Trabalho Prático 1

André Freitas PG54707

Bruna Macieira PG54467

Exercício 2.

Use o “package” Cryptography para

1. Implementar uma AEAD com “Tweakable Block Ciphers” conforme está descrito na última secção do texto "Capítulo 1: Primitivas Criptográficas Básicas". A cifra por blocos primitiva, usada para gerar a “tweakable block cipher”, é o AES-256 ou o ChaCha20.
2. Use esta cifra 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.

O pacote Cryptography é uma biblioteca Python que fornece fórmulas criptográficas e primitivas para programadores. O uso deste pacote serve para implementar uma AEAD com “Tweakable Block Ciphers”.

AEAD (Authenticated Encryption with Associated Data) é um esquema de criptografia que simultaneamente assegura a confidencialidade dos dados (também conhecida como privacidade: a mensagem criptografada é impossível de entender sem o conhecimento de uma chave secreta) e a autenticidade (ou seja, é inalterável: a mensagem criptografada inclui uma etiqueta de autenticação que o remetente só pode calcular possuindo a chave secreta).

```
import os
import cryptography
import secrets

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms,
modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import x448, ed448
from secrets import token_bytes
```

A classe TweakableChaCha20AEAD implementa uma versão ajustável do algoritmo ChaCha20 para criptografia autenticada com dados associados (AEAD). Essa classe é feita para fornecer confidencialidade e integridade aos dados transmitidos.

__init__(self, key, tweak): Inicializa o cifrador com uma chave de 256 bits (chave) e um valor de ajuste (tweak). A chave é truncada para 256 bits se for mais longa. Tem como métodos:

1. encrypt(self, plaintext, associated_data, nonce): Criptografa o plaintext fornecido junto com os dados associados usando o ChaCha20 e gera uma tag (etiqueta) de autenticação usando HMAC-SHA256. Retorna o texto cifrado e a tag.

2. decrypt(self, cyphertext, tag, associated_data, nonce): Descriptografa o texto cifrado fornecido e verifica a autenticidade usando os dados associados e HMAC-SHA256. Retorna o plaintext descriptografado.

```
class TweakableChaCha20AEAD:
    def __init__(self, key, tweak):
        self.key = key[:32]  # Ensure key is 32 bytes (256 bits)
        self.tweak = tweak

    def encrypt(self, plaintext, associated_data, nonce):
        # Create a ChaCha20 cipher with the tweaked key
        nonce = token_bytes(16)  # Generate nonce
        cipher = Cipher(algorithms.ChaCha20(self.key, nonce),
mode=None, backend=default_backend())
        encryptor = cipher.encryptor()

        # Encrypt the plaintext
        ciphertext = encryptor.update(plaintext) +
encryptor.finalize()

        # Generate a tag using HMAC with associated data
        h = hmac.HMAC(self.key, hashes.SHA256(),
backend=default_backend())
        h.update(associated_data)
        h.update(ciphertext)
        tag = h.finalize()

        return ciphertext, tag

    def decrypt(self, ciphertext, tag, associated_data, nonce):
        # Create a ChaCha20 cipher with the tweaked key
        nonce = token_bytes(16)  # Generate nonce
        cipher = Cipher(algorithms.ChaCha20(self.key, nonce),
mode=None, backend=default_backend())
        decryptor = cipher.decryptor()

        # Decrypt the ciphertext
        plaintext = decryptor.update(ciphertext) +
decryptor.finalize()

        # Verify the tag using HMAC with associated data
        h = hmac.HMAC(self.key, hashes.SHA256(),
backend=default_backend())
        h.update(associated_data)
        h.update(ciphertext)
        h.verify(tag)
```

```
        return plaintext
```

A classe KeyExchange implementa o algoritmo de troca de chaves X448, permitindo que duas partes estabeleçam um segredo compartilhado de maneira segura.

__init__(self): Gera pares de chaves privadas e públicas para dois agentes (agente1 e agente2) usando X448. Tem como método:

1. perform_key_exchange(self, private_key, public_key): Realiza a troca de chaves entre uma chave privada e uma chave pública. Retorna o segredo partilhado.

```
# Define a class for key exchange
class KeyExchange:
    def __init__(self):
        # Generate private and public keys for agent 1
        self.agent1_private_key = x448.X448PrivateKey.generate()
        self.agent1_public_key = self.agent1_private_key.public_key()

        # Generate private and public keys for agent 2
        self.agent2_private_key = x448.X448PrivateKey.generate()
        self.agent2_public_key = self.agent2_private_key.public_key()

    def perform_key_exchange(self, private_key, public_key):
        # Perform key exchange between a private key and a public key
        shared_secret = private_key.exchange(public_key)
        return shared_secret
```

A classe SigningVerification implementa o esquema de assinatura digital Ed448 para assinatura e verificação de mensagens.

__init__(self): Gera pares de chaves privadas e públicas para dois agentes (agente1 e agente2) usando Ed448. Tem como métodos:

1. sign(self, private_key, data): Assina os dados fornecidos usando a chave privada e retorna a assinatura.

2. verify(self, public_key, signature, data): Verifica a assinatura fornecida em relação aos dados usando a chave pública. Retorna True se a verificação for bem sucedida; caso contrário, imprime uma mensagem de erro e retorna False.

```
# Define a class for signing and verification of messages
class SigningVerification:
    def __init__(self):
        # Generate private and public keys for agent 1
        self.agent1_private_key = ed448.Ed448PrivateKey.generate()
        self.agent1_public_key = self.agent1_private_key.public_key()

        # Generate private and public keys for agent 2
        self.agent2_private_key = ed448.Ed448PrivateKey.generate()
```

```
        self.agent2_public_key = self.agent2_private_key.public_key()

    def sign(self, private_key, data):
        # Sign the data using the private key
        signature = private_key.sign(data)
        return signature

    def verify(self, public_key, signature, data):
        try:
            # Verify the signature using the public key
            public_key.verify(signature, data)
            return True
        except Exception as e:
            print(f"Verification failed: {e}")
            return False
```

Duas instâncias são criadas: uma para a troca de chaves (key_exchange) e outra para a verificação de assinaturas (signing_verification), de modo a inicializar o canal assimétrico.

```
key_exchange = KeyExchange()
signing_verification = SigningVerification()
```

Os agentes 1 e 2 realizam a troca de chaves X448, gerando segredos partilhados (shared_secret1 e shared_secret2). No final, verifica se os segredos partilhados dos dois agentes são iguais.

```
# Agent 1 performs key exchange
shared_secret1 =
key_exchange.perform_key_exchange(key_exchange.agent1_private_key,
key_exchange.agent2_public_key)

# Agent 2 performs key exchange
shared_secret2 =
key_exchange.perform_key_exchange(key_exchange.agent2_private_key,
key_exchange.agent1_public_key)

# Check if both shared secrets match
print("Shared secrets match:", shared_secret1 == shared_secret2)
```

```
Shared secrets match: True
```

Os segredos partilhados são usados para derivar chaves de criptografia e autenticação.

```
# Convert shared secrets to bytes for encryption key
encryption_key = shared_secret1
authentication_key = shared_secret2
```

Uma instância da classe TweakableChaCha20AEAD é inicializada com as chaves derivadas.

```
# Initialize the TweakableChaCha20AEAD with the encryption key and
authentication key
tweakable_cipher = TweakableChaCha20AEAD(encryption_key,
authentication_key)
```

A mensagem (plaintext) é então criptografada através do método encrypt (tweakable_cipher) e o texto cifrado (ciphertext) e a etiqueta de autenticação (tag), juntamente com o plaintext, são impressos.

```
# Encrypt
plaintext = b'Hello, World!'
associated_data = b'This is associated data'
nonce = token_bytes(16)  # Generate nonce
ciphertext, tag = tweakable_cipher.encrypt(plaintext, associated_data,
nonce)
print("Plaintext:", plaintext)
print("Ciphertext:", ciphertext)
print("Tag:", tag)
```

```
Plaintext: b'Hello, World!'
Ciphertext: b'11\xd9E\xdf\x01\xebg9t"\xec\xb5'
Tag: b'\xe6\x08T\x01c\x15\xa1\xd4\xf1;\x0f"\x1fRZ\x0bF\xfbN\x8d3\
x04\'\xaf\x95\xd8)\x15}\x00o\xef'
```

A mensagem é descriptografada com o decrypt da tweakable_cipher. Se a descriptografia for bem sucedida, o plaintext é impresso. Se houver um erro de decodificação Unicode, a mensagem descriptografada é impressa como bytes.

```
try:
    decrypted_plaintext = tweakable_cipher.decrypt(ciphertext, tag,
associated_data, nonce)
    print("Decrypted plaintext:", decrypted_plaintext.decode('utf-8'))
except UnicodeDecodeError:
    print("Decrypted plaintext (as bytes):", decrypted_plaintext)
```

```
Decrypted plaintext (as bytes): b'h6\x90\\\xdecs\xbb\x99\xd0E\x04\x85'
```

Finalmente, o agente 1 assina a mensagem e o agente 2 verifica a assinatura, através dos métodos sign e verify da signing_verification, respetivamente.

```
# Agent 1 signs a message
message = b'This is a message.'
signature =
signing_verification.sign(signing_verification.agent1_private_key,
message)

# Agent 2 verifies the signature
print("Signature verified:",
```

```
signing_verification.verify(signing_verification.agent1_public_key,
signature, message))

Signature verified: True
```