Trabalho Prático 0 - Problema 1 - Grupo 18

Introdução ao problema

Neste problema foi-nos indicado que deveríamos ser capazes de criar uma comunicação privada assíncrona entre um emissor e um recetor, com as seguintes características:

- Autenticação do criptograma e dos metadados, usando uma cifra simétrica num modo HMAC, seguro contra ataques aos "nounces";
- Os "nounces" devem ser gerados por um gerador pseudo aleatório construído por uma função de hash em modo XOF;
- O par de chaves para cifra e autenticação é acordado entre agentes usando o protocolo DH com autenticação dos agentes usando assinaturas DSA.

Resolução do problema

Gerar a chave partilhada

Nesta função são recebidos dois valores: a chave pública recebida e a própria chave privada e é retornado um valor referente à chave partilhada correspondente.

```
from cryptography.hazmat.primitives.asymmetric import dh

# Generate some parameters. These can be reused
parameters = dh.generate_parameters(generator=2, key_size=2048)

def generate_shared_key(received_public_key, private_key):

    # Generate the shared key
    shared_key = private_key.exchange(received_public_key)

    return shared_key
```

Cifragem

O processo de cifragem começa por receber quatro valores: o texto a ser cifrado, os metadados correspondentes, a chave de cifragem e a chave de autenticação.

Inicialmente, é gerado um "nounce" pseudo aleatoriamente, que permite garante unicidade na validade desta cifra. Depois, é gerado um "encryptor", usando a chave de cifragem e o "nounce" obtido, que encripta o texto recebido. De seguida, é o "nounce", a tag do "encryptor" e o texto cifrado são encapsulado numa mensagem, que será autenticada, através da chave de autenticação. Esta mensagem e o valor de autenticação gerado são returnados, prontos para serem enviados.

Existe, ainda, uma função auxiliar 'generateHMAC' que é usada para gerar o valor de autenticação, sendo necessário fornecer-lhe uma mensagem e a chave de autenticação.

```
import secrets
import hashlib
import os
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms,
modes
from cryptography.hazmat.primitives import hmac
from pickle import dumps

def generateHMAC(message, hmac_key):

    h = hmac.HMAC(hmac_key, hashes.SHA256())
    h.update(dumps(message))

    return h.finalize()

def encrypt(plaintext, metadata, cipher_key, hmac_key):

    # Generate a pseudo random value and use it as a "nounce"
    nounce = secrets.token_bytes(16)

    # Create an AES-GCM Cipher with given cipher_key and generated
"nounce"
    encryptor = Cipher(algorithms.AES(cipher_key),
modes.GCM(nounce)).encryptor()

    # Authenticate metadata
    encryptor.authenticate_additional_data(metadata)

    # Encrypt plaintext
    ciphertext = encryptor.update(plaintext.encode()) +
encryptor.finalize()

    # Message being sent
    message = { 'nounce': nounce, 'tag': encryptor.tag, 'ciphertext':
ciphertext }

    # Generate HMAC authentication
    hmac_auth = generateHMAC(message, hmac_key)

    # Return encrypted package
    return { 'message': message, 'hmac_auth': hmac_auth }
```

Decifragem

O processo de decifragem recebe três valores: o pacote recebido e as duas chaves partilhadas, a de cifragem e a de autenticação.

Primeiramente, é feito o "parsing" do pacote, obtendo a mensagem transmitida, o valor de autenticação e os metadados correspondentes. De forma a garantir a autenticidade da mensagem, é calculado o valor de autenticação esperado e comparado com o valor

recebido. Se os valores forem iguais, prossegu-se para o decifragem da mensagem, caso contrário, o pacote é rejeitado.

Tendo sido garantida a autenticidade do pacote, este é então "parsed". Depois, procede-se à decifragem do texto cifrado, através de um "decryptor", e é returnado o texto fonte.

```
def decrypt(package, cipher_key, hmac_key):

    # Get parameters from the received package
    message = package['message']
    hmac_auth = package['hmac_auth']
    metadata = package['metadata']

    # Authenticate received message with HMAC
    rcvd_hmac = generateHMAC(message, hmac_key)
    if (hmac_auth != rcvd_hmac):
        return 'ERROR: HMAC authentication failed.'

    # Parse received message, once it has been authenticated
    nounce = message['nounce']
    tag = message['tag']
    ciphertext = message['ciphertext']

    # Create an AES-GCM Cipher with received cipher_key, "nounce" and
encryptor tag
    decryptor = Cipher(algorithms.AES(cipher_key), modes.GCM(nounce,
tag)).decryptor()

    # Authenticate metadata
    decryptor.authenticate_additional_data(metadata)

    # Get plaintext
    try:
        plaintext = decryptor.update(ciphertext) +
decryptor.finalize()

        return plaintext.decode()

    except InvalidTag:
        print('ERROR: Invalid tag.')
```

Emissor

O emissor começa por gerar duas chaves privadas, uma associada à cifragem das mensagens e outras associada à autenticação das mesmas, usando parâmetros previamente definidos entre ambas as partes da comunicação. Depois, estas chaves são transformadas em chaves públicas e compartilhadas com o recetor, sendo que este também envia as suas chaves públicas. Assim que o emissor tem uma par de chave privada e pública, é capaz de gerar a chave partilhada correspondente, com a garantia que o recetor irá gerar uma chave igual. Desta forma, nunca é enviada uma chave privada, garantindo que não podem ser

roubadas por um atacante e sabemos que ambos os lados da comunicação são capazes de cifrar e decifrar as mensagens (usando a mesma chave, a chave partilhada).

De seguida, é criada uma mensagem, cifrada com a chave partilhada, usando o método descrito anteriormente e enviada para o recetor.

```
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.serialization import
load_pem_public_key

def Emitter(conn):

    # Generate private cipher and HMAC keys
    private_cipher_key = parameters.generate_private_key()
    private_hmac_key = parameters.generate_private_key()

    # Send and receive public cipher keys

conn.send(private_cipher_key.public_key().public_bytes(encoding=serial
ization.Encoding.PEM,
                                                       format=
serialization.PublicFormat.SubjectPublicKeyInfo))
    receiver_public_cipher_key = load_pem_public_key(conn.recv())

    # Send and receive public HMAC keys

conn.send(private_hmac_key.public_key().public_bytes(encoding=serializ
ation.Encoding.PEM,
                                                       format=
serialization.PublicFormat.SubjectPublicKeyInfo))
    receiver_public_hmac_key = load_pem_public_key(conn.recv())

    # Generate shared cipher and HMAC keys
    cipher_key =
private_cipher_key.exchange(receiver_public_cipher_key)
    shared_cipher_key = HKDF(algorithm=hashes.SHA256(),length=32,
                             salt=None,info=b'handshake
data').derive(cipher_key)

    hmac_key = private_hmac_key.exchange(receiver_public_hmac_key)
    shared_hmac_key = HKDF(algorithm=hashes.SHA256(),length=32,
                             salt=None,info=b'handshake
data').derive(hmac_key)


    # Send message to Receiver
    send_message = "Hello World!"
    print('[EMITTER]: Sent message \'' + send_message + '\'')
    metadata = os.urandom(16)
```

```
    package = encrypt(send_message, metadata, shared_cipher_key,
shared_hmac_key)
    package['metadata'] = metadata

    conn.send(package)
    print('[EMITTER]: Sent package.\n' + str(package) + '\n')

    conn.close()
```

Recetor

Tal como no emissor, o recetor começa por gerar duas chaves privadas, calcular as suas chaves públicas e enviá-las para o emissor. Recebendo as chaves públicas do emissor, é capaz de gerar as chaves partilhadas, tanto para a cifragem como para a autenticação.

Depois de possuir ambas as chaves partilhadas, está pronto para receber a mensagem vinda do emissor e decifrá-la usando o método de decifragem descrito anteriormente.

```
def Receiver(conn):

    # Generate private cipher and HMAC keys
    private_cipher_key = parameters.generate_private_key()
    private_hmac_key = parameters.generate_private_key()

    # Receive and send public cipher keys
    receiver_public_cipher_key = load_pem_public_key(conn.recv())

conn.send(private_cipher_key.public_key().public_bytes(encoding=serial
ization.Encoding.PEM,
                                                       format=
serialization.PublicFormat.SubjectPublicKeyInfo))

    # Receive and send public HMAC keys
    receiver_public_hmac_key = load_pem_public_key(conn.recv())

conn.send(private_hmac_key.public_key().public_bytes(encoding=serializ
ation.Encoding.PEM,
                                                     format=
serialization.PublicFormat.SubjectPublicKeyInfo))

    # Generate shared cipher and HMAC keys
    cipher_key =
private_cipher_key.exchange(receiver_public_cipher_key)
    shared_cipher_key = HKDF(algorithm=hashes.SHA256(),length=32,
                             salt=None,info=b'handshake
data').derive(cipher_key)

    hmac_key = private_hmac_key.exchange(receiver_public_hmac_key)
    shared_hmac_key = HKDF(algorithm=hashes.SHA256(),length=32,
```

```
                                salt=None,info=b'handshake
data').derive(hmac_key)


    # Receive package from Emitter
    package = conn.recv()
    print('[RECEIVER]: Received package.\n' + str(package) + '\n')

    plaintext = decrypt(package, shared_cipher_key, shared_hmac_key)
    print('[RECEIVER]: Message received: ' + plaintext)

    conn.close()
```

Execução

No pedaço de código seguinte, podemos observar a inicialização de uma classe que permite executar o emissor e o recetor em simultâneo e em processos diferentes, permitindo simular a comunicação entre duas entidades diferentes.

```
from multiprocessing import Process, Pipe

class Conn(object):

    def __init__(self, emitter, receiver):

        emitter_end, receiver_end = Pipe()

        self.eproc = Process(target=emitter, args=(emitter_end,))
        self.rproc = Process(target=receiver, args=(receiver_end,))

    def auto(self):
        self.eproc.start()
        self.rproc.start()
        self.eproc.join(None)
        self.rproc.join(None)

Conn = Conn(Emitter, Receiver)
Conn.auto()
```

```
[EMITTER]: Sent message 'Hello World!'
[RECEIVER]: Received package.
{'message': {'nounce': b'\xb9}\x89\xde2\x03ZJC\xf3\xc1\xfa\x11\xe3g\
x1d', 'tag': b'?\x18\xa8J\xdaI\xce\xf2\x9b\xa0\xca7\xaa\xe9\xab\x19',
'ciphertext': b'<\xf0\xf3e\x84\xa2\x85\xb7\x05\x9e`\x0b'},
'hmac_auth': b'\xac}\x1c~84\x0f\x9e\x83\xb3\xff\xec\x00+q5\xdf\xcfF\
xe3\x9ey\xed\xd4\x90p%^\x9e\xa8\xa9\xd1', 'metadata': b'\x9b\xf5\x14G\
xf1:({\n\xe5mb\x01\x15\xc5\xcf'}
[EMITTER]: Sent package.
{'message': {'nounce': b'\xb9}\x89\xde2\x03ZJC\xf3\xc1\xfa\x11\xe3g\
x1d', 'tag': b'?\x18\xa8J\xdaI\xce\xf2\x9b\xa0\xca7\xaa\xe9\xab\x19',
'ciphertext': b'<\xf0\xf3e\x84\xa2\x85\xb7\x05\x9e`\x0b'},
```

'hmac_auth': b'\xac}\x1c~84\x0f\x9e\x83\xb3\xff\xec\x00+q5\xdf\xcfF\
xe3\x9ey\xed\xd4\x90p%^\x9e\xa8\xa9\xd1', 'metadata': b'\x9b\xf5\x14G\
xf1:({\n\xe5mb\x01\x15\xc5\xcf'}

[RECEIVER]: Message received: Hello World!