



Criptografia e Segurança em Redes
ENGENHARIA DE TELECOMUNICAÇÕES E INFORMÁTICA
2023/2024

Trabalho Prático 2
Segurança na comunicação

David Alexandre Baptista Oliveira – a86732@alunos.uminho.pt

José Pedro Fernandes Peleja – a84436@alunos.uminho.pt

Miguel Fernandes Pereira – a94152@alunos.uminho.pt

Índice

1. Introdução	5
2. Comunicação	6
3. Modos de segurança	7
3.1. Integridade	7
3.2. Integridade e confidencialidade	9
3.3. Integridade, confidencialidade e autenticidade	13
4. Testes e resultados	17
4.1. Integridade	17
4.2. Integridade e Confidencialidade	22
4.3. Integridade, confidencialidade e autenticidade	28
Conclusão	32

Índice de figuras

Figura 1 – Diagrama de blocos para o método A.	7
Figura 2 - Função local de hashing.	7
Figura 3 - Concatenação de dados e chave correspondente.....	8
Figura 4 - Receção de dados.....	8
Figura 5 - Extração da mensagem e chave.	8
Figura 6 - Confirmação da integridade.	8
Figura 7 - Diagrama de blocos para o método B	9
Figura 8 - Computação de parâmetros.....	9
Figura 9 - Troca de chaves públicas.	10
Figura 10 - Derivação da chave partilhada.	10
Figura 11 - Vetor de inicialização.....	10
Figura 12 - Código executado pela thread para receber mensagens.....	11
Figura 13 - Código executado pela thread para enviar mensagens.	11
Figura 14 - Função de encriptação usando AES no modo CBC.	11
Figura 15 - Função de desencriptação usando AES no modo CBC.	12
Figura 16 - Diagrama de blocos para o método C.	13
Figura 17 – Geração de chaves sistema RSA.	13
Figura 18 - Função geradora do par de chaves.....	13
Figura 19 – Código executado pela thread para enviar mensagens.....	14
Figura 20 - Função de assinatura digital.	14
Figura 21 - Função de encriptação de mensagens.	14
Figura 22 - Código executado pela thread para receber mensagens.....	15
Figura 23 - Função de desencriptação de mensagens.....	15
Figura 24 – Função para verificar assinatura.....	16
Figura 25 - Consola do servidor após os testes para no modo A.....	17
Figura 26 - Consola do cliente após os testes no modo A.....	17
Figura 27 - Pacote com as opções enviado pelo servidor no modo A.	18
Figura 28 - Pacote com a opção escolhida no modo A.....	18
Figura 29 - Pacote com a terceira mensagem do cliente no modo A.....	19
Figura 30 - Pacote com a segunda mensagem do cliente no modo A.....	19
Figura 31 - Pacote com a primeira mensagem do cliente no modo A.....	19
Figura 32 - Pacote da primeira mensagem do servidor no modo A.	20
Figura 33 - Pacote com a segunda mensagem do servidor no modo A.	20
Figura 34 - Follow TCP Stream do servidor no modo A.	20
Figura 35 - Follow TCP Stream do cliente no modo A.	21
Figura 36- Terminal depois desta alteração.	21
Figura 37- Alteração da mensagem recebida.	21
Figura 38 - Consola do servidor após os testes no modo B.....	22
Figura 39 - Consola do cliente após os testes no modo B.	22
Figura 40 - Pacote com a opção escolhida no modo B.....	22
Figura 41 - Envio da chave pública do cliente no modo B.....	23
Figura 42 - Envio da chave pública do servidor no modo B.....	23
Figura 43 - Envio do iv por parte do cliente no modo B.	24
Figura 44 – Pacote com a primeira mensagem do cliente no modo B.	24
Figura 45 - Pacote com a segunda mensagem do cliente no modo B.	25
Figura 46 - Pacote com a terceira mensagem do cliente no modo B.....	25
Figura 47 - Pacote com a primeira mensagem do servidor no modo B.	26
Figura 48 - Pacote com a segunda mensagem do servidor no modo B.....	26
Figura 49 - Follow TCP Stream do servidor no modo B.	26
Figura 50 - Follow TCP Stream do servidor no modo B.	27
Figura 51- Alteração da key para valor de tentativa.	27

Figura 52- Resultado no terminal da alteração da key.....	27
Figura 53 - Consola do cliente após os testes no modo C.	28
Figura 54 - Consola do servidor após os testes para o modo C.	28
Figura 55 - Envio da chave pública do servidor no modo C.....	28
Figura 56 - Pacote com a primeira mensagem do cliente no modo C.....	29
Figura 57 - Pacote da primeira mensagem do servidor no modo C.	30
Figura 58 - Follow TCP Stream do servidor no modo C.	30
Figura 59 - Follow TCP Stream do cliente no modo C.	31
Figura 60- Resultado no terminal apos a forja da assinatura.....	31
Figura 61- Tentativa de alteração de assinatura.	31

1. Introdução

O exercício que nos foi proposto neste trabalho prático, incide na implementação de vários métodos de segurança e criptografia em redes debatidos nas aulas teóricas. Serve este relatório para demonstrar o algoritmo e linha de pensamento para a resolução do problema proposto.

O problema consiste na implementação de um chat, em Python, baseado no modelo cliente-servidor, um modelo amplamente usado na indústria de software nos tempos atuais. Neste chat, o utilizador (*client-side*) tem 3 opções de escolha para o modo de segurança a ser implementado na proteção das mensagens enviadas/recebidas. Cada modo de segurança garante a implementação de uma ou várias propriedades de segurança (integridade, confidencialidade e autenticidade). Após a escolha, todas as mensagens entre o cliente e o servidor implementarão esse modo de segurança.

Como referido anteriormente, o serviço desenvolvido deverá oferecer suporte a três modos de garantia de segurança distintos:

- Integridade (Modo a): Garantindo a integridade das mensagens trocadas entre os utilizadores, sem implementar um mecanismo de confidencialidade.
- Confidencialidade e Integridade (Modo b): Além da integridade, o serviço implementará um mecanismo para garantir a confidencialidade, suportado por uma cifra simétrica.
- Confidencialidade, Integridade e Autenticidade (Modo c): O modo mais seguro, no qual o serviço suportará mecanismos que garantem confidencialidade, integridade e autenticidade da origem da mensagem, utilizando uma cifra de chave pública.

Para cada modo, o grupo escolheu os mecanismos e algoritmos que achou mais indicados aos requisitos de segurança pretendidos.

2. Comunicação

A implementação do sistema adota uma arquitetura cliente-servidor, proporcionando uma estrutura robusta para a comunicação entre dispositivos. Nesse modelo, o servidor atua como o ponto central que gere e fornece serviços, enquanto os clientes se conectam a ele para utilizar esses serviços. Neste caso os serviços são um sistema de chat com vários níveis de segurança.

Para estabelecer a comunicação, o servidor está configurado para aceitar conexões via *socket* num endereço IP específico e na porta 5555. A escolha da porta 5555 foi apenas arbitrário sendo que não poderia ser menor que 1024 e não maior que 65535.

Inicialmente, a comunicação foi realizada no *localhost*, em que tanto o cliente como o servidor executavam na mesma máquina. Para testar de forma mais realista, o grupo passou a utilizar um computador em que o cliente executa na máquina virtual em modo *Bridged* e o servidor executa no *host*. Com isto o grupo pretendia simular um ambiente real de troca de mensagens entre computadores.

Com o objetivo de otimizar o envio de mensagens foram usadas *threads*. Tanto no lado cliente quanto no servidor as *threads* implementadas são utilizadas para lidar com as operações de recepção e envio, garantindo um desempenho eficiente e simultâneo dessas operações

3. Modos de segurança

Na presente secção do relatório estão presentes as decisões e os passos tomados para a implementação do projeto e desenvolvimento de código.

3.1. Integridade

Neste método de comunicação, apenas a integridade das comunicações efetuadas é garantida, ou seja, o recetor tem forma de determinar se a mensagem sofreu alterações durante a transmissão.

Esta garantia é assegurada usando o método de *hashing*. O *hashing* consiste em transformar uma entrada de dados numa *string* de letras e números sem conexão aparente, através do uso de uma função de *hash*. Através da seguinte imagem, é observável o modelo seguido pelo grupo na sua implementação.

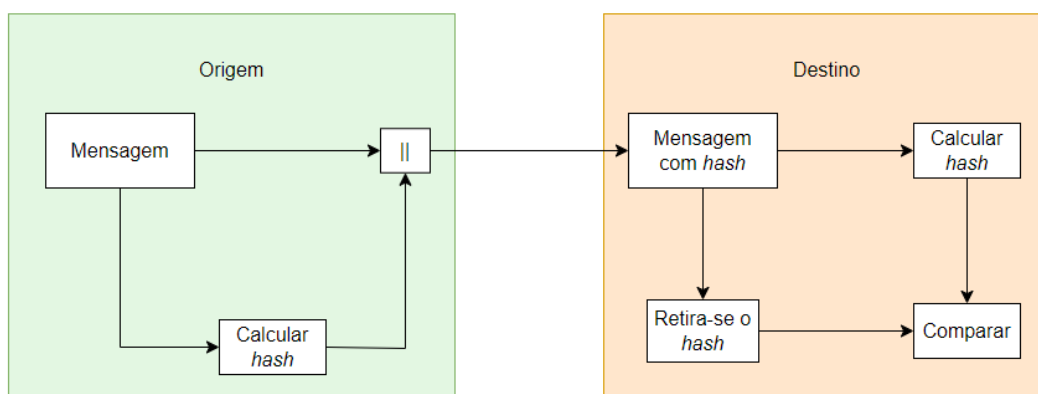


Figura 1 – Diagrama de blocos para o método A.

Na figura 1, está presente a criação da mensagem com o *hash* no final da origem. Esta é enviada através de *sockets* para o recetor. Do lado destino, este retira a *hash* recebida e calcula uma nova *hash* para a mensagem. Após isto, compara as duas *hash*, no caso de serem iguais significa que a mensagem não foi alterada no meio de comunicação e se forem diferentes significa que a mensagem foi alterada.

Para implementar o *hashing*, usamos a biblioteca *hashlib* em python e implementamos o *sha256* pois é considerado seguro para criptografia e por ser um padrão amplamente utilizado. Na implementação deste método de comunicação, a função responsável por fazer o *hash* está definida na figura abaixo:

```
def hash_data(data):
    sha256 = hashlib.sha256()
    sha256.update(data.encode("utf-8"))
    return sha256.hexdigest()
```

Figura 2 - Função local de *hashing*.

A função recebe um argumento *data*, que é a mensagem a ser *hashed*. Uma vez tendo o resultado da função, que devolve o argumento “*data_hash*”, em que está presente o hash da mensagem, a comunicação é feita enviando a mensagem original concatenada com a *hash* correspondente.

```
# Calculate the hash of the data
data_hash = hash_data(data)
# Concatenate the data and its hash
combined_data = data + data_hash
# Send the combined data to the server
client_socket.send(combined_data.encode())
```

Figura 3 - Concatenação de dados e chave correspondente.

Do lado do recetor, a mensagem é recebida no *socket* em formato *bytes* e é convertida para string através do método “*decode()*”.

```
combined_data = client_socket.recv(1024).decode("utf-8")
```

Figura 4 - Receção de dados.

Posteriormente, este bloco de dados necessita de ser processado para separar os dados da mensagem em si e a *hash*. Do bloco inteiro de dados, os últimos 64 caracteres correspondem á *hash* e os restantes são a mensagem, como mostra a figura abaixo:

```
# Extract the data and its hash based on the known hash size (64 characters)
data = combined_data[:-64]
received_hash = combined_data[-64:]
```

Figura 5 - Extração da mensagem e chave.

Uma vez tendo a mensagem, o lado do cliente executa um *hashing* à mensagem recebida e compara com a *hash* recebida, se forem iguais a mensagem é original, caso contrário a mensagem foi comprometida. A figura abaixo demonstra este processo:

```
# Calculate the hash of the received data
calculated_hash = hash_data(data)
if received_hash == calculated_hash:
    print(f"Received data: {data}")
    print("Integrity check: Passed")
else:
    print(f"Received data: {data}")
    print("Integrity check: Failed")
```

Figura 6 - Confirmação da integridade.

Através da implementação de todos os passos descritos acima, deve ser possível garantir e verificar se as mensagens enviadas na comunicação são originais ou não, ou seja, conseguimos garantir a integridade da mensagem.

3.2. Integridade e confidencialidade

Neste método de comunicação, a integridade e confidencialidade das comunicações efetuadas é garantida. Para além do método de segurança implementado atrás, é adicionada agora uma funcionalidade para garantir a confidencialidade.

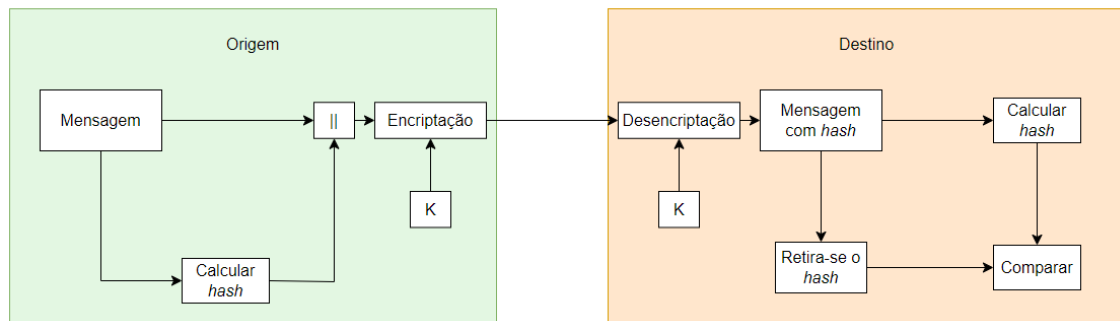


Figura 7 - Diagrama de blocos para o método B

Na figura 7, está um diagrama de blocos que reflete a implementação que o grupo fez. A diferença entre este modo e o modo anterior reside na encriptação, uma vez que a concatenação da mensagem e do *hash* é encriptada antes de ser enviada. Abaixo estão os detalhes da implementação.

A nossa implementação baseia-se no método de troca de chaves Diffie-Hellman, em que ambas as partes concordam previamente um número primo e um número gerador. De seguida cada uma das partes computa a sua chave privada aleatoriamente, e a partir dessa chave privada, cada uma das partes calcula uma chave pública, que é enviada para a parte oposta. Por fim, cada uma das partes usa a chave pública recebida e a própria chave privada para calcular a chave partilhada. Se tudo correu corretamente, ambas as chaves obtidas são iguais. A imagem seguinte apresenta mais detalhes sobre o cálculo das chaves:

```
#Computa os parametros Diffie-Hellman baseado num
#numero primo e num numero gerador
pn = dh.DHParameterNumbers(P, G)
#Computa os parametros baseado nos parametros Diffie-Hellman
parameters = pn.parameters()
#Computa uma chave privada baseada nos parametros
private_key = parameters.generate_private_key()
#Computa uma chave publica com base na chave privada
client_public_key = private_key.public_key()
# Passa a chave publica de DHPrivateKey para bytes
public = client_public_key.public_bytes(Encoding.PEM, PublicFormat.SubjectPublicKeyInfo)
```

Figura 8 - Computação de parâmetros.

Em seguida, segue-se a troca de chaves públicas entre as partes (de notar que a chave precisa de ser convertida de *bytes* para *string*, pela função “load_pem_public_key”):

```
client_public_key_bytes = conn.recv(1024)
client_public_key = load_pem_public_key(client_public_key_bytes, backend=default_backend())
conn.send(public)
```

Figura 9 - Troca de chaves públicas.

Uma vez possuindo a chave pública da parte oposta, podemos então derivar a chave partilhada usando a própria chave privada:

```
# Deriva a chave partilhada
shared_key = private_key.exchange(client_public_key)
# cria uma instancia HKDF para
# "refinar" a chave partilhada
derived_key = HKDF(
    algorithm=hashes.SHA256(),
    length=32,
    salt=None,
    info=b'handshake data'
)
key = derived_key.derive(shared_key)
```

Figura 10 - Derivação da chave partilhada.

Aqui termina o algoritmo de troca de chaves Diffie-Hellman, que será usado como base para garantir a confidencialidade das comunicações.

O próximo passo consiste no uso de cifras por blocos, nomeadamente AES (*Advanced Encryption Standard*) no modo CBC (*Cipher Block Chaining*). A escolha deste modo deve-se ao facto de prevenir a deteção de padrões, uma vez que cada bloco de texto encriptado depende dos blocos anteriores, fazendo com que a mesma *string* de texto plano resulte em blocos diferentes de texto encriptado. Para isso, necessitamos de um vetor de inicialização “iv”. Para garantir que este vetor chega o destino tal como foi enviado é feito um *hashing* para garantir a sua integridade e, de seguida envia-se para o servidor o vetor em si concatenado com o seu *hash*:

```
iv = "0123456789abcdef"
iv_b = iv.encode("utf-8")
iv_hash = hash_data(iv)
combined_iv = iv + iv_hash
client_socket.send(combined_iv.encode("utf-8"))
```

Figura 11 - Vetor de inicialização.

Feito isto, o servidor inicia duas threads para gerir as comunicações, uma thread para receber mensagens e outra para enviar. Estas funções recebem como argumentos o *socket* do cliente, a chave partilhada (que foi derivada anteriormente) e o vetor de inicialização. A implementação de cada uma das funções é apresentada nas figuras abaixo:

```
def handle_receive_data_b(conn, key, iv_b):
    while True:
        ciphertext = conn.recv(1024)
        plaintext, received_hash = decrypt_aes(key, ciphertext, iv_b)
        # Verificar a integridade dos dados recebidos
        expected_hash = hash_data(plaintext)

        if expected_hash == received_hash:
            print(f"Received data: {plaintext}")
            print("Integrity check: Passed")
        else:
            print(f"Received data: {plaintext}")
            print("Integrity check: Failed")
```

Figura 12 - Código executado pela *thread* para receber mensagens.

```
def handle_send_data_b(conn, key, iv_b):
    while True:
        data = input("Enter data: ")
        if data == 'exit':
            conn.close()
        # Calcula o hash da mensagem inserida
        data_hash = hash_data(data)
        # Encripta a mensagem inserida
        ciphertext = encrypt_aes(key, data, data_hash, iv_b)
        conn.send(ciphertext)
```

Figura 13 - Código executado pela *thread* para enviar mensagens.

Na figura 12, temos a função responsável por receber mensagens. Nesta função a mensagem é descriptada pela função “*decrypt_aes*” que retorna o texto bruto e o *hash* correspondente. De seguida é verificado se a integridade da mensagem foi comprometida comparando o *hash* recebido e o *hash* calculado.

Na figura 13, temos a função responsável por enviar mensagens. Nesta função a mensagem é encriptada pela função “*encrypt_aes*” que retorna a mensagem encriptada. De seguida a mensagem é enviada para a parte oposta.

No corpo de ambas as funções são usadas funções que são responsáveis por encriptar e descriptar as mensagens usando o protocolo AES. Iremos explorar agora essas funções.

```
def encrypt_aes(key, plaintext, data_hash, iv):
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    encryptor = cipher.encryptor()
    # Combina texto bruto e hash
    plaintext_with_hash = plaintext + data_hash
    # Aplicar o preenchimento PKCS7
    padder = padding.PKCS7(128).padder()
    padded_data = padder.update(plaintext_with_hash.encode("utf-8")) + padder.finalize()
    ciphertext = encryptor.update(padded_data) + encryptor.finalize()
    return ciphertext
```

Figura 14 - Função de encriptação usando AES no modo CBC.

Esta função “encrypt_aes” recebe como argumentos a chave partilhada previamente derivada, a mensagem de texto, o *hash* correspondente á mensagem de texto, e o vetor de inicialização. Primeiramente é instaciado um objeto *Cipher* que é inicializado com a chave partilhada e o vetor de inicialização. Este objeto servirá para encriptar a mensagem. De seguida, é combinado o texto da mensagem bruto com o seu *hash* correspondente e é aplicado um *padding*, que se encarrega de preencher o resto do bloco a ser encriptado que não foi completado de texto. Por fim, o bloco é encriptado e é devolvido pela função.

```
def decrypt_aes(key, ciphertext, iv):
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    decryptor = cipher.decryptor()
    padded_data = decryptor.update(ciphertext) + decryptor.finalize()
    padder = padding.PKCS7(128).unpadder()
    plaintext_with_hash = padder.update(padded_data) + padder.finalize()
    # Remove os últimos 64 bytes que representam o hash
    plaintext = plaintext_with_hash[:-64]
    # Obtém o hash dos últimos 64 bytes
    received_hash = plaintext_with_hash[-64:]
    return plaintext.decode("utf-8"), received_hash.decode("utf-8")
```

Figura 15 - Função de descriptação usando AES no modo CBC.

A função acima encarrega-se de descriptar a mensagem recebida usando cifra AES no modo CBC. Para tal, a função necessita da chave e do vetor de inicialização. No corpo da função, é retirado o *padding* e acontece a descriptação. De seguida é retirado o texto e o *hash* correspondente, que são devolvidos no final da função.

Após a implementação de todas as etapas acima descritas, podemos assumir que a integridade de todas as mensagens bem como a confidencialidade das mesmas são garantidas.

3.3. Integridade, confidencialidade e autenticidade

Neste modo de comunicação ambos os modos anteriores são assegurados, e para além disso também é garantida a autenticidade.

Ao contrário do modo anterior, neste modo usamos o sistema de encriptação de chave publica RSA, uma vez que consegue fazer encriptação/desencriptação de dados, assinaturas, verificação das mesmas e troca de chaves, tudo isto usando o mesmo sistema. Para além disso, a assinatura digital também garante a integridade da mensagem.

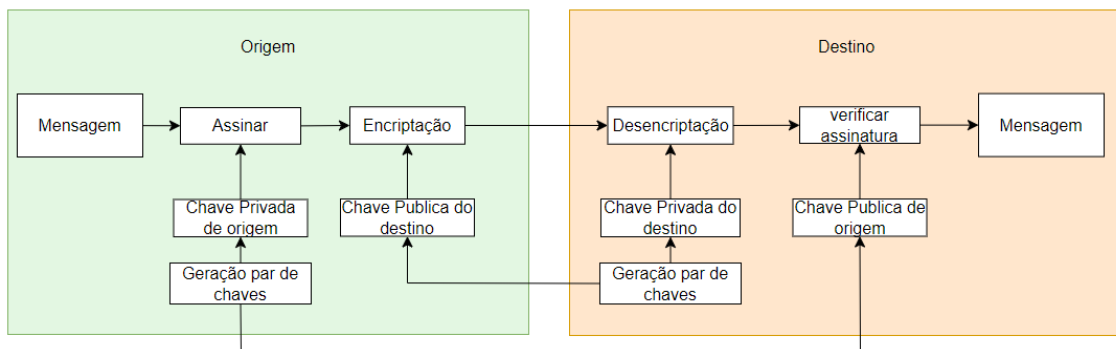


Figura 16 - Diagrama de blocos para o método C.

Na figura 16 está representado o esquema de implementação, que discutiremos ao longo desta secção.

```
# Gerar o par de chaves
server_private_key, server_public_key = generate_key_pair()

# Envio da chave publica para o cliente
conn.sendall(serialize_key(server_public_key))

# Receção da chave publica do cliente
client_public_key_data = conn.recv(2048)
client_public_key = serialization.load_pem_public_key(client_public_key_data, backend=default_backend())
```

Figura 17 – Geração de chaves sistema RSA.

```
def generate_key_pair():
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend()
    )
    public_key = private_key.public_key()
    return private_key, public_key
```

Figura 18 - Função geradora do par de chaves.

Tal como mostra a figura 17 começamos por gerar um par de chaves, constituído por uma chave privada e uma chave publica. A geração do par recorre á função “generate_key_pair” que está definida na figura 18. Ambos os lados da comunicação precisam de gerar o par de chaves. De imediato, ambos os lados enviam para o lado oposto a sua chave publica serializada.

Feito o passo anterior, o servidor inicia duas *threads*, uma para receber mensagens e outra para enviar. A baixo estão as definições das funções que são executadas pelas *threads*.

```
def handle_send_data_c(conn,private_key, public_key):
    while True:
        message = input("Enter data: ").encode("utf-8")
        signature = sign_message(message, private_key)
        encrypted_message = encrypt_message(message, public_key)
        # Send the signature and encrypted message to the server
        combined_data= signature + b'split' + encrypted_message
        conn.send(combined_data)
```

Figura 19 – Código executado pela *thread* para enviar mensagens.

```
def sign_message(message, private_key):
    from cryptography.hazmat.primitives.asymmetric import padding
    return private_key.sign(
        message,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
```

Figura 20 - Função de assinatura digital.

```
def encrypt_message(message, public_key):
    # Encrypt the message using OAEP padding with SHA-256
    from cryptography.hazmat.primitives.asymmetric import padding
    encrypted_message = public_key.encrypt(
        message,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return encrypted_message
```

Figura 21 - Função de encriptação de mensagens.

A função “handle_send_data_c” (figura 21), depois de pedir uma mensagem ao utilizador procede á assinatura dessa mensagem, usando a função “sign_message” que está definida na figura 21. Esta função, para além de assinar a mensagem, também aplica um *padding* e um *hash* á mensagem. Ao ser assinada estamos a garantir a autenticidade da mensagem assim como a integridade. De seguida, a mensagem é encriptada usando a chave pública do destinatário (oferecendo assim a confidencialidade) e é enviada para o destinatário uma concatenação da assinatura, da mensagem encriptada e uma *string* “split” entre elas. O motivo de incluir um “split” deve-se ao facto de o destinatário necessitar de saber onde existe a separação entre a assinatura e a mensagem encriptada, uma vez que a assinatura não tem um tamanho fixo.

```
def handle_receive_data_c(conn, private_key, public_key):
    while True:
        combined_data = conn.recv(4096)
        signature, encrypted_message = combined_data.rsplit(b'split', 1)
        # Decrypt the message
        decrypted_message = decrypt_message(encrypted_message, private_key)
        # Verify the signature
        if verify_signature(decrypted_message, signature, public_key):
            print("Signature verified. Message:", decrypted_message.decode())
        else:
            print("Signature verification failed.")
```

Figura 22 - Código executado pela *thread* para receber mensagens.

```
def decrypt_message(encrypted_message, private_key):
    from cryptography.hazmat.primitives.asymmetric import padding
    # Decrypt the message using OAEP padding with SHA-256
    decrypted_message = private_key.decrypt(
        encrypted_message,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return decrypted_message
```

Figura 23 - Função de descriptação de mensagens.

```
def verify_signature(message, signature, public_key):
    from cryptography.hazmat.primitives.asymmetric import padding
    try:
        public_key.verify(
            signature,
            message,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
        return True
    except:
        return False
```

Figura 24 – Função para verificar assinatura.

A função “handle_received_data_c” (figura 22), recebe a string concatenada da assinatura e da mensagem e divide-a, utilizando a palavra “split”, em *bytes*, como ponto de referência. De seguida, para poder verificar a assinatura, é necessário descriptar a mensagem usando a própria chave privada. Se o resultado da verificação for positivo, a mensagem é autêntica. Uma vez que a comunicação é bidirecional, ambas as partes implementam estas funções.

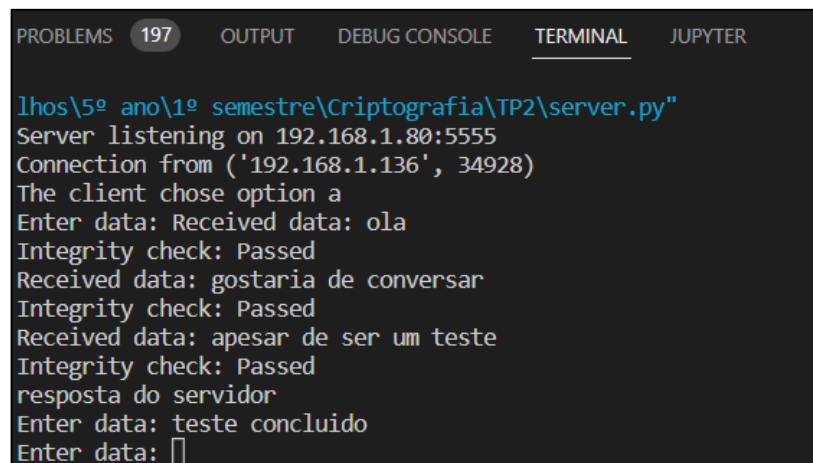
4. Testes e resultados

Nesta secção, são apresentados os resultados dos testes realizados. Para a realização dos mesmos foi utilizado a ferramenta Wireshark para a captura dos pacotes transmitidos na rede. Para visualizar os pacotes desejados foi aplicado um filtro (`ip.addr == 192.168.1.80`), sendo este endereço do servidor para a qual foram enviadas mensagens.

É de realçar que os testes foram realizados entre um computador, sendo este o servidor, e uma máquina virtual (utilizando a rede virtualizada em modo bridge), sendo esta o cliente.

4.1. Integridade

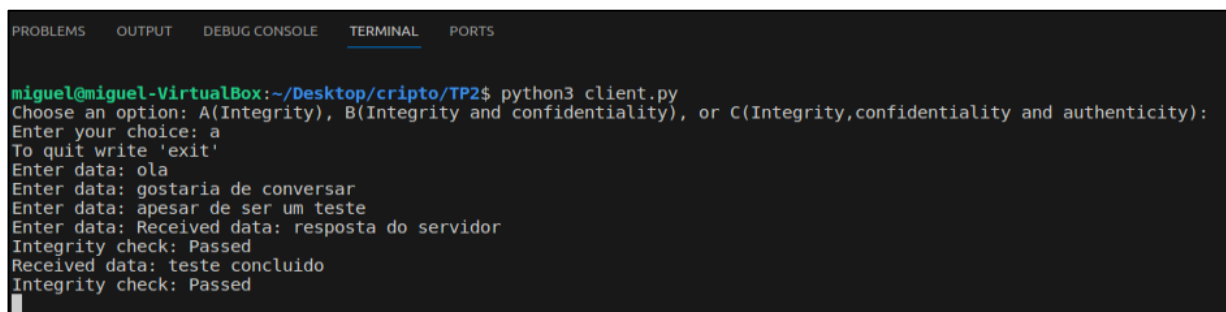
Primeiramente corremos o servidor para que fique à espera de uma ligação de algum cliente.



```
lhos\5º ano\1º semestre\Criptografia\TP2\server.py"
Server listening on 192.168.1.80:5555
Connection from ('192.168.1.136', 34928)
The client chose option a
Enter data: Received data: ola
Integrity check: Passed
Received data: gostaria de conversar
Integrity check: Passed
Received data: apesar de ser um teste
Integrity check: Passed
resposta do servidor
Enter data: teste concluido
Enter data: 
```

Figura 25 - Consola do servidor após os testes para no modo A

Iniciando um cliente, este conecta-se ao servidor, tendo o IP e a porta sido definidos *hardcoded*.



```
miguel@miguel-VirtualBox:~/Desktop/cripto/TP2$ python3 client.py
Choose an option: A(Integrity), B(Integrity and confidentiality), or C(Integrity, confidentiality and authenticity):
Enter your choice: a
To quit write 'exit'
Enter data: ola
Enter data: gostaria de conversar
Enter data: apesar de ser um teste
Enter data: Received data: resposta do servidor
Integrity check: Passed
Received data: teste concluido
Integrity check: Passed

```

Figura 26 - Consola do cliente após os testes no modo A

Depois de a ligação ter sido assegurada, o servidor envia uma mensagem, em plain text, para o cliente, com uma lista de opções de comunicação com os diferentes tipos de garantia de segurança suportadas.

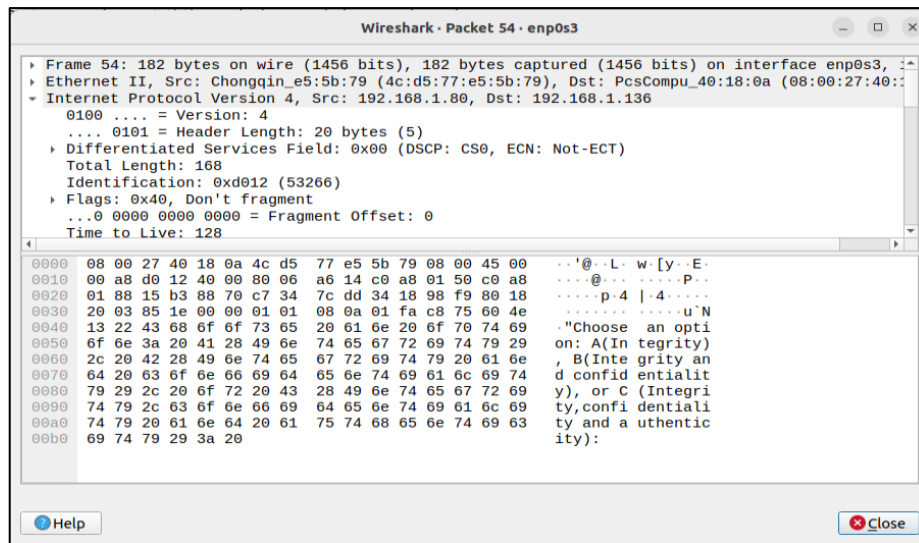


Figura 27 - Pacote com as opções enviado pelo servidor no modo A.

O cliente após receber as opções envia a letra “a”, escolhendo o modo com integridade assegurada.

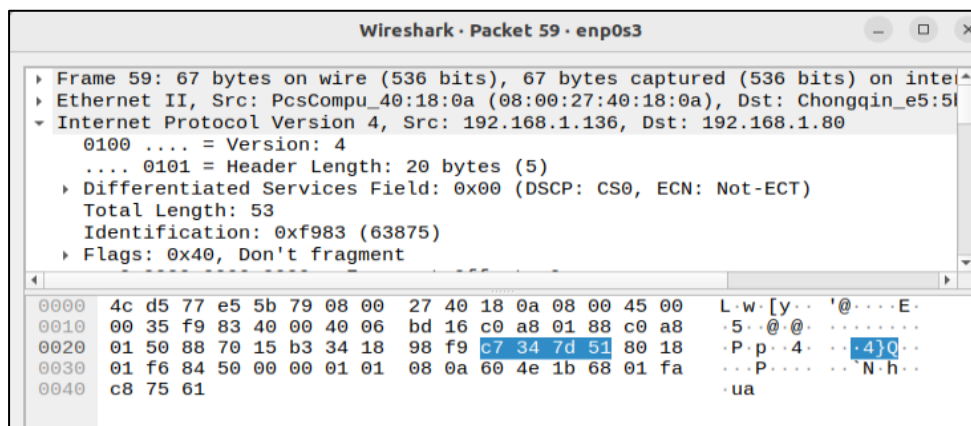


Figura 28 - Pacote com a opção escolhida no modo A.

Nas figuras seguintes, é possível visualizar as três mensagens enviadas pelo cliente ao servidor, em que se pode ver diretamente nos pacotes quais as mensagens enviadas juntamente com o *hash* calculado pelo cliente.

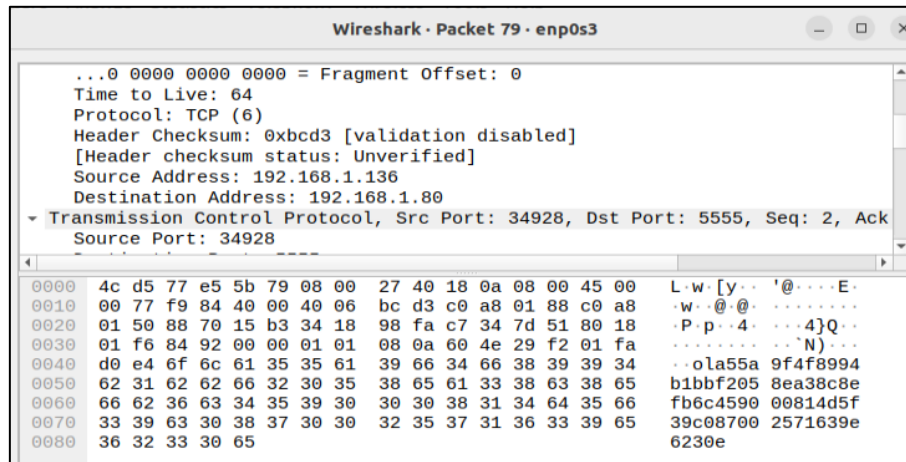


Figura 31 - Pacote com a primeira mensagem do cliente no modo A.

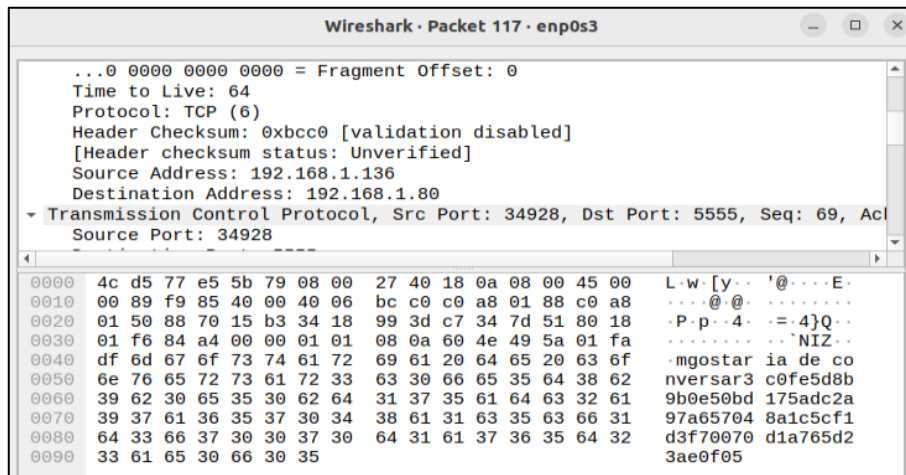


Figura 30 - Pacote com a segunda mensagem do cliente no modo A.

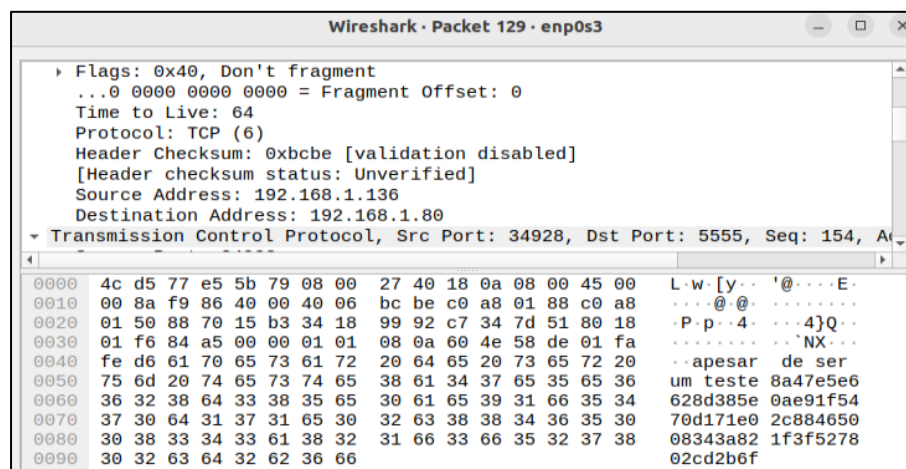


Figura 29 - Pacote com a terceira mensagem do cliente no modo A.

O servidor responde de seguida, com duas mensagens, sendo também possível ler todo o conteúdo dos pacotes.

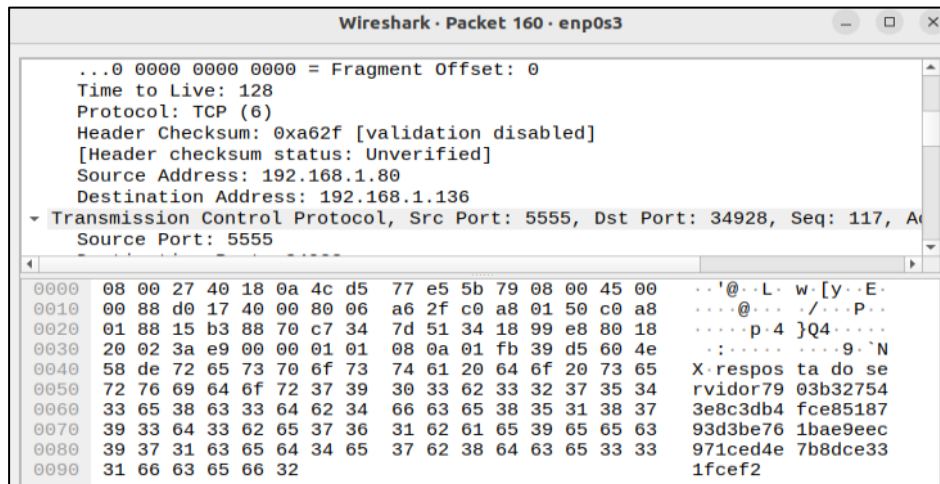


Figura 32 - Pacote da primeira mensagem do servidor no modo A.

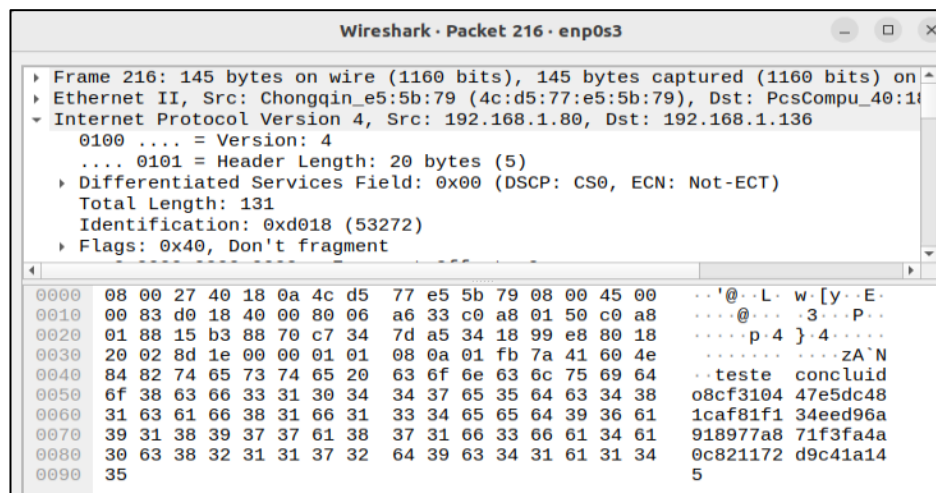


Figura 33 - Pacote com a segunda mensagem do servidor no modo A.

Usando a ferramenta "Follow TCP Stream" do Wireshark é possível visualizar todos os dados enviados pelo servidor e cliente, estando estes legíveis em *plain text*.

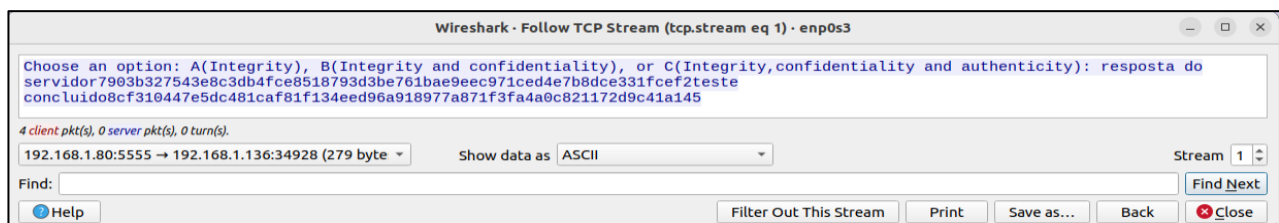


Figura 34 - Follow TCP Stream do servidor no modo A.

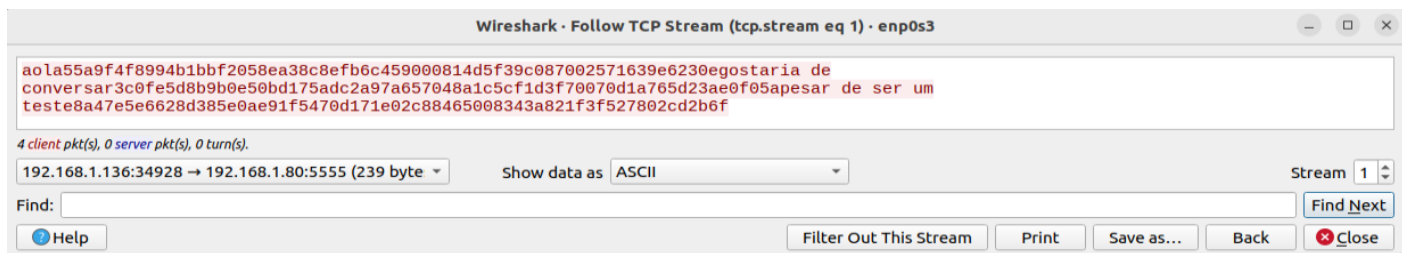


Figura 35 - Follow TCP Stream do cliente no modo A.

Como é observável na seguinte imagem a mensagem recebida é alterada provocando um *hash* calculado diferente. Desta forma, garantimos que a integridade está a ser assegurada na nossa implementação pois o resultado na imagem seguinte afirma que a verificação de integridade falhou, ou seja, a mesma foi violada.

```
# Receives the message from the other side
combined_data = client_socket.recv(1024).decode("utf-8")
# Extract the data and its hash based on the known hash size
data = combined_data[:-64] + "a"
received_hash = combined_data[-64:]
# Calculate the hash of the received data
calculated_hash = hash_data(data)
# Verifies the integrity of the message by comparing the received hash with the calculated hash
if received_hash == calculated_hash:
    print(f"Received data: {data}")
    print("Integrity check: Passed")
```

Figura 37- Alteração da mensagem recebida.

```
PS C:\Users\Pedro\Desktop\cripto\tp2> python3 Client.py
Choose an option: A(Integrity), B(Integrity and confidentiality), or C(Integrity, confidentiality)
Enter your choice: a
To quit write 'exit'
Enter data: forçando o erro
Enter data:

PS C:\Users\Pedro\Desktop\cripto\tp2> python3 Server.py
Server listening on 127.0.0.1:5555
Connection from ('127.0.0.1', 56632)
The client chose option a
Enter data: Received data: forçando o erro
Integrity check: Failed
```

Figura 36- Terminal depois desta alteração.

4.2. Integridade e Confidencialidade

Primeiramente corremos o servidor para que fique à espera de uma ligação de algum cliente.

É de notar que a porta estabelecida para a conexão, da figura 36, e a porta estabelecida, nos pacotes analisados, são diferentes pois após os testes a consola foi encerrada sem ter sido obtido um print, então foram realizados novamente os testes para a obtenção do print da consola do servidor, tendo a porta ficado com outro valor.

```
Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows
Connection from ('192.168.1.136', 33208)
The client chose option b
IV integrity check
Enter data: Received data: ola
Integrity check: Passed
Received data: este e o segundo teste
Integrity check: Passed
Received data: com confidencialidade e integridade
Integrity check: Passed
resposta do servidor
Enter data: teste concluido
Enter data: 
```

Figura 38 - Consola do servidor após os testes no modo B.

Iniciando um cliente, este conecta-se ao servidor, tendo o IP e a porta sido definidos *hardcoded*.

```
miguel@miguel-VirtualBox:~/Desktop/cripto/TP2$ python3 client.py
Choose an option: A(Integrity), B(Integrity and confidentiality), or C(Integrity, confidentiality and authenticity):
Enter your choice: b
To quit write 'exit'
Enter data: ola
Enter data: este e o segundo teste
Enter data: com confidencialidade e integridade
Enter data: Received data: resposta do servidor
Integrity check: Passed
Received data: teste concluido
Integrity check: Passed
```

Figura 39 - Consola do cliente após os testes no modo B.

Depois de a ligação ter sido assegurada, o servidor envia uma mensagem, em plain text, para o cliente, com uma lista de opções de comunicação com os diferentes tipos de garantia de segurança suportadas, semelhante ao pacote da figura 27.

O cliente após receber as opções envia a letra “b”, escolhendo o modo com confidencialidade e integridade assegurada.

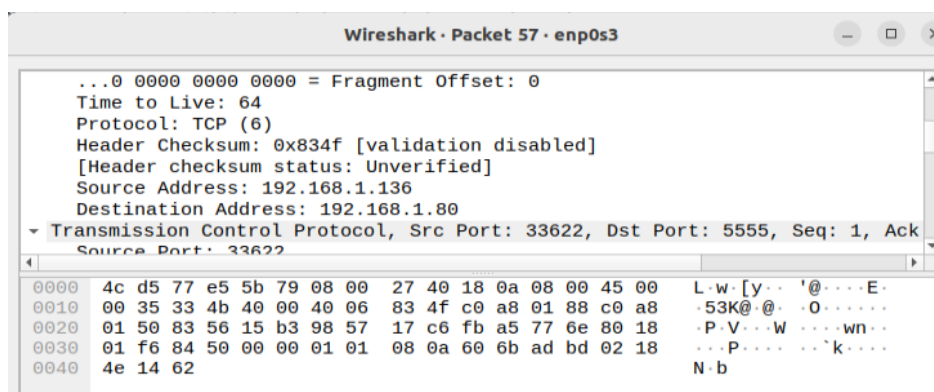


Figura 40 - Pacote com a opção escolhida no modo B.

Após o envio da opção, o cliente envia também a sua chave pública.

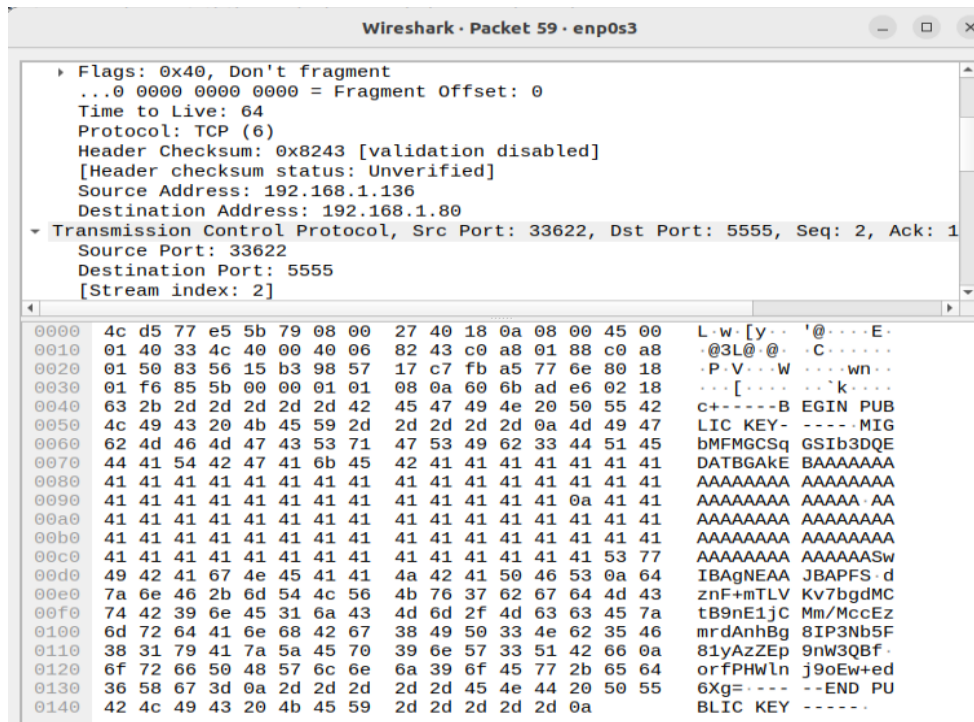


Figura 41 - Envio da chave pública do cliente no modo B.

O Servidor envia também a sua chave pública ao cliente.

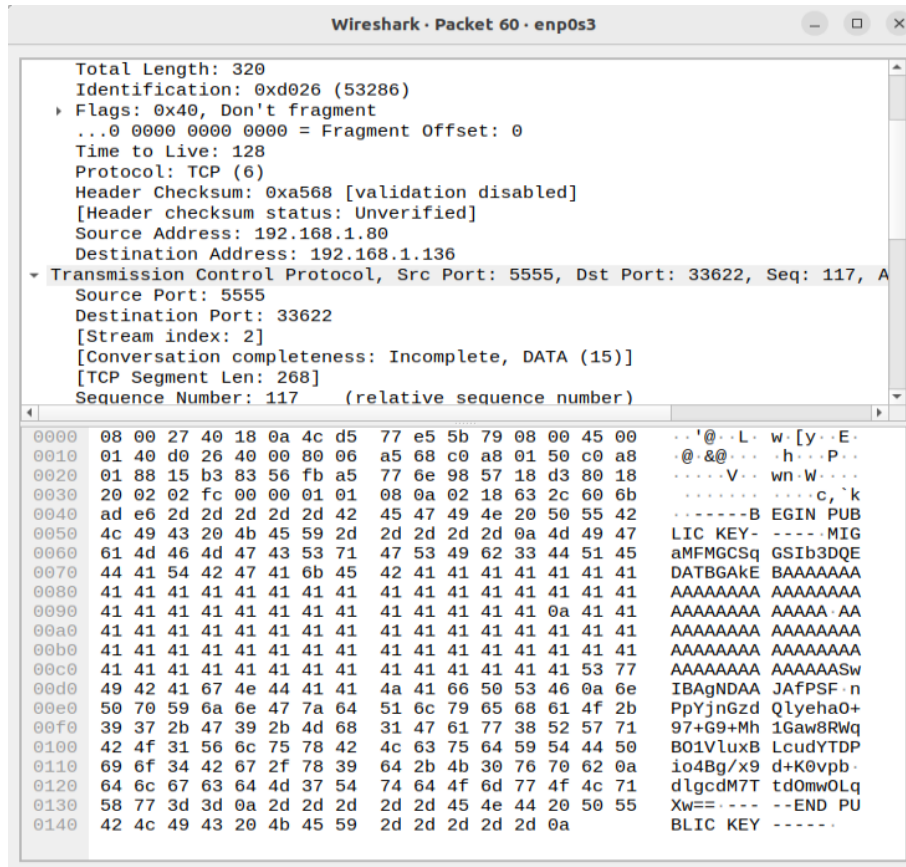


Figura 42 - Envio da chave pública do servidor no modo B.

O cliente após receber a chave pública do servidor envia o seu iv mais o hash calculado do iv.

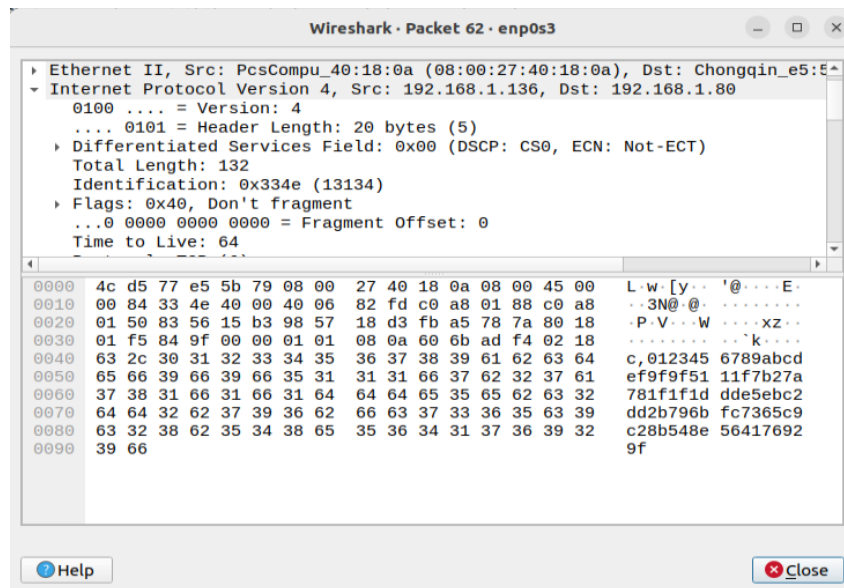


Figura 43 - Envio do iv por parte do cliente no modo B.

De seguida o cliente envia três mensagens para o servidor, como pode ser visualizado nas seguintes figuras, em que diferente do modo a, não é possível ler o conteúdo das mensagens.

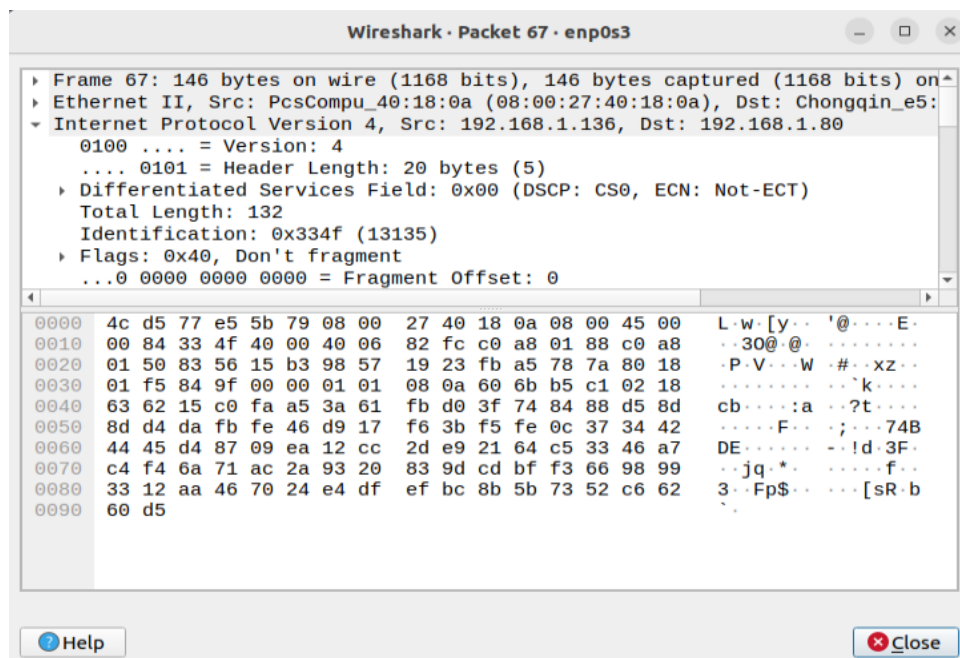


Figura 44 – Pacote com a primeira mensagem do cliente no modo B.

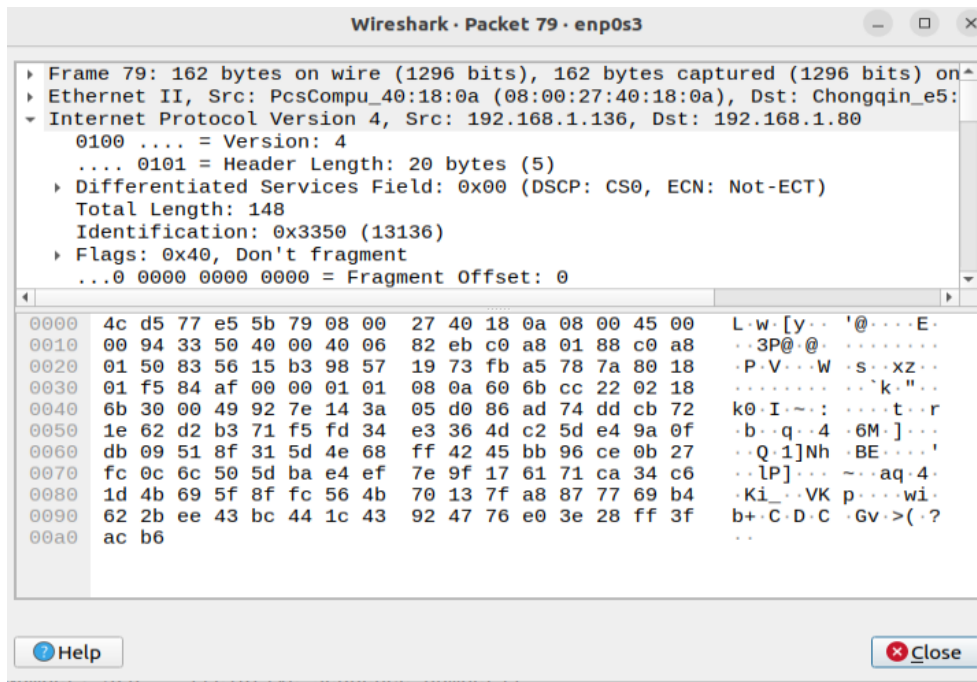


Figura 45 - Pacote com a segunda mensagem do cliente no modo B.

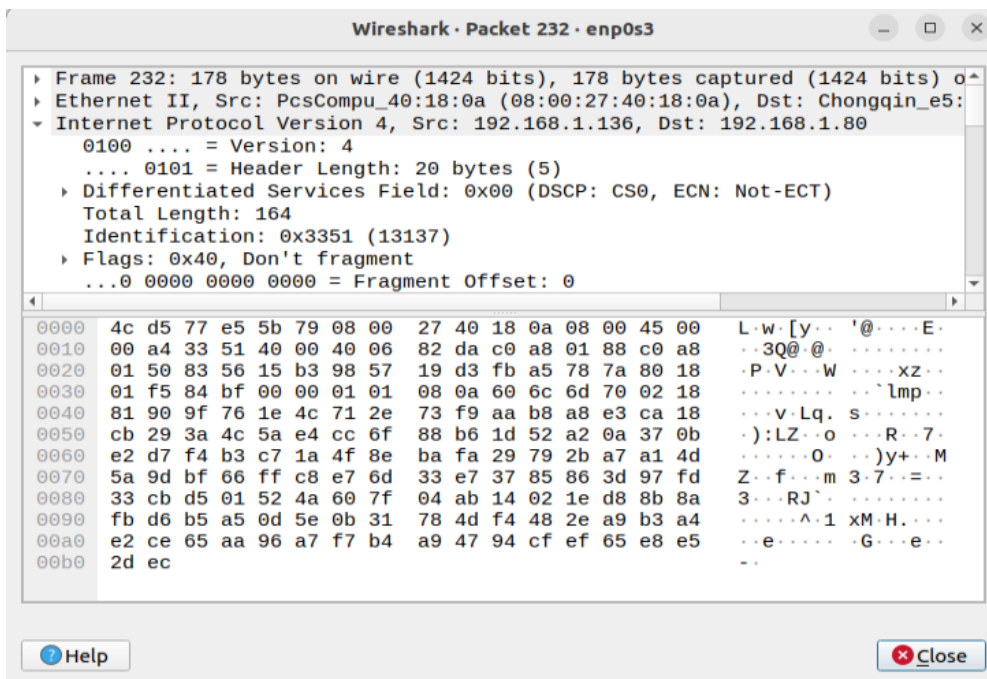


Figura 46 - Pacote com a terceira mensagem do cliente no modo B.

O servidor responde ao cliente com duas mensagens, não sendo também possível visualizar o conteúdo destas.

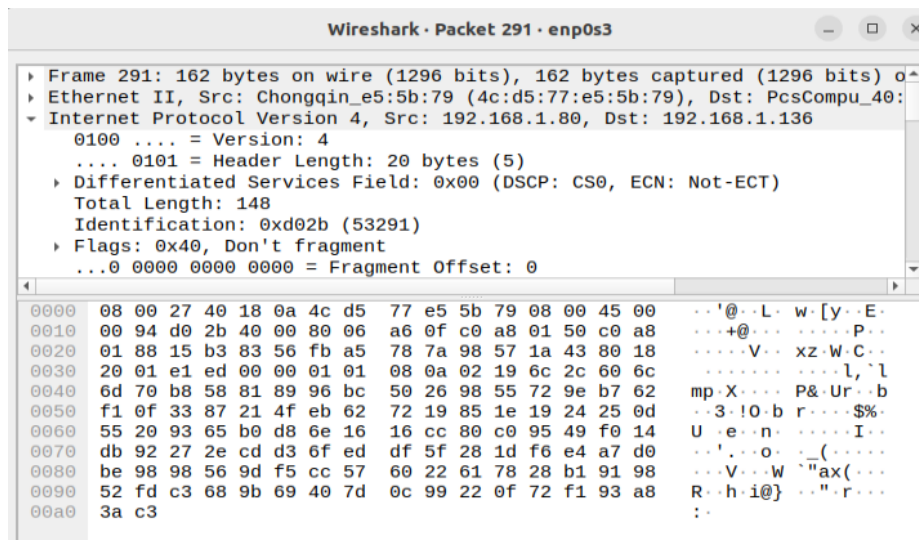


Figura 47 - Pacote com a primeira mensagem do servidor no modo B.

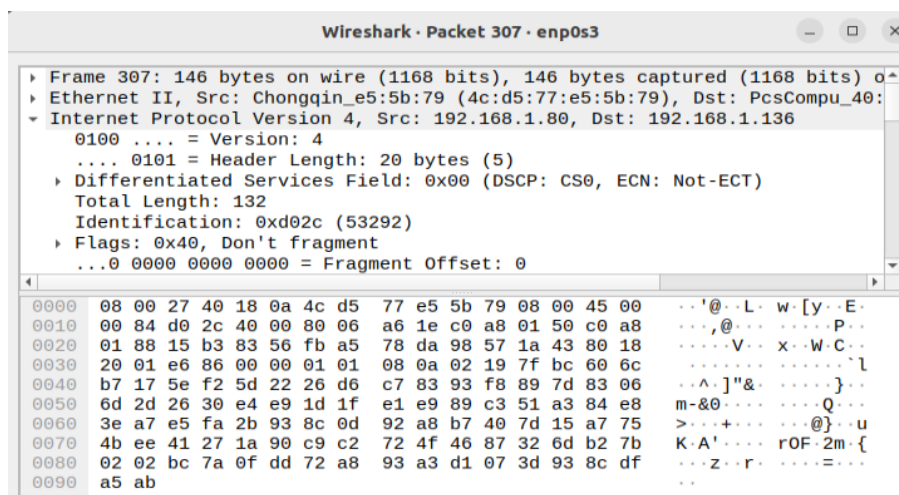


Figura 48 - Pacote com a segunda mensagem do servidor no modo B.

Usando a ferramenta "Follow TCP Stream" do Wireshark é possível visualizar todos os dados enviados pelo servidor e cliente, estando alguns legíveis em *plain text*, e ilegíveis os que foram enviados após a encriptação.

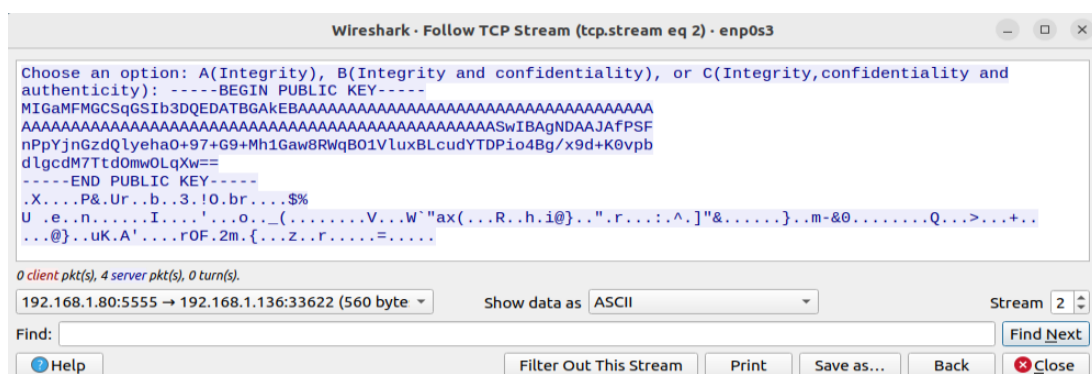
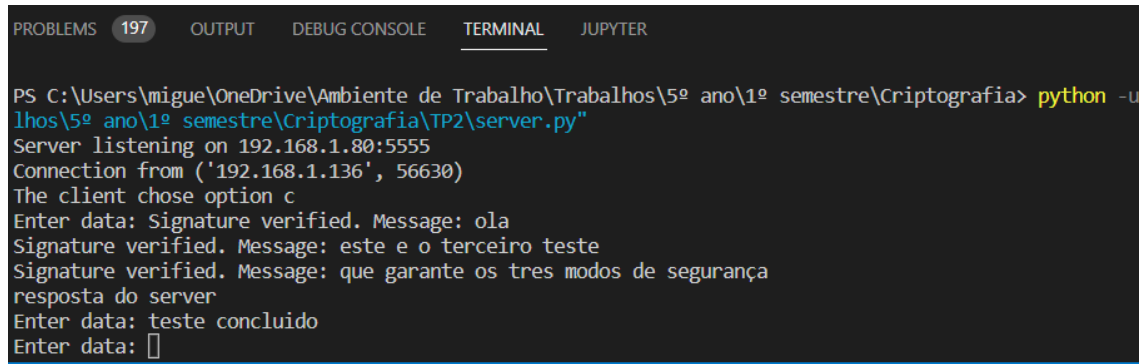


Figura 49 - Follow TCP Stream do servidor no modo B.

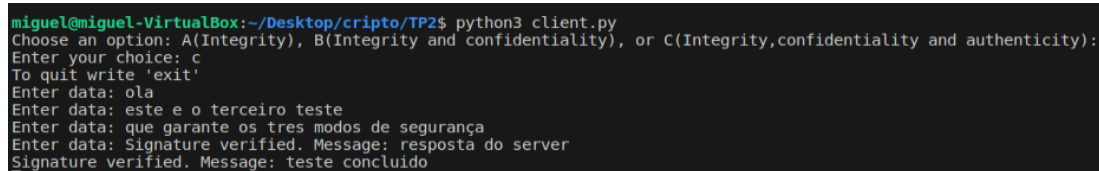
4.3. Integridade, confidencialidade e autenticidade

Semelhante aos testes do modo B, o servidor e cliente estabelecem uma ligação, trocam a lista das opções e a opção escolhida, e de seguida a troca das suas chaves públicas, sendo o servidor o primeiro a enviar.



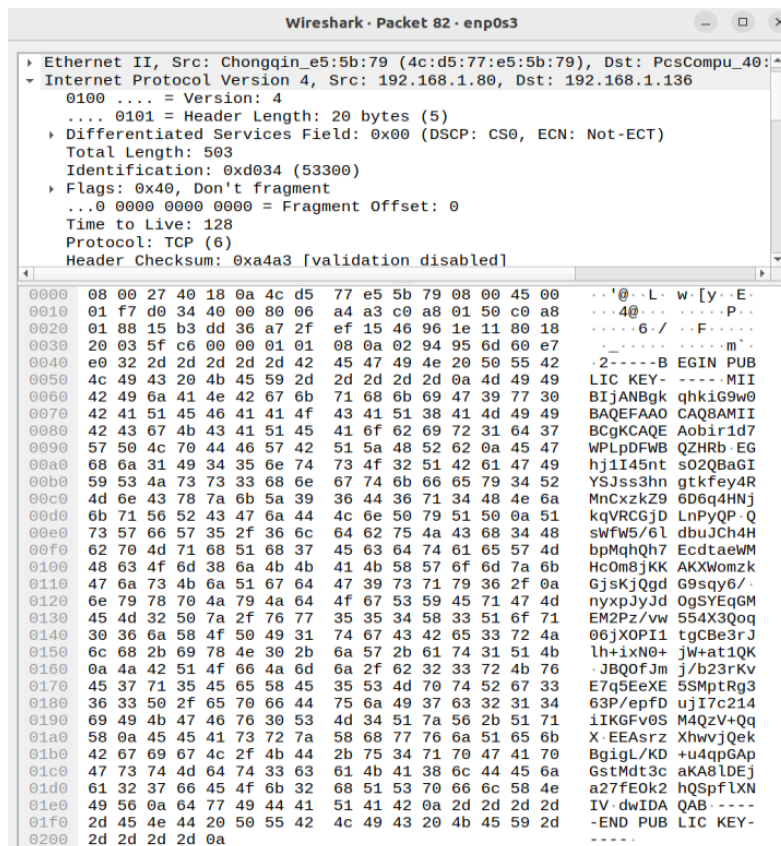
```
PROBLEMS 197 OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
PS C:\Users\miguel\OneDrive\Ambiente de Trabalho\Trabalhos\5º ano\1º semestre\Criptografia> python -u
lhos\5º ano\1º semestre\Criptografia\TP2\server.py
Server listening on 192.168.1.80:5555
Connection from ('192.168.1.136', 56630)
The client chose option c
Enter data: Signature verified. Message: ola
Signature verified. Message: este e o terceiro teste
Signature verified. Message: que garante os tres modos de segurança
resposta do server
Enter data: teste concluido
Enter data: █
```

Figura 54 - Consola do servidor após os testes para o modo C.



```
miguel@miguel-VirtualBox:~/Desktop/cripto/TP2$ python3 client.py
Choose an option: A(Integrity), B(Integrity and confidentiality), or C(Integrity, confidentiality and authenticity):
Enter your choice: c
To quit write 'exit'
Enter data: ola
Enter data: este e o terceiro teste
Enter data: que garante os tres modos de segurança
Enter data: Signature verified. Message: resposta do server
Signature verified. Message: teste concluido
█
```

Figura 53 - Consola do cliente após os testes no modo C.



Wireshark - Packet 82 - enp0s3

Ethernet II, Src: Chongqin_e5:5b:79 (4c:d5:77:e5:5b:79), Dst: PcsCompu_40:15:1d:01:00:00

Internet Protocol Version 4, Src: 192.168.1.80, Dst: 192.168.1.136

0100 = Version: 4

.... 0101 = Header Length: 20 bytes (5)

Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)

Total Length: 503

Identification: 0xd034 (53300)

Flags: 0x40, Don't fragment

...0 0000 0000 0000 = Fragment Offset: 0

Time to Live: 128

Protocol: TCP (6)

Header Checksum: 0xa4a3 [validation disabled]

0000 08 00 27 40 18 0a 4c d5 77 e5 5b 79 08 00 45 00 ...'@..L. w.[y..E..

0010 01 f7 d9 34 40 00 80 06 a4 a3 c0 a8 01 50 c0 a8 ...4@...P...

0020 01 88 15 b3 dd 36 a7 2f ef 15 46 96 1e 11 80 18 ...6./...F...

0030 20 03 5f c6 00 00 01 01 08 0a 02 94 95 6d 60 e7 ...m...

0040 e0 32 2d 2d 2d 2d 2d 42 45 47 49 4e 20 50 55 42 ...2-----B EGIN PUB

0050 4c 49 43 20 4b 45 59 2d 2d 2d 2d 2d 0a 4d 49 49 LIC KEY- ----MII

0060 42 49 6a 41 4e 42 67 6b 71 68 6b 69 47 39 77 30 BIjANBgk qhkiG9w0

0070 42 41 51 45 46 41 41 4f 43 41 51 38 41 4d 49 49 BAQEFAAO CAQ8AMII

0080 42 43 67 4b 43 41 51 45 41 6f 62 69 72 31 64 37 BCgKCAQE Aobir1d7

0090 57 50 4c 70 44 46 57 42 51 5a 48 52 62 0a 45 47 WPLpDFWB QZHRb-EG

00a0 68 6a 31 49 34 35 6e 74 73 4f 32 51 42 61 47 49 hj1I45nt s02QBaGI

00b0 59 53 4a 73 33 68 6e 67 74 6b 66 65 79 34 52 YSjss3hn gtkfey4R

00c0 4d 6e 43 78 7a 6b 5a 39 36 44 36 71 34 48 4e 6a MnCzxkZ9 6Dq4HNj

00d0 6b 71 56 52 43 47 6a 44 4c 6e 50 79 51 50 0a 51 kqVRCgJD LnPyQP-Q

00e0 73 57 66 57 35 2f 36 6c 64 62 75 4a 43 68 34 48 swfw5/6l dbuJCh4H

00f0 62 70 4d 71 68 51 68 37 45 63 64 74 61 65 57 4d bpMqhQh7 EcdtaeWM

0100 48 63 4f 6d 38 6a 4b 4b 41 4b 58 57 6f 6d 7a 6b Hc0m8jKK AKXWomzk

0110 47 6a 73 4b 6a 51 67 64 47 39 73 71 79 36 2f 0a GjsKjQgd G9sqy6/-

0120 6e 79 78 70 4a 79 4a 64 4f 67 53 59 45 71 47 4d nyxpJyJd OgSYEqGM

0130 45 4d 32 50 7a 2f 76 77 35 35 34 58 33 51 6f 71 EM2Pz/vw 554X3Qoq

0140 30 36 6a 58 4f 50 49 31 74 67 43 42 65 33 72 4a 06jXOPI1 tgCBe3rJ

0150 6c 68 2b 69 78 4e 30 2b 6a 57 2b 61 74 31 51 4b lh+ixN0+ jW+at1QK

0160 0a 4a 42 51 4f 66 4a 6d 6a 2f 62 32 33 72 4b 76 JBQOfJm j/b23rKv

0170 45 37 71 35 45 65 58 45 35 53 4d 70 74 52 67 33 E7q5EeXE 5SMptRg3

0180 36 33 50 2f 65 70 66 44 75 6a 49 37 63 32 31 34 63P/epfU ujI7c214

0190 69 49 4b 47 46 76 30 53 4d 34 51 7a 56 2b 51 71 iIKGFv0S M4qzV+Qq

01a0 58 0a 45 45 41 73 72 7a 58 68 77 76 6a 51 65 6b X-EEAsrz XhvwjQek

01b0 42 67 69 67 4c 2f 4b 44 2b 75 34 71 70 47 41 70 BgigL/KD +u4qpGAp

01c0 47 73 74 4d 64 74 33 63 61 4b 41 38 6c 44 45 6a GstMdt3c aKA8lDEj

01d0 61 32 37 66 45 4f 6b 32 68 51 53 70 66 6c 58 4e a27FE0k2 hQSpfLXN

01e0 49 56 0a 64 77 49 44 41 51 41 42 0a 2d 2d 2d 2d IV-dwIDA QAB- ----

01f0 2d 45 4e 44 20 50 55 42 4c 49 43 20 4b 45 59 2d -END PUB LIC KEY-

0200 2d 2d 2d 0a

Figura 55 - Envio da chave pública do servidor no modo C.

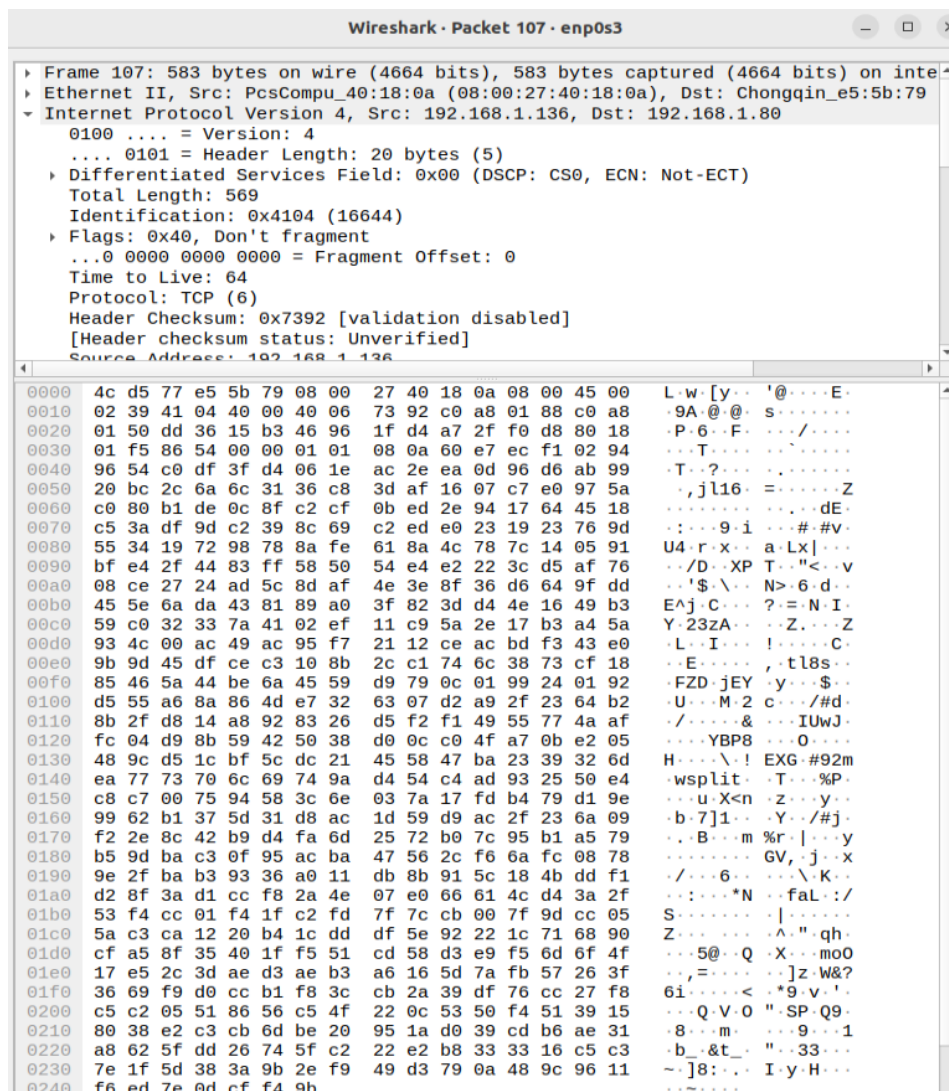


Figura 56 - Pacote com a primeira mensagem do cliente no modo C.

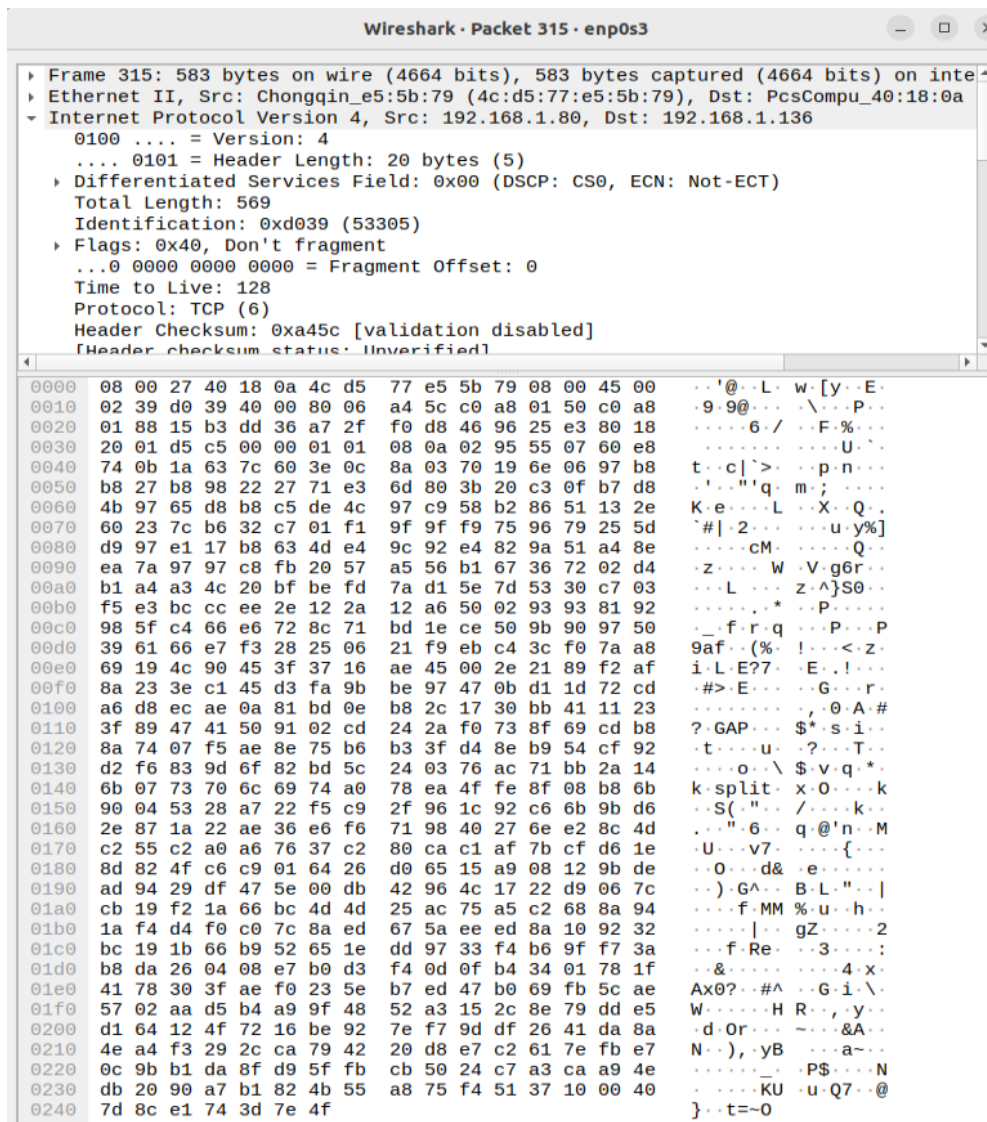


Figura 57 - Pacote da primeira mensagem do servidor no modo C.

Usando a ferramenta “Follow TCP Stream” do Wireshark é possível visualizar todos os dados enviados pelo servidor e cliente, estando alguns legíveis em texto plano, e ilegíveis os que foram enviados após a encriptação.



Figura 58 - Follow TCP Stream do servidor no modo C.

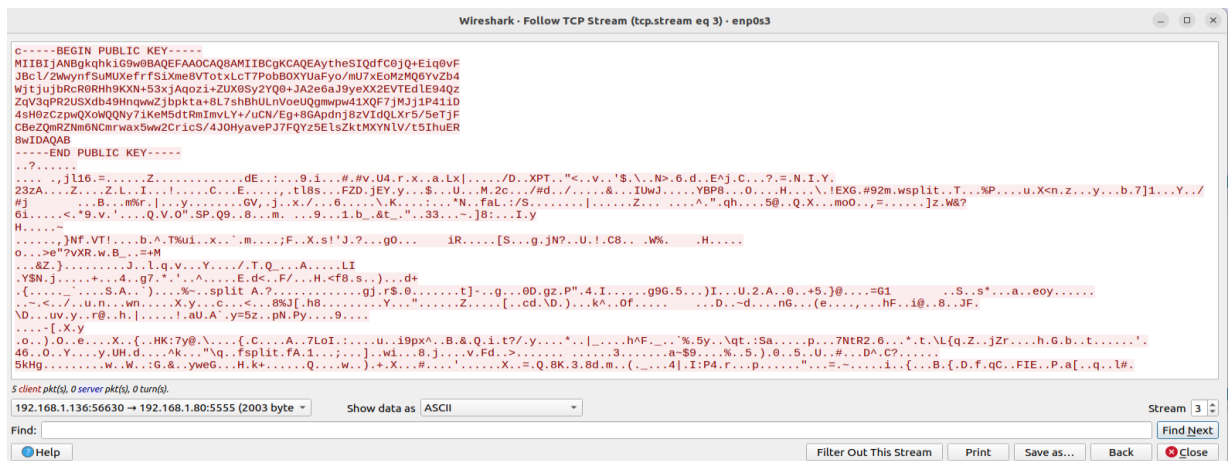


Figura 59 - Follow TCP Stream do cliente no modo C.

Após a tentativa de forjar uma assinatura o recetor da mensagem verifica que esta não vem do emissor que era esperado, demonstrando assim que a autenticidade é assegurada.

```
signature, encrypted_message = combined_data.rsplit(b'split', 1)
signature +=b"a"
# Decrypt the message
decrypted_message = decrypt_message(encrypted_message,server_private_key)

# Verifies the signature and returns the result of verification and the decrypted message
if (verify_signature(decrypted_message, signature, client_public_key)):
    print("Signature verified. Message:", decrypted_message.decode())
else:
    print("Signature verification failed.")
```

Figura 61- Tentativa de alteração de assinatura.

```
PS C:\Users\Pedro\Desktop\cripto\tp2> python3 Client.py
Choose an option: A(Integrity), B(Integrity and confidentiality), or C(Integrity,confidentiality and authenticity):
Enter your choice: c
To quit write 'exit'
Enter data: no caso de alguem tentar forjar a assinatura
Enter data:

PS C:\Users\Pedro\Desktop\cripto\tp2> python3 Server.py
Server listening on 127.0.0.1:5555
Connection from ('127.0.0.1', 56779)
The client chose option c
Enter data: Signature verification failed.
```

Figura 60- Resultado no terminal apos a forja da assinatura.

Conclusão

Com a conclusão deste Trabalho Prático, ficou evidente que a proteção de diversas propriedades de segurança requer abordagens e implementações distintas, desde a escolha de diferentes mecanismos até o design da solução como um todo.

Ao longo da execução do projeto, deparamo-nos com vários desafios, desde a utilização de bibliotecas complexas até a resolução de problemas inesperados. Uma das principais dificuldades surgiu ao empregar algoritmos como o RSA e o Diffie-Hellman, nos quais enfrentamos obstáculos na interpretação da documentação e na geração de chaves.

A utilização do RSA e do Diffie-Hellman revelou-se particularmente desafiadora, exigindo uma compreensão aprofundada dos detalhes técnicos e uma interpretação atenta da documentação. A complexidade associada à implementação desses algoritmos destacou a necessidade de um entendimento robusto das ferramentas criptográficas.

No final do processo, o grupo reconhece que adquiriu habilidades substanciais em criptografia. Este projeto não apenas proporcionou uma aprendizagem valiosa em termos técnicos, mas também estimulou o interesse em investigações futuras no campo da segurança da informação.

Em resumo, a realização deste trabalho não só consolidou o conhecimento em criptografia, mas também disponibilizou ao grupo ferramentas para enfrentar desafios práticos, contribuindo para o desenvolvimento de habilidades essenciais na construção de soluções seguras.