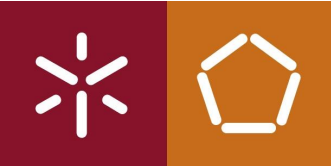
****

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 3](#_Toc150819817)
* [2.Modo de segurança 3](#_Toc150819818)
* [3. Funcionalidade Cliente/Servidor 4](#_Toc150819819)
* [4.Integridade 5](#_Toc150819820)

[4.1 códigos 5](#_Toc150819821)

[4.2 Processos no Cliente/Servidor 6](#_Toc150819822)

[4.3 Conclusão: 6](#_Toc150819823)

* [5. Confidencialidade e Integridade 7](#_Toc150819824)

[5.1 códigos 7](#_Toc150819825)

[5.2 Processos no Cliente/Servidor 11](#_Toc150819826)

[5.3 Conclusão 13](#_Toc150819827)

# 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;

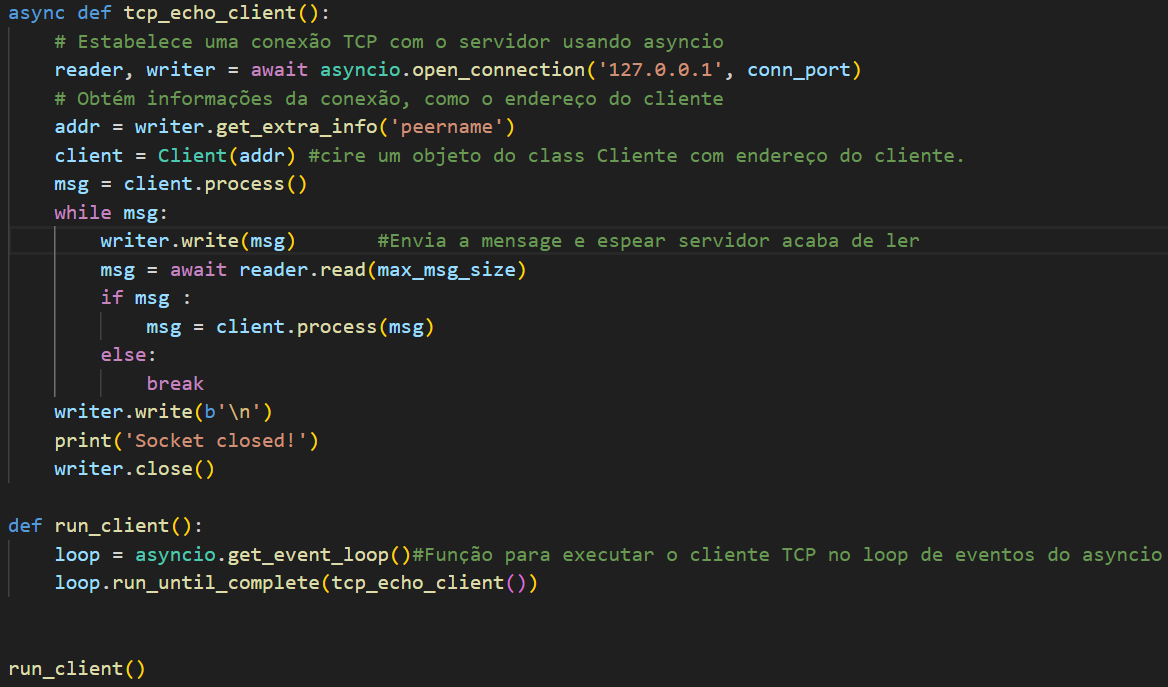
B - **Confidencialidade e Integridade**: neste modo, além da garantia da integridade das mensagens, o serviço deverá implementar um mecanismo para a garantia da confidencialidade suportado por uma cifra simétrica;

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.

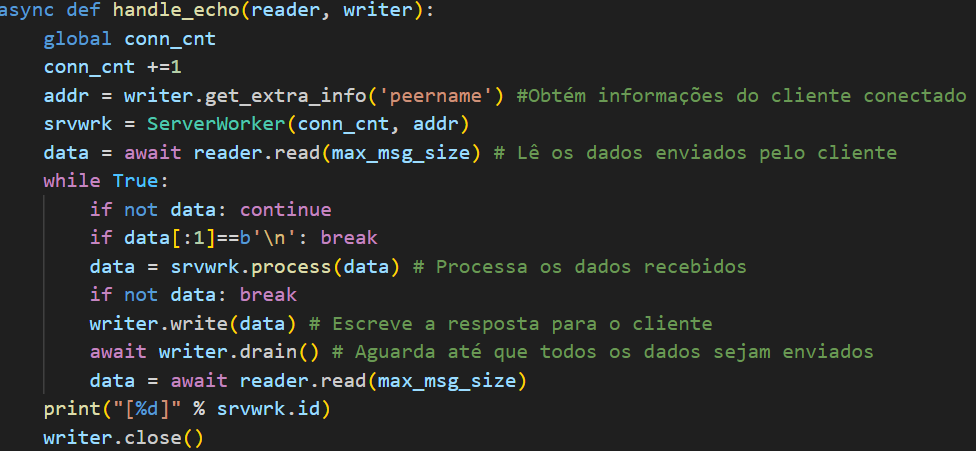
# 3. Funcionalidade Cliente/Servidor

Cliente:





Servidor:





# 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")

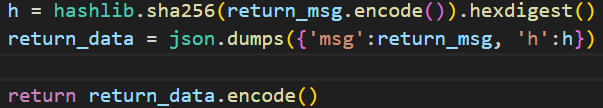
        return b'Sus'

## **Processos no Cliente/Servidor**

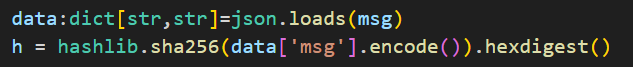
1. O cliente permite que o usuário insira uma mensagem



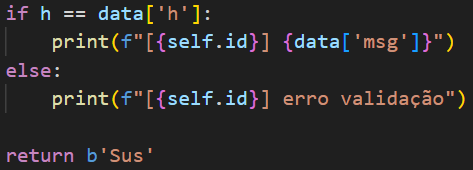
1. A mensagem é codificada em bytes e então passada para a função hashlib.sha256(), que calcula o hash SHA256 da mensagem. O método hexdigest() converte o hash em uma string hexadecimal, porque um objetos de hash geralmente não podem ser diretamente serializados em JSON.Depois criar JSON que inclui massagem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

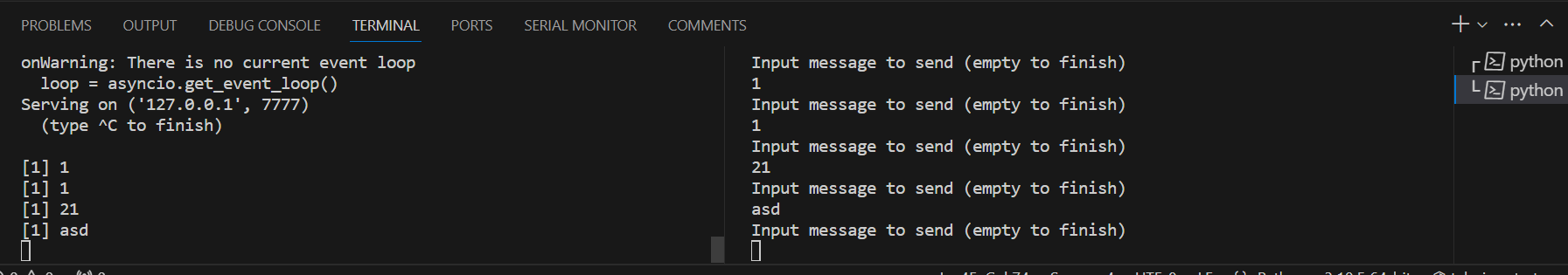


1. Carrega a mensagem recebida (msg), que é esperada como uma string JSON, para um dicionário(data). A mensagem JSON deve conter pelo menos duas chaves: 'msg', que é a mensagem original enviados pelo cliente, e 'h',que é o hash SHA256 da mensagem original.

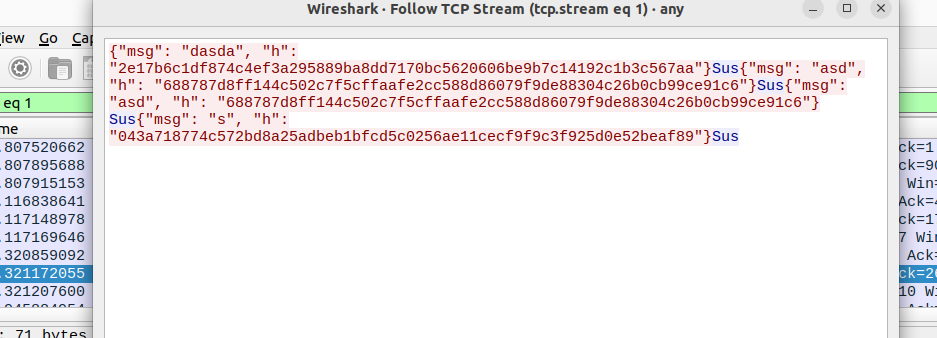


1. Compara o hash calculado (h) com o hash recebido (data['h']). Se eles são iguais, isso significa que a mensagem não foi alterada durante a transmissão e é ‘’certo’’. O servidor então imprime a mensagem com o ID da conexão. Se os hashes não coincidem, imprime uma mensagem de erro com o ID da conexão,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





## 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** e **Diffie-Hellman** são utilizados em conjunto para assegurar a segurança na comunicação, onde o AES proporciona uma criptografia forte e eficiente para o conteúdo das mensagens, e o Diffie-Hellman permite a troca segura de chaves criptográficas, mesmo através de canais não seguros. A integridade das mensagens é verificada por meio de **hashes SHA-256**, garantindo que as mensagens não sejam alteradas durante a transmissão, enquanto o **HKDF** é usado para derivar chaves fortes e seguras a partir de um chave compartilhado.

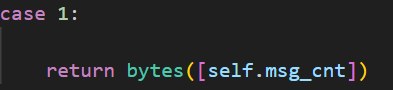
O cliente estabelece uma conexão segura com o servidor, trocando chaves publicas para criar uma chave compartilhada. Após a conexão, ele criptografa as mensagens usando AES e assegura a integridade delas com um hash SHA-256 antes de enviá-las, garantindo que as mensagens cheguem seguras ao servidor, depois servidor descriptografar as mensagens e mostra.

## 5.1 códigos

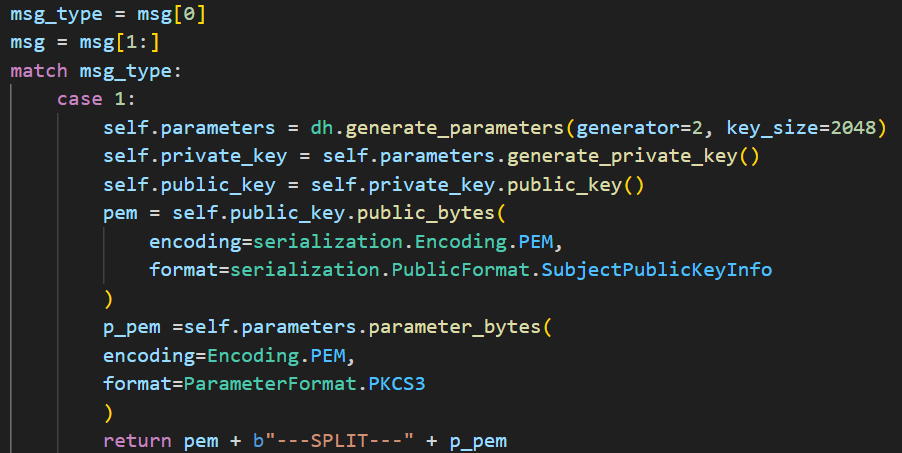
Cliente:

## **5.2 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.

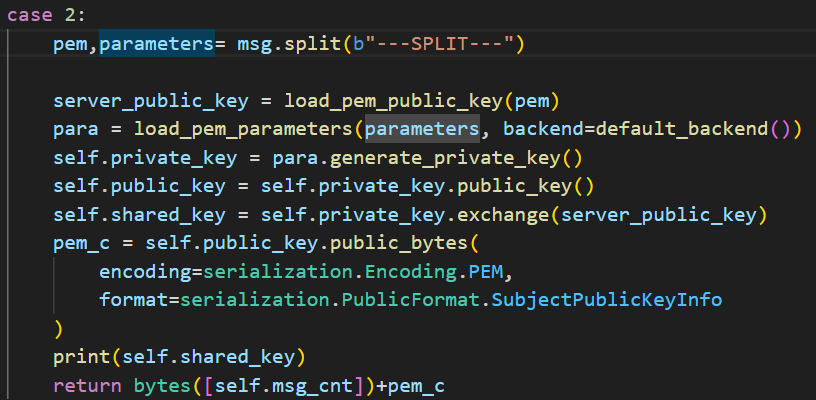


O servidor gera parâmetros Diffie-Hellman Com esses parâmetros, o servidor cria uma chave privada do servido, partir da chave privada, o servidor gera uma chave pública do servidor, depois a chave pública e parâmetros são serializadas no formato PEM. Final retorna a chave pública do servidor e o parâmetro serializados, separados por uma sequência especial "---SPLIT---" para que o cliente possa distinguir facilmente entre os dois componentes da mensagem.

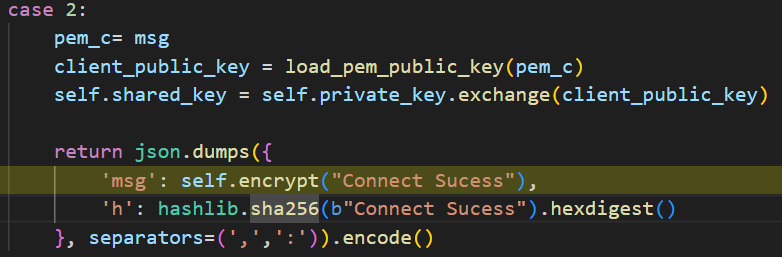
Parte superior do formulário

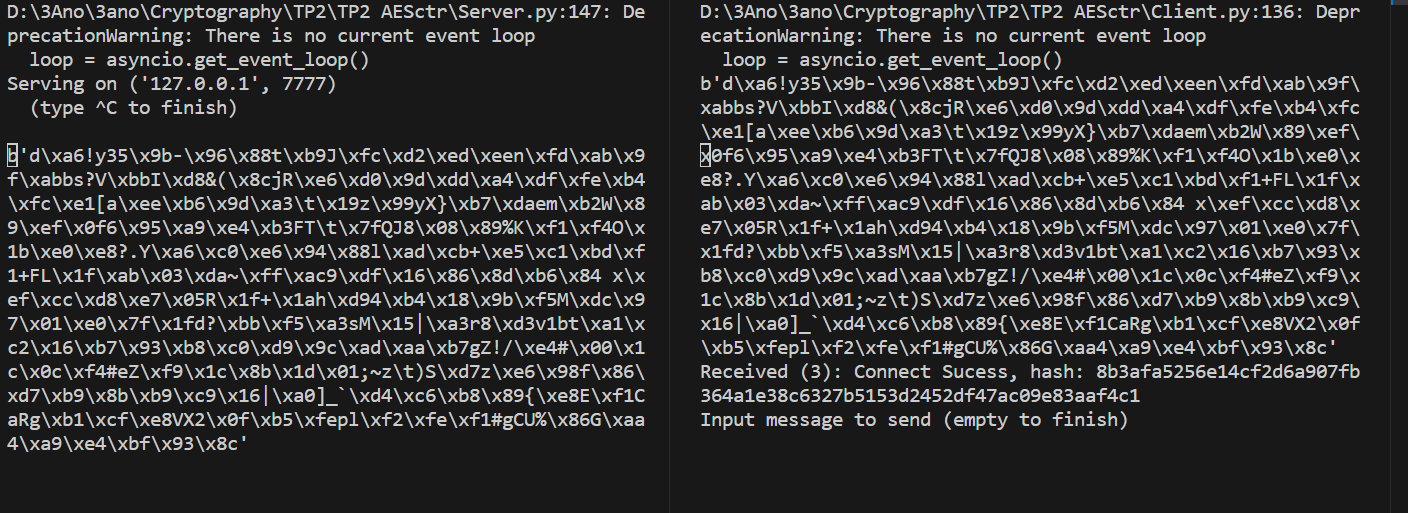
2- O cliente recebe e divide essa mensagem usando msg.split(b"---SPLIT---") para obter se a chave pública do servidor (pem) e os parâmetros Diffie-Hellman (parameters).

carrega a chave pública do servidor e parâmetro para gerar chave privada e publico de cliente para gerar chave compartilhada, Depois envia chave publico de cliente em forma PEM para servidor



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





-Cliente e Servidor têm mesmo chave compartilhada

6- 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.

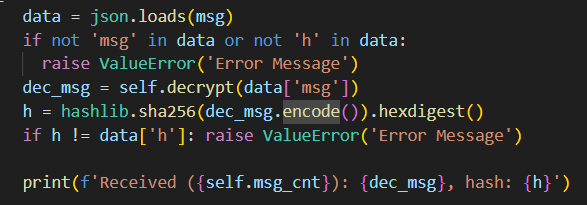
**encrypt**: Criptografa a mensagem usando o modo CTR do AES. Ele gera um nonce (número utilizado apenas uma vez), cria um contador para o AES-CTR e criptografa a mensagem. A mensagem criptografada é então codificada em Base64 junto com o nonce para transmissão.

**decrypt**: Descriptografa uma mensagem que foi criptografada com encrypt. Ele decodifica a mensagem do Base64, extrai o nonce e o dado criptografado, e usa o mesmo processo de contagem para descriptografar a mensagem.



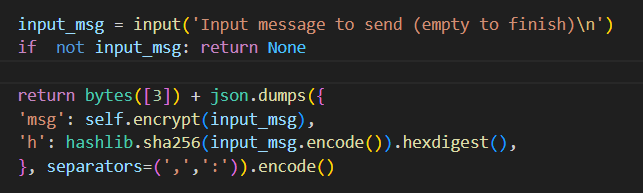
O modo CTR (Counter mode) é uma modalidade de cifra que permite o processamento paralelo, dispensa o preenchimento de dados para se adequar ao tamanho do bloco e possibilita o acesso aleatório ao texto cifrado, razões pelas quais o AES CTR é amplamente utilizado em aplicações que requerem eficiência e flexibilidade.

7- 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.

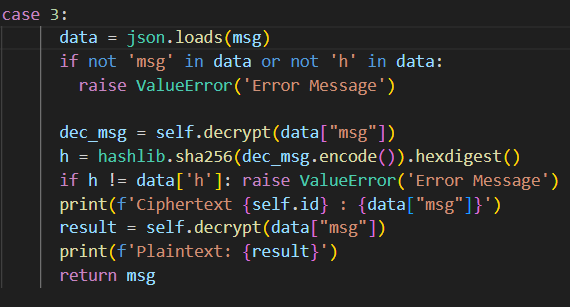


Se uma mensagem for inserida, o cliente criptografa a nova mensagem usando self.encrypt(input\_msg) e calcula o hash SHA-256 dessa nova mensagem.

O cliente retorna a nova mensagem criptografada e o hash em um JSON codificado em bytes para ser enviado ao servidor.

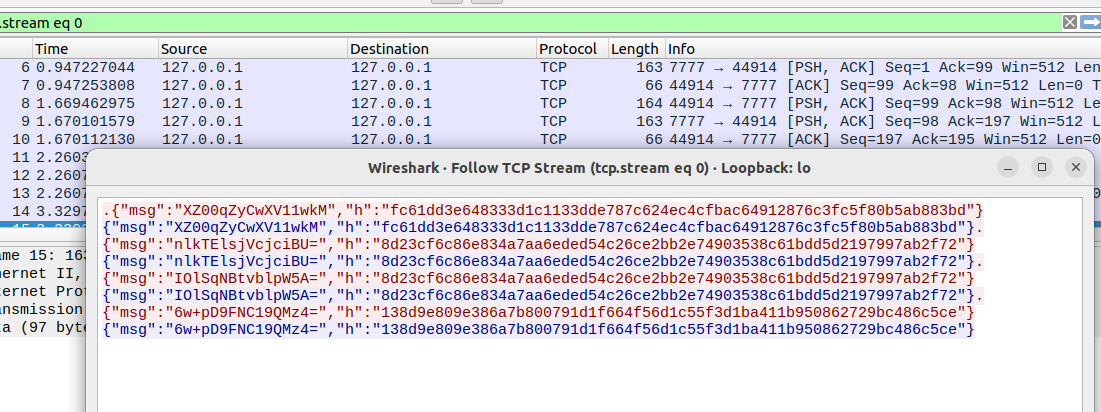


8-Finalmente o servidor vai fazer mesmo processo de verificação e decrypt…



## 5.3 Conclusão

Este método de usar o modo AES CTR e a troca de chaves Diffie-Hellman 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 para um tamanho de bloco fixo, e o algoritmo Diffie-Hellman permite que as partes negociem uma chave compartilhada de maneira segura mesmo através de um canal inseguro. Por fim, a função de hash SHA-256 é usada para verificar a integridade da mensagem, garantindo que os dados não sejam alterados durante a transmissão. No entanto, a geração de chaves pode ser lenta(Demorou 30s para gerir um par das chaves em cada debug!!), e o algoritmo Diffie-Hellman pode estar sujeito a ataques do tipo man-in-the-middle se não forem utilizadas medidas de segurança adicionais durante a troca de chaves.



-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, 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\_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

            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)

        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.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()

                    )

                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:

                  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)

        ctr = Counter.new(64, prefix=nonce)

        cipher = AES.new(key, AES.MODE\_CTR,counter=ctr)

        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('utf-8')

## **6.2 Processos no Cliente/Servidor**

A diferença deste requisito para a requisito 2 é que ele adiciona a geração de chaves RSA e o processo de assinatura de chave publico de DH e verificação de assinaturas.

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

