TP01_2

March 16, 2020

GRUPO 16 Exercício 2

Para a realização deste 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, tal como o exercício 1 deste trabalho. Inicialmente apenas é reutilizada a a o esquema de assinatura ECDSA, foi reutilizada a função para gerar nonces aleatórios e garantir que estes nonces não tenham sido utilizados anteriormente. É inicializada também uma curva Eliptica.

[32]:
```
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, ec
from cryptography.hazmat.primitives.ciphers.aead import ChaCha20Poly1305
from cryptography.exceptions import *
from BiConn import BiConn
from Auxs import hashs

#lista de nonces
nonce_list = list()


# Gerar um nonce
def get_nonce(b):

    nonce = os.urandom(b)

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

    return nonce

#gerar uma curva
default_curve = ec.SECP256R1  # curva
```

São gerados os parâmetros para o acordo de chave Diffie-Hellman através de Curvas Elipticas(ECDH), e a autenticação dos agentes através do esquema de assinaturas ECDSA.

[33]:
```
def ECDH(conn):
    # agreement
    pk_ecdh = ec.generate_private_key(default_curve, default_backend())  # ao
↪gerar a chave privada, recebe como argumento a curva definida
    pub_ecdh = pk_ecdh.public_key().public_bytes(  #gera a chave publica
↪através da privada e transforma-a de objecto para bytes
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo)

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

    # envia a chave pública
    conn.send(pub_dsa)

    # cálculo da assinatura
    signature = pk_ecdsa.sign(pub_ecdh, ec.ECDSA(hashes.SHA256()))  # ECDSA
↪recebe como argumento a hash256

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

    conn.send(pub_ecdh)
    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_ecdh.exchange(ec.ECDH(), peer_pub_key)  # em vez de se
↪trocar apenas 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 ECDSA')
        return my_tag
    else:
        print('FAIL ECDSA')
```

De forma a encriptar a comunicação entre o emissor e recetor foi utilizada a cifra **ChaCha20Poly1305** tal como indicada no enunciado, sendo que esta cifra tem a particularidade de ter incorporada um MAC para garantir autenticidade da mensagem, e esse MAC é o **Poly1305** e a cifra é **ChaCha20**, mas neste caso a biblioteca cryptography permite a utilização de uma combinação implicita de ambos, de notar que esta é umas das melhores alternativas ao **AES**, utlizado no exercício anterior.

[34]:
```
message_size = 2 ** 10


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

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

    chacha = ChaCha20Poly1305(key)
    aad = b"HELLOWORLD"

    buffer = bytearray(32)  # Buffer onde vão ser lidos os blocos

    # lê, cifra e envia sucessivos blocos do input
    c = 0
    try:
        while inputs.readinto(buffer):
            nonce = get_nonce(12)
            conn.send(nonce)
            cipher = chacha.encrypt(nonce, bytes(buffer), aad)
            conn.send(cipher)

        conn.send(nonce)
        conn.send(b'')
    except Exception as err:
        print("Erro no emissor: {0}".format(err))

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

[35]:
```python
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()

    chacha = ChaCha20Poly1305(key)
    aad = b"HELLOWORLD"
    # operar a cifra: ler da conexão um bloco, autenticá-lo, decifrá-lo e␣
↪escrever o resultado no 'stream' de output
    try:
        while True:
            try:
                nonce = conn.recv()
                buffer = conn.recv()
                if not buffer:
                    outputs.write(ct)
                    break
                ciphertext = bytes(buffer)
                ct = chacha.decrypt(nonce, ciphertext, aad)
                outputs.write(ct)

            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
```

[36]:
```python
BiConn(Emitter, Receiver, timeout=30).auto()
```

```
ok ECDH
ok ECDH
OK ECDSA
OK ECDSA
b'1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
```

```
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111'
```