Task 1: Becoming a Certificate Authority (CA)

In this task I created a new directory, copied the openssl.conf configuration file into it, and then created several sub-directories:

```
dir             = ./demoCA          # Where everything is kept
certs           = $dir/certs        # Where the issued certs are kept
crl_dir         = $dir/crl          # Where the issued crl are kept
new_certs_dir   = $dir/newcerts     # default place for new certs.
database        = $dir/index.txt    # database index file.
serial          = $dir/serial       # The current serial number
```

It was also required that the index.txt file was created and left empty, and the serial file was given a single number in string format (e.g. 1000).

Now I need to generate a self-signed certificate for my CA which means that the CA is totally trusted its certificate will serve as the root certificate. To do this, I run the command:

$ openssl req -new -x509 -keyout ca.key -out ca.crt -config openssl.cnf

The password I used for everything in this lab is 'dees'. See Figures 1 and 2 for the execution of this task.

```
Terminal
[11/18/19]JCogswell@Attacker-131:~$ mkdir demoCA
[11/18/19]JCogswell@Attacker-131:~$ cd demoCA
[11/18/19]JCogswell@Attacker-131:~/demoCA$ mkdir certs crl newcerts
[11/18/19]JCogswell@Attacker-131:~/demoCA$ echo 1000 > serial
[11/18/19]JCogswell@Attacker-131:~/demoCA$ subl index.txt
[11/18/19]JCogswell@Attacker-131:~/demoCA$ pwd
/home/JCogswell/demoCA
[11/18/19]JCogswell@Attacker-131:~/demoCA$ cp "/usr/lib/ssl/openss.cnf" "/home/J
Cogswell/demoCA"
cp: cannot stat '/usr/lib/ssl/openss.cnf': No such file or directory
[11/18/19]JCogswell@Attacker-131:~/demoCA$ cp "/usr/lib/ssl/openssl.cnf" "/home/
JCogswell/demoCA"
```

Figure 1

Figure 2

```
Terminal
[11/18/19]JCogswell@Attacker-131:~/demoCA$ openssl req -new -x509 -keyout ca.key
 -out ca.crt -config openssl.cnf
Generating a 2048 bit RSA private key
.....................................................................+++
.....................................................................+++
writing new private key to 'ca.key'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Virginia
Locality Name (eg, city) []:Ashburn
Organization Name (eg, company) [Internet Widgits Pty Ltd]:CYSE
Organizational Unit Name (eg, section) []:001
Common Name (e.g. server FQDN or YOUR name) []:James Cogswell
Email Address []:jcogswe@gmu.edu
```

Task 2: Creating a Certificate for SEEDPKILab 2018.com

In this task I have become a root CA and I'm ready to start signing certificates. The first certificate I will sign is for a company called SEEDPKILab2018.com. For this company to get a digital certificate from a CA, it needs to go through three steps:

Step 1: Generate public/private key pair

The company needs to create its own public/private key pair. Since they don't exist, I did this part for them using the command:

$ openssl genrsa -aes128 -out server.key 1024

The password I used is 'dees' and the keys will be stored in the server.key file. The server.key is an encoded text file (also encrypted), so I won't be able to see the actual content, such as the modulus, private exponent, etc. To see those, I can run the following command:

$ openssl rsa -in server.key -text

See Figure 3 for execution of this step.

```
Terminal
[11/18/19]JCogswell@Attacker-131:~/demoCA$ openssl rsa -in server.key -text
Enter pass phrase for server.key:
Private-Key: (1024 bit)
modulus:
    00:98:10:80:71:64:34:bd:78:14:26:04:15:d5:a4:
    12:ef:a1:8d:07:6a:13:59:2c:d6:40:6d:1f:48:f3:
    ce:76:32:33:1e:ae:5c:ca:1f:7d:6b:99:60:7d:cb:
    a7:77:67:ab:de:05:ea:b6:84:0d:9d:6d:1c:4d:7f:
    9d:78:b5:f1:f8:5c:24:a7:25:14:bf:9e:ec:2b:f3:
    75:38:a4:f0:82:0a:5e:f6:3f:a8:08:44:b4:0a:16:
    79:de:0f:7b:cf:2a:5d:26:2d:1a:5b:63:9b:3c:08:
    5e:e0:64:39:65:49:0f:40:1c:da:96:60:33:65:e1:
    a7:f9:e8:50:4a:eb:d6:a0:67
publicExponent: 65537 (0x10001)
privateExponent:
    0d:3d:ea:a1:75:33:6e:88:65:60:c6:1c:4c:a5:64:
    0c:c0:09:b2:ea:96:d2:be:27:ce:26:72:81:fc:d1:
    2e:d3:c2:45:da:b3:33:da:b8:26:45:8d:f4:86:66:
    87:ec:5c:38:12:55:d5:1c:fc:be:87:c0:de:32:cd:
    b8:b4:47:64:d1:e9:ff:10:5c:c1:4a:72:2d:4d:6e:
    35:48:cd:52:82:77:4e:aa:29:ae:f4:85:fe:a4:6e:
    6e:37:22:7e:06:07:e0:98:6d:5e:ad:da:2c:d6:d9:
    30:de:ea:17:af:8b:ac:be:d2:c6:6e:f3:dd:e2:4d:
    7a:93:38:49:2c:09:03:21
prime1:
    00:c9:f3:d4:f4:f1:1b:e1:38:5c:40:8c:8d:09:e7:
    11:d5:5a:70:53:de:e1:61:05:6e:ef:18:14:58:00:
    f0:8a:ff:93:d9:ea:c2:5a:cf:4c:f1:6b:27:23:e3:
    dd:17:ea:8f:1a:fe:8b:af:f7:e7:ff:04:83:e5:cd:
    a1:4d:5d:01:11
prime2:
    00:c0:c2:be:13:70:50:ec:f5:fc:a9:d3:26:d3:ee:
    f5:65:c9:20:57:e0:3f:81:b7:58:85:94:2d:ad:94:
    62:e2:07:1a:8a:e6:6e:52:02:95:76:c4:02:18:3b:
    c1:ce:09:a1:26:7c:09:2c:ec:03:1a:83:a9:a7:69:
    63:f1:01:09:f7
exponent1:
    6e:67:72:a0:85:bf:6c:1c:d7:b2:22:24:68:a9:7e:
    42:59:32:d7:67:1d:67:61:05:08:88:ed:76:3e:ed:
    41:cf:ff:10:3f:1b:09:72:12:ea:58:5e:09:6c:e4:
    cd:d2:59:71:b4:7e:c1:30:92:10:04:73:e7:09:c7:
    88:f7:b3:41
exponent2:
```

Figure 3 (both)

```
-----BEGIN RSA PRIVATE KEY-----
MIICWwIBAAKBgQCYEIBxZDS9eBQmBBXVpBLvoY0HahNZLNZAbR9I8852MjMerlzK
H31rmWB9y6d3Z6veBeq2hA2dbRxNf514tfH4XCSnJRS/nuwr83U4pPCCCl72P6gI
RLQKFnneD3vPKl0mLRpbY5s8CF7gZDllSQ9AHNqWYDNl4af56FBK69agZwIDAQAB
AoGADT3qoXUzbohlYMYcTKVkDMAJsuqW0r4nziZygfzRLtPCRdqzM9q4JkWN9IZm
h+xcOBJV1Rz8vofA3jLNuLRHZNHp/xBcwUpyLU1uNUjNUoJ3TqoprvSF/qRubjci
fgYH4JhtXq3aLNbZMN7qF6+LrL7Sxm7z3eJNepM4SSwJAyECQQDJ89T08RvhOFxA
jI0J5xHVWnBT3uFhBW7vGBRYAPCK/5PZ6sJaz0zxaycj490X6o8a/ouv9+f/BIPl
zaFNXQERAkEAwMK+E3BQ7PX8qdMm0+71ZckgV+A/gbdYhZQtrZRi4gcaiuZuUgKV
dsQCGDvBzgmhJnwJLOwDGoOpp2lj8QEJ9wJAbmdyoIW/bBzXsiIkaKl+Qlky12cd
Z2EFCIjtdj7tQc//ED8bCXIS6lheCWzkzdJZcbR+wTCSEARz5wnHiPezQQJAa+uc
J532cPS35QzG7FkZI0ebyBdQ9vsDoKX5fPla81kuzx18HoY70CalVZP/Sa+yfa1s
Ja5tfmbAkW47+i6uuQJAHzqH1fjAdPABHfQxHKgRd0lvKAYpOSs2gbhVQrIXEo/x
Wy2ZU7bZphJHaydniWIevPImI8jiI6Y1vO2uheogAA==
-----END RSA PRIVATE KEY-----
```

```
:70:f4:b7:e5:0c:c6:ec:59:19:
:f6:fb:03:a0:a5:f9:7c:f9:5a:
::1e:86:3b:d0:26:a5:55:93:ff:
::25:ae:6d:7e:66:c0:91:6e:3b:

:74:f0:01:1d:f4:31:1c:a8:11:
:39:2b:36:81:b8:55:42:b2:17:
:53:b6:d9:a6:12:47:6b:27:67:
:23:c8:e2:23:a6:35:bc:ed:ae:

TE KEY-----
ZDS9eBQmBBXVpBLvoY0HahNZLNZAbR9I8852MjMerlzK
```

Step 2: Generate a Certificate Signing Request (CSR).

Once the company has the key file, it should generate a CSR which includes the company's public key. The CSR is sent to the CA, who generates a certificate for the key (after ensuring the identity information in the CSR matches with the server's true identity). It should be noted that the command below is quite similar to the one we used in creating the self-signed certificate for the CA. The only difference is the -x509 option. Without it, the command generates a request; with it, the command generates a self-signed certificate. See Figure 4 for the execution of this step. The command used is:

$ openssl req -new -key server.key -out server.csr -config openssl.cnf

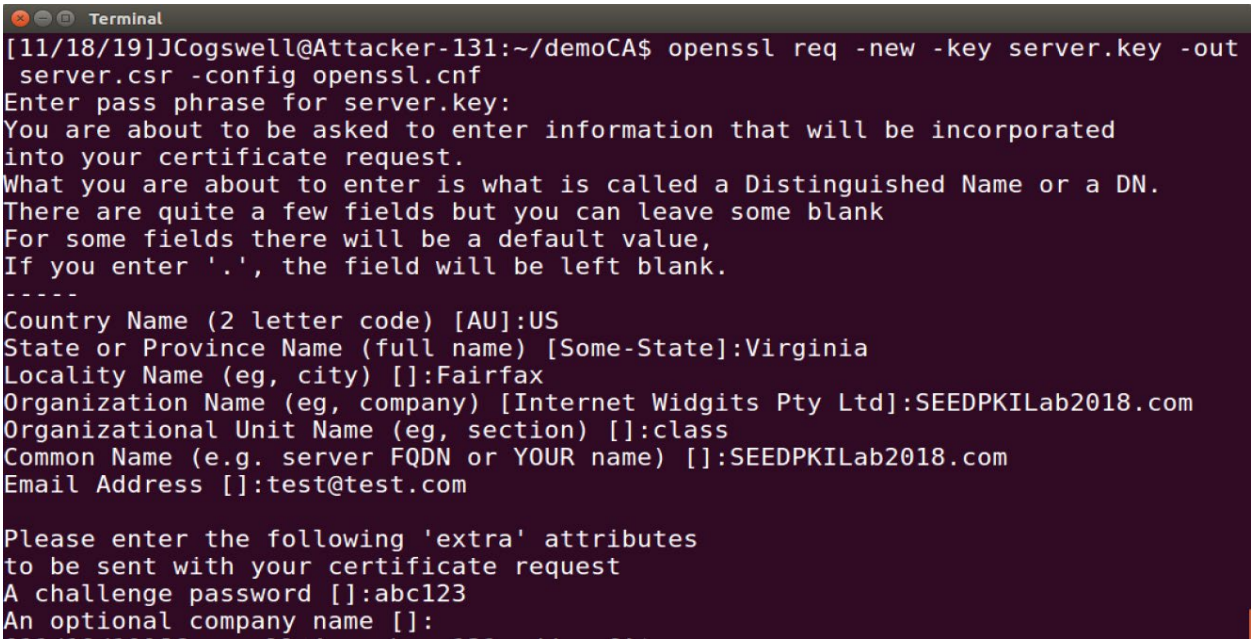


Figure 4

Step 3: Generating Certificates.

The CSR file needs to have the CA's signature to form a certificate. In the real world, the CSR files are usually sent to a trusted CA for their signature. In this lab, we used our own trusted CA to generate certificates. The following command turns the certificate signing request (server.csr) into an X509 certificate (server.crt), using the CA's ca.crt and ca.key.

$ openssl ca -in server.csr -out server.crt -cert ca.crt -keyfile ca.key -config openssl.cnf

I also opened the openssl.cnf file and updated the *policy_match* section from saying "*policy = policy_match*" to "*policy = policy_anything*" so the CA will be more flexible on policy when signing certificates.

See Figure 4 for the execution of this step.

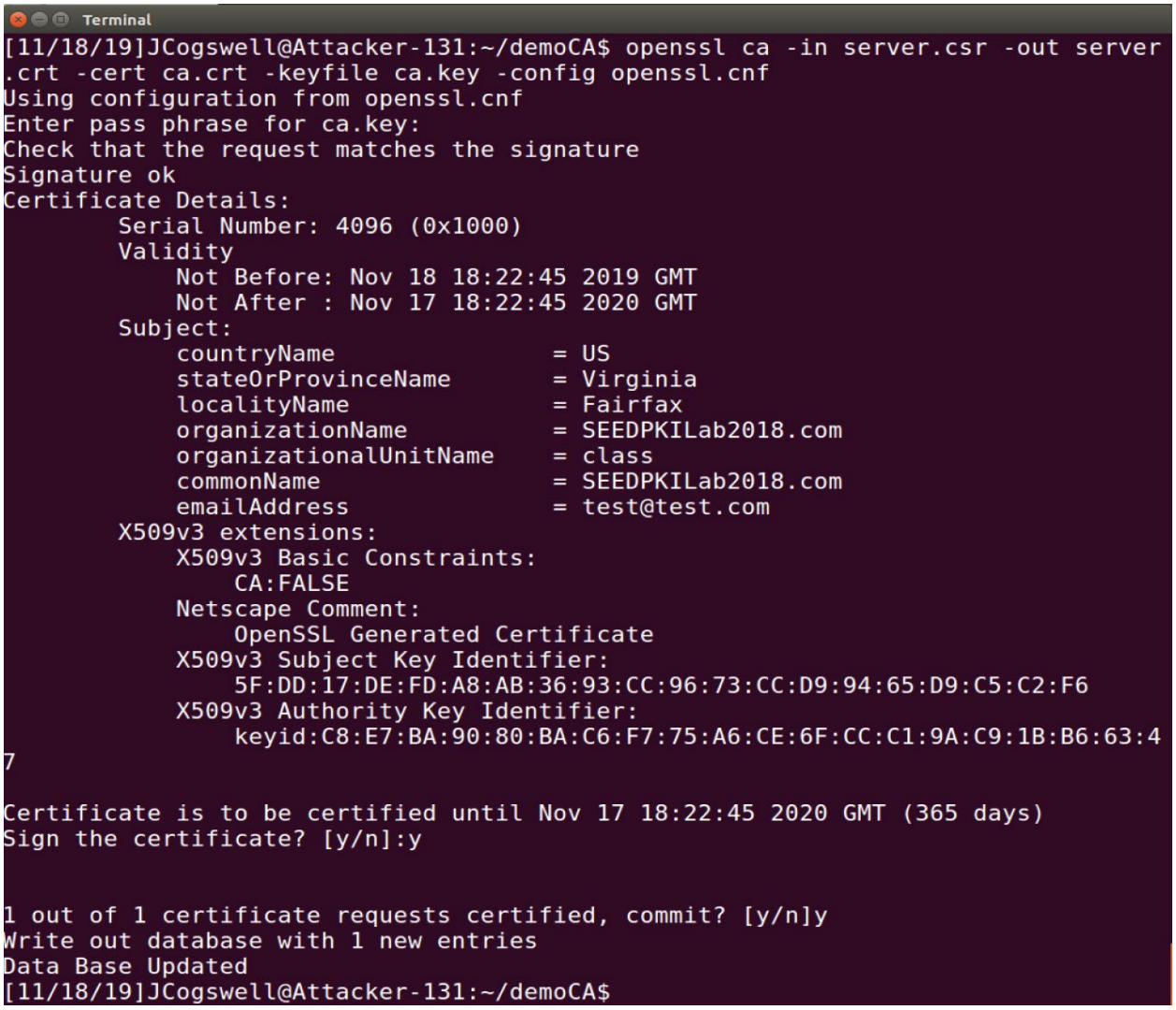

Terminal
[11/18/19]JCogswell@Attacker-131:~/demoCA$ openssl ca -in server.csr -out server
.crt -cert ca.crt -keyfile ca.key -config openssl.cnf
Using configuration from openssl.cnf
Enter pass phrase for ca.key:
Check that the request matches the signature
Signature ok
Certificate Details:
Serial Number: 4096 (0x1000)
Validity
Not Before: Nov 18 18:22:45 2019 GMT
Not After : Nov 17 18:22:45 2020 GMT
Subject:
countryName = US
stateOrProvinceName = Virginia
localityName = Fairfax
organizationName = SEEDPKILab2018.com
organizationalUnitName = class
commonName = SEEDPKILab2018.com
emailAddress = test@test.com
X509v3 extensions:
X509v3 Basic Constraints:
CA:FALSE
Netscape Comment:
OpenSSL Generated Certificate
X509v3 Subject Key Identifier:
5F:DD:17:DE:FD:A8:AB:36:93:CC:96:73:CC:D9:94:65:D9:C5:C2:F6
X509v3 Authority Key Identifier:
keyid:C8:E7:BA:90:80:BA:C6:F7:75:A6:CE:6F:CC:C1:9A:C9:1B:B6:63:4
7
Certificate is to be certified until Nov 17 18:22:45 2020 GMT (365 days)
Sign the certificate? [y/n]:y
1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
[11/18/19]JCogswell@Attacker-131:~/demoCA$

Figure 4

Task 3: Deploying Certificate in an HTTPS Web Server

This task explores how public-key certificates are used by website to secure web browsing. We will set up an HTTPS website using openssl's built-in web server.

Step 1: Configuring DNS.

I am to choose SEEDPKILab2018.com as the name of my website. To get my computer to recognize this name, I add the entry below to the */etc/hosts* directory which basically maps the hostname SEEDPKILab2018.com to our local host (127.0.0.1). See Figure 5 for details.

(in /etc/hosts): *127.0.0.1 SEEDPKILab2018.com*

/etc/hosts - Sublime Text (UNREGISTERED)

openssl.cnf | default-ssl.conf | note.txt | hosts

```
1  127.0.0.1   localhost
2  127.0.1.1   Attacker-131
3  127.0.0.1   www.seedlablenext.com
4  192.168.95.130 www.seedlabsqlinjection.com
5  127.0.0.1   SEEDPKILab2018.com
6  127.0.0.1   facebook.com
7
8  # The following lines are desirable for IPv6 capable hosts
9  ::1     ip6-localhost ip6-loopback
10 fe00::0 ip6-localnet
11 ff00::0 ip6-mcastprefix
12 ff02::1 ip6-allnodes
13 ff02::2 ip6-allrouters
14 127.0.0.1       User
15 127.0.0.1       Attacker
16 127.0.0.1       Server
17 127.0.0.1       www.SeedLabSQLInjection.com
18 127.0.0.1       www.xsslabelgg.com
19 127.0.0.1       www.csrflabelgg.com
20 127.0.0.1       www.csrflabattacker.com
21 127.0.0.1   www.repackagingattacklab.com
22 127.0.0.1   www.seedlabclickjacking.com
```

Figure 5

Step 2: Configuring the Web Server.

I'm going to launch a simple web server with a certificate generated in the previous task. OpenSSL allows us to start a simple web server using the s_server command:

```
# Combine the secret key and certificate into one file
% cp server.key server.pem
% cat server.crt >> server.pem

# Launch the web server using server.pem
% openssl s_server -cert server.pem -www
```

By default, the server will listen on port 4433 which I can alter using the -accept option. I can now access the server using the URL: *https://SEEDPKILab2018.com:4433/*. See Figure 6 for the execution of this step and Figure 7 for the resulting web page which has an invalid security certificate.

Terminal

```
[11/18/19]JCogswell@Attacker-131:~/demoCA$ openssl s_server -cert server.pem -ww
w
Enter pass phrase for server.pem:
Using default temp DH parameters
ACCEPT
```

Figure 6

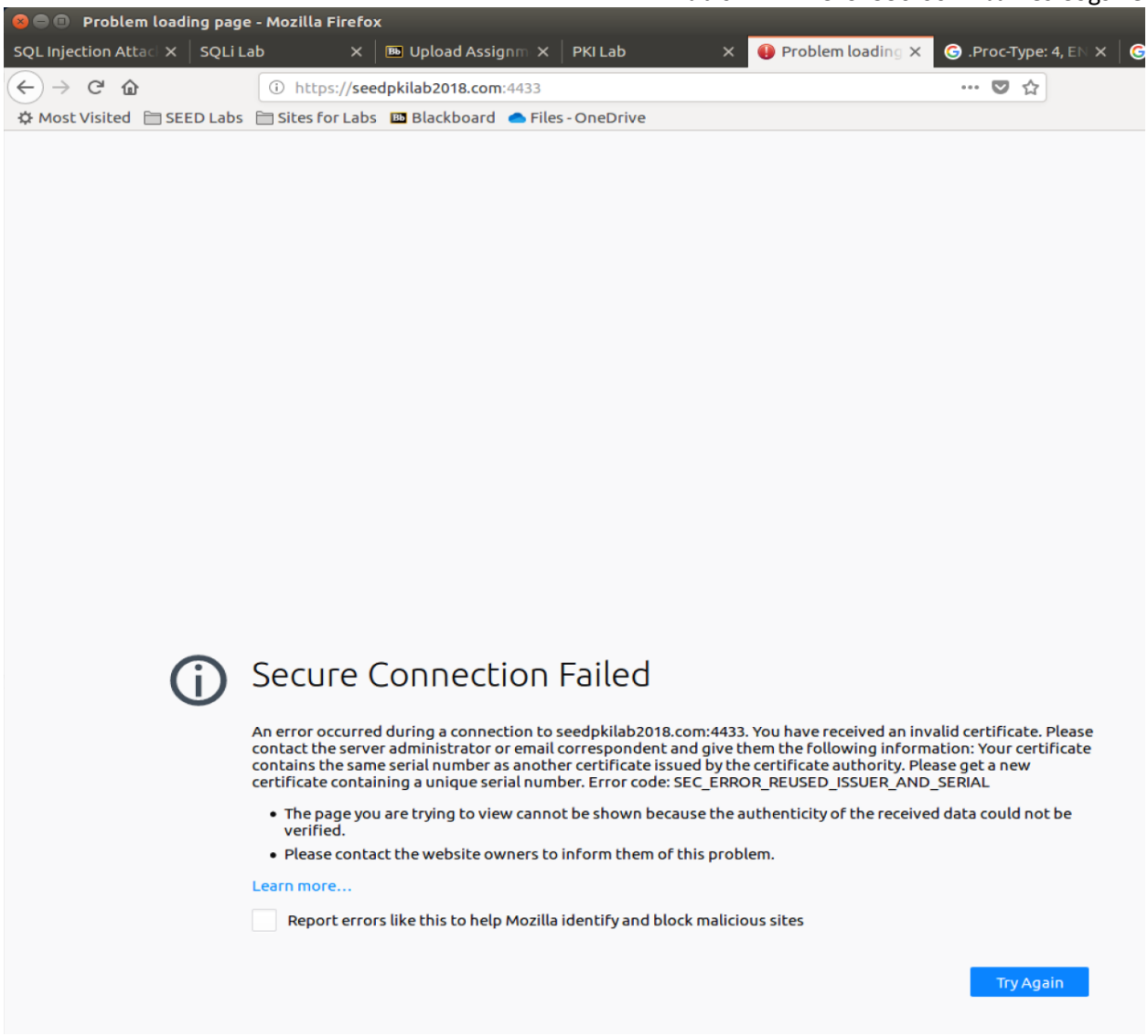


Figure 7

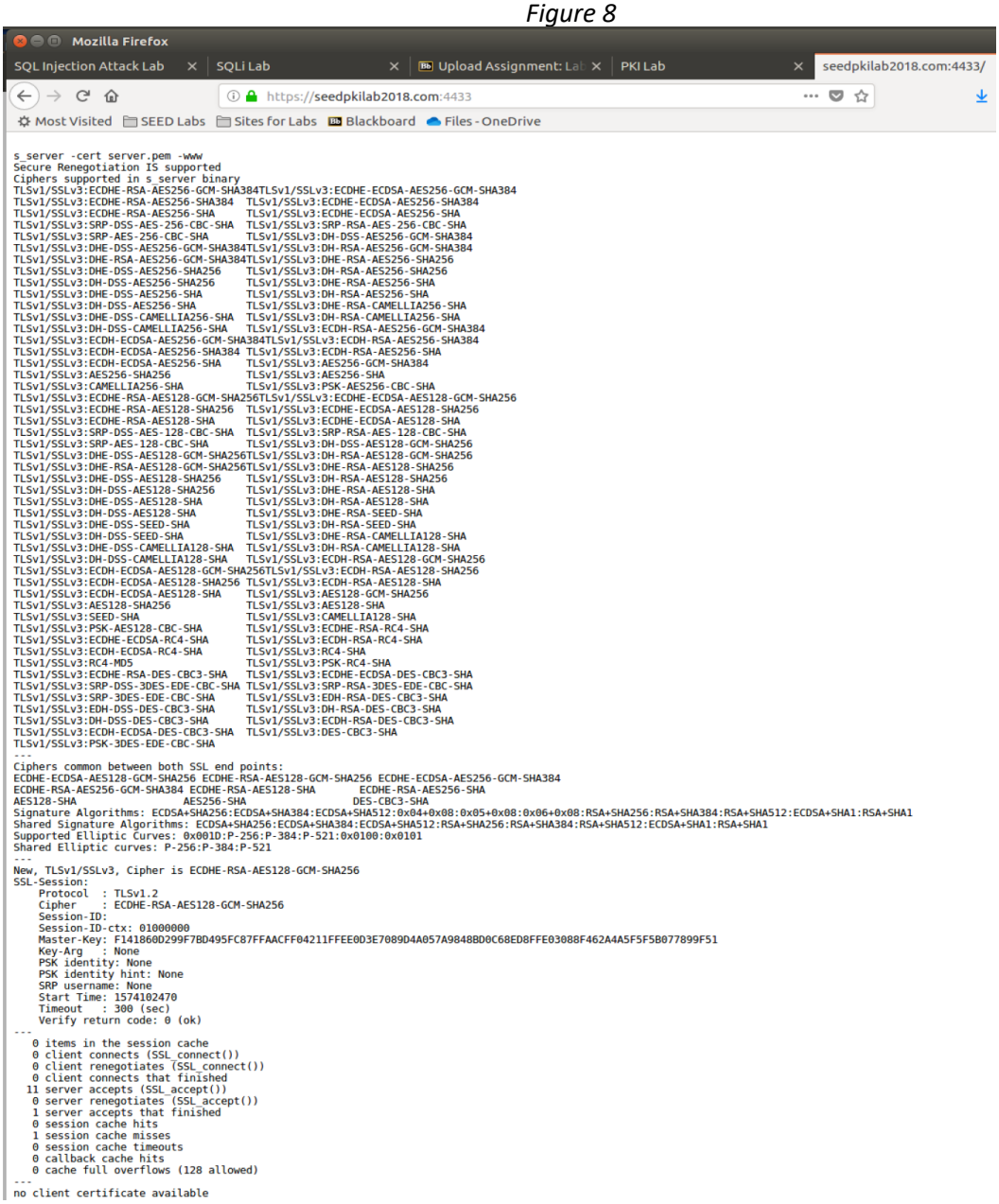
Step 3: Getting the Browser to Accept Our CA Certificate.

Since I am not a real CA that Firefox would recognize, I need to add the certificate *ca.crt* to the browser myself by going to *Edit → Preferences → Privacy & Security → View Certificates*. There will be a long list of certificates there and below them will be a button labeled "import". I click this and import the *ca.crt* certificate. No screenshot for this because everyone sees the exact same thing and it doesn't need proof since the next step won't work without doing it.

Step 4: Testing our HTTPS Website.

I now open Firefox again and go to *https://SEEDPKILab2018.com:4433*. See Figure 8 for the web page that displayed when I opened this URL.

Figure 8



Observations: I was able to obtain a certificate and view the web page. This is why we can see the information shown in Figure 8. We know that the ssl connection is successful because of the green padlock symbol to the left of the URL.

Now I'm going to modify a single byte of *server.pem* file, restart the server, and reload the URL. See Figure 9 for the screenshot of the Bless hex editor and Figure 10 for the webpage after altering the single byte and restarting the apache server.

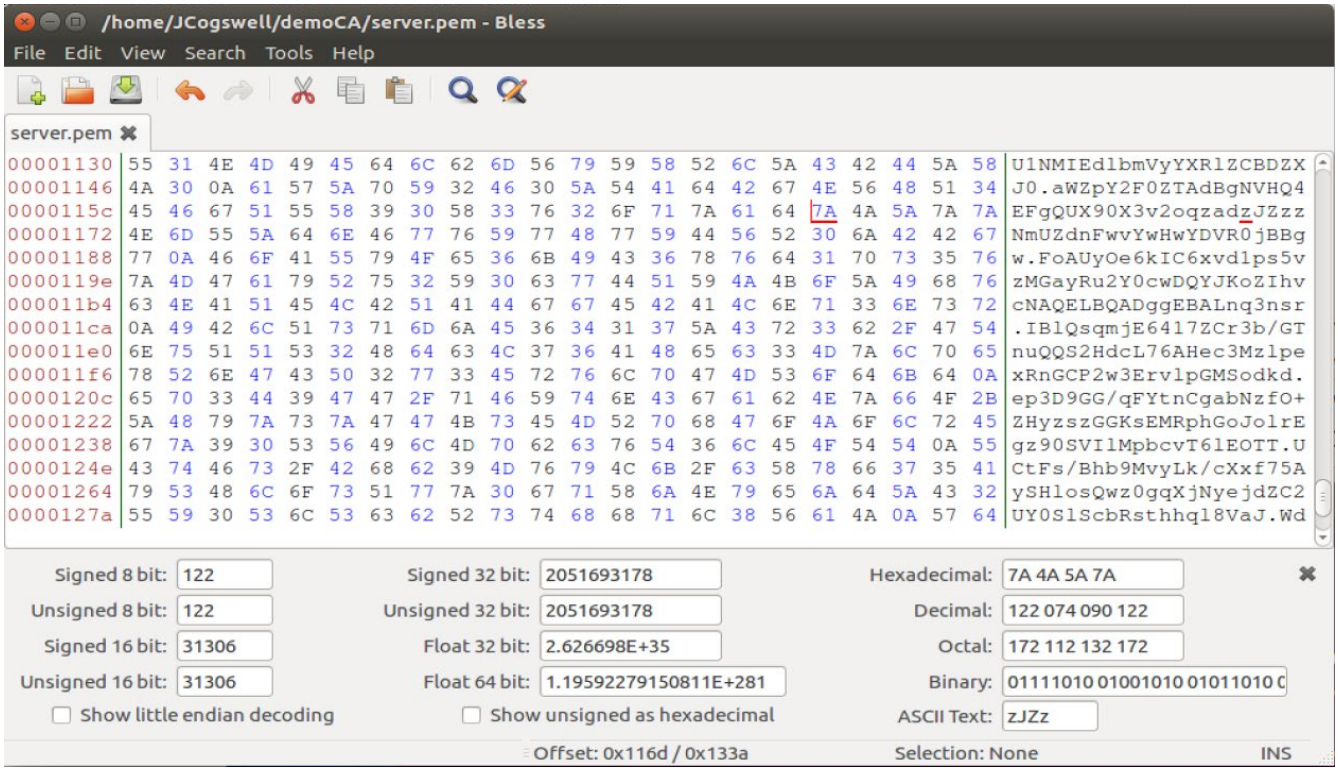
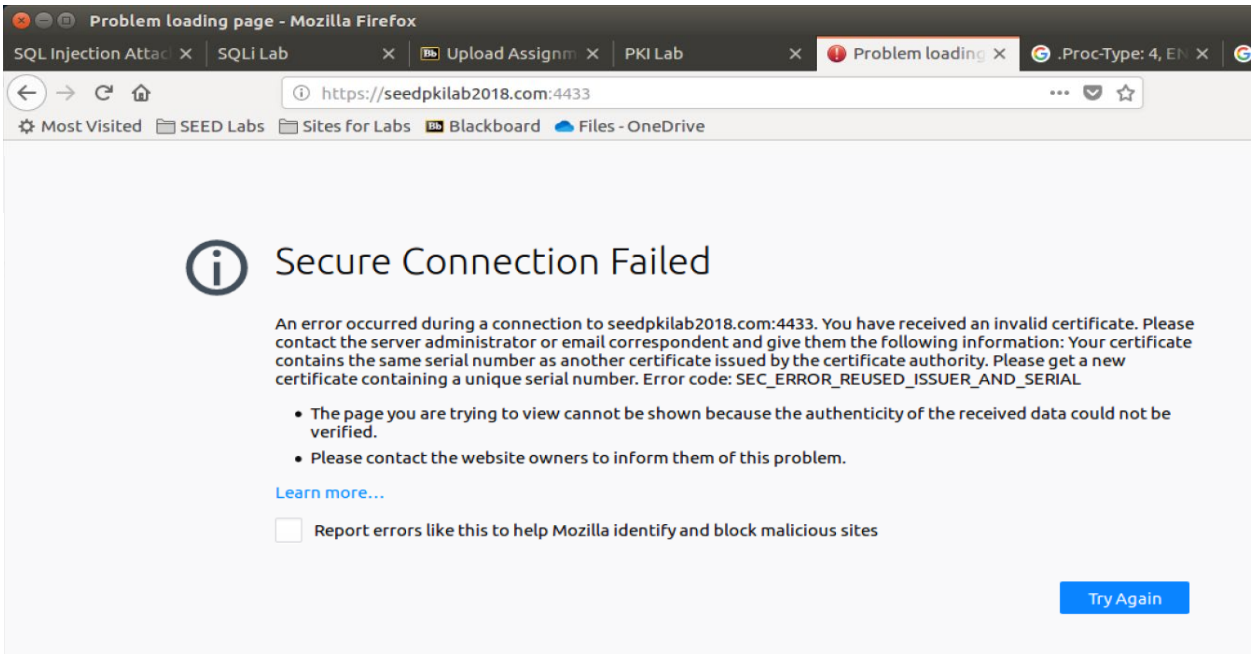


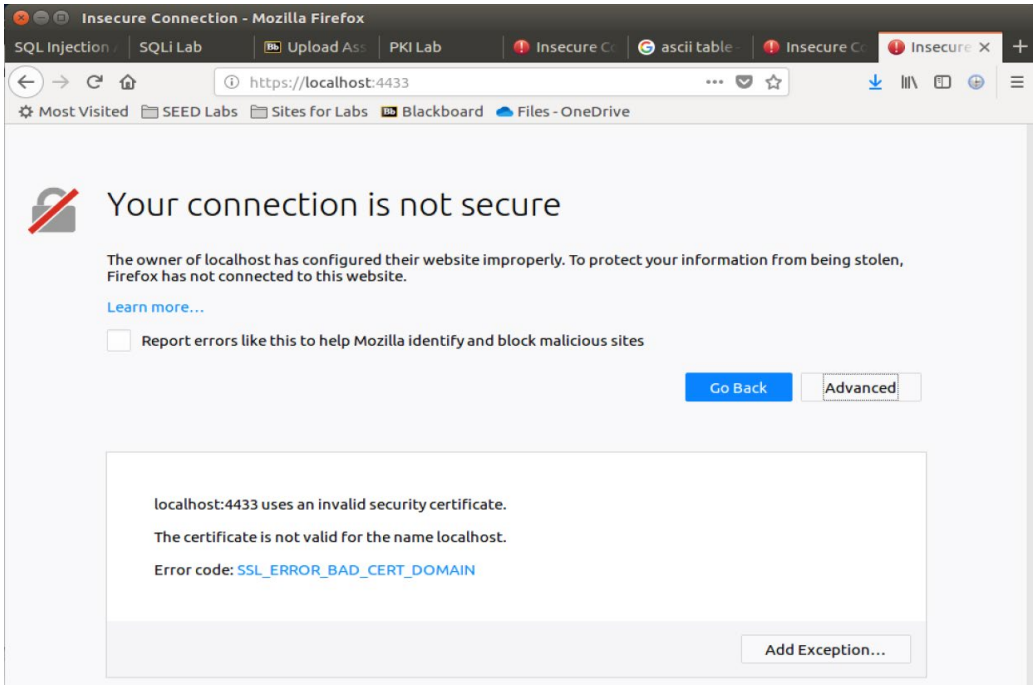
Figure 9

Figure 10



Since SEEDPKILab2018.com points (in /etc/hosts) to the localhost (127.0.0.1), if I use *https://localhost:4433* instead, I will be connecting to the same web server. I tried this and the result is shown below in Figure 11.

Figure 11 →



Observations: Previously, the key was valid so the website (https://SEEDPKILab2018.com:4433) loaded properly. The reason that doesn't load properly now is because we altered a single byte of the *server.pem* file and now the website will not have a valid certificate (Figure 10). For the second part (Figure 11), I had originally generated a key for the *actual* website *https://SEEDPKILab2018.com:4433*. Therefore, the certificate will not be signed since the URL is not the same. Refer back to the beginning where it was stated that it will check the actual identity of the site. There was no key or certificate generated for the https://localhost:4433 domain so a secure connection cannot be established.

Task 4: Deploying Certificate in an Apache-Based HTTPS Website

The HTTPS server setup using OpenSSL's *s_server* command is primarily for debugging and demonstration purposes. In this task, we set up a real HTTPS web server based on Apache. The Apache server, which is already installed on the SEED VMs, supports the HTTPS protocol. To create the HTTPS website, I just need to configure the Apache server, so it knows where to get the private key and certificates.

An Apache server can simultaneously host multiple websites. It needs to know the directory where a website's files are stored. This is done via its *VirtualHost* file, located in the */etc/apache2/sites-available* directory. To add an HTTPS website, we add a VirtualHost entry into the *default-ssl.conf* file in this folder. See Figure 12 for the execution of this step. My *cert.pem* and *key.pem* are stored in */etc/vmware-tools/GuestProxyData/server/* directory, as shown in Figure 12.

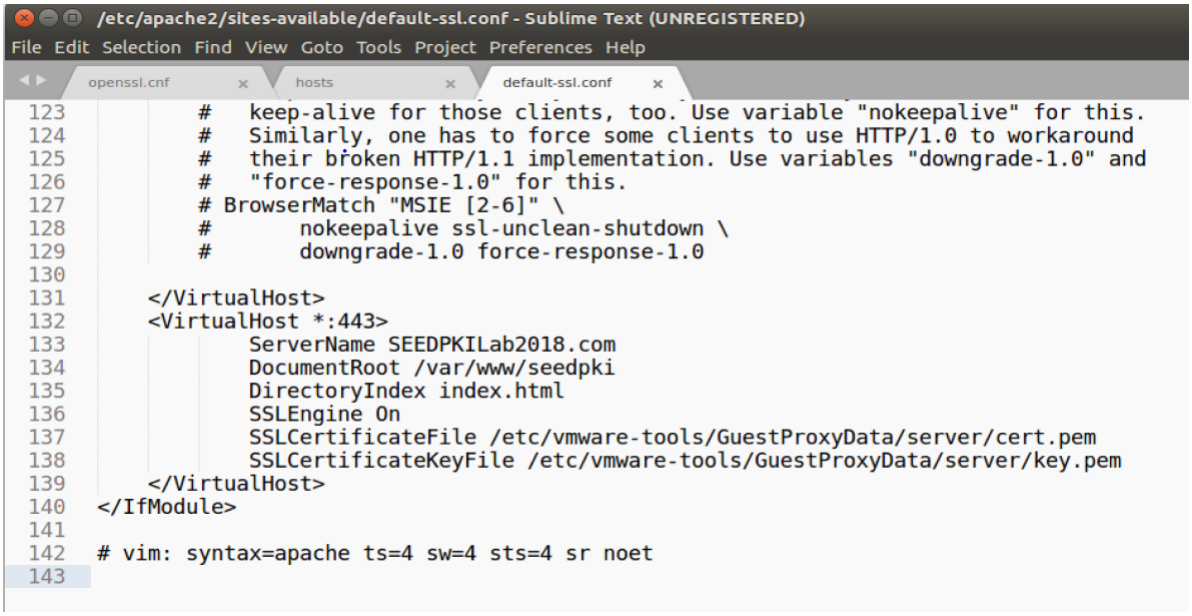
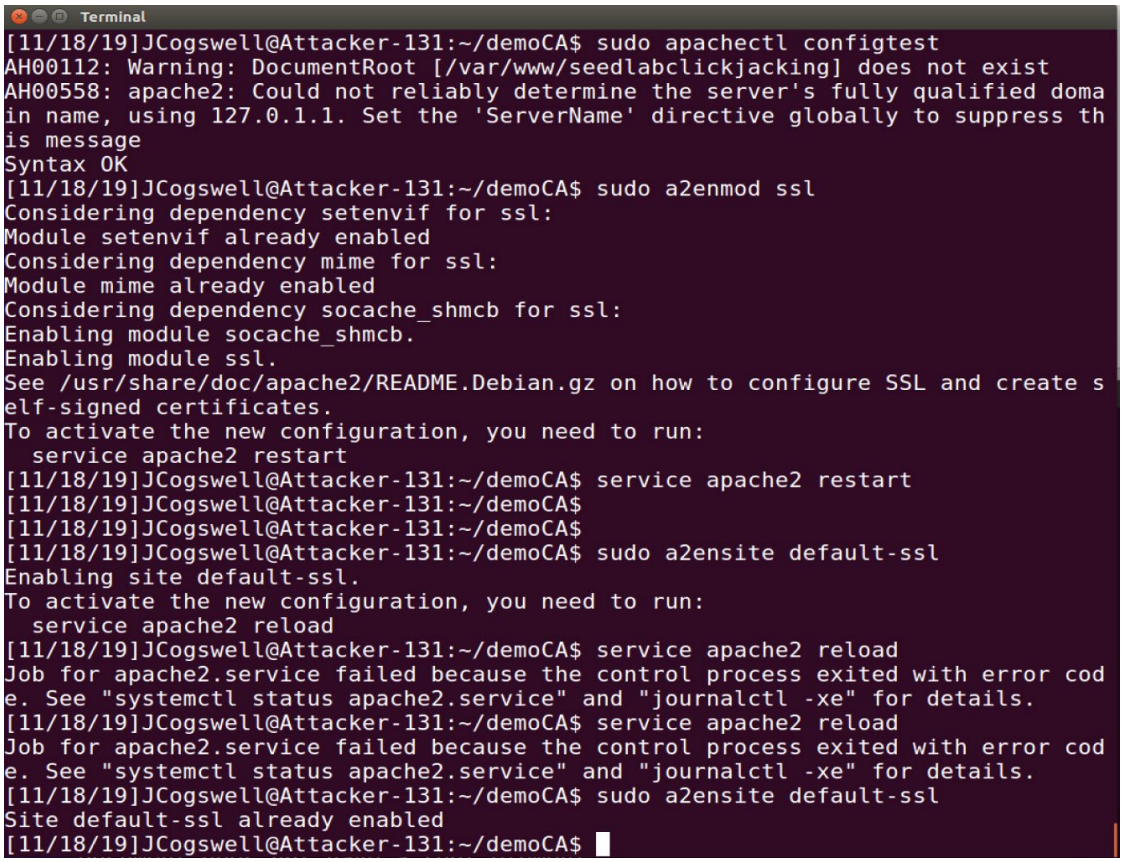


Figure 12

The *ServerName* entry specifies the name of the website, while the *DocumentRoot* entry specifies where the files for the website are stored. After modifying this file, I need to run a series of commands to enable SSL. Apache asks us to type the password used for encrypting the private key → password is dees. Once everything is set up, I can attempt to brows the site, and all the traffic between the browser and the server will be encrypted. See Figure 13 for execution of this step and Figure 14 for testing the website. The website works again!

Figure 13



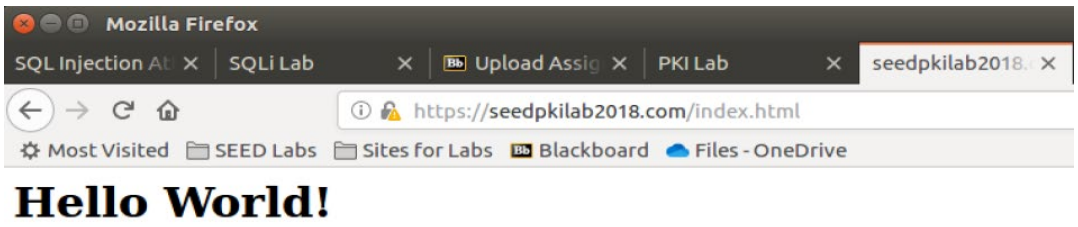


Figure 14

Task 5: Launching a Man-In-The-Middle Attack

In this task, I show how PKI can defat MITM attacks. Figure 15 depicts how MITM attacks work. Assume Alice wants to visit example.com via the HTTPS protocol. She needs to get the public key from the example.com server; Alice will generate a secret, and encrypt the secret using the server's public key, and send it to the server. If an attacker can intercept the communication between Alice and the server, the attacker can replace the server's public key with its own public key. Therefore, Alice's secret is actually encrypted with the attacker's public key, so the attacker will be able to read the secret. The attacker can forward the secret to the server using the server's public key. The secret is used to encrypt the communication between Alice and server, so the attacker can decrypt the encrypted communication.

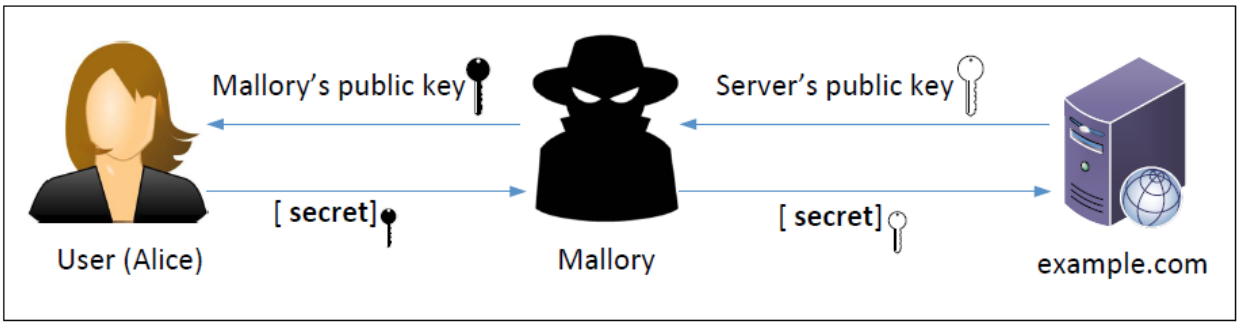


Figure 15

Over the next few steps, I am doing to emulate an MITM attack, and see exactly how PKI can defeat it. I first select a target website → facebook.com.

Step 1: Setting up the malicious website.

I already set up an HTTPS website for *SEEDPKILab2018.com* in Task for. I will use the same Apache server to impersonate *facebook.com*. To achieve this, I follow the same instructions in Task 4 to add a *VirtualHost* entry to Apache's SSL configuration file: the *ServerName* should be *facebook.com*, but the rest of the configuration can be the same as that used in Task 4. This is my goal: when a user visits *facebook.com*, the user will land in my server, which hosts a fake website for *facebook.com*. If I really wanted to do it, I could display an identical login page to *facebook.com* and steal the users credentials when the attempt to login on this fake webpage. See Figure 16 for the addition of the VirtualHost.

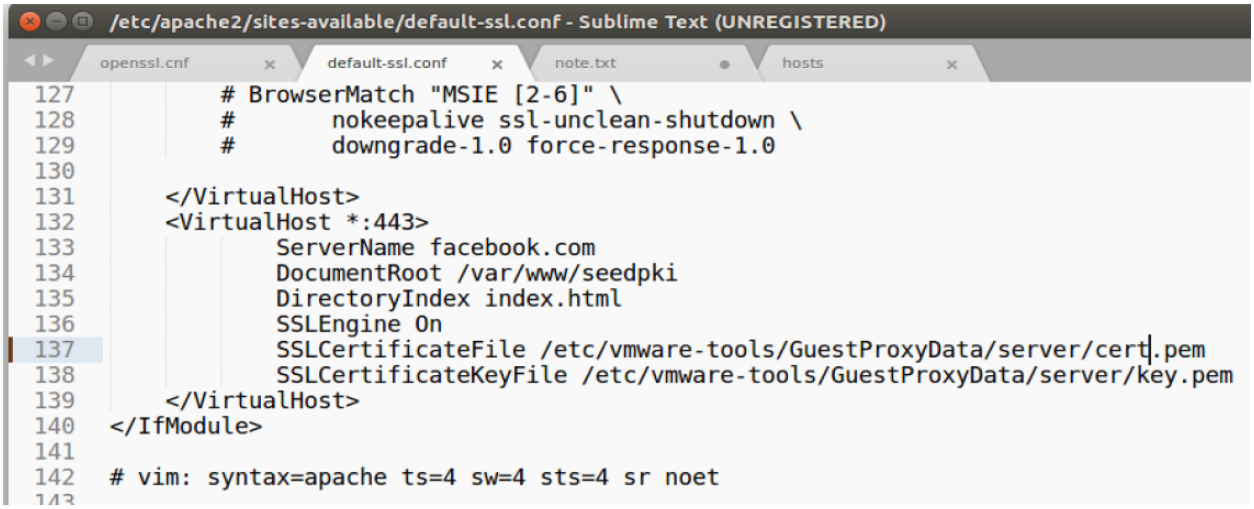


Figure 16

Step 2: Becoming the Man in the Middle.

There are several ways to get the user's HTTPS request to land in my web server. One way is to attack the routing, so the user's HTTPS request is routed to my web server. Another way is to attack DNS, so when the victim's machine tries to find out the IP address of the target web server, it gets the IP address of my web server. In this task, I'm using the "attack" DNS option. Instead of launching an actual DNS cache poisoning attack, I simply modify the victim machine's */etc/hosts* file to emulate the result of a DNS cache poisoning attack. See Figure 17 for execution of this step.

(in /etc/hosts) *127.0.0.1 facebook.com*

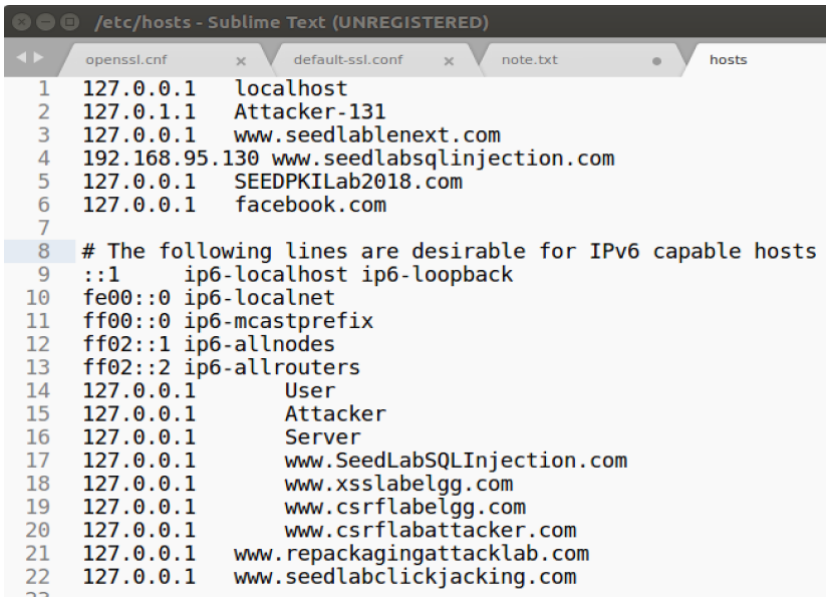


Figure 17

Step 3: Browse the Target Website.

Now that everything is set up, I visit the target real website and see what the browser says. See Figure 18 for the execution of this step.

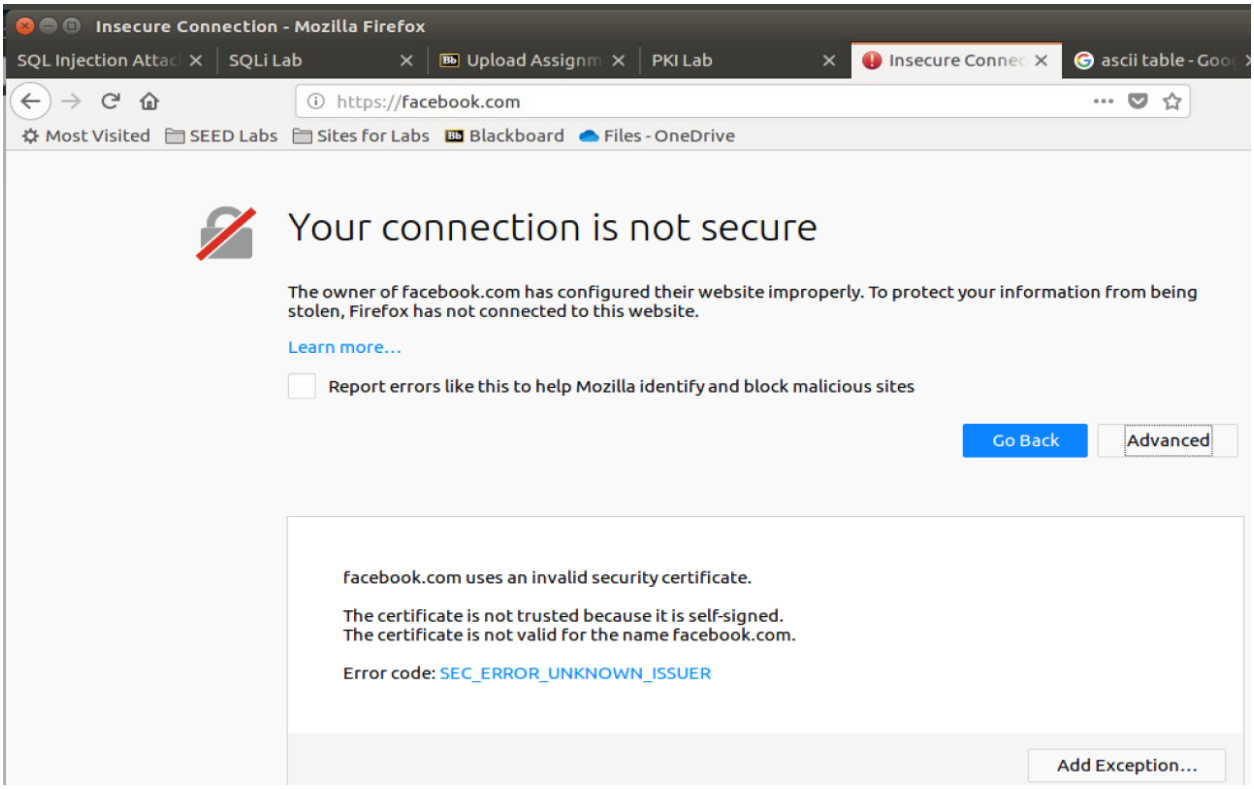


Figure 18

***Observations*:** The certificate is not trusted or valid because PKI protects against MITM attacks.

Task 6: Launching a Man-In-The-Middle Attack with a Compromised CA

The root CA that I created in Task 1 is now compromised by an attacker, and its private key is stolen. Therefore, the attacker can generate any arbitrary certificate using this CA's private key. In this task, I will show the consequence of such a compromise. In this task, the MITM attack will be successful against the HTTPS website. I will use the same settings created in Task 5, but this time, I will demonstrate that it will work. The browser will not be suspicious when the victim tries to visit a website but land in the MITM attacker's fake website.

The new certificate I created is facebook.pem. I placed this file into /etc/apache2/ssl and updated the /etc/apache2/sites-available/default-ssl.conf to reflect this change. This attack was successful and, as you can see in Figure 19, going to facebook.com will now bring you directly to my site. If I was really mean, it would do a *lot* more damage than simply say 'Hello World'!

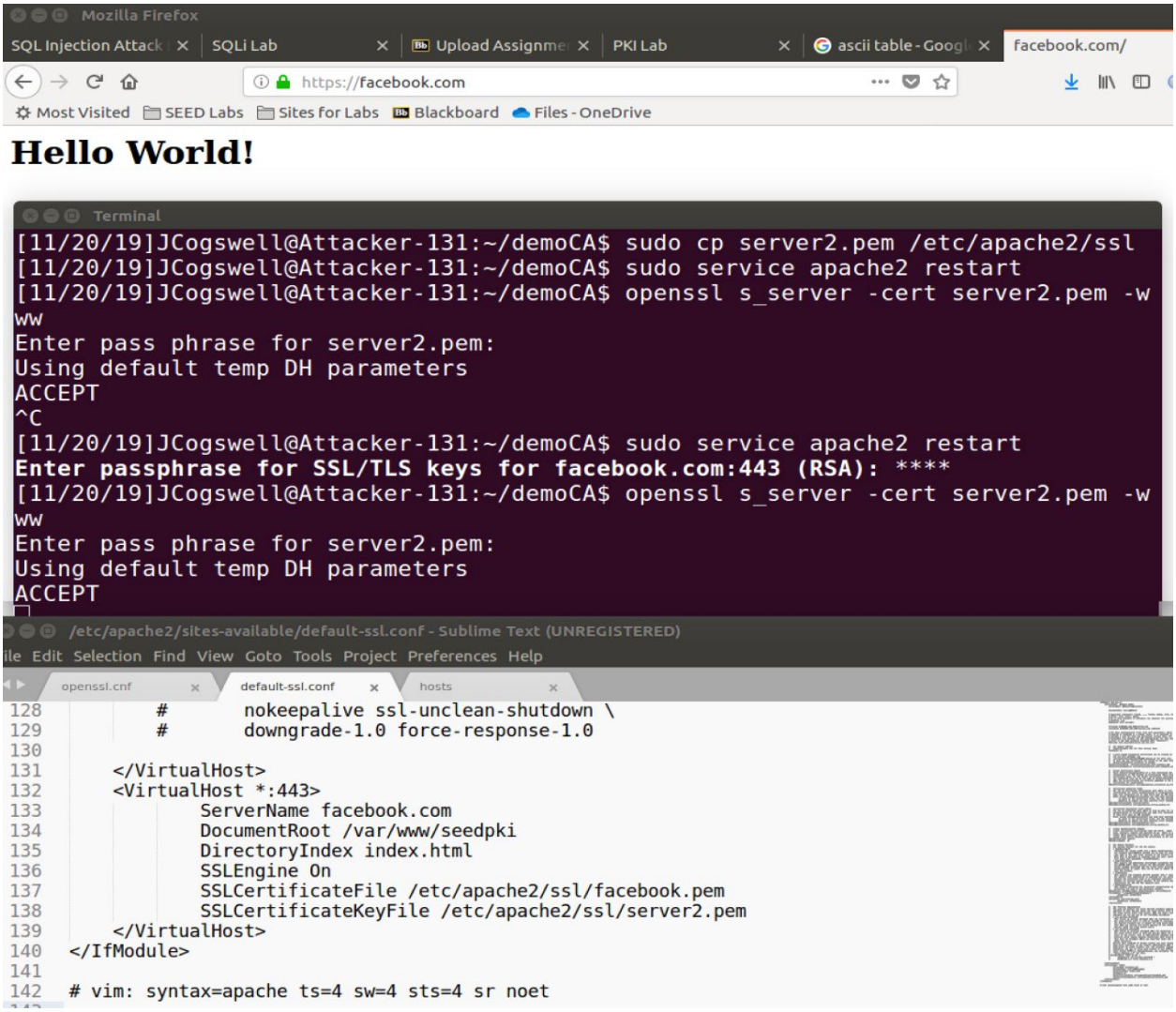


Figure 19

***It should be noted that since Task 6 did not work initially, I created all new certs. I made ca2.pem, server2.pem, etc. This was likely not necessary since it probably would have worked if I put the facebook.pem in the /etc/apache2/ssl/ directory.