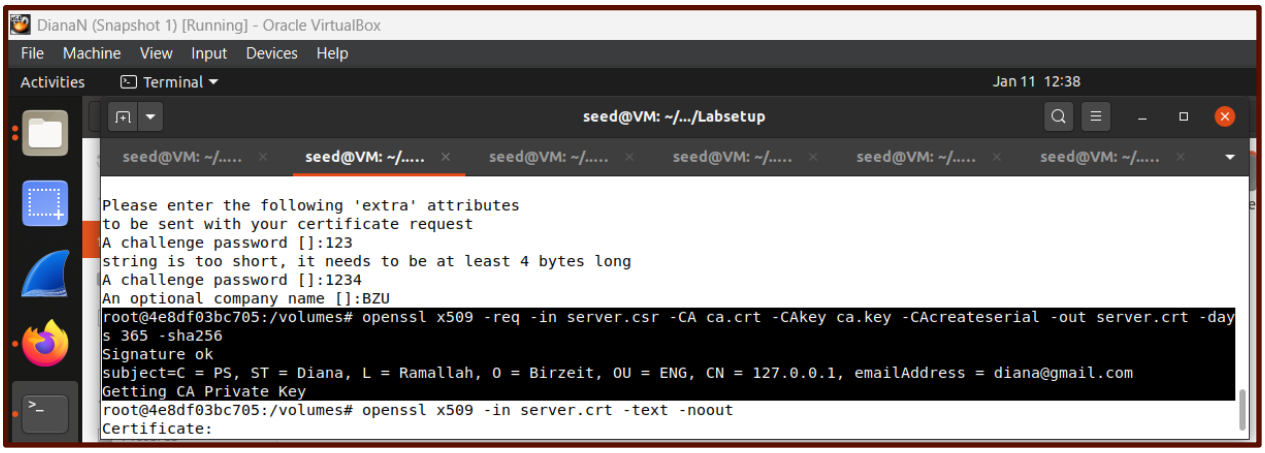
Generate a CA certificate (self-signed) using the private key and fill the info.



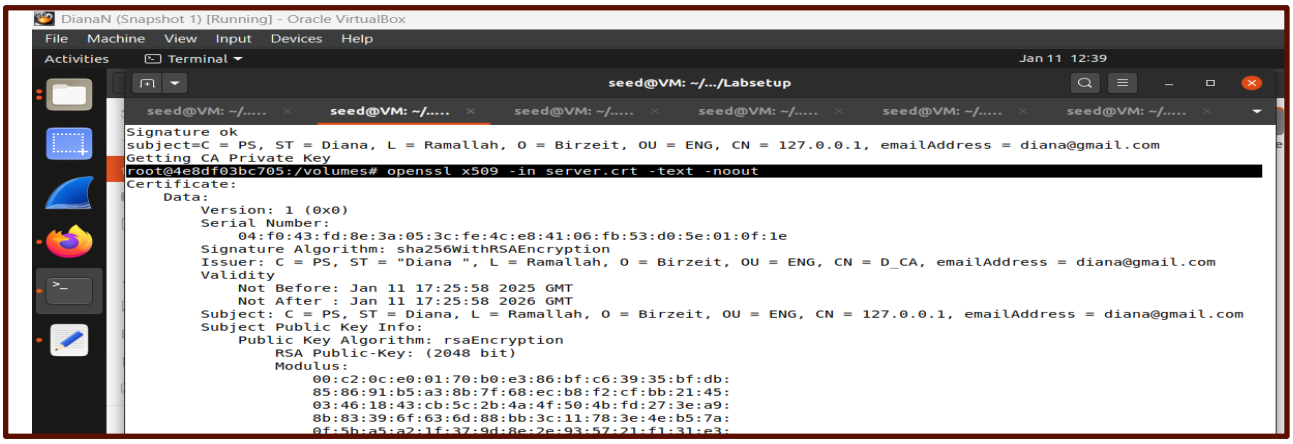
2nd step: **Generate a Server Certificate (server.crt):**

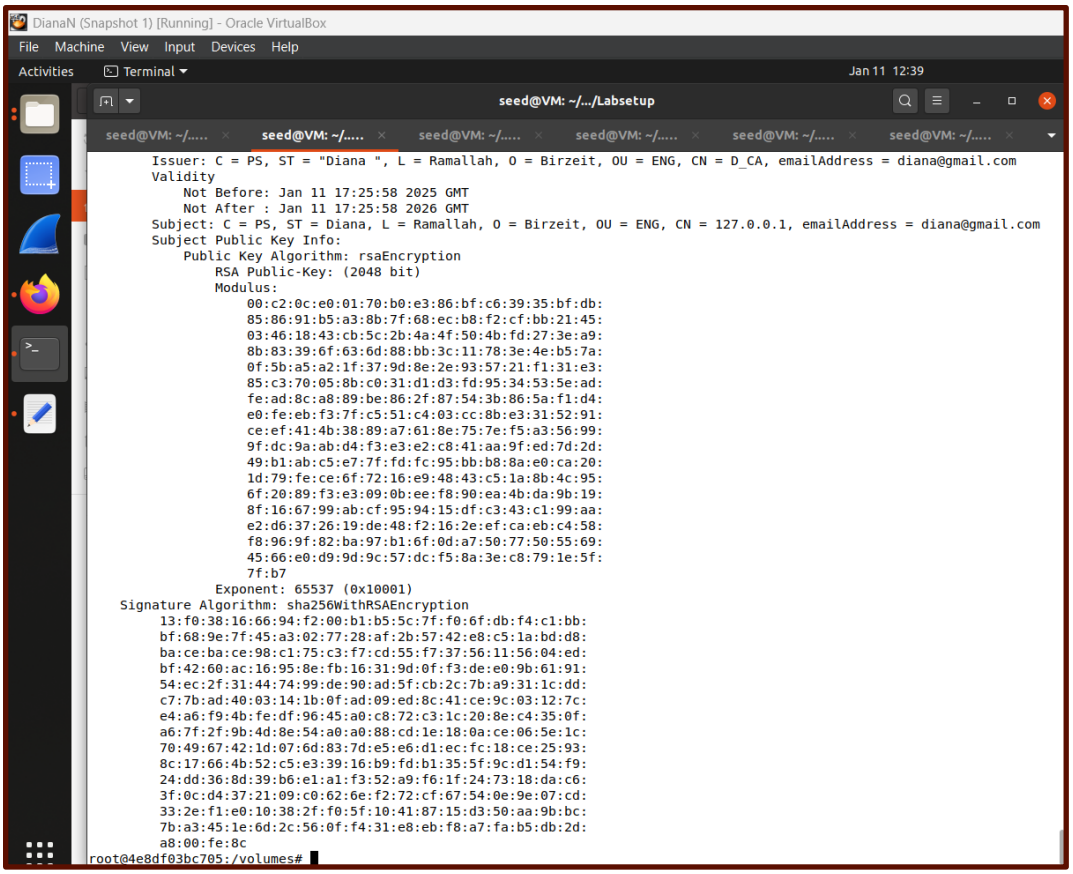


3rd step: **Sign the CSR with the CA to generate the server certificate:**

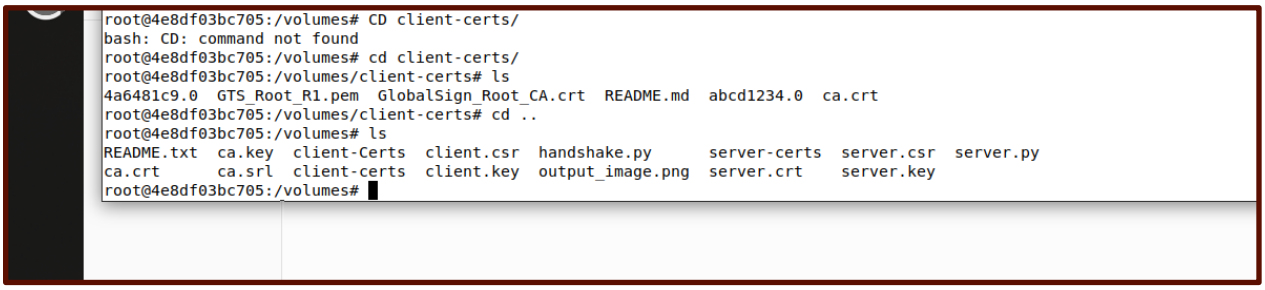


4th step: **Verify the server certificate**.



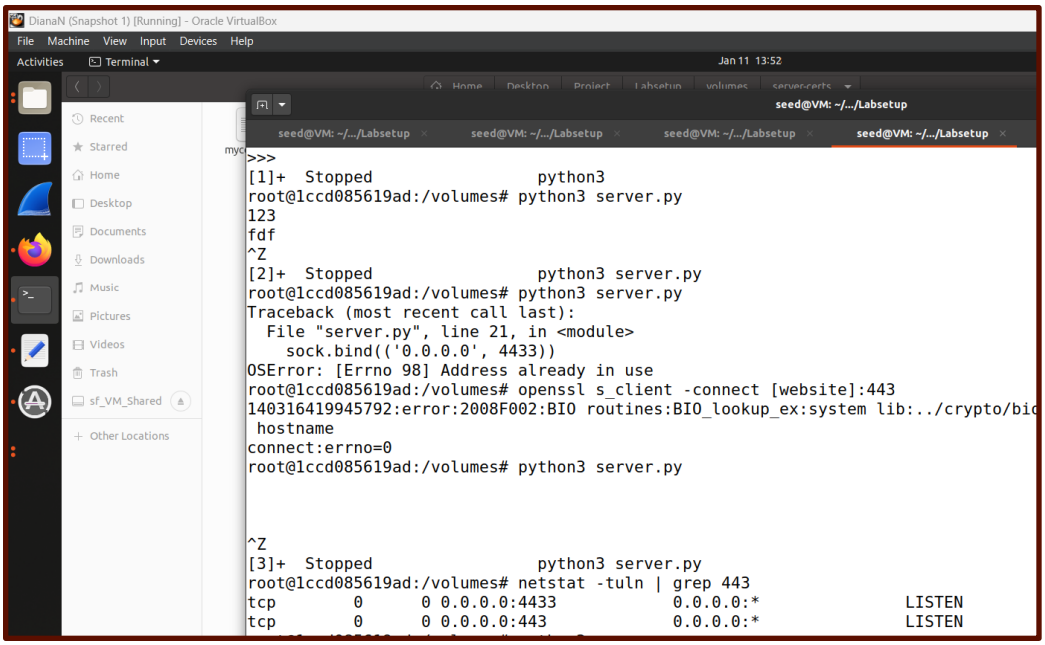


To Check if the files ca.crt, server.crt, and server.key exist:

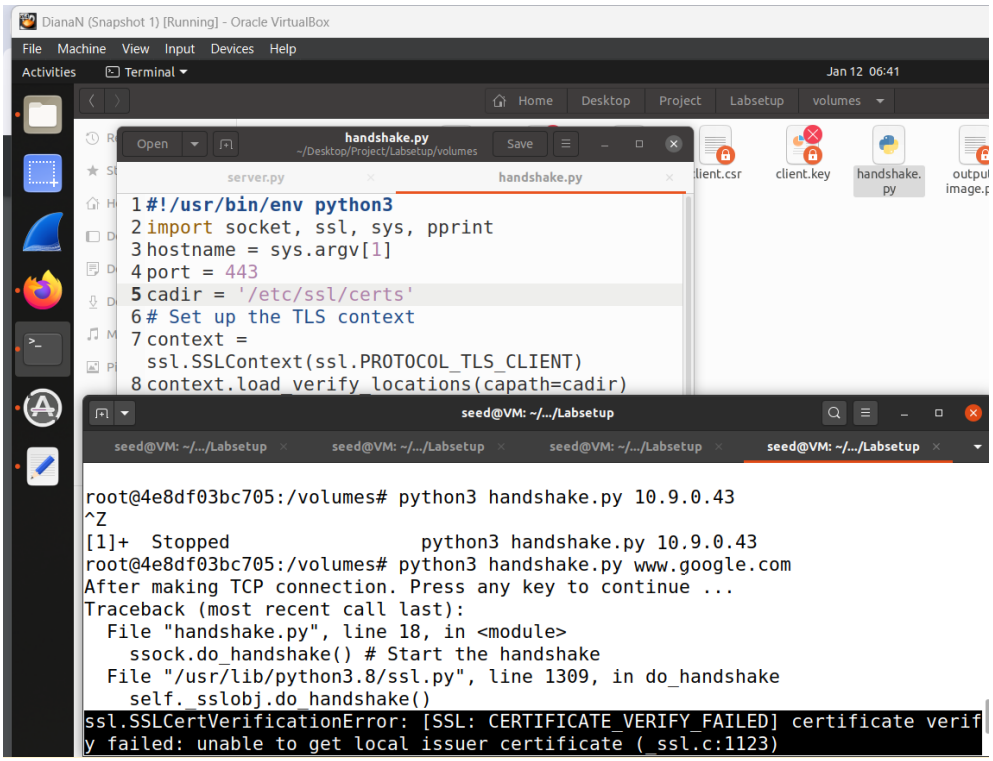


Task 2.a. Implement a simple TLS server

First, I modified the server.py by the new code that it uploaded in the TLS lab.



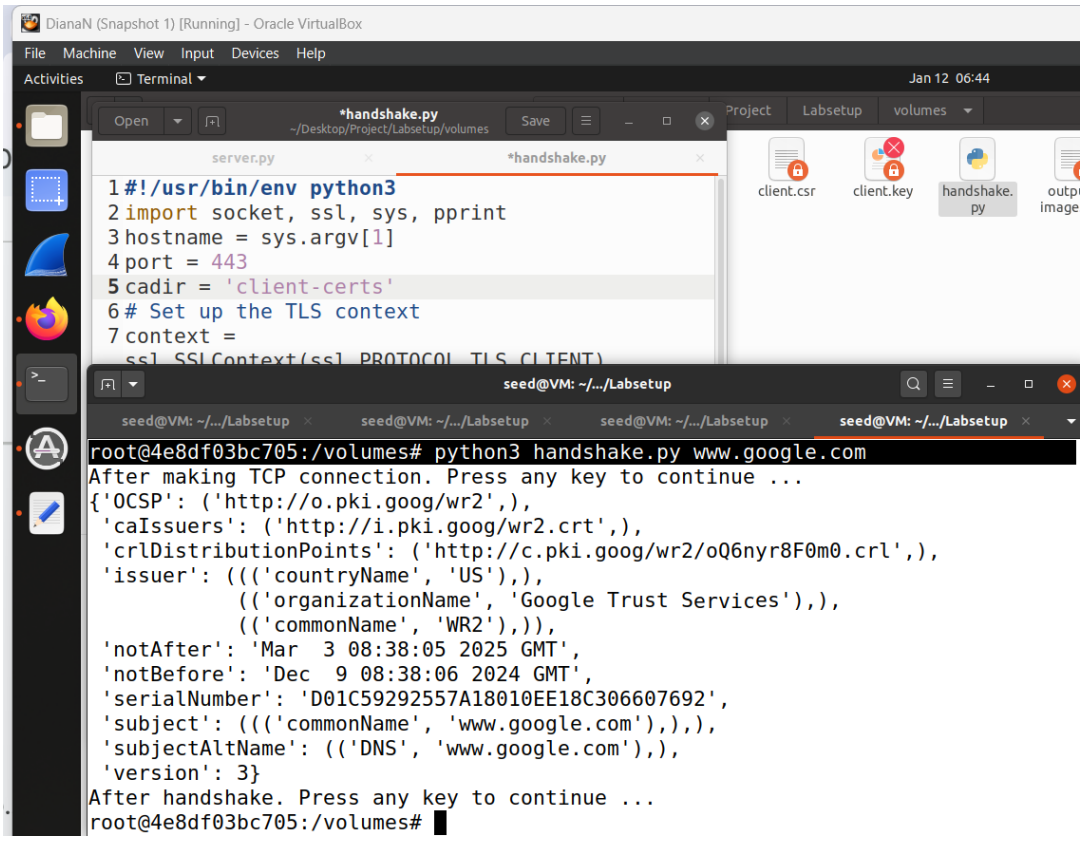
To test this process; when the cadir => from etc/ssl/certs



When testing with the **system-wide certificates directory** (/etc/ssl/certs), the client failed to verify the server's certificate if the CA that signed the server's certificate was not included in this directory. This led to a **"certificate verify failed"** error unless the custom CA certificate was

manually added to the system's trusted store. However, adding a custom CA certificate to the system-wide store is not recommended for testing purposes, as it impacts all applications relying on the system's certificates, potentially introducing broader security risks or unintended consequences.

To test this process; when the cadir => from client-certs

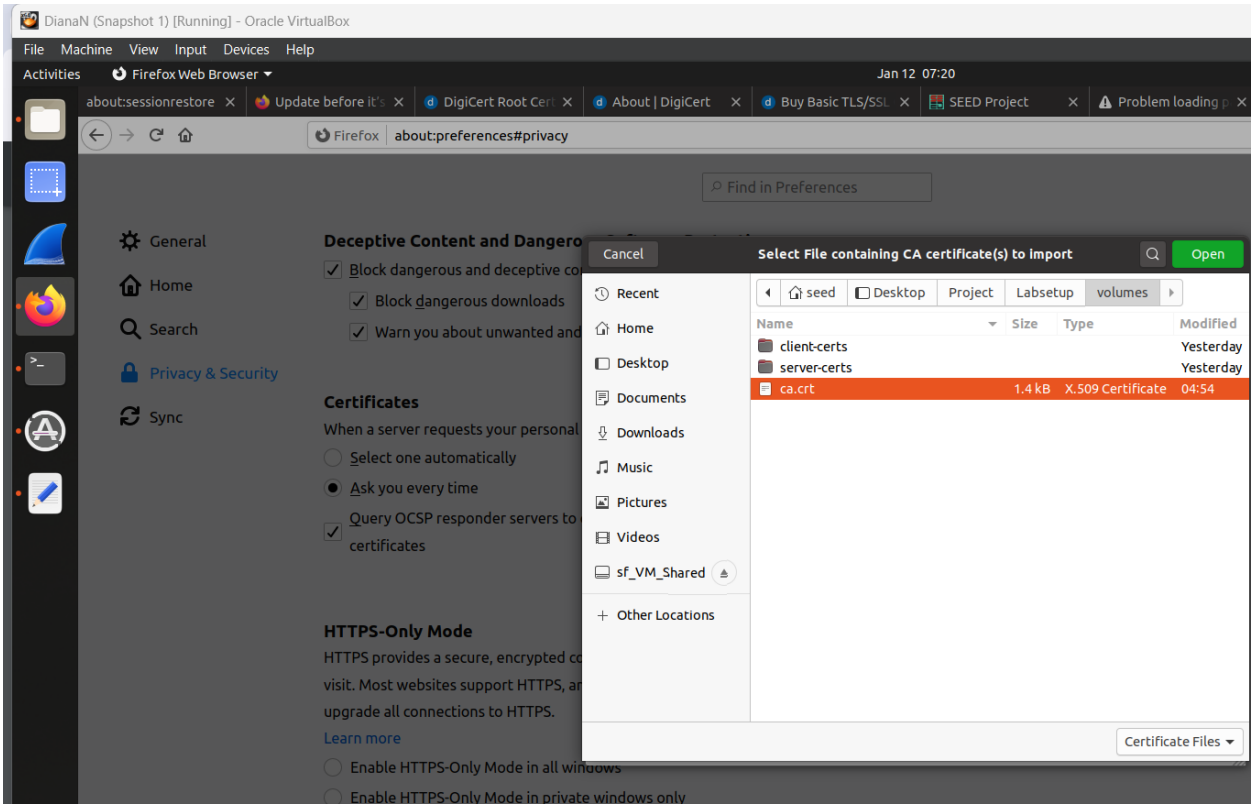


The image demonstrates testing a TLS handshake with www.google.com using a Python script that loads CA certificates from a custom directory (client-certs). The script successfully establishes the connection, validates the certificate issued by Google Trust Services, and prints details such as the certificate's validity (December 9, 2024, to March 3, 2025) and subject (www.google.com). This approach showcases the use of a custom CA directory for enhanced security and control over certificate validation.

Task 2.b. Import the CA Certificate

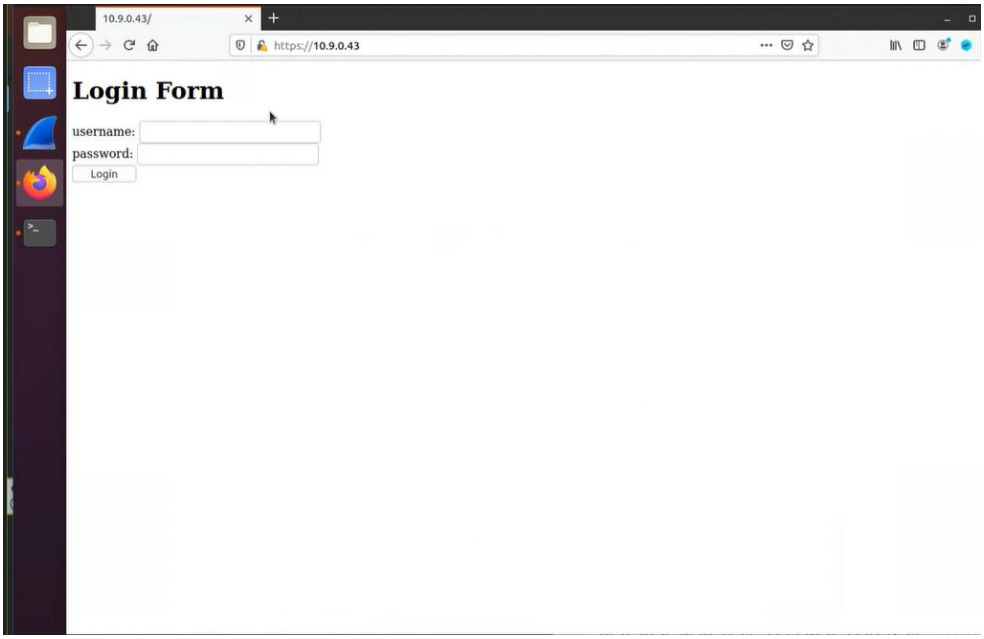
After server connection setup in port 443

1. In the Certificate Manager window,I go to the **Authorities** tab.
2. Navigate to the location where your ca.crt file is saved, Select the ca.crt file.
3. Imported it in this file.



After importing, CA appears in the list of certificate authorities in Firefox. This ensures that Firefox will trust certificates signed by this CA.

Now test it by Access theTLS Server: POST()



Task 2.c. Certificate with multiple names:

Create san_config.cnf file and add to it:

[req]

prompt = no

distinguished_name = req_distinguished_name

req_extensions = req_ext

SEED Labs 8

[req_distinguished_name]

C = US

ST = New York

L = Syracuse

O = XYZ LTD.

CN = www.bank32.com

[req_ext]

subjectAltName = @alt_names

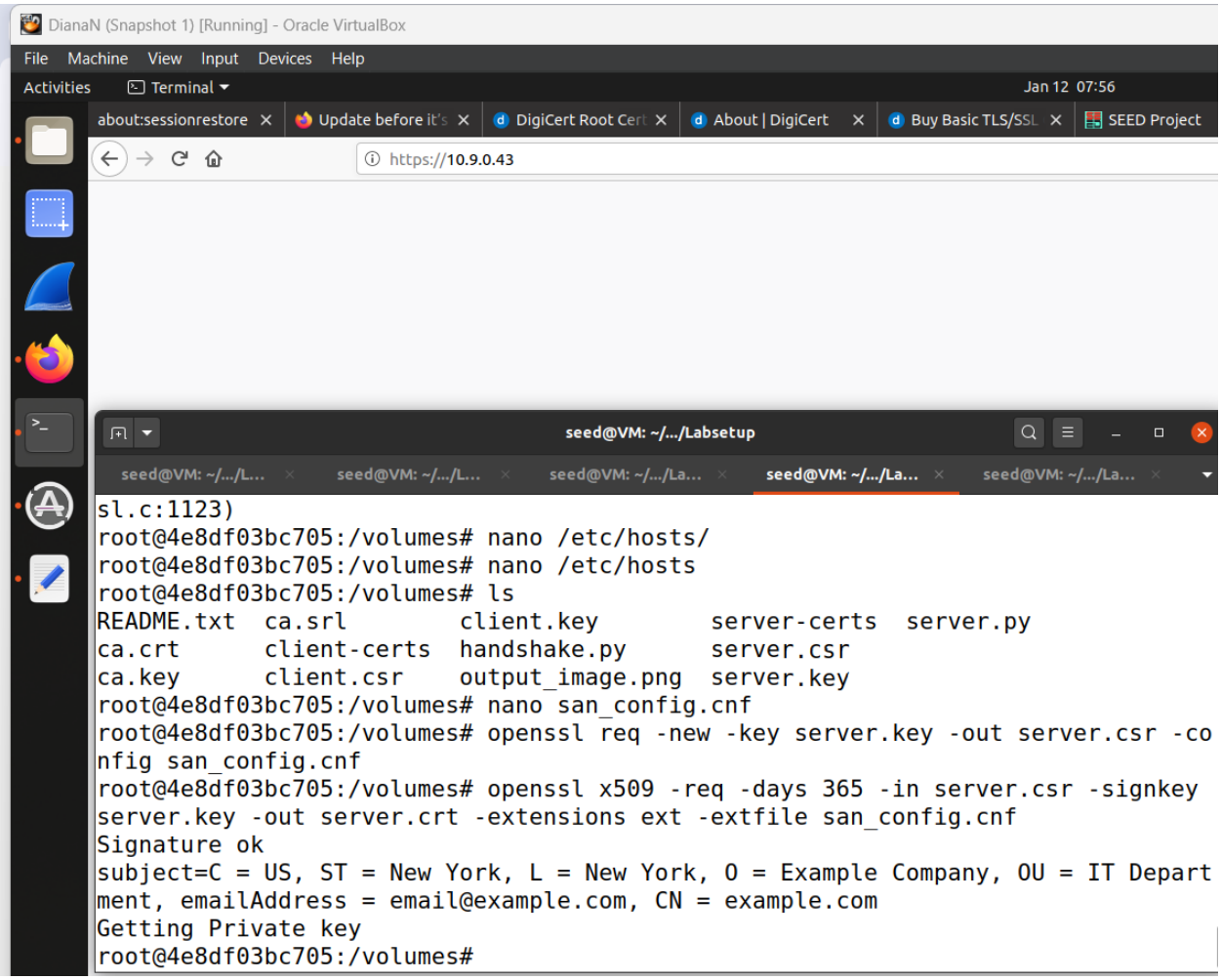
[alt_names]

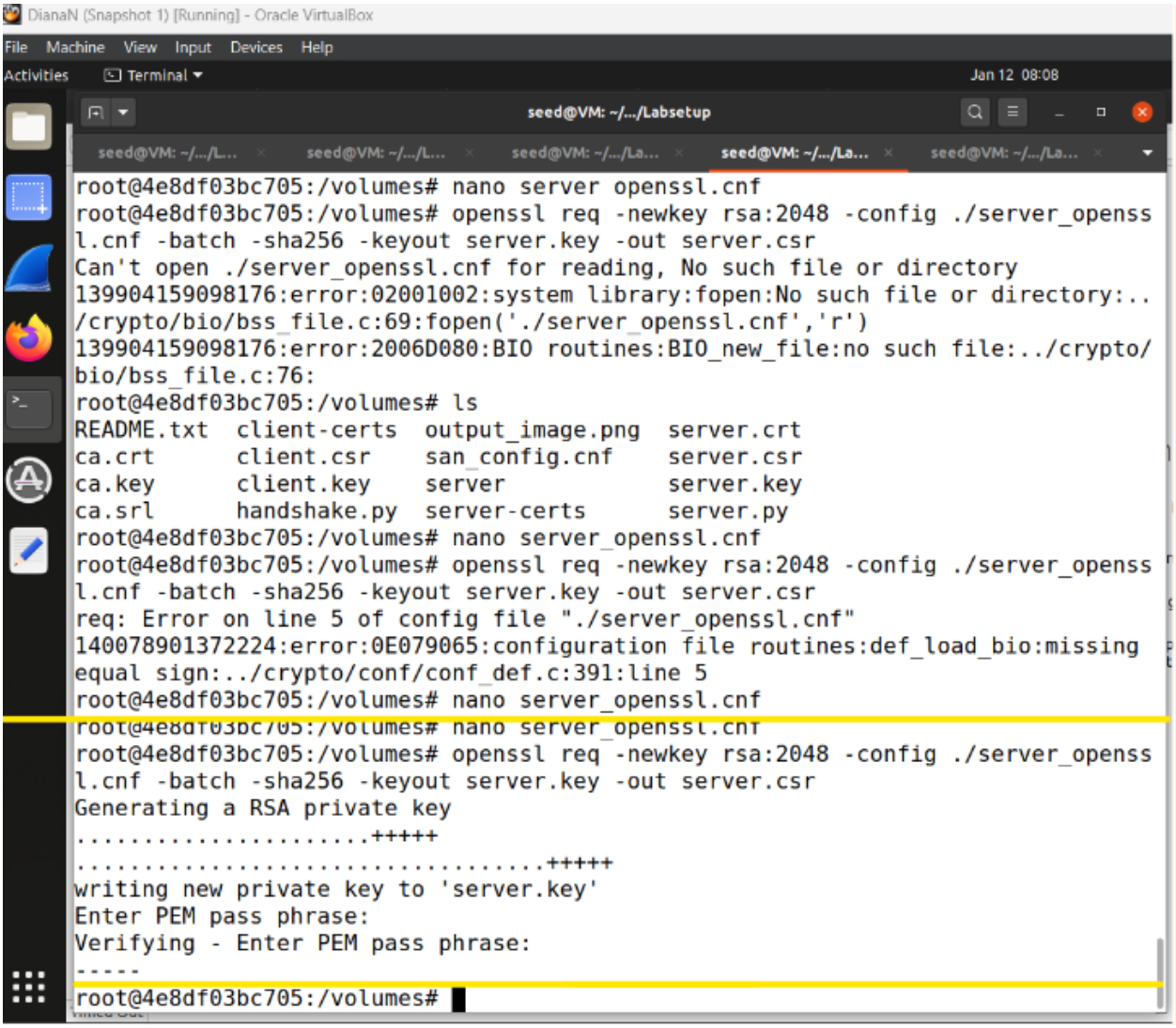
DNS.1 = www.bank32.com

DNS.2 = www.example.com

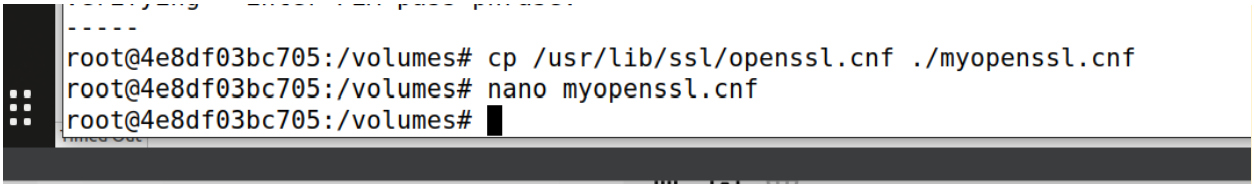
DNS.3 = *.bank32.com

Generate a New CSR and Certificate with SAN:


DianaN (Snapshot 1) [Running] - Oracle VirtualBox
File Machine View Input Devices Help
Activities Terminal Jan 12 07:56
about:sessionrestore
Update before it's
DigiCert Root Cert
About | DigiCert
Buy Basic TLS/SSL
SEED Project
https://10.9.0.43
seed@VM: ~/.../Labsetup
seed@VM: ~/.../L...
seed@VM: ~/.../L...
seed@VM: ~/.../La...
seed@VM: ~/.../La...
seed@VM: ~/.../La...
sl.c:1123)
root@4e8df03bc705:/volumes# nano /etc/hosts/
root@4e8df03bc705:/volumes# nano /etc/hosts
root@4e8df03bc705:/volumes# ls
README.txt ca.srl client.key server-certs server.py
ca.crt client-certs handshake.py server.csr
ca.key client.csr output_image.png server.key
root@4e8df03bc705:/volumes# nano san_config.cnf
root@4e8df03bc705:/volumes# openssl req -new -key server.key -out server.csr -co
nfig san_config.cnf
root@4e8df03bc705:/volumes# openssl x509 -req -days 365 -in server.csr -signkey
server.key -out server.crt -extensions ext -extfile san_config.cnf
Signature ok
subject=C = US, ST = New York, L = New York, O = Example Company, OU = IT Depart
ment, emailAddress = email@example.com, CN = example.com
Getting Private key
root@4e8df03bc705:/volumes#

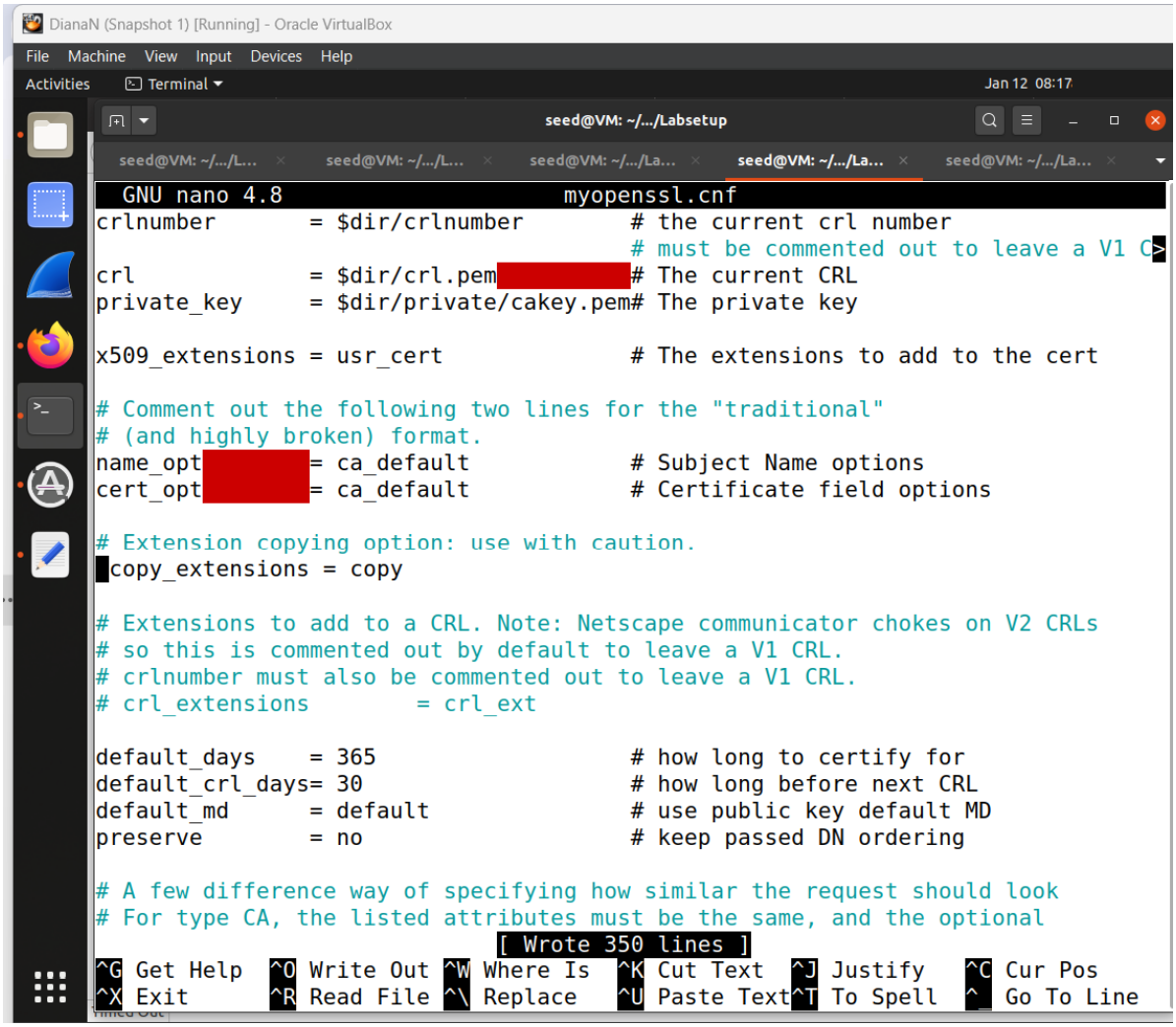


During the setup of SSL/TLS security for our server, we encountered an issue with the OpenSSL configuration file (server_openssl.cnf), which could not initially be accessed due to an incorrect file path or its absence. After locating the file and confirming its contents using the nano text editor, we successfully resolved the issue. We then executed the OpenSSL command to generate a 2048-bit RSA private key and a certificate signing request (CSR), securing the private key with a passphrase for added security. The CSR was created using the SHA-256 hashing algorithm to ensure strong cryptographic security. Resolving the file issue and completing the key and CSR generation were vital steps in configuring secure server communications.



In my terminal session, I copied the default OpenSSL configuration file from its system directory (/usr/lib/ssl/openssl.cnf) to my current working directory, renaming it to myopenssl.cnf. This allowed me to safely modify OpenSSL settings without affecting the global configuration, thus preventing potential disruptions to other system applications. Using Nano, a command-line text editor, I opened the renamed configuration file for customization, such as enabling specific certificate extensions essential for my SSL/TLS setup. This localized approach ensures that the changes are isolated and tailored to meet the specific requirements of my SSL configuration.

Modified ti by removing comment sign before copy_extention:



Generate the Certificate Using the Modified Configuration:

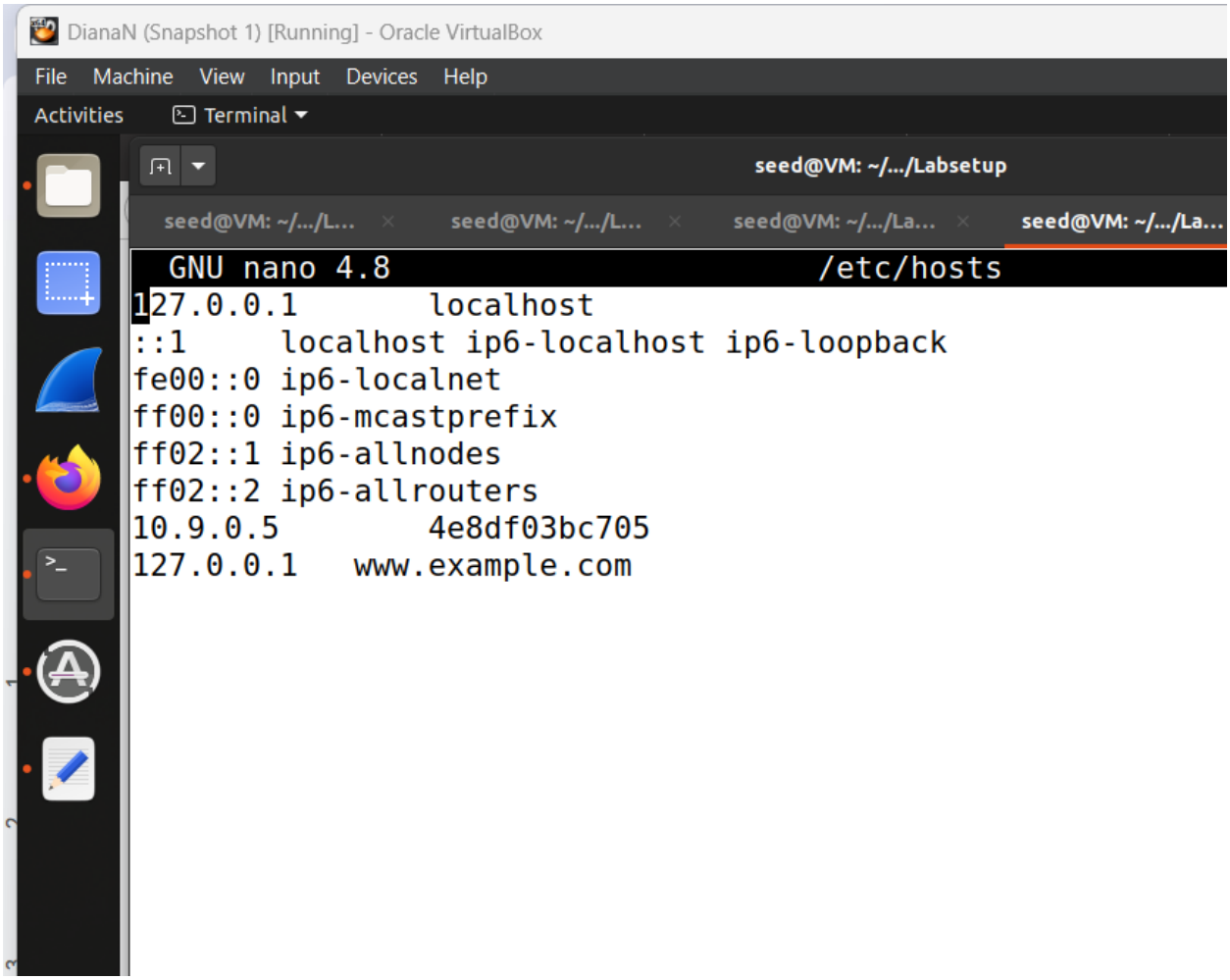
In my terminal session, I used the OpenSSL ca command to sign a Certificate Signing Request (CSR) with my Certificate Authority (CA) certificate to generate a server certificate (server.crt) valid for 3650 days, utilizing the SHA-256 hashing algorithm. The process referenced a custom configuration file, ./myopenssl.cnf. During the signing, OpenSSL validated the CSR's signature for integrity but flagged a mismatch in the countryName field between the CA certificate (PS) and the CSR (US). While this mismatch did not block the signing process, it could potentially affect the certificate's trust level in environments requiring strict validation. Repeated execution of the command confirmed a consistent issue with the country code discrepancy, emphasizing the need to align the country details in both the CSR and CA certificate to maintain consistency and avoid potential trust issues in the future.

```
root@4e8df03bc705:/volumes# echo '1000' > ./demoCA/serial
root@4e8df03bc705:/volumes# openssl ca -md sha256 -days 3650 -config ./myopenssl
.cnf -batch \
> -in server.csr -out server.crt \
> -cert ca.crt -keyfile ca.key
Using configuration from ./myopenssl.cnf
Check that the request matches the signature
Signature ok
The countryName field is different between
CA certificate (ps) and the request (US)
root@4e8df03bc705:/volumes# openssl ca -md sha256 -days 3650 -config ./myopenssl
.cnf -batch \
> -in server.csr -out server.crt \
> -cert ca.crt -keyfile ca.key
Using configuration from ./myopenssl.cnf
Check that the request matches the signature
Signature ok
The countryName field is different between
CA certificate (ps) and the request (US)
root@4e8df03bc705:/volumes# openssl ca -md sha256 -days 3650 -config ./myopenssl
.cnf -batch \
> -in server.csr -out server.crt \
> -cert ca.crt -keyfile ca.key
```

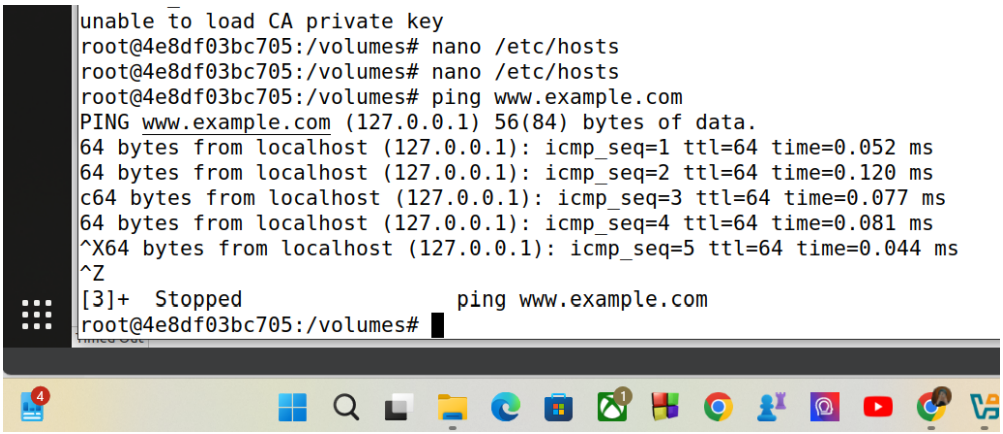
TASK3:A Simple HTTPS Proxy

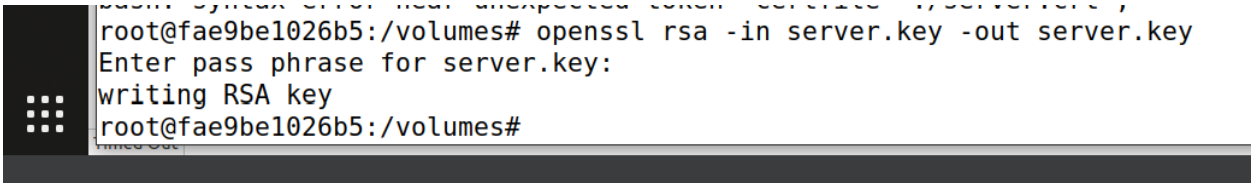
Modified the nano/hosts:

Map the domains that I want to intercept to our desired IP addresses. For testing a proxy or a MITM attack, I usually map these domains to your local machine's IP address (127.0.0.1).

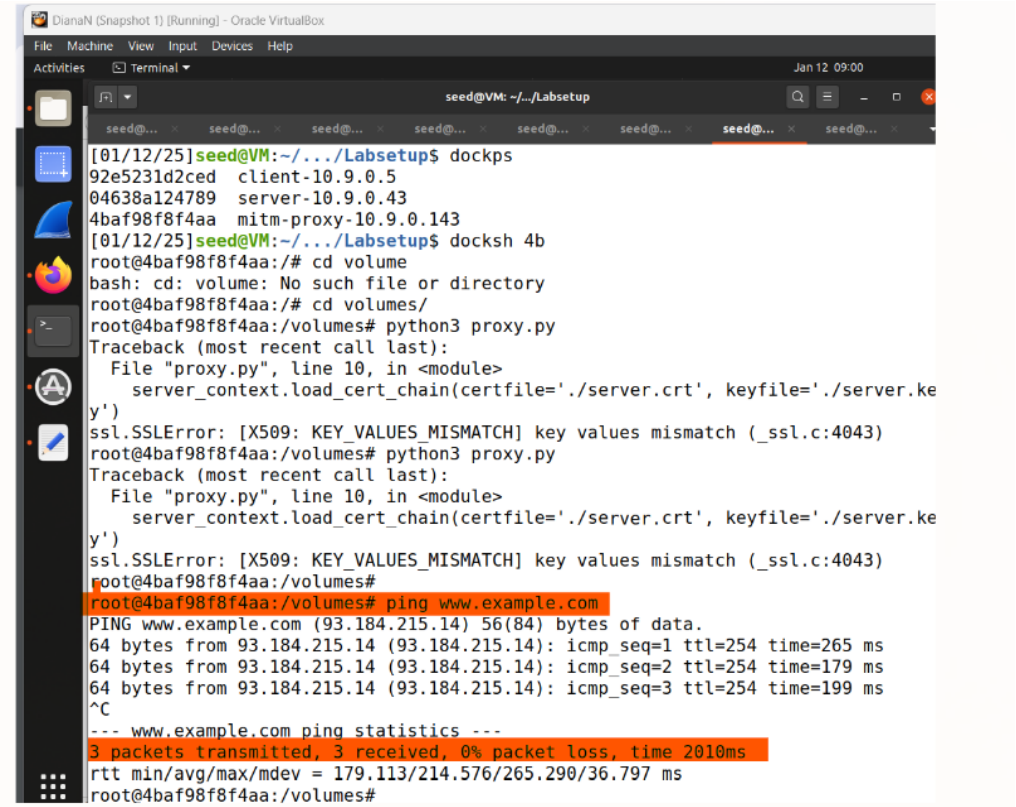


The i test it bu using ping:

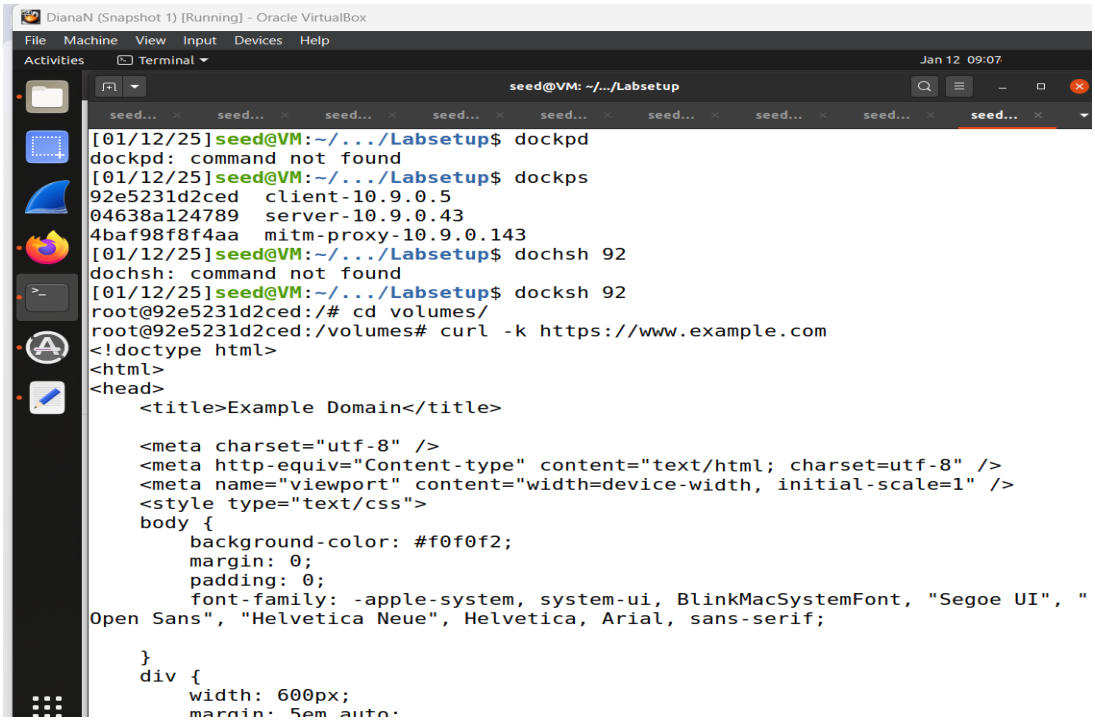




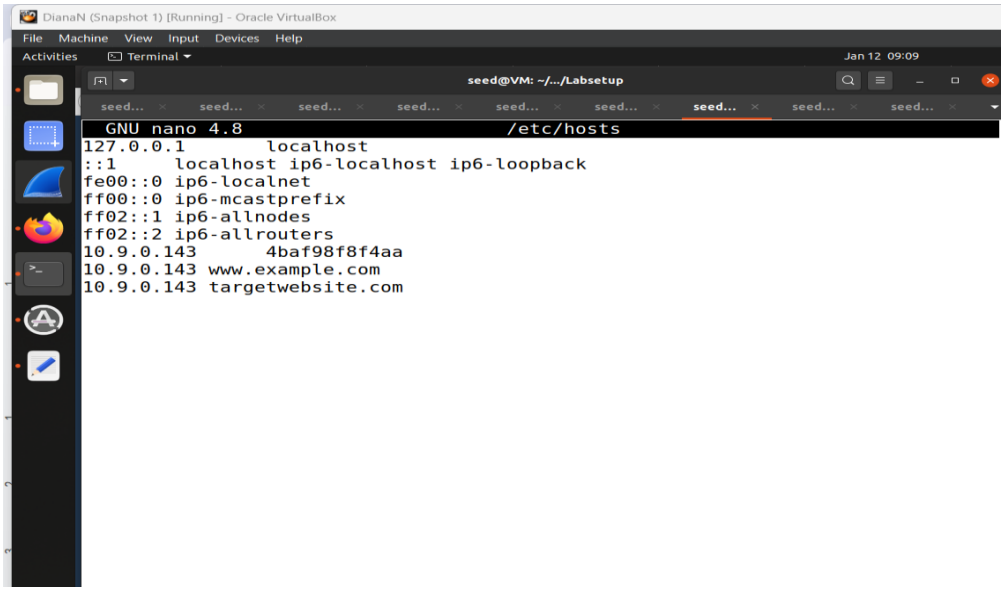
Add nameserver 8.8.8.8 to /etc/resolv.conf:

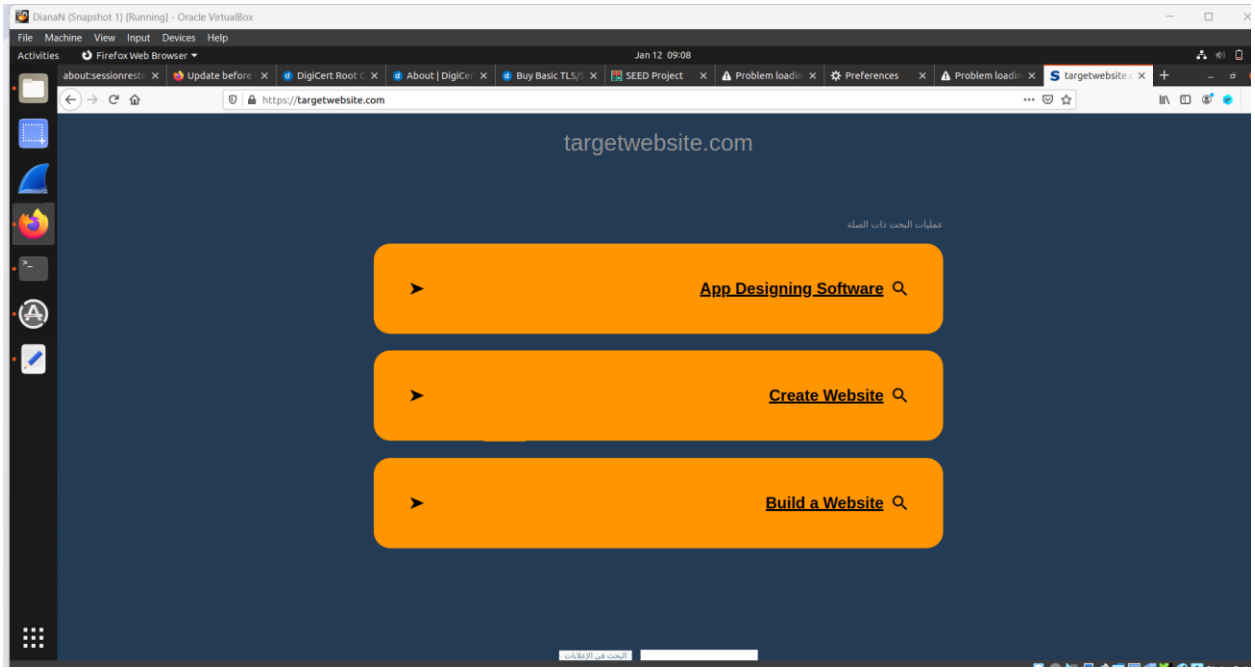


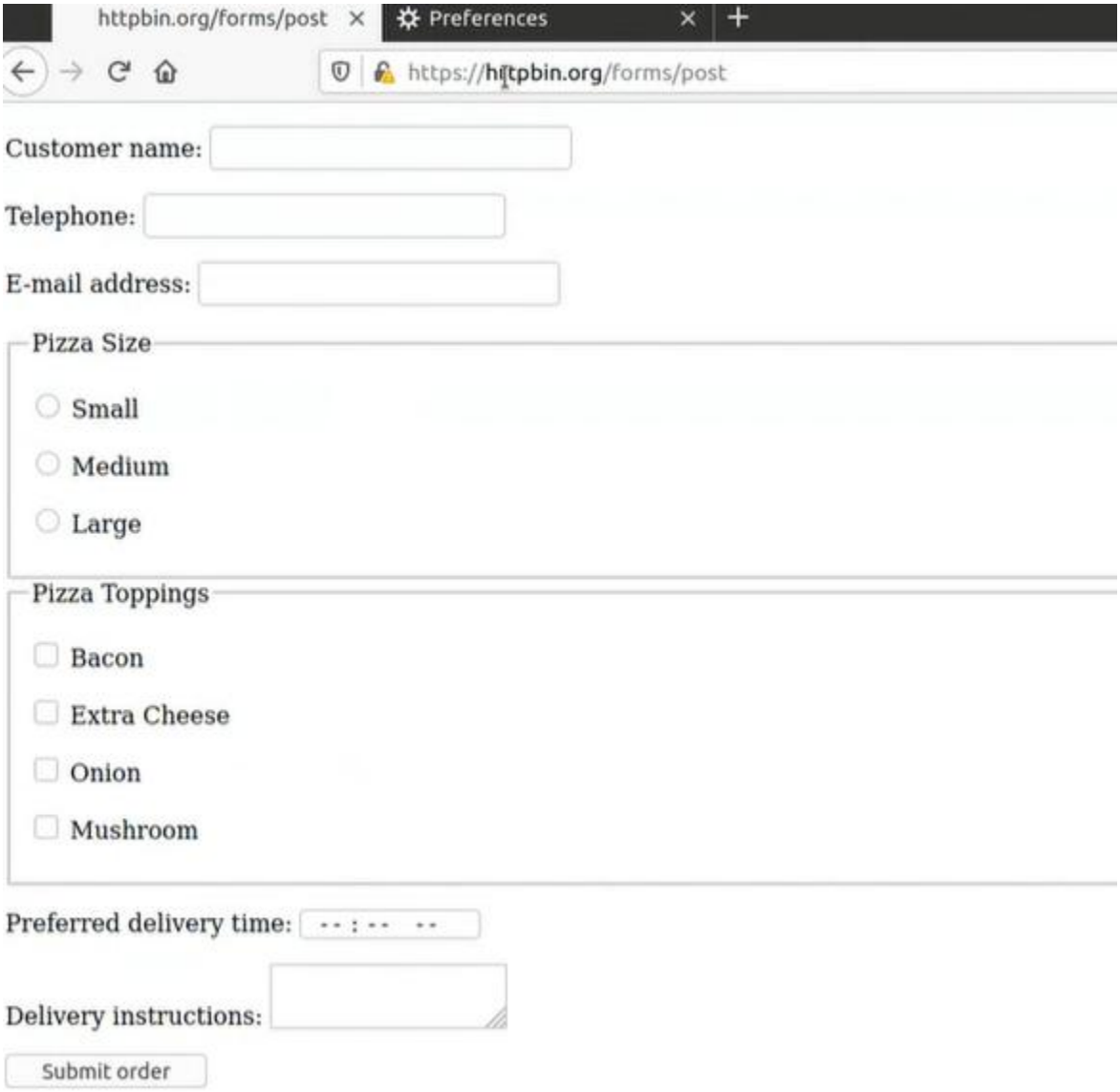
On the **client container**, send an HTTPS request to your server:



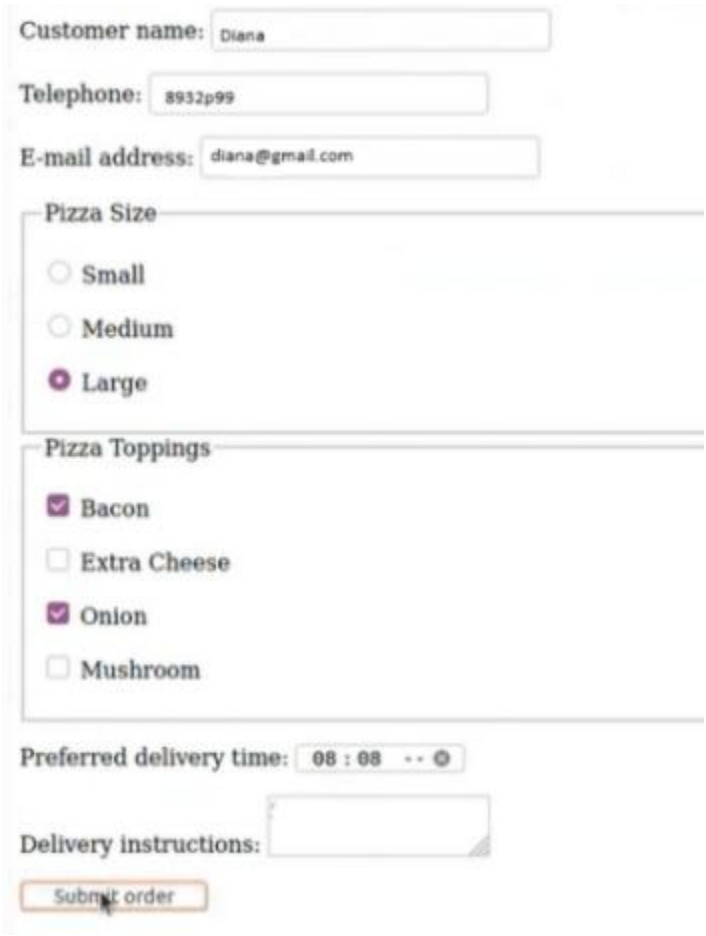
Add 10.9.0.143 targetwebsite.com to etc/hosts:







\Test httpbin.org and we noticed that the proxy can take the data when we enter it and this is very high risk.




```
seed@VM: ~/.../Labsetup
seed@VM: ~...  seed@VM: ~...  seed@VM: ~...  seed@VM: ~...  seed@VM: ~...  seed@VM: ~...
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Upgrade-Insecure-Requests: 1


[*] Handling connection for: httpbin.org
POST /post HTTP/1.1
Host: httpbin.org
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 F
efox/83.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;
0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded
Content-Length: 118
Origin: https://httpbin.org
Connection: keep-alive
Referer: https://httpbin.org/forms/post
Upgrade-Insecure-Requests: 1

custname=Diana&custtel=8932p99&custemail=diana@gmail.com&size=large&topping=bac
```

As we see that the proxy can catch the data.

In this task, I executed a Man-in-the-Middle (MITM) attack using an HTTPS proxy. To start, I modified the /etc/hosts file on my virtual machine to redirect traffic from targetwebsite.com to the MITM proxy container's IP address (10.9.0.143). This enabled interception of traffic originally intended for the target website. After setting up the proxy server within the MITM container, I tested the configuration by accessing the target website in a browser. The redirection worked as expected, and the proxy successfully intercepted HTTP and HTTPS traffic.

To further demonstrate the attack, I tested form submissions on the intercepted webpage. By submitting sample form data, I observed that the proxy captured sensitive information such as the customer's name, telephone number, email address, and preferences in plaintext. This highlighted the severe security risks associated with compromised or redirected traffic in a MITM scenario.

Finally, I documented the results and ensured proper cleanup by restoring the original state of the /etc/hosts file, removing any changes made during the attack. This task underscores the importance of secure traffic handling and the vulnerabilities of unprotected networks.

Appendcies:

Code for Task1.d:

```
#!/usr/bin/env python3

import socket
import ssl
import sys
import pprint

hostname = sys.argv[1]
port = 443
cadir = '/etc/ssl/certs'

# cadir = './client-certs'

# Set up the TLS context
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)  # For Ubuntu 20.04 VM
# context = ssl.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()
```

==========

server.py modified code:

```
#!/usr/bin/env python3

import socket
import ssl

# HTML content to be sent to the client
html = """
HTTP/1.1 200 OK\r\nContent-Type: text/html\r\n\r\n
<DOCTYPE html><html><body><h1>Hello, World!</h1></body></html>
"""
```

```
# Paths to the server certificate and private key
SERVER_CERT = './server-certs/server.crt'  # Adjust the path as needed
SERVER_PRIVATE = './server-certs/server.key'

# Set up the TLS context
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain(SERVER_CERT, SERVER_PRIVATE)

# Create a socket for the server
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
sock.bind(('0.0.0.0', 443))  # Bind to port 443 (HTTPS)
sock.listen(5)

while True:
    newsock, fromaddr = sock.accept()
    try:
        # Wrap the socket with TLS
        ssock = context.wrap_socket(newsock, server_side=True)
        print("TLS connection established!")

        # Read data from the client
        data = ssock.recv(1024)
        print(f"Request: {data.decode('utf-8')}")

        # Send HTML response to the client
        ssock.sendall(html.encode('utf-8'))

        # Close the TLS connection
        ssock.shutdown(socket.SHUT_RDWR)
        ssock.close()
    except Exception as e:
        print(f"Error: {e}")
        continue
```