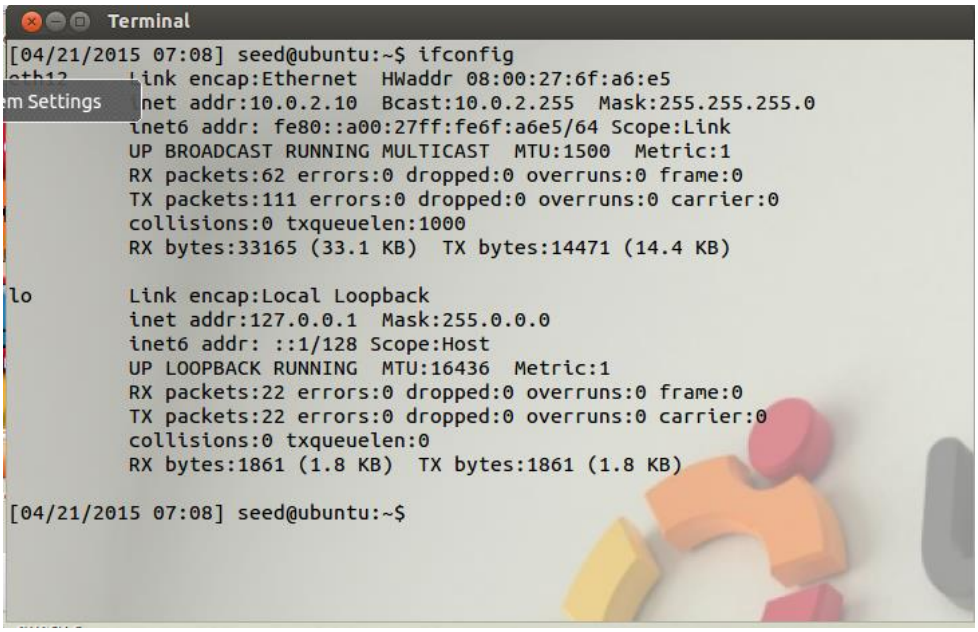
Crypto Lab III – Public-Key Cryptography and PKI

-Bharath Darapu

2 Lab Environment

For this lab we will be using one VM to perform the tasks.



VM_1 (10.0.2.10)

Installing OpenSSL:

OpenSSL is already downloaded onto our pre-built VMs. We need to configure it using the following commands:

$ cd /seed/openssl-1.0.1
$ ./config
$ make
$ make test
$ sudo make install

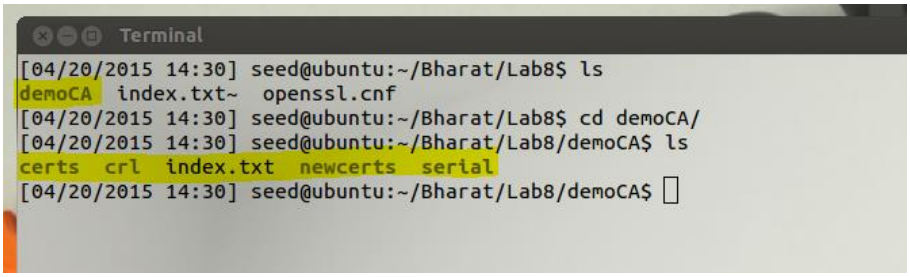
If we have /usr/local/ssl then our configuration ran successfully.

3 Lab Tasks

3.1 Task 1: Become a Certificate Authority (CA)

The Configuration File openssl.conf:

In order to use OpenSSL to create certificates, we need a configuration file. We will copy the /etc/ssl/openssl.cnf to /Bharat/Lab8/openssl.cnf. Next we create the required subdirectories as per the CA_Default section. After creating the sub directories our structure will look like:



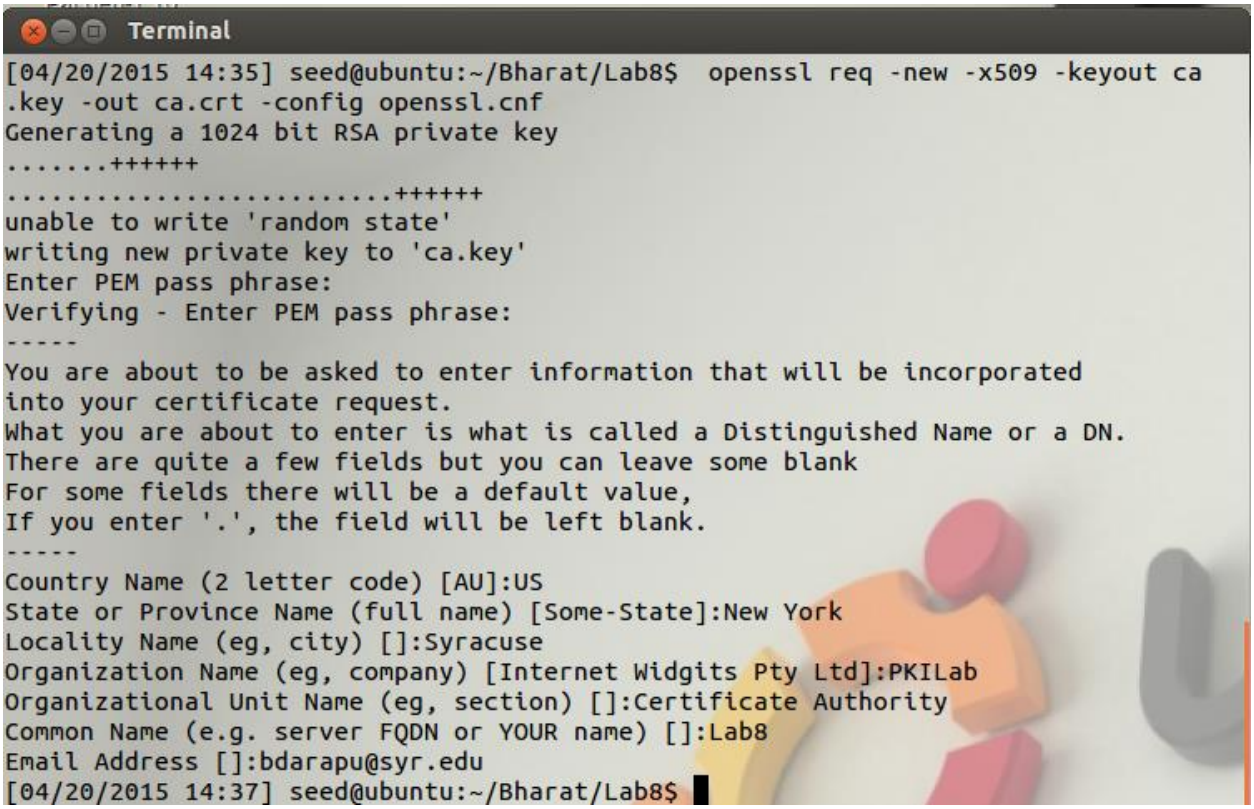
sub directories created as per CA_Default section

Directories: certs,crl and newcerts.
Files: index.txt and serial

Certificate Authority (CA):

Next we need to create a self-signed certificate for the CA. Which we can generate by using the following command: '*openssl req -new -x509 -keyout ca.key -out ca.crt -config openssl.cnf*'. I

t will ask for a few details and can be seen as follows:

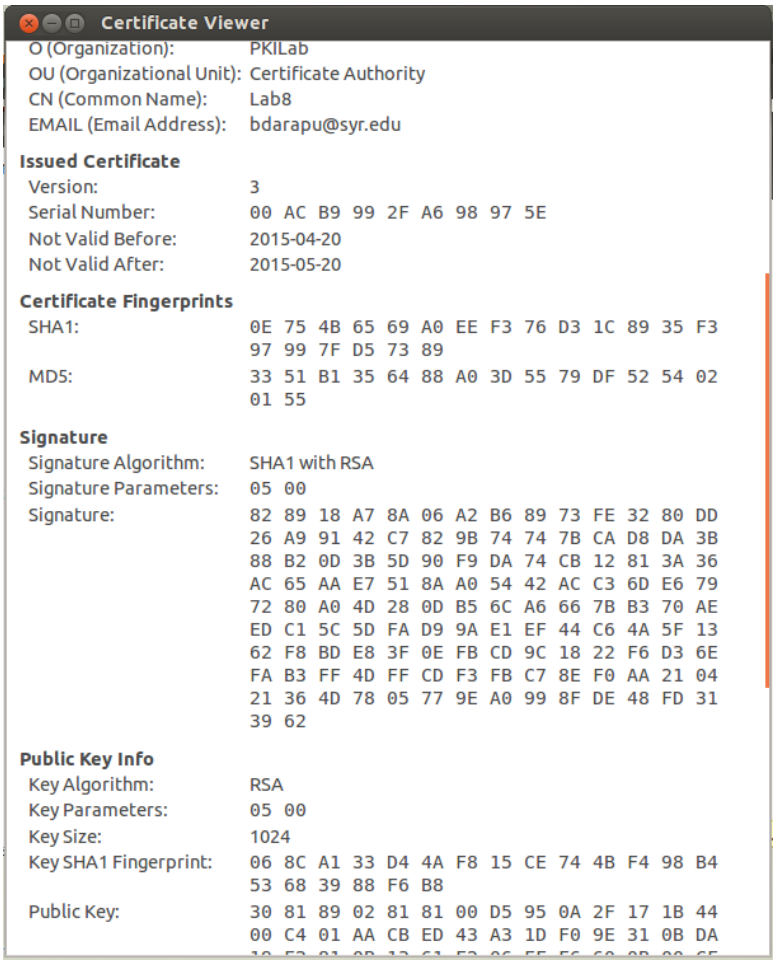


generating root certificate.

Here two new files are created with the following content:

generated ca.crt

when we actually open the file with a certificate viewer we can see the following:



ca.crt opened with a certificate viewer

Another file called ca.key is generated with content as:

ca.key contents

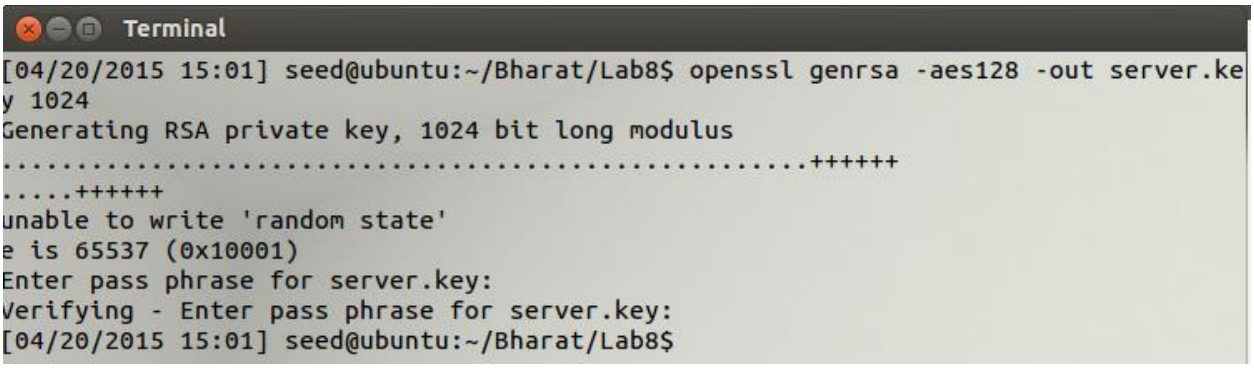
The file ca.key contains the CA's private key, while ca.crt contains the public-key among others.

3.2 Task 2: Create a Certificate for PKILabServer.com

Step 1: Generate public/private key pair:

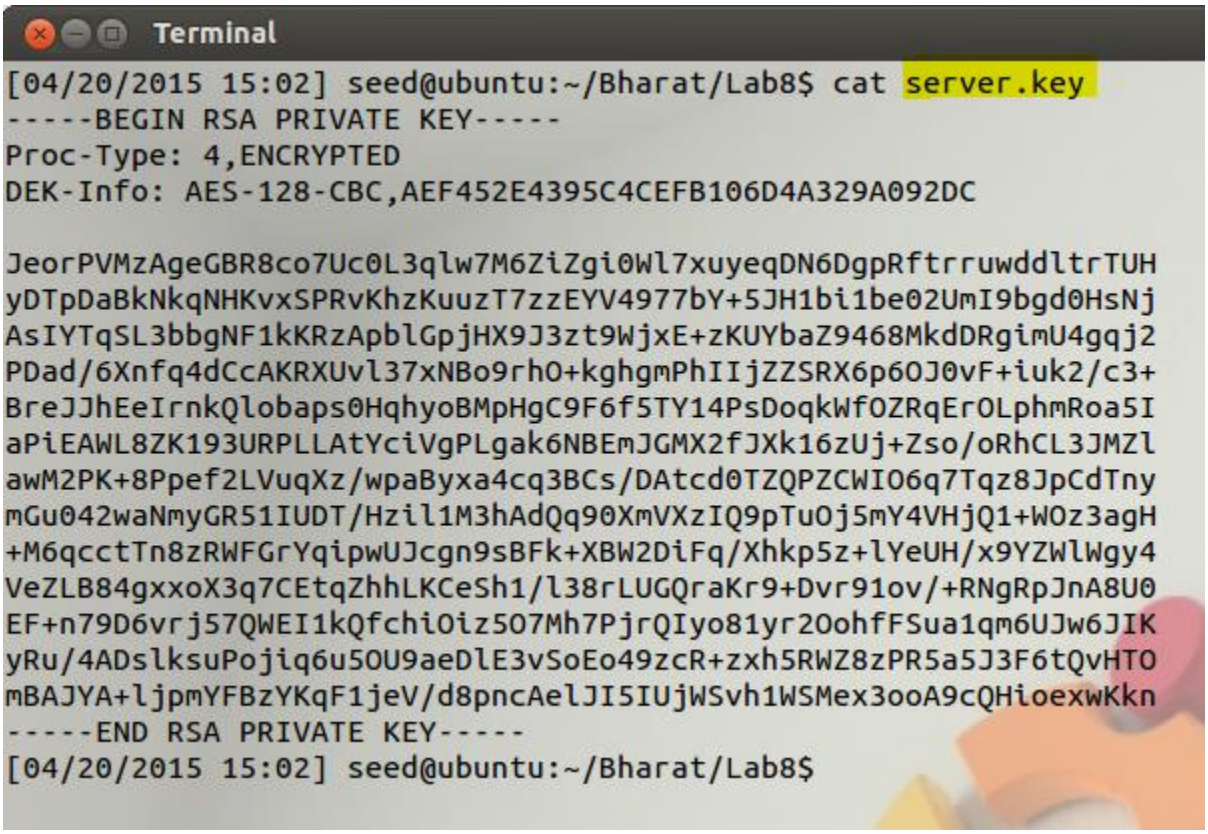
we will use the following command:

'*openssl genrsa -aes128 -out server.key 1024*'



Generate public/private key pair

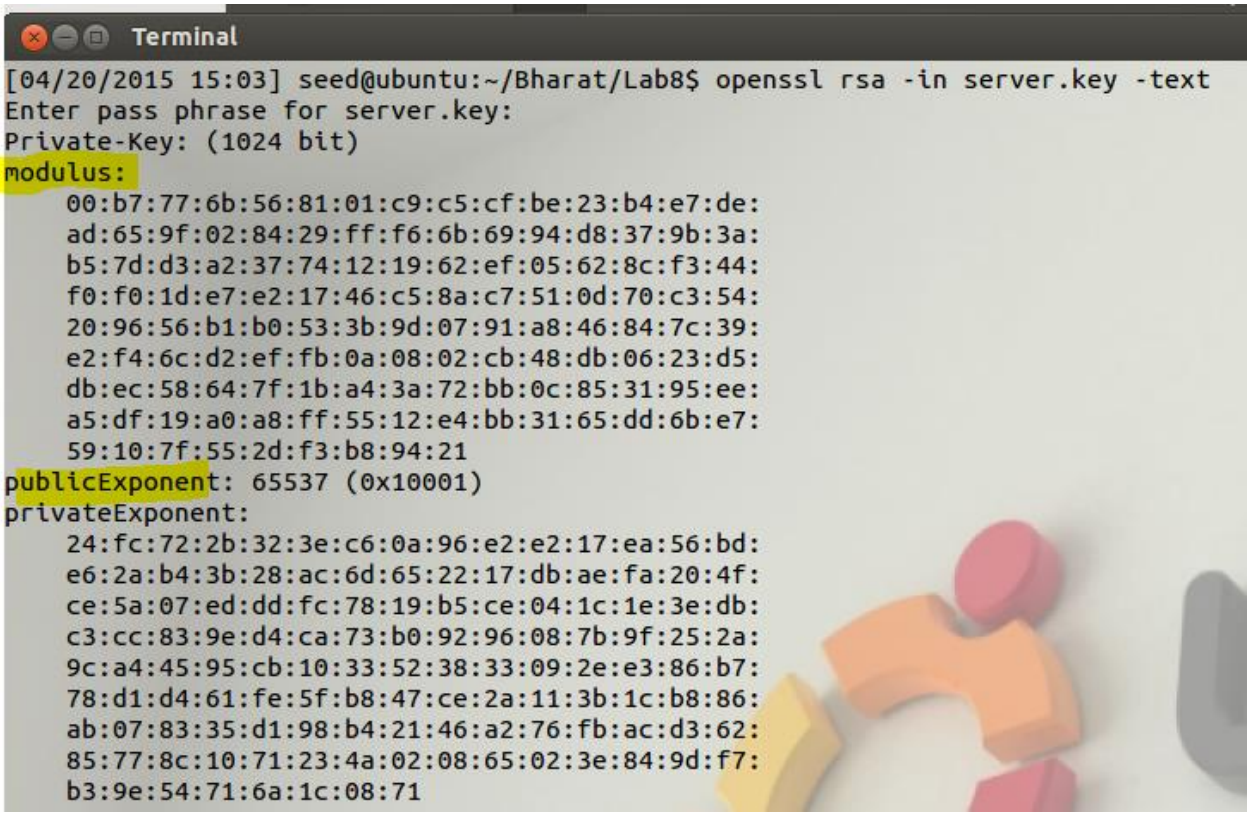
The key is written to a server.key file.



server.key file with the private key

As we can see, this file is encoded. We can use the following command to see the contents:
'*openssl rsa -in server.key -text*'.

A pass-phrase will be asked to open this file. This is the one which we used during generation of the key-pair.

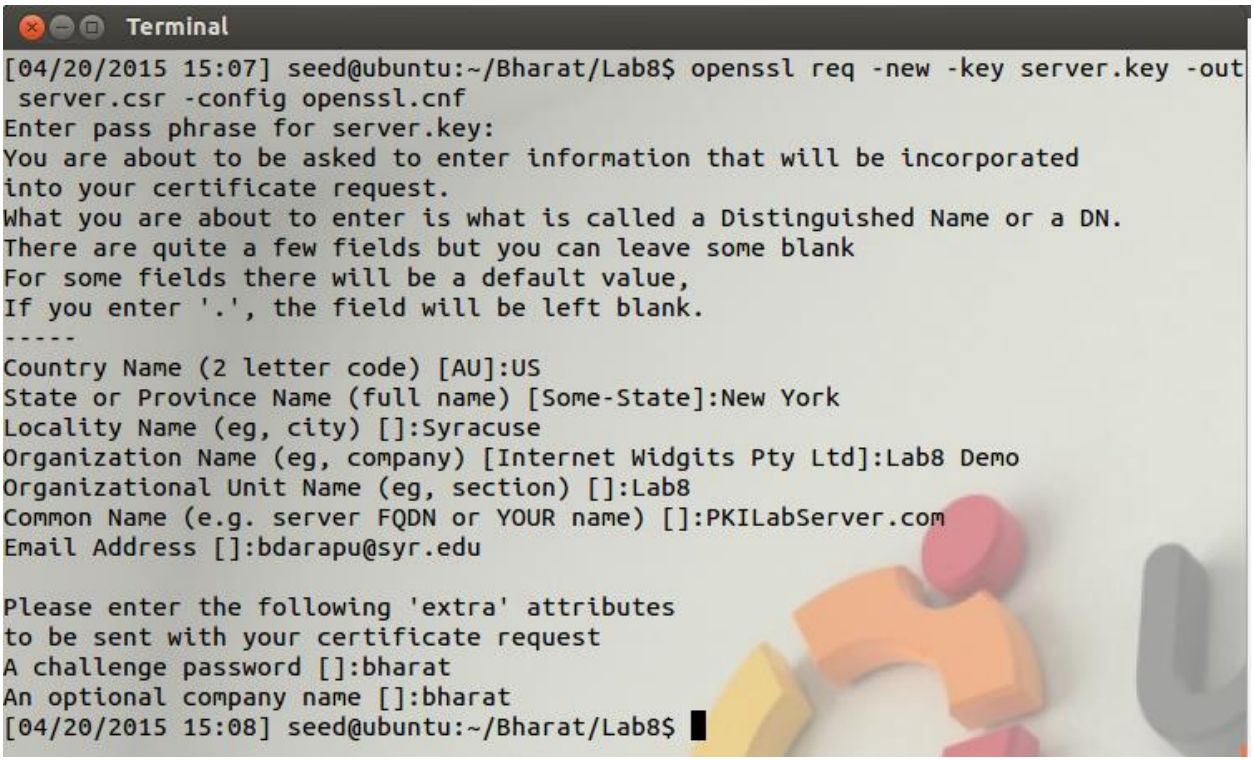


Server.key contents

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

Now that we have the key file. We can generate a Certificate Signing Request (CSR). And we will be using the following command:

'*openssl req -new -key server.key -out server.csr -config openssl.cnf*'

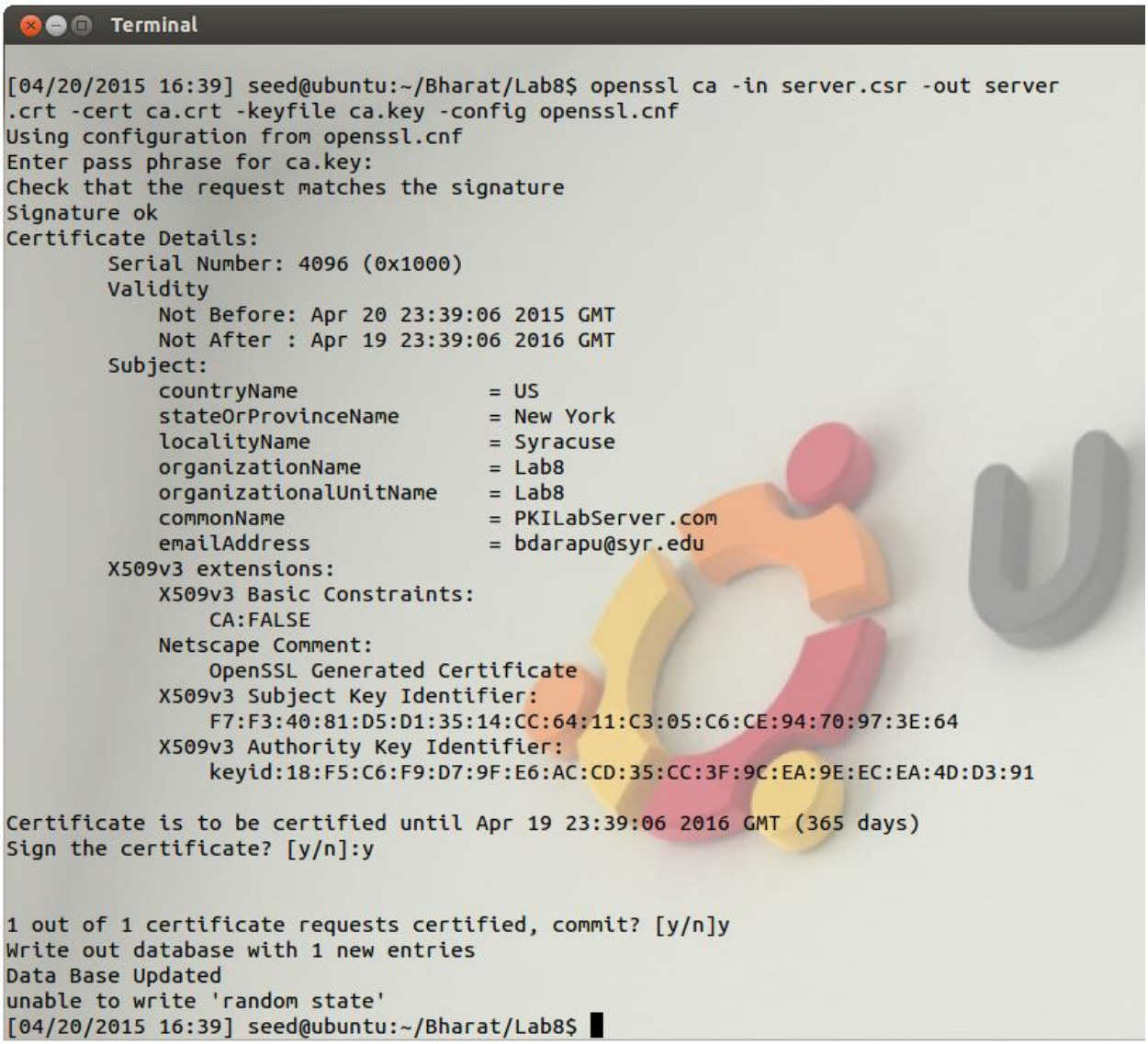


CSR generated

Step 3: Generating Certificates:

We will be using the following command to turn the certificate signing request (server.csr) into an x509 certificate (server.crt).:

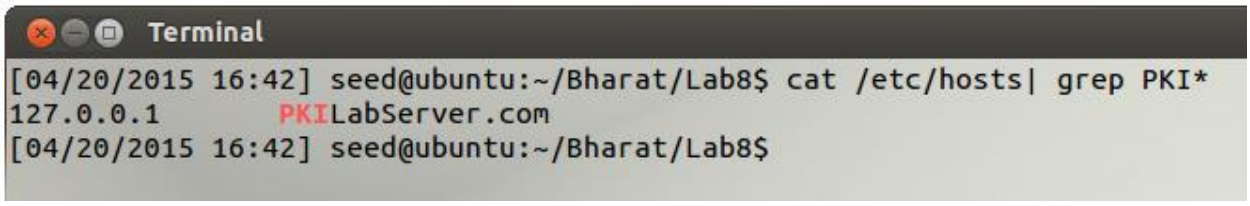
'*openssl ca -in server.csr -out server.crt -cert ca.crt -keyfile ca.key \ -config openssl.cnf***'**



Server.crt, certificate generated

3.3 Task 3: Use PKI for Web Sites

First we make PKILabServer.com as our domain name and map the domain name to local host by editing the /etc/hosts file.



host added in VM

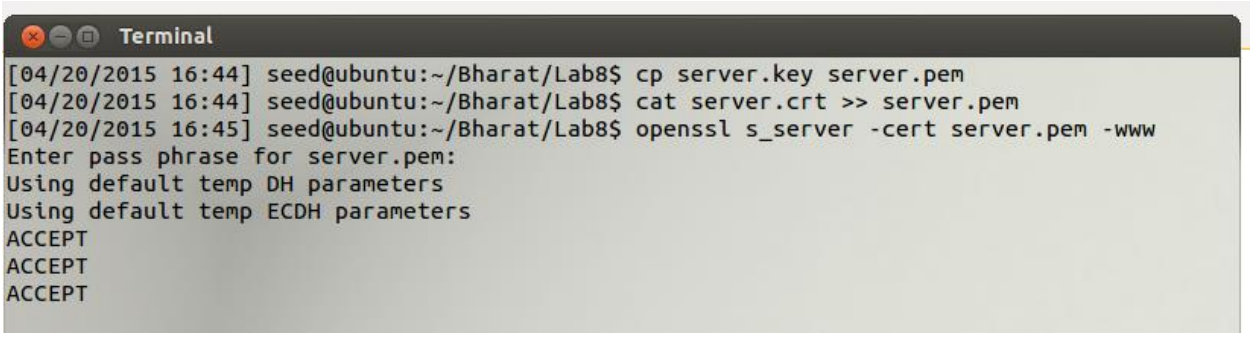
Next we combine both the certificate and the key into a single file (to load later) using the following commands:

'cp server.key server.pem'

'cat server.crt >> server.pem'

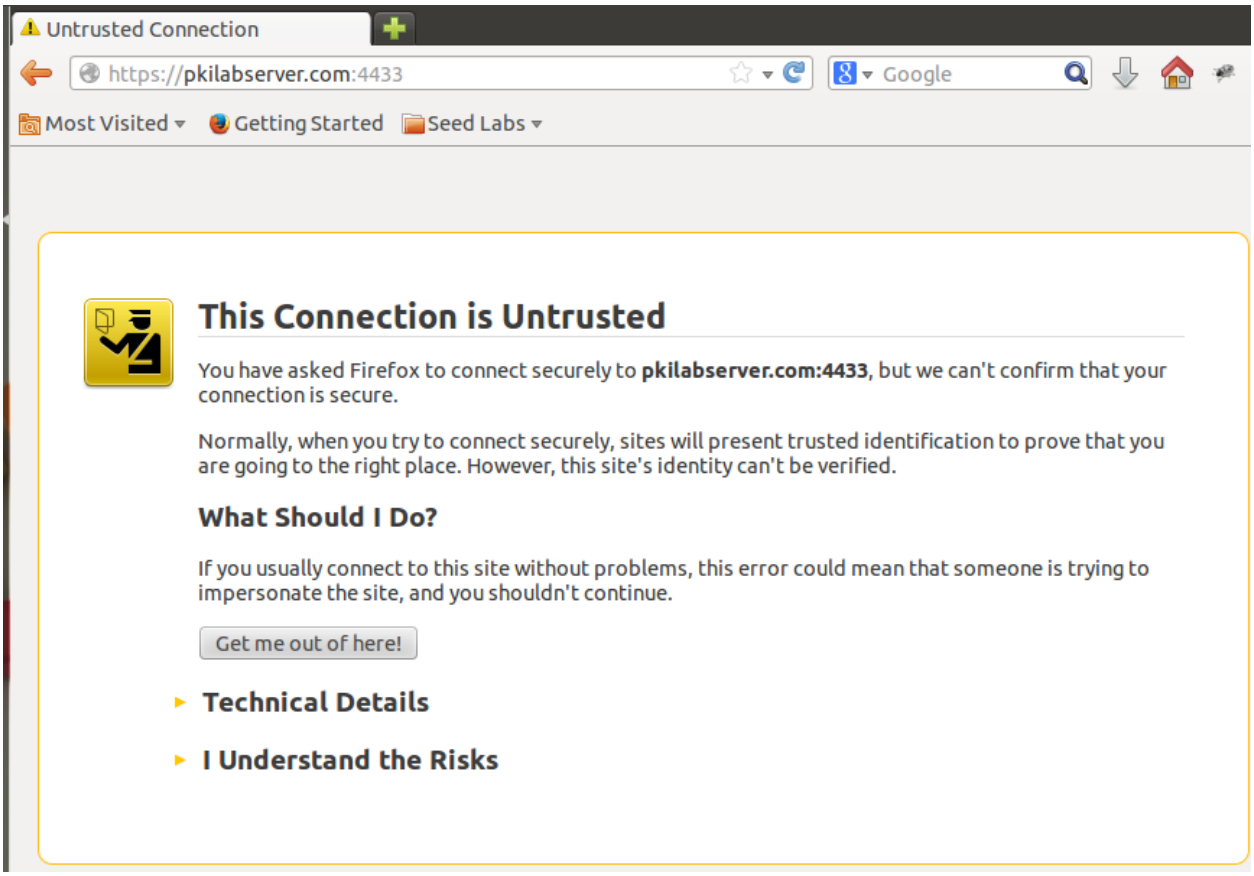
Further we launch a webserver with the certificate generated in the previous task using the following command: '*openssl s_server –cert server.pem -www*'

The result can be seen as follows:



PKILabServer.com webservice started

Now when we try to access 'https://PKILabServer.com:4433' from the browser we get the following message:

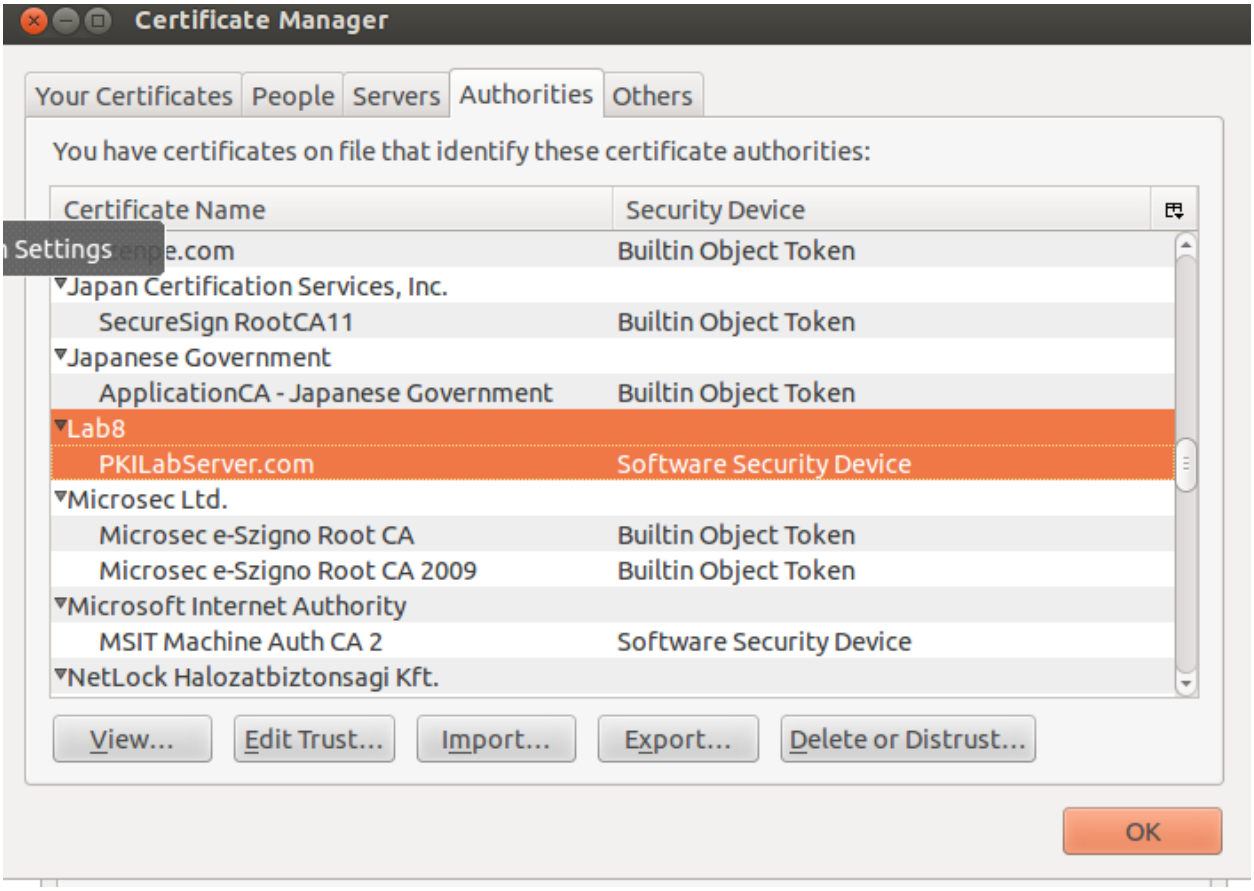


untrusted connection message in firefox.

This happens since there is no valid certificate issued previously for that server.

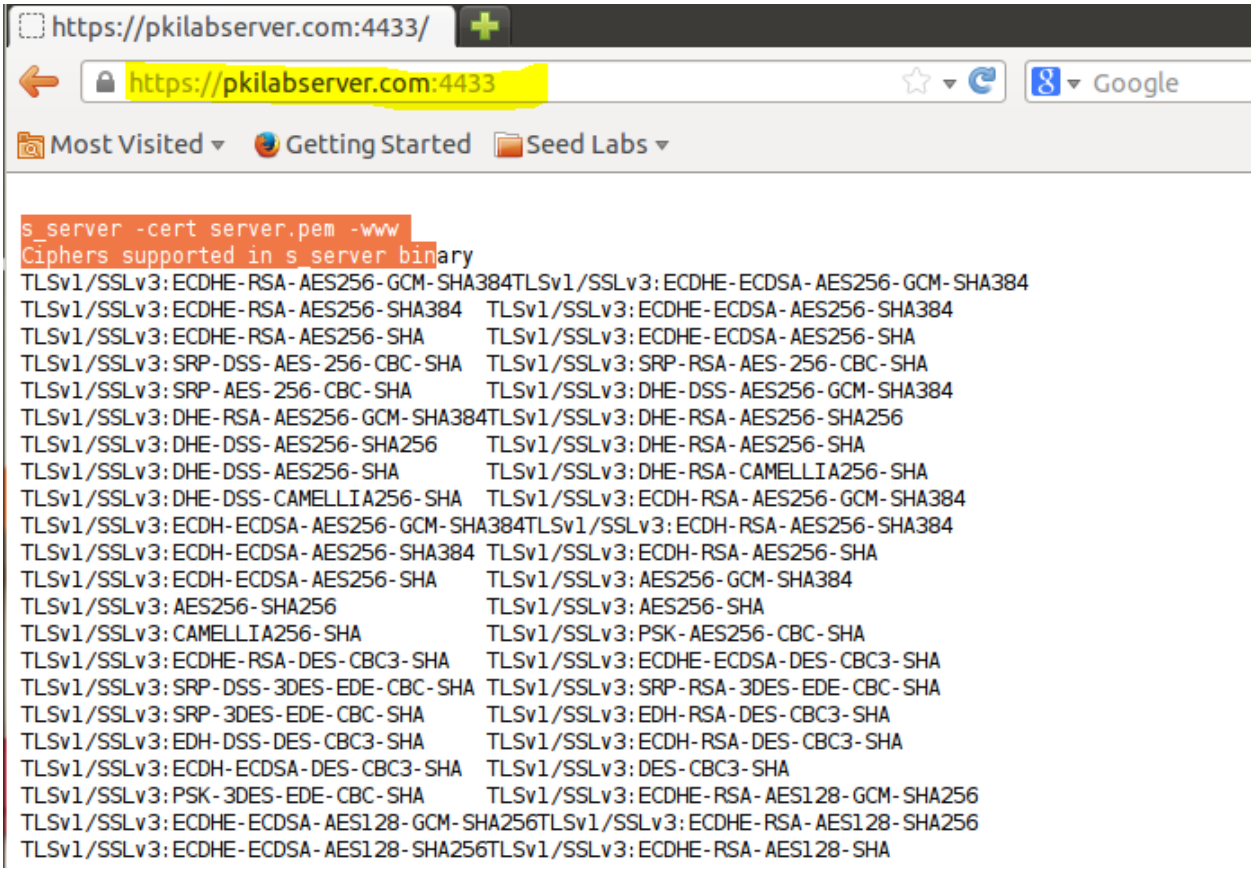
Now we have to annually add our CA's certificate to firefox browser by going to:
Edit -> Preference -> Advanced -> View Certificates

And upload the certificate which we created in the steps above.



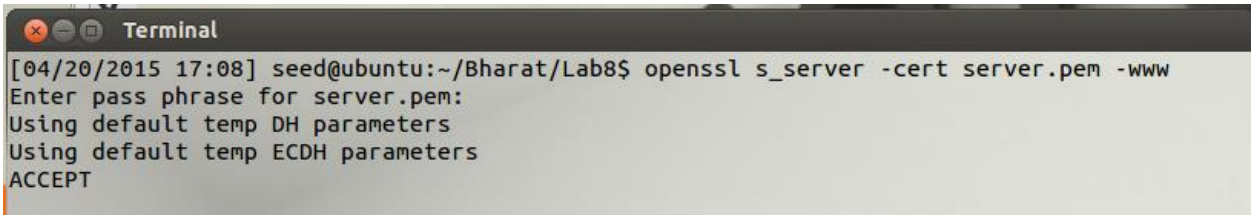
certificate uploaded and added

After loading the certificate the webpage can be seen as follows:



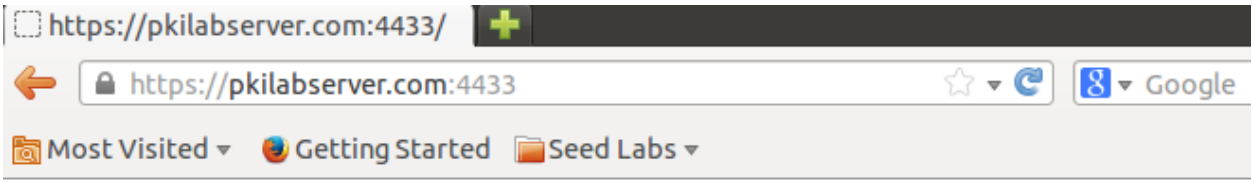
webpage after certificate load

Next we modify one byte of server.pem and try to restart the server



editing server.pem file and restarting the server

Next when we try to reload the url.

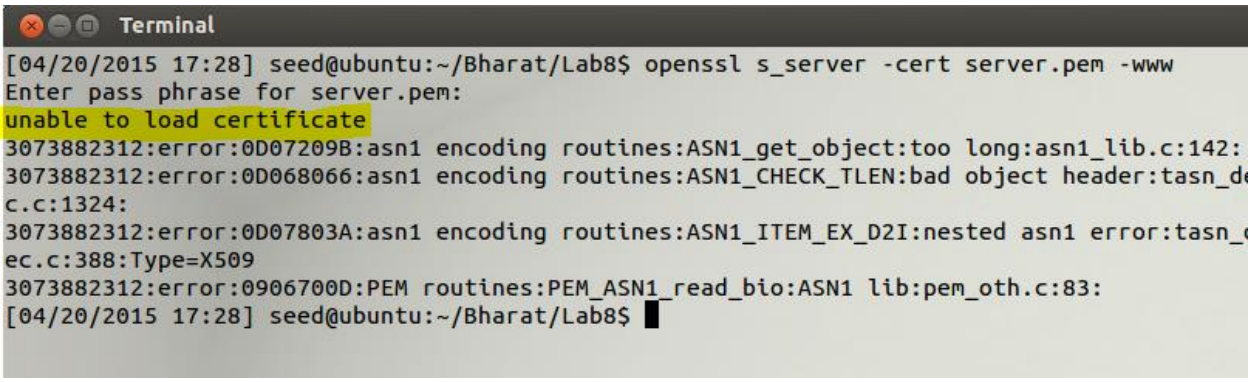



```
s_server -cert server.pem -www
Ciphers supported in s_server binary
TLSv1/SSLv3:ECDHE-RSA-AES256-GCM-SHA384TLSv1/SSLv3:ECDHE-ECDSA-AES256-GCM-SHA384
TLSv1/SSLv3:ECDHE-RSA-AES256-SHA384   TLSv1/SSLv3:ECDHE-ECDSA-AES256-SHA384
TLSv1/SSLv3:ECDHE-RSA-AES256-SHA      TLSv1/SSLv3:ECDHE-ECDSA-AES256-SHA
TLSv1/SSLv3:SRP-DSS-AES-256-CBC-SHA   TLSv1/SSLv3:SRP-RSA-AES-256-CBC-SHA
TLSv1/SSLv3:SRP-AES-256-CBC-SHA       TLSv1/SSLv3:DHE-DSS-AES256-GCM-SHA384
TLSv1/SSLv3:DHE-RSA-AES256-GCM-SHA384TLSv1/SSLv3:DHE-RSA-AES256-SHA256
TLSv1/SSLv3:DHE-DSS-AES256-SHA256     TLSv1/SSLv3:DHE-RSA-AES256-SHA
TLSv1/SSLv3:DHE-DSS-AES256-SHA        TLSv1/SSLv3:DHE-RSA-CAMELLIA256-SHA
TLSv1/SSLv3:DHE-DSS-CAMELLIA256-SHA   TLSv1/SSLv3:ECDH-RSA-AES256-GCM-SHA384
TLSv1/SSLv3:ECDH-ECDSA-AES256-GCM-SHA384TLSv1/SSLv3:ECDH-RSA-AES256-SHA384
TLSv1/SSLv3:ECDH-ECDSA-AES256-SHA384  TLSv1/SSLv3:ECDH-RSA-AES256-SHA
TLSv1/SSLv3:ECDH-ECDSA-AES256-SHA     TLSv1/SSLv3:AES256-GCM-SHA384
TLSv1/SSLv3:AES256-SHA256             TLSv1/SSLv3:AES256-SHA
TLSv1/SSLv3:CAMELLIA256-SHA           TLSv1/SSLv3:PSK-AES256-CBC-SHA
TLSv1/SSLv3:ECDHE-RSA-DES-CBC3-SHA    TLSv1/SSLv3:ECDHE-ECDSA-DES-CBC3-SHA
TLSv1/SSLv3:SRP-DSS-3DES-EDE-CBC-SHA TLSv1/SSLv3:SRP-RSA-3DES-EDE-CBC-SHA
TLSv1/SSLv3:SRP-3DES-EDE-CBC-SHA      TLSv1/SSLv3:EDH-RSA-DES-CBC3-SHA
TLSv1/SSLv3:EDH-DSS-DES-CBC3-SHA      TLSv1/SSLv3:ECDH-RSA-DES-CBC3-SHA
TLSv1/SSLv3:ECDH-ECDSA-DES-CBC3-SHA   TLSv1/SSLv3:DES-CBC3-SHA
TLSv1/SSLv3:PSK-3DES-EDE-CBC-SHA      TLSv1/SSLv3:ECDHE-RSA-AES128-GCM-SHA256
TLSv1/SSLv3:ECDHE-ECDSA-AES128-GCM-SHA256TLSv1/SSLv3:ECDHE-RSA-AES128-SHA256
TLSv1/SSLv3:ECDHE-ECDSA-AES128-SHA256TLSv1/SSLv3:ECDHE-RSA-AES128-SHA
TLSv1/SSLv3:ECDHE-ECDSA-AES128-SHA    TLSv1/SSLv3:SRP-DSS-AES-128-CBC-SHA
TLSv1/SSLv3:SRP-RSA-AES-128-CBC-SHA   TLSv1/SSLv3:SRP-AES-128-CBC-SHA
TLSv1/SSLv3:DHE-DSS-AES128-GCM-SHA256TLSv1/SSLv3:DHE-RSA-AES128-GCM-SHA256
TLSv1/SSLv3:DHE-RSA-AES128-SHA256     TLSv1/SSLv3:DHE-DSS-AES128-SHA256
TLSv1/SSLv3:DHE-RSA-AES128-SHA        TLSv1/SSLv3:DHE-DSS-AES128-SHA
TLSv1/SSLv3:DHE-RSA-SEED-SHA          TLSv1/SSLv3:DHE-DSS-SEED-SHA
TLSv1/SSLv3:DHE-RSA-CAMELLIA128-SHA   TLSv1/SSLv3:DHE-DSS-CAMELLIA128-SHA
TLSv1/SSLv3:ECDH-RSA-AES128-GCM-SHA256TLSv1/SSLv3:ECDH-ECDSA-AES128-GCM-SHA256
TLSv1/SSLv3:ECDH-RSA-AES128-SHA256    TLSv1/SSLv3:ECDH-ECDSA-AES128-SHA256
TLSv1/SSLv3:ECDH-RSA-AES128-SHA       TLSv1/SSLv3:ECDH-ECDSA-AES128-SHA
```

url loaded

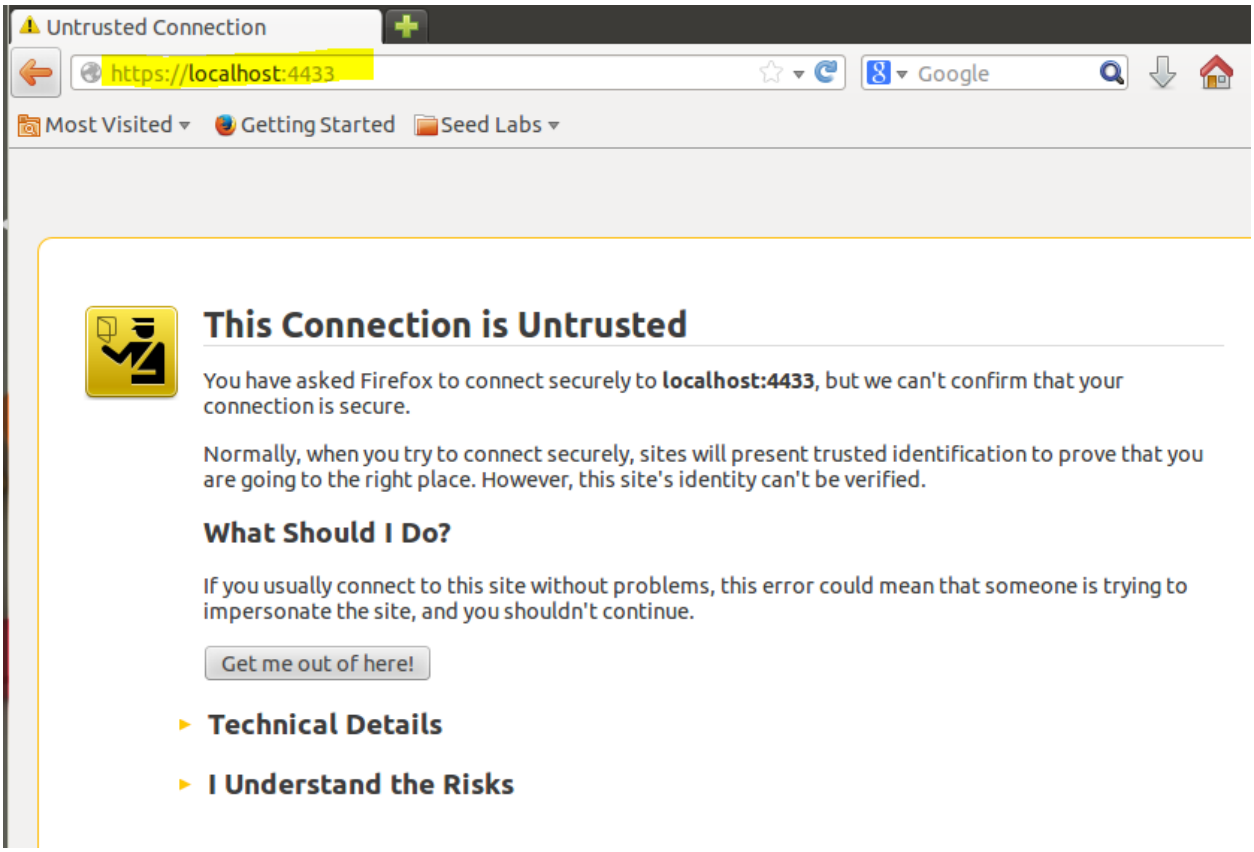
However when we change something in human readable format or the modulus or the key identifier nothing changes. The webpage loads as before.

When we change a byte in the last certificate section we get the following error:



certificate section modified

Next since PKILabServer.com is registered with 127.0.0.1 which is localhost, let see what happens if we hit https://localhost:4433.



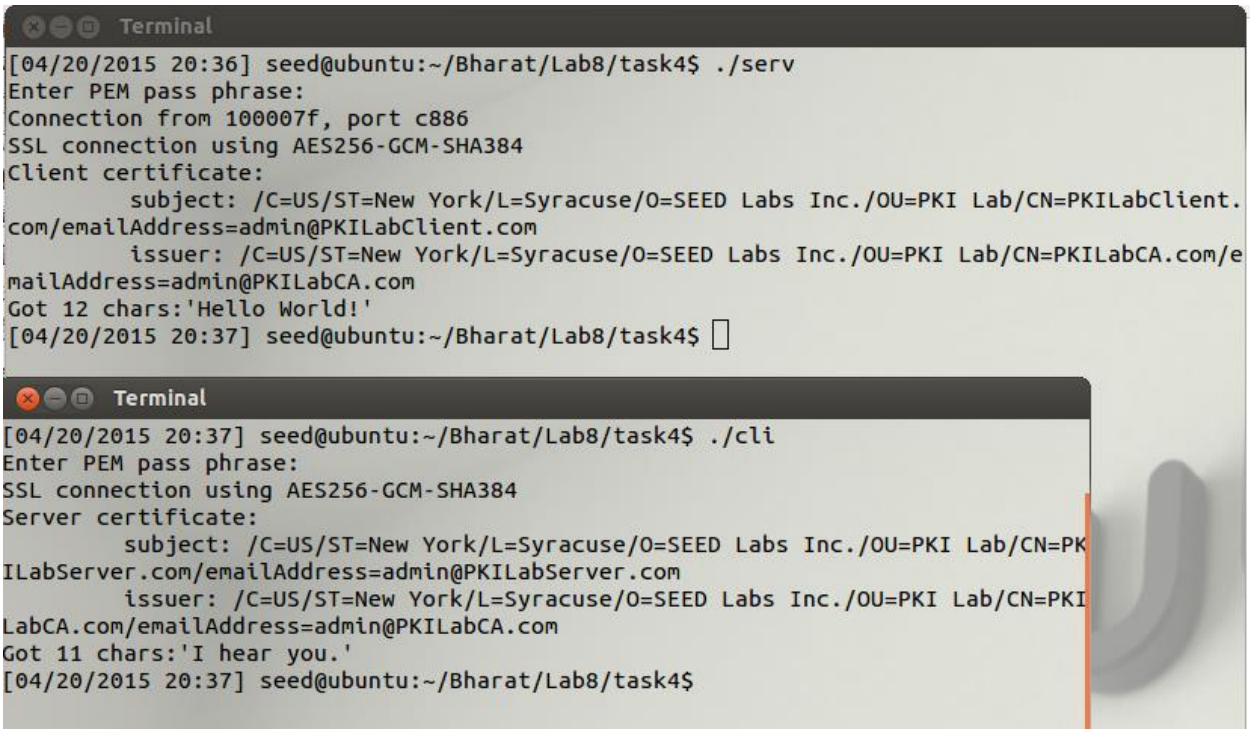
localhost hit

As we can see, it is not taking the previous certificate as the previous certificate is valid only for PKILabServer.com.

3.4 Task 4: Establishing a TLS/SSL connection with server

We will be using the example files given in the lab description.

For this task we will be using the certificates which we generated in the previous section. With valid certificates and no code changes we can see that the communication between the server and the client is successful.



transfer successful

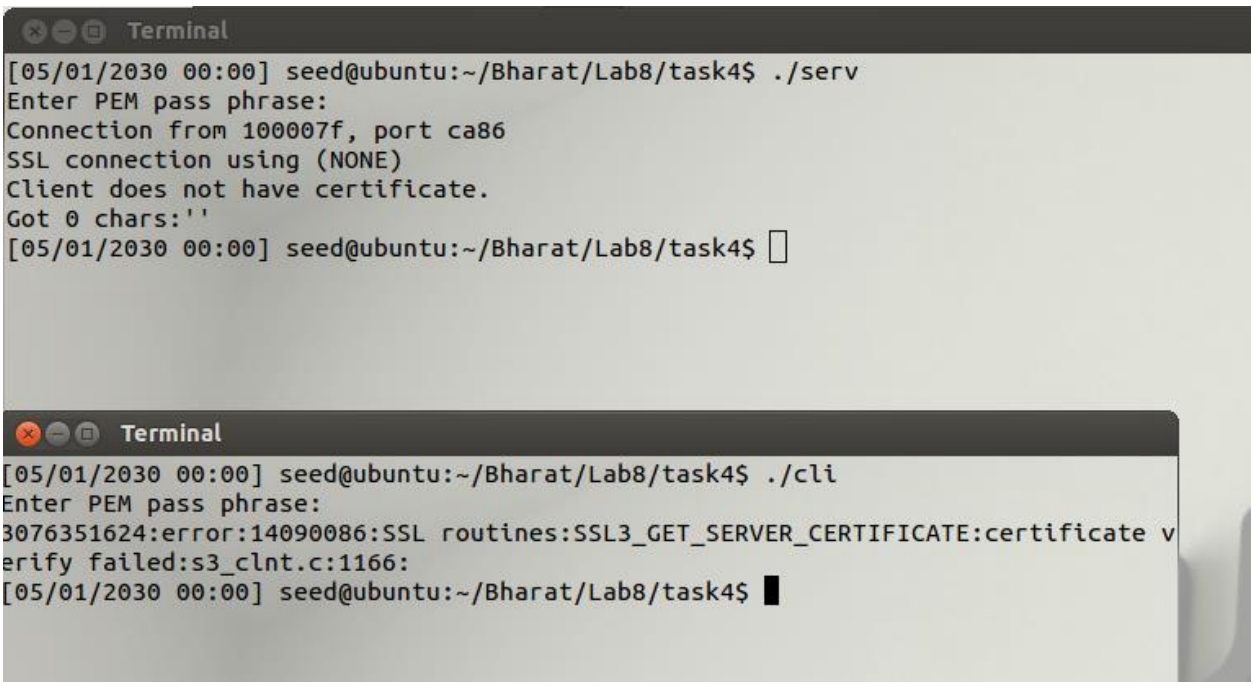
Date Validation:

Now let us change the date and see where the error comes:

Using the following command we stopeed the time sync feature:
'sudo service vboxadd-service stop'.

Using the following command we changed the system date:
'sudo date --set="1 May 2030"'

We can notice the date change in the terminal before the $ prompt. When we try to load the server we get the following error:

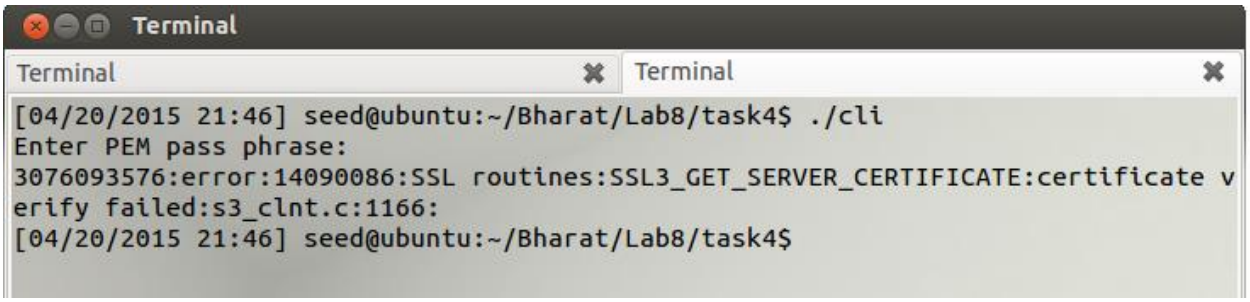


certificate verify failed due to invalid certificate.

Alternatively we can use the following lines of code to get and check the duration of the certificate:

*ASN1_TIME *not_before = X509_get_notBefore(server_cert);*
*ASN1_TIME *not_after = X509_get_notAfter(server_cert);*

Whether the server certificate is signed by an authorized CA
we can do this by loading a certificate that was not signed by us in the previous task and check. This way we are not holding the private key for the certificate.



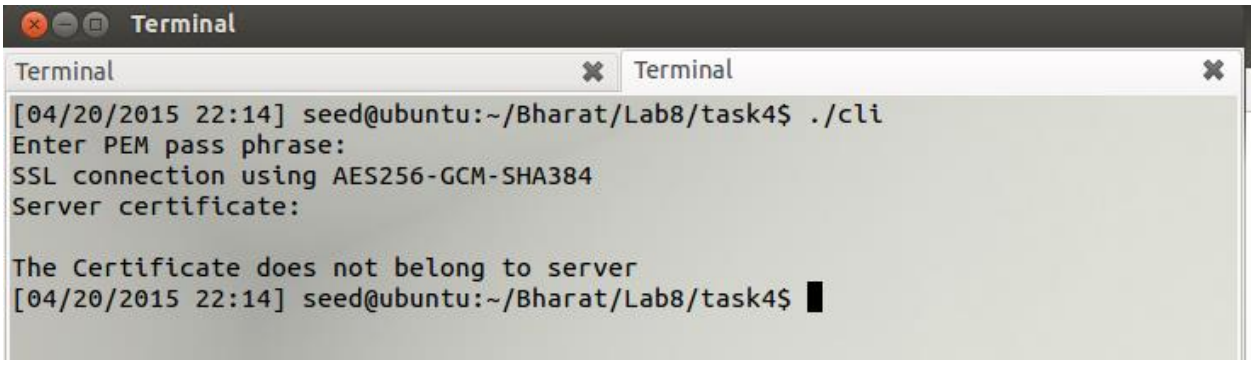
Invalid ca.cert loaded

Whether the certificate belongs to the server

The following code can be used to check if the certificate belongs to the server or not. If not then the program will exit.

```
char serverName [100];
strcpy(serverName, "PKILabServer.com");
char nameFromCertificate [100];
X509_NAME_get_text_by_NID(X509_get_subject_name(server_cert), NID_commonName, nameFromCertificate, 100);
if(strcmp(serverName, nameFromCertificate) != 0) {
        printf("The Certificate does not belong to server");
        X509_free (server_cert);
        exit(0);
 }
```

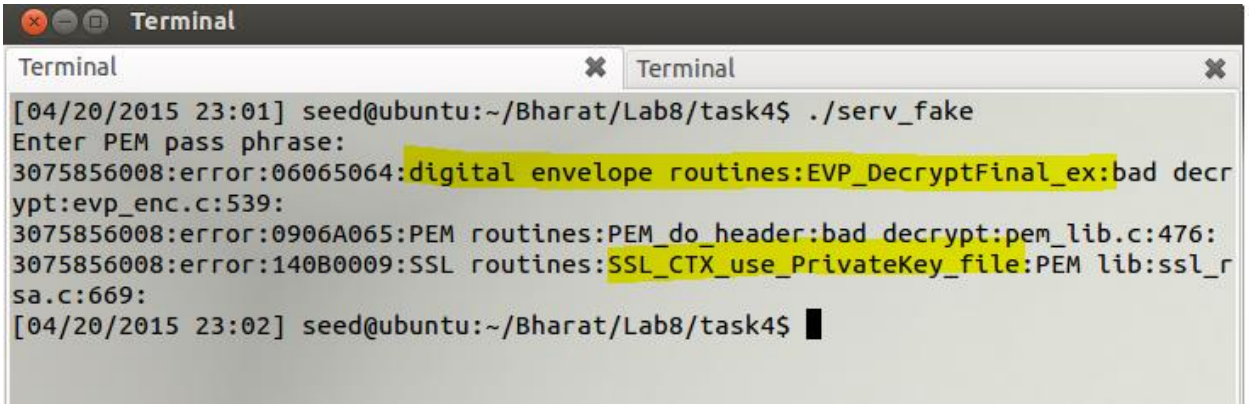
If we use a different server certificate than 'PKILabServer.com' then we get the following:



Certificate not from the expected server.

Whether the server is indeed the machine that the client wants to talk to
Now for this task we will have two servers:
1. Valid server with original certificate + valid private key.
2. Fraudulent server with original certificate + invalid private key (client_key was used).

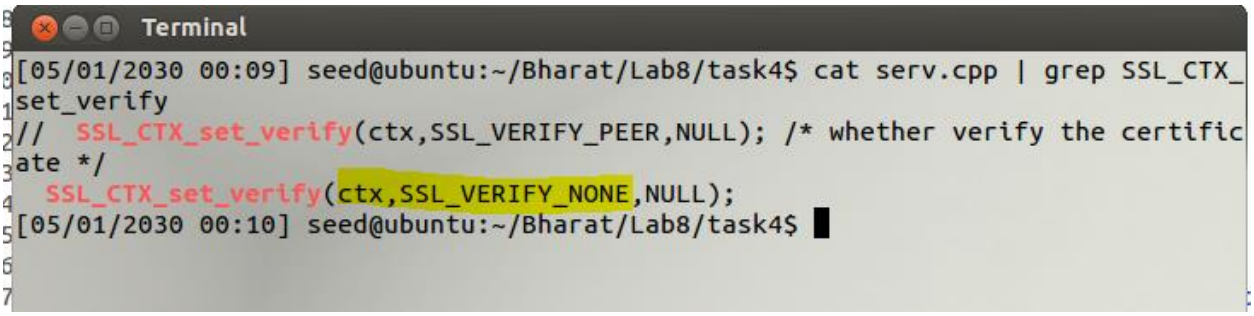
The valid server was able to connect without any trouble. However when we try to connect a fake server which has a valid certificate but not a valid private key we get the following error:



fake_server used to connect

As we can see the server went for a decryption with the private key it had but failed. So we were unable to login.

If we do not need to verify the client certificate, this can be done by modifying the following line on serv.cpp:



disable client vertificate verification

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

After both the server and client have validated their respective certificates, a secure TCP tunnel is formed. And the data is transferred via that. The data transfer code can be seen as follows:

This happens after a valid TCP connection is established and SSL negotioation starts:

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

Once the certificates are verified the data transfer begins using the following code:

Client side:

err = SSL_write (ssl, "Hello World!", strlen("Hello World!")); CHK_SSL(err);
err = SSL_read (ssl, buf, sizeof(buf) - 1); CHK_SSL(err);

The SSL_write method is used to write the key to a buffer on the channel and the SSL_read method is used to read the data from the channel.

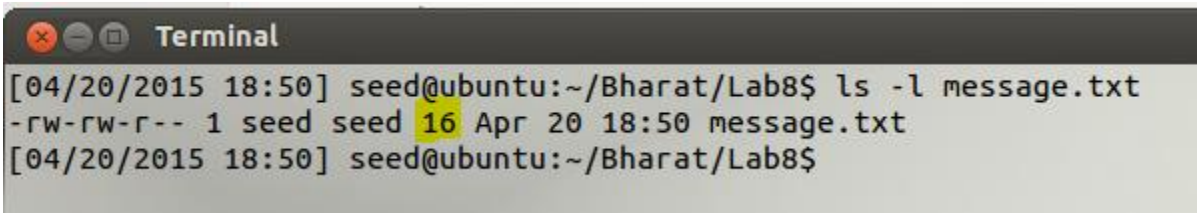
Server side:

err = SSL_read (ssl, buf, sizeof(buf) - 1); CHK_SSL(err);
err = SSL_write (ssl, "I hear you.", strlen("I hear you.")); CHK_SSL(err);

The same applies to the server side as well. The SSL_read method is used to read data from the channel and SSL_write method is used to write data to it.

3.5 Task 5: Performance Comparison: RSA versus AES

For this task we will use a 16-byte file message.txt as follows:

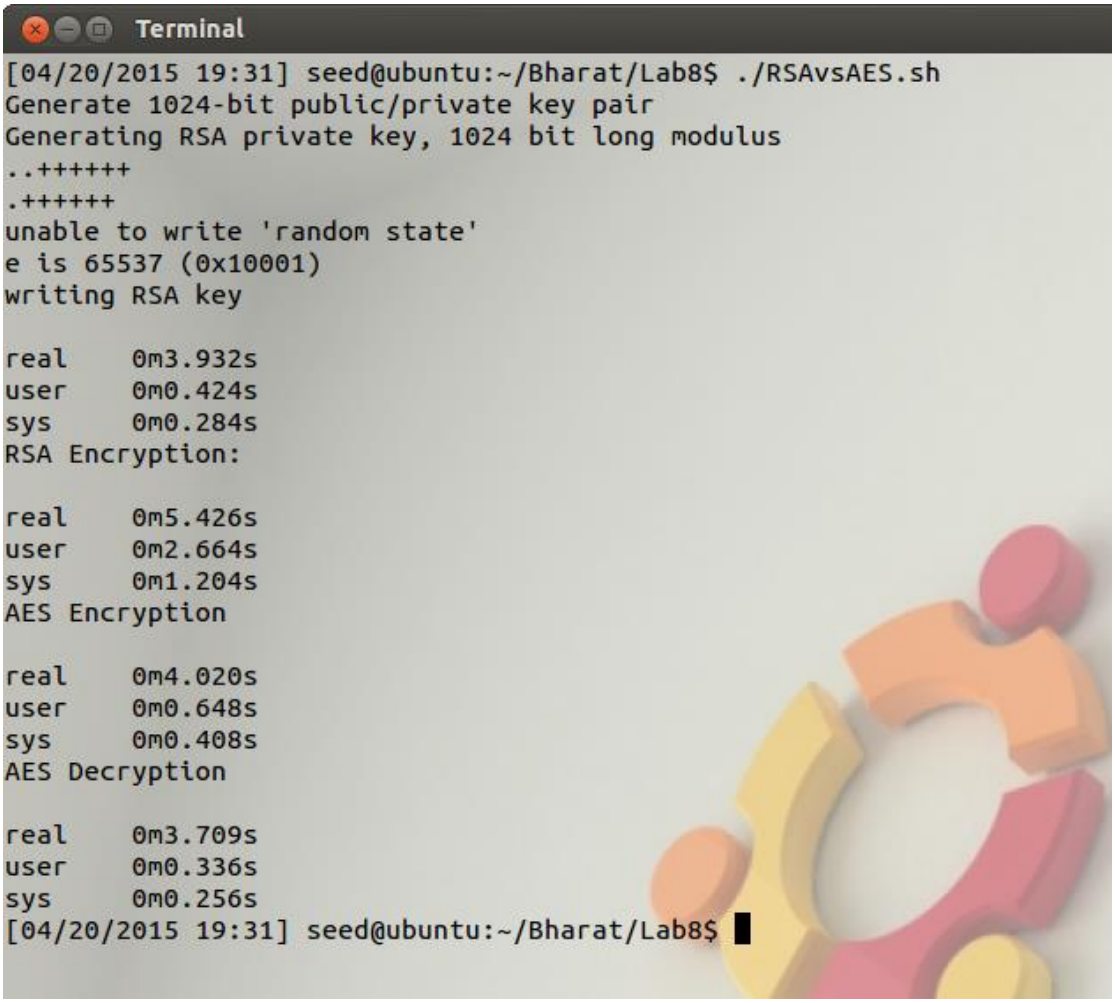


16 byte message.txt file

We will be using the following code (shell script) to see how the execution happens:

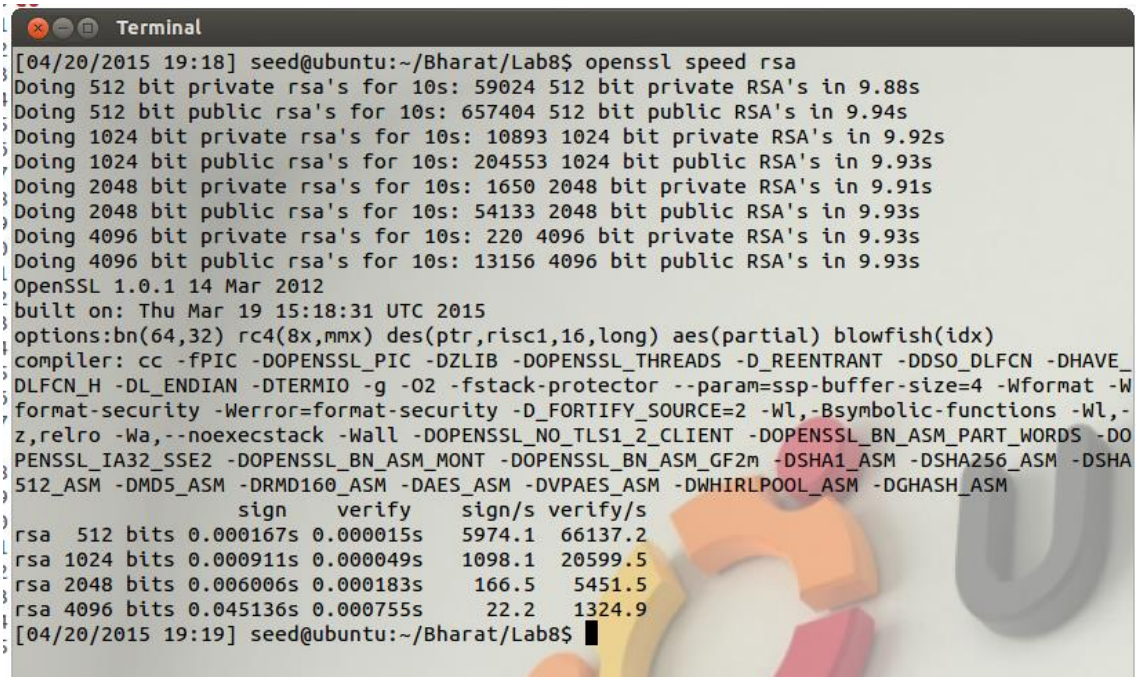
```
#!/bin/bash
echo "Generate 1024-bit public/private key pair"
openssl genrsa -out privateKey.pem 1024
openssl rsa -in privateKey.pem -out publicKey.pem -outform PEM -pubout

echo "RSA Encryption"
LOOP=1
time while (( $LOOP <= 500 ))
do
openssl rsautl -encrypt -inkey publicKey.pem -pubin -in message.txt -out message_enc.txt
LOOP=$(( LOOP+1 ))
done

echo "RSA Decryption"
LOOP=1
time while (( $LOOP <= 500 ))
do
openssl rsautl -decrypt -inkey privateKey.pem -in message_enc.txt -out message_decry.txt
LOOP=$(( LOOP+1 ))
done

echo "AES Encryption"
LOOP=1
time while (( $LOOP <= 500 ))
do
openssl enc -aes-128-cbc -e -in message.txt -out message_enc.txt -K 00112233445566778889aabbccddeeff -iv
102030405060708
LOOP=$(( LOOP+1 ))
done

echo "AES Decryption"
LOOP=1
time while (( $LOOP <= 500 ))
do
openssl enc -aes-128-cbc -d -in message_enc.txt -out message.txt -K 00112233445566778889aabbccddeeff -iv 102030405060708
LOOP=$(( LOOP+1 ))
Done
```

The output can be seen as follows:

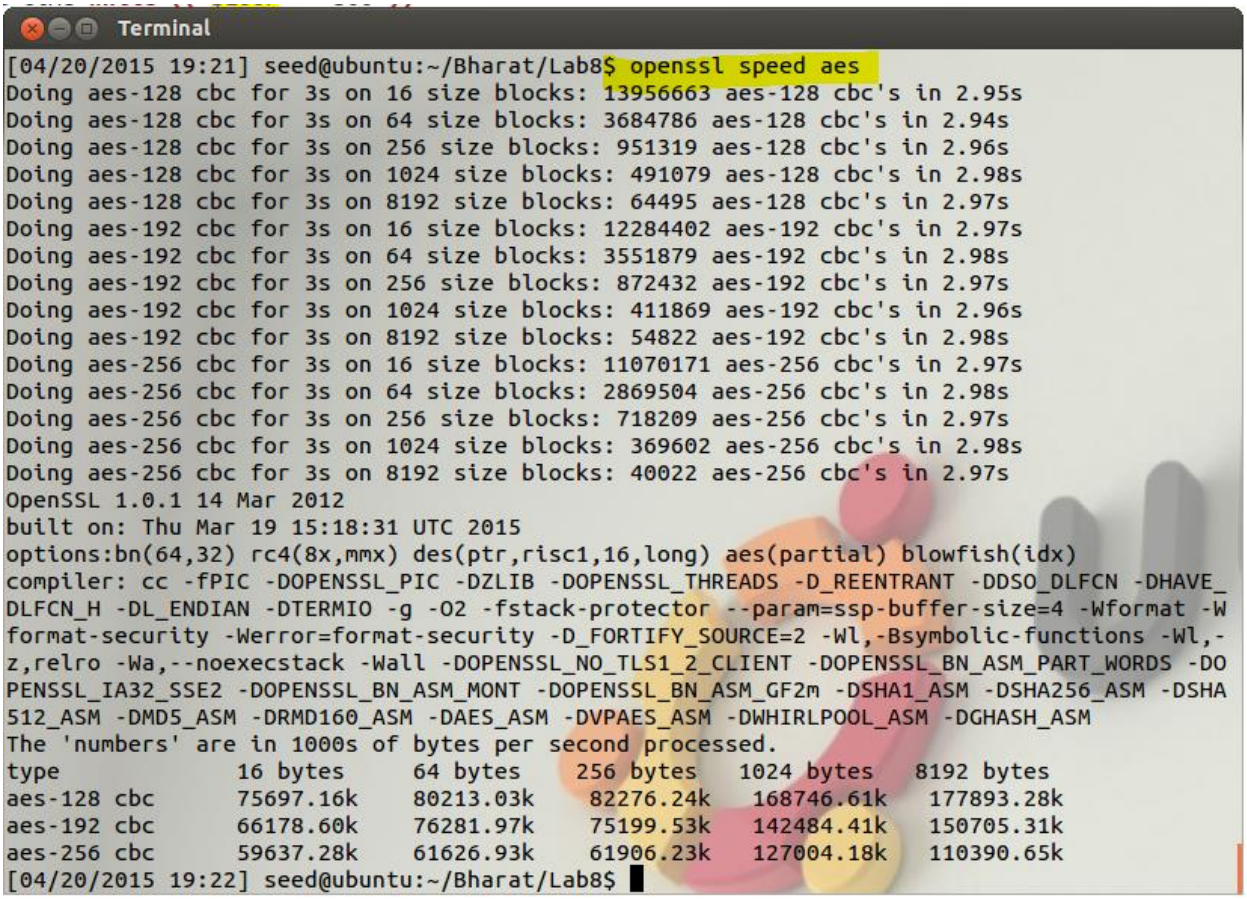


Performance comparison using custom script.

Now lets run speed benchmark provided by openssl and see the performance:



Rsa speed benchmark.



AES speed benchmark.

Custom shell script:

RSA Encryption:	3.93s
RSA Decryption:	5.42s
AES Encryption	4.02s
AES Decryption:	3.70s

Speed Benchmark:

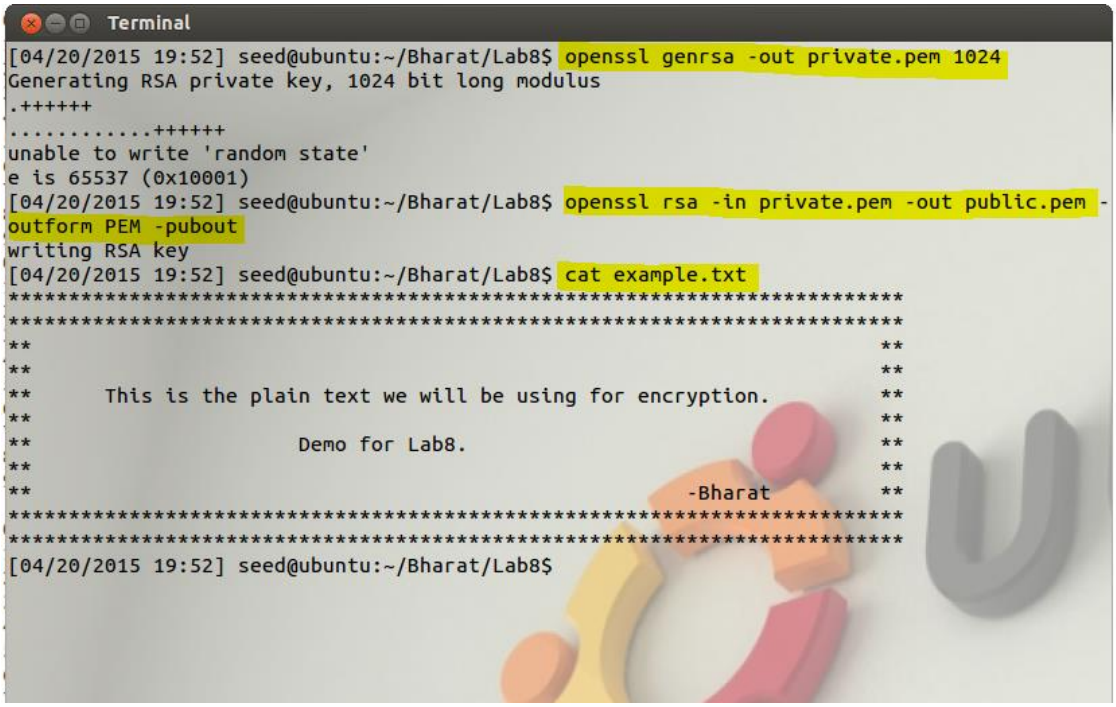
1024 bit private rsa's: 10893, 1024 bit private RSA's in 9.92s
1024 bit public rsa's: 204553, 1024 bit public RSA's in 9.93s
aes-128 cbc, 1024 size blocks: 491079 aes-128 cbc's in 2.98s

As per both the shell script and the speed benchmark **RSA is taking much greater time** to encrypt and decrypt than AES.

3.6 Task 6: Create Digital Signature

We will use the following example.txt file to generate the digest.

Next using the following command we will generate the public and private key pair:
'*openssl genrsa -out private.pem 1024*'
'*openssl rsa -in private.pem -out public.pem -outform PEM –pubout*'

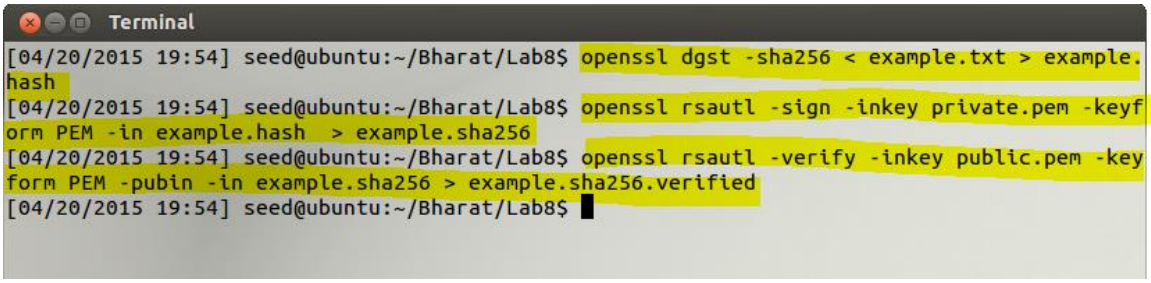


Generate the key pair and the contents of the example.txt file

Next we generate the hash using the following command:
openssl dgst -sha256 < example.txt > example.hash

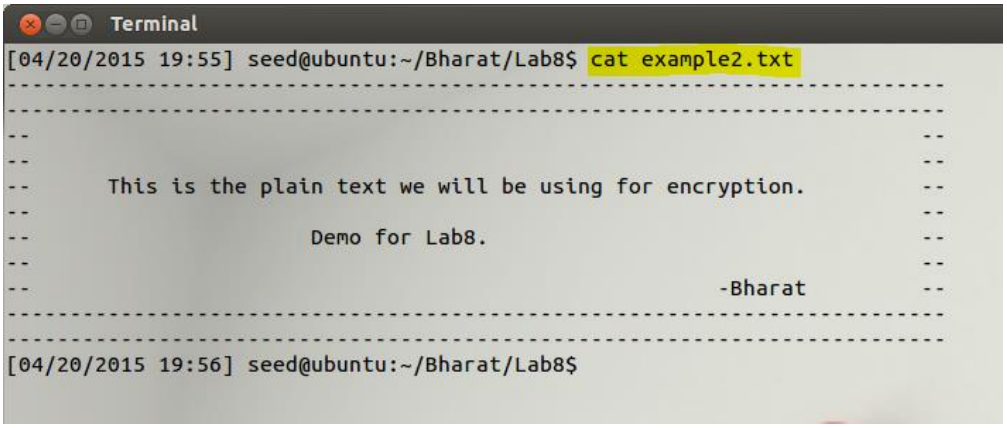
Next we sign the SHA 256 hash of example.txt:
openssl rsautl -sign -inkey private.pem -keyform PEM -in example.hash > example.sha256

Next we verify the digital signature in example.sha256 :
openssl rsautl -verify -inkey public.pem -keyform PEM -pubin -in example.sha256 > example.sha256.verified



Generate the signature of the hash of example.txt

Next we slightly modify example.txt and save it as example2.txt

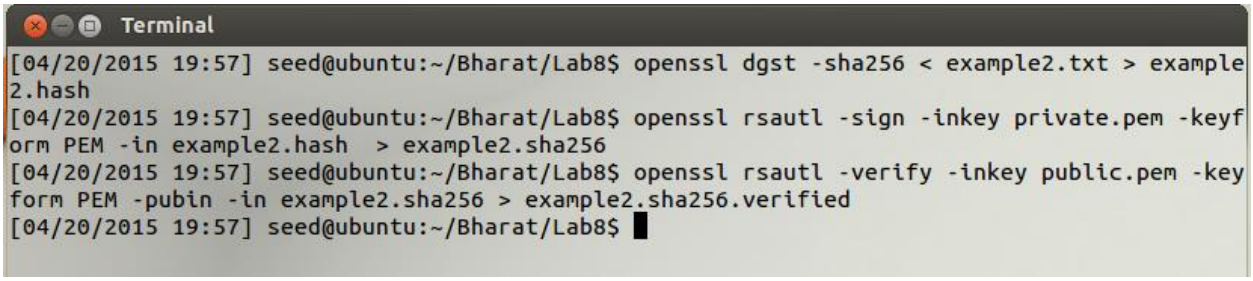


modified example.txt saved as example2.txt

Next we generate the hash using the following command:
openssl dgst -sha256 < example2.txt > example2.hash

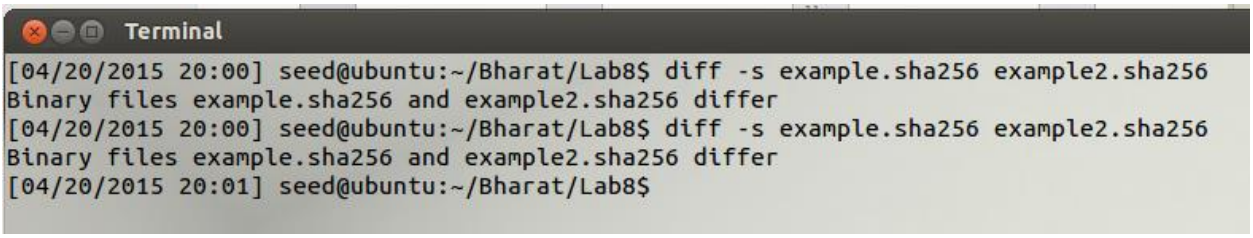
Next we sign the SHA 256 hash of example2.txt:
openssl rsautl -sign -inkey private.pem -keyform PEM -in example2.hash > example2.sha256

Next we verify the digital signature in example2.sha256 :
openssl rsautl -verify -inkey public.pem -keyform PEM -pubin -in example2.sha256 > example2.sha256.verified



Generate the signature of the hash of example2.txt

Now that we have the hashes of both example.txt and example2.txt if we comare the hashes:
'diff -s example.sha256 example2.sha256'



Comparison of hashes

Also, if we compare the signatures of the two hashes:
'diff -s example.sha256.verified example2.sha256.verified'

compariosion of signatures

A small change is file has completely changed the digital signature. Thus replicating the same signature is very difficult protecting our file making it very difficult to attack using standard dictionary attacks.