

Segurança e Auditoria de Sistemas

Atividade 2

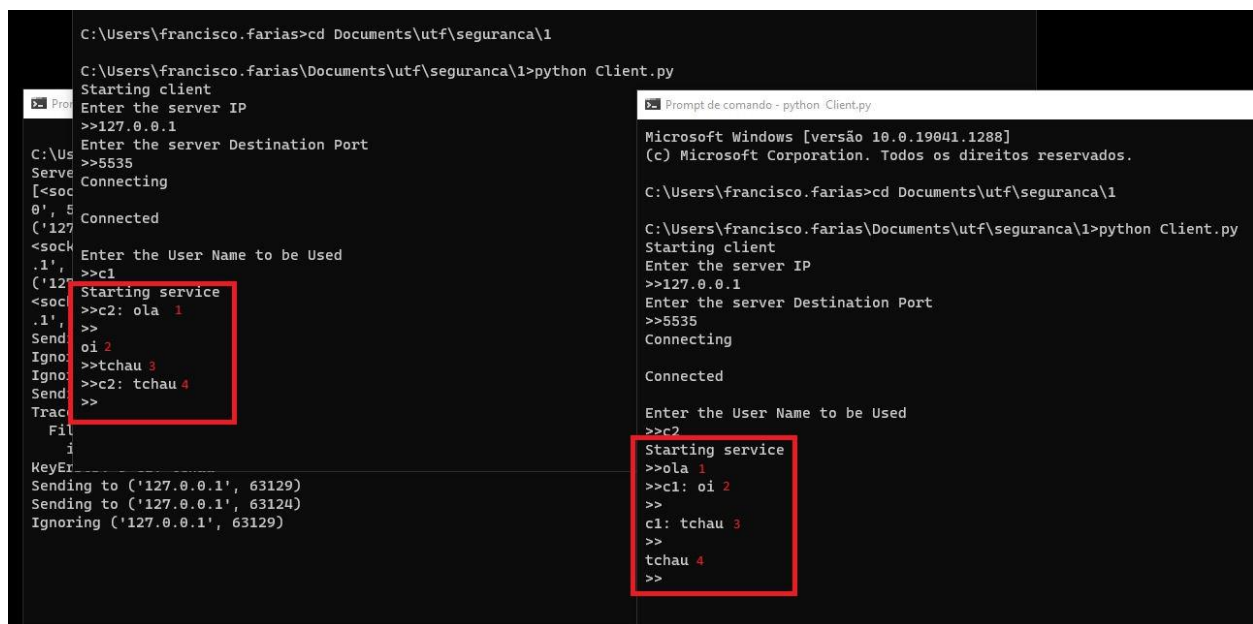
EC38D/C81 - Eng. de Software - UTFPR-CP - 2021/2

Francisco de Carvalho Farias Junior - RA: 1581660

fariasjunior96@gmail.com

A atividade

Foram fornecidos dois scripts em Python, um para o cliente e outro para o servidor, que implementam uma sala de chat peer-to-peer. Mais especificamente, cada cliente conectado a uma instância do servidor consegue enviar mensagens que são transmitidas a todos os outros clientes também conectados àquela instância. Para se conectar basta informar o IP e a porta que o servidor utiliza.



```
C:\Users\francisco.farias>cd Documents\utf\seguranca\1
C:\Users\francisco.farias\Documents\utf\seguranca\1>python Client.py
Starting client
Enter the server IP
>>127.0.0.1
Enter the server Destination Port
>>5535
Connecting
Connected
Enter the User Name to be Used
>>c1
Starting service
>>c2: ola 1
>>
oi 2
>>tchau 3
>>c2: tchau 4
>>
Sending to ('127.0.0.1', 63129)
Sending to ('127.0.0.1', 63124)
Ignoring ('127.0.0.1', 63129)

Microsoft Windows [versão 10.0.19041.1288]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\francisco.farias>cd Documents\utf\seguranca\1
C:\Users\francisco.farias\Documents\utf\seguranca\1>python Client.py
Starting client
Enter the server IP
>>127.0.0.1
Enter the server Destination Port
>>5535
Connecting
Connected
Enter the User Name to be Used
>>c2
Starting service
>>ola 1
>>c1: oi 2
>>
c1: tchau 3
>>
tchau 4
>>
```

Figura 1: Troca de mensagens básica entre cliente e servidor

Na versão original do código as mensagens eram transmitidas em plaintext. Dessa forma, qualquer pessoa que conseguisse capturar pacotes enviados entre dois nós dessa aplicação conseguiria, de maneira trivial, ler o conteúdo das mensagens trafegadas.

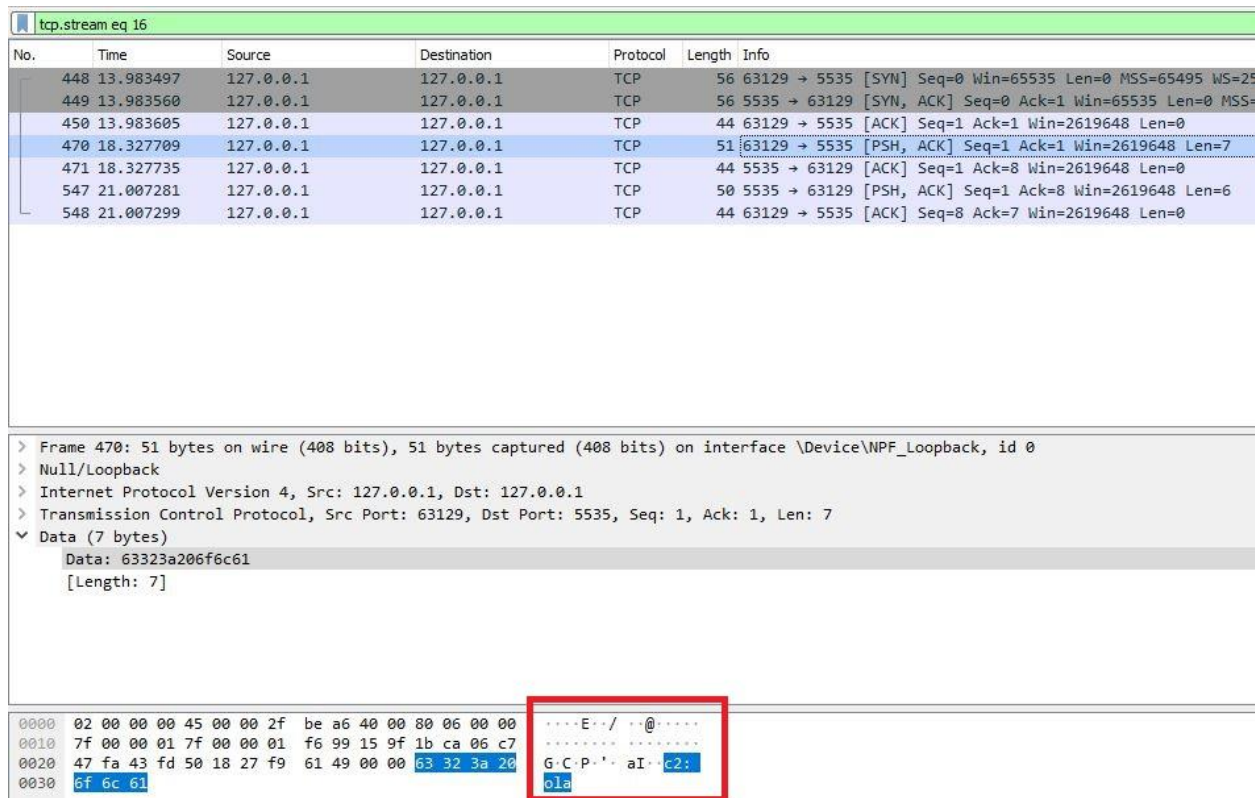


Figura 2: Inspeccionamento dos pacotes capturados durante o envio das mensagens 1 e 2, de acordo com a fig. 1. Em destaque, o envio da mensagem 1 ao servidor, com o conteúdo da mensagem em plaintext.

Dessa forma, a atividade consiste em alterar os scripts implementando uma solução de criptografia que impede que terceiros que capturarem os pacotes consigam decifrar o conteúdo das mensagens sem impedir o funcionamento normal do chat. Além disso, deve ser garantida a autenticidade e integridade das mensagens trocadas.

Retrospectiva da atividade 1

O código dessa solução foi construído com base na solução da Atividade 1. Nela, foi implementada criptografia simétrica através do algoritmo AES usando a biblioteca Cryptography. A chave secreta era pré-gerada e embutida no código do cliente, de maneira que as mensagens trafegavam de maneira ofuscada pela rede e servidor, e eram decriptadas nos clientes.

A maior limitação dessa solução do ponto de vista de segurança é que qualquer indivíduo com posse de uma cópia do cliente podia extrair a chave do código, e assim decifrar mensagens capturadas em trânsito, desde que fossem geradas por essa mesma versão do cliente. Além disso, a solução não possuía nenhum tipo de checagem de autenticidade ou integridade.

```

9
10 # >> libs usadas
11 import os
12 from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
13
14 # >> chave secreta
15 KEY = b'\xf2pz\x06)\x13\xe\x9c\xd9\x94\xd8\n\x98u\xbfB\xb8\xf4*\xe4\xe4e\xe5\xf9l,\x87\xf3d\x88\x99'
16
17 # >> tamanho do vetor de inicialização, em bytes
18 IV_SIZE = 16
19
20 # >> o tamanho da mensagem passada tem que ser múltiplo de 128 (AES). essa função "enche linguíça"
21 def fill_msg_length(msg):
22     length = len(msg)
23     fill = 128 - (length % 128)
24     return msg + ("\x00".encode() * fill)
25
26 ## >> implementação da encriptação; vetor de inicialização é passado no começo da mensagem
27 def encrypt(msg):
28     iv = os.urandom(IV_SIZE)
29     cipher = Cipher(algorithms.AES(KEY), modes.CBC(iv))
30     encryptor = cipher.encryptor()
31     encrypted_msg = encryptor.update(fill_msg_length(msg.encode())) + encryptor.finalize()
32     return iv + encrypted_msg
33
34 # >> implementação da decriptação; pega o vetor de inicialização no começo da mensagem
35 def decrypt(chunk):
36     iv = chunk[0:IV_SIZE]
37     encrypted_msg = chunk[IV_SIZE:]
38     cipher = Cipher(algorithms.AES(KEY), modes.CBC(iv))
39     decryptor = cipher.decryptor()
40     decrypted_msg = decryptor.update(encrypted_msg) + decryptor.finalize()
41     return decrypted_msg
42

```

Figura 3: A solução da atividade 1, com a chave secreta embutida no código.

A solução

Para solucionar esse problema foi implementada uma estratégia híbrida, onde durante um handshake são trocadas chaves públicas RSA, e após isso o servidor gera uma chave simétrica AES e manda para o cliente, encriptada e assinada. Isso garante **confidencialidade** e **autenticidade** na comunicação. Após isso, toda comunicação é ofuscada usando a chave simétrica, com a autenticidade implícita na ideia de que, se a mensagem foi decifrada com sucesso, ela foi encriptada pela mesma parte que enviou as chaves simétrica e pública durante o handshake.

O benefício da abordagem híbrida está no fato de que ele herda a segurança e autenticidade que a criptografia assimétrica disponibiliza, mas com a performance e flexibilidade no tamanho do payload durante a comunicação regular do algoritmo AES.

Por último, todas as mensagens enviadas possuem o hash extraído, com o mesmo sendo verificado no recebimento, garantindo assim a **integridade** da comunicação.

Tecnologias

- Python 3.10
- Implementação de criptografias e hashing da biblioteca *cryptography*
- Criptografia simétrica AES, com chaves de 32 bytes e vetor de inicialização de 16 bytes
- Criptografia assimétrica RSA, com chaves de 2048 bytes
- Hashing SHA256

O handshake

O handshake é iniciado pelo cliente no momento da inicialização. Para isso ele simplesmente envia a sua chave pública para o servidor.

```
def initiate_handshake(self) -> None:
    '''To be ran at startup; sends the pub key to the server, initiating the handshake'''
    print('Initiating handshake with the server')
    msg = Message(MessageType.PUB_KEY_EXCHANGE, self.Keys.asym.serialized_pub())
    self.client(self.__HOST, self.__PORT, msg.serialize())
    return
```

Figura 4: O cliente inicia o handshake ao enviar sua chave pública ao servidor

Ao recebê-la, o servidor interpreta isso como uma tentativa de iniciar o handshake. Assim, ele realiza as seguintes ações:

1. Verifica a integridade da mensagem;
2. Envia sua chave pública para esse cliente (junto do hash dela);
3. Gera uma chave AES que só será compartilhada entre o servidor e essa instância específica do cliente;
4. Salva ambas a) chave pública do cliente e b) chave simétrica em uma estrutura de dados interna, associando-as com o endereço de origem;
5. Criptografa a chave AES usando a chave pública do cliente;
6. Assina a chave AES criptografada usando sua chave privada;
7. Envia a chave AES, encriptada, assinada e com um hash do payload.

Para extrair a chave AES o usuário deve decriptá-la usando sua chave privada, o que garante a sua confidencialidade. Além disso o usuário pode verificar a sua autenticidade, garantindo assim que quem a enviou possui a chave privada par da chave pública enviada no passo 2.

```

def do_handshake(self, ip: tuple, serialized_pub_key: bytes) -> None:
    '''Saves a client's pub key, and sends back its own, along with a symmetric key (signed, encrypted and hashed)'''
    print('Handshaking with client ' + str(ip))

    client_pub_key = RSA.deserialize_pub_key(serialized_pub_key)

    pub_key_msg = Message(MessageType.PUB_KEY_EXCHANGE, Keys.asym.serialized_pub())
    2 self.add_msg_to_queue(ip, pub_key_msg)

    3 sym_key = AES.gen_key()
    4 Keys.setForIp(ip, KeySet(client_pub_key, sym_key))

    5,6 encrypted_sym_key = RSA.sign(RSA.encrypt(sym_key, client_pub_key), Keys.asym.pvt)
    # RSA has its own hash, but to simplify our code we'll just hash again
    sym_key_msg = Message(MessageType.SYM_KEY_EXCHANGE, encrypted_sym_key)
    7 self.add_msg_to_queue(ip, sym_key_msg)

    print('Handshake complete with client ' + str(ip))
    return

```

Figura 5: Rotina de handshake do lado do cliente. Métodos de hashing então em uma classe separada

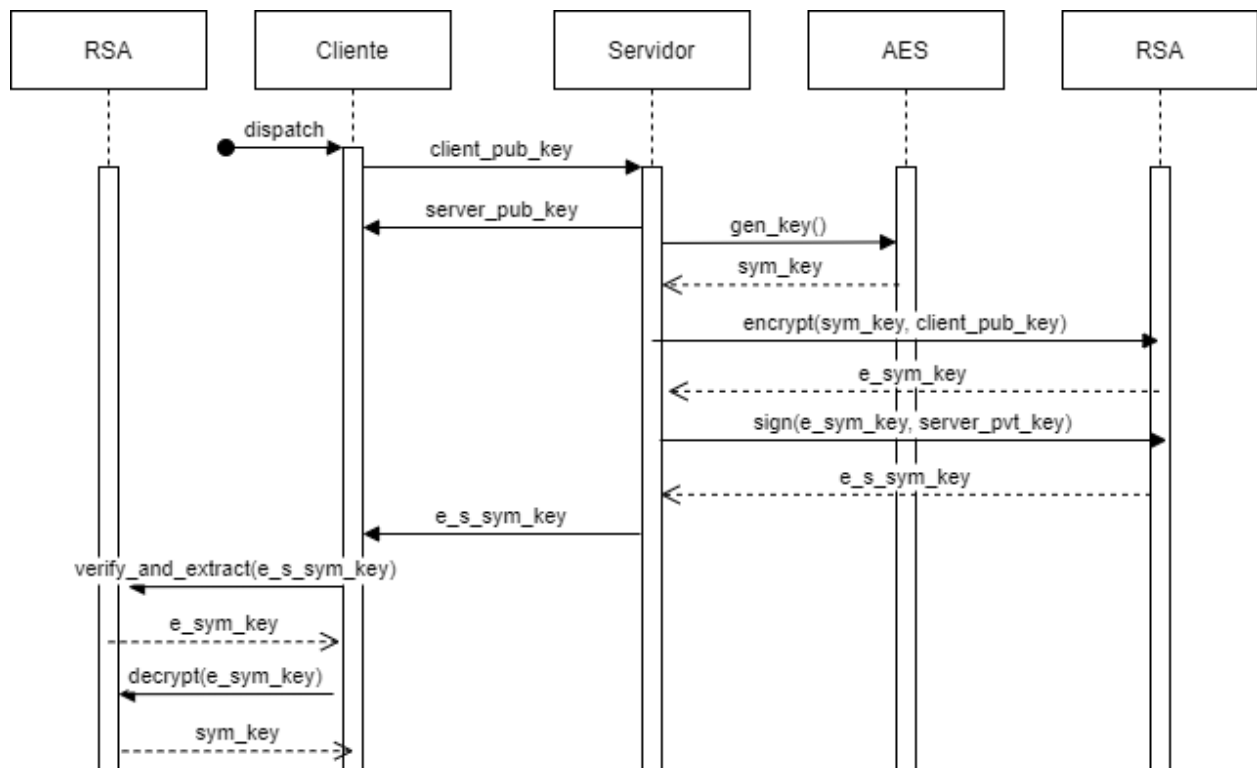


Figura 6: Diagrama de sequência ilustrando a comunicação entre cliente e servidor, além das suas classes de criptografia

Comunicação comum

A comunicação continua seguindo o modelo de broadcast simples da implementação original, onde uma mensagem enviada por um cliente é repassada para todos os outros clientes conectados àquela instância do servidor. A diferença é que, após o handshake, cada cliente passa a compartilhar uma chave AES com o servidor. Dessa forma, cada mensagem enviada é

descriptografada no servidor e encryptada novamente uma vez para cada um dos outros clientes.

```
def send_msg(self, content: str) -> None:
    '''Encrypts a regular message using AES, hashes it and sends it to the server'''
    if self.Keys.sym_key == None:
        print("reconnecting to server...")
        self.initiate_handshake()
        return

    encrypted_content = AES.encrypt(content.encode(), self.Keys.sym_key)
    msg = Message(MessageType.NORMAL, encrypted_content)
    self.client(self.__HOST, self.__PORT, msg.serialize())
    return
```

Figura 7: Rotina de envio de mensagens do lado do cliente, com encriptação simétrica

```
msg = Message.deserialize(chunk)
try:
    keys = self.__CLIENT.Keys
    match msg.type:
        case MessageType.ASK_FOR_PUB_KEY:
            print('WARN: Server asked for pub key')
            self.__CLIENT.initiate_handshake()
            continue
        case MessageType.PUB_KEY_EXCHANGE:
            keys.srv_pub_key = RSA.deserialize_pub_key(msg.content)
            continue
        case MessageType.SYM_KEY_EXCHANGE:
            keys.sym_key = RSA.decrypt(RSA.verify_and_extract(msg.content, keys.srv_pub_key), keys.asym.pvt)
            continue
        case MessageType.NORMAL:
            content = AES.decrypt(msg.content, keys.sym_key)
            print(content.decode() + '\n>>', end='')
            continue
except Exception as e:
    # self.__CLIENT.initiate_handshake()
    print('exception!')
    print(e)
    return
```

Figura 8: Rotina de recebimento de mensagens do lado do cliente. Ao receber uma mensagem tipo=NORMAL ela é descriptografada usando a chave AES que o cliente compartilha com o servidor.

```

msg = Message.deserialize(chunk)
print('> new message ' + str(msg.type))
match msg.type:
    case MessageType.PUB_KEY_EXCHANGE:
        self.do_handshake(origin, msg.content)
        continue
    case MessageType.NORMAL:
        msg.content = AES.decrypt(msg.content, Keys.getForIp(origin).sym)
        self.add_msg_to_queue(origin, msg)
        continue

```

Figura 9: Rotina de recebimento de mensagens do lado do servidor. Caso ela tenha tipo=NORMAL ela também descripta-a com a chave AES que ela compartilha com o cliente em questão.

```

def route_msg(self, destination: socket.socket, msg: Message) -> None:
    '''Sends a message to destination. If the message type=NORMAL, encrypts it with the destination's sym key.'''

    print('< sending type ' + str(msg.type))

    if msg.type == MessageType.NORMAL:
        try:
            sym_key = Keys.getForIp(destination.getpeername()).sym
            msg.content = AES.encrypt(msg.content, sym_key)
        except:
            msg = Message(MessageType.ASK_FOR_PUB_KEY)

    destination.send(msg.serialize())

```

Figura 10: Rotina de envio de mensagens do lado do servidor, onde, caso ela tenha tipo=NORMAL, ela é novamente encriptada, dessa vez com a chave do endereço de destino.

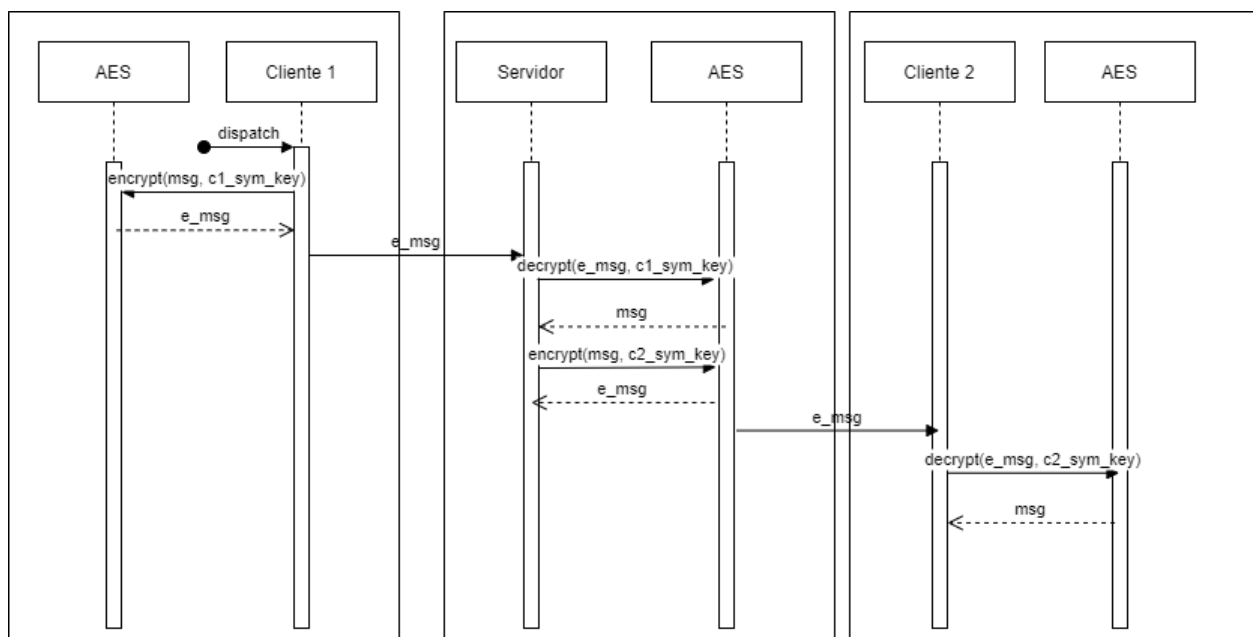


Figura 11: Diagrama de sequência descrevendo a comunicação comum entre dois clientes. Nota: Caso existam mais de 2 clientes conectados, os restantes se comportarão da mesma forma que o Cliente 2.

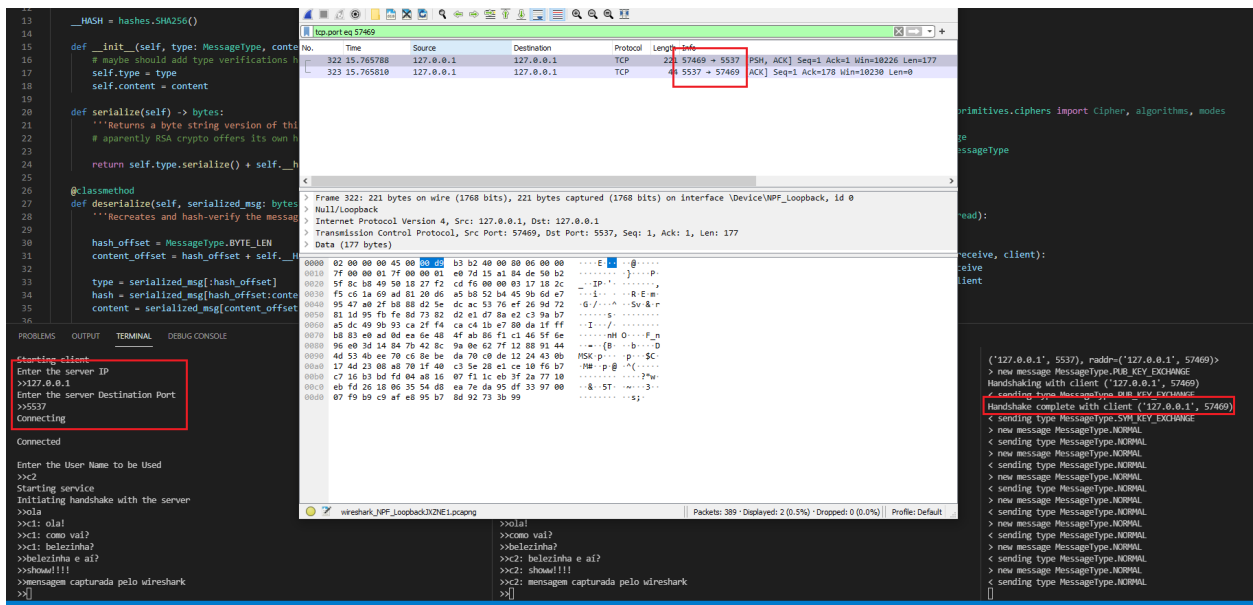


Figura 12: Captura do Wireshark, mostrando a comunicação ofuscada entre um cliente e o servidor.

Hashing

O sistema de hashing, que é o mecanismo que garante a integridade na comunicação, foi pouco discutido até agora pois ele está presente em uma classe separada.

Todas as mensagens são representadas pela classe *Message*. Ela carrega como informação o conteúdo e o tipo da mensagem, esse último usado no momento do processamento. Além disso, ela oferece métodos de serialização e desserialização, que permitem converter a mensagem para bytes que serão transmitidos de um nó para outro. Durante esse processo é gerado um hash da mensagem, e esse é adicionado ao começo da mensagem serializada.

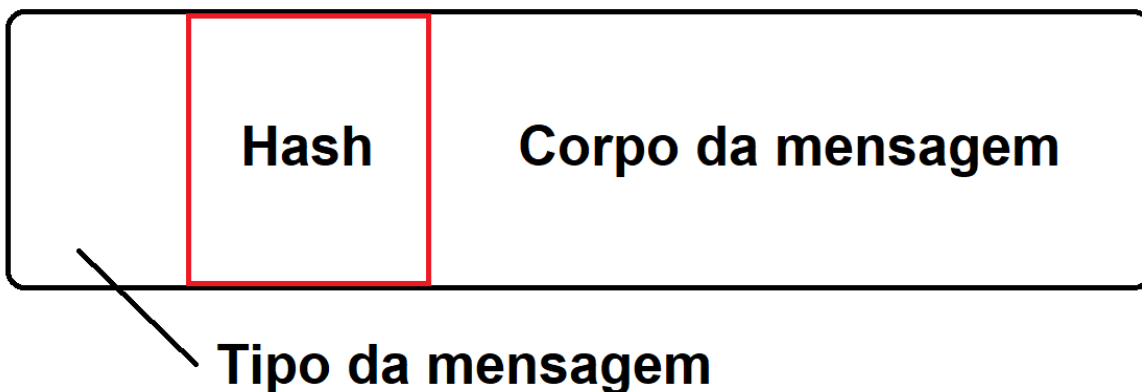


Figura 13: Diagrama ilustrando a estrutura de uma mensagem serializada. Nota: Quando a mensagem é assinada usando RSA a assinatura faz parte do corpo da mensagem.

Ao receber a mensagem, o nó a desserializa. Nesse processo o hash é extraído para que assim possa ser verificada a integridade da mensagem.

```
def serialize(self) -> bytes:
    '''Returns a byte string version of this message with hashing, to be sent to a different node'''
    # apparently RSA crypto offers its own hashing, but for simplicity we'll use this method for all serialization

    return self.type.serialize() + self.__hash(self.content) + self.content

@classmethod
def deserialize(self, serialized_msg: bytes) -> 'Message':
    '''Recreates and hash-verify the message from a byte string created with serialize()'''

    hash_offset = MessageType.BYTE_LEN
    content_offset = hash_offset + self.__HASH.digest_size

    type = serialized_msg[:hash_offset]
    hash = serialized_msg[hash_offset:content_offset]
    content = serialized_msg[content_offset:]

    if (hash != self.__hash(content)):
        raise RuntimeError('Hash verification failed')

    return Message(MessageType.deserialize(type), content)
```

Figura 14: Métodos de serialização e desserialização; o primeiro gerando e adicionando o hash à string de bytes, e o segundo extraíndo-o e verificando a integridade da mensagem.

Limitações da solução

- Essa implementação oferece criptografia cliente-servidor, ou seja, quem tiver acesso ao servidor consegue ler o conteúdo das mensagens. Uma solução ponta-a-ponta não possuiria esse problema.
- Nada na minha implementação impede que um ator mal-intencionado que consiga capturar o handshake possa forjar um servidor, fazendo o cliente acreditar que está se comunicando com um servidor legítimo.
- O código como um todo foi escrito como exercício, e portanto não sabe lidar com falhas e comportamentos inesperados como um todo. Isso fornece superfícies de ataque em potencial.
- O tipo da mensagem não faz parte do payload serializado. Na chance dele ser corrompido, isso não será identificado na desserialização. O código precisa ser reestruturado para permitir checar o hash antes de desserializá-lo.

Referências utilizadas

https://en.wikipedia.org/wiki/Hybrid_cryptosystem

<https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/>

<https://cryptography.io/en/latest/hazmat/primitives/cryptographic-hashes/>

<https://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/>