tp0_parte2

March 15, 2021

1 TP0 - Estruturas Criptográficas

1.1 Elementos do grupo 4

- André Morais, A83899
- Tiago Magalhães, A84485

1.2 PARTE 2

1.2.1 cifra.py

Definir variáveis globais e respetivos Imports

[9]:
```
import os
import random
import sys
import time

from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.hazmat.primitives import padding

N = 1
BLOCK_SIZE = 8 # 64 bits (8 bytes)
```

Derivação da chave a partir de uma password usando um *KDF*

[10]:
```
def derivate_key(password, salt):
    # derive
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
        )
    key = kdf.derive(password)
    return key
```

Cria um gerador pseudo-aleatório do tipo XOF usando o SHAKE256, gerando uma lista de palavras de 64 bits

[11]:
```
def prg(seed):
    digest = hashes.Hash(hashes.SHAKE256(BLOCK_SIZE * pow(2,N)))
    digest.update(seed)
    words = digest.finalize()
    return words
```

Cifra a mensagem através de uma cifra por blocos

[21]:
```
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
```

Decifra a mensagem

[20]:
```
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")
```

Teste para a cifra implementada

[19]:
```
def main():
    password = input("Insira a sua password: ")
    salt = os.urandom(16)
    seed = derivate_key(password.encode("utf-8"), salt)
    key = prg(seed)
```

```
    ct = encode(key, "Segredo".encode("utf-8"))
    print("Texto limpo: " + decode(key, ct))

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

```
Insira a sua password:  roo

Texto limpo: Segredo
```

1.3 Comparação de resultados

Para lermos os tempos de maneira análoga, adptamos as classes emitter e receiver de maneira a usar diferentes cifras. Isto pode ser observado através da variável **module_value**

[4]:
```
import os
import crypto
import cifra

class Emitter:
    def __init__(self, password, module):
        if module == 1:
            module_value = "crypto"
        else:
            module_value = "cifra"
        self.module = module_value
        self.key = None
        self.password = password
        self.key_salt = os.urandom(16)

    def derivate_key(self):
        # Parte 1
        if self.module == "crypto":
            key = crypto.derivate_key(self.password.encode('utf-8'), self.
 ↪key_salt)
        # Parte 2
        elif self.module == "cifra":
            seed = cifra.derivate_key(self.password.encode('utf-8'), self.
 ↪key_salt)
            key = cifra.prg(seed)
        self.key = key

    def send_message(self, message):
        key_digest = crypto.authenticate_HMAC(self.key, self.key)
        # Parte 1
        if self.module == "crypto":
            aad = None # Retirou-se só para a comparação ser mais justa
            nonce, ct = crypto.encode(message.encode('utf-8'), aad, self.key)
```

```
        # Parte 2
        elif self.module == "cifra":
            nonce = b''
            ct = cifra.encode(self.key, message.encode("utf-8"))
        return key_digest + nonce + self.key_salt + ct
```

[5]:
```
import crypto
import cifra

KEY_DIGEST_LEN = 32 # Bytes
NONCE_LEN = 12 # Bytes
SALT_LEN = 16 # Bytes

class Receiver:
    def __init__(self, password,module):
        if module == 1:
            module_value = "crypto"
        elif module == 2:
            module_value = "cifra"
        self.module = module_value
        self.key = None
        self.password = password

    def unpack_data(self, dados):
        key_digest = dados[:KEY_DIGEST_LEN] # primeiros 32 bytes ->␣
↪hash(keyEmitter)
        # Parte 1
        if self.module == "crypto":
            nonce = dados[KEY_DIGEST_LEN:KEY_DIGEST_LEN + NONCE_LEN]
            salt = dados[KEY_DIGEST_LEN + NONCE_LEN:KEY_DIGEST_LEN + NONCE_LEN␣
↪+ SALT_LEN]
            ct = dados[KEY_DIGEST_LEN + NONCE_LEN + SALT_LEN:]
        # Parte 2
        elif self.module == "cifra":
            salt = dados[KEY_DIGEST_LEN:KEY_DIGEST_LEN + SALT_LEN]
            ct = dados[KEY_DIGEST_LEN + SALT_LEN:]
            nonce = None
        return key_digest, ct, salt, nonce

    def derivate_key(self, dados):
        salt = self.unpack_data(dados)[-2]
        # Parte 1
        if self.module == "crypto":
            key = crypto.derivate_key(self.password.encode('utf-8'), salt)
        # Parte 2
        elif self.module == "cifra":
            seed = cifra.derivate_key(self.password.encode('utf-8'), salt)
```

```
            key = cifra.prg(seed)
        self.key = key

    def show_results(self, error, message):
        if error == None:
            print("Texto decifrado: %s" %message)
        elif error == 1:
            print("Falha na verificação da autenticidade.")

    def read_message(self, ct):
        key_digest, ct, salt, nonce = self.unpack_data(ct)
        # Autentica chave
        isValid = crypto.verify_HMAC(self.key, self.key, key_digest)
        if isValid == False:
            raise Exception("Falha na autenticidade da chave")
        # Parte 1
        if self.module == "crypto":
            aad = None # Para a comparação ser mais justa
            error_code, texto_limpo = crypto.decode(ct, nonce, aad, self.key)
        # Parte 2
        elif self.module == "cifra":
            texto_limpo = cifra.decode(self.key, ct)
            error_code = None
        self.show_results(error_code, texto_limpo)
```

Após isto, utilizamos o time o módulo **time** para medir os tempos usando a comunicação entre os agentes, definida na parte anterior do trabalho

[16]:
```
import os
import time
from emitter import Emitter
from receiver import Receiver

def read_input():
    password = input("Insira a sua password: ")
    return password

def main():
    print("################################### Algoritmo 1␣
 ↪########################################\n")
    print("Calcular o tempo de execução do algoritmo da parte 1\n")
    password = read_input()
    #Começa a contar o tempo
    start = time.perf_counter()
    emmiter = Emitter(password,1)
    emmiter.derivate_key()
    dados = emmiter.send_message("Segredo 1")
```

```
    #Acaba de contar o tempo
    stop = time.perf_counter()
    delta_time_1 = stop - start

    password = read_input()
    #Começa a contar o tempo
    start = time.perf_counter()
    receiver = Receiver(password,1)
    receiver.derivate_key(dados)
    try:
        receiver.read_message(dados)
    except:
        print("Falha na autenticação da chave")
    #Acaba de contar o tempo
    stop = time.perf_counter()
    #Soma dos tempos
    delta_time_2 = stop - start
    total_time = delta_time_1 + delta_time_2
    print("Tempo de execução: %f " %total_time)

    print("################################## Algoritmo 2␣
↪########################################\n")
    print("Calcular o tempo de execução do algoritmo da parte 2 \n")
    password = read_input()
    #Começa a contar o tempo
    start = time.perf_counter()
    emmiter = Emitter(password,2)
    emmiter.derivate_key()
    dados = emmiter.send_message("Segredo 1")
    #Acaba de contar o tempo
    stop = time.perf_counter()
    delta_time_1 = stop - start

    password = read_input()
    #Começa a contar o tempo
    start = time.perf_counter()
    receiver = Receiver(password,2)
    receiver.derivate_key(dados)
    try:
        receiver.read_message(dados)
    except:
        print("Falha na autenticação da chave")
    #Acaba de contar o tempo
    stop = time.perf_counter()
    #Soma dos tempos
    delta_time_2 = stop - start
    total_time = delta_time_1 + delta_time_2
```

```
    print("Tempo de execução: %f " %total_time)

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

```
################################## Algoritmo 1
########################################

Calcular o tempo de execução do algoritmo da parte 1


Insira a sua password:  tiago
Insira a sua password:  tiago

Texto decifrado: Segredo 1
Tempo de execução: 0.080389
################################## Algoritmo 2
########################################

Calcular o tempo de execução do algoritmo da parte 2


Insira a sua password:  tiago
Insira a sua password:  tiago

Texto decifrado: Segredo 1
Tempo de execução: 0.077808
```

2 Conclusão

Como podemos observar, os tempos medidos são sempre muito similares. O algoritmo de cifra simétrica no modo AESGCM, realiza juntamente com a cifragem, autenticação de texto. Já a cifra de Vernam oferece apenas confidencialidade. Assim para sistemas mais reais a 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.

A cifra da parte 2 não promove autenticidade nem integridade e restringe o tamanho e o número das mensagens. Esta cifra não é muito menos eficiente que a outra porque se comporta como uma cifra sequencial e estas tendem a ser muito eficientes.