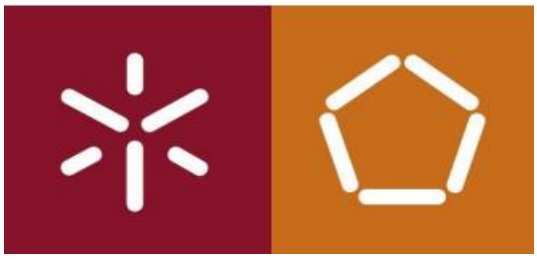


Universidade do Minho

Departamento de Informática

Criptografia e Segurança em Redes

Engenharia de Telecomunicações e Informática

Trabalho Prático 2

Grupo :

Diogo Araújo a101778

Fernando Mendes a101263

Junlin Lu a101270

Conteúdo

1.Introdução

O objetivo deste trabalho pratico é aprofundar o nosso conhecimento, no âmbito da unidade curricular de CSR, sobre Criptografia simétrica e Criptografia das chaves. Desenvolvemos um sistema de comunicação (uma arquitetura cliente-servidor) onde o servidor aceita conexão via *socket* em um endereço IP e porta conhecidos pelo cliente, e aplicamos diferentes níveis de garantia de segurança descordo com os requisitos.

Figura 1: Esquema básico do serviço de chat

Após o estabelecimento de comunicação entre as duas entidades, o servidor deverá a guarda comunicação e processa massagens com os diferentes tipos de garantia de segurança suportadas. o cliente e o servidor iniciarão a troca dos parâmetros exigidos pelo o mecanismo de segurança escolhido pelo cliente

2.Modo de segurança

O serviço desenvolvido deverá suportar três modos de garantia de segurança:

A - **Integridade**: neste modo, o serviço garante a integridade das mensagens trocadas entre os utilizadores, mas não implementa um mecanismo de garantia da confidencialidade;

B - **Confidencialidade e Integridade**: neste modo, além da garantia da integridade das mensagens, o serviço deverá implementar um mecanismo para a garantia da confidencialidade suportado por uma cifra simétrica;

C **- Confidencialidade, Integridade e autenticidade**: no modo mais seguro, o serviço deverá suportar mecanismos que garantam a confidencialidade, a integridade e a autenticidade da origem da mensagem. Para isso, recorra a uma cifra de chave pública.

3. Funcionalidade Cliente/Servidor

Cliente:

```python
conn_port = 7777      # Define a porta para se conectar ao servidor e tamanho da messagem que pode receber
max_msg_size = 9999
async def tcp_echo_client():
    # Estabelece uma conexão TCP com o servidor usando asyncio
    reader, writer = await asyncio.open_connection('127.0.0.1', conn_port)
    # Obtém informações da conexão, como o endereço do cliente
    addr = writer.get_extra_info('peername')
    client = Client(addr) #cire um objeto do class Cliente com endereço do cliente.
    msg = client.process()
    while msg:
        writer.write(msg)       #Envia a mensage e espear servidor acaba de ler
        msg = await reader.read(max_msg_size)
        if msg :
            msg = client.process(msg)
        else:
            break
    writer.write(b'\n')
    print('Socket closed!')
    writer.close()

def run_client():
    loop = asyncio.get_event_loop()#Função para executar o cliente TCP no loop de eventos do asyncio
    loop.run_until_complete(tcp_echo_client())


run_client()
```

Servidor:

```python
async def handle_echo(reader, writer):
    global conn_cnt
    conn_cnt +=1
    addr = writer.get_extra_info('peername') #Obtém informações do cliente conectado
    srvwrk = ServerWorker(conn_cnt, addr)
    data = await reader.read(max_msg_size) # Lê os dados enviados pelo cliente
    while True:
        if not data: continue
        if data[:1]==b'\n': break
        data = srvwrk.process(data) # Processa os dados recebidos
        if not data: break
        writer.write(data) # Escreve a resposta para o cliente
        await writer.drain() # Aguarda até que todos os dados sejam enviados
        data = await reader.read(max_msg_size)
    print("[%d]" % srvwrk.id)
    writer.close()
def run_server():
    loop = asyncio.get_event_loop()
    # Inicia o servidor no endereço e porta especificados
    coro = asyncio.start_server(handle_echo, '127.0.0.1', conn_port)
    server = loop.run_until_complete(coro)# Executa o loop de eventos indefinidamente
    # Serve requests until Ctrl+C is pressed
    print('Serving on {}'.format(server.sockets[0].getsockname()))
    print('  (type ^C to finish)\n')
    try:
        loop.run_forever()
    except KeyboardInterrupt:
        pass
    # Close the server
    server.close()
    loop.run_until_complete(server.wait_closed())
    loop.close()
    print('\nFINISHED!')

run_server()
```

4.Integridade

Neste primeiro modo de segurança apena só garanti a completa de a massagem ser enviado.
Escolhemos de enviar um **JSON** e usar **o hash SHA256** para verificar a integridade da mensagem
Um **dicionário** contendo a mensagem original(msg) e seu hash SHA256(h) é convertido em uma string JSON usando
O **hash SHA-256** é um algoritmo de criptografia que transforma a mensagem em um conjunto único de caracteres, de forma que qualquer alteração na mensagem original resultará em um novo valor de hash. Assim, quando o servidor recebe a mensagem e calcula seu próprio hash SHA-256, ele pode compará-lo com o hash enviado pelo cliente. Se os dois hashes forem iguais, significa que a mensagem não foi alterada durante a transmissão, garantindo assim a sua integridade."

4.1 códigos

Cliente:

```
class Client:
    """ Classe que implementa a funcionalidade de um CLIENTE. """
    def __init__(self, sckt=None):
        """ Construtor da classe. """
        self.sckt = sckt
        self.msg_cnt = 0
    def process(self, msg=b""):
        print('Input message to send (empty to finish)')
        return_msg:str = input()
        h = hashlib.sha256(return_msg.encode()).hexdigest()
        return_data = json.dumps({'msg':return_msg, 'h':h})

        return return_data.encode()
```

Servidor:

```
class ServerWorker(object):
    """ Classe que implementa a funcionalidade do SERVIDOR. """
    def __init__(self, cnt, addr=None):
        """ Construtor da classe. """
        self.id = cnt
        self.addr = addr
        self.msg_cnt = 0
    def process(self, msg):
        self.msg_cnt += 1

        data:dict[str,str]=json.loads(msg)
        h = hashlib.sha256(data['msg'].encode()).hexdigest()


        if h == data['h']:
            print(f"[{self.id}] {data['msg']}")
        else:
            print(f"[{self.id}] erro validação")
        return b'Sus'
```

4.2 Processos no Cliente/Servidor

1- O cliente permite que o usuário insira uma mensagem

```
print('Input message to send (empty to finish)')
return_msg:str = input()
```

2- A mensagem é codificada em bytes e então passada para a função hashlib.sha256(), que calcula o hash SHA256 da mensagem. O método hexdigest() converte o hash em uma string hexadecimal, porque um objetos de hash geralmente não podem ser diretamente serializados em JSON.Depois criar JSON que inclui massagem e hash de massagem,no fim retorna JSON que codificada em bytes

```
h = hashlib.sha256(return_msg.encode()).hexdigest()
return_data = json.dumps({'msg':return_msg, 'h':h})

return return_data.encode()
```

3- Carrega a mensagem recebida (msg), que é esperada como uma string JSON, para um dicionário(data). A mensagem JSON deve conter pelo menos duas chaves: 'msg', que é a mensagem original enviados pelo cliente, e 'h',que é o hash SHA256 da mensagem original.

```
data:dict[str,str]=json.loads(msg)
h = hashlib.sha256(data['msg'].encode()).hexdigest()
```

4- Compara o hash calculado (h) com o hash recebido (data['h']). Se eles são iguais, isso significa que a mensagem não foi alterada durante a transmissão e é ‘’certo’’. O servidor então imprime a mensagem com o ID da conexão. Se os hashes não coincidem, imprime uma mensagem de erro com o ID da conexão, indicando que houve um erro de validação.

```
if h == data['h']:
    print(f"[{self.id}] {data['msg']}")
else:
    print(f"[{self.id}] erro validação")

return b'Sus'
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SERIAL MONITOR  COMMENTS

onWarning: There is no current event loop
  loop = asyncio.get_event_loop()
Serving on ('127.0.0.1', 7777)
  (type ^C to finish)

[1] 1
[1] 1
[1] 21
[1] asd

Input message to send (empty to finish)
1
Input message to send (empty to finish)
1
Input message to send (empty to finish)
21
Input message to send (empty to finish)
asd
Input message to send (empty to finish)
```

4.3 Conclusão:

-msg envido e recebido são capturado por Wireshark

Embora este processo assegure a integridade da mensagem, ainda existem alguns riscos potenciais. A mensagem é transmitida juntamente com o hash SHA256 da mesma messagem, o que significa que se os dados forem intercetados, o attacker poderá ver o conteúdo da mensagem.

5. Confidencialidade e Integridade

Neste modo de segurança, a integridade da mensagem é garantida, e a criptografia da mensagem assegura a sua confidencialidade.
Escolhemos protocolo **AES** para assegurar a segurança na comunicação, onde o AES proporciona uma criptografia forte e eficiente para o conteúdo das mensagens,.A integridade das mensagens é verificada por meio de **hashes SHA-256**, garantindo que as mensagens não sejam alteradas durante a transmissão.
O cliente estabelece uma conexão segura com o servidor, criptografa as mensagens usando AES com chave fixo e vetor inicial(nonce) random,depois assegura a integridade delas com um hash SHA-256 antes de enviá-las, garantindo que as mensagens cheguem seguras ao servidor, depois servidor descriptografar as mensagens e mostra.

5.1 códigos

Cliente:

```
key = b'HelloSever114514'

class Client:
    """ Classe que implementa a funcionalidade de um CLIENTE. """
    def __init__(self, sckt=None):
        """ Construtor da classe. """
        self.sckt = sckt
        self.msg_cnt = 0
    def process(self, msg=b""):
        self.msg_cnt += 1
        nonce = os.urandom(8)
        cipher = AES.new(key, AES.MODE_CTR,nonce=nonce)

        print('Input message to send (empty to finish)')
        return_msg = input()


        h = hashlib.sha256(return_msg.encode()).hexdigest()
        encrypted_msg = cipher.encrypt(return_msg.encode())
        encrypted_msg_b64 = base64.b64encode(nonce +
encrypted_msg).decode()

        return_data = json.dumps({'msg': encrypted_msg_b64, 'h': h})
        return return_data.encode()
```

Servidor:

```
conn_cnt = 0
conn_port = 7777
max_msg_size = 9999

key = b'HelloSever114514'

class ServerWorker(object):
    """ Classe que implementa a funcionalidade do SERVIDOR. """
    def __init__(self, cnt, addr=None):
        """ Construtor da classe. """
        self.id = cnt
        self.addr = addr
        self.msg_cnt = 0

    def process(self, msg=b""):
        data = json.loads(msg)

        if not 'msg' in data or not 'h' in data:
         raise ValueError('Error Message')
        encrypted_msg_b64 = data['msg']
        encrypted_msg = base64.b64decode(encrypted_msg_b64)
        nonce = encrypted_msg[:8]
        new_encrypted_msg = encrypted_msg[8:]
        cipher = AES.new(key, AES.MODE_CTR, nonce=nonce)
        dec_msg = cipher.decrypt(new_encrypted_msg)
        h =hashlib.sha256(dec_msg).hexdigest()
        if h != data['h']: raise ValueError('Error Message')

        self.msg_cnt += 1

        print('-------------------------')
        print('Ciphertext ',data["msg"])
        print('Plaintext:',dec_msg.decode())

        return msg
```

5.2 Processos no Cliente/Servidor

1- Define chave ser usado para cifra AES. Key é Chave de encriptação necessária para criptografar e descriptografar os dados.

```
key = b'HelloSever114514'

class ServerWorker(object):
```

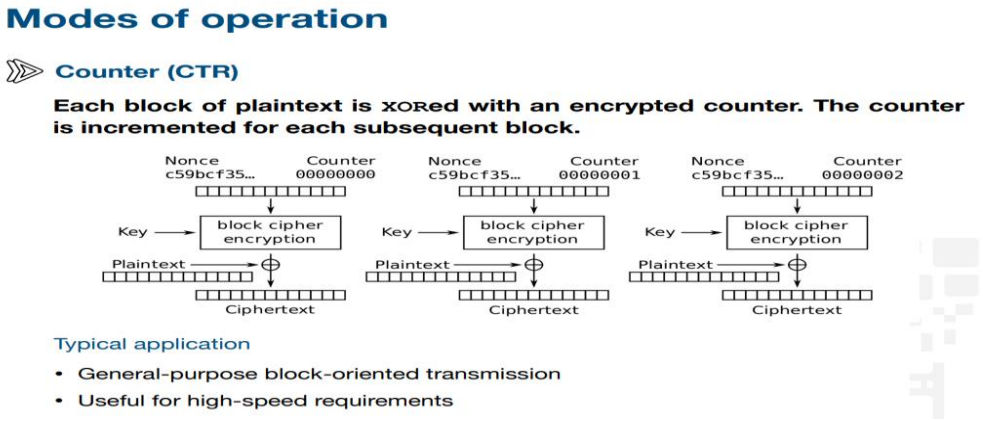
```
key = b'HelloSever114514'

class Client:
```

2- Nesta linha de código está sendo criado um **encrpytor** AES que será utilizado para encriptar massagens.

```
nonce = os.urandom(8)
cipher = AES.new(key, AES.MODE_CTR,nonce=nonce)
```

key: Esta é a chave de encriptação necessária para criptografar e descriptografar os dados. Em AES, a chave pode ter 128, 192 ou 256 bits. Esta chave deve ser mantida em segredo e conhecida apenas pelo Servidor e Cliente para garantir a confidencialidade dos dados.
AES.MODE_CTR: Isso especifica que o modo de operação da cifra é o CTR (Counter Mode), um dos vários modos de operação disponíveis no AES. O modo CTR permite encriptar dados de qualquer comprimento sem a necessidade de preenchimento e transforma a cifra de bloco AES em uma cifra de fluxo.
nonce: é gerado aleatoriamente com 8 bytes de tamanho para ser usado no processo de encriptação



3- Cliente inserir um mensagem, criptografa a nova mensagem usando encryptor calcula o hash SHA-256 dessa nova mensagem. Depois codificado em Base64. A codificação Base64 é utilizada para tornar os dados binários seguros para serem transmitidos e para ser compatível no JSON.O cliente retorna a **nonce + mensagem** criptografada e o hash em um JSON codificado em bytes para ser enviado ao servidor.

```
print('Input message to send (empty to finish)')
return_msg = input()


h = hashlib.sha256(return_msg.encode()).hexdigest()
encrypted_msg = cipher.encrypt(return_msg.encode())
encrypted_msg_b64 = base64.b64encode(nonce + encrypted_msg).decode()

return_data = json.dumps({'msg': encrypted_msg_b64, 'h': h})
return return_data.encode()
```

4- O servidor recebe uma mensagem JSON do cliente e a converte de volta em um dicionário,e verifica se a mensagem contém os campos 'msg' e 'h',depois triar nonce e messagem.Criar **decryptor** com chave e nonce, decripta massagem.
Depois calcula o hash SHA-256 da mensagem descriptografada para verificar sua integridade.

```
def process(self, msg=b""):
    data = json.loads(msg)

    if not 'msg' in data or not 'h' in data:
     raise ValueError('Error Message')
    encrypted_msg_b64 = data['msg']
    encrypted_msg = base64.b64decode(encrypted_msg_b64)
    nonce = encrypted_msg[:8]
    new_encrypted_msg = encrypted_msg[8:]
    cipher = AES.new(key, AES.MODE_CTR, nonce=nonce)
    dec_msg = cipher.decrypt(new_encrypted_msg)
    h =hashlib.sha256(dec_msg).hexdigest()
    if h != data['h']: raise ValueError('Error Message')
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SERIAL MONITOR  COMMENTS

PS D:\3Ano\3ano\Cryptography\TP2\TP2 simple AES> python Client.py
D:\3Ano\3ano\Cryptography\TP2\TP2 simple AES\Client.py:63: DeprecationWarning: There i
s no current event loop
  loop = asyncio.get_event_loop()
Input message to send (empty to finish)
yugyj
Input message to send (empty to finish)
```

```
is no current event loop
  loop = asyncio.get_event_loop()
Serving on ('127.0.0.1', 7777)
  (type ^C to finish)

----------------------
Ciphertext  5PulrfPVIZ+xvy9wHg==
Plaintext: yugyj
```

5.3 Conclusão

Este método de usar o modo AES CTR e chave para garantir a integridade e a confidencialidade das mensagens é eficaz porque o modo CTR pode processar dados de qualquer comprimento sem a necessidade de preenchimento(padding) para um tamanho de bloco fixo, a função de hash SHA-256 é usada para verificar a integridade da mensagem, garantindo que os dados não sejam alterados durante a transmissão.

-msg ser captura, agora Attacker não é possível determinar o conteúdo da mensagem origina
Hash é paintext, mas são unidirecionais, o que significa não pode recuperar os dados originais a partir do valor de hash

6. Confidencialidade, Integridade e autenticidade

Nesse último modo, Não só usa de protocolo para segurar Confidencialidade, Integridade necessário usar um protocolo de cifra adicional para garanti autenticidade.Por isso para além dos requisitos de base anteriores,adicionamos um **processo de trocar chaves,Diffe-Hellman**, e acrescentamos o algoritmo **RSA**,que usa sua chave privada para **assinar a chave pública Diffie-Hellman**. Após o cliente receber esses dados, ele usa a chave pública RSA do servidor para **verificar a assinatura**. Se a verificação da assinatura for bem-sucedida, o cliente pode confirmar que a chave pública Diffie-Hellman recebida realmente vem do servidor declarado e não foi alterada por terceiros.Assim grante autenticidade das chaves.

6.1 códigos

Cliente:

```
class Client:
    """ Classe que implementa a funcionalidade de um CLIENTE. """
    def __init__(self, sckt=None):
        """ Construtor da classe. """
        self.sckt = sckt
        self.msg_cnt = 0

    def process(self, msg=b""):

        self.msg_cnt += 1
        match self.msg_cnt :
            case 1:

                return bytes([self.msg_cnt])
            case 2:
                pem,parameters= msg.split(b"---SPLIT---")

                server_public_key = load_pem_public_key(pem)
                para = load_pem_parameters(parameters,
backend=default_backend())
                self.private_key = para.generate_private_key()
                self.public_key = self.private_key.public_key()
                self.shared_key =
self.private_key.exchange(server_public_key)
               # ShowkeyC = self.shared_key
               # print (ShowkeyC)
                pem_c = self.public_key.public_bytes(
                    encoding=serialization.Encoding.PEM,
                    format=serialization.PublicFormat.SubjectPublicKeyInf
o
                )

                return bytes([self.msg_cnt])+pem_c
            case _:
                    data = json.loads(msg)
                    if not 'msg' in data or not 'h' in data:
```

```
                    raise ValueError('Error Message')
                dec_msg = self.decrypt(data['msg'])
                h = hashlib.sha256(dec_msg.encode()).hexdigest()
                if h != data['h']: raise ValueError('Error Message')

                print(f'Received ({self.msg_cnt}): {dec_msg}, hash:
{h}')

                input_msg = input('Input message to send (empty to
finish)\n')
                if  not input_msg: return None

                return bytes([3]) + json.dumps({
                'msg': self.encrypt(input_msg),
                'h': hashlib.sha256(input_msg.encode()).hexdigest(),
                }, separators=(',',':')).encode()

    def encrypt(self, msg: str|bytes) -> str:
        if isinstance(msg, str): msg = msg.encode()
        key = HKDF(
                algorithm=hashes.SHA256(),
                length=16,
                salt=None,
                info=b'encryption',
                backend=default_backend()
                ).derive(self.shared_key)
        nonce = os.urandom(8)
        #ctr = Counter.new(64, prefix=nonce)
        #cipher = AES.new(key, AES.MODE_CTR,counter=ctr)
        cipher = AES.new(key, AES.MODE_CTR,nonce = nonce)
        encrypt_data = cipher.encrypt(msg)
        return base64.b64encode(nonce + encrypt_data).decode()

    def decrypt(self, msg:str|bytes) -> str:
        if isinstance(msg, bytes): msg = msg.decode()
        msg = base64.b64decode(msg)
        key = HKDF(
                algorithm=hashes.SHA256(),
                length=16,
                salt=None,
                info=b'encryption',
                backend=default_backend()
                ).derive(self.shared_key)
        nonce = msg[:8]
        ctr = Counter.new(64, prefix=nonce)
        encrypt_data = msg[8:]
        cipher = AES.new(key, AES.MODE_CTR,counter=ctr)
        decrypt_data = cipher.decrypt(encrypt_data)
        return decrypt_data.decode()
```

Servidor:

```
class ServerWorker(object):
    """ Classe que implementa a funcionalidade do SERVIDOR. """
    def __init__(self, cnt, addr=None):
        """ Construtor da classe. """
        self.id = cnt
        self.addr = addr
        self.msg_cnt = 0


    def process(self, msg:bytes):

        if not msg : return None #

        msg_type = msg[0]
        msg = msg[1:]
        match msg_type:
            case 1:
                self.rsa_private_key = rsa.generate_private_key(
                public_exponent=65537,
                key_size=2048,
                )
                self.rsa_public_key = self.rsa_private_key.public_key()
                self.parameters = dh.generate_parameters(generator=2,
key_size=2048)
                self.dh_private_key =
self.parameters.generate_private_key()
                self.dh_public_key = self.dh_private_key.public_key()

                pem_rsa = self.rsa_public_key.public_bytes(
                    encoding=serialization.Encoding.PEM,
                    format=serialization.PublicFormat.SubjectPublicKeyInf
o
                )
                pem_dh = self.dh_public_key.public_bytes(
                encoding=serialization.Encoding.PEM,
                format=serialization.PublicFormat.SubjectPublicKeyInfo
                )
                p_pem =self.parameters.parameter_bytes(
                encoding=Encoding.PEM,
                format=ParameterFormat.PKCS3
                )
                signature = self.rsa_private_key.sign(
                    pem_dh,
                    padding.PSS(
                    mgf=padding.MGF1(hashes.SHA256()),
                    salt_length=padding.PSS.MAX_LENGTH
                              ),
```

```
                    hashes.SHA256()
                    )
                return pem_rsa + b"---SPLIT---" +signature+b"---SPLIT---
"+pem_dh+b"---SPLIT---"+p_pem
            case 2:
                pem_c= msg
                client_public_key = load_pem_public_key(pem_c)
                self.shared_key =
self.dh_private_key.exchange(client_public_key)

                return json.dumps({
                    'msg': self.encrypt("Connect Sucess"),
                    'h': hashlib.sha256(b"Connect Sucess").hexdigest()
                }, separators=(',',':')).encode()

            case 3:
                  self.id+=2
                  data = json.loads(msg)
                  if not 'msg' in data or not 'h' in data:
                    raise ValueError('Error Message')

                  dec_msg = self.decrypt(data["msg"])
                  h = hashlib.sha256(dec_msg.encode()).hexdigest()
                  if h != data['h']: raise ValueError('Error Message')
                  print(f'Ciphertext {self.id} : {data["msg"]}')
                  result = self.decrypt(data["msg"])
                  print(f'Plaintext: {result}')
                  return msg



    def encrypt(self, msg: str|bytes) -> str:
        if isinstance(msg, str): msg = msg.encode()

        key = HKDF(
                algorithm=hashes.SHA256(),
                length=16,
                salt=None,
                info=b'encryption',
                backend=default_backend()
                ).derive(self.shared_key)
        nonce = os.urandom(8)

        cipher = AES.new(key, AES.MODE_CTR,nonce=nonce)
        encrypt_data = cipher.encrypt(msg)
        return base64.b64encode(nonce + encrypt_data).decode()

    def decrypt(self, msg:str|bytes) -> str:
        if isinstance(msg, bytes): msg = msg.decode()
```

```
        msg = base64.b64decode(msg)
        key = HKDF(
                algorithm=hashes.SHA256(),
                length=16,
                salt=None,
                info=b'encryption',
                backend=default_backend()
                ).derive(self.shared_key)
        nonce = msg[:8]

        encrypt_data = msg[8:]

        cipher = AES.new(key, AES.MODE_CTR,nonce=nonce)
        decrypt_data = cipher.decrypt(encrypt_data)
        return decrypt_data.decode('utf-8')
```

6.2 Processos no Cliente/Servidor

1- A primeira mensagem enviada pelo cliente. Ao converter o valor do contador em bytes e retorná-lo, o cliente informa ao servidor que está pronto para iniciar o processo de troca de chaves.

```
case 1:

    return bytes([self.msg_cnt])
```

Na primeira vez de comunicação entre cliente e servidor,servidor gerar as chaves RSA e sua chaves DH

```
self.rsa_private_key = rsa.generate_private_key(
public_exponent=65537,
key_size=2048,
)
self.rsa_public_key = self.rsa_private_key.public_key()
self.parameters = dh.generate_parameters(generator=2, key_size=2048)
self.dh_private_key = self.parameters.generate_private_key()
self.dh_public_key = self.dh_private_key.public_key()
```

depois as chaves públicas de RSA e DH e parâmetros são serializadas no formato PEM,e usa chave privada de RSA assina de PEM de chave publico de DH

```
pem_rsa = self.rsa_public_key.public_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PublicFormat.SubjectPublicKeyInfo
)
pem_dh = self.dh_public_key.public_bytes(
encoding=serialization.Encoding.PEM,
format=serialization.PublicFormat.SubjectPublicKeyInfo
)
p_pem =self.parameters.parameter_bytes(
encoding=Encoding.PEM,
format=ParameterFormat.PKCS3
)
signature = self.rsa_private_key.sign(
    pem_dh,
    padding.PSS(
    mgf=padding.MGF1(hashes.SHA256()),
    salt_length=padding.PSS.MAX_LENGTH
                ),
    hashes.SHA256()
    )
```

Final retorna a chave pública RSA e DH do servidor, assinatura e o parâmetro serializados, separados por "---SPLIT---" para que o cliente possa distinguir facilmente entre os dois componentes da mensagem.

```
return pem_rsa + b"---SPLIT---" +signature+b"---SPLIT---"+pem_dh+b"---SPLIT---"+p_pem
```

2- O cliente recebe e divide essa mensagem usando msg.split(b"---SPLIT---") para obter se a chave pública do servidor (pem ras e pem dh),assinatura(signature) e os parâmetros Diffie-Hellman (parameters).
carrega a chave pública rsa do servidor(pem_rsas),verifica assinatura. Usa e parâmetro para gerar chave privada e publico DH de cliente. Depois gera chave compartilhada, depois envia chave publico DH de cliente em forma PEM para servidor

```
case 2:
    pem_rsas,signature,pem_dhs,parameters= msg.split(b"---SPLIT---")

    server_rsa_public_key = load_pem_public_key(pem_rsas)
    try:
        server_rsa_public_key.verify(
            signature,
            pem_dhs,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
    except:
        raise ValueError("Signature verification failed!!!!")
    server_dh_public_key = load_pem_public_key(pem_dhs)
    para = load_pem_parameters(parameters)
    self.dh_private_key = para.generate_private_key()
    self.dh_public_key = self.dh_private_key.public_key()
    self.shared_key = self.dh_private_key.exchange(server_dh_public_key)
    pem_c = self.dh_public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )
    #print(self.shared_key)
    return bytes([self.msg_cnt])+pem_c
```

3- Volta para servidor, recebe a mensagem do cliente, que contém a chave pública do cliente em formato PEM.Carregar essa chave e gerar outro chave compartilhada.

```
D:\3Ano\3ano\Cryptography\TP2\TP2 AESctr\Server.py:147: DeprecationWarning: There is no current event loop
  loop = asyncio.get_event_loop()
Serving on ('127.0.0.1', 7777)
  (type ^C to finish)

2'd\xa6!y35\x9b-\x96\x88t\xb9J\xfc\xd2\xed\xeen\xfd\xab\x9f\xabbs?V\xbbI\xd8&(\x8cjR\xe6\xd0\x9d\xdd\xa4\xdf\xfe\xb4\xfc\xe1[a\xee\xb6\x9d\xa3\t\x19z\x99yX}\xb7\xdaem\xb2W\x89\xef\x0f6\x95\xa9\xe4\xb3FT\t\x7fQJ8\x08\x89%K\xf1\xf4O\x1b\xe0\xe8?.Y\xa6\xc0\xe6\x94\x88l\xad\xcb+\xe5\xc1\xbd\xf1+FL\x1f\xab\x03\xda~\xff\xac9\xdf\x16\x86\x8d\xb6\x84 x\xef\xcc\xd8\xe7\x05R\x1f+\x1ah\xd94\xb4\x18\x9b\xf5M\xdc\x97\x01\xe0\x7f\x1fd?\xbb\xf5\xa3sM\x15|\xa3r8\xd3v1bt\xa1\xc2\x16\xb7\x93\xb8\xc0\xd9\x9c\xad\xaa\xb7gZ!/\xe4#\x00\x1c\x0c\xf4#eZ\xf9\x1c\x8b\x1d\x01;~z\t)S\xd7z\xe6\x98f\x86\xd7\xb9\x8b\xb9\xc9\x16|\xa0]_`\xd4\xc6\xb8\x89{\xe8E\xf1CaRg\xb1\xcf\xe8VX2\x0f\xb5\xfepl\xf2\xfe\xf1#gCU%\x86G\xaa4\xa9\xe4\xbf\x93\x8c'
```

```
D:\3Ano\3ano\Cryptography\TP2\TP2 AESctr\Client.py:136: DeprecationWarning: There is no current event loop
  loop = asyncio.get_event_loop()
b'd\xa6!y35\x9b-\x96\x88t\xb9J\xfc\xd2\xed\xeen\xfd\xab\x9f\xabbs?V\xbbI\xd8&(\x8cjR\xe6\xd0\x9d\xdd\xa4\xdf\xfe\xb4\xfc\xe1[a\xee\xb6\x9d\xa3\t\x19z\x99yX}\xb7\xdaem\xb2W\x89\xef\x0f6\x95\xa9\xe4\xb3FT\t\x7fQJ8\x08\x89%K\xf1\xf4O\x1b\xe0\xe8?.Y\xa6\xc0\xe6\x94\x88l\xad\xcb+\xe5\xc1\xbd\xf1+FL\x1f\xab\x03\xda~\xff\xac9\xdf\x16\x86\x8d\xb6\x84 x\xef\xcc\xd8\xe7\x05R\x1f+\x1ah\xd94\xb4\x18\x9b\xf5M\xdc\x97\x01\xe0\x7f\x1fd?\xbb\xf5\xa3sM\x15|\xa3r8\xd3v1bt\xa1\xc2\x16\xb7\x93\xb8\xc0\xd9\x9c\xad\xaa\xb7gZ!/\xe4#\x00\x1c\x0c\xf4#eZ\xf9\x1c\x8b\x1d\x01;~z\t)S\xd7z\xe6\x98f\x86\xd7\xb9\x8b\xb9\xc9\x16|\xa0]_`\xd4\xc6\xb8\x89{\xe8E\xf1CaRg\xb1\xcf\xe8VX2\x0f\xb5\xfepl\xf2\xfe\xf1#gCU%\x86G\xaa4\xa9\xe4\xbf\x93\x8c'
Received (3): Connect Sucess, hash: 8b3afa5256e14cf2d6a907fb364a1e38c6327b5153d2452df47ac09e83aaf4c1
Input message to send (empty to finish)
```

-Cliente e Servidor têm mesmo chave compartilhada

Como neste momento os dois lados já têm chave compartilhada pronto, Podemos construir as funções encrypt e decrypt,são métodos auxiliares usados para criptografar e descriptografar mensagens.

encrypt: Uma nova chave é derivada usando o algoritmo HKDF (HMAC Key Derivation Function) com SHA-256 como função de hash. A chave é derivada de chave compartilhada.Criptografa a mensagem usando o modo CTR do AES. Ele gera um nonce (número utilizado apenas uma vez), cria um contador para o AES-CTR e criptografa a mensagem. A mensagem criptografada é então codificada em Base64 junto com o nonce para transmissão.

```python
def encrypt(self, msg: str|bytes) -> str:
    if isinstance(msg, str): msg = msg.encode()

    key = HKDF(
            algorithm=hashes.SHA256(),
            length=16,
            salt=None,
            info=b'encryption',
            backend=default_backend()
            ).derive(self.shared_key)
    nonce = os.urandom(8)

    cipher = AES.new(key, AES.MODE_CTR,nonce=nonce)
    encrypt_data = cipher.encrypt(msg)
    return base64.b64encode(nonce + encrypt_data).decode()
```

decrypt: A mensagem é então decodificada de Base64(porque nosso msg encrypt estava codificado em Base64) para obter o nonce e os dados criptografados. A mesma chave usada para criptografar a mensagem é re-derivada usando HKDF com os mesmos parâmetros.Descriptografa uma mensagem que foi criptografada com **encrypt**, extrai o nonce e o dado criptografado, e usa AES ,mesmo modo CTR para descriptografar a mensagem.

```python
def decrypt(self, msg:str|bytes) -> str:
    if isinstance(msg, bytes): msg = msg.decode()
    msg = base64.b64decode(msg)
    key = HKDF(
            algorithm=hashes.SHA256(),
            length=16,
            salt=None,
            info=b'encryption',
            backend=default_backend()
            ).derive(self.shared_key)
    nonce = msg[:8]

    encrypt_data = msg[8:]

    cipher = AES.new(key, AES.MODE_CTR,nonce=nonce)
    decrypt_data = cipher.decrypt(encrypt_data)
    return decrypt_data.decode('utf-8')
```

Depois criar JSON que inclui massagem random e hash de massagem random,no fim retorna JSON que codificada em bytes para clientes

```python
case 2:
    pem_c= msg
    client_public_key = load_pem_public_key(pem_c)
    self.shared_key = self.private_key.exchange(client_public_key)

    return json.dumps({
        'msg': self.encrypt("Connect Sucess"),
        'h': hashlib.sha256(b"Connect Sucess").hexdigest()
    }, separators=(',',':')).encode()
```

4- O cliente recebe uma mensagem JSON do servidor e a converte de volta em um dicionário,e verifica se a mensagem contém os campos 'msg' e 'h',depois calcula o hash SHA-256 da mensagem descriptografada para verificar sua integridade.

```
data = json.loads(msg)
if not 'msg' in data or not 'h' in data:
  raise ValueError('Error Message')
dec_msg = self.decrypt(data['msg'])
h = hashlib.sha256(dec_msg.encode()).hexdigest()
if h != data['h']: raise ValueError('Error Message')

print(f'Received ({self.msg_cnt}): {dec_msg}, hash: {h}')
```

Se uma mensagem for inserida, o cliente criptografa a nova mensagem usando self.encrypt(input_msg) e calcula o hash SHA-256 dessa nova mensagem.
O cliente retorna a nova mensagem criptografada e o hash em um JSON codificado em bytes para ser enviado ao servidor.

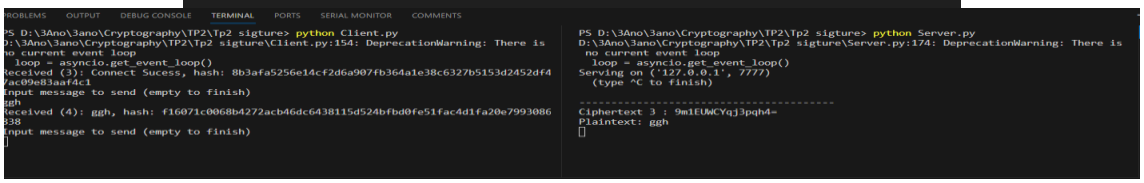
```
input_msg = input('Input message to send (empty to finish)\n')
if  not input_msg: return None

return bytes([3]) + json.dumps({
'msg': self.encrypt(input_msg),
'h': hashlib.sha256(input_msg.encode()).hexdigest(),
}, separators=(',',':')).encode()
```

5-Finalmente o servidor vai fazer mesmo processo de verificação e decrypt...

```
case 3:
        data = json.loads(msg)
        if not 'msg' in data or not 'h' in data:
          raise ValueError('Error Message')

        dec_msg = self.decrypt(data["msg"])
        h = hashlib.sha256(dec_msg.encode()).hexdigest()
        if h != data['h']: raise ValueError('Error Message')
        print(f'Ciphertext {self.id} : {data["msg"]}')
        result = self.decrypt(data["msg"])
        print(f'Plaintext: {result}')
        return msg
```



6.3 Conclusão

A utilização da assinatura RSA serve como uma forma de e garantia autenticidade. E modo AES CTR ,a troca de chaves Diffie-Hellman e Hash-256 para garantir a integridade e a confidencialidade das mensagens