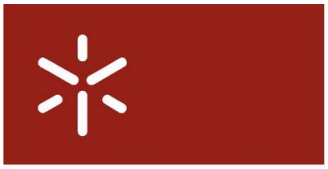

Universidade do Minho

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Estruturas Criptográficas

TP1

Autores:

João Cabo A75064

Diogo Soares A74478

12 de Março de 2019

TP1ex1

March 13, 2019

1 Exercicio 1

Pretende-se construir uma sessão sincrona segura entre um agente Emitter e um agente Receiver, usando a cifra simétrica TAES autenticando o criptograma em cada superbloco.

2 Diffie-Hellman e DSA

Usou-se o protocolo de acordo de chaves Diffie-Hellman com verificação de chaves, usando o esquema de assinaturas Digital Signature Algorithm para autenticação dos agentes. O protocolo DH contém 3 algoritmos:

- Geração dos parâmetros
- Agente A gera uma chave privada, a respetica chave pública e envia-a ao agente B
- Agente B procede de forma análoga.
- Seguidamente ambos os agentes computam a chave partilhada e usam uma autenticação MAC para confirmar.

```
In [2]: from cryptography.hazmat.backends import default_backend
        from cryptography.hazmat.primitives import hashes, hmac

        default_algorithm = hashes.SHA256

        def hashf(s):
            digest = hashes.Hash(default_algorithm(),backend=default_backend())
            digest.update(s)
            return digest.finalize()

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

```
In [3]: from cryptography.hazmat.primitives.asymmetric import dh,dsa
        from cryptography.hazmat.backends import default_backend
        from cryptography.hazmat.primitives import serialization
        from cryptography.exceptions import *

        #Generate some parameters for Diffie-Hellman
        parameters_dh = dh.generate_parameters(generator=2,key_size=3072,
```

```
                                        backend=default_backend())

#Generate some parameters for DSA
parameters_dsa = dsa.generate_parameters(key_size=3072,
                                         backend=default_backend())

def DHDSA(conn):
    """
        DH
   Generate Keys
    """
    #generate private key DH
    pk_DH = parameters_dh.generate_private_key()
    #gemerate public key DH
    pub_DH = pk_DH.public_key().public_bytes(encoding=serialization.Encoding.PEM,
                                             format=serialization.PublicFormat.SubjectI



    """
        DSA
   Generate Keys
    """

    #generate private key DSA
    pk_DSA = parameters_dsa.generate_private_key()
    #generate public key DSA
    pub_DSA = pk_DSA.public_key().public_bytes(encoding=serialization.Encoding.PEM,
                                               format=serialization.PublicFormat.Subje

    """
    Share Key
    """

    #send public key DSA
    conn.send(pub_DSA)

    #signing
    signature = pk_DSA.sign(pub_DH,hashes.SHA256())

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

    #send public key DH and signature
    conn.send(pub_DH)
    conn.send(signature)


    #check if signature is ok
```

```
            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_DH.exchange(peer_pub_key)

            #confirmation
            my_tag = hashf(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')

            #erase data
            pk_DH = None
            pub_DH = None
            peer_pub = None
            peer_pub_key = None
            shared_key = None
            my_tag = None
            peer_tag = None
```

In [4]:
```
from multiprocessing import Pipe, Process

class BiConn(object):
    def __init__(self,left,right,timeout=None):
        """
        left : a função que vei ligar ao lado esquerdo do Pipe
        right: a função que vai ligar ao outro lado
        timeout: (opcional) numero de segundos que aguarda pela terminação do processo
        """
        left_end, right_end = Pipe()
        self.timeout=timeout
        self.lproc = Process(target=left, args=(left_end,))       # os processos ligad
        self.rproc = Process(target=right, args=(right_end,))
        self.left  = lambda : left(left_end)                       # as funções ligada
```

```
        self.right = lambda : right(right_end)

    def auto(self, proc=None):
        if proc == None:              # corre os dois processos independentes
            self.lproc.start()
            self.rproc.start()
            self.lproc.join(self.timeout)
            self.rproc.join(self.timeout)
        else:                         # corre só o processo passado como parâmetro
            proc.start(); proc.join()

    def manual(self):   #  corre as duas funções no contexto de um mesmo processo Pyth
        self.left()
        self.right()
```

3 Cifra

Na comunição entre os agentes implementamos a cifra AES no seu modo tweakable, para tal, foi usado o modo XTS.

```
In [5]: import os, io
        from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

        message_size = 2**10

        def Emitter(conn):

            #Acordo de chaves DH e assinatura DSA
            key = DHDSA(conn)

            # Cria um input stream com um número grande de bytes aleatórios
            # inputs = io.BytesIO(os.urandom(message_size))
            inputs = io.BytesIO(bytes('1'*message_size,'utf-8'))

            # tweak
            tweak = os.urandom(16)

            #Cipher
            cipher = Cipher(algorithms.AES(key), modes.XTS(tweak),
                            backend=default_backend()).encryptor()


            #HMAC
            mac = my_mac(key)

            #envia pela conecção o tweak
            conn.send(tweak)
```

```
            #define um buffer para ondevão ser lidos, sucessivamente, os vários blocos do inpu
            buffer = bytearray(32)

            # 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))


            print(key)


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

            key = None
```

In [6]:
```
        def Receiver(conn):

            #Acordo de chaves DH e assinatura DSA
            key = DHDSA(conn)

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

            #tweak
            tweak = conn.recv()

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

            #HMAC
            mac = my_mac(key)

            # operar a cifra: ler da conecção um bloco, decifrá-lo e escrever o resultado no '
            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("erro autenticação do ciphertext: {}".format(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 [7]: BiConn(Emitter,Receiver,timeout=30).auto()
```

```
ok dsa
ok dsa
OK DH
OK DH
b'\xf7XS\xec\xc2H\x14\x15\x1e\x0e\xe7\xc6\xa8F\x8a\xfa\x18\x88\xe9@7\xdf\xc2\xa6`Y\xd3\xb24%\x
b'1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
```

```
In [ ]:
```

TP1ex2

March 13, 2019

1 Exercicio 2

Neste exercicio pretende se que se repita o anterior mas desta vez recorrendo a curvas elíticas. Para isso, substitui-se o acordo de chaves Diffie-Hellman pelo Elliptic-curve Diffie Hellman e usou-se Elliptic-Curve Digital Signature Algorithm em vez do Digital Signature Algorithm.

```
In [1]: from cryptography.hazmat.backends import default_backend
        from cryptography.hazmat.primitives import hashes, hmac

        default_algorithm = hashes.SHA256

        def hashf(s):
            digest = hashes.Hash(default_algorithm(),backend=default_backend())
            digest.update(s)
            return digest.finalize()

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

```
In [3]: from cryptography.hazmat.primitives.asymmetric import dh,dsa
        from cryptography.hazmat.backends import default_backend
        from cryptography.hazmat.primitives import serialization
        from cryptography.exceptions import *
        from cryptography.hazmat.primitives.asymmetric import ec

        default_curve_DH = ec.SECP521R1 #curva eliptica para acordo de chaves
        default_curve_DSA = ec.SECP256R1 #curva eliptica para autenticação agentes

        def ECDHDSA(conn):
            """
                DH
           Generate Keys
            """
            #generate private key DH
            pk_DH = ec.generate_private_key(default_curve_DH, default_backend())
            #gemerate public key DH
            pub_DH = pk_DH.public_key().public_bytes(encoding=serialization.Encoding.PEM,
                                                     format=serialization.PublicFormat.SubjectI
```

```
    """
        DSA
    Generate Keys
    """

    #generate private key DSA
    pk_DSA = ec.generate_private_key(default_curve_DSA, default_backend())
    #generate public key DSA
    pub_DSA = pk_DSA.public_key().public_bytes(encoding=serialization.Encoding.PEM,
                                               format=serialization.PublicFormat.Subje

    """
    Share Key
    """

    #send public key DSA
    conn.send(pub_DSA)

    #signing
    signature = pk_DSA.sign(pub_DH,ec.ECDSA(hashes.SHA256()))

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

    #send public key DH and signature
    conn.send(pub_DH)
    conn.send(signature)


    #check if signature is ok
    try:
        peer_pub = conn.recv()
        sig = conn.recv()
        peer_pub_DSA.verify(sig,peer_pub,ec.ECDSA(hashes.SHA256()))
        print("ok ECDSA")
    except InvalidSignature:
        print("fail ECDSA")


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

    shared_key = pk_DH.exchange(ec.ECDH(),peer_pub_key)

    #confirmation
```

```
            my_tag = hashf(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')

            #erase data
            pk_DH = None
            pub_DH = None
            peer_pub = None
            peer_pub_key = None
            shared_key = None
            my_tag = None
            peer_tag = None
```

In [4]:
```
from multiprocessing import Pipe, Process

class BiConn(object):
    def __init__(self,left,right,timeout=None):
        """
        left : a função que vei ligar ao lado esquerdo do Pipe
        right: a função que vai ligar ao outro lado
        timeout: (opcional) numero de segundos que aguarda pela terminação do processo
        """
        left_end, right_end = Pipe()
        self.timeout=timeout
        self.lproc = Process(target=left, args=(left_end,))       # os processos ligad
        self.rproc = Process(target=right, args=(right_end,))
        self.left  = lambda : left(left_end)                       # as funções ligada
        self.right = lambda : right(right_end)

    def auto(self, proc=None):
        if proc == None:             # corre os dois processos independentes
            self.lproc.start()
            self.rproc.start()
            self.lproc.join(self.timeout)
            self.rproc.join(self.timeout)
        else:                        # corre só o processo passado como parâmetro
            proc.start(); proc.join()

    def manual(self):   #  corre as duas funções no contexto de um mesmo processo Pyth
        self.left()
        self.right()
```

In [5]:
```
import os, io
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
```

```
message_size = 2**10

def Emitter(conn):

    #Acordo de chaves DH e assinatura DSA
    key = ECDHDSA(conn)

    # Cria um input stream com um número grande de bytes aleatórios
    # inputs = io.BytesIO(os.urandom(message_size))
    inputs = io.BytesIO(bytes('1'*message_size,'utf-8'))

    # tweak
    tweak = os.urandom(16)

    #Cipher
    cipher = Cipher(algorithms.AES(key), modes.XTS(tweak),
                    backend=default_backend()).encryptor()


    #HMAC
    mac = my_mac(key)

    #envia pela conecção o tweak
    conn.send(tweak)
    #define um buffer para ondevão ser lidos, sucessivamente, os vários blocos do inpu
    buffer = bytearray(32)

    # 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))


    print(key)


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

    key = None
```

```
In [6]: def Receiver(conn):

            #Acordo de chaves DH e assinatura DSA
            key = ECDHDSA(conn)

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

            #tweak
            tweak = conn.recv()

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

            #HMAC
            mac = my_mac(key)

            # operar a cifra: ler da conecção um bloco, decifrá-lo e escrever o resultado no '
            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("erro autenticação do ciphertext: {}".format(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 [7]: BiConn(Emitter,Receiver,timeout=30).auto()

ok ECDSA
ok ECDSA
```

```
OK ECDH
b'\xe4\x9c8O\x8d\xf52\xf51\x83x`\x07\xf0r\x15\x95:\xe3\x16F\xe8*Ab\xe8\xd0\xf4\x03%C\xfa'
OK ECDH
b'1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
```

In [ ]: