TP1_1

February 27, 2024

0.1 EXERCÍCIO 1

Neste trabalho é nos pedido para, através do uso dos pacotes Cryptography e Ascon, criar uma comunicação privada assíncrona em modo "Lightweight Cryptography" entre um agente Emitter e um agente Receiver. Para além disto é necessário; - Garantir atenticação dos criptogramas e metadados em modo cifra, - As chaves de cifra autenticação e os "nounces" são gerados por um gerador pseudo aleatório (PRG) - Utilização do pacote asyncio para implementar a comunicação entre cliente servidor

Para chegarmos ao resultado pretendido foi nos proposto que usassemos a biblioteca Ascon, que possui uma familia de cifragem e hashing, juntamente com a biblioteca asyncio para a criação do servidor de Emissão e o de Recebimento da mensagem.

Usamos também o nest_asyncio para podermos usar o asyncio no jupter notebook de maneira que os servidores ficassem em execução constante.

[1]:
```
import asyncio
from ascon import _ascon
import secrets
import nest_asyncio
nest_asyncio.apply()
```

A biblioteca **Secrets** do python fornece a fonte mais segura de aleatoriedade que o sistema operacional fornece, é utilizada para gerar numeros aleatórios criptograficamente fortes.

Neste caso, utilizamos para gerar uma seed de 32 bytes que deverá ser passada ao emissor e receptor para gerar as chaves e os nounces.

[2]:
```
seed = secrets.token_bytes(32)
```

0.1.1 Função para gerar chaves

A função hash Ascon-XOF, baseada em esponja, oferece uma maneira inovadora e segura de gerar chaves e nounce. Ela absorve a mensagem de entrada em blocos de 64 bits e "espreme" um valor de hash de 64 bits, ideal para diversos algoritmos criptográficos. O Ascon-XOF foi selecionado como finalista no NIST Lightweight Cryptography Competition por sua segurança e eficiência, sendo ideal para dispositivos com recursos limitados.

Foi utilizada por nós com tamanho de 16 bytes , ou seja, 128 bits, que é o tamanho recomendado pelos desenvolvedores do Ascon

[3]:
```python
def generate_keys(seed):
    key = _ascon.ascon_hash(seed, variant = "Ascon-Xof", hashlength=16)
    nonce = _ascon.ascon_hash(seed, variant="Ascon-Xof", hashlength=16)

    return key, nonce
```

0.1.2 Função para cifrar

Para a cifragem o ascon também utiliza o modo baseado em esponja dupla e para fazer isso:

1. É inicializado o estado com a chave K e o nonce N
2. Atualiza o estado com os blocos de dados correlacionados
3. Injeta a mensagem e extrai os blocos cifrados
4. Injeta a chave novamente e extrai a tag para autenticação

Após cada bloco (exceto o primeiro) a permutação Pb é aplicada no estado completo. Já durante a inicialização e finalização uma permutação mais forte (Pa) com mais rounds é utilizada, o numero de rounds e da taxa da esponja depende da variante.

Neste exercício, utilizamos a variante "Ascon-128" que possui a taxa de 64 bits, e os rounds para Pa 12 e Pb 6.

[4]:
```python
def encrypt_and_authenticate(key, nonce, data, associated_data=b''):
    cipher = _ascon.ascon_encrypt(key,␣
 ↪nonce=nonce,associateddata=associated_data, plaintext=data,␣
 ↪variant="Ascon-128")
    return cipher
```

0.1.3 Função para decifrar

Esta função tem como objetivo fazer a decifragem da mensagem, para isto acontecer é necessário que seja passado o nounce e a key de tamanhos corretos para a função, bem como a mensagem cifrada e os dados correlacionados.

[5]:
```python
def decrypt_and_verify(key, nonce, cipher_text, associated_data=b''):
    plain_txt = _ascon.ascon_decrypt(key,␣
 ↪nonce=nonce,associateddata=associated_data, ciphertext=cipher_text,␣
 ↪variant="Ascon-128")
    return plain_txt
```

0.1.4 Emissor e recebedor

Para o emissor e o recebedor utilizamos a biblioteca Asyncio para fazer rotinas concorrentes no Python.

No emissor, definimos a chave e o nonce, definimos a mensagem e ciframos a mensagem. Para a conexão utilizamos o open_connection do asyncio para estabelecer uma conexão com o localhost na porta 8888, o objeto reader e writer são obtidos, onde o writer é usado para o envio da mensagem.

No receptor, definimos a chave e o nonce e a mensagem é recebida pelo reader. Por fim utilizamos a chave, nonce e os dados recebidos para decifrar a mensagem, que é exibida na tela após a conclusão.

A função main() é usada para iniciar um servidor (receiver) que escuta na porta 8888, executamos o servidor de forma continua e o emissor de forma assincrona.

Já o asyncio.run(main()) garante que o programa ficará em um loop de execução.

[6]:
```
async def emitter():
    # Chaves do emissor

    emitter_key, emitter_nonce = generate_keys(seed)
    print(emitter_key,emitter_nonce)

    # Mensagem a ser enviada
    message = b"Hello, Receiver!"

    # Criptar a mensagem
    encrypted_message = encrypt_and_authenticate(emitter_key, emitter_nonce,␣
 ↪message)

    # Estabelecer conexão
    reader, writer = await asyncio.open_connection('localhost', 8888)

    # Enviar mensagem
    writer.write(encrypted_message)
    await writer.drain()

    # Fechar conexão
    writer.close()

async def receiver(reader, writer):
    # Chaves do recebedor
    receiver_key, receiver_nonce = generate_keys(seed)

    # Receber mensagem
    data = await reader.read()

    # Decifrar mensagem e exibir
    decrypted_data = decrypt_and_verify(receiver_key, receiver_nonce, data)
    print(f"Received and decrypted data: {decrypted_data}")

async def main():
    # Server receiver escutando na porta 8888
    server = await asyncio.start_server(receiver, 'localhost', 8888)

    # Rodar o server emissor
    await asyncio.gather(server.serve_forever(), emitter())
```

```
if __name__ == "__main__":
    asyncio.run(main())
    # loop = asyncio.get_event_loop()
    # loop.run_until_complete(main())
```

```
b'\xa0b\xdb\xfc\xa0,\xdbv\xbfMy\\a\xf4yo'
b'\xa0b\xdb\xfc\xa0,\xdbv\xbfMy\\a\xf4yo'
Received and decrypted data: b'Hello, Receiver!'
```