tp0_parte1

March 15, 2021

1 TP0 - Estruturas Criptográficas

1.1 Elementos do grupo 4

- André Morais, A83899
- Tiago Magalhães, A84485

1.2 PARTE 1

1.2.1 crypto.py

Primeiramente foi criado um módulo onde se encontram as primitivas criptográficas a serem utilizadas na comunicação entre os agentes

[1]:
```
import os
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes, hmac
import cryptography.exceptions

NONCE_LEN = 12 # Bytes
```

Recebe uma password e utiliza um KDF(PBKDF), que é tipicamente usado para derivar uma chave a partir de uma password.

Args: + password (bytes): Password + salt (bytes): Salt

Returns: + bytes: Chave derivada.

[11]:
```
    def derivate_key(password, salt):
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,
            salt=salt,
            iterations=100000,
            )
        key = kdf.derive(password)
        return key
```

Gera digest para uma mensagem.

Args: + key (bytes): Chave usada pelo hmac (é recomendado que o seu tamanho seja igual ao comprimento do digest) + message (bytes): Mensagem

Returns: + bytes: digest

[13]:
```
    def authenticate_HMAC(key, message):
        h = hmac.HMAC(key, hashes.SHA256())
        h.update(message)
        digest = h.finalize()
        return digest
```

Valida digest.

Args: + key (bytes): Chave usada pelo hmac + message (bytes): Mensagem + signature (bytes): Bytes que irão ser usados para comparar com o digest

Returns: + bool: True se for válido, false se não for

[18]:
```
    def verify_HMAC(key, message, signature) :
        h = hmac.HMAC(key, hashes.SHA256())
        # gera digest para a mensagem
        h.update(message)
        try :
            # verifica se o digest gerado acima é igual ao digest recebido como␣
↪parâmetro
            h.verify(signature)
            return True
        except cryptography.exceptions.InvalidSignature:
            return False
```

Cifra mensagem.

Args: + message (bytes): Mensagem a cifrar + aad (bytes): Metadados que irão ser autenticados, mas não cifrados + key (bytes): Chave

Returns: + tuplo: [nonce (bytes), texto cifrado (bytes)].

[4]:
```
    def encode(message, aad, key):
        nonce = os.urandom(NONCE_LEN) # NIST recommends a 96-bit length
        aad += nonce
        aesgcm = AESGCM(key)
        ct = aesgcm.encrypt(nonce, message, aad)
        return nonce, ct
```

Decifra mensagem.

Args: + ciphertext (bytes): Texto cifrado + key (bytes): Chave

Returns: + tuplo: [código de erro (int), texto limpo(str)].

[20]:
```
    def decode(ciphertext, nonce, aad, key):
        aesgcm = AESGCM(key)
        try:
            texto_limpo = aesgcm.decrypt(nonce, ciphertext, aad)
        except cryptography.exceptions.InvalidTag:
            # Falha na verificação da autenticidade
            return 1, None
        return None, texto_limpo.decode('utf-8')
```

1.2.2 emitter.py

Criação da classe Emitter que tem como função cifrar mensagem e construir o criptograma

[30]:
```
import os
import crypto

class Emitter:
    def __init__(self, password):
        self.key = None
        self.password = password
        self.key_salt = os.urandom(16)
```

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

[4]:
```
    def derivate_key(self):
        key = crypto.derivate_key(self.password.encode('utf-8'), self.key_salt)
        self.key = key
```

Função que irá cifrar e autenticar o texto limpo, bem como autenticar a chave e os metadados(digest da chave, nonce e salt), uma vez que estes são públicos e não precisam de ser cifrados

[1]:
```
    def send_message(self, message):
        key_digest = crypto.authenticate_HMAC(self.key, self.key)
        aad = self.key_salt
        nonce, ct = crypto.encode(message.encode('utf-8'), aad, self.key)
        return key_digest + nonce + self.key_salt + ct
```

1.2.3 receiver.py

Criação da classe Receiver que tem como função decifrar o criptograma e construir a mensagem

[2]:
```
import crypto

KEY_DIGEST_LEN = 32 # Bytes
NONCE_LEN = 12 # Bytes
```

```
SALT_LEN = 16 # Bytes

class Receiver:
    def __init__(self, password):
        self.key = None
        self.password = password
```

Função que decompõe os dados enviados pelo emitter em várias componentes

[3]:
```
    def unpack_data(self, dados):
        # dados : key_digest + nonce + salt + mensagem
        # 0 - 31 : key_digest (32 bytes)
        # 32 - 43 : nonce para decode (12 bytes)
        # 44 - 59 : salt para derivar chave (16 bytes)
        # 60 ... : texto cifrado
        key_digest = dados[:KEY_DIGEST_LEN]  # primeiros 32 bytes ->␣
↪hash(keyEmitter)
        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:]

        return key_digest, ct, salt, nonce
```

Função que deriva uma chave a partir de uma password

[6]:
```
    def derivate_key(self, dados):
        salt = self.unpack_data(dados)[-2]
        key = crypto.derivate_key(self.password.encode('utf-8'), salt)
        self.key = key
```

Mostra o resultado da decifragem. Em caso de erro, significa que existiu uma falha na autenticidade

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

Função que primeiramente irá autenticar a chave, através de um MAC, com o propósito de evitar uma decifragem de uma mensagem longa quando as chaves são diferentes, e depois decifrar a mensagem

[24]:
```
    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")
        aad = salt + nonce
        error_code, texto_limpo = crypto.decode(ct, nonce, aad, self.key)
        self.show_results(error_code, texto_limpo)
```

1.2.4 main.py

Simulação de uma comunicação assíncrona entre dois agentes

Emitter: 1. Escrever a password 2. Faz uma derivação da chave usando um *KDF* 3. Ciframos a mensagem com esta chave criada e envia dados para o Receiver

Receiver: 1. Voltamos a escrever a password 2. Volta a fazer a mesma derivação usando um *KDF* 3. Verficamos se a tag da chave é a mesma 4. Se a tag da chave for a mesma, decifra a mensagem

[1]:
```
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():
    # Leitura password do emmiter
    password = read_input()
    emmiter = Emitter(password)
    # Deriva chave do emitter
    emmiter.derivate_key()
    # Emmiter envia dados: key_digest + nonce + salt + mensagem
    dados = emmiter.send_message("Segredo que não se pode partilhar")

    # Leitura password do receiver
    password = read_input()
    receiver = Receiver(password)
    # Deriva chave do receiver
    receiver.derivate_key(dados)
    try:
        # Receiver decifra mensagem
        receiver.read_message(dados)
    except:
        # Falha na autenticação da chave
        print("Falha na autenticação da chave")
```

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

```
Insira a sua password:  1234qwerty
Insira a sua password:  1234qwerty

Texto decifrado: Segredo que não se pode partilhar
```