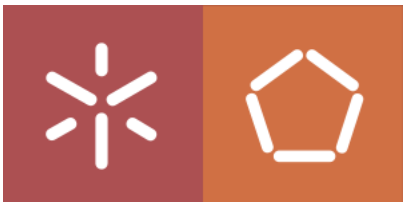

Universidade do Minho
Escola de Engenharia

Estruturas Criptográficas

Sessão Síncrona entre Emitter e Receiver

Comunicação segura com e sem Curvas Elípticas

Submitted To:
José Valença
Professor Catedrático
Tecnologias da Informação e Segurança

Submitted By :
Diogo Araújo, A78485
Diogo Nogueira, A78957
Group 4

Conteúdo

1 Sessão Síncrona entre Emitter e Receiver

1.1 Descrição do Exercício

Construção de uma **sessão síncrona** de comunicação segura entre dois agentes - **Emitter** e **Receiver**.

Requisitos a ter em conta para a comunicação:

- Gerador de *nonces* (IVs)
  - *Nonce* que nunca foi usado antes
  - Criado aleatoriamente em cada instância da comunicação
- Cifra Simétrica **AES**
  - Autenticação de cada criptograma com **HMAC** e um modo seguro contra ataques aos vetores de iniciação
- Protocolo de acordo de chaves **Diffie-Hellman**
  - Com autenticação dos agentes através do esquema de assinaturas **DSA**

1.2 Descrição da Implementação

Perante os requisitos apresentados anteriormente teve de existir uma decisão por parte do grupo em certos aspetos, de modo a tornar toda a comunicação mais segura contra determinados ataques.

Decisões tomadas:

- Cifra Simétrica **AES** usará o modo **CFB**
  - Uso do modo **CFB (*Cipher Feedback*)** dado que ao usarmos este modo não estamos sujeitos a ataque mesmo que o IV em si seja previsível de início. Apenas é necessário garantir que este valor IV seja único para cada utilização. Conseguimos esta singularidade pela atribuição de valores aleatórios.
  - Assim, IVs previsíveis são seguros no modo **CFB**, desde que não se repitam e que não se permita que o invasor os escolha.

1.3 Resolução do Exercício

[1]:
```
import os, io
from SyncPipe import SyncPipe
from DH_DSA import dh_dsa, myMAC
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms,␣
↪modes
from cryptography.exceptions import *
from cryptography.hazmat.backends import default_backend
```

Definição do Agente Emitter

Função que trata de definir o *Emitter* e seu envolvimento no processo de comunicação segura.

Descrição do processo:

- Estabelecimento do acordo de chaves *Diffie-Hellman* com assinatura *Digital Signature Algorithm*
- Criação de uma valor IV aleatório (conforme explicado na **Descrição da Implementação**)
- Criação da cifra AES no modo CFB
- Criação dum HMAC para fazer em cada bloco e depois finalizar.

Com todos estes valores necessários e com um canal pronto para enviar os dados, a ideia é ir lendo blocos de 32 *bytes* devidamente cifrados, enviando-os sucessivamente pelo canal juntamente com a *tag* HMAC criada para o efeito. Após se enviar todos os blocos de dados fecha-se o canal.

[2]:
```
tamanhoMensagem = 2**10

def Emitter(connection):

    # Acordo de chaves DH e assinatura DSA
    key = dh_dsa(connection)

    # Criação dum input stream para enviar a mensagem
    inputs = io.BytesIO(bytes('D'*tamanhoMensagem,'utf-8'))

    # Inicialização do Vector IV aleatório
    iv  = os.urandom(16)

    # Cifra AES com o modo CFB
    cipher = Cipher(algorithms.AES(key), modes.CFB(iv),
                        backend=default_backend()).encryptor()

    # HMAC
    mac = myMAC(key)

    # Enviar o IV para o peer
    connection.send(iv)

    # Criação dum buffer para ler os blocos de 32 bytes (256 bits)
    buffer = bytearray(32)

    # lê, cifra e envia sucessivos blocos do input
    try:
```

```
        # Enquanto existem blocos (while)
        while inputs.readinto(buffer):
            textocifrado = cipher.update(bytes(buffer))
            mac.update(textocifrado)
            connection.send((textocifrado, mac.copy().finalize()))

        # Envia a finalização quando acontecer o último bloco
        connection.send((cipher.finalize(), mac.finalize()))

    except Exception as err:
        print("Erro no emissor: {0}".format(err))

    # Fechar o stream para colocar inputs
    inputs.close()

    # Fechar a conexão com o peer
    connection.close()

    # Eliminar chave
    key = None
```

Definição do Agente Receiver

Função que trata de definir o *Receiver* e o seu envolvimento no proceso de comunicação segura.

Descrição do processo:

- Estabelecimento do acordo de chaves *Diffie-Hellman* com assinatura DSA
- Recebimento do valor IV enviado pelo *Emitter*
- Criação da cifra AES no modo CFB
- Criação do valor de MAC a verificar

O processo estipulado para o *Receiver* difere do agente que lhe envia a informação. Aqui é necessário verificar se a *tag* MAC criada é igual à que foi recebida, caso contrário estamos perante um erro. Com as devidas verificações pode-se ir lendo os blocos que vão surgindo por parte do *Emitter* ao mesmo tempo que decifram. Imprime-se a mensagem final no *Notebook* em si.

[3]:
```
def Receiver(connection):

    # Acordo de chaves DH e assinatura DSA
    key = dh_dsa(connection)

    # Inicializa um output stream para receber o texto cifrado
```

```
    outputs = io.BytesIO()

    # Recebe o Vetor IV
    iv = connection.recv()

    # Cifra AES com o modo CFB
    cipher = Cipher(algorithms.AES(key), modes.CFB(iv),
                        backend=default_backend()).decryptor()

    # HMAC
    mac = myMAC(key)

    # Operar a cifra: ler da conexão um bloco, autenticá-lo, decifrá-lo e␣
↪escrever o resultado no stream de output
    try:

        while True:
            try:
                # Receber do Emitter o buffer de 32 bytes e tag MAC␣
↪associada
                buffer, tag = connection.recv()

                ciphertext = bytes(buffer)
                mac.update(ciphertext)

                # Verificação se a tag é igual à criada acima.
                if tag != mac.copy().finalize():
                    raise InvalidSignature("Erro no bloco intermédio")

                # Colocar no stream o texto decifrado e corretamente␣
↪autenticado
                outputs.write(cipher.update(ciphertext))

                # Caso já não haja mais buffer (blocos)
                if not buffer:

                    # Verificação se a tag MAC foi finalizada
                    if tag != mac.finalize():
                        raise InvalidSignature("Erro na finalização")

                    # Finalizar a cifra e escrever no stream
                    outputs.write(cipher.finalize())
                    break
```

```
            except InvalidSignature as err:
                raise Exception("Autenticação do ciphertext ou metadados:␣
↪{}".format(err))

        # Escrever no Jupyter Notebook os resultados colocados na stream
        print(outputs.getvalue())

    except Exception as err:
        print("Erro no receptor: {0}".format(err))

    # Fechar o stream dos outputs
    outputs.close()

    # Fechar a conexão
    connection.close()

    # Eliminar chave
    key = None
```

Criação dos Pipes e execução dos Agentes

[4]:
```
SyncPipe(Emitter, Receiver).auto()
```

```
Está verificada a assinatura do parceiro.
Está verificada a assinatura do parceiro.
Valid DH (MAC)
Valid DH (MAC)
b'DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD'
```

1.4 Observações Finais

- Fácil integração da cifra AES às entidades e o canal em si
- Escolha de uma mensagem aleatoriamente criada pelo grupo para envio pelo canal
  - Mensagem consideravelmente grande para garantir a existência de blocos para uma **sessão síncrona**
- Toda a parte do acordo de chaves DH é feita num ficheiro à parte

1.5 Referências

- Python Documentation, Process-based parallelism https://docs.python.org/3/library/multiprocessing.html (Acedido a 2 Março 2020)
- evantotuts+, Introduction to Multiprocessing in Python https://code.tutsplus.com/tutorials/introduction-to-multiprocessing-in-python–cms-30281 (Acedido a 5 março 2020)
- Wikipedia, Block ciper mode of operation https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation (Acedido a 7 março 2020)
- Cryptography, Symmetric encryptionhttps://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/ (Acedido a 7 março 2020)

2 Sessão Síncrona entre Emitter e Receiver com uso de Curvas Elípticas

2.1 Descrição do Exercício

Versão alternativa da sessão do exercício anterior mas com o uso de **curvas Elípticas**.

Requisitos a ter em conta para esta versão de comunicação:

- Cifra Simétrica **AES** substituída pela cifra **ChaCha20Poly1305**
- Protocolo de acordo de chaves **Diffie-Hellman** agora por Curvas Elípticas (**ECDH**)
  - Esquema de assinaturas **DSA** agora por Curvas Elípticas (**ECDSA**)
- Foi escolhida a curva elíptica *SECP384R1*

2.2 Descrição da Implementação

Não foi necessário definir quaisquer decisões consoante o que foi pedido. Apenas se efetuaram as substituições pedidas ficando o programa similar ao anterior em termos de **Emitter** e **Receiver**. A grande diferença foi a retirada da autenticação por **HMAC**, dado que agora a cifra pedida pelo professor já é autenticada, através do **MAC** intitulado de **Poly1305**.

2.3 Resolução do Exercício

[1]:
```
import os, io
from SyncPipe import SyncPipe
from ECDH_ECDSA import ecdh_ecdsa
from cryptography.hazmat.primitives.ciphers.aead import ChaCha20Poly1305
from cryptography.exceptions import *
```

Definição do Agente Emitter

Função que trata de definir o *Emitter* e seu envolvimento no processo de comunicação segura.

Descrição do processo:

- Estabelecimento do acordo de chaves *Elliptic-curve Diffie-Hellman* com assinatura *Elliptic Curve Digital Signature Algorithm*
- Criação de um *nonce* aleatório a cada comunicação
- Criação da cifra ChaCha20Poly1305

Com todos estes valores necessários e com um canal pronto para enviar os dados, a ideia é ir lendo blocos de 32 *bytes* devidamente cifrados, enviando-os sucessivamente pelo canal. Após se enviar todos os blocos de dados fecha-se o canal.

[2]:
```
tamanhoMensagem = 2**10

def Emitter(connection):

    # Acordo de chaves DH e assinatura DSA
    key = ecdh_ecdsa(connection)

    # Criação dum input stream para enviar a mensagem
    inputs = io.BytesIO(bytes('D'*tamanhoMensagem,'utf-8'))

    # Inicialização do nonce aleatório (12 bytes)
    nonce  = os.urandom(12)

    # Cifra ChaCha20Poly1305
    cipher = ChaCha20Poly1305(key)

    # Enviar o IV para o peer
    connection.send(nonce)

    # Criação dum buffer para ler os blocos de 32 bytes (256 bits)
    buffer = bytearray(32)

    # lê, cifra e envia sucessivos blocos do input
    try:

        # Enquanto existem blocos (while)
        while inputs.readinto(buffer):
            ciphertext = cipher.encrypt(nonce, bytes(buffer), None)
            connection.send(ciphertext)
    except Exception as err:
        print("Erro no emissor: {0}".format(err))

    # Fechar o stream para colocar inputs
    inputs.close()

    # Fechar a conexão com o peer
    connection.close()
```

Definição do Agente Receiver

Função que trata de definir o *Receiver* e o seu envolvimento no proceso de comunicação segura.

Descrição do processo:

- Estabelecimento do acordo de chaves *Elliptic-curve Diffie-Hellman* com assinatura *Elliptic Curve Digital Signature Algorithm*
- Recebimento do *nonce* enviado pelo *Emitter*
- Decifração com cifra ChaCha20Poly1305

O processo estipulado para o *Receiver* difere do agente que lhe envia a informação. Com as devidas verificações pode-se ir lendo os blocos que vão surgindo por parte do *Emitter* ao mesmo tempo que decifram. Imprime-se a mensagem final no *Notebook* em si.

[3]:
```
def Receiver(connection):

    # Acordo de chaves DH e assinatura DSA
    key = ecdh_ecdsa(connection)

    # Inicializa um output stream para receber o texto decifrado
    outputs = io.BytesIO()

    # Recebe o Vetor IV
    nonce = connection.recv()

    # Cifra ChaCha20Poly1305
    cipher = ChaCha20Poly1305(key)

    # Operar a cifra: ler da conexão um bloco, autenticá-lo, decifrá-lo e␣
↪escrever o resultado no stream de output
    try:
        while True:
            try:
                # Receber do Emitter o buffer de 32 bytes e tag MAC␣
↪associada
                buffer = connection.recv()

                ciphertext = bytes(buffer)
                decifredtext = cipher.decrypt(nonce, ciphertext, None)

                # Colocar no stream o texto decifrado e corretamente␣
↪autenticado
                outputs.write(decifredtext)
                break
```

```
            except InvalidSignature as err:
                raise Exception("Autenticação do ciphertext ou metadados:␣
↪{}".format(err))

        # Escrever no Jupyter Notebook os resultados colocados na stream
        print(outputs.getvalue())

    except Exception as err:
        print("Erro no receptor: {0}".format(err))

    # Fechar o stream dos outputs
    outputs.close()

    # Fechar a conexão
    connection.close()
```

Criação dos Pipes e execução dos Agentes

[4]:
```
SyncPipe(Emitter, Receiver).auto()
```

```
Está verificada a assinatura do parceiro.
Está verificada a assinatura do parceiro.
Valid ECDH (MAC)
Valid ECDH (MAC)
b'DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD'
```

2.4 Observações Finais

- A parte do acordo de chaves ECDH com assinatura ECDSA foi feita, igualmente ao exercício anterior, num ficheiro à parte
- Maior dificuldade em entender como usar a cifra ChaCha20Poly1305, dado que se trata de uma *Stream Cipher* e estamos a lidar com uma **sessão síncrona** com uma implementação a pensar em blocos.

2.5 Referências

- Wikipedia, Elliptic-curve Diffie–Hellman https://en.wikipedia.org/wiki/Elliptic-curve_Diffie%E2%80%93Hellman (Acedido a 10 março 2020)
- Wikipedia, Elliptic Curve Digital Signature Algorithm https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm (Acedido a 10 março 2020)

- Doc Sagemath, Elliptic curves http://doc.sagemath.org/html/en/constructions/elliptic_curves.html (Acedido a 10 março 2020)
- Cryptography, Authenticated encryption https://cryptography.io/en/latest/hazmat/primitives/aead/ (Acedido a 10 março 2020)
- Cryptography, Elliptic curve cryptography https://cryptography.io/en/latest/hazmat/primitives/asymmetric/ec/ (Acedido a 11 março 2020)