Trabalho prático 0 - Estruturas Criptográficas

Autores: Ariana Lousada (PG47034), Cláudio Moreira (PG47844)

Grupo 12

In [ ]:
```
import multiprocessing
import os
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives import hmac
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.asymmetric import dsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives.serialization import load_pem_public_key
from cryptography.hazmat.primitives import serialization


listanouce = []

salt = os.urandom(16) # Salt partilhado
metadados = os.urandom(16)

def kdf(password, salt):
    # PBKDF2 algoritmo tipicamente usado para obter uma chave a partir de uma password,
    # tamanho da chave (32 bytes)
    kdf = PBKDF2HMAC(algorithm=hashes.SHA256(),length=32,salt=salt,iterations=100000)

    key = kdf.derive(password.encode('utf8')) # deriva a chave

    # verificação de se a password fornecida pelo user corresponde à chave derivada armazenada
    kdf = PBKDF2HMAC(algorithm=hashes.SHA256(),length=32,salt=salt,iterations=100000)
    kdf.verify(password.encode('utf8'),key)
    return key
```

Problema 1

Pergunta a)

Para ser possível criar uma comunicação entre duas entidades distintas, com segurança contra ataques aos nounces, recorreu-se à cifra simétrica AES para desenvolver as funções cifragem e decifragem. De modo a gerar nonces aleatórios, foi utilizada a função nounceGeneratorSHAKE(que será abordada na alínea b deste exercício).

Também se desenvolveram as funções mac e mac_verify, para serem posteriormente implementadas no protocolo DH com assinaturas DSA. Este par de funções garante a autenticidade na partilha de chaves entre o Emitter e o Receiver.

A função mac cria uma tag de autenticação através da password e da chave derivada, recorrendo à função hash hmac.

A função mac_verify verifica a autenticidade.

In [ ]:
```
# função que cifra mensagens
def cifragem(texto, metadados, key):
    texto = texto.encode('utf8') # conversão do texto limpo para bytes

    aesgcm = AESGCM(key)

    nonce =  nounceGeneratorSHAKE(kdf(password, salt, 12),12)
    ciphered_text = aesgcm.encrypt(nonce, texto, metadados)

    # concatenação do nonce ao ciphered text
    ciphered_text += nonce
    return ciphered_text

# função que decifra mensagens
def decifragem(ciphered_text, metadados, key):
    aesgcm = AESGCM(key)
    # atribuir os 12 últimos bytes do ciphered text ao nonce
    nonce = ciphered_text[-12:]

    # retirar os 12 últimos bytes
    ciphered_text = ciphered_text[:-12]

    # decifragem utilizando GCM
    clean_text = aesgcm.decrypt(nonce, ciphered_text, metadados)

    return clean_text

def mac(key, ciphered_text):
    h = hmac.HMAC(key, hashes.SHA256())
    h.update(ciphered_text)
    tag = h.finalize()
    return tag

def mac_verify(key, ciphered_text, tag):
    h = hmac.HMAC(key, hashes.SHA256())
    h.update(ciphered_text)
    h.verify(tag)
```

Pergunta b)

De modo a criar um gerador pseudo-aleatório com uma função do tipo XOF(Extendable Output Function) escolheu-se o SHAKE256 para ser possível criar uma sequência de 2n palavras aleatórias de 8 bytes cada.

In [ ]:
```
def nounceGeneratorSHAKE(seed, n):
    i = 0
    nounce = []
    digest = hashes.Hash(hashes.SHAKE256((2**n) * 8)) # calcula uma string com tamanho de 2^n *8 bytes (64 bits)
    digest.update(seed) # bytes a ser hashed (seed)
    p = digest.finalize()
    while i < (2**n): # dividir a mensagem em blocos de 8 bytes
        nounce.append(p[:8])
        p = p[8:]
        i += 1
    return nounce
```

Pergunta c)

Para desenvolver uma solução para esta pergunta, desenvolveram-se cinco funções distintas.

A primeira função, geraChavesDH gera as chaves pública e privada DH para os dois agentes pertencentes à comunicação. A segunda função, geraChavesDSA gera as chaves pública e privada DSA para os dois agentes. A terceira função verificacaoAssinatura verifica se a assinatura é válida a partir das chaves públicas de cada agente. A quarta função derivacaoChave é responsável pela criação da chave partilhada entre os agentes assim como a sua derivação.

Por fim, a última função DHProtocol_DSA define o protocolo de troca de chaves e de autenticação de assinaturas. O principal objetivo consiste na transferência de chaves e da assinatura entre as entidades na comunicação, assim como a verificação e derivação da chave partilhada. Esta função tira também partido das funções de aplicação hmac já anteriormente desenvolvidas, de modo a garantir a autenticação.

In [ ]:
```
# geração dos parâmteros DH
parameters = dh.generate_parameters(generator=2, key_size=2048)

# geração das chaves pública e privada DH
def geraChavesDH():
    # geração da chave privada DH
    private_keyDH = parameters.generate_private_key()

    # geração da chave pública DH e passagem para bytes
    public_keyDH = private_keyDH.public_key().public_bytes(encoding=serialization.Encoding.PEM,
                                                           format=serialization.PublicFormat.SubjectPublicKeyInfo)

    return private_keyDH, public_keyDH

# geração das chaves pública e privada DSA
def geraChavesDSA():
    # geração da chave privada DSA
    private_keyDSA = dsa.generate_private_key(key_size=1024)
    # geração da chave pública DSA e passagem para bytes
    public_KeyDSA = private_keyDSA.public_key().public_bytes(encoding=serialization.Encoding.PEM,
                                                             format=serialization.PublicFormat.SubjectPublicKeyInfo)
    return private_keyDSA, public_KeyDSA

# verificação da assinatura
def verificacaoAssinatura(assinatura, public_keyDH, public_keyDSA, nome):
    try:
        public_keyDSA.verify(assinatura, public_keyDH, hashes.SHA256())
        print(nome, "Signature verified. \n")
    except Exception as err:
        print("Error: " + str(err))

# criação da chave partilhada e respetiva derivação
def derivacaoChave(private_keyDH, public_keyDH):
    shared_key = private_keyDH.exchange(public_keyDH)
    derived_key = HKDF(algorithm=hashes.SHA256(),length=32,salt=salt,
                       info=b'handshake data').derive(shared_key)

    return derived_key


def mac(key, ciphered_text):
    h = hmac.HMAC(key, hashes.SHA256())
    h.update(ciphered_text)
    tag = h.finalize()
    return tag

def mac_verify(key, ciphered_text, tag):
    h = hmac.HMAC(key, hashes.SHA256())
    h.update(ciphered_text)
    h.verify(tag)

# protocolo de troca de chaves e autenticação
def DHProtocol_DSA(conn, password, nome):
    # criação das chaves
    private_keyDH, public_keyDH = geraChavesDH()
    private_keyDSA, public_keyDSA = geraChavesDSA()

    # assinatura
    signature = private_keyDSA.sign(public_keyDH, hashes.SHA256())

    # envio da informação (chaves + assinatura) para o outro agente
    info = [public_keyDH, public_keyDSA, signature]
    conn.send(info)

    # receção da informação do agente
    info = conn.recv()
    receiverPK_DH_Bytes = info[0]
    receiverPK_DSA_Bytes = info[1]
    receiverSign = info[2]

    # descerialização das chaves
    receiverPK_DH = load_pem_public_key(receiverPK_DH_Bytes)
    receiverPK_DSA = load_pem_public_key(receiverPK_DSA_Bytes)

    # verificação da assinatura
    verificacaoAssinatura(receiverSign, receiverPK_DH_Bytes, receiverPK_DSA,nome)

    # derivação de chaves
    derived_key = derivacaoChave(private_keyDH, receiverPK_DH)

    # Autenticação HMAC
    tag = mac(password, derived_key)
    conn.send(tag)
    tagRecebida = conn.recv()
    mac_verify(password, derived_key, tagRecebida)
    print(nome, "Finished.\n")

    return derived_key
```

Comunicação entre Emitter e Receiver

Para inicializar a comunicação, é criado um Pipe de modo a possibilitar a comunicação entre Emmiter e Receiver. Em primeiro lugar é necessária a inserção da password da parte do Receiver, Emmiter e a mensagem a ser enviada. Estas passwords são posteriormente derivadas utilizando um kdf. Em segundo lugar, são inicializados os protocolos de acordo de chaves e respetiva autenticação, utilizando a função anteriormente desenvolvida de implementação do protocolo DH-DSA (DHProtocol_DSA). Caso as passwords sejam iguais e não ocorram erros de autenticação nem de verificação de assinaturas, a chave é partilhada entre os dois agentes. Por fim, a mensagem inserida é enviada do Emmiter(onde é cifrada antes do envio) para o Receiver(onde é decifrada após a receção). Os processos de cifragem e decifragem são feitos recorrendo às funções desenvolvidas na alínea a).

In [ ]:
```
metadados = os.urandom(16)
# emissor da mensage
def emitter(conn, msgs, chave):
    shared_key = DHProtocol_DSA(conn, chave, "[Emitter]")

    for msg in msgs:
        ciphered_text = cifragem(msg, metadados, shared_key)
        print("[Emitter] Message sent.\n")
        conn.send(ciphered_text)

    conn.close()

# receptor da mensagem
def receiver(conn, chave2):
    shared_key = DHProtocol_DSA(conn, chave2, "[Receiver]")
    try:
        ciphered_text = conn.recv()
        clean_text = decifragem(ciphered_text, metadados, shared_key)
        print("[Receiver] Received message: " + clean_text.decode('utf8') + "\n")

    except Exception as err:
        print("Error: " + str(err))
        return 1


#estabelecer comunição entre receiver e emitter
def main1(passEm, passRc, msgs):
    salt = os.urandom(16) # salt comum a ambos os agentes
    chave = kdf(passEm, salt) # derivação da pass do emitter

    # criação dos pipes
    parent_conn, child_conn = multiprocessing.Pipe()
    p1 = multiprocessing.Process(target=emitter, args=(parent_conn, msgs,chave)) # envio da mensagem

    chave2 = kdf(passRc, salt) # derivação da pass do receiver
    p2 = multiprocessing.Process(target=receiver, args=(child_conn, chave2)) #recebe a mensagem

    # processos a correr
    p1.start()
    p2.start()

    # espera que ambos os processos terminem
    p1.join()
    p2.join()

if __name__ == '__main__':
    passEm = input("Password (Emitter): ")
    msgs = [input("Message:")] # escreve a mensagem que pretende enviar
    passRc = input("Password (Receiver): ")
    main1(passEm, passRc, msgs)
```

Problemas de Implementação

Devido a um erro de derivação de chaves, nomeadamente na função derivacaoChave utilizada na função de aplicação do protocolo Diffie-Hellman, não foi possível a elaboração de testes relativos ao primeiro problema. Este erro pode ter como causa algum tipo de formato num dos argumentos utilizados pela função. Contudo, a equipa de trabalho viu-se incapaz de corrigir o problema.