TP01_1

March 16, 2020

GRUPO 16 - Exercício 1

Para a realização desse exercício, foi utilizado os módulos *BiConn.py* e *Auxs.py* encontrados no material da disciplina e foi utilizado como base o script disponibilizado também no material.

Após gerado os parâmetros para o acordo de chave **Diffie-Hellman** e para o esquema de assinatura **DSA**, foi criada uma função para gerar *nonces* aleatórios e garantir que estes *nonces* não tenham sido utilizados anteriormente e outra função para autenticar as chaves geradas utilizando **HMAC**.

[1]:
```
import os
import io
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, hmac, serialization
from cryptography.hazmat.primitives.asymmetric import dh, dsa
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.exceptions import *
from BiConn import BiConn
from Auxs import hashs

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

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

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

Para cifrar a comunicação entre os agentes, foi implementada a cifra **AES** no modo **Cipher Feedback** (CFB), por ter sido considerada a mais segura contra ataques ao *initialization vector* (IV). Pois o primeiro bloco cifrado no **CFB** é definido por $C_0 = E_k(IV)\ P_0$ e mesmo se um atacante tiver acesso ao IV antecipadamente, tudo que ele saberá será o valor do primeiro bloco e não poderá interferir nos blocos subsequentes, desde que o IV não seja utilizado mais de uma vez, o que foi assegurado com a implementação do *nonce*.

[2]:
```
nonce_list = list()
```

```
# Gerar um nonce
def get_nonce():

    nonce = os.urandom(16)

    while nonce in nonce_list:
        nonce = os.urandom(16)
    nonce_list.append(nonce)

    return nonce


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

[3]:
```
def dh_dsa(conn):
    # Gerar chaves DH
    pk_dh = parameters_dh.generate_private_key()
    pub_dh = pk_dh.public_key().public_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PublicFormat.SubjectPublicKeyInfo)

    # Gerar chaves DSA
    pk_dsa = parameters_dsa.generate_private_key()
    pub_dsa = pk_dsa.public_key().public_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PublicFormat.SubjectPublicKeyInfo)

    # Enviar a chave pública DSA
    conn.send(pub_dsa)

    # Cálculo da assinatura DSA
    signature_dsa = pk_dsa.sign(pub_dh, hashes.SHA256())

    # Receber chave pública DSA
    peer_pub_dsa = serialization.load_pem_public_key(
                    conn.recv(),
                    backend=default_backend())

    # Enviar chave pública DH e assinatura DSA
    conn.send(pub_dh)
    conn.send(signature_dsa)

    # Verificar assinatura DSA
    try:
        peer_pub_dh = conn.recv()
```

```
        peer_signature_dsa = conn.recv()
        peer_pub_dsa.verify(peer_signature_dsa, peer_pub_dh, hashes.SHA256())
        print('DSA_OK')
    except InvalidSignature:
        print('DSA_FAIL')

    # shared_key calculation
    peer_pub_key = serialization.load_pem_public_key(
                    peer_pub_dh,
                    backend=default_backend())
    shared_key = pk_dh.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('DH_OK')
        return my_tag
    else:
        print('DH_FAIL')
```

Para realizar a comunicação, primeiramente o agente **Emitter** gera uma chave privada, a respetica chave pública e envia-a ao agente **Bob** que procede de forma análoga. Seguidamente ambos os agentes computam a chave partilhada e usam um MAC para confirmar a autenticidade da chave e então começar a comunicação utilizando a cifra **AES** no modo **CFB** citada acima.

[4]:
```
message_size = 2 ** 10


def Emitter(conn):

    key = dh_dsa(conn)
    # Mensagem a ser enviada
    inputs = io.BytesIO(bytes('1' * message_size, 'utf-8'))

    # nonce para a cifragem
    nonce = get_nonce()

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

    # HMAC
    mac = my_mac(key)

    conn.send(nonce)  # Envio do nonce
```

```
    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 conexão
```

[5]:
```
def Receiver(conn):
    # Acordo de chaves DH e assinatura DSA
    key = dh_dsa(conn)

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

    # Recebe o nonce
    nonce = conn.recv()

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

    # HMAC
    mac = my_mac(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:
                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: {}".
↪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 conexão
```

[6]:
```
BiConn(Emitter, Receiver, timeout=30).auto()
```

```
DSA_OK
DSA_OK
DH_OK
DH_OK
b'11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111'
```