Trabalho Prático 1 - Problema 1 - Grupo 18

Introdução ao problema

Neste problema foi-nos indicado para:

- Implementar uma AEAD com “Tweakable Block Ciphers” conforme está descrito na última secção do texto do Capítulo 1: Primitivas Criptográficas Básicas. A cifra por blocos primitiva, usada para gerar a “tweakable block cipher”, é o AES-256 ou o ChaCha20.
- Use esta construção para construir um canal privado de informação assíncrona com acordo de chaves feito com “X448 key exchange” e “Ed448 Signing & Verification” para autenticação dos agentes. Deve incluir uma fase de confirmação da chave acordada.

Resolução do problema

Imports

In [ ]:
```
import os, sys
from multiprocessing import Process, Pipe
from pickle import dumps, loads
from cryptography.hazmat.primitives import hashes, hmac, serialization
from cryptography.hazmat.primitives.asymmetric import ed448, x448
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.serialization import load_pem_public_
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, mo
from cryptography.hazmat.backends import default_backend
from cryptography.exceptions import InvalidSignature

# Número de bits
nrBits  = 128

# Número de bytes (nrBits/8)
nrBytes = 16
```

Geração de chaves assimétricas

Inicialmente tivemos de criar uma função que nos permitisse a criação de chaves assimétricas que nos permitisse realizar da forma mais correta a comunicação entre um recetor e um emissor.

Para implementar o protocolo pretendido utilizamos a curva eliptica X488. Assim, através do mesmo, conseguimos criar um par de chaves, sendo uma delas privada e a outra pública. A chave pública será uma chave partilhada entre as duas entidades. Os parâmetros utilizados fazem com que, mesmo utilizando um canal inseguro um attacker não consiga intercetar a mensagem e gerar a chave partilhada da comuniação.

In [ ]:
```
# Função responsável pela geração do par de chaves
def geracaoKeys ():

    # Gerar as chaves (private_key, peer_public_key) utilizando x488 Key
    private_key = x448.X448PrivateKey.generate()
    peer_public_key = private_key.public_key()

    # Gerar bytes de chave publica em bytes cifrados
    publicBytes = peer_public_key.public_bytes(
                        encoding = serialization.Encoding.PEM,
                        format   = serialization.PublicFormat.SubjectPubl

    # Crirar um pacote com a informação obtida
    pacote = {'packageCipher' : publicBytes}

    # Return da mensagem com a chave pública, bem como da chave privada
    return dumps(pacote), private_key
```

Geração de Assinatura

Neste momento implementamos o algoritmo EDSA. Dessa forma conseguimos assinar uma mensagem com um par de chaves assimétricas. Com a chave privada iremos criar a p´ropria assinatura e com a chave pública iremos verificar se a mesma é valida, de forma ao recetor poder verificar se a mensagem recebida foi devidamente assinada e se é autêntica.

Utilizamos para este processo o algoritmo ED448, que mais uma vez é um algoritmo de uma cruva eliptica que utiliza o algoritmo EDSA. Nele serão geradas as duas chaves, bem como a assinatura digital.

In [ ]:
```
# Função responsável por gerar a assinatura, conforme a mensagem que tem
def geracaoAssinatura (pacote):

    # Gerar a chave utilizada pela assinatura
    private_key = ed448.Ed448PrivateKey.generate()

    # Assinar o pacote que irá ser enviado para outros
    signature = private_key.sign(pacote)

    # Gerar bytes de chave publica em bytes cifrados
    public_keyBytes = private_key.public_key().public_bytes(
                            encoding = serialization.Encoding.PEM,
                            format   = serialization.PublicFormat.Subject

    # Geração de pacote final com a mensagem, assinatura e chave pública
    pacoteFinal = {'mensagem'   : pacote,
                   'assinatura' : signature,
                   'public_key' : public_keyBytes}

    # Return do pacte final obtido
    return pacoteFinal
```

Geração chave partilhada

QUando uma entidade recebe uma chave pública de outra, esta terá de gerar uma chave partilhada entre ambas. Geramos utilizando mais uma vez X448 a chave patilhada, através do uso da chave publica recebida na mensagem e a chave privada da entidade. A chave criada terá um tamanho de 56 bytes.

Como nos algoritmos utilizados não podemos possuir uma chave deste tamanho, utilizamos o algoritmo de derivação de chaves HKDF de modo a conseguirmos uma chave de apenas 16 bytes, ou seja, 128 bits.

In [ ]:
```
# Função responsável por gerar a chave partilhada que será usado no proce
def geracaoChavePartilhada (pacote, privateKey):

    # Retira chave publica do pacote
    peer_public_key = load_pem_public_key(pacote['packageCipher'])

    # Utilizando a sua chave partilhada e a chave publica de outro, gera
    key = privateKey.exchange(peer_public_key)

    # Cria uma chave de 16 bytes, ou seja 128 bits, utilizada durante o p
    sharedKey = HKDF( algorithm = hashes.SHA256(),
                      length  = nrBytes,
                      salt    = None,
                      info    = b'dados handshake',
                    ).derive(key)

    # Return da chave partilhada
    return sharedKey
```

Geração Blocos de Mensagem

Para utilizar o algoritmo AEAD teremos de preparar as mensagens, mais especificamente, dividir as mesmas em blocos de 16 byes, ou seja 128 bits. De relembrar que nem todos os blocos podem ocupar a totalidade dos 16 bytes, e por isso, é necessário efetuar um processo nos mesmos de modo a acrescentar um pading de zeros que complete os 16 bytes do bloco devidamente.

In [ ]:
```
# Função responsável por criar blocos de mensagens de n bytes (no nosso c
def criacaoBlocoMensagem (plainTextMessage):

    # Criar array que dvide a mensagem em blocos de nrBytes, no nosso cas
    messageBlocks = []
    for i in range(0, len(plainTextMessage), nrBytes):
        messageBlocks.append(plainTextMessage[i : i + nrBytes])

    # Cria uma array para a mensagem dividida
    message = []

    # Insere blocos no array message, transformando assim num byte string
    for j in range (0,len(messageBlocks)):
        message.insert(j, messageBlocks[j].encode('utf-8'))

    # Verificar qual o ultimo index do bloco, bem como o tamanho desse bl
    lastBlockIndex = len(messageBlocks)-1
    lastBlockSize  = len(message[lastBlockIndex])

    # Verifica se o bloco final precisa de padding e, se sim, acrescenta
    if lastBlockSize < nrBytes:
        message[lastBlockIndex] = message[lastBlockIndex].ljust(nrBytes,

    # Return no tamanho do ultimo bloco, bem como da mensagem diidida em
    return lastBlockSize, message
```

Geração Tweaks

Este algoritmo de cifragem, utiliza um input adicional denominada tweak. Ao contrário de uma cifra, que será igual para todos os blocos da mensagem, o tweak muda de bloco em bloco. A função que gera tweaks de 16 bytes, tal como o tamanho das mensagens, sendo gerado um para crifar os blocos de mensagens e um para autenticação. Em ambos os tweaks será utilizado um nonce para a sua geração, com metade do tamanho do tweak, ou seja, 8 bytes.

Como sabemos o utlimo bit de um tweak de autenticação possui o ultimo bit como 1. Desta fomra, a função modifyBit será responsável pela transformação do ultimo bit do tweak em 1. Os tewaks de cifra irão acabar contudo com o ultimo bit igual a 0.

In [ ]:
```python
# Função responsável por adicionar o último bit do tweak de autenticação
def modifyBit( n,  p,  b):
    mask = 1 << p
    return (n & ~mask) | ((b << p) & mask)

# Função responsável por gerar os tweaks
def generateTweak(numberBlocks, length, nonce):

    tweakBlock = []

    #Gera os tweaks para cifrar os blocos da mensagem
    for i in range(0,numberBlocks):

        #Adiciona o nonce e o respetivo contador ao tweak
        tweak = nonce + int(i).to_bytes(nrBytes // 2, byteorder='big')

        tweak = int.from_bytes(tweak, byteorder='big')
        #Remove o último bit
        tweak = tweak >> 1
        #Adiciona um bit a 0 na última posição do tweak - [N||i||0]
        tweak = tweak << 1
        tweak = tweak.to_bytes(nrBytes, byteorder='big')

        #Insere o tweak gerado no array dos tweaks
        tweakBlock.insert(i,tweak)

    #Gera o tweak de autenticação utilizando o nonce e o comprimento da m
    tweakAutenticate = nonce + length.to_bytes(nrBytes // 2, byteorder='b

    tweakAutenticate = int.from_bytes(tweakAutenticate, byteorder='big')
    #Adiciona um bit a 1 na última posição do tweak - [N||b||1]
    tweakAutenticate = modifyBit(tweakAutenticate, 0, 1)
    tweakAutenticate = tweakAutenticate.to_bytes(nrBytes, byteorder='big'

    return tweakBlock, tweakAutenticate
```

Funções auxiliares

A função generateMac será responsável por permitir que uma entidade verifique e confirme a autenticação dos dados agregados a uma mensagem, gerando um valor de Hash.

A função xor irá simplificar a operação XOR communmente conhecida, de modo a que possa ser aplicada a um bloco de bytes diretamente, em vez de ser feito no código o XOR dos bytes um a um.

In [ ]:
```
# Função responsável pela geração do valor de autenticação
def generateMac(hmac_key, package):

    h = hmac.HMAC(hmac_key, hashes.SHA256(), backend = default_backend())
    h.update(package)

    return h.finalize()


# Função responsável por realizar a operação Operação de XOR em bloco (nã
def xor(blockL, blockR):

    return [(a^b).to_bytes(1,byteorder='big') for (a,b) in zip(blockL, bl
```

Encrypt

Após a geração dos tweaks e dos blocos de mensagens, iremos então começar a cifrar os blocos.

Inicialmente ciframos todos os blocos, com a exceção do último (que poderá conter padding). Iremos percorrer cada bloco e utilizando a função generateCiphertext iremos cifrar o mesmo, utilizando ainda o tweak associada a cada bloco bem como a chave partilhada.

Seguidamente, iremos cifrar o último bloco. Primeiramente ciframos apenas o bloco da mensagem sem padding. Seguidamente, utilizaremos o texto cifrado obtido desse bloco, bem como a mensagem no ultimo bloco e iremos utilizar a função xor para obter o texto cifrado do ultimo bloco.

Finalmente, verificamos o comprimento do ultimo bloco, que nos irá fornecer a tag de autenticação. Para gerar esta tag, geramos primeiro o checksum (operação xor entre todos os blocos, inclusive o ultimo sem o padding). Seguidamente, geramos a tag, utilizando o generateCiphertext, onde queremos apenas obter o numero de bytes igual ao tamaho do ultimo bloco da mensagem.

In [ ]:
```
# Função que implementa a cifra a ser utilizada pelo Tweakable Block Ciph
def generateCiphertext(chave, nonce, tweak, message):

    #Criação da cifra AES
    cifrador = Cipher(algorithms.AES(chave), modes.CTR(nonce)).encryptor(

    #Cifra a mensagem utilizando a chave
    cifraObtida = cifrador.update(message)

    #Operação de xor entreo tweak e o output da cifra
    cifraComTewak = b"".join(xor(tweak,cifraObtida))

    #Retorna a cifra da operação xor
    return cifrador.update(cifraComTewak) + cifrador.finalize()



# Função responsável por cifrar o plainText
def encrypt(plainText, chave):

    # Divisão da mensagem em blocos
    lastBlockSize, message = criacaoBlocoMensagem(plainText)

    lengthMessage = len(plainText)
    numberBlocks  = len(message)

    # Geração dos tweaks
    nonceTweak = os.urandom(nrBytes//2)
    tweakBlock, tweakAutenticate = generateTweak(numberBlocks,lengthMessa

    lastBlockIndex = numberBlocks - 1
    cipherBlock = []

    nonce = os.urandom(16)

    # Cifragem de todos os blocs até ao penúltimo (inclusive)
    for w in range(0,lastBlockIndex) :

        textoCifrado = generateCiphertext(chave, nonce, tweakBlock[w], me
        cipherBlock.append(textoCifrado)

    # Cifragem do último bloco
    lastBlock = int(lastBlockSize).to_bytes(nrBytes, byteorder='big')
    textoCifradoUltimoBloco = generateCiphertext(chave, nonce, tweakBlock

    cipherLast = b"".join(xor(textoCifradoUltimoBloco, message[lastBlockI
    cipherBlock.append(cipherLast)


    # Atenticação da mensagem final
    checksum = message[0]

    # Geração do checksum
    for i in range(1, lastBlockIndex + 1):
        z2 = message[i]
        checksum = b"".join(xor(checksum,z2))

    tag = generateCiphertext(chave, nonce, tweakAutenticate, checksum)[:l

    #Junta todos os blocos cifrados
    ciphertext = b"".join(cipherBlock)

    # Mensagem a ser enviada para o recetor
```

Decrypt

Após a geração dos tweaks e dos blocos de mensagens, iremos então começar a cifrar os blocos, utilizando os valores recebidos nas mensagens.

Neste momento iremos primeiro decifrar todos os blocos com recurso à função generateCiphertext, dando como argumento o tweak associado a cada bloco e a chave partilhada das entidades. Iremos acabar com blocos de mensagens em plaintext com comprimento igual a 16 bytes.

Seguidamente iremos determinar a tag de autenticação. O processo será análogo ao do processo de cifragem das mensagens, onde geramos primeiro o checksum (operação xor entre todos os blocos, inclusive o ultimo, mas neste caso com o padding) e seguidamente, geramos a tag, utilizando o generateCiphertext, onde queremos apenas obter o numero de bytes igual ao tamaho do ultimo bloco da mensagem. Se a tag gerada for igual à recebida, conseguimos perceber que a mensagem é autêntica.

In [ ]:
```
# Função responsável por decifrar o cipherText
def decrypt(cipherPackage, keyCipher):

    # Retira os valores da mensagem
    ciphertext = cipherPackage['ciphertext']
    tag        = cipherPackage['tag']
    nonce      = cipherPackage['nonce']
    nonceTweak = cipherPackage['nonceTweak']

    # Criar array que dvide a mensagem cifrada em blocos de nrBytes, no n
    message = []
    for i in range(0, len(ciphertext), nrBytes):
        message.append(ciphertext[i : i + nrBytes])

    numberBlocks   = len(message)
    lastBlockIndex = numberBlocks-1
    lenMessage     = len(ciphertext)
    lenTag         = len(tag)

    length = lenMessage - (nrBytes - lenTag)

    #Cria os tweaks utilizando o nonce recbeido na mensagem cifrada
    tweakBlock, tweakAutenticate = generateTweak(numberBlocks, length, no

    plainTextBlock = []

    # Decifragem de todos os blocos até ao penúltimo
    for w in range(0, lastBlockIndex + 1) :

        ct = generateCiphertext(keyCipher, nonce, tweakBlock[w], message[
        plainTextBlock.append(ct)

    # Autenticação da mensagem
    checksum = plainTextBlock[0]

    # Geração do checksum
    for i in range(1, lastBlockIndex + 1):
        z2 = plainTextBlock[i]
        checksum = b"".join(xor(checksum,z2))

    # Gera tag de autenticação apartir do checksum
    generatedTag = generateCiphertext(keyCipher, nonce, tweakAutenticate,

    #Verifica se a mensagem está autenticada
    if tag == generatedTag:

        plainTextBlock[lastBlockIndex] = plainTextBlock[lastBlockIndex][:

        #Junta todas as mensagem decifradas
        plaintext = b"".join(plainTextBlock)

    else :
        return "R: Message not autenticated"


    return plaintext.decode('utf-8')
```

Emissor

O emissor começa por gera o par de chaves, assina a mensagem que contem a chave publica e envia para o recetor. Vai receber depois a chave publica do recetor e irá confirmar a ssinatura da mesma. Seguidamente, no caso de sucesso, geramos a chave partilhada e verificamos se a mesma é igual à que o Recetor possui. Este processo de verificação recorre àfunção generateMAC que irá gerar valores Hash para verificação de igualdade. Por fim, utilizamos a função encrypt para encriptar a mensagem e enviamos o resultado ao recetor.

In [ ]:
```
# Função responsável por implementar o emissor
def Emitter(conn):

    # Geração do par de chaves
    pacote, privateKey = geracaoKeys()

    # Geração a assinatura
    pacoteFinal = geracaoAssinatura(pacote)

    print("E: Sending public keys to receiver...")
    conn.send(pacoteFinal)

    msg = conn.recv()
    print("E: Receiving public keys from receiver...")

    # Retira a chave publica da assinatura da mensagem que recebeu
    peer_public_key = load_pem_public_key(msg['public_key'])

    try:
        # Verifica a assinatura da mensagem
        peer_public_key.verify(msg['assinatura'],msg['mensagem'])
        print("E: The message is authentic.")

        # Gera a chave partilhada
        pkg_msg = loads(msg['mensagem'])
        sharedKey = geracaoChavePartilhada(pkg_msg,privateKey)

        # Gera hash de autenticação para confirmar a chave partilhada
        hmac_key = generateMac(sharedKey,sharedKey)
        confirmKey = {'hmac': hmac_key}

        print("E: Sending confirmation for shared key ...")
        conn.send(dumps(confirmKey))

        hmacMessage = conn.recv()
        print("E: Receiving confirmation for shared key...")

        hmacReceiver = loads(hmacMessage)['hmac']

        # Confirma autenticação da achave partilhada
        if hmacReceiver == generateMac(sharedKey,sharedKey):

            print("E: Shared key is equal")

            text = "Esta sera a mensagem enviada para teste. Para verific

            print('Inicial message: ' + text)

            print("E: Encrypting message...")
            message = encrypt(text,sharedKey)

            print('Cipher message:')
            print(message)

            print("E: Sending ciphertext ...")
            conn.send(message)

        else:
            print('E: The shared keys are different.')

    except InvalidSignature:
        print("E: The receiver message is not authentic.")
```

Recetor

O recetor começa por gera o par de chaves. Vai receber depois a chave publica do emissor e irá confirmar a assinatura da mesma. Seguidamente irá enviar ao emissor a sua chave pública e espera pela receção da chave partilhada do Emissor. Irá verificar se a chave partilhada é igual á sua. Este processo de verificação recorre à função generateMAC que irá gerar valores Hash para verificação de igualdade. No caso de sucesso, envia o seu Hash gerado da sua chave partilhada para o Emissor. Por fim, o recetor recebe uma mensagem que irá verificar a autenticacidade através da comparação dos valores Hash. Se a mensagem for autêntica irá então decifrar a mensagem recebida utilizando a função decrypt.

In [ ]:
```
# Função responsável por implementar o Recetor
def Receiver(conn):

    # Geração do par de chaves
    pacote, privateKey = geracaoKeys()

    # Geração de assinatura
    pacoteFinal = geracaoAssinatura(pacote)

    msg = conn.recv()
    print("R: Receiving public keys from emitter...")

    peer_public_key = load_pem_public_key(msg['public_key'])

    try:
        # Verifica a assinatura da mensagem
        peer_public_key.verify(msg['assinatura'],msg['mensagem'])
        print("R: The message is authentic.")

        # Gera a chave partilhada
        pkg_msg = loads(msg['mensagem'])
        sharedKey = geracaoChavePartilhada(pkg_msg,privateKey)

        print("R: Sending public keys to emitter...")
        conn.send(pacoteFinal)

        hmacMessage = conn.recv()
        print("R: Receiving confirmation for shared key...")

        hmacEmitter = loads(hmacMessage)['hmac']

        # Confirma se a autenticação da chave partilhada
        if hmacEmitter == generateMac(sharedKey,sharedKey):

            print("R: Shared key is equal")

            #Gera hash de autenticação para confirmar a chave partilhada
            hmac_key = generateMac(sharedKey,sharedKey)
            confirmKey = {'hmac': hmac_key}

            print("R: Sending confirmation for shared key ...")
            conn.send(dumps(confirmKey))

            message = conn.recv()

            ciphertext =  message['pkg']
            hmac =  message['hmac']

            # Confirma autenticação da mensagem
            if hmac == generateMac(sharedKey,ciphertext['ciphertext']):

                print("R: Decrypting message...")
                final_message = decrypt(ciphertext, sharedKey)

                print('Final Message: ' + final_message)
            else :
                print("R: Message not autenticated")

        else:
            print('ERROR - Different keys used.')

    except InvalidSignature:
        print("R: The message is not authentic.")
```

Comunicação

No pedaço de código seguinte, podemos observar a inicialização de uma classe que permite executar o emissor e o recetor em simultâneo e em processos diferentes, permitindo simular a comunicação entre duas entidades diferentes.

In [ ]:
```
from multiprocessing import Process, Pipe

class Conn(object):

    def __init__(self, emitter, receiver):

        emitter_end, receiver_end = Pipe()

        self.eproc = Process(target=Emitter, args=(emitter_end,))
        self.rproc = Process(target=Receiver, args=(receiver_end,))

    def auto(self):
        self.eproc.start()
        self.rproc.start()
        self.eproc.join(None)
        self.rproc.join(None)

Conn = Conn(Emitter, Receiver)
Conn.auto()
```

```
E: Sending public keys to receiver...
R: Receiving public keys from emitter...
R: The message is authentic.
R: Sending public keys to emitter...
E: Receiving public keys from receiver...
E: The message is authentic.
E: Sending confirmation for shared key ...
R: Receiving confirmation for shared key...
R: Shared key is equal
R: Sending confirmation for shared key ...
E: Receiving confirmation for shared key...
E: Shared key is equal
Inicial message: Esta sera a mensagem enviada para teste. Para verificar
que correu tudo bem tera de ser igual no final!
E: Encrypting message...
Cipher message:
{'pkg': {'ciphertext': b'\xf5q\xa7=L\xc1*\x91\xd2l\x86\'/\xb2\xb0\x88\xd1
e\xb61L\xd7!\x95\xda-\x83fb\xa7\xbf\x89\xd1"\xa79\x1f\xc6*\xcd\x93\x1c\x8
6u#\xf7\xa8\x9c\xc2k\xb55\x0f\xd3=\xc3\xc29\x82\'!\xb8\xac\x8b\xd5w\xf3(\
x19\xd6 \xc3\xd1)\x8a\'6\xb2\xac\x9e\x90f\xb6|\x1f\xd7=\xc3\xda+\x92f.\xf
7\xb0\x90\x90d\xba2\r\xden\xe3\xb3L\xe7\x07B\xd7\xde\xfa', 'tag': b'\xe2
(\xfcf\x1b\xdco', 'nonce': b'\xeb\x8e\xbc\xe7\x7fj\xcb\xac=\xb1}\tP\xbc\x
bb\xe7', 'nonceTweak': b':<t\xf5\xc1\x8c>V'}, 'hmac': b'P9E>9\xd5\r\x8d:\
xcc\xc6\x82\xa9H Q#-\xd3Y\xce\x19I\xddD$6\xfc|L\xa2M'}
E: Sending ciphertext ...
R: Decrypting message...
Final Message: Esta sera a mensagem enviada para teste. Para verificar qu
e correu tudo bem tera de ser igual no final!
```