Trabalho Prático 0

Grupo 17, constituído por:

-- Joana Castro e Sousa, PG47282

-- Tiago Taveira Gomes, PG47702

Pergunta 3:

Compare experimentalmente a eficiência dos dois esquemas de cifra.

In [ ]:
```
#imports
import os
import time

from cryptography.exceptions import *
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives.asymmetric import dh,dsa
from cryptography.hazmat.primitives import padding, hashes, hmac, serialization
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
```

O primeiro esquema de cifra implementado foi:

In [ ]:
```
print('Gerar os parâmetros para o DH.')
parameters_dh = dh.generate_parameters(generator=2, key_size=1024,backend=default_backend())
print('Parâmetros criados!')
print('')
print('Gerar os parâmetros para as assinaturas DSA.')
parameters_dsa = dsa.generate_parameters(key_size=1024,backend=default_backend())
print('Parâmetros criados!')

class DiffieHellman:
    def generate_DH_PrivateKey(self):
        private_key = parameters_dh.generate_private_key()
        return private_key

    def generate_DH_PublicKey(self, private_key):
        public_key = private_key.public_key()
        return public_key

    def generate_DH_PublicBytes(self, public_key):
        return public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo)

class DSASignatures:
    def generate_DSA_PrivateKey(self):
        private_key = parameters_dsa.generate_private_key()
```

```
        return private_key

    def generate_DSA_PublicKey(self,private_key):
        public_key = private_key.public_key()
        return public_key

    def generate_DSA_PublicBytes(self, public_key):
        return public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo)

    def sign_message(self, message,own_private_key):
        signature = own_private_key.sign(
            message,
            hashes.SHA256()
        )
        return signature

    def verify_Signature(self, message, signature, other_public_key):
        other_public_key.verify(
            signature,
            message,
            hashes.SHA256()
        )

dsaSig = DSASignatures()

emitter_dsa_privateKey = dsaSig.generate_DSA_PrivateKey()
emitter_dsa_publicKey = dsaSig.generate_DSA_PublicKey(emitter_dsa_privateKey)

receiver_dsa_privateKey = dsaSig.generate_DSA_PrivateKey()
receiver_dsa_publicKey = dsaSig.generate_DSA_PublicKey(receiver_dsa_privateKey)
```

```
Gerar os parâmetros para o DH.
Parâmetros criados!

Gerar os parâmetros para as assinaturas DSA.
Parâmetros criados!
```

In [ ]:

```
class Encription:
    def kdf(self, password, mySalt=None):
        if mySalt is None:
            auxSalt = os.urandom(16)
        else:
            auxSalt = mySalt
        kdf = PBKDF2HMAC(
            algorithm = hashes.SHA256(),   # SHA256
            length=32,
            salt=auxSalt,
            iterations=100000,
            backend=default_backend()        # openssl
        )
        key = kdf.derive(password)
        if mySalt is None:
            return auxSalt, key
        else:
            return key
```

```
    # a função de Hash que calcula a hash de um dado input. resultado é o "nounce", construído em XOF (com SHA256)
    def Hash(self, s):
        digest = hashes.Hash(hashes.SHA256(),backend=default_backend())
        digest.update(s)
        return digest.finalize()

    def mac(self, key, msg, tag=None):
        h = hmac.HMAC(key,hashes.SHA256(),default_backend())
        h.update(msg)
        if tag is None:
            return h.finalize()
        h.verify(tag)

    def encript(self, Ckey, Hkey, msg):
        iv = os.urandom(16)
        cipher = Cipher(algorithms.AES(Ckey), modes.CTR(iv), default_backend())
        encryptor = cipher.encryptor()
        ciphertext = encryptor.update(msg) + encryptor.finalize()
        tag = self.mac(Hkey,ciphertext)
        return iv, ciphertext, tag

    def decript(self, Ckey, iv, msg):
            cipher = Cipher(algorithms.AES(Ckey), modes.CTR(iv), default_backend())
            decryptor = cipher.decryptor()
            cleant = decryptor.update(msg) + decryptor.finalize()
            return cleant
```

In [ ]:

```
def teste_cifra1():
    print("Calcular o tempo de execução da cifra 1:\n")

    #TODO

    #Acaba de contar o tempo
    stop = time.perf_counter()
    delta_time = stop - start
    print("Tempo de execução: %f " %delta_time)

if __name__ == "__main__":
    teste_cifra1()
```

```
Calcular o tempo de execução da cifra 1:
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/var/folders/wp/c5y2_nk93cx1zfwkj4932t500000gn/T/ipykernel_15800/1355562597.py in <cell line: 11>()
     10
     11 if __name__ == "__main__":
---> 12     teste_cifra1()

/var/folders/wp/c5y2_nk93cx1zfwkj4932t500000gn/T/ipykernel_15800/1355562597.py in teste_cifra1()
      6     #Acaba de contar o tempo
      7     stop = time.perf_counter()
----> 8     delta_time = stop - start
      9     print("Tempo de execução: %f " %delta_time)
     10

NameError: name 'start' is not defined
```

O segundo esquema de cifra foi:

In [ ]:
```
N = 5
BLOCK_SIZE = 8

def derivate_key(password, salt):
    # derivar
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
        )
    key = kdf.derive(password)
    return key

def prg(seed):
    digest = hashes.Hash(hashes.SHAKE256(BLOCK_SIZE * pow(2,N)))
    digest.update(seed)
    words = digest.finalize()
    return words

def decode(key, ct):
    pt = b''
    # Divide texto cifrado em blocos de 8 bytes
    p = [ct[i:i+BLOCK_SIZE] for i in range(0, len(ct), BLOCK_SIZE)]
    # XOR dos bytes do bloco do texto cifrado com os bytes do bloco de palavras chave
    for x in range (len(p)): # Percorre blocos do texto cifrado
        for index, byte in enumerate(p[x]): # Percorre bytes do bloco do texto cifrado
            pt += bytes([byte ^ key[x:(x+1)*BLOCK_SIZE][index]])
    # Algoritmo para retirar padding para decifragem
    unpadder = padding.PKCS7(64).unpadder()
    # Retira bytes adicionados
    unpadded = unpadder.update(pt) + unpadder.finalize()
    return unpadded.decode("utf-8")

def encode(key,message):
    ct = b''
    padder = padding.PKCS7(64).padder()
    # Adiciona padding ao último bloco de bytes da mensagem de modo a esta ter tamanho múltiplo do bloco
    padded = padder.update(message) + padder.finalize()
    # Divide mensagem em blocos de 8 bytes
    p = [padded[i:i+BLOCK_SIZE] for i in range(0, len(padded), BLOCK_SIZE)]
    # XOR dos bytes do bloco da mensagem com os bytes do bloco de palavras chave
    for x in range (len(p)): # Percorre blocos do texto limpo
        for index, byte in enumerate(p[x]): # Percorre bytes do bloco do texto limpo
            ct += bytes([byte ^ key[x:(x+1)*BLOCK_SIZE][index]])
    return ct
```

In [ ]:
```
def teste_cifra2():
    print("Calcular o tempo de execução da cifra 2:\n")
    #Começa a contar o tempo
    start = time.perf_counter()
    # a password que queremos ter partilhada
    password = "uma password"
    # a salt necessária para derivar a chave
    salt = os.urandom(16)
```

```
    # gerar a 'seed'
    seed = derivate_key(password.encode("utf-8"), salt)
    # assim é possível gerar a chave com essa seed
    key = prg(seed)
    # e o cypher_text
    ct = encode(key, "Segredo".encode("utf-8"))
    dt = decode(key, ct)
    #Acaba de contar o tempo
    stop = time.perf_counter()
    delta_time = stop - start
    print("Tempo de execução: %f " %delta_time)

if __name__ == "__main__":
    teste_cifra2()
```

```
Calcular o tempo de execução da cifra 2:

Tempo de execução: 0.088516
```

Conclusões

O primeiro algoritmo de cifra simétrica no modo AESCTR, realiza juntamente com a cifragem, autenticação de texto e entre agentes. Através destas autenticações entre agentes, utilizando o protocolo de DH usando assinaturas DSA, existe uma partilha de chaves secretas que pode ser usado para troca de mensagens secretas dentro de um canal de comunicação público. Assim para sistemas mais reais esta cifra da parte 1 proporciona confidencialidade, integridade e autenticidade bem como, à partida, não é preciso saber o tamanho das mensagens nem o seu número.

Já na segunda cifra implementada, segue um padrão de cifra de Vernam. Esta oferece apenas confidencialidade e não promove autenticidade nem integridade e restringe o tamanho e o número das mensagens. Esta cifra comporta-se como uma cifra sequencial e estas tendem a ser muito eficientes.