OEIT1-Blockchain Technology & Applications

Lab1B-Cryptography & PKI using Openssl

Student Name: Adwait Purao Sem: VI Date: 12/02/24

Objective: To explore cryptography, Hash Functions, Digital Signature/Certificates

Outcomes:
Creation and management of private keys, public keys and parameters
Public key cryptographic operations
Creation of X.509 certificates, CSRs and CRLs
Calculation of Message Digests
Encryption and Decryption with Ciphers
SSL/TLS Client and Server Tests
Handling of S/MIME signed or encrypted mail
Time Stamp requests, generation and verification

System Requirements:
3 workstations installed with Kali Linux/Fedora Linux Core/Ubuntu and Windows XP
OpenSSL cryptography toolkit and gpg - OpenPGP encryption and signing tool

Introdcution to OpenSSL:
OpenSSL is among the most popular cryptography libraries. It is most commonly used to implement the Secure Sockets Layer and Transport Layer Security (SSL and TLS) protocols to ensure secure communications between computers. In recent years, SSL has become basically obsolete since TLS offers a higher level of security, but some people have gotten into the habit of referring to both protocols as "SSL."

Cryptography is tricky business, and OpenSSL has too many features to cover in one article, but this OpenSSL tutorial will help you get started creating keys and certificates.

Refer online reference [1]

An Introduction to Internet Security

When a client requests a secure connection to a server, the server, in turn, requests information to figure out which types of cryptographic security the client can support. Once it determines the most secure option, the following takes place:

The server sends a security certificate that is signed with the server's public key.
Once the client verifies the certificate, it generates a secret key and sends it to the server encrypted with the public key.
Next, both sides use the secret key to create two sets of public-private keys. At last, secure communication can commence.

SSL and TLS are two of many security protocols used to accomplish these steps. To implement these protocols, we need software like OpenSSL.

Abbreviations Key

You'll come across tons of abbreviations in this guide and other OpenSSL tutorials. For quick reference, here is a short list of some terms you might encounter:

CSR: Certificate Signing Request
DER: Distinguished Encoding Rules
PEM: Privacy Enhanced Mail
PKCS: Public-Key Cryptography Standards
SHA: Secure Hash Algorithm
SSL: Secure Socket Layer
TLS: Transport Layer Security

Procedure:
Open terminal in Linux (CTRL+ALT + T)
Read help pages of openssl and gpg
$man openssl
$man gpg
Use OpenSSL Cryptography toolkit

Part 1) Getting Started
$openssl version -a

To get a full list of the standard commands, enter the following:

$openssl list-standard-commands

Check out the official OpenSSL docs for explanations of the standard commands. To view the many secret key algorithms available in OpenSSL, use:

$openssl list-cipher-commands

Now, let's try some encryption. If you wanted to encrypt the text "Hello World!" with the AES algorithm using CBC mode and a 256-bit key, you would do as follows:

$touch plain.txt
$echo "Hello World!" > plain.txt
$openssl enc -aes-256-cbc -in plain.txt -out encrypted.bin
//enter aes-256-cbc encryption password: example
//Verifying - enter aes-256-cbc encryption password: example

```
adwait_purao@itlab-OptiPlex-3000:/tmp$ touch plain.txt
adwait_purao@itlab-OptiPlex-3000:/tmp$ echo "Hello World!" > plain.txt
adwait_purao@itlab-OptiPlex-3000:/tmp$ openssl enc -aes-256-cbc -in plain.txt -out encrypted.bin
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
adwait_purao@itlab-OptiPlex-3000:/tmp$ openssl enc -aes-256-cbc -d -in encrypted.bin -pass pass:e
xample
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
Hello World!
```

You'll be prompted to enter a password from which the 256-bit secret key will be computed. In the above example, the password "example" is used, but you should have stronger passwords. You should now have a binary file called encrypted.bin that you can decrypt as follows:

$openssl enc -aes-256-cbc -d -in encrypted.bin -pass pass:example
// Hello World!

```
adwait_purao@itlab-OptiPlex-3000:/tmp$ openssl enc -aes-256-cbc -in plain.txt -out encrypted.bin
enter AES-256-CBC encryption password:
Verifying - enter AES-256-CBC encryption password:
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
```

Part 2) Public and Private Keys

For the sake of example, we can demonstrate how OpenSSL manages public keys using the RSA algorithm. You can use other algorithms of course, and the same principles will apply. The first step is to generate public and private pairs of keys. Enter the following command to create an RSA key of 1024 bits:

$openssl genrsa -out key.pem 1024

You should now have a file called key.pem containing a public key and private key. As the file's name suggests, the private key is coded using the Privacy Enhanced Email, or PEM, standard. Use the following code to display it:

$cat key.pem

You should see a long combination of characters. For detailed information about how your key was generated, enter:

```
adwait_purao@itlab-OptiPlex-3000:/tmp$ openssl genrsa -out key.pem 1024
adwait_purao@itlab-OptiPlex-3000:/tmp$ cat key.pem
-----BEGIN PRIVATE KEY-----
MIICdgIBADANBgkqhkiG9w0BAQEFAASCAmAwggJcAgEAAoGBAJ5pQ/tqNjv6SqEv
GDE+Ob8GzSKGbRyEF4h7JMtieowvlTW/zve5azSNhfWwjOyqaPQSjnv+Kj/ew9UG
kzpaPDSyBXhOaCq5bnjlm87S0l5rgGUpuF4wj4gcjnVg2LOOOgdTP7tj3rnsSXZK
STHeVnayBK1I9+3QKYqJ7yh3mXRlAgMBAAECgYBzCczlwq8aBVy3CwVNlXvIHv9P
qEGKqN6SLaN908ldDp0UCHQb8FpLMs9qqcjdga9wOB2jK+OwuGyMwhafFLKNCX/t
EApwI1Hxe3F4Xg3VS67BHzNIqR8MaNx1PYhrUKcR2e1XCTbci86cE5p+0OGP3I2T
v9ww8FChQv5ftzASYQJBAM3lhlAkPwqE+eg4u9tl2rIuj66KUNJHmZ/Bj+hHz4QF
ggqTUogoLkKXt2Uw+oWPzwS3v9i4h34gyKJKiYOd3WkCQQDE9ZrHe8TshEVTUNys
jEUa2FG8J8PWAc2v7CyjyaDxSw+iu2ljRhK62jDJmChPhE0+3PJxpCB431ul1sq3
7vOdAkA7tfqzty9JH7NR81IXmHxJ/FLS7W3u7bDRsdLILDs0MFy8MGlY7z4rNdCL
2h5tKX2CAr/yW+F/ZIaK+k7TCq8xAkB3fQUEhnrmfixMql7mM0btTVTw0DMHzxpI
4wAF8Zm5KS94Kzh5aJaOUzaL5sp1yy7Nt19CQC8PBXplNtTPF6ElAkEAgTELNTmr
IAtSgWUFfppKwN+DfHbovH53kvh6H1vGpeoHiSCULa532CgtnpD7lRZde1oeSvNt
WqeglS3qGLP0FA==
-----END PRIVATE KEY-----
```

$openssl rsa -in key.pem -text -noout

This command should return information about the public and private exponents, the modulus and the other methods and numbers used to optimize the algorithm. In this context, the -noout option prevents display of the key in base 64 format, which means that only hexadecimal numbers are visible. The public exponent is an exception, of course, since it is always 65537 for 1024 bit keys.

To encrypt our private key, we use the following code:

```
adwait_purao@itlab-OptiPlex-3000:/tmp$ openssl rsa -in key.pem -text -noout
Private-Key: (1024 bit, 2 primes)
modulus:
    00:9e:69:43:fb:6a:36:3b:fa:4a:a1:2f:18:31:3e:
    39:bf:06:cd:22:86:6d:1c:84:17:88:7b:24:cb:62:
    7a:8c:2f:95:35:bf:ce:f7:b9:6b:34:8d:85:f5:b0:
    8c:ec:aa:68:f4:12:8e:7b:fe:2a:3f:de:c3:d5:06:
    93:3a:5a:3c:34:b2:05:78:4e:68:2a:b9:6e:78:e5:
    9b:ce:d2:d2:5e:6b:80:65:29:b8:5e:30:8f:88:1c:
    8e:75:60:d8:b3:8e:3a:07:53:3f:bb:63:de:b9:ec:
    49:76:4a:49:31:de:56:76:b2:04:ad:48:f7:ed:d0:
    29:8a:89:ef:28:77:99:74:65
publicExponent: 65537 (0x10001)
privateExponent:
    73:09:cc:e5:c2:af:1a:05:5c:b7:0b:05:4d:95:7b:
    c8:1e:ff:4f:a8:41:8a:a8:de:92:2d:a3:7d:d3:c9:
    5d:0e:9d:14:08:74:1b:f0:5a:4b:32:cf:6a:a9:c8:
    dd:81:af:70:38:1d:a3:2b:e3:b0:b8:6c:8c:c2:16:
    9f:14:b2:8d:09:7f:ed:10:0a:70:23:51:f1:7b:71:
    78:5e:0d:d5:4b:ae:c1:1f:33:48:a9:1f:0c:68:dc:
    75:3d:88:6b:50:a7:11:d9:ed:57:09:36:dc:8b:ce:
    9c:13:9a:7e:d0:e1:8f:dc:8d:93:bf:dc:30:f0:50:
    a1:42:fe:5f:b7:30:12:61
prime1:
    00:cd:e5:86:50:24:3f:0a:84:f9:e8:38:bb:db:65:
    da:b2:2e:8f:ae:8a:50:d2:47:99:9f:c1:8f:e8:47:
    cf:84:05:82:0a:93:52:88:28:2e:42:97:b7:65:30:
    fa:85:8f:cf:04:b7:bf:d8:b8:87:7e:20:c8:a2:4a:
    89:83:9d:dd:69
prime2:
    00:c4:f5:9a:c7:7b:c4:ec:84:45:53:50:dc:ac:8c:
    45:1a:d8:51:bc:27:c3:d6:01:cd:af:ec:2c:a3:c9:
    a0:f1:4b:0f:a2:bb:69:63:46:12:ba:da:30:c9:98:
    28:4f:84:4d:3e:dc:f2:71:a4:20:78:df:5b:a5:d6:
    ca:b7:ee:f3:9d
exponent1:
    3b:b5:fa:b3:b7:2f:49:1f:b3:51:f3:52:17:98:7c:
    49:fc:52:d2:ed:6d:ee:ed:b0:d1:b1:d2:c8:2c:3b:
    34:30:5c:bc:30:69:58:ef:3e:2b:35:d0:8b:da:1e:
    6d:29:7d:82:02:bf:f2:5b:e1:7f:64:86:8a:fa:4e:
    d3:0a:af:31
exponent2:
    77:7d:05:04:86:7a:e6:7e:2c:4c:aa:5e:e6:33:46:
```

```
publicExponent: 65537 (0x10001)
privateExponent:
    73:09:cc:e5:c2:af:1a:05:5c:b7:0b:05:4d:95:7b:
    c8:1e:ff:4f:a8:41:8a:a8:de:92:2d:a3:7d:d3:c9:
    5d:0e:9d:14:08:74:1b:f0:5a:4b:32:cf:6a:a9:c8:
    dd:81:af:70:38:1d:a3:2b:e3:b0:b8:6c:8c:c2:16:
    9f:14:b2:8d:09:7f:ed:10:0a:70:23:51:f1:7b:71:
    78:5e:0d:d5:4b:ae:c1:1f:33:48:a9:1f:0c:68:dc:
    75:3d:88:6b:50:a7:11:d9:ed:57:09:36:dc:8b:ce:
    9c:13:9a:7e:d0:e1:8f:dc:8d:93:bf:dc:30:f0:50:
    a1:42:fe:5f:b7:30:12:61
prime1:
    00:cd:e5:86:50:24:3f:0a:84:f9:e8:38:bb:db:65:
    da:b2:2e:8f:ae:8a:50:d2:47:99:9f:c1:8f:e8:47:
    cf:84:05:82:0a:93:52:88:28:2e:42:97:b7:65:30:
    fa:85:8f:cf:04:b7:bf:d8:b8:87:7e:20:c8:a2:4a:
    89:83:9d:dd:69
prime2:
    00:c4:f5:9a:c7:7b:c4:ec:84:45:53:50:dc:ac:8c:
    45:1a:d8:51:bc:27:c3:d6:01:cd:af:ec:2c:a3:c9:
    a0:f1:4b:0f:a2:bb:69:63:46:12:ba:da:30:c9:98:
    28:4f:84:4d:3e:dc:f2:71:a4:20:78:df:5b:a5:d6:
    ca:b7:ee:f3:9d
exponent1:
    3b:b5:fa:b3:b7:2f:49:1f:b3:51:f3:52:17:98:7c:
    49:fc:52:d2:ed:6d:ee:ed:b0:d1:b1:d2:c8:2c:3b:
    34:30:5c:bc:30:69:58:ef:3e:2b:35:d0:8b:da:1e:
    6d:29:7d:82:02:bf:f2:5b:e1:7f:64:86:8a:fa:4e:
    d3:0a:af:31
exponent2:
    77:7d:05:04:86:7a:e6:7e:2c:4c:aa:5e:e6:33:46:
    ed:4d:54:f0:d0:33:07:cf:1a:48:e3:00:05:f1:99:
    b9:29:2f:78:2b:38:79:68:96:8e:53:36:8b:e6:ca:
    75:cb:2e:cd:b7:5f:42:40:2f:0f:05:7a:65:36:d4:
    cf:17:a1:25
coefficient:
    00:81:31:0b:35:39:ab:20:0b:52:81:65:05:7e:9a:
    4a:c0:df:83:7c:76:e8:bc:7e:77:92:f8:7a:1f:5b:
    c6:a5:ea:07:89:20:94:2d:ae:77:d8:28:2d:9e:90:
    fb:95:16:5d:7b:5a:1e:4a:f3:6d:5a:a7:a0:95:2d:
    ea:18:b3:f4:14
```

$openssl rsa -in key.pem -des3 -out enc-key.pem

Once the key file has been encrypted, you will then be prompted to create a password. Next, we can extract the public key from the file key.pem with this command:

$openssl rsa -in key.pem -pubout -out pub-key.pem

```
adwait_purao@itlab-OptiPlex-3000:/tmp$ openssl rsa -in key.pem -des3 -out enc-key.pem
writing RSA key
Enter pass phrase:
Verifying - Enter pass phrase:
adwait_purao@itlab-OptiPlex-3000:/tmp$ openssl rsa -in key.pem -pubout -out pub-key.pem
writing RSA key
```

Finally, we are ready to encrypt a file using our keys. Use the following format:

openssl pkeyutl -encrypt -in <input_file> -inkey <key.pem> -out <output_file>

```
adwait_purao@itlab-OptiPlex-3000:/tmp$ openssl pkeyutl -encrypt -in plain.txt -inkey key.pem -out output1.txt
adwait_purao@itlab-OptiPlex-3000:/tmp$
```

In the above context, <input_file> is the file you want to encrypt. Since we're using RSA, keep in mind that the file can't exceed 116 bytes. The <key.pem> is the file containing the public key. If that file doesn't also include the private key, you must indicate so using -pubin. The <output_file> is the encrypted file name.

Now, to decrypt the file, you can simply flip the equation: Change -encrypt to -decrypt, and switch the input and output files.

```
adwait_purao@itlab-OptiPlex-3000:/tmp$ openssl pkeyutl -decrypt -in output1.txt -inkey key.pem -out plain.txt
```

Part 3) Creating Digital Signatures

At last, we can produce a digital signature and verify it. Signing a large file directly with a public key algorithm is inefficient, so we should first compute the digest value of the information to be signed. This can be accomplished using the following command:

openssl dgst -<hash_algorithm> -out <digest> <input_file>

```
adwait_purao@itlab-OptiPlex-3000:/tmp$ openssl dgst -sha1 -out digest.txt plain.txt
```

In this example, <hash_algorithm> is whichever algorithm you choose to compute the digest value. The <input_file> is the file containing the data you want to hash while "digest" is the file that will contain the results of the hash application.

The next step is to compute the signature of the digest value as follows:

openssl pkeyutl -sign -in <digest> -out <signature> -inkey <key>

```
adwait_purao@itlab-OptiPlex-3000:/tmp$ echo "a0b65939670bc2c010f4d5d6a0b3e4e4590fb92b" openssl pkeyutl -sign -in digest.txt -out signature.txt -inkey
key.pem
```

Finally, you can check the validity of a signature like so:

openssl pkeyutl -verify -in <signature> -out <digest> -inkey <key> -pubin

```
adwait_purao@itlab-OptiPlex-3000:/tmp$ echo "a0b65939670bc2c010f4d5d6a0b3e4e4590fb92b" openssl pkeyutl -verify -in signature.txt -out verified_digest.
txt -inkey key.pem -pubin
a0b65939670bc2c010f4d5d6a0b3e4e4590fb92b openssl pkeyutl -verify -in signature.txt -out verified_digest.txt -inkey key.pem -pubin
```

Here, “signature” is the filename of your signature, and “key.pem” is the file with the public key. To confirm the verification for yourself, you can compute the digest value of the input file and compare it to the digest value produced from the verification of the digital signature.

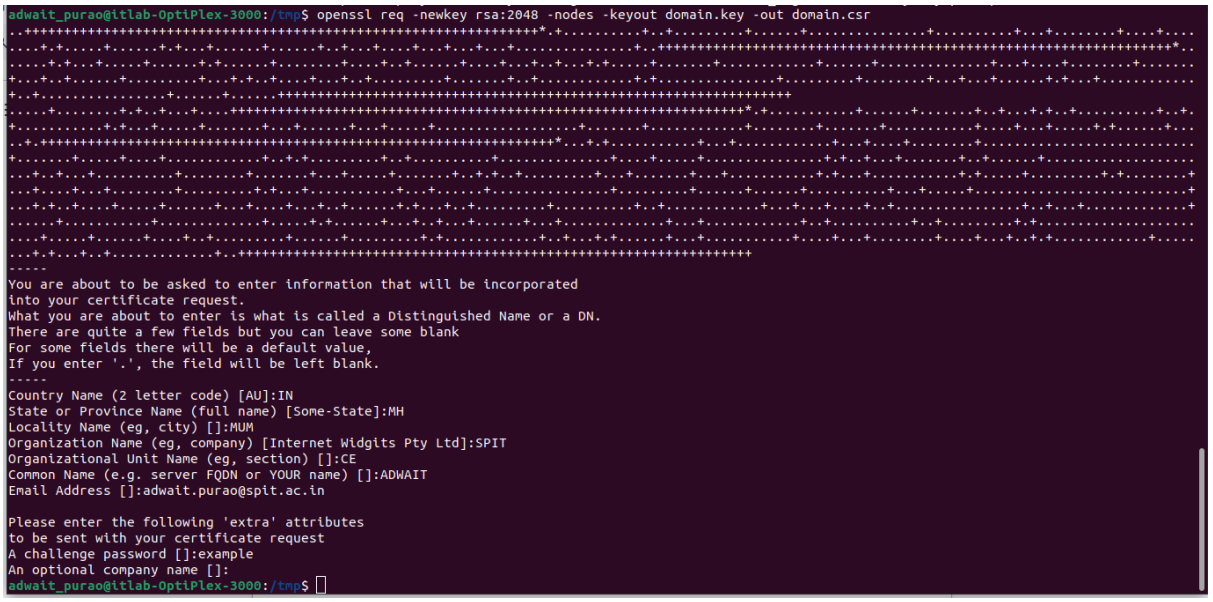
Part 4) Certificate Signing Requests

Let’s say that you want to create digital certificates signed by your own certificate authority. Before you can get an SSL certificate from a certificate authority, or CA, you must first generate a certificate signing request or a CSR. A CSR includes a public key as well as some extra information that gets inserted into the certificate when signed.

When you first create a CSR, you’ll be asked to supply some information about yourself or your organization. In the field “Common Name,” or CN, you must provide the fully qualified domain name of the host for which the certificate is intended. If you’re actually purchasing an SSL certificate from a CA, then the information you provide should be factual and accurate! Imagine you want to secure an Apache HTTP or Nginx web server with HTTPS. You can use the following snippet to create a new 2048-bit private key along with a CSR from scratch:

Imagine you want to secure an Apache HTTP or Nginx web server with HTTPS. You can use the following snippet to create a new 2048-bit private key along with a CSR from scratch:

$openssl req -newkey rsa:2048 -nodes -keyout domain.key -out domain.csr



Just replace “domain” with your domain name. Here, the -newkey rsa:2048 option tells OpenSSL that it should use the RSA algorithm to create a 2048-bit key, and the -nodes option indicates that the key shouldn’t be password protected.

After you’ve provided all of the necessary information, your CSR will be generated. Now, you can send it to a CA and request an SSL certificate. If your CA supports SHA-2, be sure to add the -sha256 option if you want your CSR to be signable with SHA-2.

To create a CSR for a private key that already exists, you would use this format:

*$*openssl req -key domain.key -new -out domain.csr

```
adwait_purao@itlab-OptiPlex-3000:/tmp$ openssl req -key domain.key -new -out domain.csr
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]:IN
State or Province Name (full name) [Some-State]:MH
Locality Name (eg, city) []:MUM
Organization Name (eg, company) [Internet Widgits Pty Ltd]:SPIT
Organizational Unit Name (eg, section) []:CE
Common Name (e.g. server FQDN or YOUR name) []:ADWAIT
Email Address []:adwait.purao@spit.ac.in

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

Again, replace “domain” with your domain name. The -key option here indicates that you’re using an existing private key while -new indicates that you’re creating a new CSR. In the prior example, -new was implied because you were making a new key.

Let’s say you already have a certificate that you want to renew, yet you somehow lost the original CSR. Don’t panic; you can generate a new one **based on information from your certificate and the private key.** For example, if you were using an X509 certificate, you’d use the following code:

$openssl x509 -in domain.crt -signkey domain.key -x509toreq -out domain.csr

```
adwait_purao@itlab-OptiPlex-3000:/tmp$ openssl x509  -in domain.crt  -signkey domain.key  -x509toreq -out domain.csr
Could not open file or uri for loading certificate from domain.crt
4017F85DA37F0000:error:16000069:STORE routines:ossl_store_get0_loader_int:unregistered scheme:../crypto/store/store_register.c:237:scheme=file
4017F85DA37F0000:error:80000002:system library:file_open:No such file or directory:../providers/implementations/storemgmt/file_store.c:267:calling sta
t(domain.crt)
Unable to load certificate
adwait_purao@itlab-OptiPlex-3000:/tmp$ openssl req -key domain.key -new -x509 -days 365 -out domain.crt
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]:IN
State or Province Name (full name) [Some-State]:MH
Locality Name (eg, city) []:MUM
Organization Name (eg, company) [Internet Widgits Pty Ltd]:SPIT
Organizational Unit Name (eg, section) []:CE
Common Name (e.g. server FQDN or YOUR name) []:ADWAIT
Email Address []:adwait.purao@spit.ac.in
adwait_purao@itlab-OptiPlex-3000:/tmp$ openssl x509  -in domain.crt  -signkey domain.key  -x509toreq -out domain.csr
```

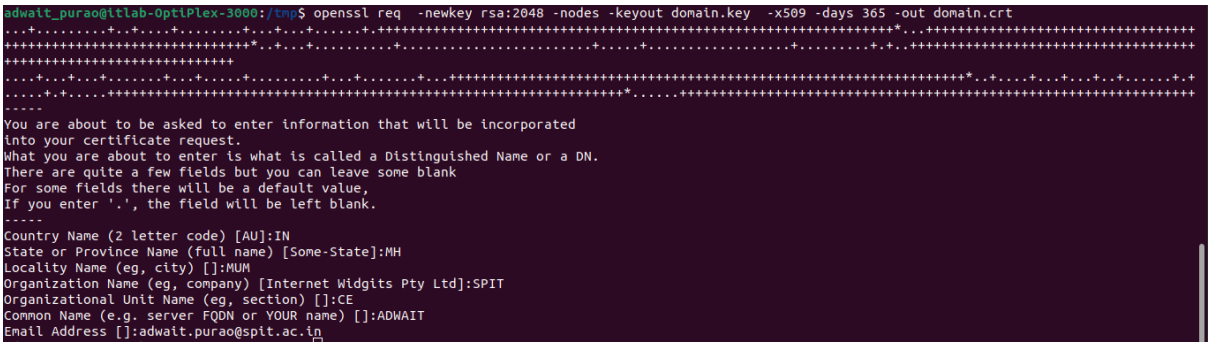
The -x509toreq option is needed to let OpenSSL know the certificate type.

Part 5) Generating SSL Certificates

If you desire the extra security of an SSL certificate, but you can’t afford or don’t want to be bothered with a CA, a less expensive alternative is to sign your own certificates. Self-signed certificates are signed with their own private keys, and they are just as effective at encrypting data as CA-signed certificates; however, users might receive an alert from their browser indicating that the connection is not secure, so self-signed certificates are really only recommended in environments where you’re not required to prove your service’s identity such as on a non-public server.

Again, assume that you’re using HTTPS to secure an Apache HTTP or Nginx web server. The following command will create a 2048-bit private key along with a self-signed certificate:

$openssl req -newkey rsa:2048 -nodes -keyout domain.key -x509 -days 365 -out domain.crt



The -x509 option tells OpenSSL that you want a self-signed certificate, while -days 365 indicates that the certificate should be valid for a year. OpenSSL will generate a temporary CSR for the purpose of gathering information to associate with the certificate, so you will have to answer the prompts per usual.

If you already have a private key that you want to associate with a self-signed certificate, just change the code slightly:

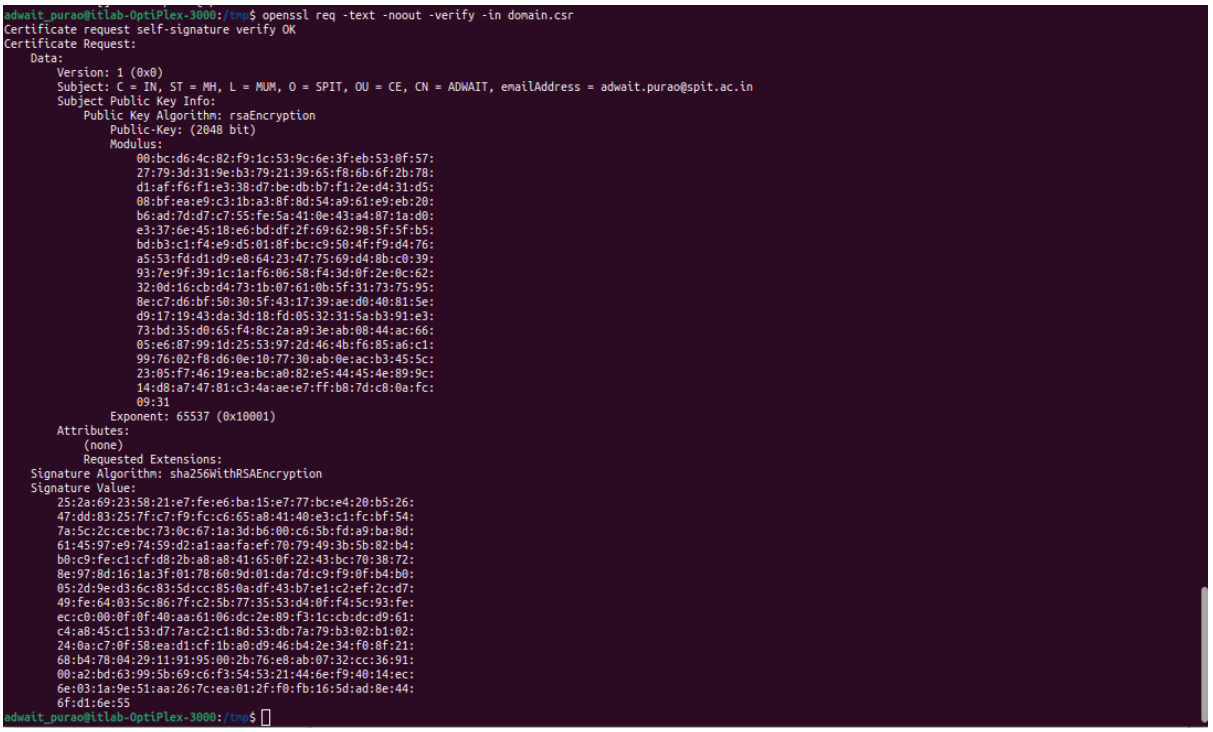
$openssl req -key domain.key -new -x509 -days 365 -out domain.crt

Remember that inclusion of the -new option is necessary since you are creating a new CSR from an existing key.

Part 6) Viewing Certificates

To view certificates and CSR files, you must decode them from the PEM format. Use the following command to view the contents of a CSR in plain text:

$openssl req -text -noout -verify -in domain.csr



To view a certificate's content in plain text, use:

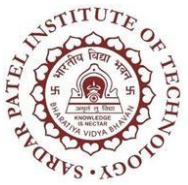
$openssl x509 -text -noout -in domain.crt

```
adwait_purao@itlab-OptiPlex-3000:/tmp$ openssl x509 -text -noout -in domain.crt
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            07:eb:ee:fb:4e:e8:64:66:56:b7:ca:6c:ea:93:0e:c7:dd:5d:5a:ce
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = IN, ST = MH, L = MUM, O = SPIT, OU = CE, CN = ADWAIT, emailAddress = adwait.purao@spit.ac.in
        Validity
            Not Before: Feb 12 10:37:54 2024 GMT
            Not After : Feb 11 10:37:54 2025 GMT
        Subject: C = IN, ST = MH, L = MUM, O = SPIT, OU = CE, CN = ADWAIT, emailAddress = adwait.purao@spit.ac.in
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:cc:fb:32:f5:cc:b4:a6:05:47:14:84:f3:9e:64:
                    3e:3c:85:54:04:ee:c2:cc:b9:b0:29:a6:98:f6:72:
                    7e:3d:a6:90:87:7b:f5:fd:ef:81:33:a1:de:b6:ea:
                    44:77:33:78:f1:05:99:42:e0:6b:be:8d:db:90:a8:
                    e7:4d:7f:90:05:9b:61:5b:b4:d7:f2:82:49:38:7b:
                    c1:a6:82:52:db:11:8a:c0:7b:5d:aa:2e:f2:08:00:
                    87:56:51:69:bb:a1:b0:4a:1f:c9:d2:cd:02:0a:b3:
                    56:13:3b:12:53:0d:dd:13:f6:2a:d8:19:8d:bb:f7:
                    46:c2:76:5f:76:ff:0c:bb:98:2a:fd:86:36:8b:4c:
                    6e:e1:6a:f6:32:95:1a:dd:94:6d:e8:b9:33:49:c4:
                    74:39:40:ad:59:2e:1c:00:25:92:a8:50:fc:01:28:
                    21:46:53:65:03:da:4c:b9:7d:20:f2:b3:fe:ef:9a:
                    8e:41:83:93:61:f7:b5:4f:47:16:bb:68:57:43:7d:
                    4b:d3:72:75:95:85:55:64:cb:4b:fa:09:5d:64:50:
                    dc:a2:b7:52:fa:40:33:d4:c3:88:ae:85:d6:09:22:
                    35:e7:30:51:a1:11:bd:a0:16:58:3a:38:02:04:b7:
                    c4:9d:84:ab:83:b7:ff:16:21:c6:a0:17:ce:35:53:
                    7c:93
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                EB:99:15:DE:38:A8:44:7A:80:07:7A:F3:1D:DF:10:C3:63:AE:C4:8E
            X509v3 Authority Key Identifier:
                EB:99:15:DE:38:A8:44:7A:80:07:7A:F3:1D:DF:10:C3:63:AE:C4:8E
            X509v3 Basic Constraints: critical
                CA:TRUE
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
        43:96:86:6d:d8:1f:5a:02:c3:05:0b:a1:82:ed:fc:9b:bf:55:
        60:e0:df:47:aa:c6:1c:ad:f1:44:0f:8a:49:3c:e4:55:14:2f:
        ae:51:42:53:d0:9f:98:25:67:6c:ce:3f:51:cd:eb:a4:93:62:
        79:da:3a:05:4f:df:33:e9:88:ec:a8:67:cd:b1:8e:88:24:e7:
        77:f4:24:7b:a8:64:6a:3a:dc:d2:70:bb:7c:97:e2:2d:26:2d:
        10:44:96:49:eb:c2:77:15:cf:42:56:bd:8c:62:30:65:95:da:
        88:15:a4:05:89:70:bc:26:bf:42:cc:4d:c1:d6:c1:2b:7f:33:
        10:cc:2d:3a:ee:42:6d:1e:4f:bc:74:f8:d6:0f:46:fd:bf:1f:
        4c:eb:74:d3:df:05:93:fe:4d:6c:34:02:82:92:1e:c5:3b:54:
        21:77:c3:9f:87:85:ea:f4:60:10:42:7f:cc:8f:47:2e:55:2e:
        20:11:00:22:9f:4c:81:91:bb:09:35:38:c5:05:6b:44:e1:fc:
        c6:a1:eb:39:ac:4e:e1:8b:0b:65:ed:07:73:20:e2:57:6e:02:
        b9:bc:e2:33:13:a8:41:ad:30:23:0a:aa:df:45:ad:01:15:e5:
        4d:6e:0d:48:00:57:d0:97:f1:42:62:71:52:1f:d5:ba:37:c7:
        a6:34:e8:34
adwait_purao@itlab-OptiPlex-3000:/tmp$
```

You can verify that a certificate was signed by a specific CA by plugging its name into the following code:

$openssl verify -verbose -CAFile ca.crt domain.crt

```
adwait_purao@itlab-OptiPlex-3000:/tmp$ openssl verify -verbose -CAfile domain.crt domain.crt
domain.crt: OK
adwait_purao@itlab-OptiPlex-3000:/tmp$
```



OEIT1-Blockchain Technology & Applications

OpenSSL Summary:

In today's increasingly digital world, improving internet security is imperative to protect our own security. Many website databases contain treasure troves of information about visitors, and hackers are always learning new ways to navigate system vulnerabilities and exploit such data. That's why security protocols must continue to evolve. Stay informed to make sure you're providing adequate protection for your users.

Conclusion:

The experiment focused on exploring various aspects of cryptography, including the creation and management of private keys, public keys, and parameters, public key cryptographic operations, creation of X.509 certificates, Certificate Signing Requests (CSRs), and Certificate Revocation Lists (CRLs). Additionally, the experiment involved the calculation of message digests, encryption and decryption with ciphers, SSL/TLS client and server tests, handling of S/MIME signed or encrypted mail, and time stamp requests, generation, and verification.

References:

https://www.keycdn.com/blog/openssl-tutorial#part-3-creating-digital-signatures
https://www.madboa.com/geek/openssl/