```cpp
#include <stdlib.h>
#include <string.h>
#include <openssl/hmac.h>
#include <openssl/evp.h>
#include <openssl/kdf.h>
#include <openssl/pem.h>
#include <openssl/x509.h>
#include <openssl/x509v3.h>
#include <openssl/bn.h>
#include <openssl/param_build.h>
#include <openssl/core_names.h>
#include <openssl/err.h>

#ifdef WIN
#pragma comment (lib, "libcrypto.lib")
#pragma comment (lib, "openssl.lib")
#endif // #ifdef WIN

static constexpr size_t PEM_BUFFER_SIZE_BYTES = 10000;
static constexpr size_t HASH_SIZE_BYTES = 32; // SHA256 hash size
static constexpr size_t IV_SIZE_BYTES = 12;
static constexpr size_t GMAC_SIZE_BYTES = 16;

bool CryptoWrapper::hmac_SHA256(const BYTE* key, size_t keySizeBytes, const
  BYTE* message, size_t messageSizeBytes, BYTE* macBuffer, size_t
  macBufferSizeBytes)
{
    if (key == NULL || message == NULL || macBuffer == NULL ||
      macBufferSizeBytes < HASH_SIZE_BYTES)
    {
        return false;
    }

    unsigned int len = 0;
    HMAC_CTX* ctx = HMAC_CTX_new();
    if (ctx == NULL)
    {
        return false;
    }

    if (HMAC_Init_ex(ctx, key, static_cast<int>(keySizeBytes), EVP_sha256(),
       NULL) != 1 ||
        HMAC_Update(ctx, message, static_cast<int>(messageSizeBytes)) != 1
          ||
        HMAC_Final(ctx, macBuffer, &len) != 1)
    {
        HMAC_CTX_free(ctx);
        return false;
    }

    HMAC_CTX_free(ctx);
    return true;
}
```

```
bool CryptoWrapper::deriveKey_HKDF_SHA256(const BYTE* salt, size_t
  saltSizeBytes,
    const BYTE* secretMaterial, size_t secretMaterialSizeBytes,
    const BYTE* context, size_t contextSizeBytes,
    BYTE* outputBuffer, size_t outputBufferSizeBytes)
{
    if (salt == NULL || secretMaterial == NULL || context == NULL ||
      outputBuffer == NULL || outputBufferSizeBytes < HASH_SIZE_BYTES)
    {
        return false;
    }

    EVP_PKEY_CTX* pctx = EVP_PKEY_CTX_new_id(EVP_PKEY_HKDF, NULL);
    if (pctx == NULL)
    {
        return false;
    }

    if (EVP_PKEY_derive_init(pctx) <= 0 ||
        EVP_PKEY_CTX_set_hkdf_md(pctx, EVP_sha256()) <= 0 ||
        EVP_PKEY_CTX_set1_hkdf_salt(pctx, salt, static_cast<int>
          (saltSizeBytes)) <= 0 ||
        EVP_PKEY_CTX_set1_hkdf_key(pctx, secretMaterial, static_cast<int>
          (secretMaterialSizeBytes)) <= 0 ||
        EVP_PKEY_CTX_add1_hkdf_info(pctx, context, static_cast<int>
          (contextSizeBytes)) <= 0 ||
        EVP_PKEY_derive(pctx, outputBuffer, &outputBufferSizeBytes) <= 0)
    {
        EVP_PKEY_CTX_free(pctx);
        return false;
    }

    EVP_PKEY_CTX_free(pctx);
    return true;
}

size_t CryptoWrapper::getCiphertextSizeAES_GCM256(size_t plaintextSizeBytes)
{
    return plaintextSizeBytes + IV_SIZE_BYTES + GMAC_SIZE_BYTES;
}

size_t CryptoWrapper::getPlaintextSizeAES_GCM256(size_t ciphertextSizeBytes)
{
    return (ciphertextSizeBytes > IV_SIZE_BYTES + GMAC_SIZE_BYTES ?
      ciphertextSizeBytes - IV_SIZE_BYTES - GMAC_SIZE_BYTES : 0);
}

bool CryptoWrapper::encryptAES_GCM256(const BYTE* key, size_t keySizeBytes,
    const BYTE* plaintext, size_t plaintextSizeBytes,
    const BYTE* aad, size_t aadSizeBytes,
    BYTE* ciphertextBuffer, size_t ciphertextBufferSizeBytes, size_t*
      pCiphertextSizeBytes)
```

```
{
    if (key == NULL || plaintext == NULL || ciphertextBuffer == NULL ||
      pCiphertextSizeBytes == NULL ||
        keySizeBytes != 32 || ciphertextBufferSizeBytes <
          getCiphertextSizeAES_GCM256(plaintextSizeBytes))
    {
        return false;
    }

    EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
    if (ctx == NULL)
    {
        return false;
    }

    BYTE iv[IV_SIZE_BYTES];
    if (!Utils::generateRandom(iv, IV_SIZE_BYTES))
    {
        EVP_CIPHER_CTX_free(ctx);
        return false;
    }

    if (EVP_EncryptInit_ex(ctx, EVP_aes_256_gcm(), NULL, key, iv) != 1 ||
        EVP_EncryptUpdate(ctx, NULL, (int*)&aadSizeBytes, aad, (int)
          aadSizeBytes) != 1 ||
        EVP_EncryptUpdate(ctx, ciphertextBuffer + IV_SIZE_BYTES, (int*)
          &plaintextSizeBytes, plaintext, (int)plaintextSizeBytes) != 1)
    {
        EVP_CIPHER_CTX_free(ctx);
        return false;
    }

    int len = 0;
    if (EVP_EncryptFinal_ex(ctx, ciphertextBuffer + IV_SIZE_BYTES +
      plaintextSizeBytes, &len) != 1)
    {
        EVP_CIPHER_CTX_free(ctx);
        return false;
    }

    if (EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_GET_TAG, GMAC_SIZE_BYTES,
      ciphertextBuffer + IV_SIZE_BYTES + plaintextSizeBytes + len) != 1)
    {
        EVP_CIPHER_CTX_free(ctx);
        return false;
    }

    memcpy(ciphertextBuffer, iv, IV_SIZE_BYTES);
    *pCiphertextSizeBytes = plaintextSizeBytes + IV_SIZE_BYTES +
      GMAC_SIZE_BYTES + len;

    EVP_CIPHER_CTX_free(ctx);
    return true;
```

```cpp
}

bool CryptoWrapper::decryptAES_GCM256(const BYTE* key, size_t keySizeBytes,
    const BYTE* ciphertext, size_t ciphertextSizeBytes,
    const BYTE* aad, size_t aadSizeBytes,
    BYTE* plaintextBuffer, size_t plaintextBufferSizeBytes, size_t*     ↩
      pPlaintextSizeBytes)
{
    if (key == NULL || ciphertext == NULL || plaintextBuffer == NULL ||     ↩
      pPlaintextSizeBytes == NULL ||
        keySizeBytes != 32 || ciphertextSizeBytes < IV_SIZE_BYTES +         ↩
          GMAC_SIZE_BYTES)
    {
        return false;
    }

    EVP_CIPHER_CTX* ctx = EVP_CIPHER_CTX_new();
    if (ctx == NULL)
    {
        return false;
    }

    BYTE iv[IV_SIZE_BYTES];
    memcpy(iv, ciphertext, IV_SIZE_BYTES);

    if (EVP_DecryptInit_ex(ctx, EVP_aes_256_gcm(), NULL, key, iv) != 1 ||
        EVP_DecryptUpdate(ctx, NULL, (int*)&aadSizeBytes, aad, (int)        ↩
          aadSizeBytes) != 1 ||
        EVP_DecryptUpdate(ctx, plaintextBuffer, (int*)pPlaintextSizeBytes,  ↩
          ciphertext + IV_SIZE_BYTES, (int)getPlaintextSizeAES_GCM256       ↩
          (ciphertextSizeBytes)) != 1 ||
        EVP_CIPHER_CTX_ctrl(ctx, EVP_CTRL_GCM_SET_TAG, GMAC_SIZE_BYTES,     ↩
          (void*)(ciphertext + IV_SIZE_BYTES + getPlaintextSizeAES_GCM256   ↩
          (ciphertextSizeBytes))) != 1 ||
        EVP_DecryptFinal_ex(ctx, plaintextBuffer + *pPlaintextSizeBytes,    ↩
          (int*)&*pPlaintextSizeBytes) != 1)
    {
        EVP_CIPHER_CTX_free(ctx);
        return false;
    }

    EVP_CIPHER_CTX_free(ctx);
    return true;
}

bool CryptoWrapper::readRSAKeyFromFile(const char* keyFilename, const char* ↩
  filePassword, KeypairContext** pKeyContext)
{
    FILE* keyFile = fopen(keyFilename, "r");
    if (keyFile == NULL)
    {
        return false;
    }
    EVP_PKEY* key = PEM_read_PrivateKey(keyFile, NULL, NULL, (void*)
      filePassword);
    fclose(keyFile);

    if (key == NULL)
    {
        return false;
    }

    *pKeyContext = EVP_PKEY_CTX_new(key, NULL);
    EVP_PKEY_free(key);

    return *pKeyContext != NULL;
}

bool CryptoWrapper::signMessageRsa3072Pss(const BYTE* message, size_t
  messageSizeBytes, KeypairContext* privateKeyContext, BYTE*
  signatureBuffer, size_t signatureBufferSizeBytes)
{
    if (message == NULL || privateKeyContext == NULL || signatureBuffer ==
      NULL || signatureBufferSizeBytes < 256)
    {
        return false;
    }

    EVP_MD_CTX* mdctx = EVP_MD_CTX_new();
    if (mdctx == NULL)
    {
        return false;
    }

    if (EVP_DigestSignInit(mdctx, NULL, EVP_sha256(), NULL,
      privateKeyContext) != 1 ||
        EVP_DigestSignUpdate(mdctx, message, messageSizeBytes) != 1 ||
        EVP_DigestSignFinal(mdctx, signatureBuffer, (size_t*)
          &signatureBufferSizeBytes) != 1)
    {
        EVP_MD_CTX_free(mdctx);
        return false;
    }

    EVP_MD_CTX_free(mdctx);
    return true;
}

bool CryptoWrapper::verifyMessageRsa3072Pss(const BYTE* message, size_t
  messageSizeBytes, KeypairContext* publicKeyContext, const BYTE* signature,
   size_t signatureSizeBytes, bool* result)
{
    if (message == NULL || publicKeyContext == NULL || signature == NULL ||
      signatureSizeBytes < 256 || result == NULL)
    {
        return false;
    }

    EVP_MD_CTX* mdctx = EVP_MD_CTX_new();
    if (mdctx == NULL)
    {
        return false;
    }

    if (EVP_DigestVerifyInit(mdctx, NULL, EVP_sha256(), NULL,
      publicKeyContext) != 1 ||
        EVP_DigestVerifyUpdate(mdctx, message, messageSizeBytes) != 1 ||
        EVP_DigestVerifyFinal(mdctx, signature, signatureSizeBytes) != 1)
    {
        *result = false;
    }
    else
    {
        *result = true;
    }

    EVP_MD_CTX_free(mdctx);
    return true;
}

bool CryptoWrapper::writePublicKeyToPemBuffer(KeypairContext* keyContext,
  BYTE* publicKeyPemBuffer, size_t publicKeyBufferSizeBytes)
{
    if (keyContext == NULL || publicKeyPemBuffer == NULL ||
      publicKeyBufferSizeBytes < PEM_BUFFER_SIZE_BYTES)
    {
        return false;
    }

    BIO* bio = BIO_new(BIO_s_mem());
    if (bio == NULL)
    {
        return false;
    }

    if (PEM_write_bio_PUBKEY(bio, keyContext) != 1)
    {
        BIO_free(bio);
        return false;
    }

    int len = BIO_read(bio, publicKeyPemBuffer, static_cast<int>
      (publicKeyBufferSizeBytes));
    if (len <= 0)
    {
        BIO_free(bio);
        return false;
    }
    BIO_free(bio);
    return true;
}

bool CryptoWrapper::loadPublicKeyFromPemBuffer(KeypairContext* context,
  const BYTE* publicKeyPemBuffer, size_t publicKeyBufferSizeBytes)
{
    if (context == NULL || publicKeyPemBuffer == NULL ||
      publicKeyBufferSizeBytes == 0)
    {
        return false;
    }

    BIO* bio = BIO_new_mem_buf(publicKeyPemBuffer, static_cast<int>
      (publicKeyBufferSizeBytes));
    if (bio == NULL)
    {
        return false;
    }

    EVP_PKEY* key = PEM_read_bio_PUBKEY(bio, NULL, NULL, NULL);
    if (key == NULL)
    {
        BIO_free(bio);
        return false;
    }

    EVP_PKEY_CTX* pctx = EVP_PKEY_CTX_new(key, NULL);
    EVP_PKEY_free(key);
    if (pctx == NULL)
    {
        BIO_free(bio);
        return false;
    }

    *context = pctx;
    BIO_free(bio);
    return true;
}

bool CryptoWrapper::startDh(DhContext** pDhContext, BYTE* publicKeyBuffer,
  size_t publicKeyBufferSizeBytes)
{
    if (pDhContext == NULL || publicKeyBuffer == NULL ||
      publicKeyBufferSizeBytes < DH_KEY_SIZE_BYTES)
    {
        return false;
    }

    *pDhContext = EVP_PKEY_CTX_new_id(EVP_PKEY_DH, NULL);
    if (*pDhContext == NULL)
    {
        return false;
    }

    EVP_PKEY* pkey = EVP_PKEY_new();
    if (pkey == NULL)
    {
        EVP_PKEY_CTX_free(*pDhContext);
        return false;
    }

    DH* dh = DH_new();
    if (dh == NULL)
    {
        EVP_PKEY_free(pkey);
        EVP_PKEY_CTX_free(*pDhContext);
        return false;
    }

    if (DH_set0_pqg(dh, BN_get_rfc3526_prime_3072(NULL), NULL, BN_bin2bn
      ((const unsigned char*)&generator, 1, NULL)) != 1 ||
        EVP_PKEY_assign(pkey, EVP_PKEY_DH, dh) != 1 ||
        EVP_PKEY_CTX_set_dh_pkey_ctx(*pDhContext, pkey) != 1 ||
        EVP_PKEY_get_raw_public_key(pkey, publicKeyBuffer, (size_t*)
          &publicKeyBufferSizeBytes) != 1)
    {
        EVP_PKEY_free(pkey);
        EVP_PKEY_CTX_free(*pDhContext);
        return false;
    }

    EVP_PKEY_free(pkey);
    return true;
}

bool CreatePeerPublicKey(const BYTE* peerPublicKey, size_t
  peerPublicKeySizeBytes, EVP_PKEY** genPeerPublicKey)
{
    if (peerPublicKey == NULL || genPeerPublicKey == NULL ||
      peerPublicKeySizeBytes == 0)
    {
        return false;
    }

    DH* dh = DH_new();
    if (dh == NULL)
    {
        return false;
    }

    BIGNUM* peerPublicBN = BN_bin2bn(peerPublicKey, (int)
      peerPublicKeySizeBytes, NULL);
    if (peerPublicBN == NULL || DH_set0_key(dh, peerPublicBN, NULL) != 1)
    {
        DH_free(dh);
        return false;
    }

    *genPeerPublicKey = EVP_PKEY_new();
    if (*genPeerPublicKey == NULL || EVP_PKEY_assign(*genPeerPublicKey,     ⤾
      EVP_PKEY_DH, dh) != 1)
    {
        DH_free(dh);
        EVP_PKEY_free(*genPeerPublicKey);
        return false;
    }

    return true;
}

bool CryptoWrapper::getDhSharedSecret(DhContext* dhContext, const BYTE*     ⤾
  peerPublicKey, size_t peerPublicKeySizeBytes, BYTE* sharedSecretBuffer,   ⤾
  size_t sharedSecretBufferSizeBytes)
{
    if (dhContext == NULL || peerPublicKey == NULL || sharedSecretBuffer == ⤾
      NULL || peerPublicKeySizeBytes != DH_KEY_SIZE_BYTES ||                ⤾
      sharedSecretBufferSizeBytes < DH_KEY_SIZE_BYTES)
    {
        return false;
    }

    EVP_PKEY* genPeerPublicKey = NULL;
    EVP_PKEY_CTX* derivationCtx = NULL;

    if (!CreatePeerPublicKey(peerPublicKey, peerPublicKeySizeBytes,         ⤾
      &genPeerPublicKey))
    {
        return false;
    }

    if (EVP_PKEY_derive_init(dhContext) != 1 ||
        EVP_PKEY_derive_set_peer(dhContext, genPeerPublicKey) != 1 ||
        EVP_PKEY_derive(dhContext, sharedSecretBuffer,                      ⤾
          &sharedSecretBufferSizeBytes) != 1)
    {
        EVP_PKEY_free(genPeerPublicKey);
        return false;
    }

    EVP_PKEY_free(genPeerPublicKey);
    return true;
}

void CryptoWrapper::cleanDhContext(DhContext** pDhContext)
{
    if (*pDhContext != NULL)
    {
        EVP_PKEY_CTX_free(*pDhContext);
        *pDhContext = NULL;
    }
}

X509* loadCertificate(const BYTE* certBuffer, size_t certSizeBytes)
{
    int ret = 0;
    BIO* bio = NULL;
    X509* cert = NULL;

    bio = BIO_new(BIO_s_mem());
    if (bio == NULL)
    {
        printf("BIO_new() fail \n");
        return NULL;
    }

    ret = BIO_write(bio, (const void*)certBuffer, (int)certSizeBytes);
    if (ret <= 0)
    {
        printf("BIO_write() fail \n");
        BIO_free(bio);
        return NULL;
    }

    cert = PEM_read_bio_X509(bio, NULL, NULL, NULL);
    if (cert == NULL)
    {
        printf("PEM_read_bio_X509() fail \n");
    }

    BIO_free(bio);
    return cert;
}

bool CryptoWrapper::checkCertificate(const BYTE* cACcertBuffer, size_t     ↩
  cACertSizeBytes, const BYTE* certBuffer, size_t certSizeBytes, const char*↩
   expectedCN)
{
    X509* userCert = NULL;
    X509* caCert = NULL;
    X509_STORE* store = X509_STORE_new();
    X509_STORE_CTX* ctx = X509_STORE_CTX_new();
    bool ret = false;

    caCert = loadCertificate(cACcertBuffer, cACertSizeBytes);
    if (caCert == NULL)
    {
        printf("loadCertificate() fail \n");
        goto err;
    }
    userCert = loadCertificate(certBuffer, certSizeBytes);
    if (userCert == NULL)
    {
        printf("loadCertificate() fail \n");
        goto err;
    }

    if (X509_STORE_add_cert(store, caCert) != 1)
    {
        printf("X509_STORE_add_cert() fail \n");
        goto err;
    }

    if (X509_STORE_CTX_init(ctx, store, userCert, NULL) != 1 ||
        X509_verify_cert(ctx) != 1)
    {
        printf("X509_verify_cert() fail \n");
        goto err;
    }

    X509_NAME* subjectName = X509_get_subject_name(userCert);
    if (subjectName == NULL)
    {
        printf("X509_get_subject_name() fail \n");
        goto err;
    }

    char cn[256];
    if (X509_NAME_get_text_by_NID(subjectName, NID_commonName, cn, sizeof   ⤶
      (cn)) < 0 ||
        strcmp(cn, expectedCN) != 0)
    {
        printf("CN mismatch\n");
        goto err;
    }

    ret = true;

err:
    X509_STORE_free(store);
    X509_STORE_CTX_free(ctx);
    X509_free(caCert);
    X509_free(userCert);
    return ret;
}

bool CryptoWrapper::getPublicKeyFromCertificate(const BYTE* certBuffer,      ⤶
  size_t certSizeBytes, KeypairContext** pPublicKeyContext)
{
    if (certBuffer == NULL || pPublicKeyContext == NULL || certSizeBytes == ⤶
      0)
    {
        return false;
    }

    X509* cert = loadCertificate(certBuffer, certSizeBytes);
    if (cert == NULL)
    {
        return false;
    }

    EVP_PKEY* key = X509_get_pubkey(cert);
    if (key == NULL)
    {
        X509_free(cert);
        return false;
    }

    EVP_PKEY_CTX* pctx = EVP_PKEY_CTX_new(key, NULL);
    EVP_PKEY_free(key);
    if (pctx == NULL)
    {
        X509_free(cert);
        return false;
    }

    *pPublicKeyContext = pctx;
    X509_free(cert);
    return true;
}
```