TP01_P1

April 4, 2021

Solução Para o Trabalho Prático 01

Problema 01:

Este problema trata-se da construção de uma sessão síncrona de comunicação segura entre dois agentes (Emitter e Receiver), onde nós: 1. Implementamos um gerador de nounces, que cria aleatoriamente um nonce que não foi utilizado ainda, em cada instância da comunicação; 2. A implementação da cifra AES, autenticando cada criptograma com HMAC e um modo seguro contra ataques aos IV's; 3. Uso do protocolo de acordo de chaves Diffie-Hellman, verificação da chave, e autenticação dos agentes a partir de um esquema de assinaturas DSA.

[2]:
```
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
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 Esta função deriva a chave partilhada gerada pelo DH de forma a quebrar a esturura presente, adicionar informação e, se necessário, derivar várias chaves.

[ ]:
```
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 do Encrypt e Decrypt Estas funções implementam o AES em modo CTR para cifrar e decifrar conforme necessário. As duas motivações para a escolha do modo são o facto do CTR ser facilmente paralelizável e também não precisar de padding.

[ ]:
```python
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)
```

[ ]:
```python
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)
```

Implementação da Assinatura e Validação para o DSA A assinatura e a validação usam os métodos existente e no caso da validação lidamos com assinaturas inválidas.

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

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

Implementação do HMAC Aqui usamos a SHA256 para gerar o HMAC de cada mensagem. Do lado do recetor geramos o HMAC esperado e comparamos com o recebido.

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

[ ]:
```python
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 Cada mensagem transmitida é da forma m1 = len(signature) || iv || signature || c O IV é gerado aleatóriamente para cada mensagem. O c é obtido ao cifrar o texto limpo usando a chave derivada e o IV. A assinatura é aplicada ao texto limpo de forma a ser essa a mensagem a ser assinada pelo emissor e não o criptograma ou afins. Por fim geramos o HMAC de m1 e fazemos m2 = hmac(m1) || m1, assim não só o criptograma é autenticado mas também o resto da mensagem enviada. A mensagem final tem comprimento variável com o máximo de 4096 bytes, tendo em conta que o texto limpo tem tamanho máximo de 1024 bytes de forma a não haver problema nenhum. len(m2) = 32 (hmac) + 1 (len(signature)) + 16 + len(signature) + len(c)

[ ]:
```
def prepare_bundle(key, message, dsa_key):
    # gerar o nonce
    iv = os.urandom(16)
    # obtenção do criptograma
    ct = encrypt(key, iv, message)
    # assinatura do texto limpo
    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
```

Descrição da Execução do Emitter e Receiver Para ser mais fácil de digerir a explicação será documentada no código do Emitter. O código do Receiver é semelhante mas: troca a ordem da inserção da mensagem e receção; em vez de se inserir uma mensagem é nos enviado um 'ok'.

[ ]:
```
def execucaoemitter(conn, private_key, receiver_public_key, private_dsa_key,␣
↪receiver_dsa_key):
    # gerar a chave combinada usando a chave pública do receiver
    shared_key = private_key.exchange(receiver_public_key)
    # derivar a chave
    derived_key = derive(shared_key)

    while True:
        # primeiro o emitter insire uma mensagem
        msg = input('Emitter: ').encode()
        # se uma das mensagens tiver o comprimento da sua representação em␣
↪bytes maior que o definido a comunicação é encerrada
        if len(msg) > msg_size:
            break
        # obtenção da mensagem pronta a enviar
        bundle = prepare_bundle(derived_key, msg, private_dsa_key)
        # envio da mensagem
```

```
        conn.send(bundle)
        try:
            # preparação do buffer para receber a mensagem (importante para não␣
↪ficarem restos da mensagem anterior)
            buffer = bytearray(buffer_size)
            # receção da mensagem
            buffer = conn.recv()
            # hmac da menssagem recebida
            mac = buffer[0:32]
            # resto da mensagem recebida
            pre_bundle = buffer[32:]
            # se o hmac for válido prosseguimos para o resto do processo
            if validate_hmac(derived_key, pre_bundle, mac):
                # comprimento da assinatura (0..256) (possível que seja␣
↪necessário mais que um byte para chaves de assinatura maiores)
                sig_len = pre_bundle[0]
                # proximos 16 bytes são o iv
                iv = pre_bundle[1:17]
                # a seguir temos sig_len bytes de assinatura
                signature = pre_bundle[17:17 + sig_len]
                # por fim temos o criptograma
                ct = pre_bundle[17 + sig_len:]
                # decifframos o criptograma
                plain_text = decrypt(derived_key, iv, ct)
                # e por fim verificamos a assinatura (em vida real iriamos␣
↪guardá-la junto com a mensagem)
                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()
```

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

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

receiver_conn, emitter_conn = Pipe()

# parametros para os pares de chaves
parameters = dh.generate_parameters(generator=2, key_size=2048,␣
 ↪backend=default_backend())

# par de chaves do emitter
emitter_private_key = parameters.generate_private_key()
# chave publica do emitter
emitter_public_key = emitter_private_key.public_key()
```

```
# par de chaves do receiver
receiver_private_key = parameters.generate_private_key()
# chave publica do receiver
receiver_public_key = receiver_private_key.public_key()

# par de chaves dsa do emitter
emitter_private_dsa_key = dsa.generate_private_key(key_size=2048,␣
 ↪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 = dsa.generate_private_key(key_size=2048,␣
 ↪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)
```