Cybersecurity MSc: Applied Cryptography 2023-24

Practical Exercises: Asymmetric Cryptography – Elliptic Curves (EC)

September 14, 2023 Due date: no date

Changelog

- v1.0 - Initial version.

Introduction

In order to elaborate this laboratory guide it is required to install the Java Development Environment (JDK), Python 3 or the C compiler and development environment.

The examples provided will use both Java, Python and C. For Python you need to install the `cryptography` module. For C we will use the EVP (Digital Envelope) functions from the Crypto library that is part of OpenSSL, and for that you need to use install the packages `libssl1.0-dev` and `libssl1.1`.

1 Elements of interest for each language

1.1 Java

Since version 9 Java uses a specific provider for implementing EC operations.

- java.security.**KeyPairGenerator** An instance of the class `KeyPairGenerator` is a generator of asymmetric keys. Some important methods are:
  - `getInstance`: creates an instance of a generator for producing keys for a given cipher algorithm;
  - `init`: initializes the key generator with some elements for helping the key production. For EC, you should use an instance of the class `EcGenParameterSpec`.
  - `generateKeyPair` effectively creates an asymmetric key.
- java.security.spec.**ECGenParameterSpec** An instance of this class defines an EC curve for creating a key pair for it. It can be instanciated with the name of the curve.
- java.security.spec.**ECParameterSpec** An instance of this class contains all the parameters related with the curve that an EC key pair belongs to (e.g. the curve order).
- java.security.spec.**EllipticCurve** An instance of the `EllipticCurve` allows to implement the operations necessary to perform encryptions and decryption with EC. Different curves may be used by using the appropriate constructor parameters. The number of curves is endless.
- java.security.spec.**ECField** An interface for managing an EC finite field, with two sub-interfaces: java.security.spec.**ECFieldFp**, for prime finite fields, and java.security.spec.**ECFieldF2m**, for characteristic 2 finite fields.
- java.security.interfaces.**ECKey** An interface for managing an EC key pair, with two sub-interfaces: java.security.interfaces.**ECPrivateKey** and java.security.interfaces.**ECPublicKey**.

1.2 Python

- The cryptography module is both a frontend, high-level interface for dealing with cryptography and a default backend implementation of the fundamental cryptographic primitives.
- The Python interpreter distinguishes bytes from text characters, and thus byte arrays from strings. A string is more than a byte array, it has also an encoding (e.g. UTF-8, Unicode, etc.). You can get the bytes of characters (strings) using the `bytes` function, and you can convert bytes to characters (strings) by using the method `decode` on a byte array (with a target encoding).

  By default, in cryptography we always use bits and bytes, nothing else. So, do not attempt to solve interpreter errors by changing everything to text (characters and strings)!

1.3 C

- A structure `EVP_PKEY` describes an asymmetric key pair that can be an EC key pair. Such key pair contains an `EC_POINT` public key (a point) and a `BIGNUM` private key.
- Function `EVP_PKEY_keygen_new_id` is used to create a context to generate an EC key pair. That context is initialized with the function `EVP_PKEY_keygen_init`, and the EC curve can be defined with the function `EVP_PKEY_CTX_set_group_name`. The key pair is efectively generated with the function `EVP_PKEY_keygen`.
- Functions `EC_POINT_`$\cdots$ are functions that perform operations on a given EC.

2 Creation and storage of a random asymmetric key pair

Develop a program to generate a random EC key pair and save it to a file.

2.1 Generation parameters

An EC key pair is formed by a scalar private key and a EC point public key. Thus, the generation of an EC key pair boils down to the generation of a large integer, lower than the order of the EC generator.

ECs can be arbitrarily defined, but that is not a usual approach, because you can pick up an insecure curve. Instead, you should stick to one of the existing, well-studied curves, which have a common name.

The number of bits of the private key depends on the chosen curve, namely on the order of its generator.

The following code samples exemplify how we can generate random, EC key pairs, using several curves, and inspect their values using Java:

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.interfaces.ECPrivateKey;
import java.security.interfaces.ECPublicKey;
import java.security.spec.AlgorithmParameterSpec;
import java.security.spec.ECParameterSpec;
import java.security.spec.ECGenParameterSpec;

import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.security.InvalidAlgorithmParameterException;

String curves[] = {
    // NIST P curves
    "secp192r1", // NIST P-192
    "secp224r1", // NIST P-224
    "secp256r1", // NIST P-256
    "secp384r1", // NIST P-384
    "secp521r1", // NIST P-521
    // NIST K curves
    "sect163k1", // NIST K-163
    "sect233k1", // NIST K-233
    "sect283k1", // NIST K-238
    "sect409k1", // NIST K-409
    "sect571k1", // NIST K-571
    // NIST B curves
    "sect163r2", // NIST B-163
    "sect233r1", // NIST B-233
    "sect283r1", // NIST B-283
    "sect409r1", // NIST B-409
    "sect571r1", // NIST B-571
    // Brainpool curves, RFC5639 (legacy use, only)
    "brainpoolP256r1",
    "brainpoolP320r1",
    "brainpoolP384r1",
    "brainpoolP512r1"
    };

void dump( String curveName, KeyPair keyPair )
{
    ECPrivateKey privKey = (ECPrivateKey) keyPair.getPrivate();
    ECPublicKey pubKey = (ECPublicKey) keyPair.getPublic();

    out.println( "--------------------------" );
    out.printf( "Curve = %s, private key size = %d bits\n",
                curveName,
                privKey.getParams().getOrder().bitLength() );
    out.printf( "  private key = %s\n  public key =\n    %s,\n    %s\n",
                privKey.getS(),
                pubKey.getW().getAffineX(),
                pubKey.getW().getAffineY() );
}

KeyPairGenerator generator = KeyPairGenerator.getInstance( "EC", "SunEC" );
AlgorithmParameterSpec spec;
KeyPair keyPair;

for (String curve: curves) {
    spec = new ECGenParameterSpec( curve );
    generator.initialize( spec );
    keyPair = generator.generateKeyPair();
    dump( curve, keyPair );
}
```

The following code does the same in Python:

```
from cryptography.hazmat.primitives.asymmetric import ec

def dump( key_pair ):
    print( 'Curve = %s, private key size = %d bits' %
             ( key_pair.curve.name,
               key_pair.key_size ) )
    print( '  private key = %x\n  public key =\n    %x,\n    %x' %
             ( key_pair.private_numbers().private_value,
               key_pair.public_key().public_numbers().x,
               key_pair.public_key().public_numbers().y) )

curves = [
    # NIST P curves
    ec.SECP192R1(), # NIST P-192
    ec.SECP224R1(), # NIST P-224
    ec.SECP256R1(), # NIST P-256
    ec.SECP384R1(), # NIST P-384
    ec.SECP521R1(), # NIST P-521
    # NIST K curves
    ec.SECT163K1(), # NIST K-163
    ec.SECT233K1(), # NIST K-233
    ec.SECT283K1(), # NIST K-238
    ec.SECT409K1(), # NIST K-409
    ec.SECT571K1(), # NIST K-571
    # NIST B curves
    ec.SECT163R2(), # NIST B-163
    ec.SECT233R1(), # NIST B-233
    ec.SECT283R1(), # NIST B-283
    ec.SECT409R1(), # NIST B-409
    ec.SECT571R1(), # NIST B-571
    # Brainpool curves, RFC5639 (legacy use, only)
    ec.BrainpoolP256R1(),
    ec.BrainpoolP384R1(),
    ec.BrainpoolP512R1()
    ]

for curve in curves:
    key_pair = ec.generate_private_key( curve )
    dump( key_pair )
```

The following code does the same in C:

```c
#include <stdio.h>

#include <openssl/evp.h>
#include <openssl/core_names.h>
#include <openssl/obj_mac.h>

char * curves[] = {
    // NIST P curves
    SN_secp224r1,
    SN_secp384r1,
    SN_secp521r1,
    // NIST K curves
    SN_sect163k1,
    SN_sect233k1,
    SN_sect283k1,
    SN_sect409k1,
    SN_sect571k1,
    // NIST B curves
    SN_sect163r2,
    SN_sect233r1,
    SN_sect283r1,
    SN_sect409r1,
    SN_sect571r1,
    // Other K curves
    SN_secp160k1,
    SN_secp192k1,
    SN_secp224k1,
    SN_secp256k1,
    SN_sect239k1,
    // Other B curves
    SN_sect193r1,
    SN_sect193r2,
    SN_sect113r1,
    SN_sect113r2,
    SN_sect131r1,
    SN_sect131r2,
    SN_sect163r1,
    // Brainpool curves, RFC5639 (legacy use, only)
    // Random
    SN_brainpoolP160r1,
    SN_brainpoolP192r1,
    SN_brainpoolP224r1,
    SN_brainpoolP256r1,
    SN_brainpoolP320r1,
    SN_brainpoolP384r1,
    SN_brainpoolP512r1,
    // Twisted
    SN_brainpoolP160t1,
    SN_brainpoolP192t1,
    SN_brainpoolP224t1,
    SN_brainpoolP256t1,
    SN_brainpoolP320t1,
    SN_brainpoolP384t1,
    SN_brainpoolP512t1,
    0
};

void
dump( EVP_PKEY * key_pair )
{
   char * curve_name;
   size_t len;
   BIGNUM * order, * private_key, * public_key[2];
   char * str1, * str2, * str3;

   EVP_PKEY_get_utf8_string_param( key_pair, OSSL_PKEY_PARAM_GROUP_NAME, 0, 0, &len );
   curve_name = alloca( len + 1 );
   if (EVP_PKEY_get_utf8_string_param( key_pair, OSSL_PKEY_PARAM_GROUP_NAME, curve_name, len
       + 1, 0 ) <= 0) {
       return;
   }

   order = 0;
   if (EVP_PKEY_get_bn_param( key_pair, OSSL_PKEY_PARAM_EC_ORDER, &order ) <= 0) {
```

```
        return;
    }

    printf( "Curve %s, private key size = %d bits\n", curve_name, BN_num_bits( order ) );
    BN_free( order );

    private_key = 0;
    if (EVP_PKEY_get_bn_param( key_pair, OSSL_PKEY_PARAM_PRIV_KEY, &private_key ) <= 0) {
        return;
    }

    public_key[0] = 0; // X
    public_key[1] = 0; // Y
    if (EVP_PKEY_get_bn_param( key_pair, OSSL_PKEY_PARAM_EC_PUB_X, &(public_key[0]) ) <= 0 ||
        EVP_PKEY_get_bn_param( key_pair, OSSL_PKEY_PARAM_EC_PUB_Y, &(public_key[1]) ) <= 0) {
        return;
    }

    str1 = BN_bn2hex( private_key );
    str2 = BN_bn2hex( public_key[0] );
    str3 = BN_bn2hex( public_key[1] );
    printf( "  private key = %s\n  public key =\n    %s,\n    %s\n", str1, str2, str3 );

    BN_free( private_key );
    BN_free( public_key[0] );
    BN_free( public_key[1] );
    OPENSSL_free( str1 );
    OPENSSL_free( str2 );
    OPENSSL_free( str3 );
}

EVP_PKEY_CTX * ctx;
EVP_PKEY * key_pair;
ENGINE * engine = 0;

for (int i = 0; curves[i] != 0; i++) {
    // Create an EC key pair context with the default engine

    ctx = EVP_PKEY_CTX_new_id( EVP_PKEY_EC, engine );
    if (ctx == 0) {
        // Error
    }

    if (EVP_PKEY_keygen_init( ctx ) <= 0) {
        // Error
    }

    // Define required key pair parameters

    if (EVP_PKEY_CTX_set_group_name( ctx, curves[i] ) <= 0) {
        // Error
    }

    // Generate key pair.
    // The space for it is allocated by the generator

    key_pair = 0;
    if (EVP_PKEY_keygen( ctx, &key_pair ) <= 0) {
        // Error
    }

    dump( key_pair );

    // Release the space occupied by the key pair and its generation context

    EVP_PKEY_free( key_pair );
    EVP_PKEY_CTX_free( ctx );
}
```

2.2 Key pair storage

EC key pairs are stored as follows:

- A file that contains both keys of the key pair.

This is how the key pair owner stores it. The private key can be protected by a password, so that only the key owner could have access to the private key from the file (otherwise, anyone with access to the file could do it).

- A file that contains only the public key.

  This file is used for transmitting the public key to others than the key pair owner.

There are many formats that can be used to store key pairs and public keys:

- Distinguished Encoding Rule (DER), This is a binary format that uses an ASN.1 encoding.
- Public Key Cryptography Standard #12 (PKCS #12), This is also a binary format that uses an ASN.1 encoding. This format does not store directly public keys, but rather certificates of public keys.
- Privacy Enhanced Mail (PEM). This is a textual format used to encode DER or PKCS #12 representations.

PEM is very convenient for transmiting keys through communication channels that at their lowest level only handle textual contents (e-mail, HTTP, etc.) and facilitates the management of keys and key pairs on files, since they can be concatenated.

PEM files start and end with a characeristic header and trailer line, respectively:

- `---BEGIN PRIVATE KEY---` and `---END PRIVATE KEY---`. These lines are used to bound a key pair (with a cleartext private key). They surround a textual encoding of an PKCS #8 unencrypted private key information.
- `---BEGIN ENCRYPTED PRIVATE KEY---` and `---END ENCRYPTED PRIVATE KEY---`. These lines are used to bound a key pair (with an encrypted private key). They surround a textual encoding of a PKCS #8 encrypted private key information.
- `---BEGIN PUBLIC KEY---` and `---END PUBLIC KEY---`. These lines are used to bound a public key. They surround a textual encoding of an ANS.1-encoded (using BER or DER) public key information.

When a file stores a password-encrypted private key, it must as well record how the private key was encrypted (which algorithm was used). This includes the algorithm to transform the password to a symmetric key (for a given algorithm) and the algorithm to encrypt the private key with that key. This is implemented by means of a different format, PKCS #8.

The following code samples exemplify how we can print PEM encodings of an EC key pair using Java:

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.interfaces.ECPrivateKey;
import java.security.interfaces.ECPublicKey;
import java.security.spec.AlgorithmParameterSpec;
import java.security.spec.ECParameterSpec;
import java.security.spec.ECGenParameterSpec;

import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.security.InvalidAlgorithmParameterException;

import java.util.Base64;

KeyPairGenerator generator = KeyPairGenerator.getInstance( "EC", "SunEC" );
AlgorithmParameterSpec spec;
KeyPair keyPair;
ECPrivateKey prvKey;
ECPublicKey pubKey;
byte[] asn1Object;
byte[] lineSeparator = { 0x0a };
Base64.Encoder pemEncoder;

spec = new ECGenParameterSpec( "secp521r1" );
generator.initialize( spec );
keyPair = generator.generateKeyPair();

prvKey = (ECPrivateKey) keyPair.getPrivate();
pubKey = (ECPublicKey) keyPair.getPublic();

// Let's dump the generated keys in different PEM formats (with different contents)

// Create a PEM encoder, which is a Base64 MIME encoder with 64-character lines

pemEncoder = Base64.getMimeEncoder( 64, lineSeparator );

// Write the key pair to stdout
// No private key encryption

asn1Object = prvKey.getEncoded();
out.printf( "-----BEGIN PRIVATE KEY-----\n" +
            pemEncoder.encodeToString( asn1Object ) +
            "\n-----END PRIVATE KEY-----\n" );

//  Write the publick key of a key pair to stdout

asn1Object = pubKey.getEncoded();
out.printf( "-----BEGIN PUBLIC KEY-----\n" +
            pemEncoder.encodeToString( asn1Object ) +
            "\n-----END PUBLIC KEY-----\n" );
```

The following code does the same in Python:

```
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import serialization

private_key = ec.generate_private_key( ec.SECP521R1() )
public_key = private_key.public_key()

# Let's dump the generated keys in different PEM formats (with different contents)

# Write the key pair to stdout
# No private key encryption

pem = private_key.private_bytes( encoding=serialization.Encoding.PEM,
                                 format=serialization.PrivateFormat.TraditionalOpenSSL,
                                 encryption_algorithm=serialization.NoEncryption() )
print( pem.decode( 'ascii' ) )

# Write the key pair to stdout
# The private key is encrypted with AES+CBC and a key derived from the password "senha"

pem = private_key.private_bytes( encoding=serialization.Encoding.PEM,
                                    format=serialization.PrivateFormat.PKCS8,
                                    encryption_algorithm=serialization.BestAvailableEncryption(
                                        b'senha' ) )
print( pem.decode( 'ascii' ) )

# Write the publick key of a key pair to stdout

pem = public_key.public_bytes( encoding=serialization.Encoding.PEM,
                                 format=serialization.PublicFormat.SubjectPublicKeyInfo )
print( pem.decode( 'ascii' ) )
```

The following code does the same in C:

```c
#include <openssl/evp.h>
#include <openssl/core_names.h>
#include <openssl/obj_mac.h>
#include <openssl/pem.h>

EVP_PKEY_CTX * ctx;
EVP_PKEY * key_pair;
ENGINE * engine = 0;

// Create an EC key pair context with the default engine

ctx = EVP_PKEY_CTX_new_id( EVP_PKEY_EC, engine );
if (ctx == 0) {
    // Error
}

if (EVP_PKEY_keygen_init( ctx ) <= 0) {
    // Error
}

// Define required key pair parameters

if (EVP_PKEY_CTX_set_group_name( ctx, SN_secp521r1 ) <= 0) {
    // Error
}

// Generate key pair.
// The space for it is allocated by the generator

key_pair = 0;
if (EVP_PKEY_keygen( ctx, &key_pair ) <= 0) {
    // Error
}

// Let's dump the generated keys in different PEM formats (with different contents)

// Write the key pair to stdout
// No private key encryption

PEM_write_PrivateKey( stdout, key_pair, 0, 0, 0, 0, 0 );

// Write the key pair to stdout
// The private key is encrypted with AES+CBC and a key derived from the password "senha"

PEM_write_PrivateKey( stdout, key_pair, EVP_aes_128_cbc(), "senha", 5, 0, 0 );

// Write the publick key of a key pair to stdout

PEM_write_PUBKEY( stdout, key_pair );

// Release the space occupied by the key pair and its generation context

EVP_PKEY_free( key_pair );
EVP_PKEY_CTX_free( ctx );
```

3 Bulk data encryption and decryption using an EC key pair

EC public key encryption, and subsequent private key decryption, follows a strategy that has nothing in common with RSA. Instead, it uses a strategy similar to the Diffie-Hellman key exchange protocol. The key agreement process described as Elliptic Curve Diffie-Hellman (ECDH) was standardized by two NIST publications, first in 800-56A, and later in 800-56Ar2.

When Alice wants to encrypt some content with the public key of Bob, Alice generates a fresh key pair, produces a symmetric key with the private component of that key pair and the public component of Bob's key pair and uses it, with some symmetric approach, to encrypt the data. Next, Alice send the cryptogram, together it the public component of the fresh key pair, to Bob. Bob uses the received public component, and his private one, to compute the same symmetric key, and uses it to decrypt the cryptogram. This is somewhat similar to hybrid, or mixed, cryptography.

In EC, the combination of a private component (a scaler) and a public one (a point) is their multiplication, which yields another point. Then, we can derive a symmetric key from that point using a digest function, for instance.

Note that, similarly to the original DH key exchange protocol, where both ends need to use the same modular group, in ECDH both ends need to use the same elliptic curve.

The following code samples exemplify how we can get one common shared key using ECDH using Java:

```
import static java.lang.System.out;

import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.interfaces.ECPrivateKey;
import java.security.interfaces.ECPublicKey;
import java.security.spec.AlgorithmParameterSpec;
import java.security.spec.ECParameterSpec;
import java.security.spec.ECGenParameterSpec;

import javax.crypto.KeyAgreement;

import javax.crypto.spec.PBEKeySpec;
import javax.crypto.spec.SecretKeySpec;
import javax.crypto.SecretKeyFactory;
import javax.crypto.SecretKey;

import java.util.Arrays;

import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.security.InvalidAlgorithmParameterException;
import java.security.InvalidKeyException;
import java.security.spec.InvalidKeySpecException;

KeyPairGenerator generator = KeyPairGenerator.getInstance( "EC", "SunEC" );
KeyAgreement agreement = KeyAgreement.getInstance( "ECDH", "SunEC" );
SecretKeyFactory skf = SecretKeyFactory.getInstance( "PBKDF2WithHmacSHA1" );
KeyPair aliceKeyPair;
KeyPair bobKeyPair;
PBEKeySpec pbeKeySpec;
byte[] sharedSecret;
byte[] aliceDerivedKey;
byte[] bobDerivedKey;
byte[] salt = new byte[1];

salt[0] = 0; // We need to provide one salt ...

// Generate Alice and Bob key pairs

generator.initialize( new ECGenParameterSpec( "secp521r1" ) );
aliceKeyPair = generator.generateKeyPair();
bobKeyPair = generator.generateKeyPair();

// Perform Alice DH operation and key derivation

agreement.init( aliceKeyPair.getPrivate() );
agreement.doPhase( bobKeyPair.getPublic(), true );
sharedSecret = agreement.generateSecret();

pbeKeySpec = new PBEKeySpec( (new String( sharedSecret )).toCharArray(), salt, 1000, 256 );

aliceDerivedKey = skf.generateSecret( pbeKeySpec ).getEncoded();

// Perform Bob DH operation and key derivation

agreement.init( bobKeyPair.getPrivate() );
agreement.doPhase( aliceKeyPair.getPublic(), true );
sharedSecret = agreement.generateSecret();

pbeKeySpec = new PBEKeySpec( (new String( sharedSecret )).toCharArray(), salt, 1000, 256 );

bobDerivedKey = skf.generateSecret( pbeKeySpec ).getEncoded();

out.println( Arrays.equals( aliceDerivedKey, bobDerivedKey ) );
```

The following code does the same in Python:

```
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF

# Generate Alice and Bob key pairs

alice_key_pair = ec.generate_private_key( ec.SECP521R1() )
bob_key_pair = ec.generate_private_key( ec.SECP521R1() )

# Perform Alice DH operation and key derivation

alice_shared_secret = alice_key_pair.exchange( ec.ECDH(), bob_key_pair.public_key() )

alice_derived_key = HKDF( algorithm=hashes.SHA256(),
                          length=32,
                          salt=None,
                          info=b'handshake data' ).derive( alice_shared_secret )

# Perform Bob DH operation and key derivation

bob_shared_secret = bob_key_pair.exchange( ec.ECDH(), alice_key_pair.public_key() )

bob_derived_key = HKDF( algorithm=hashes.SHA256(),
                          length=32,
                          salt=None,
                          info=b'handshake data' ).derive( bob_shared_secret )

print( alice_derived_key == bob_derived_key )
```

The following code does the same in C:

```c
#include <stdio.h>
#include <string.h>

#include <openssl/evp.h>
#include <openssl/core_names.h>

EVP_PKEY_CTX * ctx;
EVP_PKEY * alice_key_pair, * bob_key_pair;
ENGINE * engine = 0;
unsigned int pad = 1;
OSSL_PARAM params[6];
uint8_t alice_derived_key[32], bob_derived_key[32];
size_t alice_key_len, bob_key_len, out_len;
char * info = "handshake data";

out_len = alice_key_len = bob_key_len = sizeof(alice_derived_key);

// Create an EC key pair context with the default engine

ctx = EVP_PKEY_CTX_new_id( EVP_PKEY_EC, engine );
if (ctx == 0) {
    // Error
}

if (EVP_PKEY_keygen_init( ctx ) <= 0) {
    // Error
}

// Define required key pair parameters

if (EVP_PKEY_CTX_set_group_name( ctx, SN_secp521r1 ) <= 0) {
    // Error
}

// Generate Alice and Bob key pairs.
// The space for it is allocated by the generator

alice_key_pair = 0;
if (EVP_PKEY_keygen( ctx, &alice_key_pair ) <= 0) {
    // Error
}
bob_key_pair = 0;
if (EVP_PKEY_keygen( ctx, &bob_key_pair ) <= 0) {
    // Error
}

EVP_PKEY_CTX_free( ctx );

params[0] = OSSL_PARAM_construct_uint( OSSL_EXCHANGE_PARAM_PAD, &pad );
params[1] = OSSL_PARAM_construct_utf8_string( OSSL_EXCHANGE_PARAM_KDF_TYPE, "HKDF", 0 );
params[2] = OSSL_PARAM_construct_utf8_string(OSSL_EXCHANGE_PARAM_KDF_DIGEST, "SHA256", 0 );
params[3] = OSSL_PARAM_construct_size_t(OSSL_EXCHANGE_PARAM_KDF_OUTLEN, &out_len );
params[4] = OSSL_PARAM_construct_octet_string(OSSL_EXCHANGE_PARAM_KDF_UKM, info, strlen( info
    ) );
params[5] = OSSL_PARAM_construct_end();

// Perform Alice DH operation and key derivation

ctx = EVP_PKEY_CTX_new_from_pkey( NULL, alice_key_pair, NULL );
EVP_PKEY_derive_init( ctx );
EVP_PKEY_CTX_set_params( ctx, params );
EVP_PKEY_derive_set_peer( ctx, bob_key_pair );
EVP_PKEY_derive( ctx, alice_derived_key, &alice_key_len );
EVP_PKEY_CTX_free( ctx );

// Perform Alice DH operation and key derivation

ctx = EVP_PKEY_CTX_new_from_pkey( NULL, bob_key_pair, NULL );
EVP_PKEY_derive_init( ctx );
EVP_PKEY_CTX_set_params( ctx, params );
EVP_PKEY_derive_set_peer( ctx, alice_key_pair );
EVP_PKEY_derive( ctx, bob_derived_key, &bob_key_len );
EVP_PKEY_CTX_free( ctx );
```

```
if (alice_key_len == bob_key_len && memcmp( alice_derived_key, bob_derived_key,
    alice_key_len) == 0) {
    printf( "True\n" );
} else {
    printf( "False\n" );
}
```

Write a program to encrypt the contents of a file using an hybrid encryption. The program should accept three parameters: the key file, the input and the output files.

Tip: you can use less arguments and consider the use of `stdin` and `stdout` in the absence of file specifications.

Check the size of the resulting encrypted files for each input file. Explain why they are always bigger and the actual size increment.

Develop a program similar to the previous but able to decipher a file. Three arguments should be sent to the program: the key file, file to decipher and the output file.

Tip: Try to use a program written in a different language than the one with which you did the encryption.

References

- PKCS #12: Personal Information Exchange Syntax v1.1, RFC 7292, July 2014 (`https://www.rfc-editor.org/rfc/pdfrfc/rfc7292.txt.pdf`)
- The Java Tutorial: Security Features in Java SE `https://docs.oracle.com/javase/tutorial/security/index.html`
- Java Documentation: Security `http://docs.oracle.com/javase/8/docs/technotes/guides/security/index.html`
- Welcome to pyca/cryptography (`https://cryptography.io`)
- Libcrypto API (`https://wiki.openssl.org/index.php/Libcrypto_API`)