Trabalho Prático 1 de Estruturas Criptográficas

- **Autores:** (Grupo 9)
  - Nelson Faria (A84727)
  - Miguel Oliveira (A83819)

Exercício 1

```
'''
Imports necessários para a execução deste mesmo Notebook
'''
import io, os
import multiprocessing as mp

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.hazmat.primitives import padding

from cryptography.hazmat.primitives.asymmetric import dsa
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.asymmetric.padding import PSS, MGF1, PKCS1v15

from cryptography.hazmat.primitives.serialization import Encoding, PublicFormat, PrivateForm
from cryptography.hazmat.primitives.serialization import load_pem_public_key
from cryptography.hazmat.primitives.serialization import load_pem_private_key
```

O principal objetivo deste primeiro exercício prático passava pela implementação de uma **comunicação síncrona** e **segura** entre duas entidades, um **receiver** e um **emitter** seguindo determinados parâmetros que vão sendo explicitados e demonstrados em cada uma das alíneas que se seguem.

Alínea a - Implementar um gerador de nounces

Como se encontra explícito no próprio título desta subsecção, a principal finalidade desta alínea era a implementação de um **gerador de nounces** de modo a que, sempre que seja solicitado uma nova instância de **nounce**, temos a garantia de que ele é **aleatório** e ainda que **nunca foi retornado**, numa ocasião prévia por esse mesmo gerador.

Com isto em mente, desenvolvemos a seguinte classe em **Python**, designada de **NGenerator**, a qual apenas apresenta um método que permite o acesso a um novo **nounce**, designado de **get()**. De modo a implementar essa funcionalidade no **NGenerator**, recorremos a uma lista que vai armazenando os **nounces** que vão sendo retornados, à qual acedemos sempre aquando da solicitação da geração de um novo nounce de modo a verificar se tal nounce já foi retornado previamente. Em caso afirmativo, essa instância recém-gerada **é descartada** e

é gerada **uma nova instância** para ser retornada, adicionando-a, previamente ao *historic* do respetivo **NGenerator**.

```
'''
Classe que permite gerar
nounces que nunca se repetem
'''
class NGenerator :

    # Construtor para objetos da classe NGenerator
    def __init__(self,size) :

        self.size = size
        self.historic = []

    '''
    Método que nos permite obter um novo nounce
    '''
    def get(self) :

        nounce = os.urandom(self.size)
        while nounce in self.historic :
            nounce = os.urandom(self.size)
        self.historic.append(nounce)
        return nounce

    '''
    Método que nos permmite adicionar um novo nounce
    ao historico. Método importante para registar os
    nounces usados pelo peer.
    '''
    def addToHistoric(self,nounce) :

        self.historic.append(nounce)
```

De seguida, mostramos o exemplo de execução do método que permite obter uma nova instância de **nounce**.

```
ng = NGenerator(32)
ng.get()
```

```
b'\xcb=\xbc\xea\xcb\xf8\x8b\xb1\x9b\xb7\xfdf\xccL\x9d\x1d\x95B\xec:)\x87]aW\xe8\n\xe3g\xd3\x
```

Implementação da comunicação entre as duas entidades (*Emitter* e *Receiver*) - Alíneas b) e c)

As alíneas **b** e **c** não podem ser isoladas do mesmo modo que aconteceu com a alínea a, visto que estas últimas representam alguns **requisitos** aos quais a

comunicação entre as duas diferentes partes devem obedecer, pelo que falaremos de cada uma delas ao longo da explicação acerca da implementação realizada pelo nosso grupo quando tal seja oportuno.

Numa primeira fase, pensamos que seria necessário ambas as entidades conhecerem a **chave pública** uma da outra, de modo a que fosse possível, para qualquer uma destas, verificarem as **assinaturas** uma da outra, na fase de **handshake**, a qual segue o **algorítmo de Diffie-Hellman**, de modo a ambas as partes acordarem num **chave simétrica** a ser usada para a **cifragem de mensagens** durante a comunicação. Definimos, para além disso, os parâmetros **p** e **g** a serem usados no **Diffie-Hellman**.

```
# Chaves públicas DSA de emitter e receiver
sender_public_key = None
receiver_public_key = None

# RFC 3526's parameters for Diffie-Hellman. Easier to hardcode...
p = 0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020BBEA63B139B22514A08
g = 2
params_numbers = dh.DHParameterNumbers(p,g)
parameters = params_numbers.parameters()
```

Os métodos apresentados já de seguida são responsáveis pela geração de **chaves DSA privadas** e **públicas**, de modo a que seja possível, aquando da fase de **handshake**, para ambos os intervinientes na comunicação se conseguirem autenticar. No nosso contexto de teste em particular, o processo "principal" (método **main**) é responsável por gerar ambas as chaves privadas numa primeira instância, e, de seguida, transmití-las aos respetivos portadores das mesmas, cada um a correr num **processo distinto** (recorremos ao **multiprocessing do Python**).

De seguida, a partir do par de **chaves privadas** gerado, gera, para cada uma, a sua respetiva **chave pública**, esta sim, conhecida de ambas as entidades.

```
# Size of dsa keys
DSA_KEY_SIZE = 2048 #bits

'''
Função que permite gerar uma chave privada e
que irá ser usada por ambas as entidades, sender e
emitter para poder realizar a troca de chaves
'''
def generatePrivateKeys() :

    sender_private_key = dsa.generate_private_key(
        key_size=DSA_KEY_SIZE,
    )
    receiver_private_key = dsa.generate_private_key(
```

```
        key_size=DSA_KEY_SIZE,
    )
    return sender_private_key,receiver_private_key

'''
Função que recebe o par de chaves privadas geradas
e gera cada uma das correspondentes chaves públicas
'''
def generatePublicKeys(private_pair) :

    global sender_public_key, receiver_public_key
    sender_public_key = private_pair[0].public_key()
    receiver_public_key = private_pair[1].public_key()
```

Fase de Handshake Uma vez explicado o procedimento para a **geração** dos **pares de chaves DSA públicas** e **privadas** para cada uma das entidades que fazem parte da comunicação neste nosso exemplo prático, passamos agora para a explicação acerca da fase de **handshake**, na qual decorre não só o acordo, entre ambas as partes, para a **chave simétrica** a ser usada para a **cifragem** e respetiva **garatia de integridade (mac)** de mensagens, como também a **autenticação** de cada uma das partes intervenientes na comunicação.

Esta fase é composta por 4 etapas essenciais:

- **Emitter** gera a sua chave privada **DH**, g^x, e envia-a para o **Receiver**.
- **Receiver** gera, por sua vez, a sua chave **DH**, g^y, estando já em condições de computar a chave partilhada, g^{xy}. De seguida envia a seguinte mensagem para o **Emitter**: (g^y, $cifra_k$(S(g^y,g^x))), onde k=g^{xy}, **cifra** é o algorítmo de cifragem optado pelo grupo para o problema em questão e que irá ser aprofundado mais à frente e em que o oráculo S(.) representa a assinatura do par de chaves privadas **DH** geradas por cada uma das entidades realizada com a **chave privada DSA** do **Receiver**, de modo a que seja possível ao **Emitter** verificar a assinatura através da **chave pública DSA** do próprio **Receiver** e, deste modo, comprovar a autenticidade desse mesmo agente. Neste ponto, o **Receiver** já se encontra em condições de calcular a **chave simétrica**, g^{xy}, que irá ser diretamente aplicada no algorítmo de cifragem, que, no nosso caso, consiste na aplicação da cifra **AES** no modo **CBC**, assim como no **HMAC**, que será usado para a geração de um digest, com o intuito de que seja sempre possível verificar a **integridade** de toda e qualquer mensagem trocada entre estas duas entidades.
- **Emitter**, após receber a chave privada, g^y, gerada proveniente do seu peer nesta fase de handshake, poderá, desde já, calcular, à semelhança do que já foi explicado no ponto anterior, a chave g^{xy}. Após isto, em jeito de **autenticação** de si próprio para com o seu peer, envia a seguinte mensagem: $cifra_k$(S(g^x,g^y)), com k=g^{xy}, **cifra** é o algorítmo de cifragem

optado pelo grupo para o problema em questão e que irá ser aprofundado mais à frente, para que tal assinatura seja, depois, verificada pelo **Receiver**.
- **Receiver** faz uso da chave gerada e envia para o **Emitter** a seguinte mensagem: "END HANDSHAKE", fazendo já uso dos métodos de **cifragem** e **mac** já explicados anteriormente, de modo a que seja garantido o **sincronismo** característico desta comunicação.

Mostra-se, de seguida, as funções em python responsáveis pela implementação daquilo que foi descrito até agora, em que cada uma representa o procedimento seguido por cada um dos peers.

Ambas estas funções recebem uma instância de **Connection**, **conn**, a qual permite a cada uma das entidades comunicar com a outra, a respetiva chave privada da entidade que representam, assim como um **NGenerator**, este último que permite a geração de **nounces** e que tem utilidade na cifragem de mensagens, cujo procedimento será explicado mais à frente. Nota para o facto de que ambas as funções retornam a **chave simétrica** acordada para a cifragem de mensagens no decorrer da comunicação entre ambos os intervenientes.

```
'''
Função responsável pelo handshake a ser
realizado por parte do sender. Retorna a chave a
ser usada para a comunicação entre ambas as entidades.
'''
def sender_handshake(conn, private_key, ng) :

    # Generate and Send the client's generated key: g^x
    dh_g_x = parameters.generate_private_key()
    dh_g_x_as_bytes = dh_g_x.private_bytes(Encoding.PEM,PrivateFormat.PKCS8,NoEncryption())
    conn.send(dh_g_x_as_bytes)

    # Recebemos a primeira mensagem do receiver
    data = conn.recv()
    args = data.split(sep=b'\r\n\r\n')
    dh_g_y_as_bytes = args[0]
    dh_g_y = load_pem_private_key(dh_g_y_as_bytes,password=None)
    ciphered_signature = args[1]

    # Generate the shared key between server<->client communication
    shared_key = dh_g_x.exchange(dh_g_y.public_key())
    # Perform key derivation.
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',
    ).derive(shared_key)
```

```
    signature = decrypt(derived_key,ciphered_signature,ng)

    # Verificamos a assinatura do receiver -> FIXME WITH TRY... CATCH
    verify(receiver_public_key,dh_g_y_as_bytes + b'\r\n\r\n' + dh_g_x_as_bytes, signature)


    # Envia a respetiva assinatura para ser verificada pelo receiver
    conn.send(encrypt(derived_key,sign(private_key, dh_g_x_as_bytes + b'\r\n\r\n' + dh_g_y_a

    # No final é recebida, pelo sender (emitter) uma mensagem de termino da fase de handshak
    assert decrypt(derived_key,conn.recv(),ng).decode('utf-8') == 'END HANDSHAKE'

    print('[SENDER] Handshake phase successfully done!')

    return derived_key
'''
Função responsável pelo handshake a ser
realizado por parte do receiver. Retorna a chave a
ser usada para a comunicação entre ambas as entidades.
'''
def receiver_handshake(conn, private_key, ng) :

    # Recebemos a chave gerada pelo sender
    dh_g_x_as_bytes = conn.recv()
    dh_g_x = load_pem_private_key(dh_g_x_as_bytes,password=None)

    # Generate and Send the server's generated key: g^y
    dh_g_y = parameters.generate_private_key()
    dh_g_y_as_bytes = dh_g_y.private_bytes(Encoding.PEM,PrivateFormat.PKCS8,NoEncryption())

    # Generate the shared key between server<->client communication
    shared_key = dh_g_y.exchange(dh_g_x.public_key())

    # Perform key derivation.
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',
    ).derive(shared_key)

    # Enviamos para o sender a chave gerada e a respetiva assinatura
    conn.send(dh_g_y_as_bytes + b'\r\n\r\n' + encrypt(derived_key,sign(private_key, dh_g_y_a
```

```
    # Recebemos a assinatura e verificamos se é autentica -> FIXME WITH TRY... CATCH
    data = conn.recv()
    signature = decrypt(derived_key,data,ng)
    verify(sender_public_key, dh_g_x_as_bytes + b'\r\n\r\n' + dh_g_y_as_bytes, signature)

    # No final é enviada uma mensagem de termino da fase de handshake
    conn.send(encrypt(derived_key,'END HANDSHAKE'.encode('utf-8'),ng))

    print('[RECEIVER] Handshake phase successfully done!')

    return derived_key
```

Gestão das assinaturas (com o uso de chaves DSA) Para a agilizar todo o processo de **assinatura** e **verificação** da mesma mediante a apresentação de uma assinatura propriamente dita, foram implementadas as funções **sign()** e **verify()**. A primeira recebe a **chave privada DSA** da entidade que pretende assinar uma determinada mensagem, assim como o próprio conteúdo da mensagem e retorna um digest de **32 bytes**, fazendo uso do hash **SHA256**. Já a segunda, recebe a **chave pública DSA** da entidade que assinou determinada mensagem, o conteúdo da própria mensagem, como não poderia deixar de ser e ainda a respetiva **tag gerada** pelo gerador da mensagem. Nota para o facto de que, aquando da não correspondência entre mensagem e tag, aquando do processo de verificação, é lançada uma exceção pelo próprio método **verify()**.

Contextualizando com a subsecção de **handshake**, os oráculos S(.) de cada uma das entidades mencionados na explicação de implementação desta fase inicial da comunicação entre o **Emitter** e o **Receiver**, não são mais do que a aplicação direta da função **sign** com a **chave privada DSA** da entidade respetiva.

```
'''
Função que permite assinar uma determinada mensagem
com uma determinada chave privada
'''
def sign(private_key, data) :

    signature = private_key.sign(
        data,
        hashes.SHA256()
    )
    return signature

'''
Função que permite verificar uma determinada assinatura.
'''
def verify(public_key, data, signature) :
```

```
public_key.verify(
    signature,
    data,
    hashes.SHA256()
)
```

Processo de cifragem e decifragem de mensagens No contexto da cifragem e decifragem de dados, realizados quer pelo **Emitter**, quer pelo **Receiver**, aquando da intenção de comunicar com o seu peer, foram também implementados dois métodos com o intuito de facilitar a aplicação prática não só de **cifragem** dos dados, de modo a garantir a **confidencialidade** da comunicação, mas também, o uso do algorítmo **hmac** de modo a garantir a integridade das mensagens trocadas entre os dois intervenientes.

Por outras palavras, os métodos **encrypt()** e **decrypt()** não podem ser interpretados no seu sentido literal, tendo em conta que cobrem também todas as questões relacionadas com a geração e verificação de tags relacionadas com o **hmac**, para além de aplicarem a cifra **AES** no modo **CBC**. Tendo optado, uma vez, pelo modo **CBC**, seria necessário usar algum tipo de **padding**, de modo a permitir o envio de mensagem de qualquer comprimento, não restringindo o conjunto de mensagens a serem enviadas àquelas que possuam um comprimento que seja múltiplo do tamanho de cada bloco da cifra **AES**, de **16 bytes**. O algorítmo de padding utilizado foi, então, o **PKCS7**. Para além disto, o modo **CBC** implica o uso de um vetor de inicialização, **iv** que precisa de ser único para cada mensagem trocada entre peers. Partindo deste pressuposto, seria necessário algum tipo de gerador que garanta esta característica. Ora, já resolvemos este problema no contexto da **alínea a**, com a implementação da classe **NGenerator**. Com esta podemos definir o tamanho de **nounces** a serem gerados para **16 bytes** e temos este problema completamente resolvido. Será importante não esquecer que, aquando do consumo de uma mensagem, o **recetor** (quer seja ele **Emitter** ou **Receiver**) da mesma terá de notificar a sua instância de **NGenerator** de que o **iv** nela presente, e gerado pelo peer, já foi usado, de modo a que este não seja reutilizado na comunicação.

A construção do "criptograma" como um todo, isto é, incluindo já a **tag** para a verificação de integridade do criptograma própriamente dito, ou seja, gerado pela aplicação do **AES** no modo **CBC**, foi então feita do seguinte modo:

> criptograma = **(N,iv,ct,tag)** where N=**sizeof**(iv+ct), tag=$hmac_k$(iv,ct), ct=$cifra_k$(m)

No pseudo-código acima mencionado importa que referir que **criptograma** é um array de bytes composto por **N**, com um comprimento de **4 bytes** que representa o tamanho do **(iv,ct)**, em que **ct** é o resultado da aplicação da cifra **AES** com a **chave k** no modo **CBC** com o uso do **vetor de inicialização iv**. Este **N** é necessário de modo a que seja possível, aquando do processo de decifragem,

saber onde termina o campo **(iv,ct)** e onde começa o campo **tag**, uma vez que o **(iv,ct)** não possui um comprimento fixo (apesar de ser sempre múltipo de **16**).

Dito isto, apresentamos agora as funções **encrypt()** e **decrypt()**.

```
PKCS7_BIT_LEN = 128 # bits
AES_BLOCK_LEN = 16 # bytes
HMAC_KEY_LEN = 32 # bytes

# Header do ciphertext que guarda o tamanho do ciphertext propriamente dito (sem o digest)
CIPHERTEXT_HEADER_LEN = 4 # bytes

# Receives and returns bytes.
def encrypt(k, m, iv_gen):

    padder = padding.PKCS7(PKCS7_BIT_LEN).padder()
    padded_data = padder.update(m) + padder.finalize()
    iv = iv_gen.get()
    cipher = Cipher(algorithms.AES(k), modes.CBC(iv))
    encryptor = cipher.encryptor()

    ct = encryptor.update(padded_data) + encryptor.finalize()
    ct = iv+ct
    len_ct = len(ct)
    len_ct_bytes = len_ct.to_bytes(CIPHERTEXT_HEADER_LEN,'little')

    h = hmac.HMAC(k, hashes.SHA256())
    h.update(ct)
    digest = h.finalize()
    return len_ct_bytes+ct+digest

# Receives and returns bytes.
def decrypt(k, c, iv_gen):

    len_ct_bytes = c[:CIPHERTEXT_HEADER_LEN]
    len_ct = int.from_bytes(len_ct_bytes,'little',signed=True)
    ct, digest = c[CIPHERTEXT_HEADER_LEN:(len_ct+CIPHERTEXT_HEADER_LEN)], c[(len_ct+CIPHERTE
    iv, ct = ct[:AES_BLOCK_LEN], ct[AES_BLOCK_LEN:]

    iv_gen.addToHistoric(iv)

    # Verificamos, logo, a assinatura
    h = hmac.HMAC(k, hashes.SHA256())
    h.update(iv+ct)
    h.verify(digest)

    cipher = Cipher(algorithms.AES(k), modes.CBC(iv))
```

```
    decryptor = cipher.decryptor()
    pt = decryptor.update(ct) + decryptor.finalize()
    unpadder = padding.PKCS7(PKCS7_BIT_LEN).unpadder()
    pt = unpadder.update(pt) + unpadder.finalize()
    return pt
```

Nota para o facto de que a função **encrypt()** para além da chave simétrica, **k** e a mensagem para ser cifrada, **m**, recebe também o **NGenerator** iv_gen, o qual é responsável pela geração de **iv's** para aplicação direta do modo **CBC** com a cifra **AES**. Algo semelhante acontece com a função **decrypt()**.

Comunicação entre Emitter e Receiver Neste ponto, encontramo-nos em condições de descrever como decorre a comunicação propriamente dita entre **Emitter** e **Receiver**. Após a fase de **handshake**, com a aplicação do algorítmo de **Diffie-Hellman**, ambas as partes possuem a chave simétrica a ser utilizada para a cifragem de conteúdos trocados entre si. Assim sendo, a implementação da comunicação torna-se algo trivial.

Os métodos apresentados no excerto de código seguinte implementam o comportamento de cada uma das partes na comunicação entre essas mesmas partes. Ambos os métodos **sender()** e **receiver()** recebem uma instância de **Connection**, **conn**, as quais representam as extremidades de um **pipe bidirecional**, que lhes permite comunicar um com o outro e a respetiva **chave privada DSA** de cada um, as quais serão usadas na fase de **handshake**, como já foi explicado anteriormente.

No entanto, a função **sender()** exige um outro parâmetro designado de **stdin**. Este parâmetro não é mais do que outra instância de **Connection**, mas uma outra que permite a comunicação com o processo **pai** (método main). Fomos obrigado a recorrer a isto, visto que pretendiamos que o processo que encapsulava a execução do **Emitter (sender)**, lêsse mensagens do **standard in** (ler do teclado recorrendo à função **input()** disponibilizada pelo **python**). No entanto, após vários testes verificamos que tal só poderia acontecer no **processo pai**. Deste modo, a nossa solução passou por criar um **pipe** adicional em que uma das extremidades fica entregue ao **Emitter** e a outra, ao processo responsável por lançar o **Emitter** e o **Receiver**.

```
'''
Função que representa a execução da entidade sender/emitter
'''
def sender(conn,private_key,stdin):

    iv_generator = NGenerator(AES_BLOCK_LEN)
    # Iniciamos a fase de handshake com o receiver...
    shared_key = sender_handshake(conn,private_key,iv_generator)
    stdin.send('ready')
```

```
    message = stdin.recv()

    while message != 'exit' and len(message) > 0 :
        conn.send(encrypt(shared_key,message.encode('utf-8'),iv_generator))
        assert 'ok' == decrypt(shared_key,conn.recv(),iv_generator).decode('utf-8') # para g
        ''' Pedimos à thread principal a
        próxima mensagem lida a partir do stdin'''
        stdin.send('next')
        message = stdin.recv()

    # Para terminar a conexão
    conn.send(encrypt(shared_key,'exit'.encode('utf-8'),iv_generator))
    conn.close()
    print('[Emitter] SHUTDOWN')
'''
Função que representa a execução da entidade receiver
'''
def receiver(conn,private_key):

    iv_generator = NGenerator(AES_BLOCK_LEN)
    # Iniciamos a fase de handshake com o sender
    shared_key = receiver_handshake(conn,private_key,iv_generator)

    try:
        message = decrypt(shared_key,bytes(conn.recv()),iv_generator).decode('utf-8')
        while message != 'exit':
            # Imprimimos o conteudo da mensagem recebida
            print('[Receiver] RECEIVED: ' + message)
            conn.send(encrypt(shared_key,'ok'.encode('utf-8'),iv_generator)) # para garanti
            message = decrypt(shared_key,bytes(conn.recv()),iv_generator).decode('utf-8')
    except EOFError:
        print('[Receiver] SHUTDOWN')

    print('[Receiver] SHUTDOWN')
    conn.close()
```

Posto isto, estamos em condições de apresentar o código do programa principal, responsável por arrancar ambas as entidades, **Emitter** e **Receiver**:

```
'''
Função responsável por arrancar a
execução de ambas as entidades, o emitter
e o receiver. Para além disto é responsável por
mediar a comunicação entre o stdin e o sender
'''
def communication() :
```

```
    try:
        mp.set_start_method('fork')
    except:
        print("O start_method já foi inicializado anteriormente ")

    # Geramos a chave privada para cada uma das entidades e, de seguida, as chaves públicas
    sender_private_key, receiver_private_key = generatePrivateKeys()
    generatePublicKeys((sender_private_key, receiver_private_key))

    receiver_conn, sender_conn = mp.Pipe()
    receiver_stdin, main_stdin = mp.Pipe()
    s = mp.Process(target=sender, args=(sender_conn,sender_private_key,receiver_stdin))
    r = mp.Process(target=receiver, args=(receiver_conn,receiver_private_key))
    s.start()
    r.start()

    # Esperamos que o emitter esteja a postos para receber mensagens para serem enviadas
    assert 'ready' == main_stdin.recv()

    message = input('[Emitter] > Mensagem a ser enviada: \n')
    while len(message) != 0 and message != 'exit':
        main_stdin.send(message)
        ''' Esperamos a confirmação do sender para
        receber nova mensagem do stdin'''
        assert 'next' == main_stdin.recv()
        message = input('[Emitter] > Mensagem a ser enviada: \n')
    # Para terminar a ligação
    main_stdin.send('')
communication()
```

```
[SENDER] Handshake phase successfully done![RECEIVER] Handshake phase successfully done!


[Emitter] > Mensagem a ser enviada:
 Hello!

[Receiver] RECEIVED: Hello!

[Emitter] > Mensagem a ser enviada:
 Tudo bem contigo?

[Receiver] RECEIVED: Tudo bem contigo?

[Emitter] > Mensagem a ser enviada:
 Xau

[Receiver] RECEIVED: Xau
```

```
[Emitter] > Mensagem a ser enviada:


[Emitter] SHUTDOWN[Receiver] SHUTDOWN
```

Alínea d - Uso de ECDH e ECDSA

Nesta alínea é-nos pedido para substituir o algorítmo de acordo de chaves **DH** pelo **ECDH** e, para além disto, ao invés de utilizar o **DSA** como algoritmo de assinatura de dados, passar a usar o **ECDSA**. Assim sendo, a principal diferença para o esquema anterior reside na fase de **handshake**, uma vez que o desenrolar da comunicação após esta fase em nada depende destes algorítmos citados anteriormente. Após a fase de handshake, os algorítmos usados são a cifra **AES** com o modo **CBC**, tirando partido da chave acordada na fase de **handshake** e de **iv's** que vão sendo gerados por instâncias de **NGenerator** pertencentes a ambas as partes envolvidas na comunicação, assim como o **HMAC** para geração de tag's com o intuito de garantir integridade na comunicação.

Partindo disto, notamos que à partida, temos que mudar a geração de **chaves públicas** e **privadas DSA** para a geração de *EllipticCurvePublicKeys* e *EllipticCurvePrivateKeys*.

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

sender_ec_public_key = None
receiver_ec_public_key = None

'''
Função que permite gerar uma chave privada e
que irá ser usada por ambas as entidades, sender e
emitter para poder realizar a troca de chaves
'''
def generateEllipticCurvePrivateKeys() :

    sender_private_key = ec.generate_private_key(
        ec.SECP384R1()
    )
    receiver_private_key = ec.generate_private_key(
        ec.SECP384R1()
    )
    return sender_private_key,receiver_private_key

'''
Função que recebe o par de chaves privadas geradas
e gera cada uma das correspondentes chaves públicas
```

```
'''
def generateEllipticCurvePublicKeys(private_pair) :

    global sender_ec_public_key, receiver_ec_public_key
    sender_ec_public_key = private_pair[0].public_key()
    receiver_ec_public_key = private_pair[1].public_key()
```

O passo seguinte passa por alterar as funções de **sign()** e **verify()**, visto que agora estamos a usar **curvas elípticas** e não **DSA**.

```
'''
Função que permite assinar uma determinada mensagem
com uma determinada chave privada
'''
def sign_ec(private_key, data) :

    signature = private_key.sign(
        data,
        ec.ECDSA(hashes.SHA256())
    )
    return signature

'''
Função que permite verificar uma determinada assinatura.
'''
def verify_ec(public_key, data, signature) :

    public_key.verify(
        signature,
        data,
        ec.ECDSA(hashes.SHA256())
    )
```

De momento, tendo em conta que na nossa fase de handshake estamos a utilizar **Diffie-Hellman**, esta terá que ser alterada no sentido de passarmos a aplicar **ECDH** - *Elliptic Curve Diffie-Hellman Key Exchange algorithm.*

```
'''
Função responsável pelo handshake a ser
realizado por parte do sender. Retorna a chave a
ser usada para a comunicação entre ambas as entidades.
'''
def sender_ec_handshake(conn, private_key, ng) :

    # Generate and Send the client's generated key: g^x
    dh_g_x = ec.generate_private_key(ec.SECP384R1())
    dh_g_x_as_bytes = dh_g_x.private_bytes(Encoding.PEM,PrivateFormat.PKCS8,NoEncryption())
```

```
    conn.send(dh_g_x_as_bytes)

    # Recebemos a primeira mensagem do receiver
    data = conn.recv()
    args = data.split(sep=b'\r\n\r\n')
    dh_g_y_as_bytes = args[0]
    dh_g_y = load_pem_private_key(dh_g_y_as_bytes,password=None)
    ciphered_signature = args[1]

    # Generate the shared key between server<->client communication
    shared_key = dh_g_x.exchange(ec.ECDH(),dh_g_y.public_key())
    # Perform key derivation.
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',
    ).derive(shared_key)

    signature = decrypt(derived_key,ciphered_signature,ng)

    # Verificamos a assinatura do receiver -> FIXME WITH TRY... CATCH
    verify_ec(receiver_ec_public_key,dh_g_y_as_bytes + b'\r\n\r\n' + dh_g_x_as_bytes, signat


    # Envia a respetiva assinatura para ser verificada pelo receiver
    conn.send(encrypt(derived_key,sign_ec(private_key, dh_g_x_as_bytes + b'\r\n\r\n' + dh_g_

    # No final é recebida, pelo sender (emitter) uma mensagem de termino da fase de handshak
    assert decrypt(derived_key,conn.recv(),ng).decode('utf-8') == 'END HANDSHAKE'

    print('[SENDER] Handshake phase successfully done!')

    return derived_key


'''
Função responsável pelo handshake a ser
realizado por parte do receiver. Retorna a chave a
ser usada para a comunicação entre ambas as entidades.
'''
def receiver_ec_handshake(conn, private_key, ng) :

    # Recebemos a chave gerada pelo sender
    dh_g_x_as_bytes = conn.recv()
    dh_g_x = load_pem_private_key(dh_g_x_as_bytes,password=None)
```

```
    # Generate and Send the server's generated key: g^y
    dh_g_y = ec.generate_private_key(ec.SECP384R1())
    dh_g_y_as_bytes = dh_g_y.private_bytes(Encoding.PEM,PrivateFormat.PKCS8,NoEncryption())

    # Generate the shared key between server<->client communication
    shared_key = dh_g_y.exchange(ec.ECDH(),dh_g_x.public_key())

    # Perform key derivation.
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',
    ).derive(shared_key)

    # Enviamos para o sender a chave gerada e a respetiva assinatura
    conn.send(dh_g_y_as_bytes + b'\r\n\r\n' + encrypt(derived_key,sign_ec(private_key, dh_g_

    # Recebemos a assinatura e verificamos se é autentica -> FIXME WITH TRY... CATCH
    data = conn.recv()
    signature = decrypt(derived_key,data,ng)
    verify_ec(sender_ec_public_key, dh_g_x_as_bytes + b'\r\n\r\n' + dh_g_y_as_bytes, signatu

    # No final é enviada uma mensagem de termino da fase de handshake
    conn.send(encrypt(derived_key,'END HANDSHAKE'.encode('utf-8'),ng))

    print('[RECEIVER] Handshake phase successfully done!')

    return derived_key
```

Agora teremos que alterar as funções **sender()** e **receiver()**, aplicando as funções agora implementadas:

```
'''
Função que representa a execução da entidade sender/emitter
'''
def sender_ec(conn,private_key,stdin):

    iv_generator = NGenerator(AES_BLOCK_LEN)
    # Iniciamos a fase de handshake com o receiver...
    shared_key = sender_ec_handshake(conn,private_key,iv_generator)
    stdin.send('ready')

    message = stdin.recv()
```

```
    while message != 'exit' and len(message) > 0 :
        conn.send(encrypt(shared_key,message.encode('utf-8'),iv_generator))
        assert 'ok' == decrypt(shared_key,conn.recv(),iv_generator).decode('utf-8') # para g
        ''' Pedimos à thread principal a
        próxima mensagem lida a partir do stdin'''
        stdin.send('next')
        message = stdin.recv()

    # Para terminar a conexão
    conn.send(encrypt(shared_key,'exit'.encode('utf-8'),iv_generator))
    conn.close()
    print('[Emitter] SHUTDOWN')
'''
Função que representa a execução da entidade receiver
'''
def receiver_ec(conn,private_key):

    iv_generator = NGenerator(AES_BLOCK_LEN)
    # Iniciamos a fase de handshake com o sender
    shared_key = receiver_ec_handshake(conn,private_key,iv_generator)

    try:
        message = decrypt(shared_key,bytes(conn.recv()),iv_generator).decode('utf-8')
        while message != 'exit':
            # Imprimimos o conteudo da mensagem recebida
            print('[Receiver] RECEIVED: ' + message)
            conn.send(encrypt(shared_key,'ok'.encode('utf-8'),iv_generator)) # para garanti
            message = decrypt(shared_key,bytes(conn.recv()),iv_generator).decode('utf-8')
    except EOFError:
        print('[Receiver] SHUTDOWN')

    print('[Receiver] SHUTDOWN')
    conn.close()
```

Por fim, teremos que implementar a função que arranca com o **Emitter** e **Receiver**:

```
'''
Função responsável por arrancar a
execução de ambas as entidades, o emitter
e o receiver. Para além disto é responsável por
mediar a comunicação entre o stdin e o sender
'''
def ec_communication() :
    try:
        mp.set_start_method('fork')
```

```
    except:
        print("O start_method já foi inicializado anteriormente ")

    # Geramos a chave privada para cada uma das entidades e, de seguida, as chaves públicas
    sender_private_key, receiver_private_key = generateEllipticCurvePrivateKeys()
    generateEllipticCurvePublicKeys((sender_private_key, receiver_private_key))

    receiver_conn, sender_conn = mp.Pipe()
    receiver_stdin, main_stdin = mp.Pipe()
    s = mp.Process(target=sender_ec, args=(sender_conn,sender_private_key,receiver_stdin))
    r = mp.Process(target=receiver_ec, args=(receiver_conn,receiver_private_key))
    s.start()
    r.start()

    # Esperamos que o emitter esteja a postos para receber mensagens para serem enviadas
    assert 'ready' == main_stdin.recv()

    message = input('[Emitter] > Mensagem a ser enviada: \n')
    while len(message) != 0 and message != 'exit':
        main_stdin.send(message)
        ''' Esperamos a confirmação do sender para
        receber nova mensagem do stdin'''
        assert 'next' == main_stdin.recv()
        message = input('[Emitter] > Mensagem a ser enviada: \n')
    # Para terminar a ligação
    main_stdin.send('')

ec_communication()
```

```
O start_method já foi inicializado anteriormente
[RECEIVER] Handshake phase successfully done![SENDER] Handshake phase successfully done!


[Emitter] > Mensagem a ser enviada:
 Hello!

[Receiver] RECEIVED: Hello!

[Emitter] > Mensagem a ser enviada:
 Tudo bem contigo?

[Receiver] RECEIVED: Tudo bem contigo?

[Emitter] > Mensagem a ser enviada:
 Xau...

[Receiver] RECEIVED: Xau...

[Emitter] > Mensagem a ser enviada:
 exit
```

```
[Emitter] SHUTDOWN[Receiver] SHUTDOWN
```

Com o intuito de embelezar a nossa solução deste exercício desenvolvemos o seguinte programa que permite ao utilizador testar ambas as versões desenvolvidas, isto é, com e sem o uso de **Elliptic Curves**.

```python
def main():

    msg = input('Want to use Elliptic curves? (y/n)')
    if msg=='y' or msg=='Y':
        print('> Started using EC...')
        ec_communication()
    elif msg=='n' or msg=='N':
        print('> Started not using EC...')
        communication()
    else:
        print('Invalid option, please try again... bye')

if __name__ == "__main__":
    main()
```

```
Want to use Elliptic curves? (y/n) y

> Started using EC...
O start_method já foi inicializado anteriormente
[RECEIVER] Handshake phase successfully done![SENDER] Handshake phase successfully done!


[Emitter] > Mensagem a ser enviada:
 Hello

[Receiver] RECEIVED: Hello

[Emitter] > Mensagem a ser enviada:
 Este programa foi realizado no âmbito da UC de Estruturas Criptográficas!

[Receiver] RECEIVED: Este programa foi realizado no âmbito da UC de Estruturas Criptográfica

[Emitter] > Mensagem a ser enviada:
 Xau...

[Receiver] RECEIVED: Xau...

[Emitter] > Mensagem a ser enviada:


[Emitter] SHUTDOWN[Receiver] SHUTDOWN
```