

# APPROACH

## Problem Statement :

Cryptography Simulation with mbedTLS/OpenSSL Library Usage and User Interaction.

## Team Name

Adventurers

## Team Size

2 ( TWO )

## Team Member 1

Karthikeyan A

## Team Member 2

Sri Ganesh R

## College Name :

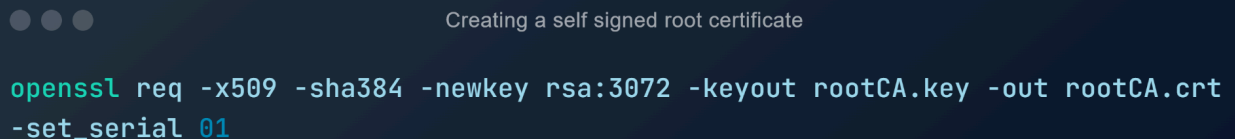
Sairam Group of Institutions

## Topics:

1. Creating Digital Certificates.
2. CryptoWrapper class Implementation.
3. Understanding the flow in custom protocol udp\_party.
4. Implementing SIGMA protocol.
5. Implementing encryption and decryption.
6. Destroying session.
7. Takeaways.

# 1. CREATING DIGITAL CERTIFICATES :

1.1 Creating a self signed root certificate (rootCA.crt) with RSA key size of 3072 with SHA384 and setting serial number as 01.

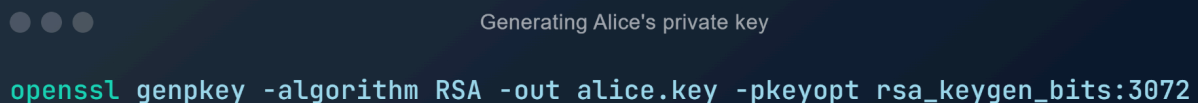


Creating a self signed root certificate

```
openssl req -x509 -sha384 -newkey rsa:3072 -keyout rootCA.key -out rootCA.crt -set_serial 01
```

1.2 Generating RSA keypair of size 3072 with SHA384 for “Alice” and signing with root CA and setting serial number as 02.

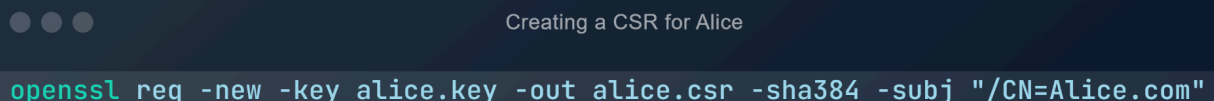
1.2.1 Generating Alice’s Private Key.



Generating Alice's private key

```
openssl genpkey -algorithm RSA -out alice.key -pkeyopt rsa_keygen_bits:3072
```

1.2.2 Creating a Certificate Signing Request ( CSR ) for Alice with a common name “Alice.com”.



Creating a CSR for Alice

```
openssl req -new -key alice.key -out alice.csr -sha384 -subj "/CN=Alice.com"
```

### 1.2.3 Signing Alice's CSR with the root CA for producing Alice's certificate.

Signing Alice's CSR

```
openssl x509 -req -in alice.csr -CA rootCA.crt -CAkey rootCA.key -CAcreateserial  
-out alice.crt -days 365 -sha384 -set_serial 02
```

## 1.3 Generating RSA keypair of size 3072 with SHA384 for "Bob" and signing with root CA and setting serial number as 03.

### 1.3.1 Generating Bob's Private Key.

Generating Bob's private key

```
openssl genpkey -algorithm RSA -out bob.key -pkeyopt rsa_keygen_bits:3072
```

### 1.3.2 Creating a Certificate Signing Request ( CSR ) for Bob with a common name "Bob.com".

Creating a CSR for Bob

```
openssl req -new -key bob.key -out bob.csr -sha384 -subj "/CN=Bob.com"
```

### 1.2.3 Signing Bob's CSR with the root CA for producing Bob's certificate.

Signing Bob's CSR

```
openssl x509 -req -in bob.csr -CA rootCA.crt -CAkey rootCA.key -CAcreateserial -out bob.crt -  
days 365 -sha384 -set_serial 03
```

## 2. 'CryptoWrapper' CLASS IMPLEMENTATION WITH OPENSSSL :

### 2.1 Implementing HMAC :

1. We've implemented an HMAC-SHA256 function.
2. The function name "hmac\_SHA256" suggests that it computes an HMAC using the SHA-256 hash algorithm.
3. Since SHA-256 produces a 256-bit hash, which is equivalent to 32 bytes, we've defined HMAC\_SIZE\_BYTES as 32.
4. Implemented with these API's  
EVP\_MD\_CTX\_new,  
EVP\_get\_digestbyname,  
EVP\_PKEY\_new\_raw\_private\_key,  
EVP\_DigestSignInit,  
EVP\_DigestSignUpdate,  
EVP\_DigestSignFinal

### 2.2 Implementing HKDF :

1. We've defined SYMMETRIC\_KEY\_SIZE\_BYTES as 32 bytes, following the Intel crypto guidelines.
2. API's used:  
EVP\_PKEY\_CTX\_new\_id,  
EVP\_PKEY\_derive\_init,  
EVP\_PKEY\_CTX\_set\_hkdf\_md,  
EVP\_PKEY\_CTX\_set1\_hkdf\_salt,  
EVP\_PKEY\_CTX\_set1\_hkdf\_key,  
EVP\_PKEY\_CTX\_add1\_hkdf\_info,  
EVP\_PKEY\_derive.

## 2.3 Implementing encryption :

1. We've defined IV\_SIZE\_BYTES as 12 bytes, corresponding to the Initialization Vector (IV) of 96 bits.
2. Additionally, we've set GMAC\_SIZE\_BYTES to 16 bytes, as the final GMAC (Galois Message Authentication Code) will be 128 bits
3. After the encryption we have appended the GMAC to the ciphertext.
4. API's used,

```
EVP_CIPHER_CTX_new,  
EVP_aes_256_gcm,  
EVP_EncryptInit_ex ,  
EVP_EncryptUpdate,  
EVP_EncryptFinal_ex,  
EVP_CIPHER_CTX_ctrl.
```

## 2.4 Implementing decryption :

1. We have extracted the attached GMAC from the ciphertext and then we have set the extracted GMAC before finalising the decryption.
2. API's used

```
EVP_CIPHER_CTX_new,  
EVP_CIPHER_CTX_ctrl,  
EVP_aes_256_gcm,  
EVP_DecryptInit_ex ,  
EVP_DecryptUpdate,  
EVP_DecryptFinal_ex.
```

## 2.5 Implementing Digital signature (RSA) :

1. API's used for reading RSA key from file :

BIO\_new\_file,  
PEM\_read\_bio\_PrivateKey\_ex,  
EVP\_PKEY\_CTX\_new.

2. API's used for signing message using private key :

EVP\_PKEY\_CTX\_get0\_pkey,  
EVP\_MD\_CTX\_create,  
EVP\_get\_digestbyname,  
EVP\_DigestInit\_ex,  
EVP\_DigestSignInit,  
EVP\_DigestSignUpdate,  
EVP\_DigestSignFinal.

3. API's used for verifying RSA:

EVP\_PKEY\_CTX\_get0\_pkey,  
EVP\_get\_digestbyname,  
EVP\_DigestInit\_ex,  
EVP\_DigestVerifyInit,  
EVP\_DigestVerifyUpdate,  
EVP\_DigestVerifyFinal.

## 2.6 Implementing Diffie Helman :

### 2.6.1 Starting diffie hellman :

1. After generating the parameters  $p$  and  $g$ , we proceed to build an `OSSL_PARAM` using these values.
2. We have created a parameter key context with the name 'DH'.
3. From the parameter key context, we obtained the parameter key and its associated parameters.
4. We have created a key generation context using the parameter key.
5. We generated the actual key from the key generation context.
6. Using the generated key, we have created a key pair context.
7. We have retrieved the key from the key pair context.
8. Finally, we have obtained the public key as a `BIGNUM` and converted it into a buffer.

### 2.6.2 Creating peer public key:

1. We generated  $p$  and  $g$  parameters.
2. The public key, previously stored as a buffer, was converted to a `BIGNUM`.
3. We constructed an `OSSL_PARAMS` object using the values of  $p$ ,  $g$ , and the public key.
4. A peer key context was created with the name "DH".
5. We obtained the peer key along with its associated parameters.

### 2.6.3 Getting dh Shared secret :

1. We created a peer public key.
2. The shared secret was derived by setting the peer public key.

### 2.6.4 API's used :

OSSL\_PARAM\_BLD\_new,  
OSSL\_PARAM\_BLD\_push\_BN,  
OSSL\_PARAM\_BLD\_to\_param,  
EVP\_PKEY\_CTX\_new\_from\_name,  
EVP\_PKEY\_fromdata\_init,  
EVP\_PKEY\_fromdata,  
EVP\_PKEY\_CTX\_new\_from\_pkey,  
EVP\_PKEY\_keygen\_init,  
EVP\_PKEY\_generate,  
EVP\_PKEY\_CTX\_get0\_pkey,  
EVP\_PKEY\_get\_bn\_param,  
BN\_bn2bin,  
EVP\_PKEY\_CTX\_new,  
EVP\_PKEY\_derive\_init,  
EVP\_PKEY\_derive\_set\_peer,  
EVP\_PKEY\_derive.

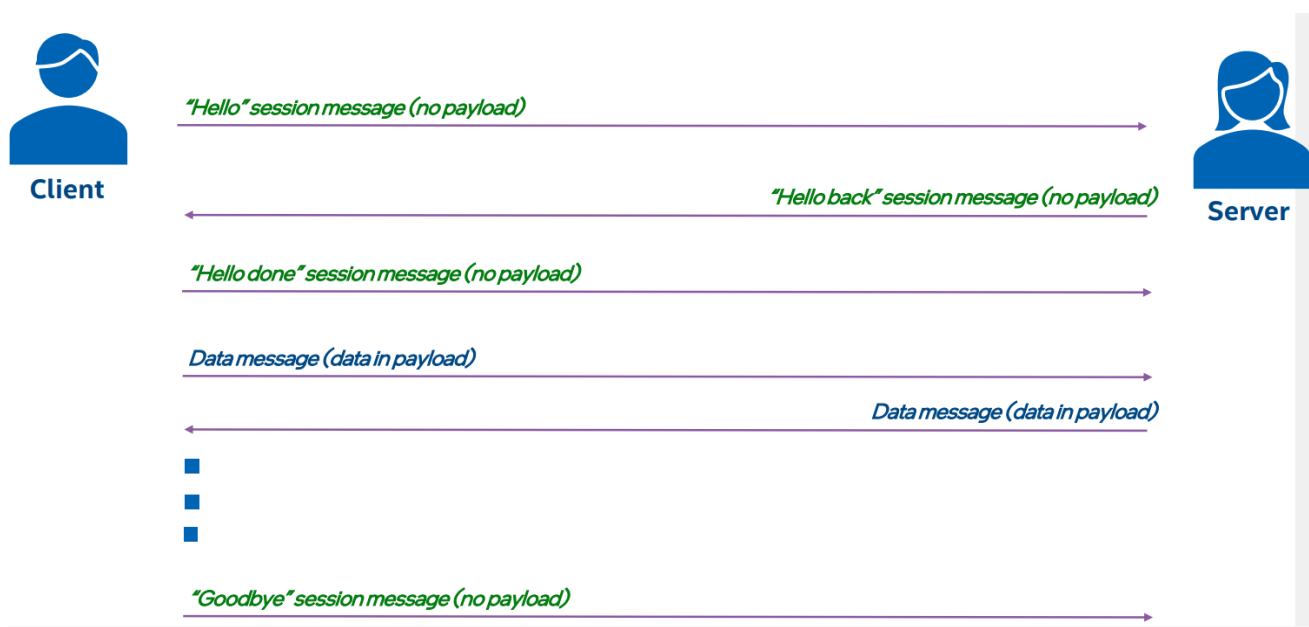
### 2.7 Certificate checking (API's Used) :

X509\_STORE\_new,  
X509\_STORE\_add\_cert,  
X509\_STORE\_CTX\_new,  
X509\_STORE\_CTX\_init,  
X509\_verify\_cert,  
X509\_check\_host.



### 3. UNDERSTANDING THE FLOW IN CUSTOM PROTOCOL `udp_party` :

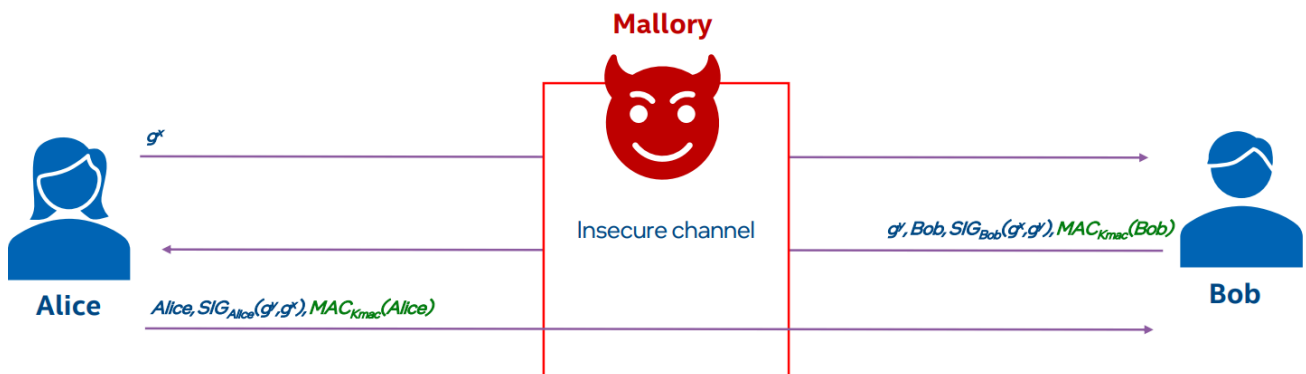
1. We used both symmetric and asymmetric key cryptography in a hybrid approach.
2. Asymmetric key cryptography was employed to prevent man-in-the-middle attacks.
3. Initially, we used asymmetric key cryptography to identify and authenticate the remote party using the SIGMA protocol.
4. For actual message exchange, we switched to symmetric key cryptography.
5. We utilised the SIGMA (SIGn and MAC) protocol, which required asymmetric key cryptography.
6. Note that the SIGMA protocol was necessary for every new session.



Here, we need to execute SIGMA protocol during the

1. "Hello" (HELLO\_SESSION\_MESSAGE),
2. "Hello back" (HELLO\_BACK\_SESSION\_MESSAGE),
3. "Hello done" (HELLO\_DONE\_SESSION\_MESSAGE)

$$K_{mac} = KDF_{mac}(g^{xy})$$



"Hello" is a SIGMA message #1 and can carry the public key of alice (i.e client).

"Hello back" is SIGMA message #2 and can carry the prepared message which has 4 information

1. public key of bob (i.e server),
2. certificate of bob,
3. signature over the concatenated public keys of alice and bob with bob's private key.
4. Mac over the bob's certificate with a mac key which is derived from the shared secret.

"Hello done" is SIGMA message #3 which also carries the 4 information,

1. public key of alice(i.e client),
2. certificate of alice,

3. signature over the concatenated public keys of bob and alice with alice's private key.
4. Mac over the alice's certificate with a mac key which is derived from the shared secret.

## **4. IMPLEMENTING "SIGMA" PROTOCOL :**

It has two main functions: preparing SIGMA messages and verifying SIGMA messages.

### **4.1 Prepare SIGMA Message :**

Preparing a SIGMA message doesn't have any message specific (HELLO\_SESSION\_MESSAGE, ... ) implementation.

It just has to prepare and pack the four parts required for SIGMA messages.

1. Read the local certificate.
2. Read the private key from the private key file.
3. Concatenate localDhBuffer and remoteDhBuffer.
4. Sign the concatenated buffer using the private key and get the signature.
5. Derive mac key from shared secret and use sessionId and root CA certificate name as salt (assuming that client and server will have the root CA certificate with the same name).
6. Prepare HMAC with a mac key and local certificate.
7. Pack localDhBuffer, local certificate buffer, signature and HMAC.

## 4.2 Verify SIGMA message :

1. Unpack the 4 parts.
2. Extract remoteDhBuffer, remote certificate buffer, signature and MAC.
3. If message type is 3 check that remoteDhBuffer received in SIGMA#1 and SIGMA#3 are the same.
4. Verify that the remote certificate is signed with a root CA certificate.
5. Read the remote public key from the remote certificate.
6. Concatenate remoteDhBuffer and localDhBuffer.
7. Verify the signature over the concatenated buffer with a remote public key.
8. If message type is 2 then get shared secret with remoteDhBuffer.
9. Derive mac key ( same as prepare sigma message ).
10. Prepare HMAC with a derived mac key.
11. Compare HMAC prepared and MAC received.

## 4.3 Changes at initialization of client session :

We have to start diffie hellman and pass localDhBuffer as a payload for SIGMA#1.

## 4.4 Changes to receive messages in server sessions :

Do all operations with a new server session.

1. Read the payload from SIGMA#1 and store it in remoteDhBuffer.
2. Start diffie hellman at server
3. Get shared secret with remoteDhBuffer.
4. Prepare SIGMA#2
5. Pass SIGMA#2 parts as a payload.

## **5. IMPLEMENTING ENCRYPTION AND DECRYPTION :**

1. Before starting the implementation of encryption and decryption we have to implement a function to derive a session key.
2. Session key is used for encryption and decryption and should be independent of mackey.
3. Session key is derived from shared secret with sessionId + content of root CA certificate used as salt.
4. For encryption and decryption, we can make use of CryptoWrapper implementation directly with a AAD (additional authentication data) as message type.

## **6. DESTROYING SESSION :**

For destroying the session we just have to use the Utils::securelyCleanMemory for releasing privateKeyPassword and cleaningDhData is simple we can use cryptowrapper's cleanDhContext.

## **7. TAKEAWAYS / LEARNINGS :**

1. Symmetric encryption.
2. HMAC.
3. Digital signature (RSA).
4. Diffie Hellman.
5. Certificate checking.
6. SIGMA protocol.
7. Simulating a client server model.
8. Protecting the traffic by using above concepts from attack like man in the middle.