CNT 5410 Assignment 2

Harsh Shah

September 12, 2016

Instructor: Professor Patrick Traynor

1 Aim:

To gain a first-hand experience on Public-key encryption, Public-Key Infrastructure (PKI), digital signature, public-key certificate, certificate authority, authentication based on PKI.

2 Machine used:

System: VMware Virtual Platform ()
OS: Ubuntu 12.04
Processor: Intel(R) Core(TM) i5-6200U CPU @ 2.3GHz

3 Tasks:

3.1 Task 1: Becoming a Certificate Authority (CA)

In order to get a digital certificate issued for one's website, one needs to contact a CA such as VeriSign. Since we do not wish to pay for a digital certificate, we become a CA ourselves. We create files and directories required to store all our keys and certificates, as specified in the openssl configuration file (openssl.cnf). Using the command:
$ openssl req -new -x509 -keyout ca.key -out ca.crt -config openssl.cnf
we create a self signed public key certificate (ca.crt) and our private key (ca.key) as shown in Figure 1. Now, we are in a position to sign certificates for others.

3.2 Task 2: Create a Certificate for PKILabServer.com

As a customer seeking a digital certificate from a CA, we need to perform 3 steps:
1. Generate public/private key pair:
The following command uses RSA algorithm to generate public/private key pair

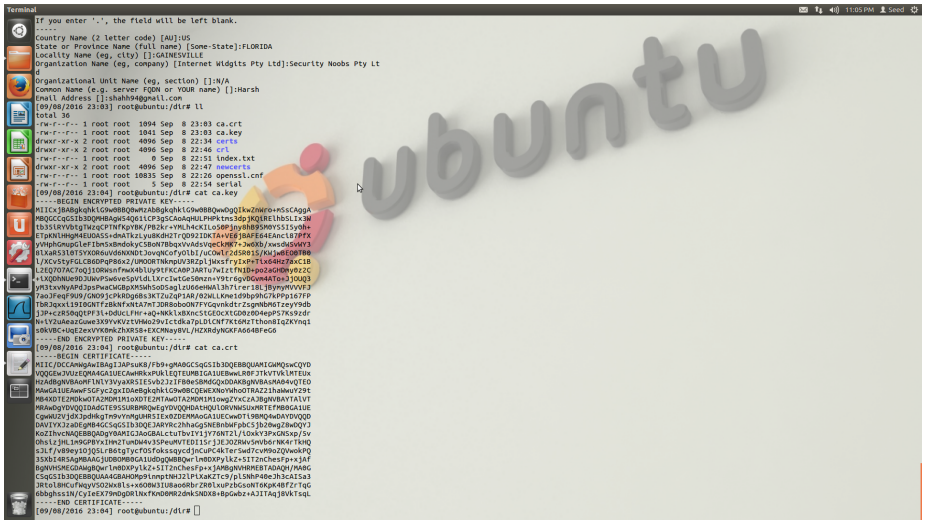

Figure 1: CA

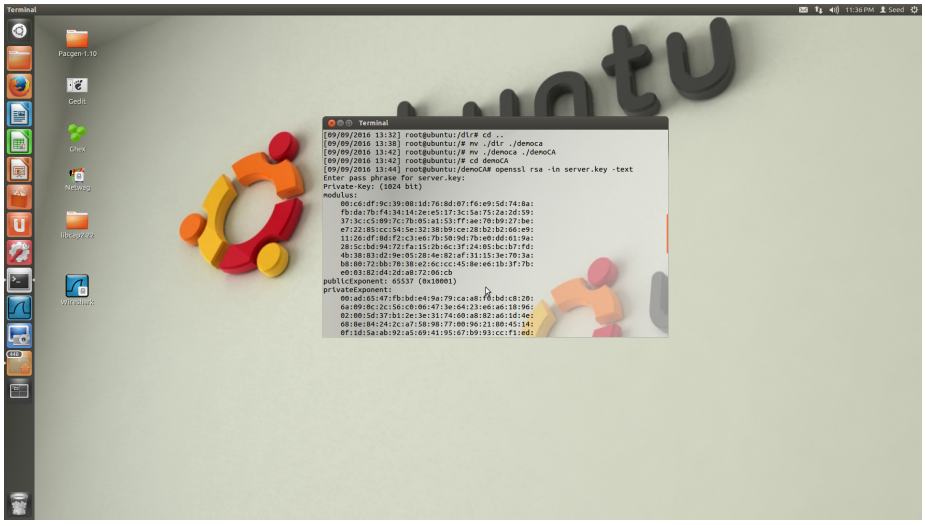

Figure 2: RSA

and asks for a password, which would then be used to encrypt the private key using AES-128 encryption algorithm, as shown in Figure 2.
$ openssl genrsa -aes128 -out server.key 1024

Observations: Both the keys are contained in the same file server.key. If you open the file, you only find the private key section. To use the public key, we would need to extract it from the server.key file.

2. Generate a Certificate Signing Request (CSR):
Using the command specified in the lab, we generate a CSR file (server.csr)
$ openssl req -new -key server.key -out server.csr -config openssl.cnf

3. Generating Certificates: We use the command specified in the manual to generate a certificate by giving server.csr as input, ca.key as the keyfile and ca.cert as the authorizing certificate.
$ openssl ca -in server.csr -out server.crt -cert ca.crt -keyfile ca.key config openssl.cnf

Observations: Initially, this command gave a *./demoCA/newcerts does not exist* error, as it was searching for the key and certificate files under */home/seed*. After moving the files there, the command executed and created a server.crt file.

3.3 Task 3: Use PKI for Web Sites:

1. Add PKILabServer.com mapping to localhost (127.0.0.1) to the /etc/hosts file so that everyone recognizes the domain name.
2. Combine server.key and server.crt into server.pem
3. Launch webserver using server.pem as follows,
$ openssl s_server -cert server.pem -www

4. Launching the browser and trying to access URL gives an error since the browser certificate list does not contain our own CA certificate. So, we add it to the list and try again.

Observations: As shown in Figure 3, the server can now be accessed and it gives a list of ciphers supported in s_server binary. It also lists the ciphers common between both SSL end points. Indicates that no client certificate is available.
Cipher used - ECDHE-RSA-AES256-SHA
Protocol - TLSv1

Subtask 1: Modify a single byte of server.pem, and restart the server, and reload the URL. What do you observe?

Observations: If you modify a byte in the private key section, the server would fail to restart, since it would be unable to load the certificate private key

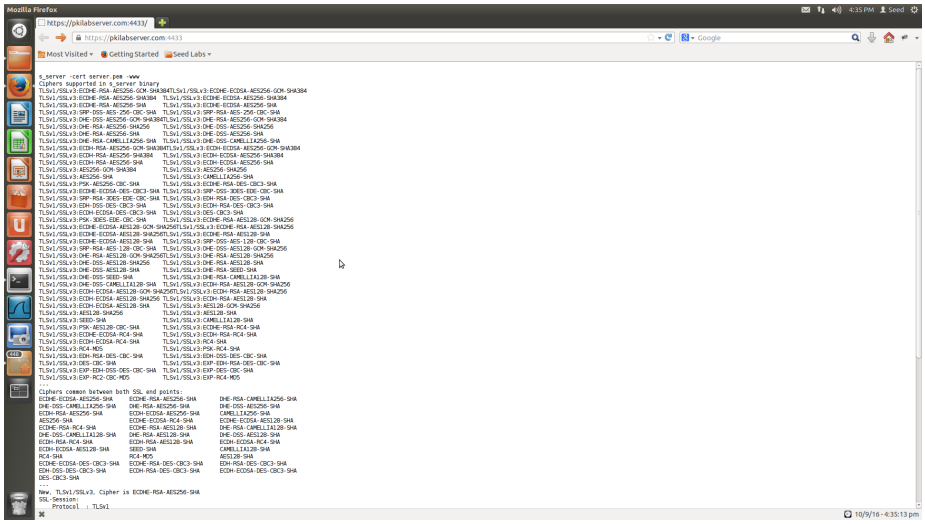

Figure 3: PKILabServer.com

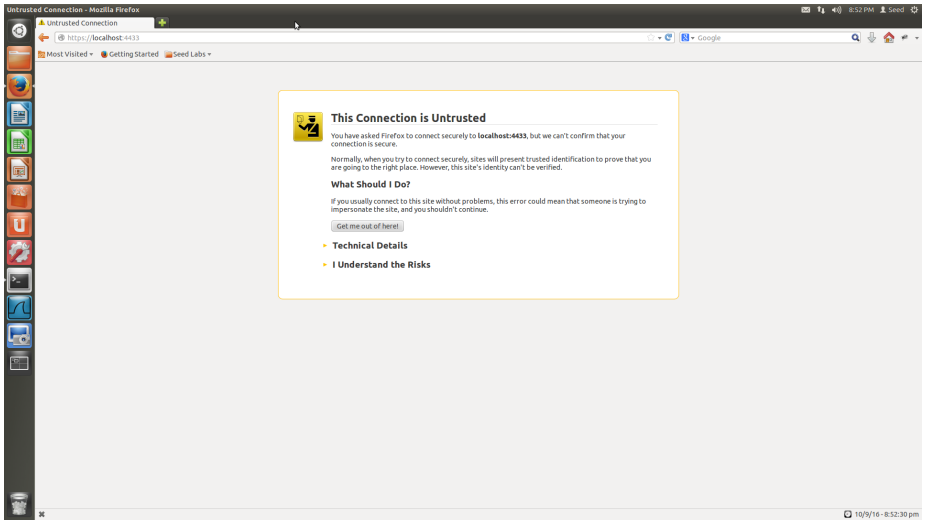

Figure 4: localhost

file. Although, changing a byte in the issuer section worked just fine. I changed the issuer email-id from harshyshah@ufl.edu to harsh.shah@ufl.edu using gHex and the URL gave the same result. This is because the cetificate is issued by the CA and making changes in server.pem do not affect the certificate stored by the browser. Even if we make changes to ca.crt, it would still work as the browser does not dynamically load the ca.crt in our case.

Subtask 2: Since PKILabServer.com points to the localhost, if we use https://localhost:4433 instead, we will be connecting to the same web server. Please do so, describe and explain your observations.

Observations: As seen in Figure 4, the browser is unable to verify the certificate even when PKILabServer.com points to localhost. This is because the browser checks for domain name only. So, even if you use the IP address 127.0.0.1 instead of PKILabServer.com, it would fail to recognize the certificate.

3.4 Task4: Establishing a TLS/SSL connection with server

In this task, we make use of public-key certificates to create secure connection between server and client. We run the cli.cpp and serv.cpp programs provided and observe that both the programs check for peer certificates to authenticate the other party.

Subtask 1: Please use the server certicate that you generated in Task 2 as the certicate for the server.

Observations: As seen in Figure 5, using the specified client and server keys and certificates, the connection works fine i.e. both parties are able to verify each other. Now, as seen in Figure 6, when we use our own server.crt, server.key and ca.crt, the client is still able to verify the server certificate, but the server throws an SSL routine error: no certificate returned.

Subtask 2: The client program needs to verify the server certicate. The verication consists of several checks. Please show where each check is conducted in your code (i.e., which line of your code does the corresponding check):
1. The effective date
In order to find the line of code that verifies the expiration date, I changed the system date using command: *$ sudo date –set="1 May 2000"*
The error thrown is SSL3_GET_SERVER_CERTIFICATE:
certificate verify failed:s3_client.c:1166:

2. Whether the server certicate is signed by an authorized CA
3. Whether the certicate belongs to the server
4. Whether the server is indeed the machine that the client wants to talk to(as opposed to a spoofed machine).

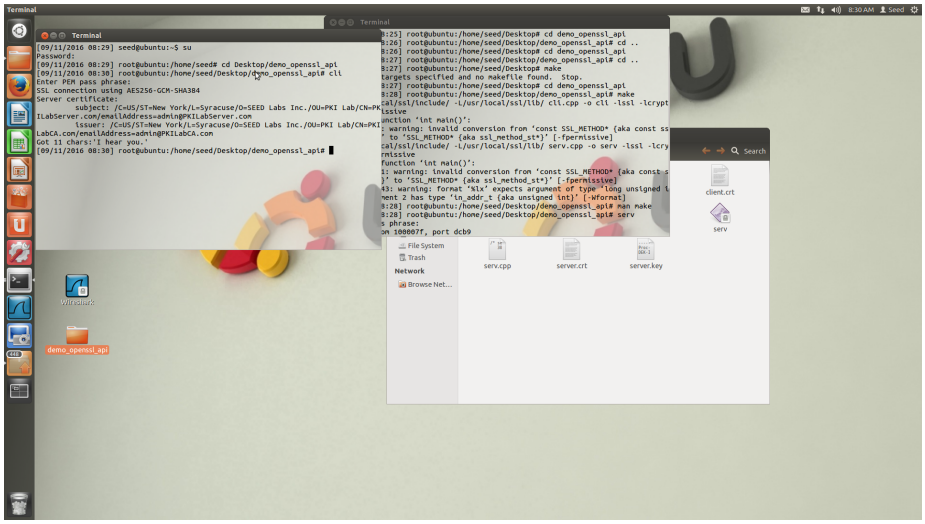

Figure 5: Default Server

Figure 6: Own Server

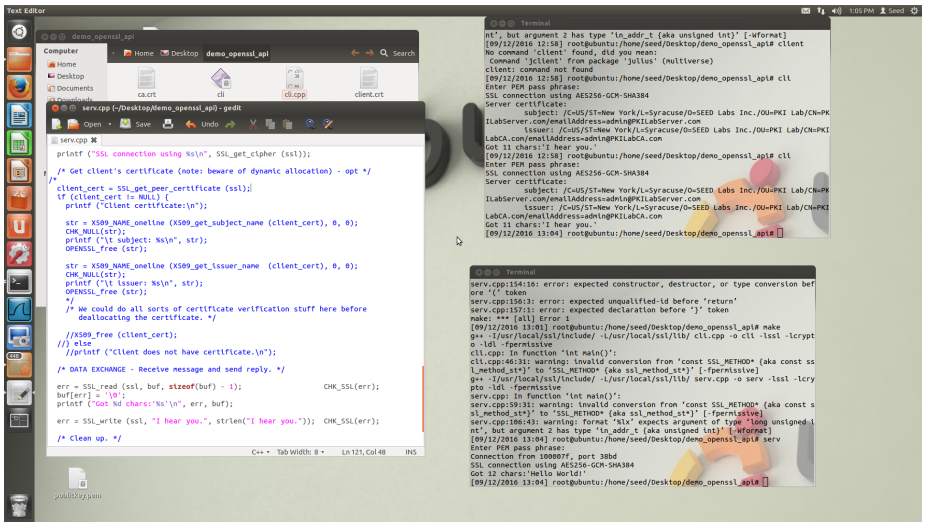

Figure 7: No Client Verification

Similarly for these checks, I made changes into the server.crt, server.key and ca.crt files and each of them threw the same error as above.

Subtask 3: The provided sample code for the server also verifies the clients certificate. We do not need this, please remove this part of code, and show us what changes you made in the server-side code.

As seen in Figure 7, I commented out the part where the server decides whether to verify the client certificate and the part where it acquires the certificate and performs verification checks. If we only comment out the part where it decides whether to verify the client, the program prints "Client does not have certificate." This can also be achieved by changing
SSL_CTX_set_verify(ctx,SSL_VERIFY_PEER,NULL); to
SSL_CTX_set_verify(ctx,NULL,NULL);

Subtask 4: What part of the code is responsible for the key exchange, i.e. for both sides to agree upon a secret key?

The line of code SSL_get_cipher(ssl), is responsible for the key exchange.

3.5 Task5: Performance Comparison: RSA versus AES

We create a message.txt file containing a 16-byte message. Using the following command, we generate an RSA public/private key pair and store it in task5.key file.
$ openssl genrsa -out task5.key 1024

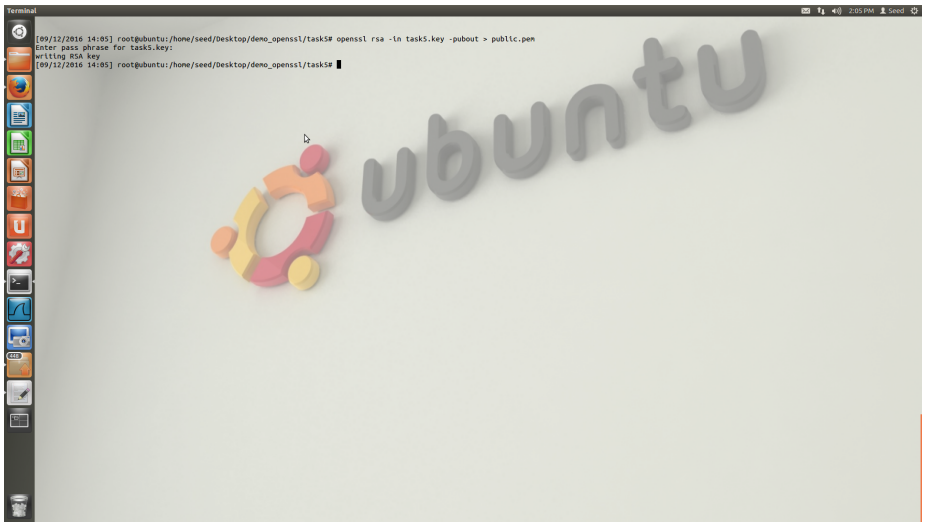

Figure 8: Public Key Extraction

Now, we extract the public key using the following command. This is shown in Figure 8.
$ openssl rsa -in task5.key -pubout >public.pem

The next step is to encrypt the message.txt file using RSA public key. This is shown in Figure 9, using the following command,
$ openssl rsautl -in message.txt -encrypt -pubin -inkey public.pem >message_enc.txt

Now, we decrypt message_enc.txt as shown in Figure 10, using RSA private key task5.key using the following command,
$ openssl rsautl -in message_enc.txt -decrypt -inkey task5.key -out decrypted.txt

Observations: On using the same key and same message, RSA encryption gives different output each time. This is because the algorithm does not encrypt pure data. Instead, it provides appropriate padding by appending some random pattern to the data before encryption. This random string is hidden by using XORing or hashing.

Let us now perform encryption and decryption using AES-128.
The encryption is shown in Figure 11, using the following command,
$ openssl aes-128-cbc -in message.txt -out message_aes_enc.txt

The decryption is shown in Figure 12, using the following command,
$ openssl aes-128-cbc -d -in message_aes_enc.txt -out plaintext.txt

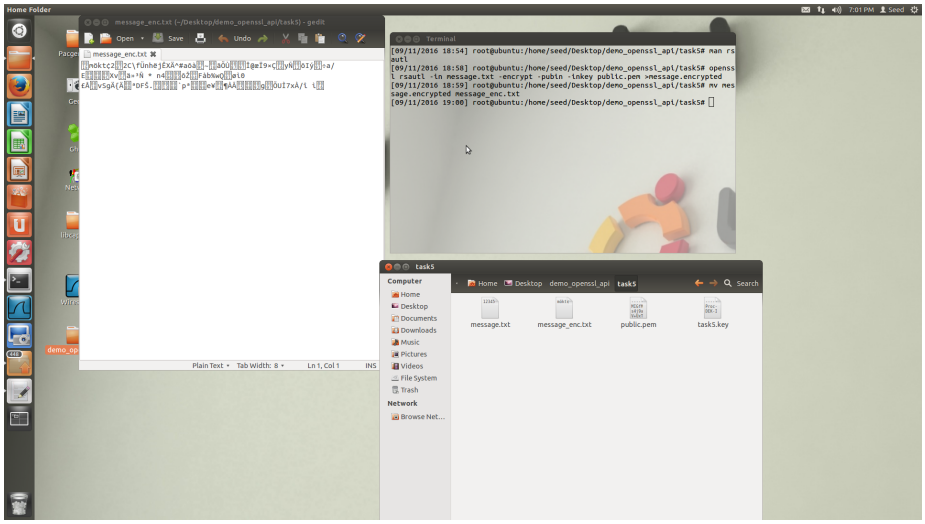

Figure 9: RSA encryption

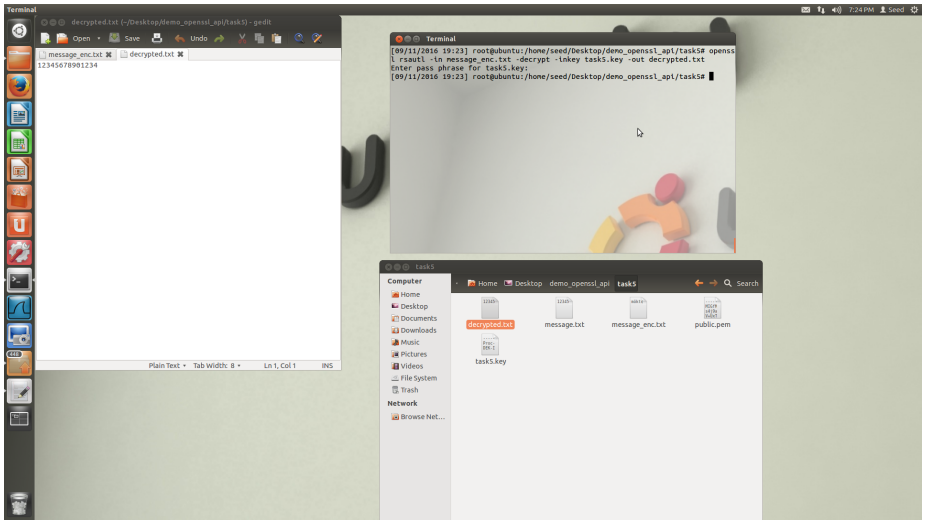

Figure 10: RSA decryption

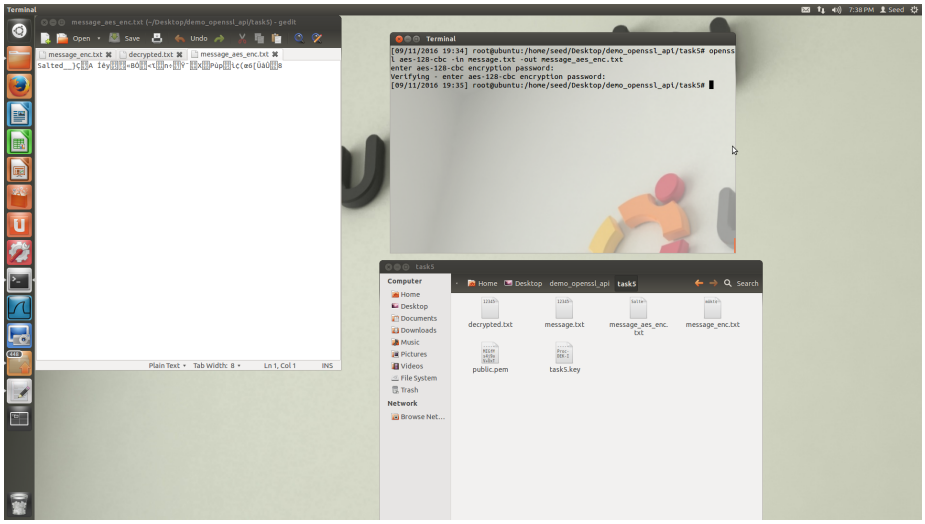

Figure 11: AES encryption

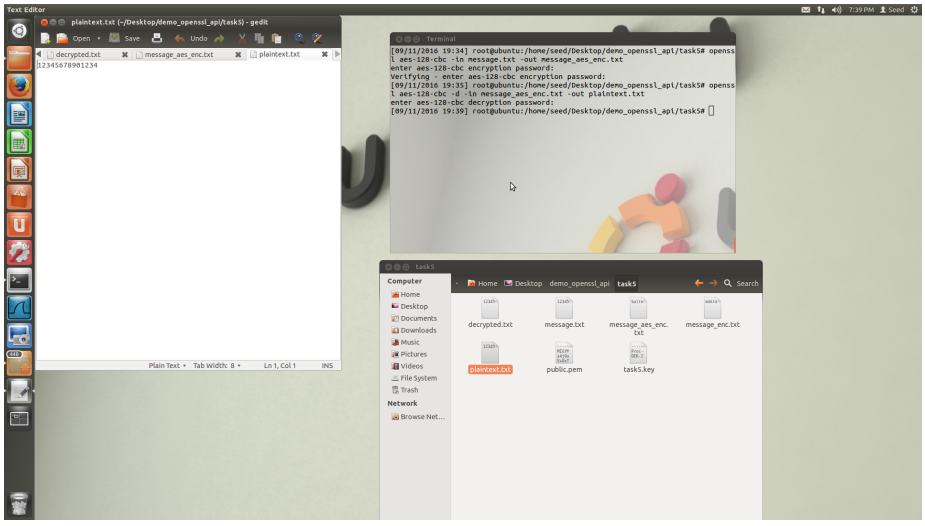

Figure 12: AES decryption

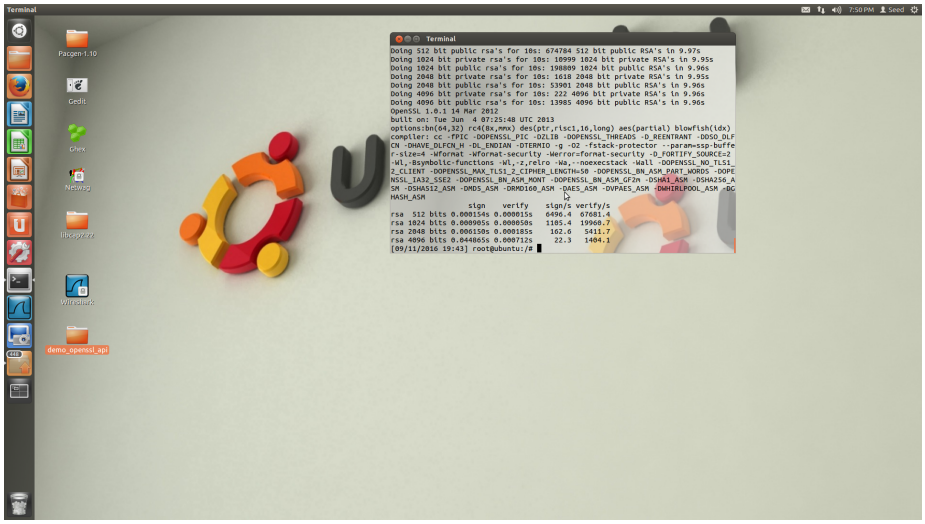

Figure 13: RSA speed

Figure 14: AES speed

Observations: Since, the size of the message is very small (16 bytes), there is no significant difference in the performance of the algorithms. The *Time* command also gave the output as 0.004s for both the algorithms. Although, on performing the process for a few number of times and manually checking speed using stopwatch, I found AES to be performing a little faster than RSA. But this does not answer the question as the time spent includes not only the encryption process, but also time spent outside CPU. For small inputs of data, it is extremely defficult to point out the difference in speeds.
Although, on executing the speed commands for benchmarking, as shown in Figure 13 and Figure 14, AES proves to be executing faster, which should be correct theoretically.

3.6 Task6: Create Digital Signature

We prepare a example.txt containing random text data ("Welcome to Computer & Network Security Assignment 2"). We generate an RSA key pair. The next step is to create a hash or a message digest, which is shown in Figure 15. The command used,
$ openssl dgst -sha256 example.txt¿hash.txt

The next step is to sign the SHA256 hash i.e. hash.txt, using the RSA private key. This is shown in Figure 16 and done using the following command,
$ openssl dgst -sha256 -sign rsaprivate.key -out example.sha256 hash.txt

To verify the digital signature, a client would calculate the hash of the message received, decrypt example.sha256 using the sender's public key and matching the decrypted value with the calculated hash of the message. If these values match, the signature is verified. This is shown in Figure 17 and is done using the following command,
$ openssl dgst -sha256 -verify publickey.pem -signature example.sha256 hash.txt

The final task is to slightly modify example.txt and check the digital signature. So, we assume that an intruder has managed to change the message slightly. So, the new message received by receiver is example.txt. ("Welcome to Computer & Network Security Assignment 1") We notice that "Assignment 2 has been changed to Assignment 1". Now, the receiver calculates the hash of the modified message using same method as above. The resulting hash file i.e. hashcheck.txt is verified against the Signature.txt received as follows,
$ openssl dgst -sha256 -verify publickey.pem -signature example.sha256 hashcheck.txt

As a result, verification is failed. This is shown in Figure 18.

Figure 15: hash

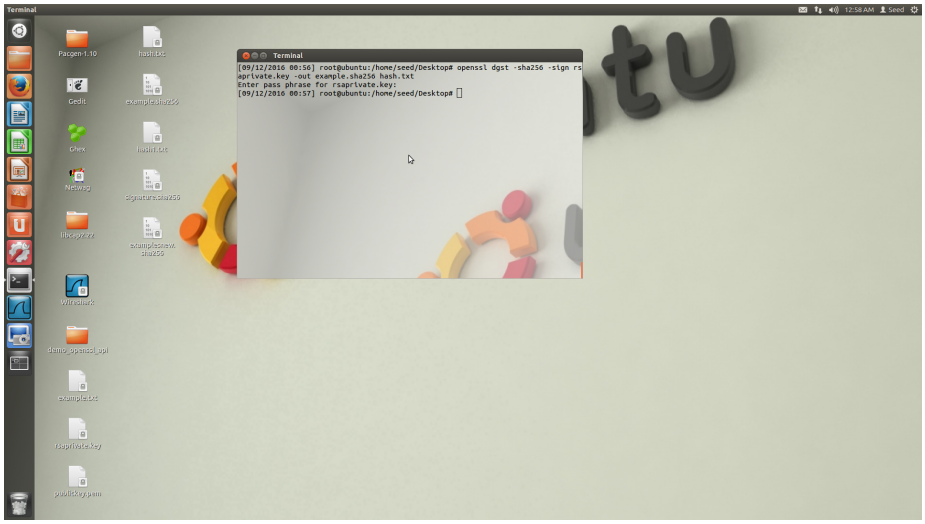

Figure 16: Signature

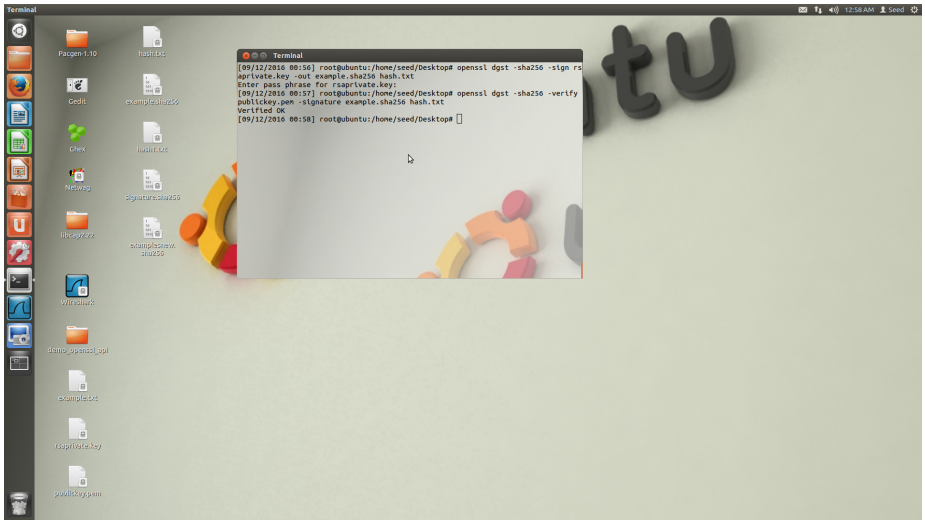

Figure 17: Verification OK

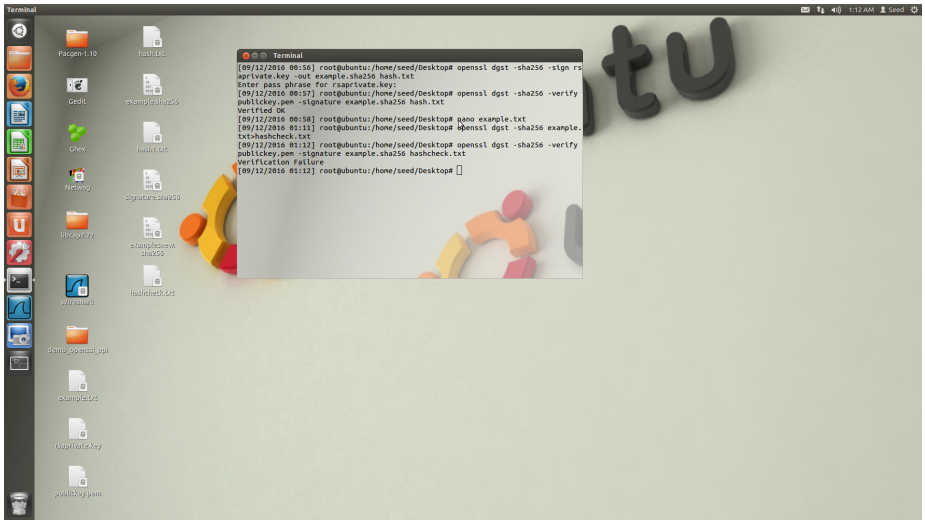

Figure 18: Verification Failure

Question: Please also explain why digital signatures are useful.

Answer: Digital Signatures provide both authenticity and data integrity. As seen above, if an intruder tries to modify the message, the receiver would easily detect it, as the hash of a different message is always different. Also, it provides non-repudiation. The sender cannot deny having sent the message as it was encrypted using his private key which is assumed to be available to him and him only.