Digital Certificates Generation and Cryptography Simulation Using mbedTLS/OpenSSL

Objective:

The goal is to create an interactive cryptography simulation platform that uses mbedTLS or OpenSSL libraries. This platform will enable users to establish secure server-client connections, ensuring encrypted communication that only the respective key can decrypt.

Digital Certificates Generation

1. Creating a Self-Signed Root Certificate

Generate a root certificate (rootCA.crt) with an RSA key size of 3072 and SHA384, setting the serial number to 01:

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

2. Generating RSA Key Pair for “Alice”

a. Generate Alice’s Private Key:

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

b. Create a Certificate Signing Request (CSR) for Alice:

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

c. Sign Alice's CSR with the Root CA:

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

3. Generating RSA Key Pair for “Bob”

a. Generate Bob’s Private Key:

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

b. Create a Certificate Signing Request (CSR) for Bob:

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

c. Sign Bob’s CSR with the Root CA:

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

Crypto-wrapper Implementation:

HMAC-SHA256: Function: `CryptoWrapper::hmac_SHA256`

- **APIs:**
  - `EVP_MD_CTX_new`, `EVP_PKEY_new_raw_private_key`, `EVP_DigestSignInit`, `EVP_DigestSignUpdate`, `EVP_DigestSignFinal`, `EVP_MD_CTX_free`, `EVP_PKEY_free`
- **Purpose:** Create an HMAC using the SHA-256 hashing algorithm to ensure data integrity and authenticity.

HKDF-SHA256: Function: `CryptoWrapper::deriveKey_HKDF_SHA256`

- **APIs:**
  - `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`, `EVP_PKEY_CTX_free`
- **Purpose:** Derive strong cryptographic keys from initial keying material, salt, and context information using SHA-256.

AES-GCM-256: Functions: `CryptoWrapper::encryptAES_GCM256`, `CryptoWrapper::decryptAES_GCM256`

- **APIs:**
  - `EVP_CIPHER_CTX_new`, `EVP_EncryptInit_ex`, `EVP_CIPHER_CTX_ctrl`, `EVP_EncryptUpdate`, `EVP_EncryptFinal_ex`, `EVP_CIPHER_CTX_free`, `EVP_DecryptInit_ex`, `EVP_DecryptUpdate`, `EVP_DecryptFinal_ex`
- **Purpose:** Encrypt and decrypt data using AES with 256-bit keys in GCM mode for confidentiality and integrity.

RSA-PSS: Functions: `CryptoWrapper::signMessageRsa3072Pss`, `CryptoWrapper::verifyMessageRsa3072Pss`

- **APIs:**
  - `EVP_MD_CTX_create`, `EVP_get_digestbyname`, `EVP_DigestSignInit`, `EVP_DigestSignUpdate`, `EVP_DigestSignFinal`, `EVP_DigestVerifyInit`, `EVP_DigestVerifyUpdate`, `EVP_DigestVerifyFinal`, `EVP_MD_CTX_destroy`
- **Purpose:** Sign messages and verify signatures using RSA-3072 with PSS padding for secure authentication.

Diffie-Hellman: Function: `CryptoWrapper::startDh`

- **APIs:**
  - `BN_get_rfc3526_prime_3072`, `BN_bin2bn`, `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`
- **Purpose:** Generate public/private key pairs for secure key exchange using the Diffie-Hellman algorithm.

RSA Key Management: Functions: `CryptoWrapper::readRSAKeyFromFile`, `CryptoWrapper::writePublicKeyToPemBuffer`, `CryptoWrapper::loadPublicKeyFromPemBuffer`

- **APIs:**
  - `BIO_new_file`, `PEM_read_bio_PrivateKey_ex`, `EVP_PKEY_CTX_new`, `EVP_PKEY_free`, `BIO_free`, `EVP_PKEY_CTX_get0_pkey`, `EVP_PKEY_get_bn_param`, `BN_bn2bin`
- **Purpose:** Read RSA keys from files, convert keys to PEM format, and load keys from PEM buffers for cryptographic operations.

Context Management: Function: `CryptoWrapper::cleanKeyContext`

- **APIs:**
  - `EVP_PKEY_CTX_free`
- **Purpose:** Free memory associated with cryptographic contexts to prevent memory leaks.

Usage Summary

HMAC-SHA256: Create message authentication codes to verify data integrity and authenticity.

HKDF-SHA256: Derive secure keys from a combination of input keying material, salt, and context.

AES-GCM-256: Encrypt and decrypt data, ensuring confidentiality and data integrity with authenticated encryption.

RSA-PSS: Sign messages and verify signatures to authenticate the source and integrity of messages.

Diffie-Hellman: Securely exchange cryptographic keys over a public channel.

RSA Key Management: Handle RSA keys, including reading from files, writing to buffers, and loading from buffers.

Context Management: Manage cryptographic contexts and ensure proper resource deallocation.

Protocol Flow Understanding

- **Hybrid Cryptography:** Use both symmetric and asymmetric cryptography.
- **Asymmetric Cryptography:** Prevent man-in-the-middle attacks.
- **SIGMA Protocol:** Authenticate the remote party.
- **Symmetric Cryptography:** Switch for message exchange.
- **New Sessions:** Execute the SIGMA protocol for each session.

SIGMA Protocol Steps:

1. **"Hello" (SIGMA#1):** Send Alice's public key.
2. **"Hello Back" (SIGMA#2):** Send Bob's public key, certificate, signature, and MAC.
3. **"Hello Done" (SIGMA#3):** Send Alice's public key, certificate, signature, and MAC.

SIGMA Protocol Implementation:

Prepare SIGMA Message:

1. Read the local certificate and private key.
2. Concatenate local and remote DH buffers.
3. Sign the concatenated buffer.
4. Derive MAC key from a shared secret.
5. Prepare HMAC and pack the SIGMA message.

Verify SIGMA Message:

1. Unpack the SIGMA message.
2. Verify the certificate and public key.
3. Verify the signature over the concatenated buffer.
4. Derive the MAC key and prepare HMAC.
5. Compare HMACs.

Initialization and Session Handling:

1. Start Diffie-Hellman for a client session.
2. Read and handle payloads for server sessions.

Encryption and Decryption Mechanism:

- Derive the session key from a shared secret.
- Use `CryptoWrapper` for encryption and decryption with AAD as the message type.

Session Termination:

- Use `Utils::securelyCleanMemory` to release the private key password.
- Clean DH context using `CryptoWrapper::cleanDhContext`.

Key Learnings

- HMAC-SHA256 & HKDF Key Derivation
- AES-GCM-256 Encryption/Decryption
- Diffie-Hellman Key Exchange
- Error Handling
- Memory Management
- Constants and Buffer Sizes
- Library Integration
- Digital Signature (RSA)
- Certificate Verification
- SIGMA Protocol
- Client-Server Model Simulation
- Protection against Man-in-the-Middle Attack