TP1 - G11

Trabalho realizado por:

- PG57879 - João Andrade Rodrigues
- PG57889 - Martim José Amaro Redondo
- PG57511 - Benjamim Meleiro Rodrigues

Exercício 1

Implementa um canal seguro entre um `Emitter` e um `Receiver` utilizando:

- `asyncio` para comunicação assíncrona. (done, mas falta melhorar)
- `SHAKE-256` em modo `XOFHash` para cifra AEAD. (not done at all)
- Gerador KDF para derivação de chaves e nonces. (done, mas falta verificar)

In [11]:
```
import asyncio
import time
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.backends import default_backend
import hmac
```

SHAKE-256 como XOF:

1. Usamos a função XOF do SHAKE-256, que permite gerar um fluxo ilimitado de bits seguros. Isso substitui a necessidade de usar um HKDF ou outro KDF separado.

2. Ao escolher SHAKE-256 usa-se como base a construção do SHA-3, que tem resistência a colisões e pré-imagens alinhadas com o nível de segurança esperado (256 bits).

Geração de chave de tamanho configurável:

1. Usamos 256 bits (32 bytes) por default por ser considerado o mais seguro para a maioria das aplicações que requerem um nível de segurança elevado.

Uso do shared_input:

1. O shared_input pode ser um segredo compartilhado entre as duas partes como o resultado de um handshake ou troca de chaves. Ao processá-lo com SHAKE-256, obtem-se uma chave derivada segura que benificia da segurança do SHA-3.

In [12]:
```
def derive_key_shake256(shared_input: bytes, length: int = 32) -> bytes:
    """
    Gera 'length' bytes de chave usando SHAKE-256 como XOF,
    a partir de 'shared_input'.
```

```
    """
    shake = hashes.Hash(hashes.SHAKE256(128), backend=default_backend())
    shake.update(shared_input)

    return shake.finalize()[:length]
```

NONCE obtido com ajuda de contador:

1. Na maioria, se não em todos é necessário um nonce único por mensagem. Para se conseguir isso usa-se um contador em vez do urandom(x).

SHAKE-256 em vez do/no lugar do KDF para Nonce

1. Uso das propriedades de segurança do SHAKE-256 para garantir que a mudança do counter resulta em um nonce imprevisível, mesmo que o shared_input seja igual/fixo.

16 bytes no NONCE

1. Evita colisões ao máximo, pois o tamanho é "gigante" (claro que é necessário uma boa gestão do contador).

In [13]:
```
def derive_nonce_shake256(shared_input: bytes, counter: int, length: int = 16) -
    """
    Gera um nonce de 'length' bytes usando SHAKE-256 como XOF,
    a partir de 'shared_input' + um contador (para não repetir).
    """
    shake = hashes.Hash(hashes.SHAKE256(128), backend=default_backend())
    shake.update(shared_input)

    counter_bytes = counter.to_bytes(8, 'big')
    shake.update(counter_bytes)
    return shake.finalize()[:length]
```

XOR com keystream derivado de SHAKE-256:

1. No inicio usou-se GCM para tentar simular o pretendido no TP, contudo optou-se pela construção de uma "keystream" própria a partir da key e do NONCE.

2. Esta "nova keystream" é segura por se aproveutar das propriedades random do SHAKE-256 (desde que haja uma chave screta e o NONCE não se repita).

Encrypt:

1. Depois de cifrar o plaintext, gera-se uma tag usando a key, o nonce, o ad e ociphertext.

2. Com a implementação acima assegura-se que qualquer alteração no ciphertext resultará em uma tag inválida.

3. Esta implementação é denominada "Encrypt-then-MAC", pois protege contra ataques que tentem modificar o texto cifrado.

In [14]:
```python
def shake256_aead_encrypt(key: bytes, nonce: bytes, plaintext: bytes) -> tuple[b
    """
    Retorna (ciphertext, tag).
    - Gera um keystream do mesmo tamanho do plaintext (XOR).
    - Gera uma tag para autenticar (key, nonce, ciphertext).
    """
    shake_for_cipher = hashes.Hash(hashes.SHAKE256(128), backend=default_backend
    shake_for_cipher.update(key)
    shake_for_cipher.update(nonce)
    keystream = shake_for_cipher.finalize()[:len(plaintext)]

    ciphertext = bytes(a ^ b for (a, b) in zip(plaintext, keystream))

    shake_for_tag = hashes.Hash(hashes.SHAKE256(128), backend=default_backend())
    shake_for_tag.update(key)
    shake_for_tag.update(nonce)
    shake_for_tag.update(ciphertext)
    tag = shake_for_tag.finalize()[:32]

    return ciphertext, tag
```

Aqui simplesmente se faz o processo inverso da incriptação:

1. Verificação da Tag antes de decifrar
2. hmac.compare_digest para garantir que a comparação seja feita em tempo real
3. Descriptografia via XOR com o mesmo keystream

In [15]:
```python
def shake256_aead_decrypt(key: bytes, nonce: bytes, ciphertext: bytes, tag: byte
    """
    - Recalcula a tag para (key, nonce, ciphertext) e compara.
    - Se bater, faz XOR com o mesmo keystream e recupera o plaintext.
    """
    shake_for_tag = hashes.Hash(hashes.SHAKE256(128), backend=default_backend())
    shake_for_tag.update(key)
    shake_for_tag.update(nonce)
    shake_for_tag.update(ciphertext)
    computed_tag = shake_for_tag.finalize()[:32]

    if not hmac.compare_digest(computed_tag, tag):
        raise ValueError("Tag inválida! Mensagem adulterada ou chave incorreta."

    shake_for_cipher = hashes.Hash(hashes.SHAKE256(128), backend=default_backend
    shake_for_cipher.update(key)
    shake_for_cipher.update(nonce)
    keystream = shake_for_cipher.finalize()[:len(ciphertext)]

    plaintext = bytes(a ^ b for (a, b) in zip(ciphertext, keystream))
    return plaintext
```

Encrypt e decrypt:

1. stas funções encapsulam a lógica AEAD, retornando o ciphertext e a tag de autenticação em um único buffer.
2. Uso de 32 bytes porque foi o usado até então.
3. O sistema detecta adulteração do ciphertext ou da tag durante a verificação garantindo assim os princípios de "Authenticated Encryption".

In [16]:
```python
def shake256_encrypt(key: bytes, nonce: bytes, message: bytes) -> bytes:
    """
    Retorna ciphertext + tag
    """
    ciphertext, tag = shake256_aead_encrypt(key, nonce, message)
    return ciphertext + tag
```

In [17]:
```python
def shake256_decrypt(key: bytes, nonce: bytes, encrypted_message: bytes) -> byte
    """
    Separa o ciphertext da tag, verifica e retorna plaintext.
    """
    tag = encrypted_message[-32:]
    ciphertext = encrypted_message[:-32]

    plaintext = shake256_aead_decrypt(key, nonce, ciphertext, tag)
    return plaintext
```

Emitter e Receiver

1. Uso de asyncio
2. Chave derivada via derive_key_shake256
3. Nonce derivado do counter
4. O shared_input serve como base para derivar tanto a chave quanto o nonce.

In [18]:
```python
class Emitter:
    def __init__(self, shared_input: bytes):
        print("[Emitter] Inicializando...")
        self.shared_input = shared_input
        self.key = derive_key_shake256(shared_input, 32)

    async def send_message(self, message: str, counter: int):
        print(f"{time.strftime('%X')} - [Emitter] Preparando envio (contador {cc
        await asyncio.sleep(1)

        nonce = derive_nonce_shake256(self.shared_input, counter, 16)
        encrypted_msg = shake256_encrypt(self.key, nonce, message.encode())

        print(f"{time.strftime('%X')} - [Emitter] Mensagem enviada (contador {cc
        return encrypted_msg, counter
```

In [19]:
```python
class Receiver:
    def __init__(self, shared_input: bytes):
        print("[Receiver] Inicializando...")
        self.shared_input = shared_input
        self.key = derive_key_shake256(shared_input, 32)
```

```
    async def receive_message(self, encrypted_msg: bytes, counter: int):
        print(f"{time.strftime('%X')} - [Receiver] Preparando para receber (cont
        await asyncio.sleep(1)

        nonce = derive_nonce_shake256(self.shared_input, counter, 16)
        decrypted_msg = shake256_decrypt(self.key, nonce, encrypted_msg).decode(

        print(f"{time.strftime('%X')} - [Receiver] Mensagem recebida (contador {
        return decrypted_msg
```

Testes:

Teste simples para mostrar como funciona todo o processo entre o Emitter e o Receiver

In [20]:
```
async def main():
    shared_input = b'shared_secret'
    emitter = Emitter(shared_input)
    receiver = Receiver(shared_input)

    tasks = []
    num_mensagens = 3

    for i in range(num_mensagens):
        tasks.append(asyncio.create_task(emitter.send_message(f"Mensagem secreta

    results = await asyncio.gather(*tasks)
    print(f"{time.strftime('%X')} - [MAIN] Todas as mensagens foram enviadas.")

    receive_tasks = []
    for encrypted_msg, counter in results:
        receive_tasks.append(asyncio.create_task(receiver.receive_message(encryp

    await asyncio.gather(*receive_tasks)
    print(f"{time.strftime('%X')} - [MAIN] Todas as mensagens foram recebidas.")

await main()
```

```
[Emitter] Inicializando...
[Receiver] Inicializando...
22:53:08 - [Emitter] Preparando envio (contador 0)...
22:53:08 - [Emitter] Preparando envio (contador 1)...
22:53:08 - [Emitter] Preparando envio (contador 2)...
22:53:09 - [Emitter] Mensagem enviada (contador 0): 56aa71f59b587043bb5cfd531acdb
341f595ab025b06faae4aba0f988228ed8939068f04f1c884084ef0b0c8951e69084bfc
22:53:09 - [Emitter] Mensagem enviada (contador 1): 77f72ad71b6df2f9c5555d453e5bf
ad35a2fea90d7d96804c7b18b442181b700697dae6cdcfb75868aec6410e145f61b22e3
22:53:09 - [Emitter] Mensagem enviada (contador 2): 1b8bc3ad8bc498bdaae12746676a8
9f1ea6fec067ba1cc1eecf1d616bbc5b7bd044e06e8c41cd9f289d476c54d426be1303c
22:53:09 - [MAIN] Todas as mensagens foram enviadas.
22:53:09 - [Receiver] Preparando para receber (contador 0)...
22:53:09 - [Receiver] Preparando para receber (contador 1)...
22:53:09 - [Receiver] Preparando para receber (contador 2)...
22:53:10 - [Receiver] Mensagem recebida (contador 0): Mensagem secreta 1
22:53:10 - [Receiver] Mensagem recebida (contador 1): Mensagem secreta 2
22:53:10 - [Receiver] Mensagem recebida (contador 2): Mensagem secreta 3
22:53:10 - [MAIN] Todas as mensagens foram recebidas.
```