TP1 - Estruturas Criptográficas

Elementos do grupo 4:

- André Morais, A83899
- Tiago Magalhães, A84485

Parte 1

Primeiramente foi criada um classe denominada crypto, que tem como objetivo responder aos requisitos das alíneas a e b do enunciado, esta classe guarda um estado acerca dos nonces, de modo a evitar que ocorram repetições, para cifrar e autenticar os criptogramas utilizamos o modo Encrypt-then-Mac.

Crypto.py

In [13]:
```
import os

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, mod
from cryptography.hazmat.primitives import padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives import hmac
import cryptography.exceptions

AES_BLOCK_LEN_BITS = 128
AES_BLOCK_LEN_BYTES = 16
MAC_LEN = 32 # BYTES

class Crypto:
    def __init__(self, key):
        self.key = key
        self.nounce_list = []

    def nounce_generator(self, size):
        nounce = os.urandom(size)
        while nounce in self.nounce_list:
            nounce = os.urandom(size)
        self.nounce_list.append(nounce)
        return nounce
```

Funções para cifrar e decifrar com a cifra AES no modo CBC.

In [14]:

```
    def encode(self, msg):
        padder = padding.PKCS7(AES_BLOCK_LEN_BITS).padder()
        # Adiciona padding ao último bloco de bytes da mensagem de modo a 
        padded_data = padder.update(msg) + padder.finalize()
        iv = self.nounce_generator(AES_BLOCK_LEN_BYTES)
        cipher = Cipher(algorithms.AES(self.key), modes.CBC(iv))
        encryptor = cipher.encryptor()
        ct = encryptor.update(padded_data) + encryptor.finalize()
        return iv + ct

    def decode(self, msg):
        iv, ct = msg[:AES_BLOCK_LEN_BYTES], msg[AES_BLOCK_LEN_BYTES:]
        cipher = Cipher(algorithms.AES(self.key), modes.CBC(iv))
        decryptor = cipher.decryptor()
        pt = decryptor.update(ct) + decryptor.finalize()
        unpadder = padding.PKCS7(AES_BLOCK_LEN_BITS).unpadder()
        pt = unpadder.update(pt) + unpadder.finalize()
        return pt
```

Funções para autenticar o criptograma com HMAC e para verificar a autenticação do mesmo.

In [15]:

```
    def authenticate_hmac(self, msg):
        h = hmac.HMAC(self.key, hashes.SHA256())
        h.update(msg)
        digest = h.finalize()
        return digest

    def verify_hmac(self, signature, msg):
        h = hmac.HMAC(self.key, hashes.SHA256())
        # Gera digest para a mensagem.
        h.update(msg)
        try :
            # Verifica se o digest gerado acima é igual ao digest recebido
            h.verify(signature)
            print("Criptograma autenticado.")
            return True
        except cryptography.exceptions.InvalidSignature:
            print("Falha na autenticação do criptograma.")
            return False
```

Função para autenticar e cifrar texto limpo (Encrypt-then-Mac).

Primeiramente é cifrada a mensagem, após isto é gerada uma tag com a mesangem cifrada.

Retorna MAC || IV || CT.

In [16]:

```
    def etm_enc(self, msg):
        # Cifra a mensagem: E(m)
        c = self.encode(msg)
        iv, ct = c[:AES_BLOCK_LEN_BYTES], c[AES_BLOCK_LEN_BYTES:]
        # Digest do texto cifrado hmac(E(m)).
        digest = self.authenticate_hmac(ct)
        return digest + iv + ct
```

Função que verifica autenticação e decifra criptograma, isto é, realiza a operação inversa à anterior.

In [17]:

```
    def etm_dec(self, msg):
        # Retira mac da msg = digest + iv + ct.
        sig, c = msg[:MAC_LEN], msg[MAC_LEN:]
        ct = c[AES_BLOCK_LEN_BYTES:]
        # Valida mac / mac etm é dado por mac(E(m)).
        self.verify_hmac(sig, ct)
        # Decifra mensagem.
        pt = self.decode(c)
        return pt
```

Emmiter

Criação da classe Emmiter que tem como função cifrar a mensagem e construir o criptograma.
Esta classe recebe como parâmetros uma conexão que representa o pipe por onde esta irá enviar e receber informação do Receiver e também se pretende que o protocolo de acordo de chaves use a versão com curvas elípticas.

In [18]:

```
from multiprocessing import Process
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import dsa
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.serialization import Encoding, PublicF
from cryptography.hazmat.primitives.serialization import PrivateFormat
from cryptography.hazmat.primitives.serialization import NoEncryption
from cryptography.hazmat.primitives.serialization import load_der_private_
from cryptography.hazmat.primitives.serialization import load_pem_private_
from cryptography.hazmat.primitives.serialization import load_pem_public_k
from cryptography.hazmat.primitives.serialization import load_der_public_k
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import ec

# RFC 3526's parameters.
p = 0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020BI
g = 2
params_numbers = dh.DHParameterNumbers(p,g)
parameters = params_numbers.parameters()

class Emitter(Process):
    def __init__(self, conn, elicurves = False):
        super(Emitter, self).__init__()
        # Conexão pipe.
        self.conn = conn
        # Módulo Crypto.
        self.crypto = None
        # Chave simétrica.
        self.key = None

        # Uso do protocolo com Curvas Elípticas.
        self.ec = elicurves

        # Uso de curvas elípticas DSA.
        if self.ec == True:
            # Chave privada assimétrica.
            self.private_key = ec.generate_private_key(ec.SECP384R1())
            # Chave pública assimétrica.
            self.public_key = self.private_key.public_key().public_bytes(
                                    encoding=serialization.Encoding.DER,
                                    format=serialization.PublicFormat.Subj
        # Uso DSA.
        else:
            # Chave privada assimétrica.
            self.private_key = dsa.generate_private_key(key_size=1024)
            # Chave pública assimétrica.
            self.public_key = self.private_key.public_key().public_bytes(
                                    encoding = serialization.Encoding.DER,
                                    format = serialization.PublicFormat.Sul
```

Função que estabelece o protocolo station to station, que utiliza o protocolo de acordo de chaves Diffie-Hellman(DH) e esquema de assinaturas DSA.
Esquema do protocolo:

- Receiver envia para o emitter - a chave privada DH
- Emmiter envia para o receiver - a chave privada DH, Assinatura cifrada com a mensagem : chave privada dh do emitter || chave privada dh do receiver, chave pública

emitter.

- Receiver envia para o emitter - Assinatura cifrada com a mensagem : chave privada dh do emitter || chave privada dh do receiver, chave pública receiver

In [19]:

```
    def sts(self):
        if self.ec == True:
            dh_priv_key = ec.generate_private_key(ec.SECP384R1())
        else:
            # Gera g^y.
            dh_priv_key = parameters.generate_private_key()

        # g^y em bytes para ser transmido no pipe.
        dh_priv_key_as_bytes = dh_priv_key.private_bytes(Encoding.DER, Priv
        dh_pub_key_as_bytes = dh_priv_key.public_key().public_bytes(Encodi

        # Recebe do receiver g^x.
        dh_peer_pub_key_as_bytes = self.conn.recv()
        # g^x serializado.
        dh_peer_pub_key = load_der_public_key(dh_peer_pub_key_as_bytes, No

        if self.ec == True:
            shared_key = dh_priv_key.exchange(ec.ECDH(), dh_peer_pub_key)
        else:
            # Chave partilhada (g^y^x)
            shared_key = dh_priv_key.exchange(dh_peer_pub_key)

        # Deriva a chave
        derived_key = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=b'handshake data',
        ).derive(shared_key)

        self.key = derived_key

        self.crypto = Crypto(self.key)
        separador = b"\r\n\r\n"

        # Assinatura
        sig = self.sign(dh_pub_key_as_bytes + dh_peer_pub_key_as_bytes)
        # Mensagem a ser enviada ao receiver : g^y + Enc[S(g^y,g^x)] + Chav
        msg = dh_pub_key_as_bytes + separador + self.crypto.etm_enc(sig) +

        # Envia msg ao receiver
        self.conn.send(msg)

        # Recebe do receiver signature e chave pública.
        mensagem = self.conn.recv()

        # Separa a mensagem em [[E(S(g^x,g^y))],[Certificado]].
        msg_splitted = mensagem.split(sep=b"\r\n\r\n")

        # Chave pública do emmiter.
        public_key_receiver_as_bytes = msg_splitted[1] # Chave pública do e
        public_key_receiver = load_der_public_key(public_key_receiver_as_by
        # Retira assinatura do cliente.
        signature = self.crypto.etm_dec(msg_splitted[0])

        try:
            # Verifica assinatura do emitter.
            self.verify(public_key_receiver, dh_peer_pub_key_as_bytes +  dh
            print("Assinatura do Receiver válida.")
        except:
            print("Assinatura do Receiver inválida.")
```

Função usada para assinar a mensagem com a chave privada, serve para provar/autenticar que é o receiver que está a mandar a mensagem(não repúdio), uma vez que só o receiver tem esta chave privada, a mensagem ao ser as chaves DH tem como objetivo manter a integridade destas, já que um atacante no meio da conexão as poderia alterar, após assinatura, o emitter pode verificar a assinatura garantindo o não repúdio e integridade da mensagem.

In [20]:
```
    def sign(self, msg):
            if self.ec == True:
                signature = self.private_key.sign(
                msg,
                ec.ECDSA(hashes.SHA256()))
            else:
                signature = self.private_key.sign(
                msg,
                hashes.SHA256())

            return signature
```

Função usada para verificar se a assinatura do receiver foi mesmo realizada por este, já que ao verificarmos com a chave pública temos uma maior segurança contra ataques man-in-the-middle, com a verificação garantimos que a mensagem não foi modificada e foi gerada com a chave privada do receiver, dado que só a chave pública do receiver a pode verificar.

In [21]:
```
    def verify(self, public_key, msg, sig):
        if self.ec == True:
            public_key.verify(
            sig,
            msg,
            ec.ECDSA(hashes.SHA256()))
        else:
            public_key.verify(
            sig,
            msg,
            hashes.SHA256())
```

Funções para representar envio de mensagem e para definir comportamento do processo(run).

In [22]:
```
    def send_msg(self, msg):
        msg = self.crypto.etm_enc(msg)
        self.conn.send(msg)

    def run(self):
        self.sts()
        msg = b"Aqui vai mensagem!"
        print("Mensagem a enviar:" + msg.decode('utf-8') )
        self.send_msg(msg)
```

Receiver

Criação da classe Receiver que tem como função como decifrar o criptograma e construir a mensagem. Segue a mesma estrutura que a classe emitter.

In [23]:

```
from multiprocessing import Process
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import dsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.serialization import Encoding, PublicF
from cryptography.hazmat.primitives.serialization import PrivateFormat
from cryptography.hazmat.primitives.serialization import NoEncryption
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.serialization import load_der_private_
from cryptography.hazmat.primitives.serialization import load_pem_private_
from cryptography.hazmat.primitives.serialization import load_pem_public_k
from cryptography.hazmat.primitives.serialization import load_der_public_k
from cryptography.hazmat.primitives.asymmetric import ec


p = 0xFFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74020B
g = 2
params_numbers = dh.DHParameterNumbers(p,g)
parameters = params_numbers.parameters()

class Receiver(Process):
    def __init__(self, conn, elicurves = False):
        super(Receiver, self).__init__()
        # Conexão pipe
        self.conn = conn
        # Módulo Crypto
        self.crypto = None
        # Chave simétrica
        self.key = None

        # Uso do protocolo com Curvas Elípticas
        self.ec = elicurves

        # Uso de curvas elípticas
        if self.ec == True :
            # Chave privada assimétrica
            self.private_key = ec.generate_private_key(ec.SECP384R1())
            # Chave pública assimétrica
            self.public_key = self.private_key.public_key().public_bytes(
                                    encoding=serialization.Encoding.DER,
                                    format=serialization.PublicFormat.Subj
        else:
            # Chave privada assimétrica
            self.private_key = dsa.generate_private_key(key_size=1024)
            # Chave pública assimétrica
            self.public_key = self.private_key.public_key().public_bytes(
                                    encoding = serialization.Encoding.DER,
                                    format = serialization.PublicFormat.Sul

    def sts(self):
        if self.ec == True:
            dh_priv_key = ec.generate_private_key(ec.SECP384R1())
        else:
            # Gerar (g^x).
            dh_priv_key = parameters.generate_private_key()

        # g^x em bytes para poder ser transmitido pelo pipe.
        dh_priv_key_as_bytes = dh_priv_key.private_bytes(Encoding.DER, Priv
        dh_pub_key_as_bytes = dh_priv_key.public_key().public_bytes(Encodi

        # Envia para o emitter (g^x).
```

```
        # g y em bytes.
        dh_peer_pub_key_as_bytes = msg_splitted[0]
        # g^y serializado.
        dh_peer_pub_key = load_der_public_key(dh_peer_pub_key_as_bytes, No

        if self.ec == True:
            shared_key = dh_priv_key.exchange(ec.ECDH(), dh_peer_pub_key)
        else:
            # Chave partilhada (g^y^x)
            shared_key = dh_priv_key.exchange(dh_peer_pub_key)

        # Deriva a chave partilhada.
        derived_key = HKDF(
            algorithm=hashes.SHA256(),
            length=32,
            salt=None,
            info=b'handshake data',
        ).derive(shared_key)

        self.key = derived_key

        self.crypto = Crypto(self.key)

        # Chave pública do emitter.
        public_key_emitter_as_bytes = msg_splitted[2]
        # Serializa chave.
        public_key_emitter = load_der_public_key(public_key_emitter_as_byt

        # Assinatura do emitter.
        signature = self.crypto.etm_dec(msg_splitted[1])

        try:
            # Verifica assinatura do emitter.
            self.verify(public_key_emitter, dh_peer_pub_key_as_bytes + dh_
            print("Assinatura do Emitter válida.")
        except:
            print("Assinatura do Emitter inválida")

        separador = b"\r\n\r\n"

        # Mensagem com assinatura e chave pública do receiver.
        sig = self.sign(dh_pub_key_as_bytes + dh_peer_pub_key_as_bytes)
        mensagem = self.crypto.etm_enc(sig) + separador + self.public_key
        # Envia mensagem ao emitter.
        self.conn.send(mensagem)

    def sign(self, msg):
        if self.ec == True:
            signature = self.private_key.sign(
            msg,
            ec.ECDSA(hashes.SHA256()))
        else:
            signature = self.private_key.sign(
            msg,
            hashes.SHA256())

        return signature
```

```
        msg = self.conn.recv()
        print("Mensagem recebida: " + self.crypto.etm_dec(msg).decode('utf

    def run(self):
        self.sts()
        self.read_msg()
```

Main

Reprensenta um sessão síncrona entre os dois agentes (o Emitter e o Receiver).

In [24]:

```
from multiprocessing import set_start_method, Pipe, Process
from emitter import Emitter
from receiver import Receiver

def main():

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

    # Cria pipe.
    parent_conn, child_conn = Pipe()
    # Cria novos processos.
    e = Emitter(parent_conn)
    r = Receiver(child_conn)
    # Corre os processos.
    e.start()
    r.start()
    # Espera que os processos terminem.
    e.join()
    r.join()

    print("---------Com Curvas Elípticas-------------")
    # Cria pipe.
    parent_conn, child_conn = Pipe()
    # Cria novos processos.
    e = Emitter(parent_conn,True)
    r = Receiver(child_conn,True)
    # Corre os processos.
    e.start()
    r.start()
    # Espera que os processos terminem.
    e.join()
    r.join()

if __name__ == "__main__":
    main()
```

```
Criptograma autenticado.
Assinatura do Emitter válida.
Criptograma autenticado.
Assinatura do Receiver válida.
Mensagem a enviar:Aqui vai mensagem!
Criptograma autenticado.
Mensagem recebida: Aqui vai mensagem!
---------Com Curvas Elípticas-------------
Criptograma autenticado.
Assinatura do Emitter válida.
Criptograma autenticado.
Assinatura do Receiver válida.
Mensagem a enviar:Aqui vai mensagem!
Criptograma autenticado.
Mensagem recebida: Aqui vai mensagem!
```

In [ ]: