Exercício 1 - ASCON

February 27, 2024

1 Estruturas Criptográficas - Criptografia e Segurança da Informação

Grupo 03

(PG54177) Ricardo Alves Oliveira

(PG54236) Simão Oliveira Alvim Barroso

1.1 TP1 - Exercício 1

1. Use a package Cryptography e o package ascon (instalar daqui) para criar um comunicação privada assíncrona em modo "Lightweight Cryptography" entre um agente Emitter e um agente Receiver que cubra os seguintes aspectos:
    1. Autenticação do criptograma e dos metadados (associated data) usando Ascon (ver implementação aqui) em modo de cifra.
    2. As chaves de cifra, autenticação e os "nounces" são gerados por um gerador pseudo aleatório (PRG) usando o Ascon em modo XOF. As diferentes chaves para inicialização do PRG são inputs do emissor e do receptor.
    3. Para implementar a comunicação cliente-servidor use o package python `asyncio`.

1.2 Resolução

Para resolver este exercício comecamos por instalar e importar os packages necessarios.

Este setup inicial envolveu, para além do referido no exercício, a utilização do package `nest_asyncio` para garantir a execução apropriada do `asyncio` no `Jupyter Notebook`.

[ ]:
```
%pip install ascon asyncio nest_asyncio

import ascon
import random
import asyncio
import hashlib
import nest_asyncio
```

```
Requirement already satisfied: ascon in
/home/kirim/miniconda3/envs/DAA/lib/python3.10/site-packages (0.0.9)
Requirement already satisfied: asyncio in
/home/kirim/miniconda3/envs/DAA/lib/python3.10/site-packages (3.4.3)
```

```
Requirement already satisfied: nest_asyncio in
/home/kirim/.local/lib/python3.10/site-packages (1.5.6)
Note: you may need to restart the kernel to use updated packages.
```

Para continuar o setup do ambiente iniciamos o `nest_asyncio` e as variáveis globais do programa. Estas incluem o tamanho utilizado para a geração das hashs e a variável `at` utilizada para a geração dos `nounces` através da expansão de chaves com o `XOF`.

[ ]:
```
nest_asyncio.apply()
hashlength=16
at=2
```

Adicionalmente foi criada uma função para calcular a hash de uma mensagem. A hash é calculada utilizando o `SHA3-256` e é utilizada para autenticar a mensagem original, garantindo que esta não é alterada.

[ ]:
```
def calculate_sha256(message):
    if isinstance(message, str):
        message = message.encode()

    sha256_hash = hashlib.sha256(message).hexdigest()
    return sha256_hash
```

O próximo passo foi a definição da função para cifrar as mensagens para que estas possam ser enviadas. Para tal seguiram-se os seguintes passos:

- Foi criada a `associated data` para a mensagem especifica, calculando a hash da mensagem original através da função criada anteriormente.
- Segue-se a criação da `nounce_seed` utilizando, nesta primeira fase, a geração de 128 bits pseudo-aleatórios através do package `random`
- Para gerar o nounce final, a seed previamente gerada é utilizada como input da função de hash do `ascon`. Assim, com recurso ao `Ascon-XOF`, são gerados os bits "aleatórios" para o próprio nounce.
- De seguida cifra-se a mensagem original utilizando a chave, o `nounce` e a `associated data`, utilizando o `ascon-128`.
- Por fim deve-se retornar o `ciphertext`, o `nounce` e a `associated_data` para que possam ser enviados.

[ ]:
```
def cipher_message(in_message,key):
    associated_data=calculate_sha256(in_message).encode()
    nounce_seed=str(random.getrandbits(128))
    nounce=ascon.hash(nounce_seed.encode(),variant="Ascon-Xof",␣
 ↪hashlength=hashlength)

    try:
        out_message=ascon.encrypt(key, nounce, associated_data, in_message.
 ↪encode(), variant="Ascon-128")
        print(f"Sending: {in_message}")
        print(f"Outgoing >>> ({out_message},{nounce},{associated_data})")
```

```
    except Exception as e:
        print(e)
    return (out_message,nounce,associated_data)
```

Alternativamente, a função pode ser alterada de modo a que os `nounces` sejam obtidos a partir da hash gerada com base na `seed` original e no algoritmo `Ascon-XOF`. Esta alteração tira proveito da natureza do `XOF` para gerar uma `stream` pseudo-aleatória de bits, que pode ser utilizada para gerar os `nounces` de forma segura, evitando repetições dos mesmos.

[ ]:
```
def cipher_message(in_message,key,seed):
    global at
    associated_data=calculate_sha256(in_message).encode()
    nounce=ascon.hash(seed.encode(),variant="Ascon-Xof",␣
 ↪hashlength=hashlength*at)[(-1*hashlength):]
    at+=1
    try:
        out_message=ascon.encrypt(key, nounce, associated_data, in_message.
 ↪encode(), variant="Ascon-128")
        print(f"Sending: {in_message}")
        print(f"Outgoing >>> ({out_message},{nounce},{associated_data})")
    except Exception as e:
        print(e)
    return (out_message,nounce,associated_data)
```

Para decifrar as mensagens recebidas o processo passa por:

- Decifrar a mensagem utilizando a chave, o `nounce` e a `associated data` com o `ascon-128`.
- Garantir que foi possível decifrar a mensagem, caso contrário a mensagem é inválida.
- Após decifrar a mensagem é calculada a sua hash (através da função criada anteriormente) e comparada com a `associated data` recebida. Caso estas não coincidam a mensagem é inválida pois foi alterada.
- Por fim é retornada a mensagem original ou o erro encontrado.

[ ]:
```
def read_message(text,key,nounce,associated_data):
    print(f"Incoming <<< ({text},{nounce},{associated_data})")
    try:
        out_message=ascon.decrypt(key, nounce, associated_data, text,␣
 ↪variant="Ascon-128")
    except Exception as e:
        return "[ERROR] Message could not be decrypted"
    if out_message==None:
        return "[ERROR] Message has been tampered"
    elif calculate_sha256(out_message.decode())!=associated_data.decode():
        return "[ERROR] Message has been tampered"
    return out_message.decode()
```

Para além destas funções, foi criada uma função para enviar mensagens para o recetor. Esta segue os seguintes passos:

- O processo inicia-se com a geração da chave privada através do `ascon-XOF` e da `seed` fornecida.
- Aguardar `input` do utilizador para enviar uma nova mensagem.
- Cifrar a mensagem introduzida com recurso à chave e à função previamente criada para cifrar mensagens.
- Adicionar a mensagem cifrada à queue do recetor.
- Repetir até que seja inserido o código para terminar pelo utilizador ("exit")

[ ]:
```
async def emitter(queue,seed):
    key=ascon.hash(seed.encode(),variant="Ascon-Xof", hashlength=hashlength)
    print(f"Sender Key: {key}")
    loop = asyncio.get_event_loop()
    while True:
        message=await loop.run_in_executor(None, input, "Message to send > ")
        out=cipher_message(message,key,seed)
        await queue.put(out)
        if message=="exit":
            break
```

A função para receber as mensagens cifradas, para além de as decifrar, deve verificar se o *nounce* associado já foi recebido previamente, descartando a mensagem caso tal aconteça. Assim, esta função segue os seguintes passos:

- O processo inicia-se com a geração da chave privada através do `ascon-XOF` e da `seed` fornecida.
- Inicia uma lista para armzenar os `nounces` recebidos na sessão
- Aguarda que uma nova mensagem seja obtida da queue
- Verifica se o `nounce` já foi recebido previamente
  - Se sim a mensagem deve ser rejeitada
  - Se não o `nounce` é adicionado à lista de `nounces` recebidos previamente
- Decifrar a mensagem com recurso à chave, `nounce`, `associated_data` e à função previamente criada
- Repetir até que seja recebido um código para terminar pelo emissor ("exit")

[ ]:
```
async def receiver(queue,seed):
    key=ascon.hash(seed.encode(),variant="Ascon-Xof", hashlength=hashlength)
    print(f"Receiver Key: {key}")
    known_nounces=[]
    while True:
        message,nounce,associated_data=await queue.get()
        if nounce in known_nounces:
            print("Repeated nounce, ignoring message")
        else:
            known_nounces.append(nounce)
            text=read_message(message,key,nounce,associated_data)
            print(f"Received: {text}")
        if text=="exit":
            break
```

Finalmente a função principal do programa foi criada com a seguinte estrutura para a execução das diferentes tarefas necessárias para a comunicação segura entre emissor e recetor de forma assíncrona:

- Criar a queue onde serão transmitidas mensagens entre o emissor e o recetor
- Obter uma seed fornecida pelo utilizador para a posterior geração de chaves privadas e nounces
- Criar e lançar os processos para o emissor e recetor das mensagens
- Aguardar que ambos os processos terminem

[ ]:
```
async def main():
    queue = asyncio.Queue()
    key_seed=input("Seed for key > ")
    # Create separate tasks for emitter and receiver
    emitter_task = asyncio.create_task(emitter(queue,key_seed))
    receiver_task = asyncio.create_task(receiver(queue, key_seed))
    # Wait for both tasks to complete
    await asyncio.gather(emitter_task, receiver_task)
    print("All tasks completed")
```

Assim o programa pode ser testado:

[ ]:
```
await main()
```

```
Sender Key: b'\xf1\x94\xac_\xfdj\xdc-\xd9\xd5D*\x87\xb6\x8en'
Receiver Key: b'\xf1\x94\xac_\xfdj\xdc-\xd9\xd5D*\x87\xb6\x8en'
Sending: hello
Outgoing >>> (b'FB\xf2$\xa6gBZ{\xfd\xe0^\xe7\xa5g7\x9cyr*\x05',b'\x91\xc1`\x92\x
08\xe8h\xba\x9e\x1a\x93\x10\x1cTi5',b'2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1f
a7425e73043362938b9824')
Incoming <<< (b'FB\xf2$\xa6gBZ{\xfd\xe0^\xe7\xa5g7\x9cyr*\x05',b'\x91\xc1`\x92\x
08\xe8h\xba\x9e\x1a\x93\x10\x1cTi5',b'2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1f
a7425e73043362938b9824')
Received: hello
Sending: how are you bro
Outgoing >>> (b'{\xe5\xef\xc4G\xc0\x08h\x07|\xe5\x00\xe2\xa2\xf4\xb6\xc5e\xfdo\x
0c\xcb\xde\xff@[Ss<3[',b'\xa8&\xee\xb9\xcb\x85H>\x90\x9bOh_\xb7S\x9b',b'f25bfb5b
88cdd7ba9b16c956fb1033ef8b74e94f31fa0dda216bc70807bd7f26')
Incoming <<< (b'{\xe5\xef\xc4G\xc0\x08h\x07|\xe5\x00\xe2\xa2\xf4\xb6\xc5e\xfdo\x
0c\xcb\xde\xff@[Ss<3[',b'\xa8&\xee\xb9\xcb\x85H>\x90\x9bOh_\xb7S\x9b',b'f25bfb5b
88cdd7ba9b16c956fb1033ef8b74e94f31fa0dda216bc70807bd7f26')
Received: how are you bro
Sending: did it all work?
Outgoing >>> (b'{\x9f^\x98:\xed\x08r\xd0\x99\x06\xc5z^\xc2\xd4f\xe9\xe7\xbe\xcb\
xc8\x89\xfb\x0621l6\x80\xd2\xfd',b'_ \x8d(F\x04a]*A\x9d4\xaeD]{',b'5551d8ef23f21
f9ecb2bef4443a4f175f408373d46f705218f8bdd7ccbf3e230')
Incoming <<< (b'{\x9f^\x98:\xed\x08r\xd0\x99\x06\xc5z^\xc2\xd4f\xe9\xe7\xbe\xcb\
xc8\x89\xfb\x0621l6\x80\xd2\xfd',b'_ \x8d(F\x04a]*A\x9d4\xaeD]{',b'5551d8ef23f21
f9ecb2bef4443a4f175f408373d46f705218f8bdd7ccbf3e230')
Received: did it all work?
Sending: bruhhh
Outgoing >>> (b'\xcfh\xf24y\xef\xf6-
```

```
\x84\x1dV\xb2#2\x10\xb88\xe5\xbc{\xf4\xaf',b"[J\xfa\xd1V\x92Z>pe\x85\xfb\x8e'\x0
e\xf9",b'7873ed83db223b48032429e6efe02c1ce2d0d8afe44edecc0721aef250e8f818')
Incoming <<< (b'\xcfh\xf24y\xef\xf6-
\x84\x1dV\xb2#2\x10\xb88\xe5\xbc{\xf4\xaf',b"[J\xfa\xd1V\x92Z>pe\x85\xfb\x8e'\x0
e\xf9",b'7873ed83db223b48032429e6efe02c1ce2d0d8afe44edecc0721aef250e8f818')
Received: bruhhh
Sending: exit
Outgoing >>> (b'\x9c\x08?/\xe8reu\xc2$\xc5\x17<\x1aq#\xea\xe6\xfdX',b'0\x9a,\xf2
<\x13\x88\xf10\x7f\x12\xaa\xd7\x93\xaeu',b'e596899f114b5162402325dfb31fdaa792fab
ed718628336cc7a35a24f38eaa9')
Incoming <<< (b'\x9c\x08?/\xe8reu\xc2$\xc5\x17<\x1aq#\xea\xe6\xfdX',b'0\x9a,\xf2
<\x13\x88\xf10\x7f\x12\xaa\xd7\x93\xaeu',b'e596899f114b5162402325dfb31fdaa792fab
ed718628336cc7a35a24f38eaa9')
Received: exit
All tasks completed
```

Possiveis Problemas:

- Alterações à mensagem enviada
- Retransmissões de mensagens com o mesmo `nounce`

Para testar estes possiveis problemas foram adicionados os comandos `altered_test` e `repeat_test` à função `cipher_message()` para simular, respetivamente, estas atitudes.

Para o primeiro teste, a mensagem é alterada após ser cifrada. Isto irá resultar numa falha na autenticação da mensagem, uma vez que a hash da mensagem original não irá coincidir com a `associated data` recebida. A alteração da mensagem é feita através da alteração de um byte (neste caso o terceiro da mensagem).

No que toca ao segundo teste, a mensagem é repetida com o mesmo `nounce`. Isto irá resultar numa falha na autenticação da mensagem, uma vez que o `nounce` já foi utilizado previamente. A repetição da mensagem é feita através da reutilização do primeiro `nounce` gerado após a chave através do `Ascon-Xof`.

[ ]:
```python
sent_nounces=[]

def cipher_message(in_message,key,seed):
    global at
    associated_data=calculate_sha256(in_message).encode()
    nounce=ascon.hash(seed.encode(),variant="Ascon-Xof",␣
 ↪hashlength=hashlength*at)[(-1*hashlength):]
    # Message is repeated
    if in_message=="repeat_test":
        nounce=sent_nounces[-1]
    at+=1


    try:
        out_message=ascon.encrypt(key, nounce, associated_data, in_message.
 ↪encode(), variant="Ascon-128")
```

```
        print(f"Sending: {in_message}")

        # Message data is altered
        if in_message=="altered_test":
            print(f"Original >>> ({out_message},{nounce},{associated_data})")
            out_message=out_message[:2]+(f'{out_message[2]+1}'.
↪encode())+out_message[3:]

        print(f"Outgoing >>> ({out_message},{nounce},{associated_data})")
        sent_nounces.append(nounce)
    except Exception as e:
        print(e)

    return (out_message,nounce,associated_data)
```

Assim podemos testar estas atitudes iniciando o programa e, após o envio da primeira mensagem, introduzir o comando pretendido:

- `altered_test` => Para enviar uma mensagem com a `associated_data` alterada
- `repeat_test` => Para enviar uma mensagem com o `nounce` anterior

[ ]:
```
await main()
```

```
Sender Key: b'\xa7\xed\xc5_?FO;\x8a\xc5g\x0e\xeb\xe4\xd1\xf3'
Receiver Key: b'\xa7\xed\xc5_?FO;\x8a\xc5g\x0e\xeb\xe4\xd1\xf3'
Sending: altered_test
Original >>> (b'd\xb0N\xfd\x9c\xc2+\x8cB\xb6\xaa\xe9\xfa\xc8\x82{\x1ak\xf0\xa9\x
85q\xdf\x9d\xbePw\xee',b'X$\xba\x8e\xc5R\xf8\xd7\xd3\xa3\xc5\x98;\xfbR\xd2',b'4d
b0985a79cdcb5f0ae754067b4f6316812e19b52edb3c833ae81250e81b1c53')
Outgoing >>> (b'd\xb079\xfd\x9c\xc2+\x8cB\xb6\xaa\xe9\xfa\xc8\x82{\x1ak\xf0\xa9\
x85q\xdf\x9d\xbePw\xee',b'X$\xba\x8e\xc5R\xf8\xd7\xd3\xa3\xc5\x98;\xfbR\xd2',b'4
db0985a79cdcb5f0ae754067b4f6316812e19b52edb3c833ae81250e81b1c53')
Incoming <<< (b'd\xb079\xfd\x9c\xc2+\x8cB\xb6\xaa\xe9\xfa\xc8\x82{\x1ak\xf0\xa9\
x85q\xdf\x9d\xbePw\xee',b'X$\xba\x8e\xc5R\xf8\xd7\xd3\xa3\xc5\x98;\xfbR\xd2',b'4
db0985a79cdcb5f0ae754067b4f6316812e19b52edb3c833ae81250e81b1c53')
Received: [ERROR] Message could not be decrypted
Sending: hello
Outgoing >>> (b'} \xbf} \xc9\x13\x1b\xfa0\xc1\xc4\n\x0e1O\x0e\xaa\xbdO"',b'\x06\
xfe=wN$\x87\x95\x96\x06\x7f\xcc6\x92\xed\x9a',b'2cf24dba5fb0a30e26e83b2ac5b9e29e
1b161e5c1fa7425e73043362938b9824')
Incoming <<< (b'} \xbf} \xc9\x13\x1b\xfa0\xc1\xc4\n\x0e1O\x0e\xaa\xbdO"',b'\x06\
xfe=wN$\x87\x95\x96\x06\x7f\xcc6\x92\xed\x9a',b'2cf24dba5fb0a30e26e83b2ac5b9e29e
1b161e5c1fa7425e73043362938b9824')
Received: hello
Sending: repeat_test
Outgoing >>> (b'v(}\xd5\xc4\xaf\x9e\xb9\xd2z\xf2\x16\xf8\xa1\xd7\xc1\xe8!\x10\x1
4+SF\xc5\xb5.f',b'\x06\xfe=wN$\x87\x95\x96\x06\x7f\xcc6\x92\xed\x9a',b'0e62efe6e
d04ab0881f10deff96d7271d6c7999317d566ef684a101ee40704be')
```

```
Repeated nounce, ignoring message
Sending: hello
Outgoing >>> (b'\x01R\xc4$\xc1]^\xefJ@)\xef+P\x89Bt\xbf\x96k\xeb',b'}\xf9`\xc0\x
12\xd6\xb7\xcd\xc4\x0f\xf6\xfd\xdd~\x88\xcc',b'2cf24dba5fb0a30e26e83b2ac5b9e29e1
b161e5c1fa7425e73043362938b9824')
Incoming <<< (b'\x01R\xc4$\xc1]^\xefJ@)\xef+P\x89Bt\xbf\x96k\xeb',b'}\xf9`\xc0\x
12\xd6\xb7\xcd\xc4\x0f\xf6\xfd\xdd~\x88\xcc',b'2cf24dba5fb0a30e26e83b2ac5b9e29e1
b161e5c1fa7425e73043362938b9824')
Received: hello
Sending: exit
Outgoing >>> (b'RL\xe9r\xd7\xd3\x07\xb2\xe5Es\xbcl\x13\x80\x95\xc9=\x1eV',b'0"/\
xbdF\t\xf1\xb1`e\xd1=q\xe3\xe1\xdd',b'e596899f114b5162402325dfb31fdaa792fabed718
628336cc7a35a24f38eaa9')
Incoming <<< (b'RL\xe9r\xd7\xd3\x07\xb2\xe5Es\xbcl\x13\x80\x95\xc9=\x1eV',b'0"/\
xbdF\t\xf1\xb1`e\xd1=q\xe3\xe1\xdd',b'e596899f114b5162402325dfb31fdaa792fabed718
628336cc7a35a24f38eaa9')
Received: exit
All tasks completed
```