

SSI - TP1 - G21 - Relatório

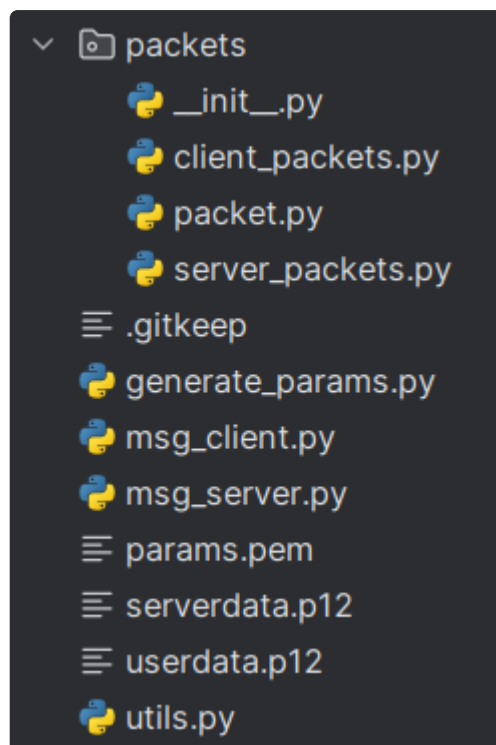
Projecto de Criptografia Aplicada (TP1)

Este relatório é relativo ao trabalho prático nº 1 da UC Segurança de Sistemas Informáticos realizado por:

- Rodrigo Monteiro, a100706
- Diogo Abreu, 100646

No qual é construído um serviço de *Message Relay* que permite aos membros de uma organização trocarem mensagens com garantias de autenticidade.

Estrutura do código



Packets

`packets.py`

Primeiramente, decidimos definir estruturas de dados genéricas para os pacotes que são enviados entre os clientes e o servidor, e que fazem parte do

protocolo definido.

Portanto, definimos as classes `PacketType` e a classe `Packet` que possui um `PacketType` como atributo.

```
class PacketType(IntEnum):
    @abstractmethod
    def __str__(self) -> str:
        return self.name

class Packet:
    def __init__(self, packet_id: int, packet_type: PacketType):
        self.id = packet_id
        self.type = packet_type
```

Para além disso, contém também superclasses base para as subclasses de serialização e deserialização de *packets* (como `ClientPacketSerializer` e `ServerPacketSerializer`).

```
class PacketSerializer:
    @staticmethod
    @abstractmethod
    def serialize(packet: Union[Type[Packet], Packet]) -> bytes:
        raise NotImplementedError

class PacketDataDeserializer:
    @staticmethod
    @abstractmethod
    def deserialize(data: bytes) -> Union[Type[Packet], Packet]:
        raise NotImplementedError
```

Assim, os métodos de serialização e deserialização serão agrupados em classes próprias. Caso os métodos fossem implementados nas subclasses de `Packet`, estes seriam úteis se fossem `static`, o que não resultaria com herança, e levaria a repetição de código.

Também possui as classes `EncryptedPacketSerializer`, e `EncryptedPacketDeserializer`, que serão explicadas posteriormente.

client_packets.py

Contém diversas classes que definem os pacotes do protocolo usado.
Exemplo:

```
class ClientPacketType(PacketType):
    ASK_QUEUE = 1
    # ...

class ClientAskQueuePacket(Packet):
    def __init__(self, packet_id: int) -> None:
        super().__init__(packet_id, ClientPacketType.ASK_QUEUE)

class ClientPacketSerializer(PacketSerializer):
    @staticmethod
    def serialize(packet: Packet) -> bytes:
        format_string = '!BH'
        flat_data = [packet.type.value, packet.id]

        match packet.type:
            case ClientPacketType.ASK_QUEUE:
                return ClientPacketSerializer
                    ._serialize_ask_queue(format_string,
flat_data)
```

- **SEND:**
 - **UID** do cliente a quem se envia a mensagem
 - *max length*: 255 bytes
 - **subject**
 - *max length*: 255 bytes
 - **body**
 - *max length*: 65535 bytes)
 - **signature** da mensagem a enviar (UID, subject e body) com a chave RSA privada
 - *max length*: 65535 bytes
 - **encrypted_symmetric_key**: uma chave simétrica é gerada e utilizada para encriptar a mensagem a enviar (UID, subject e body), sendo depois enviada mas encriptada com a chave pública do cliente a quem se envia a mensagem

- *max length*: 65535 bytes
- **ASKQUEUE** : (sem informação adicional)
- **GET_MSG** :
 - **msg_num** : número da mensagem na *queue* (4 bytes)
- **PARAMS** :
 - **params** : parâmetros para a geração de chaves DL
 - *max length*: 65535 bytes
- **PUBLIC_KEY** :
 - **public_key**
 - *max length*: 65535 bytes
- **HANDSHAKE** :
 - **signature** ;
 - *max length*: 65535 bytes
 - **certificate** com a chave pública para verificação da assinatura;
 - *max length*: 65535 bytes
 - **salt** : são usados 16 bytes aleatórios, mas podem ser usados até 255 bytes
- **GET_CERT** :
 - **UID** de um cliente
 - *max length*: 65535 bytes

server_packets.py

- **QUEUE_INFO** :
 - **state** : *empty* ou *not empty* (1 byte)
 - **num_elements** : caso o estado seja *not empty* (4 bytes)
- **QUEUE_ELEMENT** : pacotes seguidos do **QUEUE_INFO**
 - **msg_num** : 4 bytes
 - **sender_uid** :
 - *max length*: 255 bytes
 - **timestamp** : 4 bytes
 - **subject** :
 - *max length*: 65535 bytes
 - **encrypted_symmetric_key** :
 - *max length*: 65535 bytes

- **MSG**: enviado em resposta a um pedido **GET_MSG** de um cliente
 - **status**: *ok, not_found*
 - **sent_by**: *encrypted*
 - *max length*: 255 bytes
 - **timestamp**: 4 bytes
 - **subject**: *encrypted*
 - *max length*: 255 bytes
 - **body**: *encrypted*
 - *max length*: 65535 bytes
 - **signature**
 - *max length*: 65535 bytes
 - **cert** do cliente que enviou a mensagem
 - *max length*: 65535 bytes
 - **encrypted_symmetric_key**
 - *max length*: 65535 bytes
- **STATUS**:
 - **status**: *ok* ou *error* (1 byte)
- **HANDSHAKE**:
 - **public_key**:
 - *max length* 65535 bytes
 - **signature**:
 - *max length* 65535 bytes
 - **certificate**:
 - *max length* 65535 bytes
- **SEND_CERT**:
 - **certificate**:
 - *max length* 65535 bytes

Funcionamento básico do `msg_server.py`

O Server funciona com um socket TCP do tipo `SOCK_STREAM`, e cria uma *thread* por cada conexão com um cliente.

```
def run(self) -> None:
    self.print('Starting server ...')
```

```

        self.sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        # ...
        while not self.done:
            try:
                client, address = self.sock.accept()
                thread = Thread(target=self.listen_to_client,
                                args=(client,
address))
                self.threads.append(thread)
                thread.start()
            except socket.timeout:
                # ...

```

Para além disso, possuí as classes:

- `Message` : guarda as mensagens recebidas pelos clientes, para depois as redirecionar para os clientes corretos, quando estes realizam o comando `askqueue` ou `getmsg`.
- `ClientMessages` : dá IDs às mensagens - usados no comando `getmsg` - e mantém um dicionário de mensagens lidas e outro de mensagens não lidas em que chaves são os IDs e os valores são as mensagens.

```

self.msgs: Dict[str, ClientMessages] =
defaultdict(ClientMessages)

```

```

class ClientMessages:
    def __init__(self) -> None:
        self.read_msgs: Dict[int, Message] = {}
        self.unread_msgs: Dict[int, Message] = {}
        self.next_id: int = 0
    def add_unread_msg #...
    def read_unread_msg(self, num: int) -> Message | None: # ...

```

Funcionamento básico do `msg_client.py`

- Função `main`
 - Esta função lê os argumentos passados ao programa e executa os métodos necessários - que são estáticos e encontram-se na classe

ClientController. Exemplo:

```
if len(argv) > 1 + p and argv[1 + p] == 'askqueue':
    c = Command(Command.Type.ASK_QUEUE, [])

# ...

client_thread = Thread(target=client.run)
client_thread.start()

match c.command_type:
    ClientController.get_queue_elements(client)
```

- Caso os argumentos estejam errados, imprime um erro para o `sys.stderr`. Caso não possua argumentos entra num modo de "terminal", correndo o `ClientController`



```
rodrigo@pop-os: ~/Desktop/SSI-GIT/TPs/TP1
> python3 msg_client.py
> send MSG_CLI2 "ola" "bom dia"
>

rodrigo@pop-os: ~/Desktop/SSI-GIT/TPs/TP1
> python3 msg_client.py -user info/MSG_CLI2.p12
> askqueue
0:MSG_CLI2:2024-03-29 22:09:39:ola
> getmsg 0
MSG_CLI1:2024-03-29 22:09:39:ola:bom dia
>
```

- Classe `ClientController`

```
class ClientController:
    def __init__(self, msg_client: Client) -> None:
        self.commands = # ...

    def run(self) -> None:
        while not self.done and not self.client.done:
            try:
                command = input('> ').strip()
                if self.done:
                    break
                self.parse_command(command)
            except KeyboardInterrupt:
```

```

# ...

def parse_command(self, command: str):
    for regex, func in self.commands:
        if match := regex.match(command):
            func(match)
    return

@staticmethod
def send_msg(client: Client, send_to: str, subject: str,
body: str, timeout: int = 3) -> None:
    # ...

@staticmethod
def get_asked_msg(client: Client, timeout: int = 3) ->
None:
    # ...

@staticmethod
def check_lengths(subject: str, body: str) -> bool:
    # ...

@staticmethod
def get_queue_elements(client: Client,
timeout_info: int = 3, timeout_element: int = 3) ->
None:
    # ...

```

Os métodos são estáticos de modo a que possam ser usados fora do modo de "terminal"

- Classe `Client`
 - Ao iniciar, conecta ao servidor, efetua o *handshake*, e de seguida fica a ler os *packets* enviados pelo servidor, deserializando-os. Também possui métodos para tratar dos pacotes recebidos, e para enviar pacotes.

```

while not self.done:
    try:
        p = EncryptedPacketDeserializer.deserialize(
            self.socket, self.handshake_info.aes_key,
            self.handshake_info.hmac_key,
            ServerPacketDataDeserializer

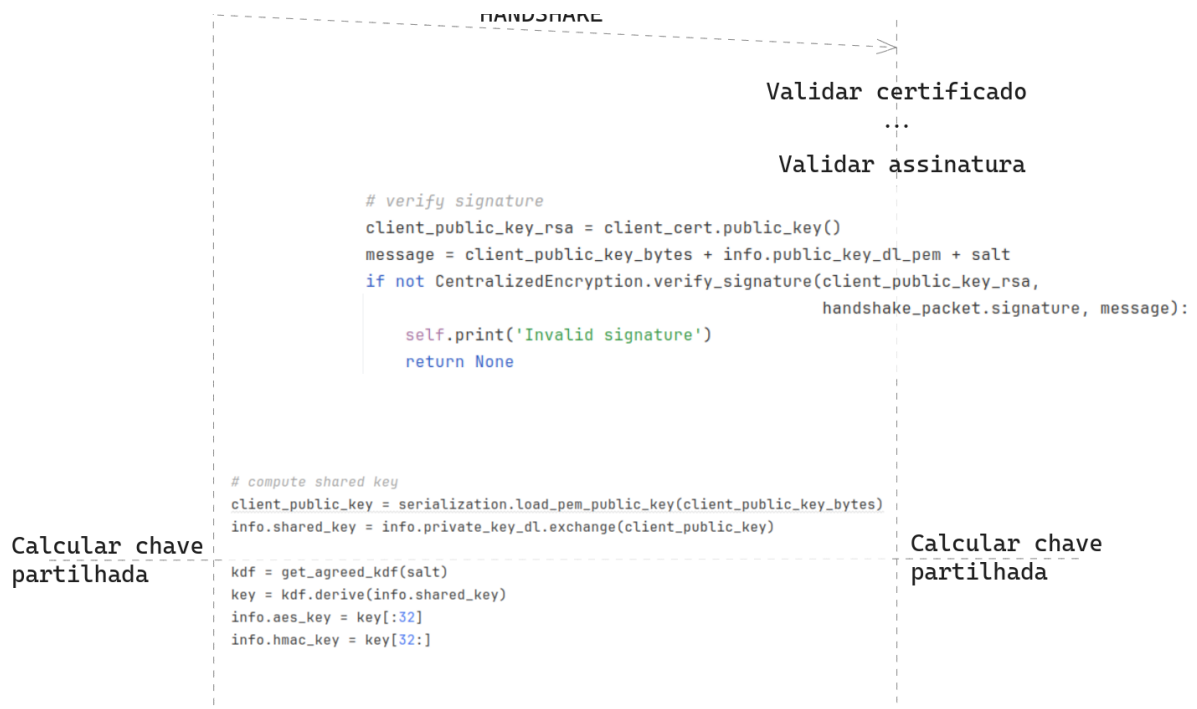
```


)

...

Handshake inicial





- Pode-se confiar na chave pública do Server, por exemplo, pois esta está no certificado enviado que é comprovado por uma *trusted third party*, CA.
- Caso, por exemplo, a chave pública do servidor (DL) for modificada no caminho, esta não será igual à que está na assinatura e, portanto, não será possível validar a assinatura.
- Assim, para além de um *intruder* nunca conseguir saber qual é a chave partilhada, também não conseguirá realizar ataques de *man-in-the-middle*.
- Com recurso à KDF utilizada, foram pedidos mais 32 bytes para se obter uma chave simétrica para o MAC - nunca se devem reutilizar chaves criptográficas para fins distintos.
- Decidimos incluir o salt na assinatura, para assegurar que este não é modificado - e caso seja, é detetado - e que assim o cliente e o servidor chegam de certeza à mesma chave. Alternativamente, o cliente poderia encriptar o salt com a chave pública do servidor, e tal seria possível pois o salt neste caso é de pequena dimensão.

Confidencialidade e Integridade

Para assegurar a confidencialidade (em relação ao "exterior") e integridade dos pacotes enviados entre os clientes e o servidor, é utilizado *encrypt-then-mac* (onde os bytes passam originalmente pela cifra, e o MAC é calculado já sobre o criptograma):

1. Serializar o pacote e gerar um NONCE
2. Utilizar AES no modo CTR para encriptar o pacote
3. Calcular o MAC utilizando o HMAC, definido sobre a função de hash SHA256 , do NONCE + pacote encriptado
4. Enviar o NONCE + pacote encriptado + MAC

```
class EncryptedPacketSerializer:
    @staticmethod
    def serialize(packet: Packet, aes_key, hmac_key, serializer:
        Type[PacketSerializer], private_key=None, public_key=None):

        packet_bytes = serializer.serialize(packet,
                                           private_key=private_key,
public_key=public_key)
        nonce = os.urandom(16)

        algorithm = algorithms.AES(aes_key)
        cipher = Cipher(algorithm, modes.CTR(nonce))
        encryptor = cipher.encryptor()

        encrypted_packet = encryptor.update(packet_bytes)
                                +
encryptor.finalize()
        len_encrypted_packet = struct.pack('!I',
len(encrypted_packet))

        h = hmac.HMAC(hmac_key, hashes.SHA256())
        h.update(nonce + len_encrypted_packet
                                + encrypted_packet)
        tag = h.finalize()

        return nonce + len_encrypted_packet + encrypted_packet
                                + tag
```

Comando `send`

Primeiramente, é necessário pedir o certificado do cliente, a quem se quer enviar a mensagem, ao servidor e validar esse certificado. De seguida, é enviada a mensagem, através de alguns passos.

Objetivos:

- **Autenticidade através da assinatura**
- **Confidencialidade (perante um servidor "curioso") através de um `hybrid cryptosystem`**

Passos:

1. Serializar a mensagem (UID, *subject* e *body*)
2. Assinar a mensagem com a própria chave privada de modo a garantir autenticidade, gerando uma assinatura
3. Gerar uma chave simétrica (com `Fernet`)
4. Encriptar o *subject* e o *body* com a chave simétrica
5. Encriptar a chave simétrica gerada com a chave pública do cliente a quem se envia a mensagem
6. Enviar a mensagem serializada com o *subject* e o *body* encriptados, juntamente com a assinatura, e com a chave simétrica encriptada

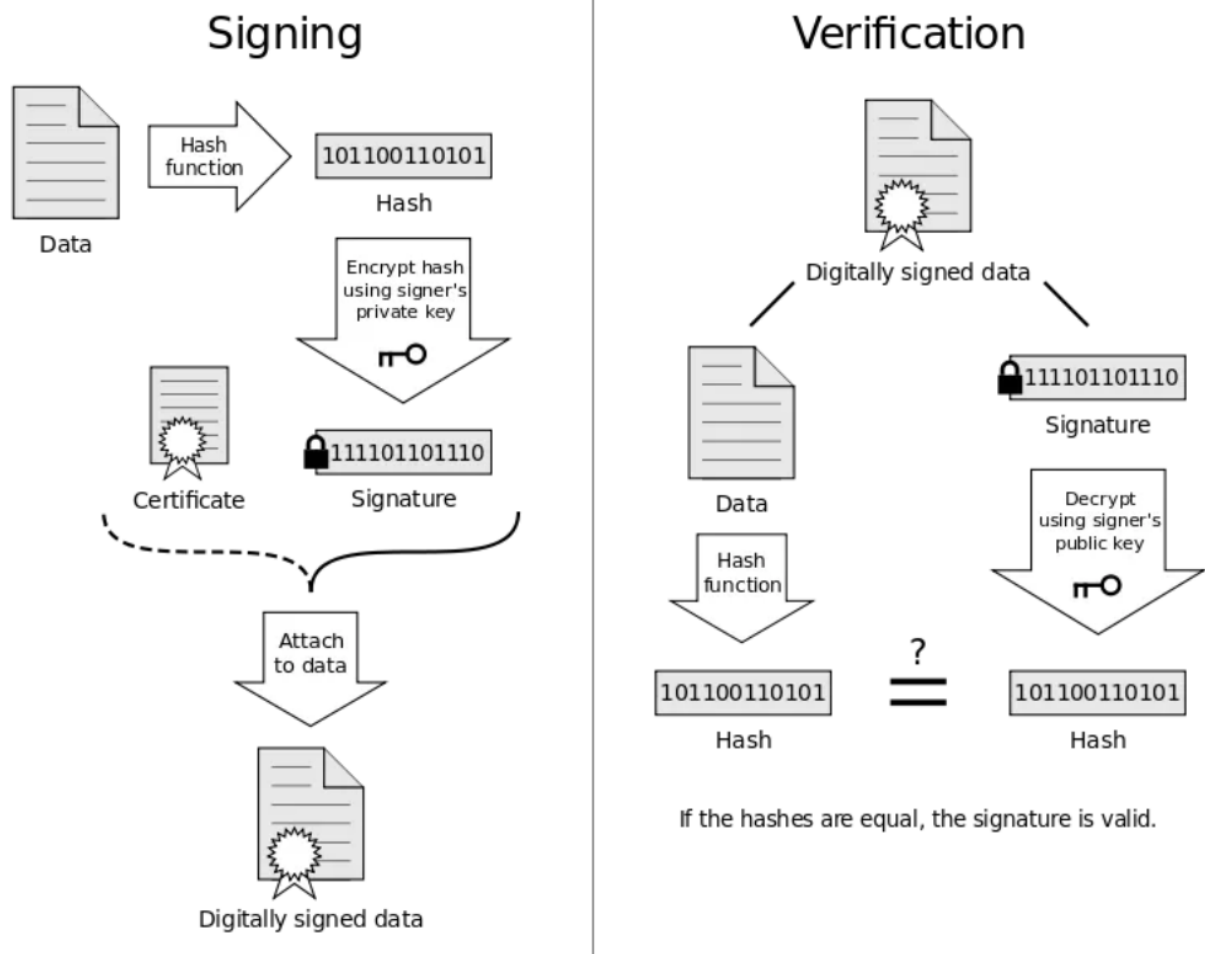
```
def sign_msg(private_key_rsa, peer_public_key, uid: str,
              subject: str, body: str) -> bytes:

    msg_bytes = CentralizedEncryption._serialize_msg(...)
    sig = CentralizedEncryption.sign_bytes(private_key_rsa,
msg_bytes)
    # ...
    symmetric_key = Fernet.generate_key()
    f = Fernet(symmetric_key)
    encrypted_symmetric_key = CentralizedEncryption
                              .encrypt_bytes(peer_public_key,
symmetric_key)

    encrypted_subject = f.encrypt(subject_bytes)
    encrypted_body = f.encrypt(body_bytes)

    serialized_msg_bytes = CentralizedEncryption
                              ._serialize_msg(uid_bytes,
encrypted_subject,
encrypted_body)
```

```
return serialized_msg_bytes + len_and_sig
      + len_and_enc_sym_key
```



Source: https://en.wikipedia.org/wiki/Electronic_signature

Comando `getmsg`

1. Validar certificado
2. Validar assinatura
 1. Desencriptar chave simétrica com a própria chave privada
 2. Desencriptar o *subject* e o *body* utilizando a chave simétrica
 3. Serializar a mensagem novamente com os dados desencryptados de modo a se poder validar a assinatura

```
def verify_signed_msg(public_key_rsa, private_key,
uid: bytes, subject, body, sig, f_key):
```

```
    symmetric_key = CentralizedEncryption
```

```

        .decrypt_bytes(private_key, f_key)

    f = Fernet(symmetric_key)
    decrypted_subject = f.decrypt(subject)
    decrypted_body = f.decrypt(body)
    msg_bytes = CentralizedEncryption
        ._serialize_msg(uid, decrypted_subject,
decrypted_body)

    if not CentralizedEncryption
        .verify_signature(public_key_rsa, sig,
msg_bytes):
        return None
    else:
        return decrypted_subject, decrypted_body

```

Comando `askqueue`

Quando é enviado um pacote `ASKQUEUE` o servidor responde sempre primeiro com um pacote do tipo `QUEUE_INFO` de modo a informar se a *queue* está vazia ou não, e, se não estiver, informa também acerca do tamanho da *queue*, i.e., dos pacotes que irá então mandar.

```

def get_queue_elements(client: Client, timeout_info: int = 3,
                        timeout_element: int
= 3) -> None:
    client.send_ask_queue_packet()
    queue_info: ServerQueueInfoPacket = client
        .responses['queue_info']
        .get(timeout=timeout_info, block=True)

    if queue_info.state == ServerQueueInfoPacket.State.EMPTY:
        return

    count = 0
    queue_elements = []
    while count < queue_info.num_elements:
        try:
            e: ServerQueueElementPacket = client
                .responses['queue_elements']

```

```

        .get(timeout=timeout_element,
block=True)

    except Empty:
        break

    # ...

```

Funcionamento

- Comandos (Servidor)

```

rodrigo@pop-os: ~/Desktop/SSI-GIT/TPs/TP1
rodrigo@pop-os:~/Desktop/SSI-GIT/TPs/TP1$ python3 msg_server.py --help
usage: msg_server.py [-h] [-p PORT] [-d] [-m MAX] [-t TIMEOUT] [-D DATA]

MSG Server Command Line Options

options:
  -h, --help            show this help message and exit
  -p PORT, --port PORT  Port to bind the server to
  -d, --debug            Enable debug mode
  -m MAX, --max MAX     Maximum number of connections
  -t TIMEOUT, --timeout TIMEOUT
                        Timeout for connections
  -D DATA, --data DATA Server data file path
rodrigo@pop-os:~/Desktop/SSI-GIT/TPs/TP1$

```

- Comandos (Cliente)

```

rodrigo@pop-os: ~/Desktop/SSI-GIT/TPs/TP1
rodrigo@pop-os:~/Desktop/SSI-GIT/TPs/TP1$ python3 msg_client.py help
Usage:
  msg_client.py [-user <filename>] send <uid> <subject>
  msg_client.py [-user <filename>] askqueue
  msg_client.py [-user <filename>] getmsg <num>
  msg_client.py [-user <filename>] help
rodrigo@pop-os:~/Desktop/SSI-GIT/TPs/TP1$

```

- Exemplo de utilização (1)

```

rodrigo@pop-os: ~/Desktop/SSI-GIT/TPs/TP1
rodrigo@pop-os:~/Desktop/SSI-GIT/TPs/TP1$ python3 msg_server.py -d
>> 2024-03-30 01:06:51.882976 Starting server ...
>> 2024-03-30 01:06:51.883062 Server listening on 127.0.0.1:8443
>> 2024-03-30 01:06:51.883074 Max connections: 5
>> 2024-03-30 01:06:56.122949 Accepted connection from ('127.0.0.1', 42472)
>> 2024-03-30 01:06:56.191194 Waiting for data from MSG_CLI2
>> 2024-03-30 01:07:18.074249 Accepted connection from ('127.0.0.1', 38268)
>> 2024-03-30 01:07:18.156549 Waiting for data from MSG_CLI1
>> 2024-03-30 01:07:18.156806 Received data from ('127.0.0.1', 38268):
GET_CERT
>> 2024-03-30 01:07:18.156971 Waiting for data from MSG_CLI1
>> 2024-03-30 01:07:18.159816 Received data from ('127.0.0.1', 38268):
SEND
>> 2024-03-30 01:07:18.159982 Waiting for data from MSG_CLI1
>> 2024-03-30 01:07:18.159934 [listen_to_client] Connection closed
>> 2024-03-30 01:07:34.623127 Received data from ('127.0.0.1', 42472):
ASK_QUEUE
>> 2024-03-30 01:07:34.623330 Waiting for data from MSG_CLI2
>> 2024-03-30 01:07:37.934215 Received data from ('127.0.0.1', 42472):
GET_MSG
>> 2024-03-30 01:07:37.934537 Waiting for data from MSG_CLI2
>>

```

```

rodrigo@pop-os: ~/Desktop/SSI-GIT/TPs/TP1
rodrigo@pop-os:~/Desktop/SSI-GIT/TPs/TP1$ python3 msg_client.py send MSG_CLI2 "LUSIADAS" < other/lusiadas3f.txt
rodrigo@pop-os:~/Desktop/SSI-GIT/TPs/TP1$

```

```

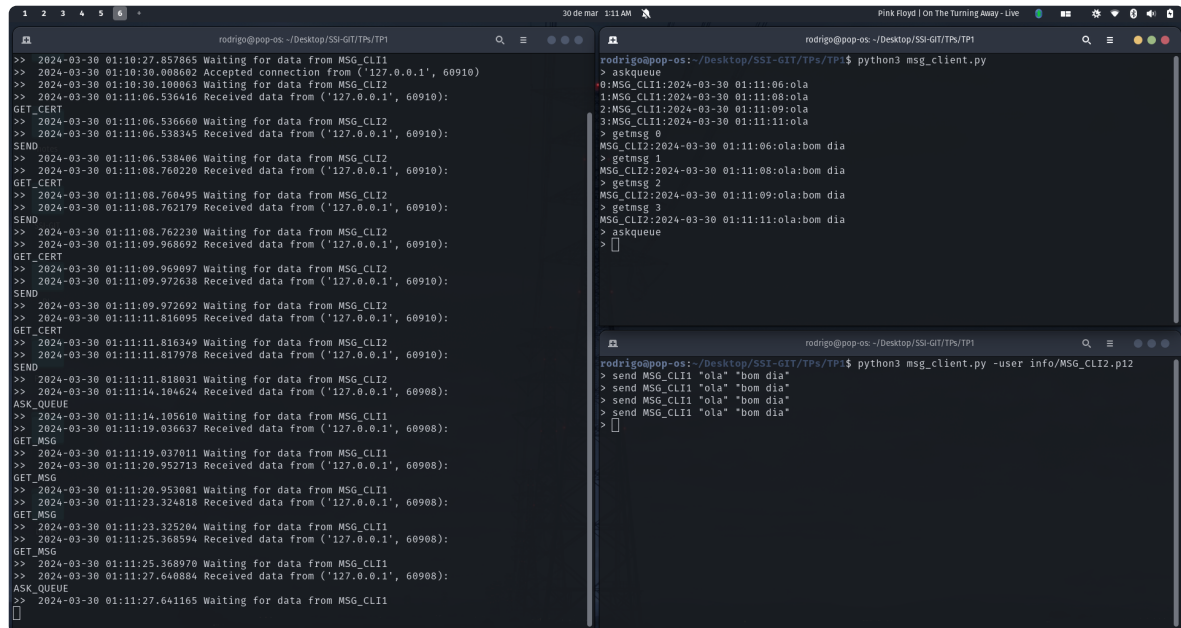
rodrigo@pop-os: ~/Desktop/SSI-GIT/TPs/TP1
Novo Reino, que tanto sublimaram;

2
E também as memórias gloriosas
Daqueles Reis, que foram dilatando
A Fé, o Império, e as terras viciosas
De África e de Ásia andaram devastando;
E aqueles, que por obras valerosas
Se vão da lei da morte libertando;
Cantando espalharei por toda parte,
Se a tanto me ajudar o engenho e arte.

3
Cessem do sábio Grego e do Troiano
As navegações grandes que fizeram;
Cale-se de Alexandro e de Trajano
A fama das vitórias que tiveram;
Que eu canto o peito ilustre Lusitano,
A quem Neptuno e Marte obedeceram;
Cesse tudo o que a Musa antiga canta,
Que outro valor mais alto se alevanta.
>>

```


- Exemplo de utilização (2)



```
rodrigo@pop-os: ~/Desktop/SSI-GIT/TPs/TP1
>>> 2024-03-30 01:10:27.857865 Waiting for data from MSG_CL11
>> 2024-03-30 01:10:30.008602 Accepted connection from ('127.0.0.1', 60910)
>> 2024-03-30 01:10:30.100063 Waiting for data from MSG_CL12
>> 2024-03-30 01:11:06.536416 Received data from ('127.0.0.1', 60910):
GET_CERT
>> 2024-03-30 01:11:06.536660 Waiting for data from MSG_CL12
>> 2024-03-30 01:11:06.538345 Received data from ('127.0.0.1', 60910):
SEND
>> 2024-03-30 01:11:06.538406 Waiting for data from MSG_CL12
>> 2024-03-30 01:11:08.760220 Received data from ('127.0.0.1', 60910):
GET_CERT
>> 2024-03-30 01:11:08.760495 Waiting for data from MSG_CL12
>> 2024-03-30 01:11:08.762179 Received data from ('127.0.0.1', 60910):
SEND
>> 2024-03-30 01:11:08.762230 Waiting for data from MSG_CL12
>> 2024-03-30 01:11:09.968692 Received data from ('127.0.0.1', 60910):
GET_CERT
>> 2024-03-30 01:11:09.969097 Waiting for data from MSG_CL12
>> 2024-03-30 01:11:09.972638 Received data from ('127.0.0.1', 60910):
SEND
>> 2024-03-30 01:11:09.972692 Waiting for data from MSG_CL12
>> 2024-03-30 01:11:11.816095 Received data from ('127.0.0.1', 60910):
GET_CERT
>> 2024-03-30 01:11:11.816340 Waiting for data from MSG_CL12
>> 2024-03-30 01:11:11.817970 Received data from ('127.0.0.1', 60910):
SEND
>> 2024-03-30 01:11:11.818031 Waiting for data from MSG_CL12
>> 2024-03-30 01:11:14.104624 Received data from ('127.0.0.1', 60908):
ASK_QUEUE
>> 2024-03-30 01:11:14.105610 Waiting for data from MSG_CL11
>> 2024-03-30 01:11:19.036637 Received data from ('127.0.0.1', 60908):
GET_MSG
>> 2024-03-30 01:11:19.037011 Waiting for data from MSG_CL11
>> 2024-03-30 01:11:20.952713 Received data from ('127.0.0.1', 60908):
GET_MSG
>> 2024-03-30 01:11:20.953081 Waiting for data from MSG_CL11
>> 2024-03-30 01:11:23.324818 Received data from ('127.0.0.1', 60908):
GET_MSG
>> 2024-03-30 01:11:23.325204 Waiting for data from MSG_CL11
>> 2024-03-30 01:11:25.368594 Received data from ('127.0.0.1', 60908):
GET_MSG
>> 2024-03-30 01:11:25.368970 Waiting for data from MSG_CL11
>> 2024-03-30 01:11:27.640884 Received data from ('127.0.0.1', 60908):
ASK_QUEUE
>> 2024-03-30 01:11:27.641165 Waiting for data from MSG_CL11
>>

rodrigo@pop-os: ~/Desktop/SSI-GIT/TPs/TP1$ python3 msg_client.py
> askqueue
0:MSG_CL11:2024-03-30 01:11:06:ola
1:MSG_CL11:2024-03-30 01:11:08:ola
2:MSG_CL11:2024-03-30 01:11:09:ola
3:MSG_CL11:2024-03-30 01:11:11:ola
> getmsg 0
MSG_CL12:2024-03-30 01:11:06:ola:bom dia
> getmsg 1
MSG_CL12:2024-03-30 01:11:08:ola:bom dia
> getmsg 2
MSG_CL12:2024-03-30 01:11:09:ola:bom dia
> getmsg 3
MSG_CL12:2024-03-30 01:11:11:ola:bom dia
> askqueue
>

rodrigo@pop-os: ~/Desktop/SSI-GIT/TPs/TP1$ python3 msg_client.py -user info/MSG_CL12.p12
> send MSG_CL11 "ola" "bom dia"
> send MSG_CL11 "ola" "bom dia"
> send MSG_CL11 "ola" "bom dia"
> send MSG_CL11 "ola" "bom dia"
>
>
```

Conclusões

Para concluir, achamos que este trabalho prático foi uma boa oportunidade para finalmente aplicar criptografia em protocolos de comunicação entre clientes e servidor, aproveitando também conhecimentos adquiridos noutras UCs como Comunicações por Computador e Sistemas Distribuídos.

Referências

- <https://cryptography.io/en/stable/fernet/>
- <https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/#encryption>
- https://en.wikipedia.org/wiki/Hybrid_cryptosystem
- <https://security.stackexchange.com/questions/27776/block-chain-modes-to-avoid/27780#27780>