CS5460 – ASSIGNMENT 2

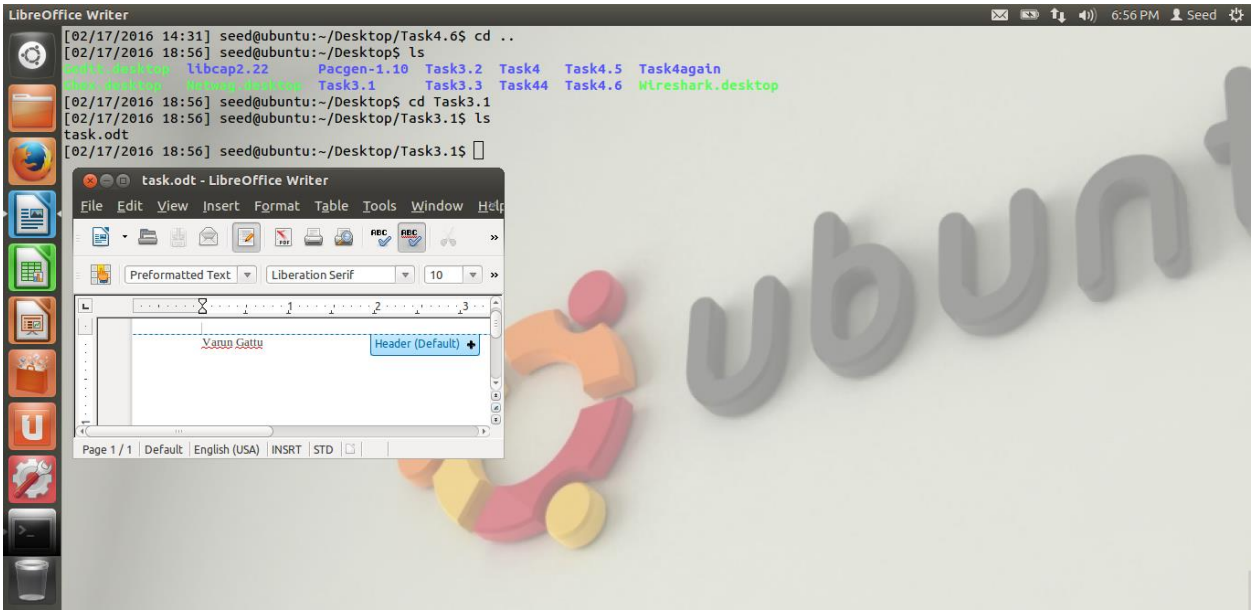
FEBRUARY 19, 2016

VARUN GATTU
A02092613

3. One-Way Hash Lab

3.1 Task 1: Creating Message Digest and MAC

To understand various one-way hash algorithms, I have created a file named task.odt with some data in it. The following screenshot shows it:



I have then used four different variants of the message digests. The digest types that I have used are:
MD5, SHA. SHA1 and SHA256 as shown in the following screenshot.

I can see that MD5 creates the smallest hash of all hash functions because it produces a 128 bit hash value. The SHA hash creates a hash with 160 bits similar to SHA1. The only difference between SHA and SHA1 is that SHA1 has one extra bitwise rotation in it. Whereas SHA256 produces the biggest hash value of 256bits.

3.2 Keyed Hash and HMAC

Here, I have generated keyed hash for the file named task.odt. I have used MD5, SHA1 and SHA256 hashing methods to create the keyed hash. I have used various lengths for the string to be included as shown below:

I could see that the hash value is not effected with the string length for any possible length. Similar to the Task3.1, MD5, SHA1 and SHA256 had 128bit, 160bit and 256bit keys respectively.

3.3 The randomness of One-way Hash

I have created a file named task.odt with some random data in it. I have used the file to create MD5 and SHA1 hash values on it. Then, I have changed a part of the file and again create similar hash on the new files. I could notice that the hash value completely changes in both the algorithms because they are based on various XOR operations. The hash value was same when I tried the hash on similar file for n number of times. But, once the file is tampered, hash value changes completely. The screenshot below shows my results:

Task 4: Public Key Cryptography Lab

4.1 Task1: Become a Certificate Authority (CA):

I have created a folder which has the files needed for this task. I have copied the openssl.cnf configuration file from /usr/lib/ssl/openssl.cnf to my local directory. The base folder had a subfolder called demoCA. demoCA folder had files called index.txt which is an empty database to store the certificate data. It also had a file called serial with some random number in it which will be the name of the certificate. It also had another folder named newcerts which holds the newly created certificates. The folder structure is as follows:

I have then created a self-signed certificate authority. The following screenshot shows how it is done:

4.2 Task 2: Create a Certificate for PKILabServer.com:

Step 1: Generate a public/private key pair:

I have created a public and private key pair with RSA algorithm. The following screenshots shows how this is achieved:

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

I have then created a Certificate signing request which will be sent to the Certificate Authority to generate a certificate for the public key.

Step 3: Generating Certificates:

I have now converted the CSR request into a certificate using our own certificate authority.

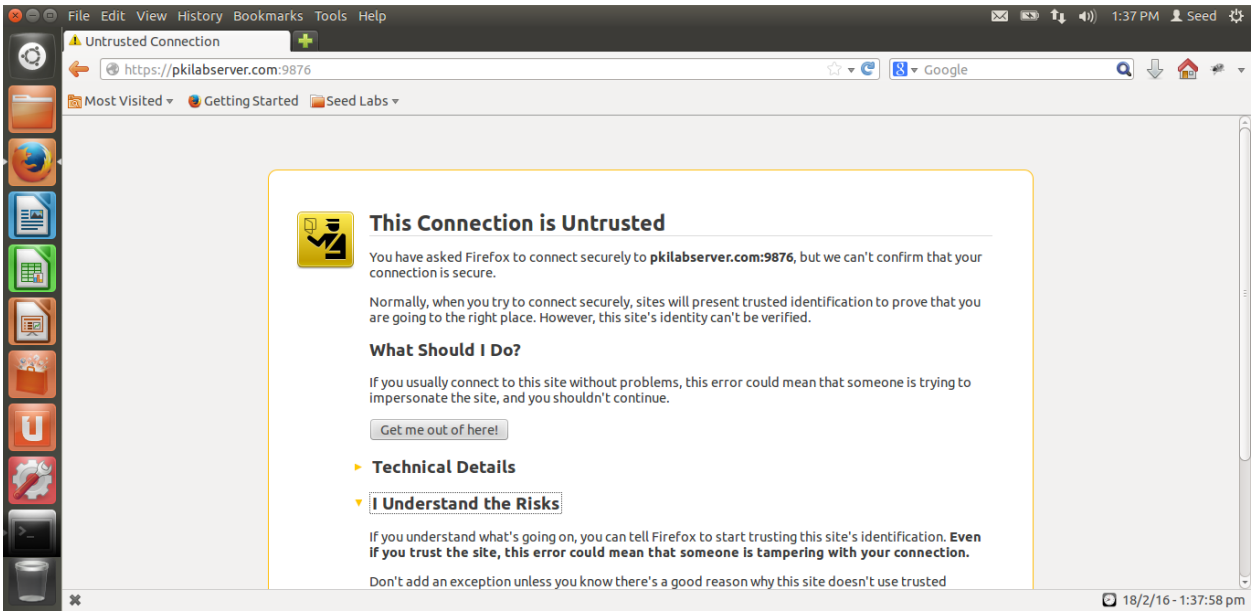
After successful creation of the certificate the folder structure looks like this:

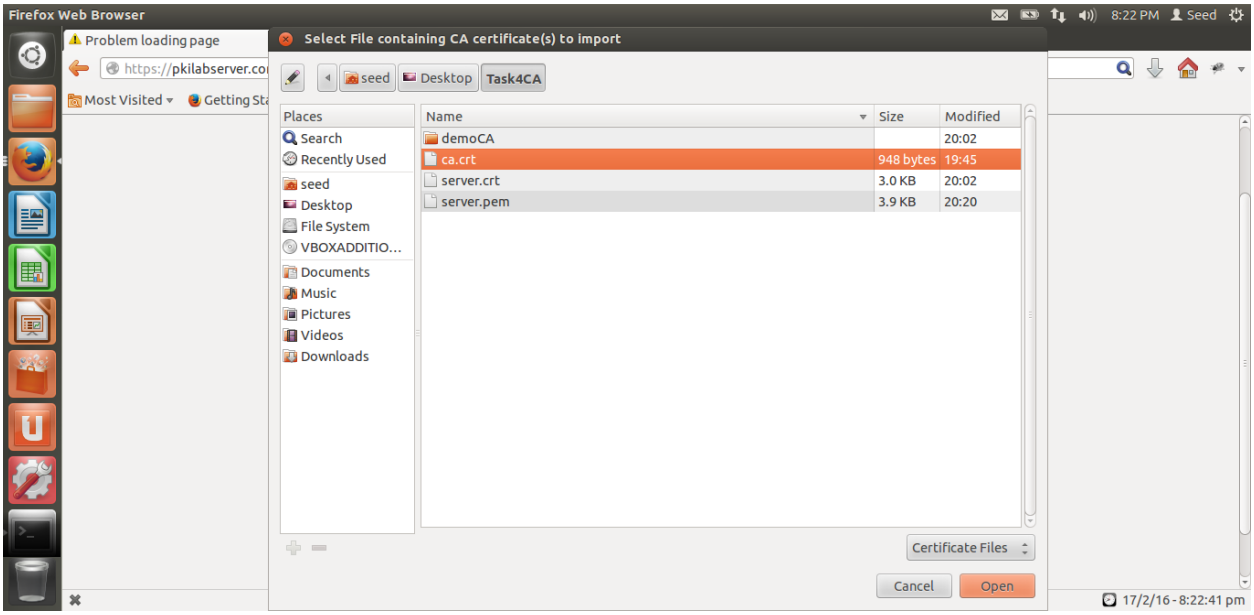
4.3 Task 3: Use PKI for Web Sites:

I have then added the PKILabServer.com address to the hosts file for it to be recognized by the system.

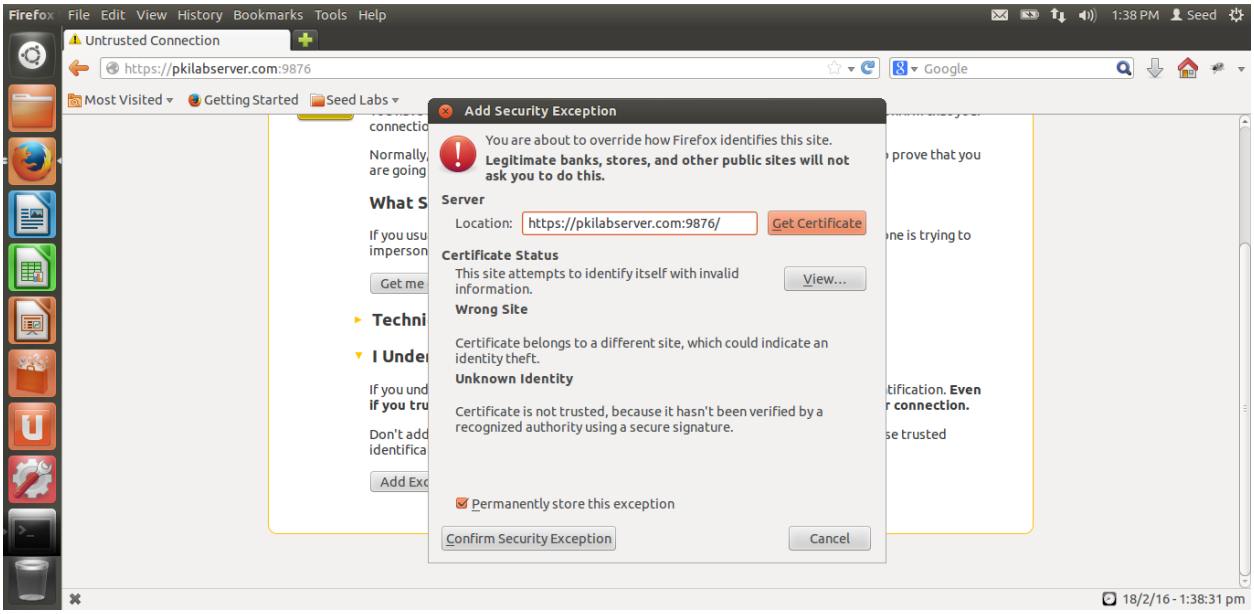
I have then combined the secret key and certificate into a single file and launched a server with the new certificate. Once the server starts running, we can access the website through a web browser.

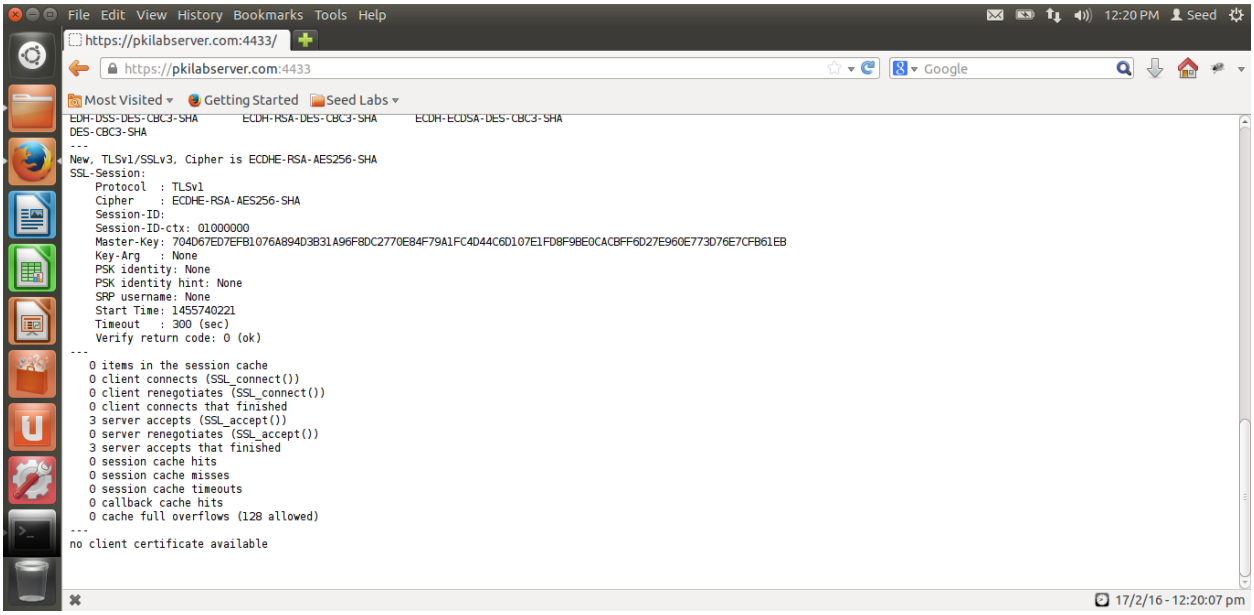
Mozilla firefox will not allow to access the page because the certificate authority is not global and is not imported to the firefox. We do it in the following steps:





After importing and refreshing the browser, we will now be able to access the site.





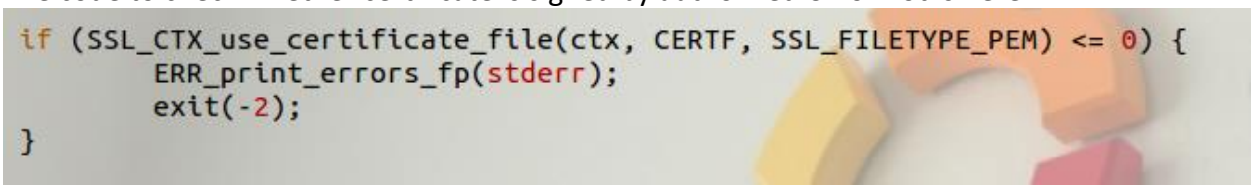
Now, if we modify the server.pem and start the server again to check the website, it will not be displayed. The browser throws an error message saying “secure connection failed” and doesn’t allow to access the website.

If we use localhost instead of pkilabserver, we still get a warning stating the connection is not trusted because of the CA and hosts file data.

4.4 Task 4: Establishing a TSL/SSL connection with server

I have downloaded the files into a folder and extracted them. I have then copied the server.key and server.crt from the previous folder to the current one.

The code to check whether certificate is signed by authorized CA or not is here:

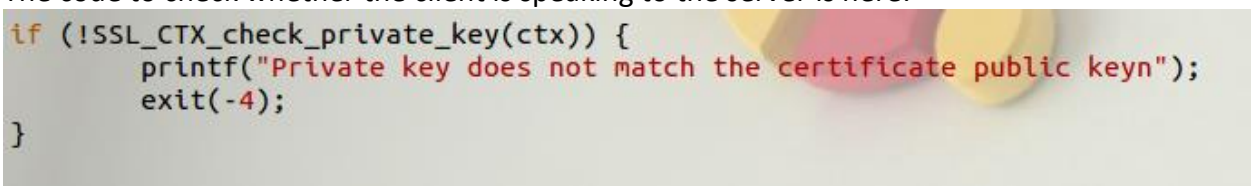


```
if (SSL_CTX_use_certificate_file(ctx, CERTF, SSL_FILETYPE_PEM) <= 0) {
        ERR_print_errors_fp(stderr);
        exit(-2);
}
```

The code to check whether the certificate belongs to the server or not is here:

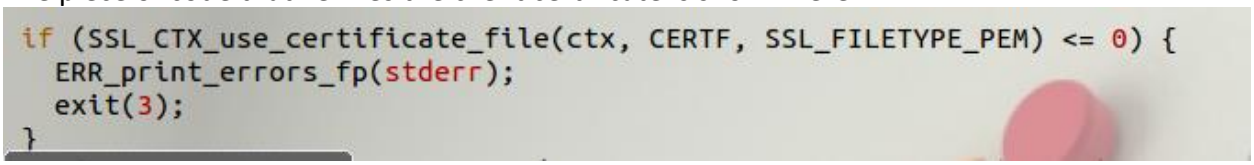
```
if (SSL_CTX_use_PrivateKey_file(ctx, KEYF, SSL_FILETYPE_PEM) <= 0) {
        ERR_print_errors_fp(stderr);
        exit(-3);
}
```

The code to check whether the client is speaking to the server is here:



```
if (!SSL_CTX_check_private_key(ctx)) {
        printf("Private key does not match the certificate public keyn");
        exit(-4);
}
```

The piece of code that verifies the client certificate is shown here:



```
if (SSL_CTX_use_certificate_file(ctx, CERTF, SSL_FILETYPE_PEM) <= 0) {
  ERR_print_errors_fp(stderr);
  exit(3);
}
```

This part of the code is removed because it is unwanted.

The next screenshot shows the client code responsible for key exchange with the server:

```
/* Get server's certificate (note: beware of dynamic allocation) - opt */

server_cert = SSL_get_peer_certificate (ssl);       CHK_NULL(server_cert);
printf ("Server certificate:\n");

str = X509_NAME_oneline (X509_get_subject_name (server_cert),0,0);
CHK_NULL(str);
printf ("\t subject: %s\n", str);
OPENSSL_free (str);

str = X509_NAME_oneline (X509_get_issuer_name  (server_cert),0,0);
CHK_NULL(str);
printf ("\t issuer: %s\n", str);
OPENSSL_free (str);
```

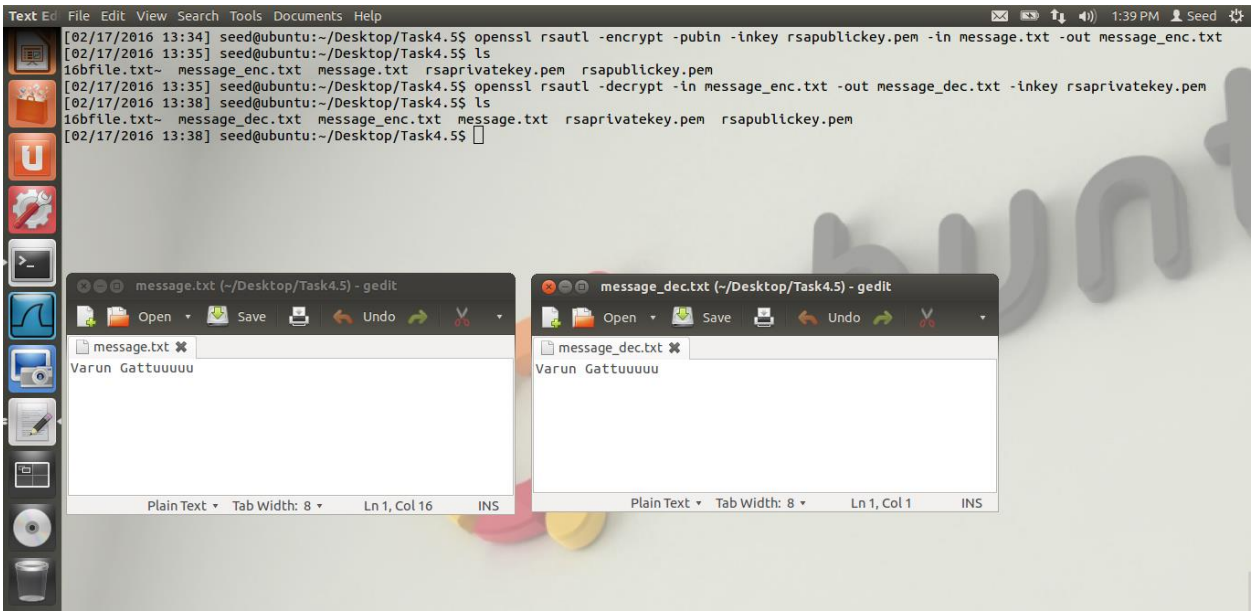
Similarly, the below screenshot shows the server side code that is responsible for key exchange:

```
client_cert = SSL_get_peer_certificate (ssl);
if (client_cert != NULL) {
  printf ("Client certificate:\n");

  str = X509_NAME_oneline (X509_get_subject_name (client_cert), 0, 0);
  CHK_NULL(str);
  printf ("\t subject: %s\n", str);
  OPENSSL_free (str);

  str = X509_NAME_oneline (X509_get_issuer_name  (client_cert), 0, 0);
  CHK_NULL(str);
  printf ("\t issuer: %s\n", str);
  OPENSSL_free (str);
```

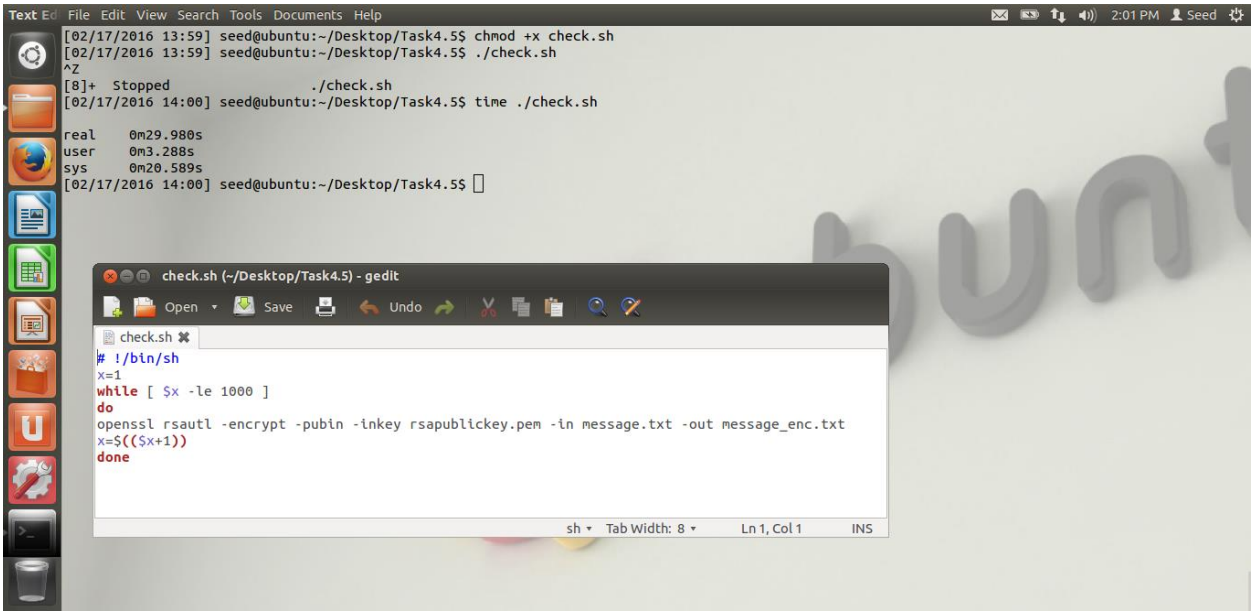
4.5 Task 5: Performance Comparison: RSA vs AES

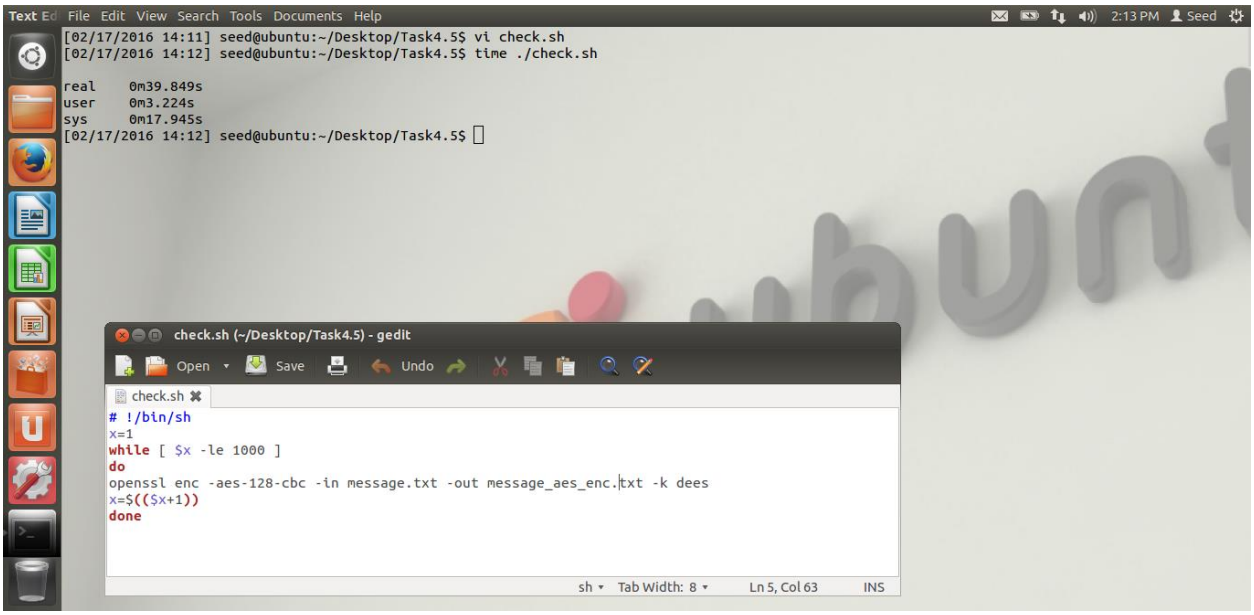
For this task, I have created a file named message.txt which is of 16bytes. I have then generated a public and private key pair with RSA like the following:

Now, I have encrypted the message.txt with the private key to create message_enc.txt. This is now decrypted using the public key. Now, the file is encrypted with 128 bit AES to create message_aes_enc.txt.



To compare the time taken for both these processes, I have written a script which executes the commands 1000 times to calculate the total time as follows:





This shows that AES encryption takes a little longer than RSA to be computed.
Now, I have tried to run the speed command for RSA1024 and AES to calculate their times to run the commands. The output is like the following:

This shows that RSA is slower then AES to execute the commands.

4.6 Task 6: Create Digital Signature:

I have created a file named example.txt with some random text in it. I have also created a public and private key pair with RSA for the same.

I then signed the example.txt with SHA256 hash to create example.sha256. I have verified the digital signature as below:

Now, I tried to modify the example.txt and then verify the signature. It doesn't verify the signature because of different hash values for both of them as below: