Assignment 4

Task 3.1: Generating Message Digest and MAC

First, I created a sample.txt file with a string “Hello Computer Security Hackers”.

In this task, the Hash, or Message Digest for the following three algorithms were generated:

1) Md5
2) Sha1
3) Sha256

Result: Every algorithm leads to different hash size output. The Md5 algorithm generates 128 bits hash, Sha1 algorithm generates 160 bits hash and Sha256 algorithm generates 256 bits hash. Even if a single bit of value in the sample.txt changes, the hash value also changes. So, there is a unique hash value for every file.

```
[10/10/2017 12:02] seed@ubuntu:~/Desktop/assn4$ man openssl
[10/10/2017 13:05] seed@ubuntu:~/Desktop/assn4$ man dgst
[10/10/2017 13:05] seed@ubuntu:~/Desktop/assn4$ openssl dgst -md5 sample.txt
MD5(sample.txt)= ce01cb3ab8cf5d072cf5c3f199770578
[10/10/2017 13:05] seed@ubuntu:~/Desktop/assn4$ openssl dgst -sha1 sample.txt
SHA1(sample.txt)= 3f9afc095e7141e0f24d44a522518c2534d5fb16
[10/10/2017 13:05] seed@ubuntu:~/Desktop/assn4$ openssl dgst -sha256 sample.txt
SHA256(sample.txt)= 71820d9d12f7016526b4b8ab1620799c4f409896f1bf47bdc4065af9ab98
d7df
```

Task 3.2: Keyed Hash and HMAC

First, I created a sample.txt file with a string “Hello Computer Security Hackers”.

In this task, Keyed Hash for a file with different key sizes were generated by using following algorithms.

1) Md5
2) Sha1

Three different key sizes used in this task were

1) “a”
2) “abcdefg”
3) “Hello Computer Security class”

Md5

```
[10/10/2017 13:10] seed@ubuntu:~/Desktop/assn4$ openssl dgst -md5 -hmac "a" sample.txt
HMAC-MD5(sample.txt)= 99602e99e6b2f562f661dab56f9b0fab
[10/10/2017 13:10] seed@ubuntu:~/Desktop/assn4$ openssl dgst -md5 -hmac "abcdefg" sample.txt
HMAC-MD5(sample.txt)= 76757d57daa42312bec94cd3611190ce
[10/10/2017 13:11] seed@ubuntu:~/Desktop/assn4$ openssl dgst -md5 -hmac "Hello Computer Security class" sample.txt
HMAC-MD5(sample.txt)= 4a121ae67a80c36162aeec5b9c4af2a5
```

Sha1

```
[10/10/2017 13:20] seed@ubuntu:~/Desktop/assn4$ openssl dgst -sha1 -hmac "a" sample.txt
HMAC-SHA1(sample.txt)= 9f13eb637a806375ce31f6e9b8449af1856302f0
[10/10/2017 13:33] seed@ubuntu:~/Desktop/assn4$ openssl dgst -sha1 -hmac "abcdefg" sample.txt
HMAC-SHA1(sample.txt)= 9ebcdf7687156ca46932c4e98b9caaf4d8e51a70
[10/10/2017 13:33] seed@ubuntu:~/Desktop/assn4$ openssl dgst -sha1 -hmac "Hello Computer Security class" sample.txt
HMAC-SHA1(sample.txt)= f444636be70eed505bc153546b41bf527d3afade
```

Sha256

```
[10/10/2017 13:33] seed@ubuntu:~/Desktop/assn4$ openssl dgst -sha256 -hmac "a" sample.txt
HMAC-SHA256(sample.txt)= 8537631016526a8d661aac96327432ea8f00a300395712ed3f75bc330a01ecce
[10/10/2017 13:34] seed@ubuntu:~/Desktop/assn4$ openssl dgst -sha256 -hmac "abcdefg" sample.txt
HMAC-SHA256(sample.txt)= 19bdb92d92c23d840bc6b4de5fba97c8854f99b10628db3ec8bd83a49982823d.
[10/10/2017 13:34] seed@ubuntu:~/Desktop/assn4$ openssl dgst -sha256 -hmac "Hello Computer Security class" sample.txt
HMAC-SHA256(sample.txt)= e487c742758deae32057b95396b9267f093bad83b0b62a9c4d17f14a41c7bba9
```

No, we don't have to use a key with a fixed size in HMAC because the HMAC algorithm is quite flexible. We can give any length of key size and the output will return a fixed hash value. But for a better security we should use a key size that is related to the algorithm's hash output. So, for md5 – 128-bit key size

Sha1 – 160-bit key size

Sha256 – 256-bit key size

Can be used. This will increase the difficulty of an attacker to crack the key.

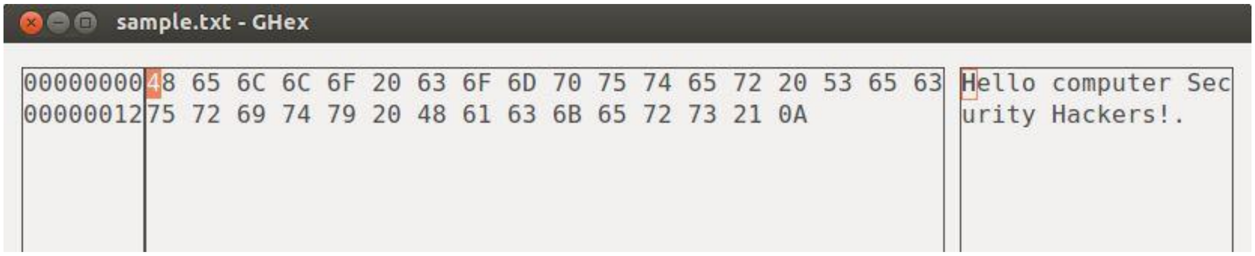
Task 3.3:

First, I created a sample.txt file with a string "Hello Computer Security Hackers".

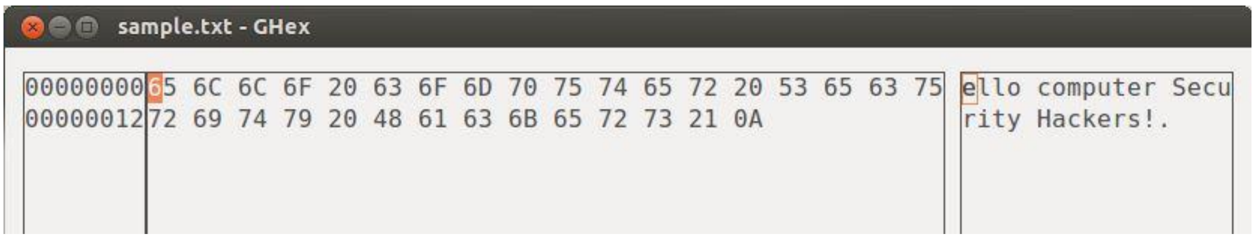
In this task, the randomness of one-way hash for Md5 and Sha1 were compared.

Md5

```
[10/10/2017 13:11] seed@ubuntu:~/Desktop/assn4$ openssl dgst -md5 sample.txt
MD5(sample.txt)= ce01cb3ab8cf5d072cf5c3f199770578
[10/10/2017 13:14] seed@ubuntu:~/Desktop/assn4$ ghex sample.txt
[10/10/2017 13:16] seed@ubuntu:~/Desktop/assn4$ openssl dgst -md5 sample.txt
MD5(sample.txt)= d611d967f7666f3c99754d09ca6328f2
```



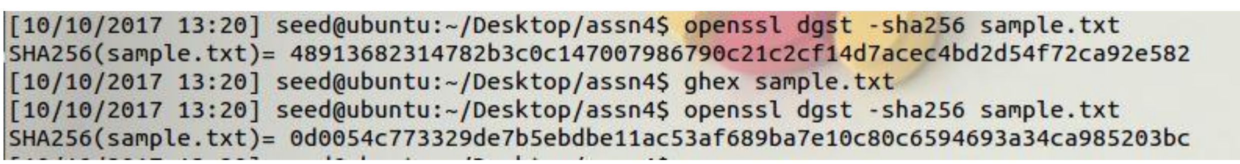
Here, the first bit 48 is flipped.

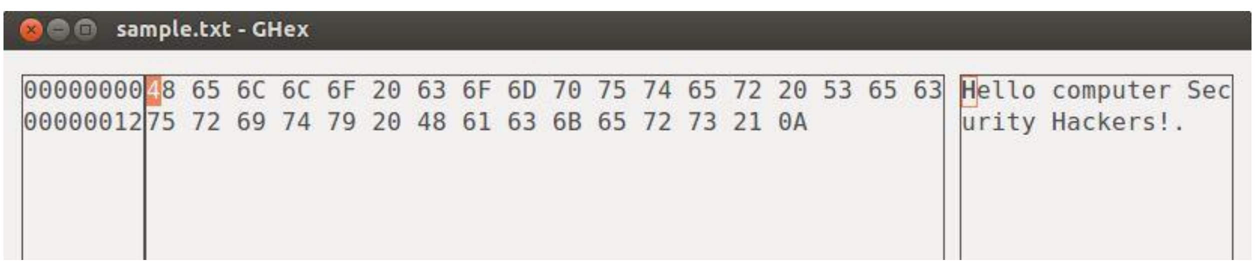


The Hash value (H1) is generated by using Md5 algorithm. Next, one bit of the input file is flipped by using the ghex editor. Again, the new hash value (H2) is generated by using the same md5 algorithm.

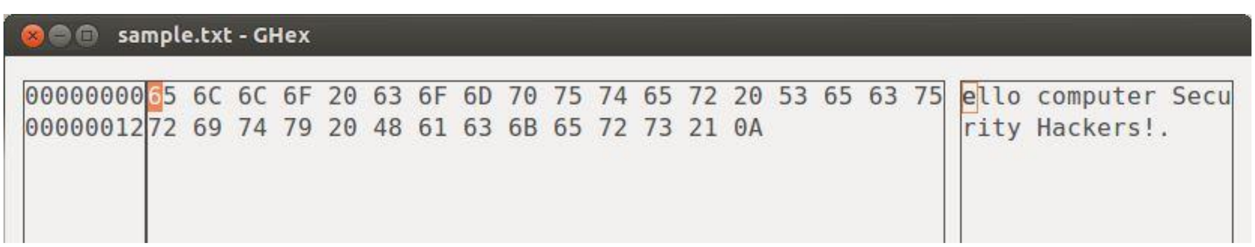
Result: The two hash values H1 and, H2 are different which means that it will generate a new hash value for every file.

Sha256





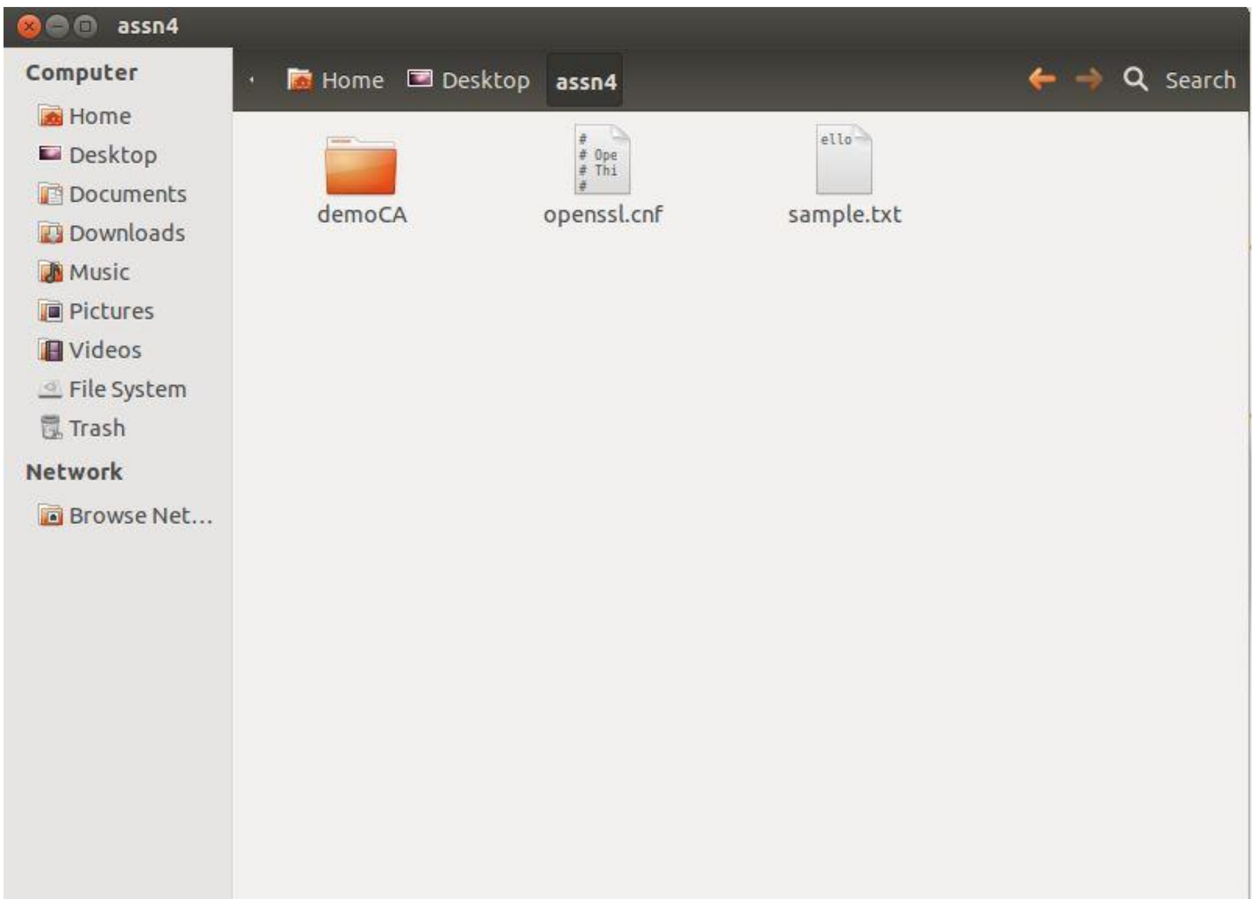
Here, the first bit 48 is flipped.



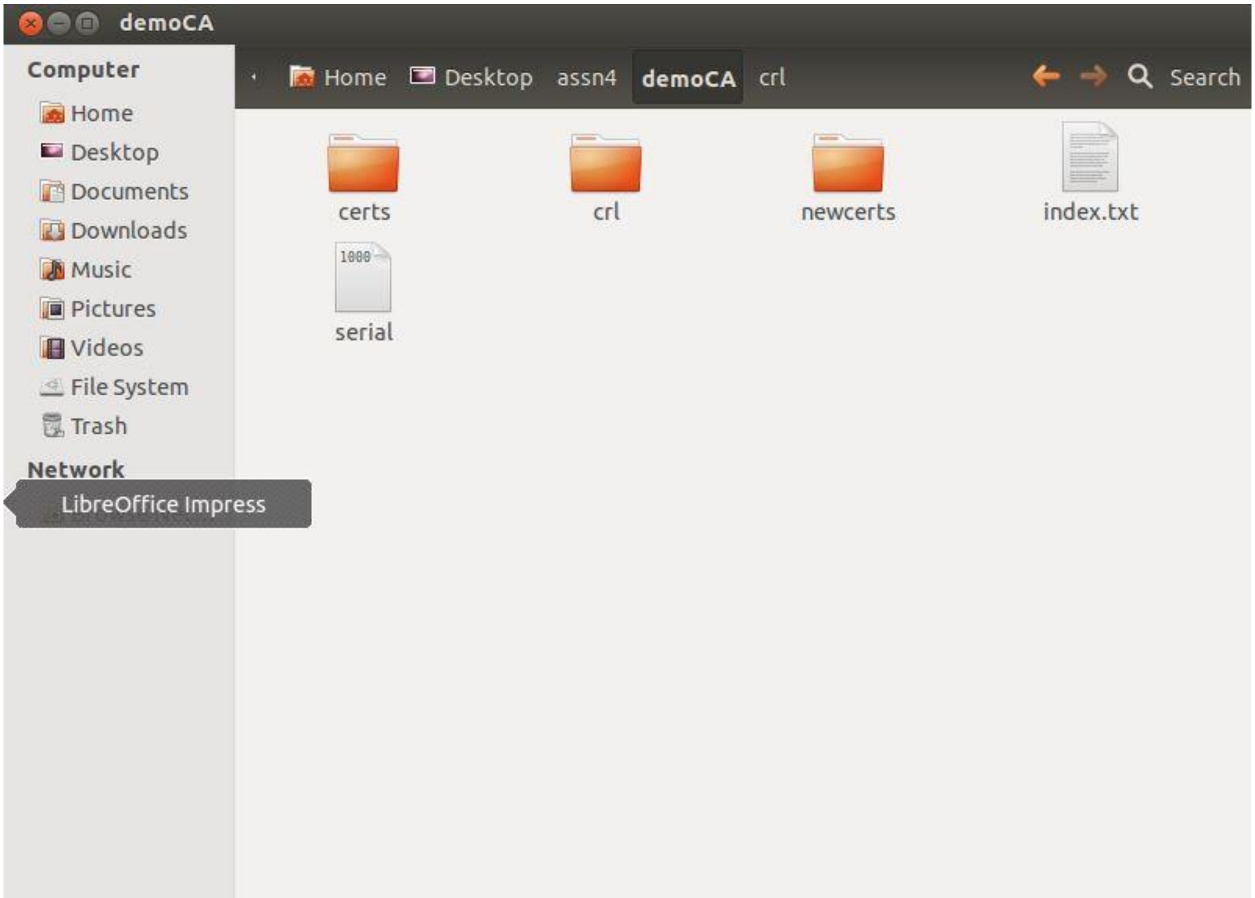
The Hash value (H1) is generated by using sha256 algorithm. Next, one bit of the input file is flipped by using the ghex editor. Again, the new hash value (H2) is generated by using the same sha256 algorithm.

Result: The two hash values H1 and H2 are different which means that it will generate a new hash value for every file.

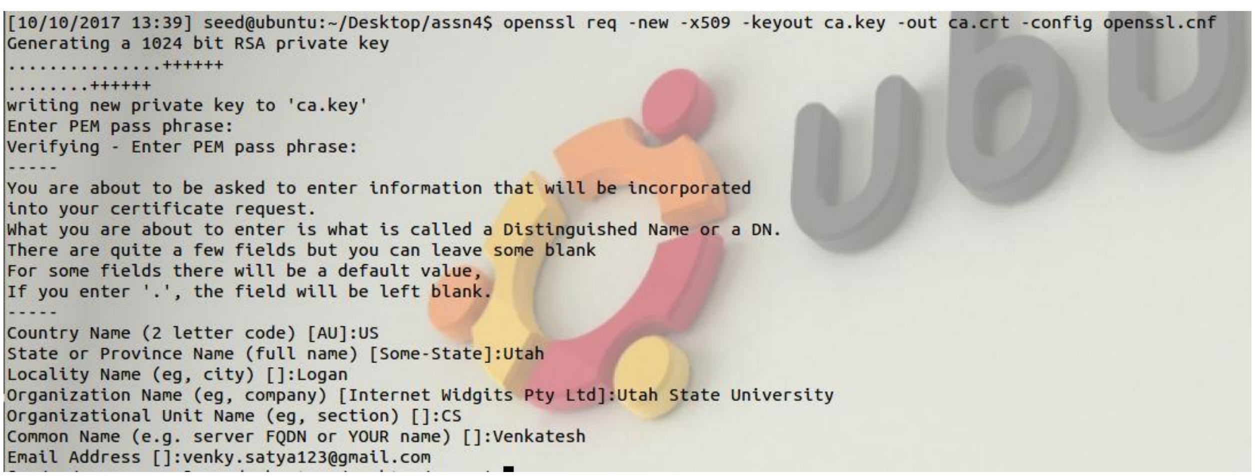
Task 4.1 : Become a certificate authority



First, I created the demoCA folder, sample.txt and imported openssl.cnf.



Then I created the certs, crl and newcerts folder in demoCA folder. Index.txt is an empty folder and serial file consists of string 1000.



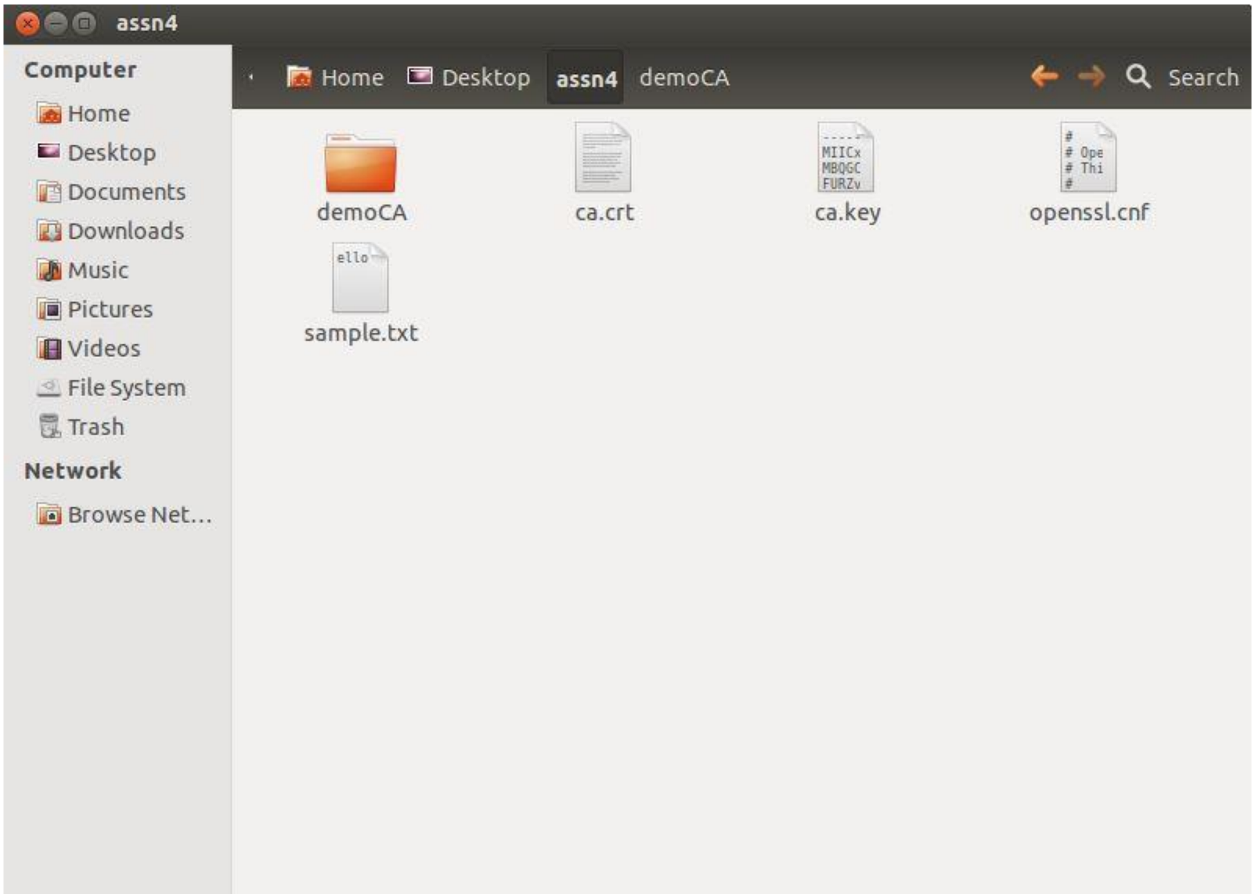
```
[10/10/2017 13:39] seed@ubuntu:~/Desktop/assn4$ openssl req -new -x509 -keyout ca.key -out ca.crt -config openssl.cnf
Generating a 1024 bit RSA private key
...............++++++
........++++++
writing new private key to 'ca.key'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You 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]:Utah
Locality Name (eg, city) []:Logan
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Utah State University
Organizational Unit Name (eg, section) []:CS
Common Name (e.g. server FQDN or YOUR name) []:Venkatesh
Email Address []:venky.satya123@gmail.com
```

Then a self-signed certificate for a CA is generated by using the above command. Then it is prompted for password, country, state, locality, Organization, Organizational unit, Common name, and Email address.

Two files were generated after executing the above command.

ca.key – contains CA's private key.

ca.crt – contains public-key certificate.



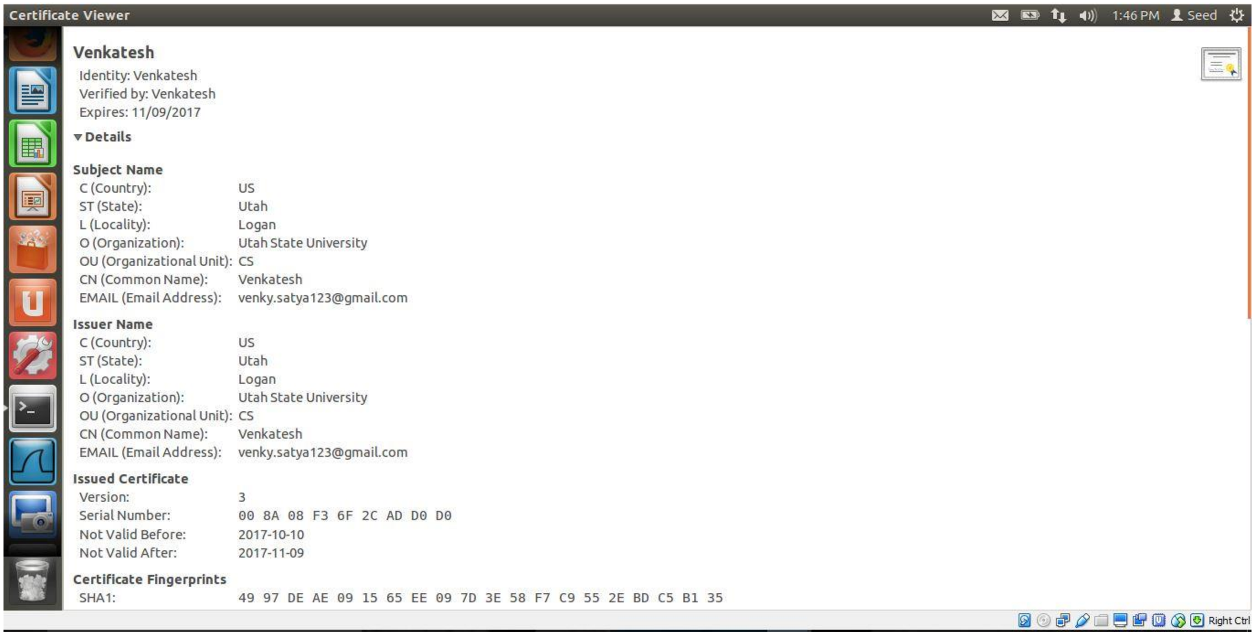
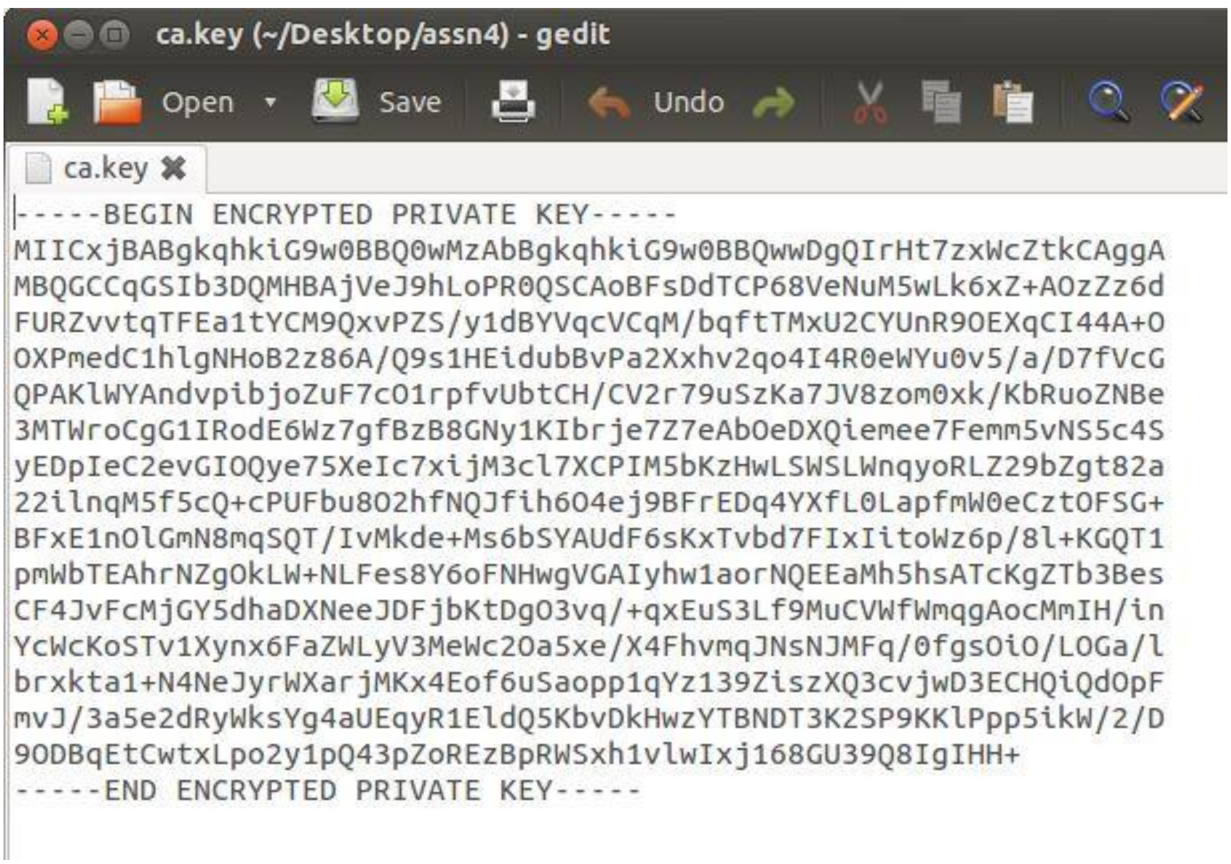


Fig: Ca.crt



-----BEGIN ENCRYPTED PRIVATE KEY-----
MIICxjBABgkqhkiG9w0BBQ0wMzAbBgkqhkiG9w0BBQwwDgQIrHt7zxWcZtkCAggA
MBQGCCqGSIb3DQMHBAjVeJ9hLoPR0QSCAoBFsDdTCP68VeNuM5wLk6xZ+AOzZz6d
FURZvvtqTFEa1tYCM9QxvPZS/y1dBYVqcVCqM/bqftTMxU2CYUnR9OEXqCI44A+O
OXPmedC1hlgNHoB2z86A/Q9s1HEidubBvPa2Xxhv2qo4I4R0eWYu0v5/a/D7fVcG
QPAKlWYAndvpibjoZuF7cO1rpfvUbtCH/CV2r79uSzKa7JV8zom0xk/KbRuoZNBe
3MTWroCgG1IRodE6Wz7gfBzB8GNy1KIbrje7Z7eAbOeDXQiemee7Femm5vNS5c4S
yEDpIeC2evGIOQye75XeIc7xijM3cl7XCPIM5bKzHwLSWSLWnqyoRLZ29bZgt82a
22ilnqM5f5cQ+cPUFbu8O2hfNQJfih6O4ej9BFrEDq4YXfL0LapfmW0eCztOFSG+
BFxE1nOlGmN8mqSQT/IvMkde+Ms6bSYAUdF6sKxTvbd7FIxIitoWz6p/8l+KGQT1
pmWbTEAhrNZgOkLW+NLFes8Y6oFNHwgVGAIyhw1aorNQEEaMh5hsATcKgZTb3Bes
CF4JvFcMjGY5dhaDXNeeJDFjbKtDgO3vq/+qxEuS3Lf9MuCVWfWmqgAocMmIH/in
YcWcKoSTv1Xynx6FaZWLyV3MeWc2Oa5xe/X4FhvmqJNsNJMFq/0fgsOiO/LOGa/l
brxkta1+N4NeJyrWXarjMKx4Eof6uSaopp1qYz139ZiszXQ3cvjwD3ECHQiQdOpF
mvJ/3a5e2dRyWksYg4aUEqyR1EldQ5KbvDkHwzYTBNDT3K2SP9KKlPpp5ikW/2/D
9ODBqEtCwtxLpo2y1pQ43pZoREzBpRWSxh1vlwIxj168GU39Q8IgIHH+
-----END ENCRYPTED PRIVATE KEY-----

Fig: Ca.key

Task 4.2: Create a Certificate for PKILabServer.com

Step 1: Generate public/ private key pair

The RSA keypair (both public key and private key) is generated by executing the following command. It is asked for a password after the command execution to encrypt the private key and the both keys were stored in the server.key file.

The second command is to see the actual content in the server.key file.

```
[10/10/2017 13:53] seed@ubuntu:~/Desktop/assn4$ openssl genrsa -aes128 -out server.key 1024
Generating RSA private key, 1024 bit long modulus
........++++++
.......................................++++++
e is 65537 (0x10001)
Enter pass phrase for server.key:
Verifying - Enter pass phrase for server.key:
[10/10/2017 13:55] seed@ubuntu:~/Desktop/assn4$ openssl rsa -in server.key -text
Enter pass phrase for server.key:
Private-Key: (1024 bit)
modulus:
    00:c0:e0:3c:b6:b1:62:0c:ad:5a:46:53:f5:a5:7f:
    fe:3c:e8:33:e4:b2:90:f8:f0:1f:96:0d:7f:7b:09:
    f5:bd:f3:89:76:80:f0:76:07:ca:78:c8:ea:42:c1:
    d1:ea:23:b9:c2:85:c1:cc:85:2e:26:63:74:b6:f7:
    10:ad:d8:41:ec:3f:29:12:c3:97:fb:56:34:0d:6b:
    da:91:c9:e4:db:d4:3c:96:99:1d:fe:f6:16:6c:1a:
    61:d0:14:b3:60:f0:f2:63:9f:6e:02:c9:52:7e:d9:
    87:d7:76:51:5f:38:17:f6:11:29:14:62:97:e2:fd:
    d7:ee:66:ca:0b:12:b6:8e:a1
publicExponent: 65537 (0x10001)
privateExponent:
    6c:a4:22:2e:b9:fd:c3:ac:45:a4:45:98:a1:6f:56:
    12:92:ac:e0:4e:20:d0:c7:d8:d2:d9:a9:8b:f1:91:
    45:3c:8f:9a:7b:88:76:c3:6e:ad:d8:65:f3:d2:5e:
    de:26:df:74:8d:89:1b:1d:8d:60:3c:37:3a:f4:31:
    a5:ea:a1:3e:69:1f:76:8c:eb:f2:5b:6a:4c:4b:73:
    fa:29:d1:f5:45:96:ae:85:97:60:ad:8c:21:54:38:
    fd:1e:0f:18:92:a4:c1:46:1b:96:a1:e3:b0:74:db:
    b7:cb:90:64:58:b8:dc:a0:ab:9c:69:f5:fb:f4:e2:
    f3:36:ee:bb:ec:ce:08:01
```

Fig: Commands to generate public and private key.

```
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-128-CBC,C7620DDD7919B7B2DEBC3E4662481235

ZJiAF6Wu787Vh4YJeimryyHMiWqPuJqWIg9umAMzgWnW7bmJakFlGOuDigo/nlnm
sxePHyAhCzc7dKOFyah8FqphN8CE9LYoriOIkvGKs1GpsAliA4VhfTga36siCgru
gMCCx3dOGwJuEtxZ9qrsrlWeNYNw+osiZtQFcjcCA0XGjq2ovGjBP3MWMHI1Kqsy
Q6ENBm+ZggCgfBslZuEhjEi24Tw66+KtH+UM/++5VnWDDa4NczeXbyOWdGWNn8Mq
d/sQQRcrbvzpSQp7P6BfTjAzJhiBZF7+RzlRmrHbd/CSpmnm+fm7sM2pz3ZGfCYd
WzNkA5mNZ8QMvxxT9MQn7ErTpqlvUGmbHEjfohgwfMXq3sW653Z3HEU/xeDcTfgm
W3Qd9x6z6IgUgilXy3Wj27wzM5eXhEQOF9vaHeMlpk6PMdBUlqeNmsYcYOovAktj
MPQiGCG7lC/MXkopMLcTc67AZ9P9yDMXOErK8lU+rcuf88g2JixaQiRHYyCY9rwU
v6k3uddBJ2FffZ1d7hl0ibwBopQ5JYNZmmnA1zb24f+rvPBZBdTx69MtAtnDOhC0
S0hby15A4afKTOvFOGWyjIKZrU+Hb6q1p2Wh77W6G5fZjcu1YXVDHviNS4mndQOm
/obtKknjjgseswSVodbeGgVaxK2V2SupNkNHt2ZODDqaNYrd8QZODGBBnaF9nXWc
Ni9vA1tJ27uhTS16Fqbf6wm+Z5beP2YP1OKyy89SWm1hfaBDbhc8CvjJ5vC2xxeg
mAk3FC52zk63LFe0dVmW1tPs1u7ob4wLlKEee7S8I9Gy6t4YA+a+t6hC4z/SQMJZ
-----END RSA PRIVATE KEY-----
```

Fig: Server.key file

Step 2: Generate a Certificate Signing request (CSR)

A certificate Signing request is generated which basically includes the company's public key. This CSR is sent to the CA, who generates a certificate for the key.

```
[10/10/2017 13:58] seed@ubuntu:~/Desktop/assn4$ openssl req -new -key server.key -out server.csr -config openssl.cnf
Enter pass phrase for server.key:
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]:Utah
Locality Name (eg, city) []:Logan
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Utah State University
Organizational Unit Name (eg, section) []:CS
Common Name (e.g. server FQDN or YOUR name) []:PKILabServer.com
Email Address []:venky.satya123@gmail.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

After executing the CSR command, it is prompted for password, country, state, locality, Organization, Organizational unit, Common name, and Email address. Then a server.csr certificate signing request file is generated.

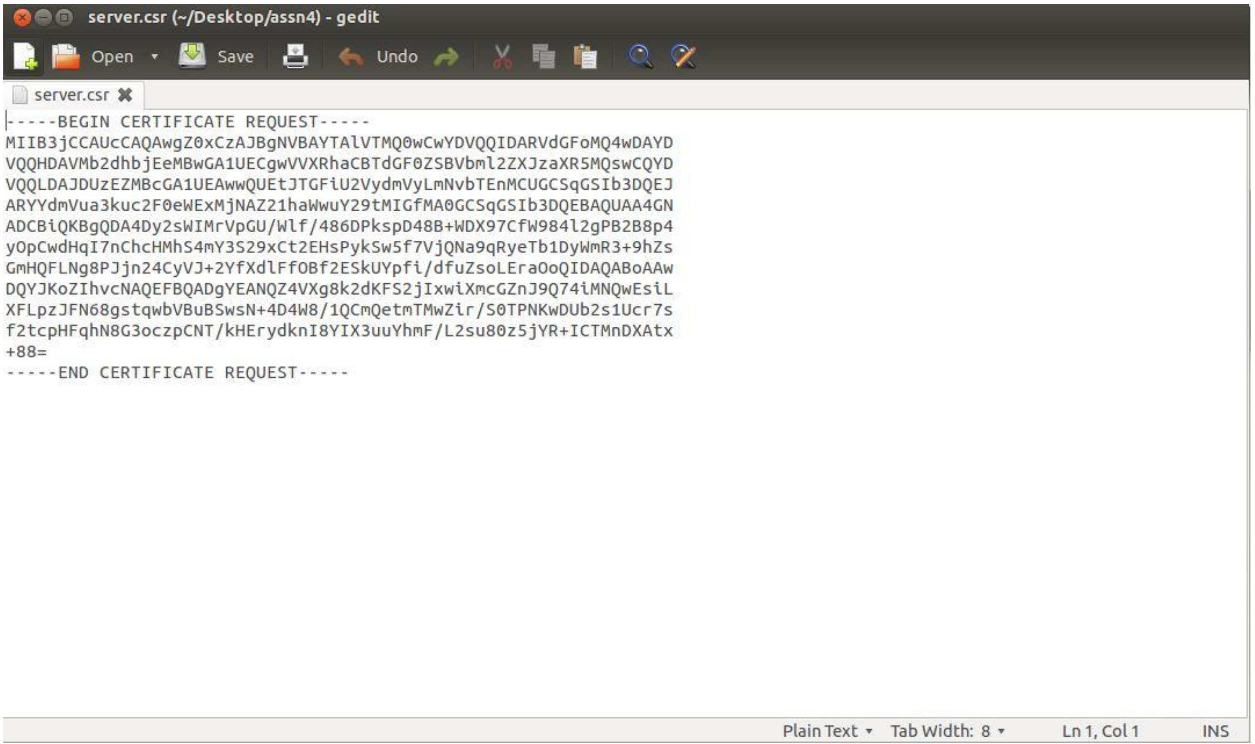


Fig: Server.csr

Step 3: Generating Certificates

The following command is executed to turn the certificate signing request (server.csr) into an X509 certificate (server.crt) using a CA's certificate (ca.crt) and CA's key (ca.key). First, it refused to generate the certificates then I modified the policy in openssl.cnf file from "policy_match" to "policy_anything". Then a certificate (server.crt) is generated.

```
[10/10/2017 14:08] seed@ubuntu:~/Desktop/assn4$ 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: 4101 (0x1005)
        Validity
            Not Before: Oct 10 21:09:17 2017 GMT
            Not After : Oct 10 21:09:17 2018 GMT
        Subject:
            countryName               = US
            stateOrProvinceName       = Utah
            localityName              = Logan
            organizationName          = Utah State University
            organizationalUnitName    = CS
            commonName                = PKILabServer.com
            emailAddress              = venky.satya123@gmail.com
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            Netscape Comment:
                OpenSSL Generated Certificate
            X509v3 Subject Key Identifier:
                C9:A6:05:77:A8:B3:8B:C2:06:C2:41:04:2B:AC:0E:CF:67:8A:1C:AF
            X509v3 Authority Key Identifier:
                keyid:76:CF:70:98:F0:3D:79:18:41:37:58:E0:4A:27:E5:14:42:5B:84:00

Certificate is to be certified until Oct 10 21:09:17 2018 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
```

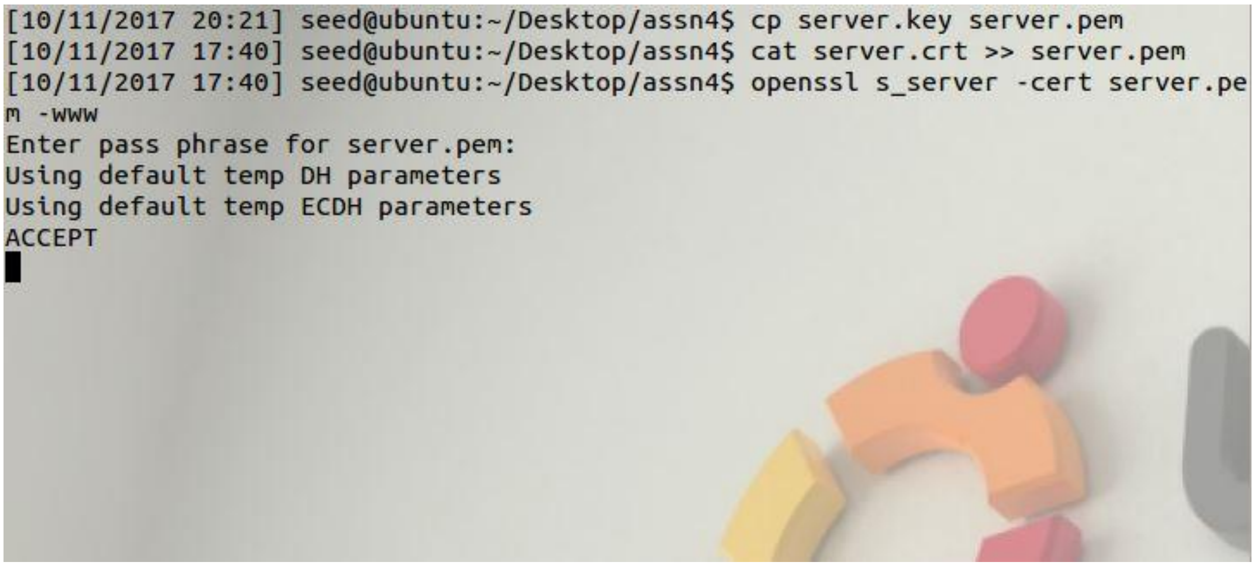
Fig: command execution for generating certificate for PKILabServer.com

```
1000.pem
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 4096 (0x1000)
    Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=US, ST=Utah, L=Logan, O=Utah State University, OU=CS, CN=Venkatesh/emailAddress=venky.satya123@gmail.com
        Validity
            Not Before: Oct 10 05:05:37 2017 GMT
            Not After : Oct 10 05:05:37 2018 GMT
        Subject: C=US, ST=Utah, L=Logan, O=Utah State University, OU=CS, CN=PKILabServer.com/emailAddress=venky.satya123@gmail.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (1024 bit)
                Modulus:
                    00:aa:35:09:f0:e4:22:d7:2d:3f:3d:fc:97:0d:d1:
                    18:bd:de:d8:ac:7b:82:c6:2b:7e:35:ff:a0:92:3c:
                    e7:21:9f:ac:e9:d5:1e:24:a1:10:cb:5c:a4:84:5c:
                    65:10:4c:36:e8:70:bc:34:9b:aa:fc:4d:eb:5b:ab:
                    67:90:af:38:3a:cc:67:dc:dc:a4:4a:5a:7e:be:89:
                    f9:d5:81:31:24:44:f5:e2:18:e0:a6:6d:b1:ce:fb:
                    21:06:66:4e:1b:38:83:cd:e1:e6:22:f5:02:fc:0b:
                    bd:b5:32:be:80:18:44:ea:a2:b5:c4:30:1d:3e:a5:
                    68:b8:21:0b:43:3b:32:a3:6b
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            Netscape Comment:
                OpenSSL Generated Certificate
            X509v3 Subject Key Identifier:
                2D:6D:30:B2:24:D7:F4:99:E8:EC:B3:5E:D5:A5:C7:D3:09:21:9F:DD
```

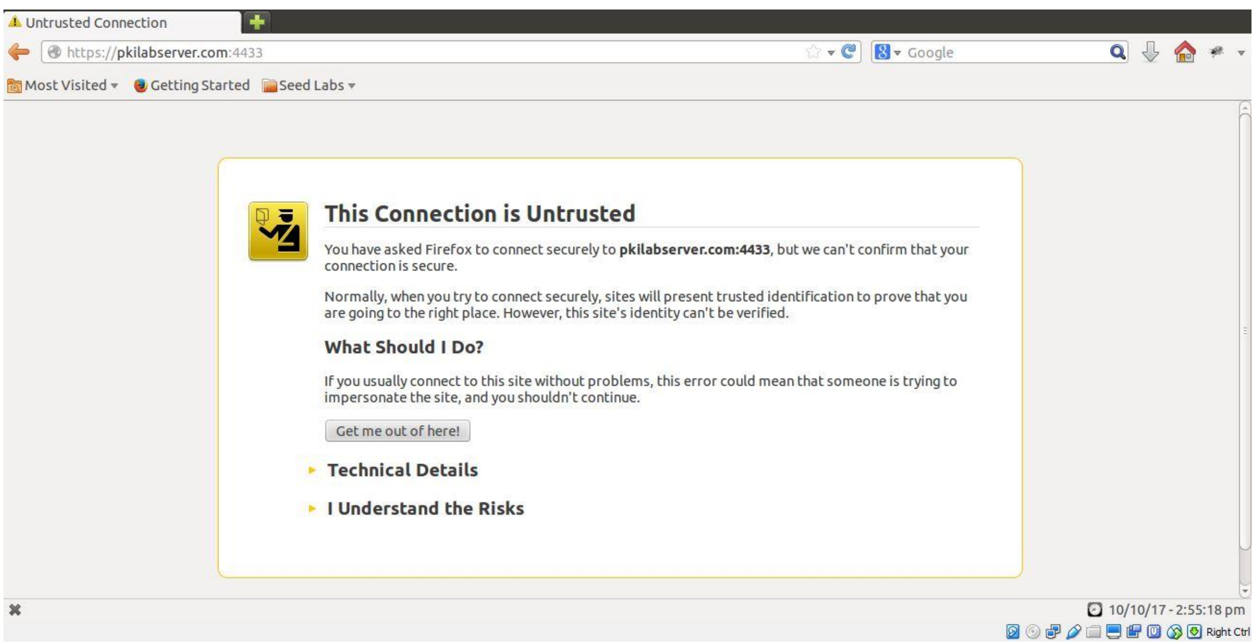
Fig: Certificate with a serial 1000 which is assigned from serial file.

Task 4.3: Use PKI for websites

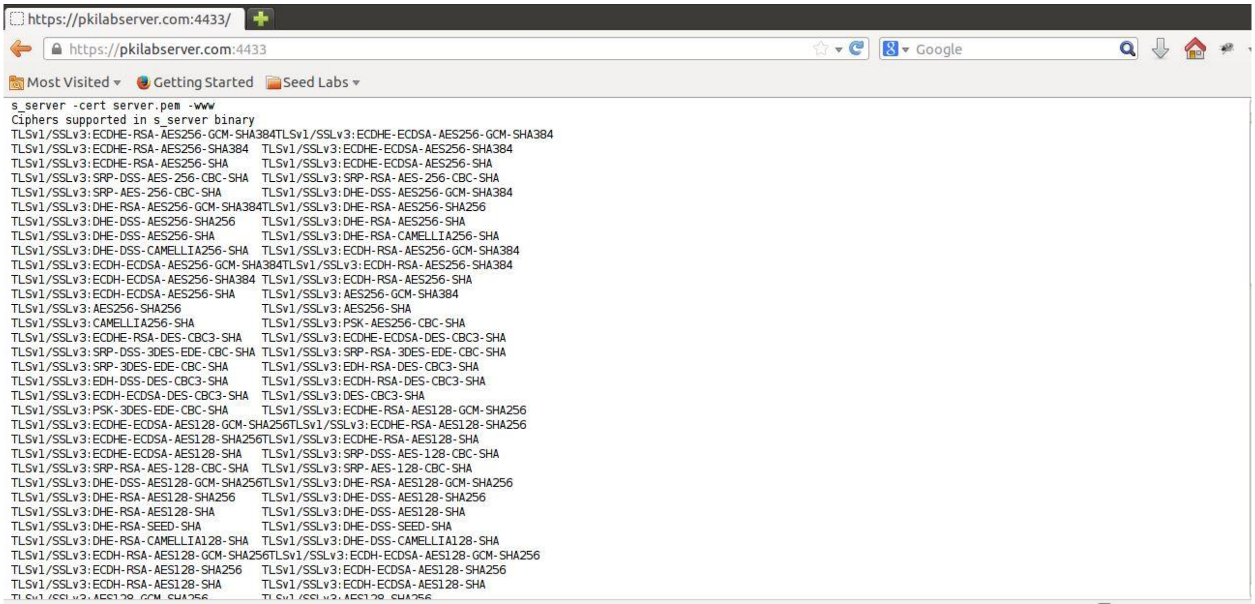
In this task, PKILabServer.com is used as our domain name. So, PKILabServer.com is added to the localhost (127.0.0.1) by using the "sudo gedit /etc/hosts" to make the computer recognize the domain name.



A secret key and certificate is combined into a single file (server.pem) to launch a simple web server using the s_server command.



Next, I tried to access the server using the url – https://PKILabserver.com . Then an error message is displayed on the web browser as I used an invalid security certificate.



Ca.crt certificate is imported into the Mozilla Firefox web browser to recognize it as a valid security certificate. Then a web page is displayed without an error.

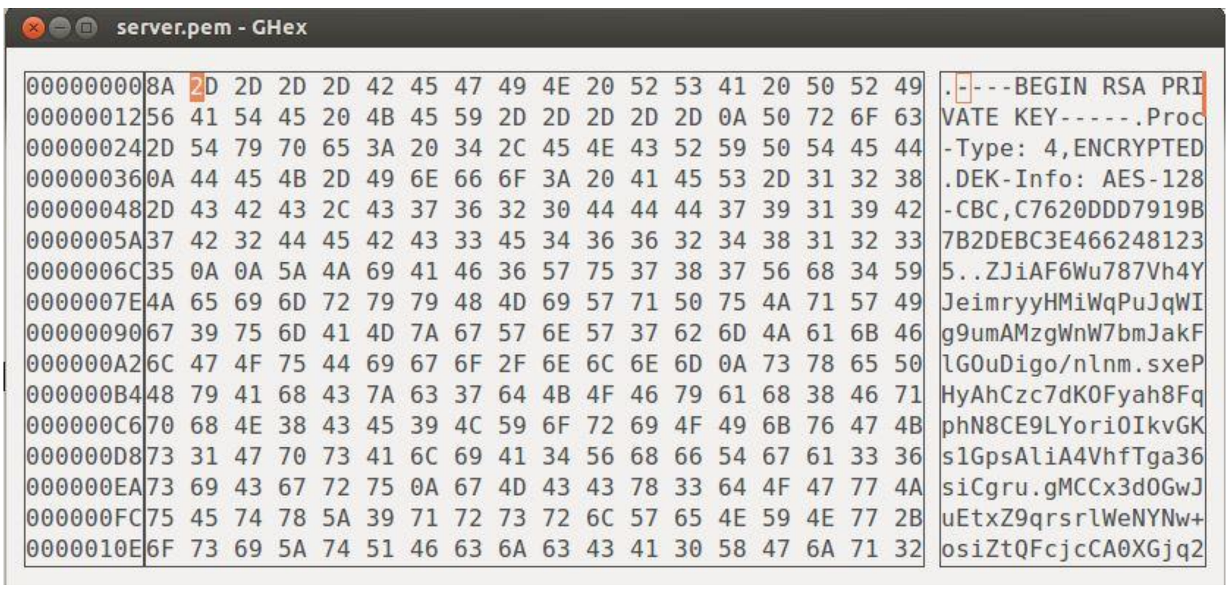


Fig: server.pem file in ghex editor

```
server.pem - GHex

000000002D 2D 2D 2D 2D 42 45 47 49 4E 20 52 53 41 20 50 52 49  -----BEGIN RSA PRI
0000001256 41 54 45 20 4B 45 59 2D 2D 2D 2D 2D 0A 50 72 6F 63  VATE KEY-----.Proc
000000242D 54 79 70 65 3A 20 34 2C 45 4E 43 52 59 50 54 45 44  -Type: 4,ENCRYPTED
000000360A 44 45 4B 2D 49 6E 66 6F 3A 20 41 45 53 2D 31 32 38  .DEK-Info: AES-128
000000482D 43 42 43 2C 43 37 36 32 30 44 44 44 37 39 31 39 42  -CBC,C7620DDD7919B
0000005A37 42 32 44 45 42 43 33 45 34 36 36 32 34 38 31 32 33  7B2DEBC3E466248123
0000006C35 0A 0A 5A 4A 69 41 46 36 57 75 37 38 37 56 68 34 59  5..ZJiAF6Wu787Vh4Y
0000007E4A 65 69 6D 72 79 79 48 4D 69 57 71 50 75 4A 71 57 49  JeimryyHMiWqPuJqWI
0000009067 39 75 6D 41 4D 7A 67 57 6E 57 37 62 6D 4A 61 6B 46  g9umAMzgWnW7bmJakF
000000A26C 47 4F 75 44 69 67 6F 2F 6E 6C 6E 6D 0A 73 78 65 50  lGOuDigo/nlnm.sxeP
000000B448 79 41 68 43 7A 63 37 64 4B 4F 46 79 61 68 38 46 71  HyAhCzc7dKOFyah8Fq
000000C670 68 4E 38 43 45 39 4C 59 6F 72 69 4F 49 6B 76 47 4B  phN8CE9LYoriOIkvGK
000000D873 31 47 70 73 41 6C 69 41 34 56 68 66 54 67 61 33 36  s1GpsAliA4VhfTga36
000000EA73 69 43 67 72 75 0A 67 4D 43 43 78 33 64 4F 47 77 4A  siCgru.gMCCx3dOGwJ
000000FC75 45 74 78 5A 39 71 72 73 72 6C 57 65 4E 59 4E 77 2B  uEtxZ9qrsrlWeNYNw+
0000010E6F 73 69 5A 74 51 46 63 6A 63 43 41 30 58 47 6A 71 32  osiZtQFcjcCA0XGjq2
```

Fig: The first bit is flipped and saved.

```
[10/10/2017 15:04] seed@ubuntu:~/Desktop/assn4$ ghex server.pem
[10/10/2017 15:06] seed@ubuntu:~/Desktop/assn4$ openssl s_server -cert server.pe
m -www
unable to load server certificate private key file
3074394312:error:0906B072:PEM routines:PEM_get_EVP_CIPHER_INFO:unsupported encry
ption:pem_lib.c:530:
```

1. A single bit of server.pem is modified by using ghex editor. Then the server is restarted and reloaded the URL. Then it doesn't work and it is giving an error (unable to load server certificate private key file). This means that if private key is changed then, server won't run as usual. So, the key is changed to its original state and restarted the server which worked fine.

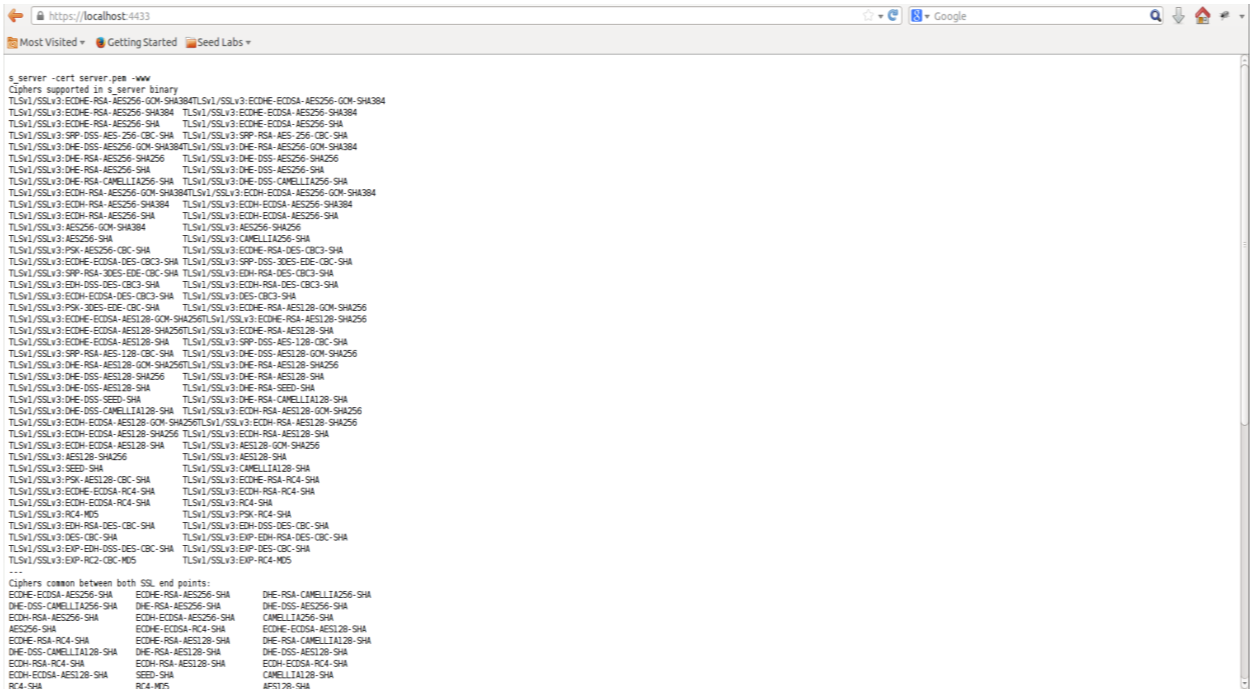


Fig: Localhost

2. As PKILabServer.com is pointing to the localhost, https://localhost:4433 is used to load the server. Then it pointed to the same web server.

Task 4.4: Establishing a TLS/SSL connection with server

Step 1: Communication between the client and server.

In this task, we implemented a TCP client and TCP server connection. For that first, the server.crt, server.cpp and ca.crt replaced in the demo file. Next, I ran make to execute the client and server files. First, I got some warnings which were resolved and then ran make again. Then the client and server executable files are generated which established the connection between client and server.

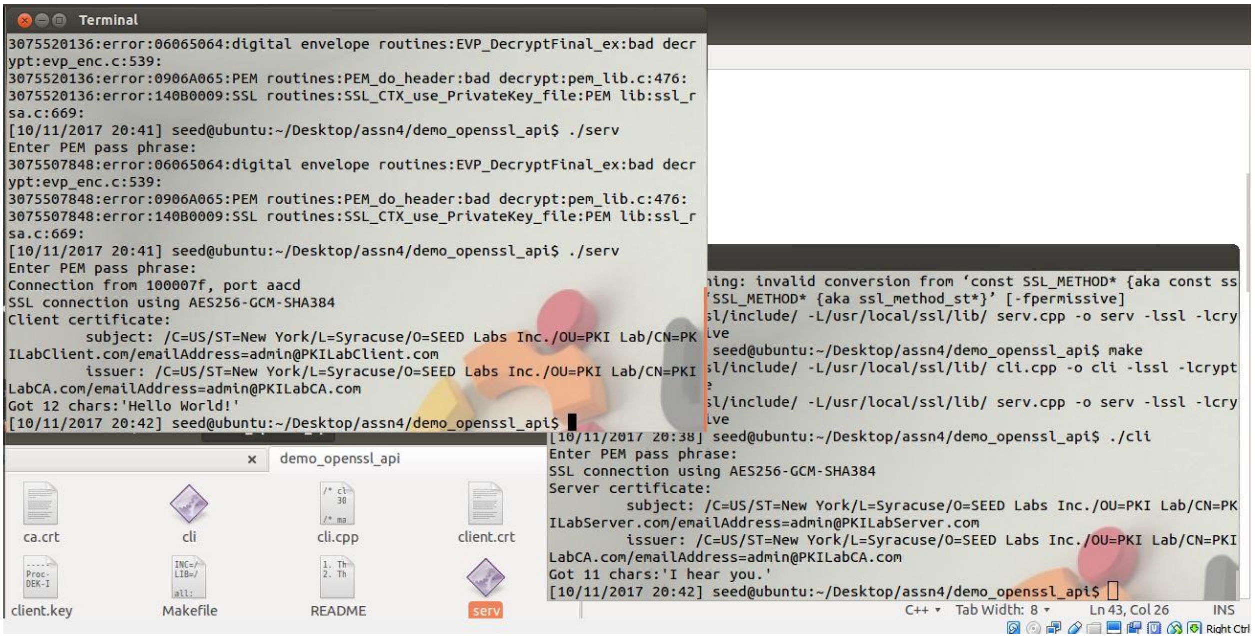


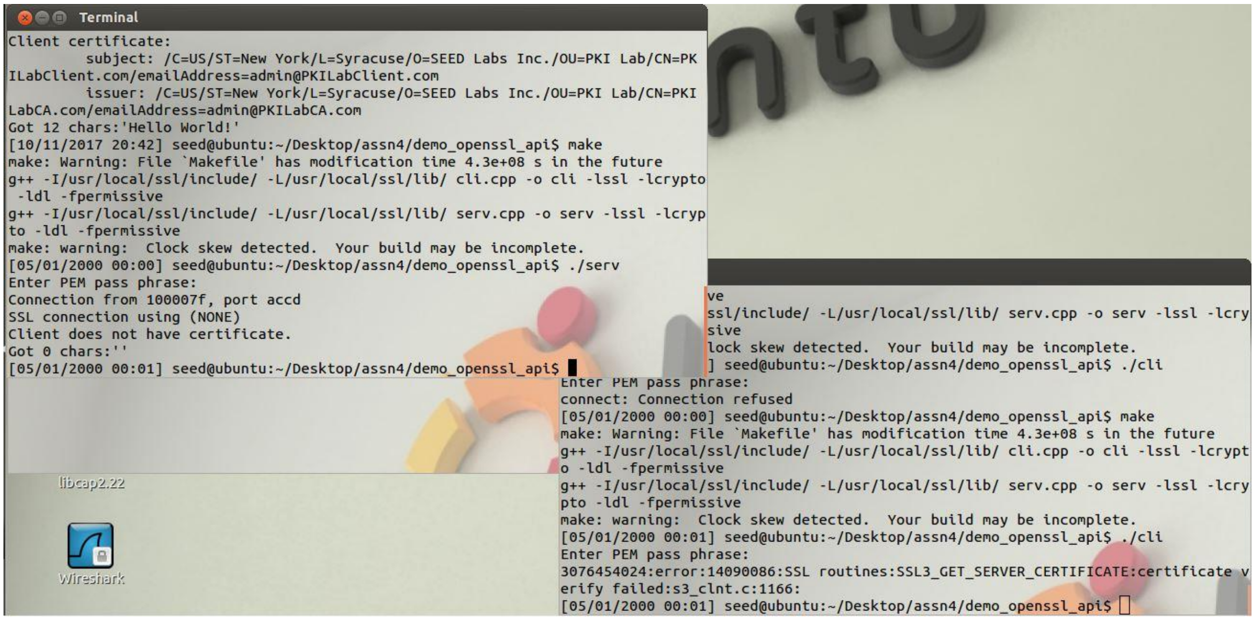
Fig: Communication between client server established.

Step 2: The effective date change

First, the time synchronization service is disabled by using “sudo service vboxadd-service stop”.

Second, the system time is changed to “1 May 2000” by using “sudo date --set="1 May 2000"”.

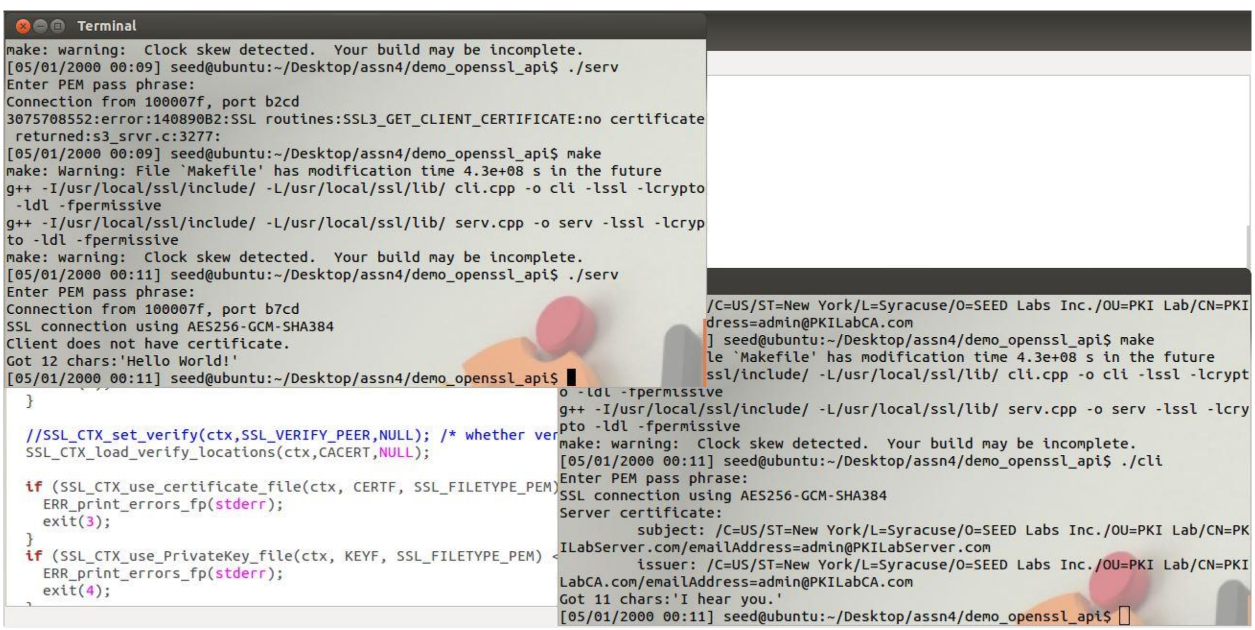
Third, the client server communication is checked where the certification verification is failed.



Step 3: part of the code responsible for key exchange.

The part of the code which is responsible for key exchange is

"SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL)". So, this line is commented and then checked the client-server communication which is a success.



Step 4: Whether the server certificate is signed by an authorized CA or not.

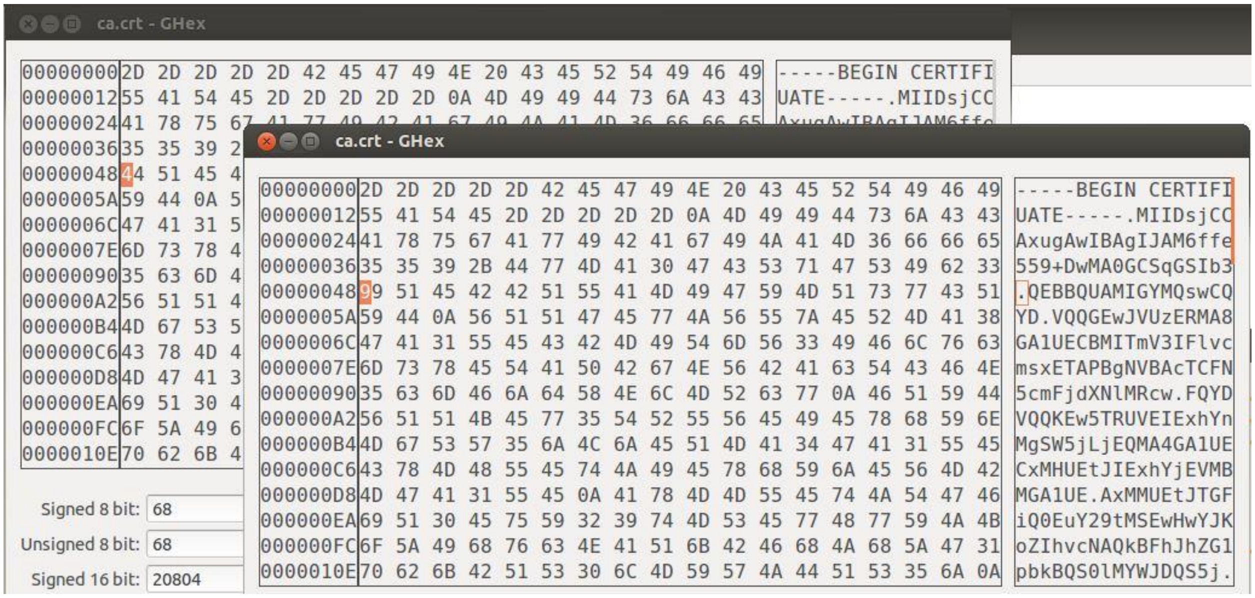
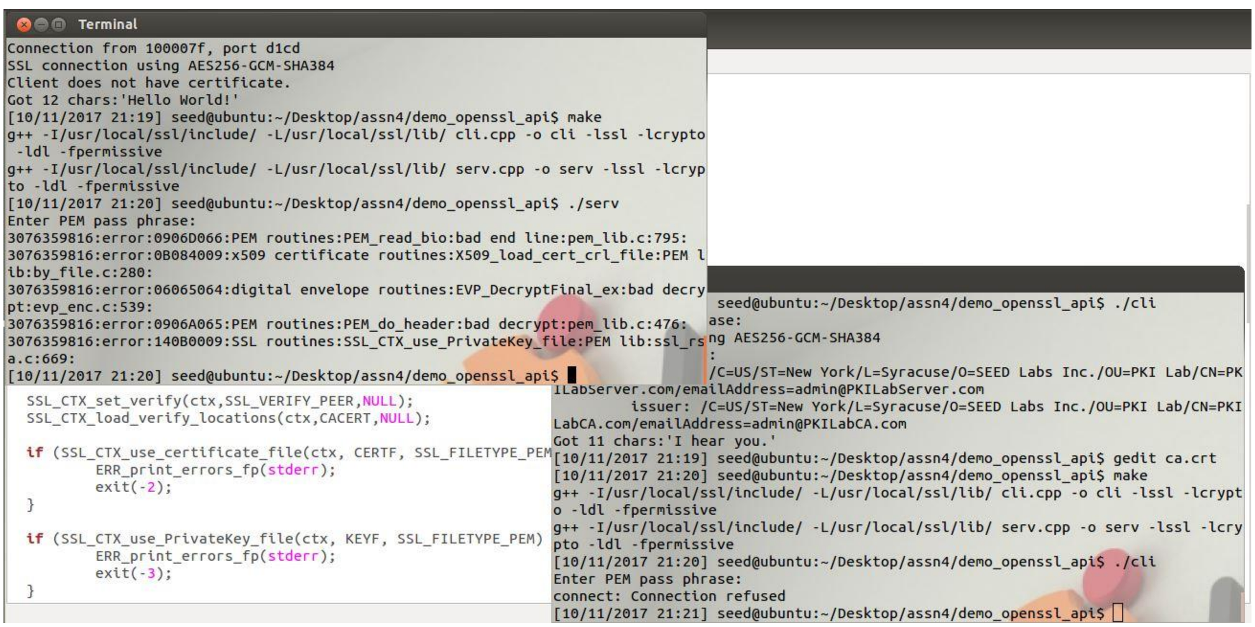


Fig: ca.crt file is modified by changing a bit.



Then the code "SSL_CTX_set_verify(ctx,SSL_VERIFY_PEER, NULL)" is uncommented and then verified the client server communication which refused the connection. So, the above line is verifying whether it is an authorized certificate or not.

Step 5: Whether the certificate belongs to the server

In this task we are checking whether the certificate belongs to the server or not.

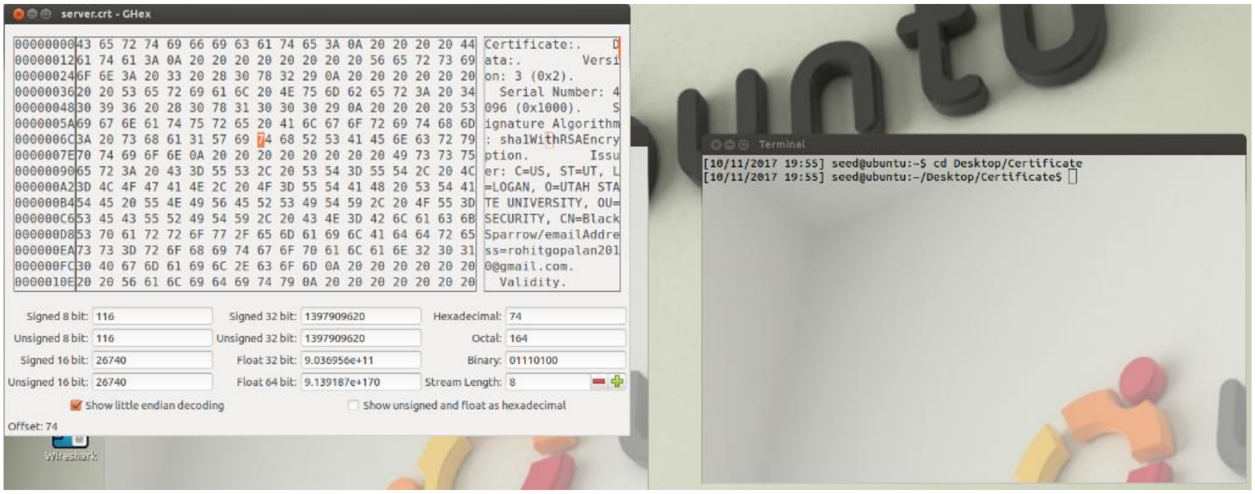


Fig: Before server.crt bit modification

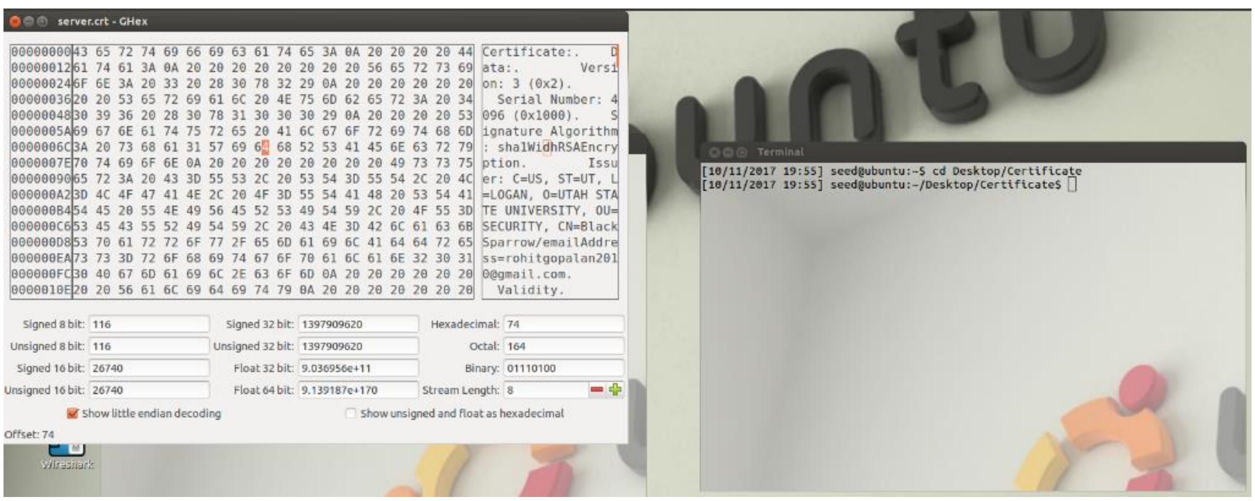


Fig: After server.crt bit modification

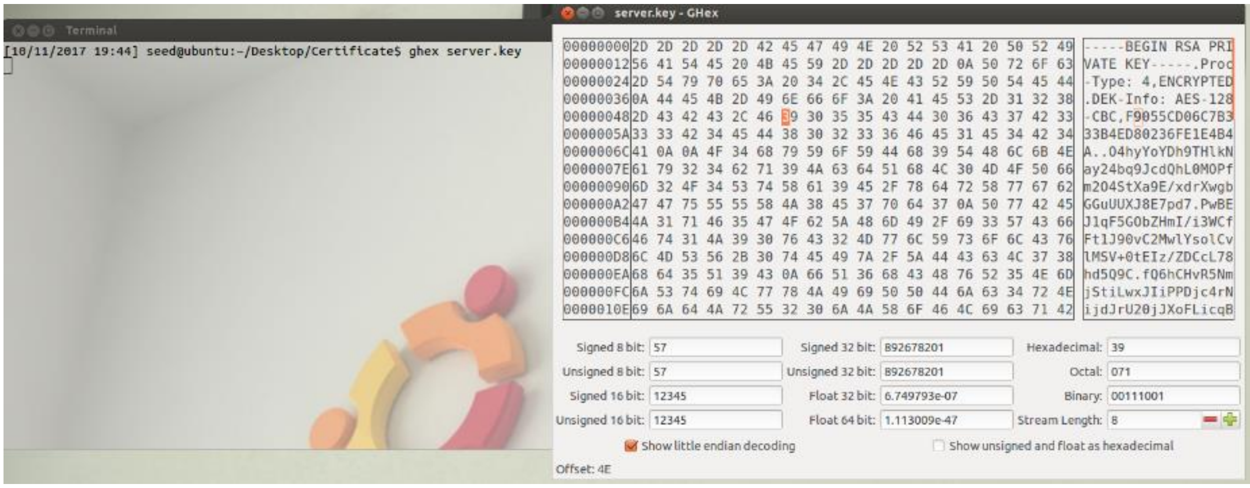


Fig: Before server.key bit modification

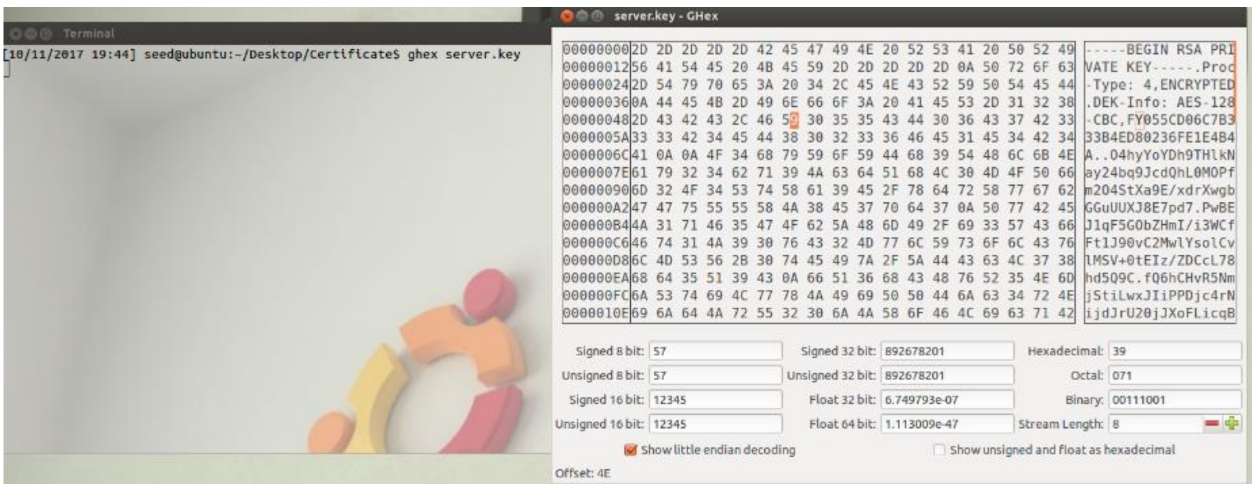


Fig: After server.key bit modification

From the above task we can see that changing bits in the server code leads to errors. So, the successful transmission is not possible. The lines of code which is responsible for this error are:

1) This line of code checking for server.crt authentication

   "SSL_CTX_use_certificate_file(ctx, CERTF, SSL_FILETYPE_PEM)"

2) This line of code checking for server.key authentication

   "SSL_CTX_use_PrivateKey_file(ctx, KEYF, SSL_FILETYPE_PEM)"

3) This line of coding checking for both private and public key matching or not

   "SSL_CTX_check_private_key(ctx)"

Step 6: What part of the code is responsible for the key exchange, i.e. for both sides to agree upon a secret key?

The below code is responsible for key exchange in serv.cpp:

```
ssl = SSL_new (ctx);
SSL_set_fd (ssl, sd);
err = SSL_accept (ssl);
```

The below code is responsible for key exchange in cli.cpp:

```
ssl = SSL_new (ctx);
SSL_set_fd (ssl, sd);
err = SSL_connect (ssl);
```

Step 7: Whether the server is indeed the machine that the client wants to talk to (as opposed to a spoofed machine).

```
memset (&sa, '\0', sizeof(sa));
sa.sin_family      = AF_INET;
sa.sin_addr.s_addr = inet_addr ("127.0.0.1");   /* Server IP */
sa.sin_port        = htons     (1111);          /* Server Port number */

err = connect(sd, (struct sockaddr*) &sa,
              sizeof(sa));                    CHK_ERR(err, "connect");
```

The above lines of code are responsible ensuring that the client is talking to the correct server.

Step 8: The provided sample code for the server also verifies the client's certificate. We do not need this, please remove this part of code, and show us what changes you made in the server-side code.

This is the code in the server that verifies the client's certificate.

"SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL)".

Task 4.5: Performance Comparison: RSA vs AES

```
[10/10/2017 22:43] seed@ubuntu:~/Desktop/assn4$ openssl genrsa -out private.pem 1024
Generating RSA private key, 1024 bit long modulus
.....++++++
.............................++++++
e is 65537 (0x10001)
[10/10/2017 22:47] seed@ubuntu:~/Desktop/assn4$ openssl rsa -in private.pem -out public.pem -outform PEM -pubout
writing RSA key
[10/10/2017 22:47] seed@ubuntu:~/Desktop/assn4$ echo "too many secrets" > message.txt
[10/10/2017 22:48] seed@ubuntu:~/Desktop/assn4$ openssl rsautl -encrypt -inkey public.pem -pubin -in message.txt -out message_enc.txt
[10/10/2017 22:48] seed@ubuntu:~/Desktop/assn4$ openssl rsautl -decrypt -inkey private.pem -in message_enc.txt -out decrypted.txt
[10/10/2017 22:48] seed@ubuntu:~/Desktop/assn4$ cat decrypted.txt
too many secrets
[10/10/2017 22:48] seed@ubuntu:~/Desktop/assn4$ cat message_enc.txt
J†]�[◆◆◆WH◆◆&◆◆◆M◆&�M◆◆◆◆~a◆H9x◆◆◆◆�s^◆^◆=nKI◆◆◆◆)
                    6y◆◆hx◆I◆v◆@"
                                  ��B◆#bN◆◆◆◆◆◆�◆◆�`◆r◆{n◆◆◆pO◆[10/10/2017 22:48] seed@ubuntu:~/Desktop/assn4$
```

First, I created a message.txt file with a string “too many secrets”.

Next, a 1024-bit RSA public/private key pair is generated. Message.txt file is encrypted by using the public key and saved the output in message_enc.txt. You can see the encrypted message in the above figure. Then message_enc.txt file is decrypted by using the private key and saved the output in decrypted.txt.

Encryption:

```
[10/11/2017 18:45] seed@ubuntu:~/Desktop/assn4$ time openssl enc -aes-256-cbc -e -in message.txt -out message_enc.txt -k 123456 -iv 123456

real    0m0.003s
user    0m0.000s
sys     0m0.000s
[10/11/2017 18:45] seed@ubuntu:~/Desktop/assn4$ time openssl enc -aes-256-cbc -e -in message.txt -out message_enc.txt -k 123456 -iv 123456

real    0m0.004s
user    0m0.000s
sys     0m0.000s
[10/11/2017 18:46] seed@ubuntu:~/Desktop/assn4$ time openssl enc -aes-256-cbc -e -in message.txt -out message_enc.txt -k 123456 -iv 123456

real    0m0.004s
user    0m0.000s
sys     0m0.000s
```

The message.txt file is encrypted multiple times using aes-256-cbc algorithm to find out the average time taken by the algorithm for encryption. The average time taken by the aes algorithm is 0.033s.

```
[10/11/2017 18:47] seed@ubuntu:~/Desktop/assn4$ time openssl rsautl -encrypt -inkey public.pem -pubin -in message.txt -out message_enc.txt

real    0m0.004s
user    0m0.000s
sys     0m0.000s
[10/11/2017 18:47] seed@ubuntu:~/Desktop/assn4$ time openssl rsautl -encrypt -inkey public.pem -pubin -in message.txt -out message_enc.txt

real    0m0.004s
user    0m0.000s
sys     0m0.000s
[10/11/2017 18:47] seed@ubuntu:~/Desktop/assn4$ time openssl rsautl -encrypt -inkey public.pem -pubin -in message.txt -out message_enc.txt

real    0m0.004s
user    0m0.000s
sys     0m0.000s
```

The message.txt file is encrypted multiple times using RSA algorithm to find out the average time taken by the algorithm for encryption. The average time taken by the aes algorithm is 0.04s.

So, here we can observe that aes is running faster than RSA.

```
[10/10/2017 22:40] seed@ubuntu:~/Desktop/assn4$ openssl speed rsa
Doing 512 bit private rsa's for 10s: 61751 512 bit private RSA's in 9.75s
Doing 512 bit public rsa's for 10s: 728737 512 bit public RSA's in 9.76s
Doing 1024 bit private rsa's for 10s: 10595 1024 bit private RSA's in 9.75s
Doing 1024 bit public rsa's for 10s: 220594 1024 bit public RSA's in 9.76s
Doing 2048 bit private rsa's for 10s: 1563 2048 bit private RSA's in 9.76s
Doing 2048 bit public rsa's for 10s: 59185 2048 bit public RSA's in 9.76s
Doing 4096 bit private rsa's for 10s: 225 4096 bit private RSA's in 9.80s
Doing 4096 bit public rsa's for 10s: 14900 4096 bit public RSA's in 9.77s
OpenSSL 1.0.1 14 Mar 2012
built on: Mon Jan 30 20:36:37 UTC 2017
options:bn(64,32) rc4(8x,mmx) des(ptr,risc1,16,long) aes(partial) blowfish(idx)
compiler: cc -fPIC -DOPENSSL_PIC -DZLIB -DOPENSSL_THREADS -D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H -DL_ENDIAN -DTERMIO -g -O2 -fstack-protector -
-param=ssp-buffer-size=4 -Wformat -Wformat-security -Werror=format-security -D_FORTIFY_SOURCE=2 -Wl,-Bsymbolic-functions -Wl,-z,relro -Wa,--noex
ecstack -Wall -DOPENSSL_NO_TLS1_2_CLIENT -DOPENSSL_BN_ASM_PART_WORDS -DOPENSSL_IA32_SSE2 -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_GF2m -DSHA1_ASM
-DSHA256_ASM -DSHA512_ASM -DMD5_ASM -DRMD160_ASM -DAES_ASM -DVPAES_ASM -DWHIRLPOOL_ASM -DGHASH_ASM
                  sign    verify    sign/s verify/s
rsa  512 bits 0.000158s 0.000013s   6333.4  74665.7
rsa 1024 bits 0.000920s 0.000044s   1086.7  22601.8
rsa 2048 bits 0.006244s 0.000165s    160.1   6064.0
rsa 4096 bits 0.043556s 0.000656s     23.0   1525.1
[10/10/2017 22:42] seed@ubuntu:~/Desktop/assn4$ openssl speed aes
Doing aes-128 cbc for 3s on 16 size blocks: 15150792 aes-128 cbc's in 2.91s
Doing aes-128 cbc for 3s on 64 size blocks: 4269678 aes-128 cbc's in 2.92s
Doing aes-128 cbc for 3s on 256 size blocks: 1087814 aes-128 cbc's in 2.92s
Doing aes-128 cbc for 3s on 1024 size blocks: 699111 aes-128 cbc's in 2.92s
Doing aes-128 cbc for 3s on 8192 size blocks: 88523 aes-128 cbc's in 2.93s
Doing aes-192 cbc for 3s on 16 size blocks: 13184652 aes-192 cbc's in 2.91s
Doing aes-192 cbc for 3s on 64 size blocks: 3683070 aes-192 cbc's in 2.92s
Doing aes-192 cbc for 3s on 256 size blocks: 903391 aes-192 cbc's in 2.91s
Doing aes-192 cbc for 3s on 1024 size blocks: 601901 aes-192 cbc's in 2.93s
Doing aes-192 cbc for 3s on 8192 size blocks: 74349 aes-192 cbc's in 2.92s
Doing aes-256 cbc for 3s on 16 size blocks: 11178203 aes-256 cbc's in 2.92s
Doing aes-256 cbc for 3s on 64 size blocks: 2965999 aes-256 cbc's in 2.86s
Doing aes-256 cbc for 3s on 256 size blocks: 773833 aes-256 cbc's in 2.93s
```

“Openssl speed rsa” and “openssl speed aes” are used to find the speed of rsa and aes for different size blocks. In this Image we can observe that aes is taking an average of 2.9 seconds and RSA is taking 9.7seconds on an average, for 1024 bits. So, aes is running faster than RSA.

The above readings are resembling the readings in this figure.

Task 4.6: Create Digital Signature

First, I created a example.txt file with a string “too many secrets are there”.

Second, RSA public/private key pair is generated.

a) Creating public/private key pair:

   openssl genrsa -out private.pem 1024

b) Extracting public key:

   openssl rsa -in private.pem -out public.pem -outform PEM -pubout

```
[10/10/2017 23:03] seed@ubuntu:~/Desktop/assn4$ openssl genrsa -out private.pem 1024
Generating RSA private key, 1024 bit long modulus
......++++++
......................++++++
e is 65537 (0x10001)
[10/10/2017 23:07] seed@ubuntu:~/Desktop/assn4$ openssl rsa -in private.pem -out public.pem -outform PEM -pubout
writing RSA key
[10/10/2017 23:07] seed@ubuntu:~/Desktop/assn4$ openssl dgst -sha256 -sign private.pem -out example.sha256 example.txt
[10/10/2017 23:07] seed@ubuntu:~/Desktop/assn4$ openssl dgst -sha256 -verify public.pem -signature example.sha256 example.txt
Verified OK
[10/10/2017 23:07] seed@ubuntu:~/Desktop/assn4$ gedit example.txt
[10/10/2017 23:09] seed@ubuntu:~/Desktop/assn4$ openssl dgst -sha256 -verify public.pem -signature example.sha256 example.txt
Verification Failure
```

1. Signed the SHA256 hash of example.txt and saved the output in example.sha256 by using the private key.
   openssl dgst -sha256 -sign private.pem -out example.sha256 example.txt

2. Verified the digital signature in example.sha256 by using the public key. The verification is succeeded.
   openssl dgst -sha256 -verify public.pem -signature example.sha256 example.txt

3. Example.txt is modifies slightly and verified the digital signature. Then the verification is failed.
   openssl dgst -sha256 -verify public.pem -signature example.sha256 example.txt

From the above task we can observe that the digital signature is failed to recognize with a change of even on bit of file. So, the Integrity of a file is maintained. When we use a digital signature in a file, it captures information regarding the metadata. This protects the validity of the signature.