# TP01_P1EC

April 4, 2021

# Solução Para o Trabalho Prático 01

## Problema 01:

4. Criação de uma versão do esquema anterior, que usa curvas elípticas substituindo o DH pelo ECDH e o DSA pelo ECDSA.

Abaixo vamos distinguir as diferenças deste documento para o outro, são muito poucas e subtís.

[1]:
```
import io, os, time
from multiprocessing import set_start_method, Pipe, Process
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.hazmat.primitives.asymmetric import dh, dsa, ec
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.ciphers import (Cipher, algorithms, modes)
import cryptography.exceptions

buffer_size = 4096
msg_size = 1024
```

**Implementação da Função Derive** Nada de novo aqui.

[2]:
```
def derive(shared_key):
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',
        backend=default_backend(),
    ).derive(shared_key)
    return derived_key
```

**Implementação das funções de Encrypt e Decrypt** Nada de novo aqui.

[3]:
```
def encrypt(key, iv, plaintext):
    # inicialização
    encryptor = Cipher(
        algorithms.AES(key),
        modes.CTR(iv),
```

```
        backend=default_backend(),
    ).encryptor()

    # cifrar a mensagem
    return encryptor.update(plaintext)
```

[4]:
```
def decrypt(key, iv, ciphertext):
    # inicialização
    decryptor = Cipher(
        algorithms.AES(key),
        modes.CTR(iv),
        backend=default_backend(),
    ).decryptor()

    # decifrar a mensagem
    return decryptor.update(ciphertext)
```

**Assinatura e Validação ECDSA** Tanto na assinatura como na validação usamos ec.ECDSA.

[5]:
```
def sign_message(key, message):
    return key.sign(message, ec.ECDSA(hashes.SHA256()))
```

[6]:
```
def validate_signature(key, message, signature):
    try:
        key.verify(signature, message, ec.ECDSA(hashes.SHA256()))
    except InvalidSignature:
        return False
    return True
```

**Implementação do HMAC** Nada de novo aqui.

[7]:
```
def get_hmac(key, message):
    h = hmac.HMAC(key, hashes.SHA256(), backend=default_backend())
    h.update(message)
    return h.finalize()
```

[8]:
```
def validate_hmac(key, message, signature):
    try:
        h = hmac.HMAC(key, hashes.SHA256(), backend=default_backend())
        h.update(message)
        h.verify(signature)
    except InvalidSignature:
        return False
    return True
```

**Preparação da Mensagagem** Nada de novo aqui.

[9]:
```python
def prepare_bundle(key, message, dsa_key):
    # gerar o nonce
    iv = os.urandom(16)
    # obtenção do criptograma
    ct = encrypt(key, iv, message)
    # assinatura do criptograma
    signature = sign_message(dsa_key, message)
    # junção do iv, assinatura e criptograma
    pre_bundle = len(signature).to_bytes(1,'little') + iv + signature + ct
    # 'prepending' do hmac
    bundle = get_hmac(key, pre_bundle) + pre_bundle
    return bundle
```

**Execução do Emitter:** Única diferença ao produzir a chave partilhada, usamos ec.ECDH.

[10]:
```python
def execucaoemitter(conn, private_key, receiver_public_key, private_dsa_key,␣
↪receiver_dsa_key):
    # gerar a chave combinada
    shared_key = private_key.exchange(ec.ECDH(), receiver_public_key)
    # derivar a chave
    derived_key = derive(shared_key)

    while True:
        msg = input('Emitter: ').encode()
        if len(msg) > msg_size:
            break
        bundle = prepare_bundle(derived_key, msg, private_dsa_key)
        conn.send(bundle)
        try:
            buffer = bytearray(buffer_size)
            buffer = conn.recv()
            mac = buffer[0:32]
            pre_bundle = buffer[32:]
            if validate_hmac(derived_key, pre_bundle, mac):
                sig_len = pre_bundle[0]
                iv = pre_bundle[1:17]
                signature = pre_bundle[17:17 + sig_len]
                ct = pre_bundle[17 + sig_len:]
                plain_text = decrypt(derived_key, iv, ct)
                if validate_signature(receiver_dsa_key, plain_text, signature):
                    print('Emitter got: ', plain_text.decode())
                else:
                    print('Emitter got bad signature!')
                    break
            else:
                print('Emitter got bad MAC!')
                break
```

```
        except EOFError:
            break
    conn.close()
    inputs.close()
```

**Execução do Receiver:**

[11]:
```
def execucaoreceiver(conn, private_key, emitter_public_key, private_dsa_key,␣
↪emitter_dsa_key):
    # gerar a chave combinada
    shared_key = private_key.exchange(ec.ECDH(), emitter_public_key)
    # derivar a chave
    derived_key = derive(shared_key)

    while True:
        try:
            buffer = bytearray(buffer_size)
            buffer = conn.recv()
            mac = buffer[0:32]
            pre_bundle = buffer[32:]
            if validate_hmac(derived_key, pre_bundle, mac):
                iv = pre_bundle[1:17]
                sig_len = pre_bundle[0]
                signature = pre_bundle[17:17 + sig_len]
                ct = pre_bundle[17 + sig_len:]
                plain_text = decrypt(derived_key, iv, ct)
                if validate_signature(emitter_dsa_key, plain_text, signature):
                    print('Receiver got: ', plain_text.decode())
                else:
                    print('Receiver got bad signature!')
                    break
            else:
                print('Receiver got bad MAC!')
                break
        except EOFError:
            break
        msg = "ok"
        msg = msg.encode()
        if len(msg) > msg_size:
            break
        bundle = prepare_bundle(derived_key, msg, private_dsa_key)
        conn.send(bundle)
    conn.close()
```

**Inicialização do Processo** A diferença aqui é que não há um gerador de parâmetros e em cada par de chaves usamos ec.SECP384R1.

[ ]:
```
try:
    set_start_method('fork')     ## a alteração principal
except:
    pass

receiver_conn, emitter_conn = Pipe()

# par de chaves do emitter
emitter_private_key = ec.generate_private_key(ec.SECP384R1(),␣
 ↪backend=default_backend())
# chave publica do emitter
emitter_public_key = emitter_private_key.public_key()

# par de chaves do receiver
receiver_private_key = ec.generate_private_key(ec.SECP384R1(),␣
 ↪backend=default_backend())
# chave publica do receiver
receiver_public_key = receiver_private_key.public_key()

# par de chaves dsa do emitter
emitter_private_dsa_key = ec.generate_private_key(ec.SECP384R1(),␣
 ↪backend=default_backend())
# chave publica dsa do emitter
emitter_public_dsa_key = emitter_private_dsa_key.public_key()

# par de chaves dsa do receiver
receiver_private_dsa_key = ec.generate_private_key(ec.SECP384R1(),␣
 ↪backend=default_backend())
# chave publica dsa do receiver
receiver_public_dsa_key = receiver_private_dsa_key.public_key()

q = Process(target=execucaoreceiver, args=(receiver_conn, receiver_private_key,␣
 ↪emitter_public_key, receiver_private_dsa_key, emitter_public_dsa_key,))

q.start()
execucaoemitter(emitter_conn, emitter_private_key, receiver_public_key,␣
 ↪emitter_private_dsa_key, receiver_public_dsa_key)

q.join(timeout=120)
```