tp1-ex3

March 7, 2023

0.1 Trabalho Prático 1

Grupo 13, constituído por:
– Rui Monteiro, PG50739 – Rodrigo Rodrigues, PG50726

3. 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.

0.2 Implementação

Este código implementa uma classe SecureChannel que oferece um canal de comunicação seguro entre dois agentes usando criptografia de chave pública e simétrica.

0.3 Imports

O código importa vários módulos do pacote cryptography para lidar com funções de criptografia e hash.

[1]:
```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.ciphers.aead import AESGCM, ChaCha20Poly1305
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import x448, ed25519
from cryptography.exceptions import InvalidSignature
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.ciphers.aead import AESCCM
```

0.4 Tweakable AEAD

A classe TweakableAEAD é responsável por cifrar e decifrar as mensagens usando as cifras simétricas AES-GCM ou ChaCha20Poly1305, dependendo do tamanho da chave fornecida (16, 24 ou 32 bytes). Ela gera um nonce aleatório e usa o HKDF (HMAC-based Extract-and-Expand Key Derivation Function) para derivar um tweak a partir do nonce e da chave simétrica. O tweak é usado para inicializar a cifra simétrica e garantir que o mesmo nonce nunca seja usado duas vezes.

[2]:
```python
class TweakableAEAD:
    def __init__(self, key, tweak_size=16):
        if len(key) not in [16, 24, 32]:
            raise ValueError("Key must be 128, 192, or 256 bits long")
        self.key = key
        self.tweak_size = tweak_size

    def encrypt(self, plaintext, associated_data):
        # Gerar o tweak a partir do nonce
        nonce = os.urandom(12)
        hkdf = HKDF(
            algorithm=hashes.SHA256(),
            length=self.tweak_size,
            salt=None,
            info=nonce,
        )
        tweak = hkdf.derive(self.key)

        # Selecionar a cifra primitiva de bloco e o modo de operação
        if len(self.key) == 16:
            cipher = Cipher(algorithms.AES(self.key), modes.CTR(tweak))
        else:
            cipher = Cipher(algorithms.ChaCha20(self.key, tweak), mode=None)

        # Criar a AEAD
        aead = AESGCM(self.key) if len(self.key) == 16 else␣
↪ChaCha20Poly1305(self.key)

        # Criptografar a mensagem
        ciphertext = aead.encrypt(nonce, plaintext, associated_data)

        # Retornar a mensagem criptografada com o nonce anexado
        return nonce + ciphertext

    def decrypt(self, ciphertext, associated_data):
        # Separar o nonce da mensagem criptografada
        nonce = ciphertext[:12]
        ciphertext = ciphertext[12:]

        # Gerar o tweak a partir do nonce
        hkdf = HKDF(
            algorithm=hashes.SHA256(),
            length=self.tweak_size,
            salt=None,
            info=nonce,
        )
        tweak = hkdf.derive(self.key)
```

```
        # Selecionar a cifra primitiva de bloco e o modo de operação
        if len(self.key) == 16:
            cipher = Cipher(algorithms.AES(self.key), modes.CTR(tweak))
        else:
            cipher = Cipher(algorithms.ChaCha20(self.key, tweak), mode=None)

        # Criar a AEAD
        aead = AESGCM(self.key) if len(self.key) == 16 else␣
↪ChaCha20Poly1305(self.key)

        # Descriptografar a mensagem
        plaintext = aead.decrypt(nonce, ciphertext, associated_data)

        # Retornar a mensagem descriptografada
        return plaintext
```

0.4.1 SecureChannel

A classe SecureChannel utiliza a classe TweakableAEAD para cifrar e decifrar as mensagens trocadas pelos agentes.

A classe SecureChannel implementa o protocolo de troca de chaves de Diffie-Hellman (DH) com a curva X448. O agente 1 gera uma chave privada X448, envia sua chave pública X448 para o agente 2 e recebe a chave pública X448 do agente 2. O agente 2 gera uma chave privada X448, gera sua chave pública X448 e compartilha com o agente 1. Ambos os agentes usam o HKDF para derivar as chaves de criptografia e autenticação. O agente 2 também assina a chave compartilhada usando a curva de assinatura digital Ed25519 e envia sua chave pública X448 e a assinatura para o agente 1. O agente 1 verifica a assinatura e finaliza o processo de troca de chaves.

O canal de comunicação é considerado seguro se o processo de troca de chaves for bem-sucedido e as mensagens trocadas pelos agentes estiverem criptografadas e autenticadas corretamente.

[3]:
```
class SecureChannel:
    def __init__(self):
        self.agent1_x448_private_key = x448.X448PrivateKey.generate()
        self.tweakable_aead = None

    def initiate_key_exchange(self):
        # Agente 1 gera a chave pública X448 e envia para o agente 2
        agent1_x448_public_key = self.agent1_x448_private_key.public_key()
        return agent1_x448_public_key.public_bytes(
            encoding=serialization.Encoding.Raw,
            format=serialization.PublicFormat.Raw
        )

    def complete_key_exchange(self, agent1_x448_public_key_bytes):
```

```
        # Agente 2 recebe a chave pública X448 do agente 1 e gera a chave pública X448 dele mesmo
        agent1_x448_public_key = x448.X448PublicKey.from_public_bytes(agent1_x448_public_key_bytes)
        agent2_x448_private_key = x448.X448PrivateKey.generate()
        agent2_x448_public_key = agent2_x448_private_key.public_key()

        # Agente 2 gera a chave secreta compartilhada a partir da chave pública do agente 1 e da chave privada dele mesmo
        shared_secret = agent2_x448_private_key.exchange(agent1_x448_public_key)

        # Ambos os agentes usam HKDF para derivar as chaves de criptografia e autenticação
        hkdf = HKDF(
            algorithm=hashes.SHA256(),
            length=64,
            salt=None,
            info=b'secure_channel')

        key_material = hkdf.derive(shared_secret)
        encryption_key = key_material[:32]
        authentication_key = key_material[32:]

        # Agente 2 inicia o processo de assinatura digital da chave compartilhada
        agent2_private_key = ed25519.Ed25519PrivateKey.generate()
        agent2_signature = agent2_private_key.sign(shared_secret)

        # Agente 2 envia sua chave pública X448 e sua assinatura para o agente 1
        return (
            agent2_x448_public_key.public_bytes(
                encoding=serialization.Encoding.Raw,
                format=serialization.PublicFormat.Raw
            ),
            agent2_signature
        )

    def verify_and_finalize_key_exchange(self, agent2_x448_public_key_bytes, agent2_signature):
        # Agente 1 recebe a chave pública X448 e a assinatura do agente 2 e verifica a assinatura
        agent2_public_key = x448.X448PublicKey.from_public_bytes(agent2_x448_public_key_bytes)
        shared_secret = self.agent1_x448_private_key.exchange(agent2_public_key)
```

```
        # Ambos os agentes usam HKDF para derivar as chaves de criptografia e␣
↪autenticação
        hkdf = HKDF(
            algorithm=hashes.SHA256(),
            length=64,
            salt=None,
            info=b'handshake data',)
        key_material = hkdf.derive(shared_secret)
        encryption_key = key_material[:32]
        authentication_key = key_material[32:]

        # Cria o objeto AEAD usando a cifra AES-CCM e as chaves de criptografia␣
↪e autenticação
        self.tweakable_aead = AESCCM(
            key=encryption_key,
            tag_length=16,
        )



    def send_message(self, message):
        # Cifra a mensagem usando a cifra AEAD e adiciona um tweak único
        nonce = b'\x00' * 13
        tweak = b'\x00' * 16
        ciphertext = self.tweakable_aead.encrypt(nonce, message, tweak)

        # Retorna a mensagem cifrada junto com o tweak
        return ciphertext + tweak

    def receive_message(self, ciphertext_with_tweak):
        # Divide a mensagem cifrada em ciphertext e tweak
        ciphertext = ciphertext_with_tweak[:-16]
        tweak = ciphertext_with_tweak[-16:]

        # Decifra a mensagem usando a cifra AEAD e o tweak
        nonce = b'\x00' * 13
        message = self.tweakable_aead.decrypt(nonce, ciphertext, tweak)

        # Retorna a mensagem decifrada
        return message
```

0.5 Definir a função principal `main`

Envio e receção da mensagem "Hello, world!" através do canal privado.

[4]:
```
def main():
    print("Starting secure channel...")
```

```
    secure_channel = SecureChannel()

    # Agente 1 inicia o processo de troca de chaves
    agent1_x448_public_key_bytes = secure_channel.initiate_key_exchange()

    # Agente 2 completa o processo de troca de chaves e envia sua assinatura
    agent2_x448_public_key_bytes, agent2_signature = secure_channel.
↪complete_key_exchange(agent1_x448_public_key_bytes)

    # Agente 1 verifica a assinatura e finaliza o processo de troca de chaves
    secure_channel.
↪verify_and_finalize_key_exchange(agent2_x448_public_key_bytes,␣
↪agent2_signature)

    # Agente 1 envia uma mensagem cifrada para o agente 2
    message = b'Hello, world!'
    ciphertext_with_tweak = secure_channel.send_message(message)
    print(f'Agente 1 enviou: {ciphertext_with_tweak.hex()}')

    # Agente 2 recebe e decifra a mensagem do agente 1
    decrypted_message = secure_channel.receive_message(ciphertext_with_tweak)
    print(f'Agente 2 recebeu: {decrypted_message.decode()}')

main()
```

```
Starting secure channel…
Agente 1 enviou: 0012cbaf0d4e7b57e8c3b9ff6f32bef9adee02dd0cefcc57658185fc7000000
00000000000000000000000000000
Agente 2 recebeu: Hello, world!
```