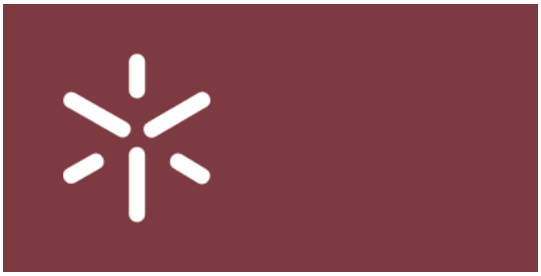

TP1 - Estruturas Criptográficas

MIEI - 4º ano - 2º semestre

Universidade do Minho

Grupo 9

Guilherme Guerreiro	José Bastos	Ricardo Certo
A73860	A74696	A75315

20 de Março de 2018

Exercício 1

Pretende-se implementar uma comunicação cifrada em regime síncrono entre um agente **Emitter** e um **Receiver**, usando a cifra simétrica **AES** e a autenticação de cada criptograma com **Hash-based Message Authentication Code** .

Diffie Hellman & DSA

Implementou-se também o protocolo de acordo de chaves **Diffie-Hellman** com verificação da chave e autenticação mútua dos agente através do esquema de assinaturas **Digital Signature Algorithm**. O protocolo **Diffie-Hellman** contém 3 algoritmos:

- A criação dos parâmetros
- Agente Emitter gera a chave privada, a sua respetiva chave pública e envia ao Receiver
- Neste acordo de chaves o agente Receiver procede de igual forma.
- Finalmente, ambos agente geram a chave partilhada e é usada uma autenticação MAC na respetiva chave na comunicação entre os agente.
- No final da implementação obliterámos os registos dos dados, removendo assim a informação relacionado aos agentes.

In [48]:

```
import os, io
from getpass import getpass
from BiConn import BiConn
from Auxs   import hashs
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.exceptions import *
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, hmac, cmac
from cryptography.hazmat.primitives.asymmetric import dh,dsa
from cryptography.hazmat.primitives import serialization,hashes

# Generate some parameters DH
parameters_dh = dh.generate_parameters(generator=2, key_size=1024,
                                       backend=default_backend())

# Generate some parameters DSA
parameters_dsa = dsa.generate_parameters(key_size=1024,backend=default_backend
())

default_algorithm = hashes.SHA256
# seleciona-se um dos vários algorimos implementados na package

def my_mac(key):
     return hmac.HMAC(key,default_algorithm(),default_backend())
```

In [49]:

```
def Dh(conn):
    # agreement
    pk = parameters_dh.generate_private_key()
    pub = pk.public_key().public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo)

   #ASSINAR

   #gerar as chaves privada e pública
    private_key_dsa = parameters_dsa.generate_private_key()
    pub_dsa = private_key_dsa.public_key().public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo)

    #envia a chave pública
    conn.send(pub_dsa)

    #cálculo da assinatura
    signature = private_key_dsa.sign(pub, hashes.SHA256())

    peer_pub_dsa =serialization.load_pem_public_key(
        conn.recv(),
        backend=default_backend())

    conn.send(pub)
    conn.send(signature)

    #verificar
    try:
        peer_pub = conn.recv()
        sig = conn.recv()
        peer_pub_dsa.verify(sig, peer_pub, hashes.SHA256())
        print("ok dsa")
    except InvalidSignature:
        print("fail dsa")


    # shared_key calculation
    peer_pub_key = serialization.load_pem_public_key(
            peer_pub,
            backend=default_backend())
    shared_key = pk.exchange(peer_pub_key)

    # confirmation
    my_tag = hashs(bytes(shared_key))
    conn.send(my_tag)
    peer_tag = conn.recv()
    if my_tag == peer_tag:
        print('OK DH')
        return my_tag
    else:
        print('FAIL DH')

    #eliminar dados
    pk = None
    pub = None
    peer_pub = None
    peer_pub_key = None
```

```
    shared_key = None
    my_tag = None
    peer_tag = None
```

Cifra

Na comunicão entre os agente foi implementeada a cifra **AES** na qual foi usado o modo **Cipher Feedback** pois considerámos que é dos melhores modos para evitar ataques aos vetores de inicialização. Sendo que o primeiro bloco é C0=Ek(IV)⊕P0, mesmo que o atacante saiba de início o valor do **IV** ele apenas sabe o primeiro bloco que será apresentado ao bloco da cifra, logo, desde que esse vetor de inicialização não se repete (o que nos assegurámos disso ao preenchê-lo com valores aleatórios) podemos concluir que este modo é seguro contra ataques de inicialização.

Emitter

In [50]:

```
message_size = 2**10

def Emitter(conn):
    # Acordo de chaves DH e assinatura DSA
    key = Dh(conn)

    # Mensagem
    inputs = io.BytesIO(bytes('1'*message_size,'utf-8'))

    # iv para a cifra
    iv  = os.urandom(16)

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

    # HMAC
    mac = my_mac(key)

    conn.send(iv) # Envio do iv
    buffer = bytearray(32) # Buffer onde vão ser lidos os blocos

    # lê, cifra e envia sucessivos blocos do input
    try:
        while inputs.readinto(buffer):
            ciphertext = cipher.update(bytes(buffer))
            mac.update(ciphertext)
            conn.send((ciphertext, mac.copy().finalize()))

        conn.send((cipher.finalize(), mac.finalize()))    # envia a finalização
    except Exception as err:
        print("Erro no emissor: {0}".format(err))

    inputs.close()          # fecha a 'input stream'
    conn.close()            # fecha a conecção

    # Eliminar chave
    key = None
```

Receiver

In [51]:

```
def Receiver(conn):
    # Acordo de chaves DH e assinatura DSA
    key = Dh(conn)

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

    # Recebe o iv
    iv = conn.recv()

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

    # HMAC
    mac = my_mac(key)

    # operar a cifra: ler da conecção um bloco, autenticá-lo, decifrá-lo e escre
ver o resultado no 'stream' de output
    try:
        while True:
            try:
                buffer, tag = conn.recv()
                ciphertext = bytes(buffer)
                mac.update(ciphertext)
                if tag != mac.copy().finalize():
                    raise InvalidSignature("erro no bloco intermédio")
                outputs.write(cipher.update(ciphertext))
                if not buffer:
                    if tag != mac.finalize():
                        raise InvalidSignature("erro na finalização")

                    outputs.write(cipher.finalize())
                    break

            except InvalidSignature as err:
                raise Exception("autenticação do ciphertext ou metadados: {}".fo
rmat(err))
        print(outputs.getvalue())     # verificar o resultado

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

    outputs.close()    # fechar 'stream' de output
    conn.close()       # fechar a conecção

    # Eliminar chave
    key = None
```

In [52]:

```
BiConn(Emitter, Receiver, timeout=30).auto()
```

```
ok dsa
OK DH
ok dsa
OK DH
b'1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
111111'
```

Exercício 2 - Uso de Curvas Elíticas

Sessão igual ao esquema do exercício anterior, mas agora com o uso de curvas elíticas.

Em vez do protocolo de acordo chaves **Diffie–Hellman** usou-se **Elliptic-curve Diffie–Hellman** e em vez do **Digital Signature Algorithm** usou-se **Elliptic Curve Digital Signature Algorithm** .

In [94]:

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import hashes, hmac, cmac
from cryptography.hazmat.primitives.asymmetric import dh, dsa
from cryptography.hazmat.primitives import serialization, hashes
from getpass import getpass
from cryptography.exceptions import *
from cryptography.hazmat.primitives.asymmetric import ec

default_algorithm = hashes.SHA256
# seleciona-se um dos vários algorimos implementados na package

def my_mac(key):
     return hmac.HMAC(key,default_algorithm(),default_backend())
```

In [95]:

```
from BiConn import BiConn
from Auxs import hashs, mac, kdf
import getpass, os, io

default_curve = ec.SECP256R1 #curva SECP256R1
def ECDH(conn):
    # agreement
    pk = ec.generate_private_key(default_curve, default_backend()) #ao gerar a c
have privada,
    pub = pk.public_key().public_bytes(                             #recebe como
 argumento a curva definida
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo)


    #gerar as chaves privada e pública
    private_key_dsa = ec.generate_private_key(default_curve, default_backend())
    pub_dsa = private_key_dsa.public_key().public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo)

    #envia a chave pública
    conn.send(pub_dsa)

    #cálculo da assinatura
    signature = private_key_dsa.sign(pub, ec.ECDSA(hashes.SHA256())) #ECDSA rece
be como argumento a hash256

    peer_pub_dsa =serialization.load_pem_public_key(
        conn.recv(),
        backend=default_backend())

    conn.send(pub)
    conn.send(signature)

    #ASSINAR

    try:
        peer_pub = conn.recv()
        sig = conn.recv()
        peer_pub_dsa.verify(sig, peer_pub, ec.ECDSA(hashes.SHA256()))
        print("ok ecdh")
    except InvalidSignature:
        print("fail ecdh")

    # shared_key calculation
    peer_pub_key = serialization.load_pem_public_key(
            peer_pub,
            backend=default_backend())
    shared_key = pk.exchange(ec.ECDH(), peer_pub_key) #em vez de se trocar apena
s a chave, tambem se troca ECDH


    # confirmation
    my_tag = hashs(bytes(shared_key))
    conn.send(my_tag)
    peer_tag = conn.recv()
    if my_tag == peer_tag:
        print('OK ECDH')
```

```
        return my_tag
    else:
        print('FAIL ECDH')

    #eliminar dados
    pk = None
    pub = None
    peer_pub = None
    peer_pub_key = None
    shared_key = None
    my_tag = None
    peer_tag = None
```

In [96]:

```
message_size = 2**10

def Emitter(conn):
    # Acordo de chaves DH e assinatura DSA
    key = ECDH(conn)

    # Mensagem
    inputs = io.BytesIO(bytes('1'*message_size,'utf-8'))

    # iv para a cifra
    iv  = os.urandom(16)

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

    # HMAC
    mac = my_mac(key)

    conn.send(iv) # Envio do iv
    buffer = bytearray(32) # Buffer onde vão ser lidos os blocos

    # lê, cifra e envia sucessivos blocos do input
    try:
        while inputs.readinto(buffer):
            ciphertext = cipher.update(bytes(buffer))
            mac.update(ciphertext)
            conn.send((ciphertext, mac.copy().finalize()))

        conn.send((cipher.finalize(), mac.finalize()))    # envia a finalização
    except Exception as err:
        print("Erro no emissor: {0}".format(err))

    inputs.close()          # fecha a 'input stream'
    conn.close()            # fecha a conecção
```

In [97]:

```
def Receiver(conn):
    # Acordo de chaves DH e assinatura DSA
    key = ECDH(conn)

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

    # Recebe o iv
    iv = conn.recv()

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

    # HMAC
    mac = my_mac(key)

    # operar a cifra: ler da conecção um bloco, autenticá-lo, decifrá-lo e escre
ver o resultado no 'stream' de output
    try:
        while True:
            try:
                buffer, tag = conn.recv()
                ciphertext = bytes(buffer)
                mac.update(ciphertext)
                if tag != mac.copy().finalize():
                    raise InvalidSignature("erro no bloco intermédio")
                outputs.write(cipher.update(ciphertext))
                if not buffer:
                    if tag != mac.finalize():
                        raise InvalidSignature("erro na finalização")

                    outputs.write(cipher.finalize())
                    break

            except InvalidSignature as err:
                raise Exception("autenticação do ciphertext ou metadados: {}".fo
rmat(err))
        print(outputs.getvalue())     # verificar o resultado

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

    outputs.close()    # fechar 'stream' de output
    conn.close()       # fechar a conecção
```

In [98]:

```
BiConn(Emitter, Receiver, timeout=30).auto()
```

```
ok ecdh
ok ecdh
OK ECDH
OK ECDH
b'1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111
111111'
```