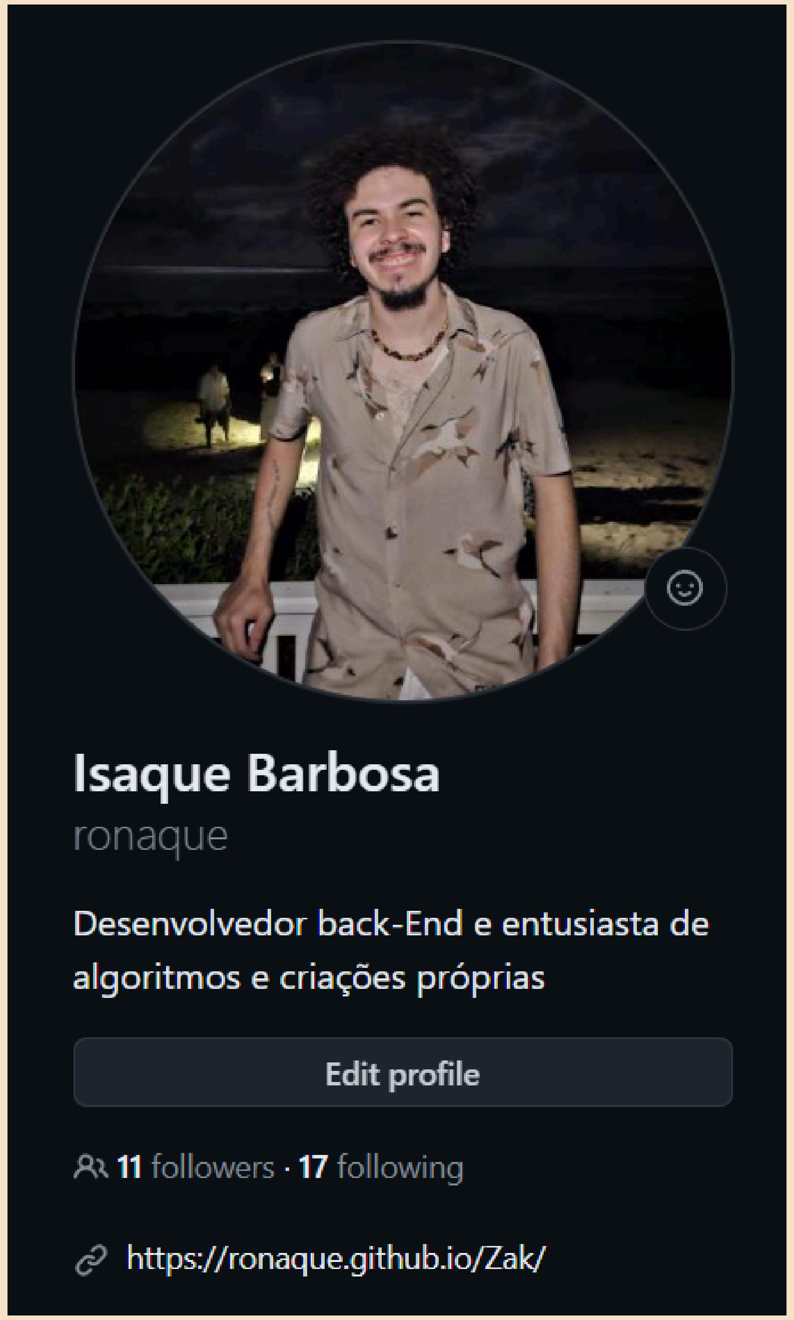
CRIPTOGRAFIA DE DADOS

Analisando a aplicação de algoritmos criptográficos em pacotes de redes

Isaque Barbosa

ISAQUE



Técnico em Jogos Digitais pelo IFRN

Formando em Tecnologia da Informação pela UFRN

Aspirante a cientista da computação com foco em algoritmos criptográficos pós-quânticos

Participante do grupo de comunicação quântica do IMD/UFRN

TÓPICOS

- Criptografia
- Fluxo Cliente-Servidor
- Envio de pacotes utilizando scapy
- Verificação de pacotes utilizando wireshark
- Criptografia pós-quântica
- Implementação da criptografia pós-quântica

CRIPTOGRAFIA

No nível mais básico, a criptografia é o processo de proteger informações ou dados usando modelos matemáticos para embaralhá-los de modo que apenas as partes que têm a chave para decifrar possam acessá-lo.

Text
RSA
DES
AES
ECC
9EgP+InT+
2LbpeCQ8
K0rmA==

TIPOS DE CRIPTOGRAFIA

SIMÉTRICA

Também conhecida como chave compartilhada ou algoritmo de chave privada, usa a mesma chave para criptografia e descriptografia. As criptografias de chave simétrica são consideradas mais baratas para produzir e não usam tanta força de computação para criptografar e descriptografar.

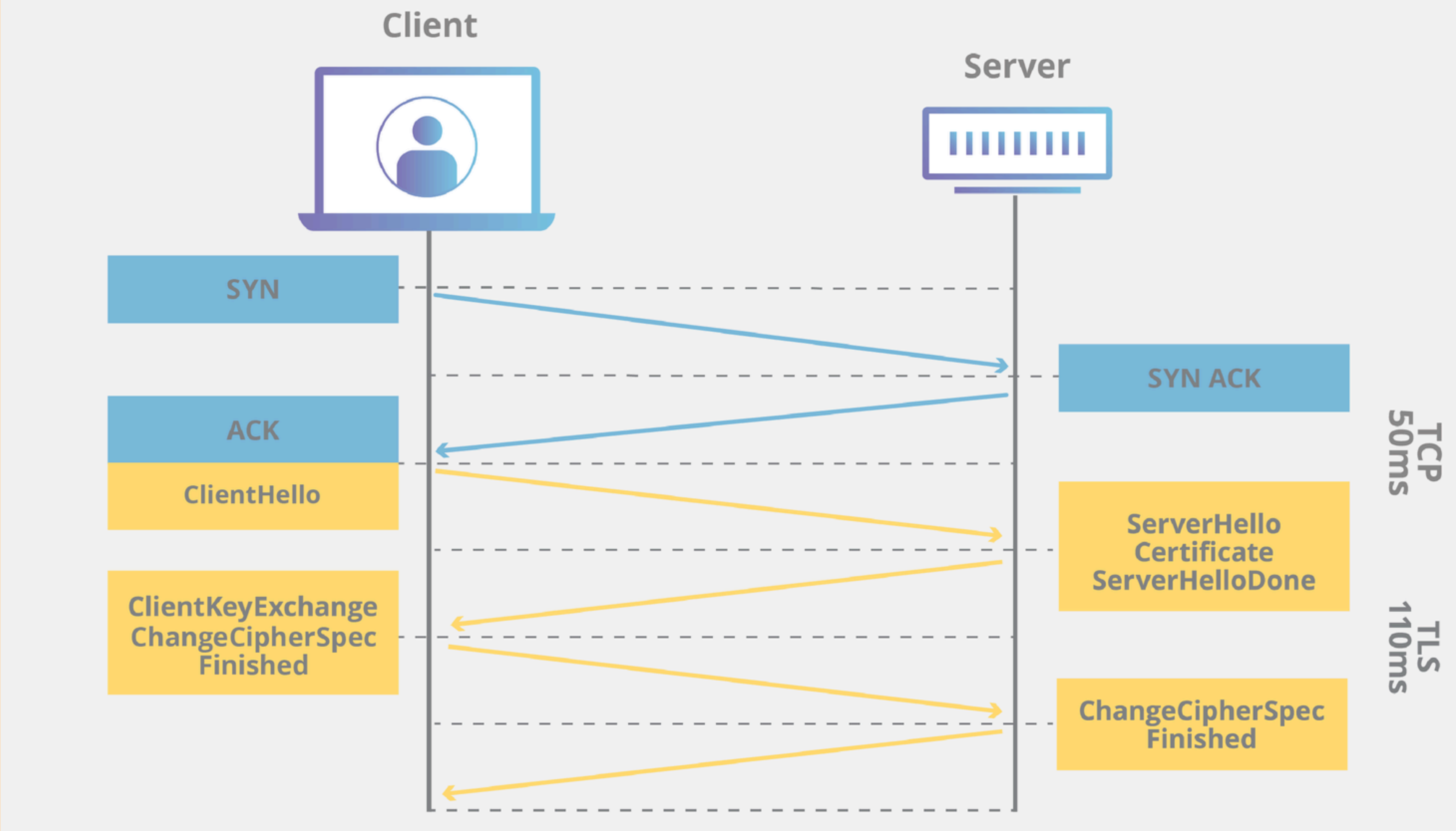
ASSIMÉTRICA

Também conhecida como criptografia de chave pública, usa duas chaves separadas para criptografar e descriptografar dados, sendo uma chave pública. Qualquer pessoa com a chave pública pode enviar uma mensagem criptografada, mas apenas os detentores da have privada poderão descriptografá-la.

3-WAY-HANDSHAKE

O 3-Way-Handshake é a forma utilizada para se fazer a conexão cliente-servidor, sendo atualmente separadas em duas etapas:

- TCP-Handshake: estabelece a conexão entre os pares
- TLS-Handshake: realiza a troca de chaves assimétricas, para cuminar na chave simétrica para criptografar os dados.

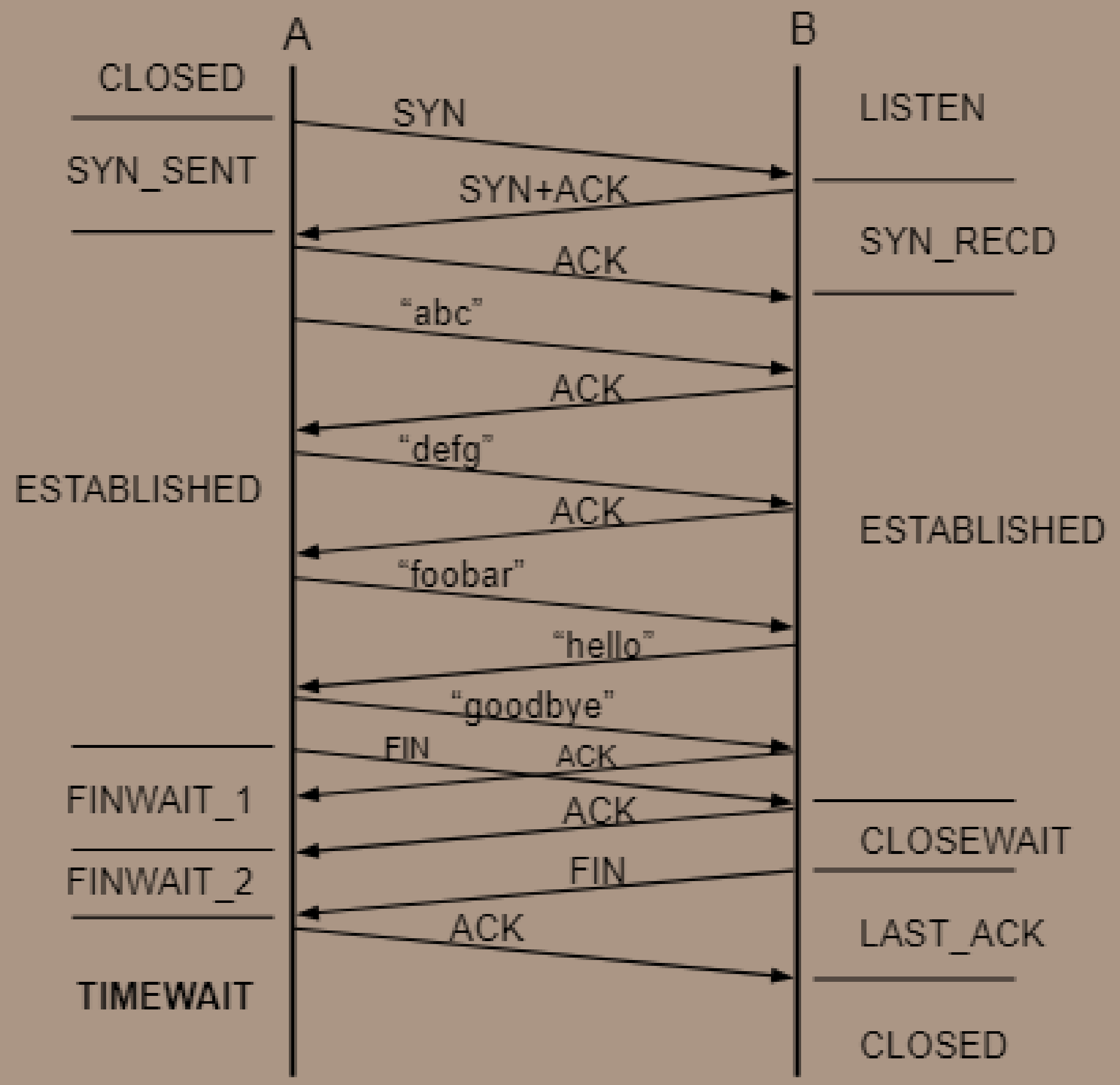


Fonte: cloudflare

TCP HANDSHAKE

Primeira etapa realizada para conexão entre cliente e servidor

Dado o estabelecimento da conexão, pode ser realizada a troca de dados NÃO CRIPTOGRAFADO

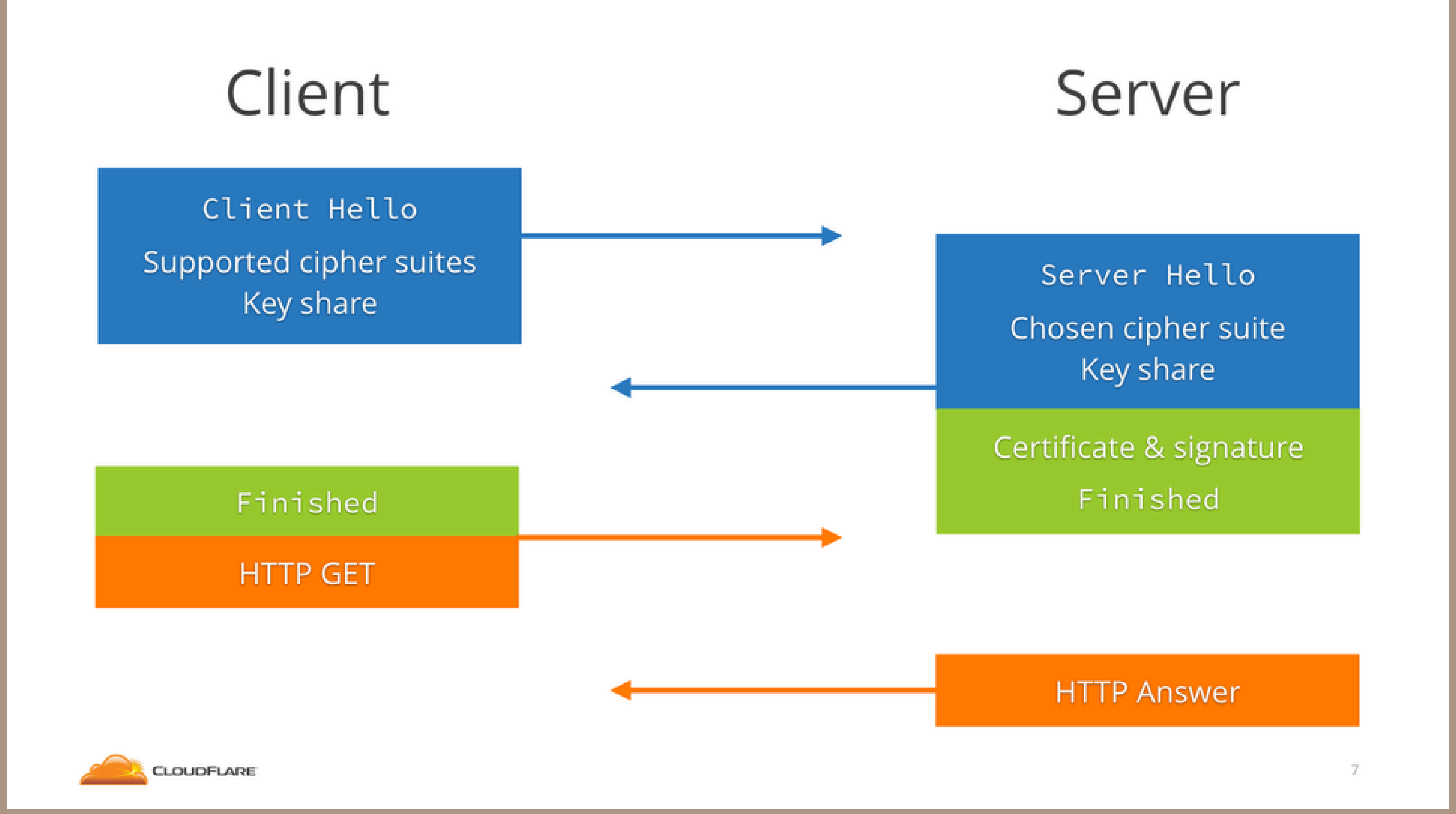


TLS HANDSHAKE

Etapa realizada após o TCP-Handshake

O Handshake estabelece a criptografia simétrica para a conexão.
Para tal, é necessário:

- O cliente enviar a versão do protocolo utilizada, lista de cifras suportadas e suas chaves públicas;
- O servidor envia seu certificado, uma assinatura digital, chave pública e a cifra escolhida para uso.
- Ambos cálculam, com os valores enviados, a chave simétrica.



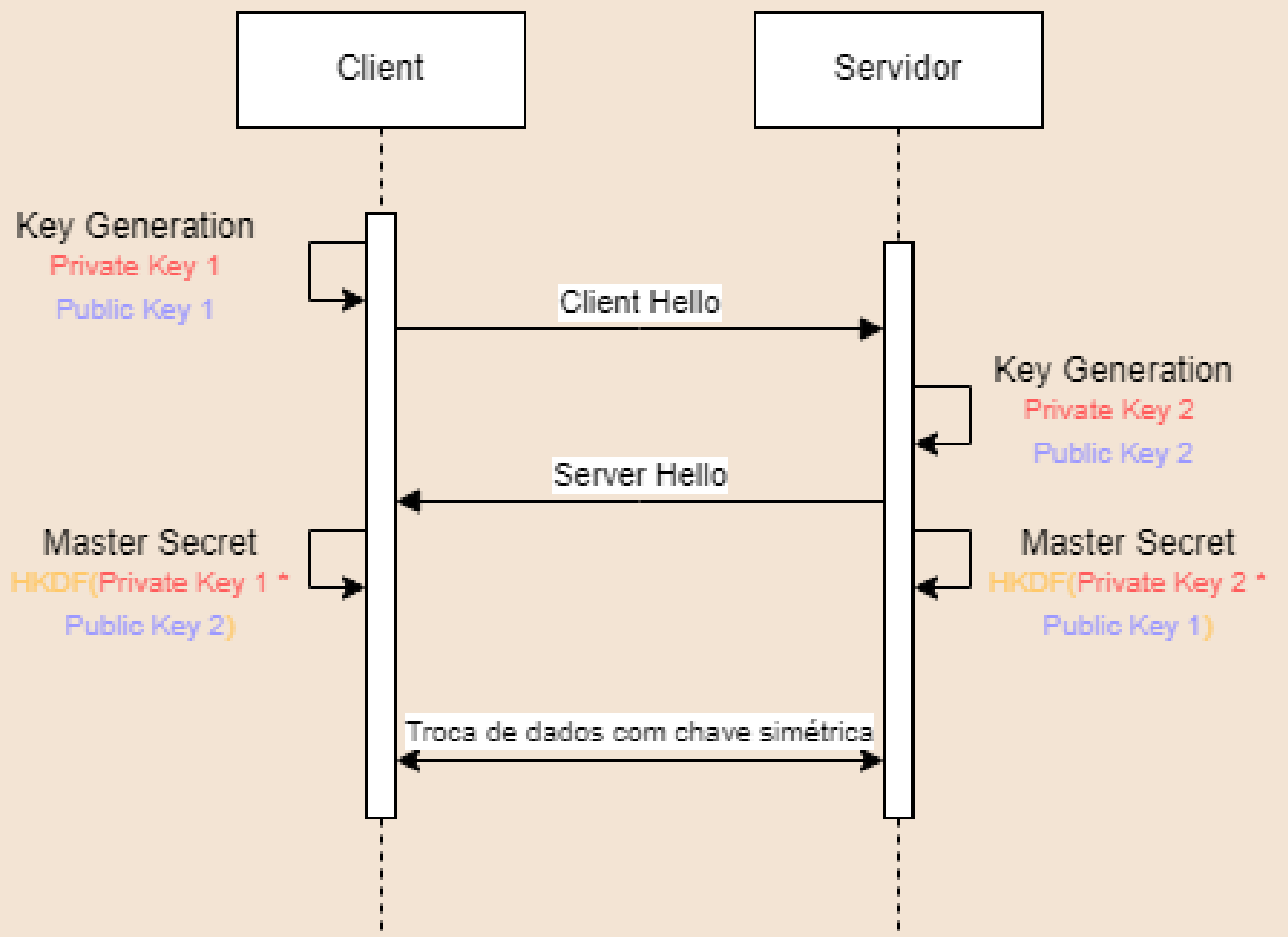
TLS HANDSHAKE

Etapa realizada após o TCP-Handshake

Para cada algoritmo a troca de chaves pode ser realizada de uma forma diferente

- Para o algoritmo x25519, há a troca das chaves públicas e em seguida ambos cliente e servidor utilizam a função HKDF para computar a chave simétrica
- Outra forma utilizada é a partir do envio da chave pública do cliente, para o servidor enviar sua chave pública já encriptada e, com a chave pública do servidor, o cliente envia sua chave privada

Troca de chaves para o algoritmo x25519

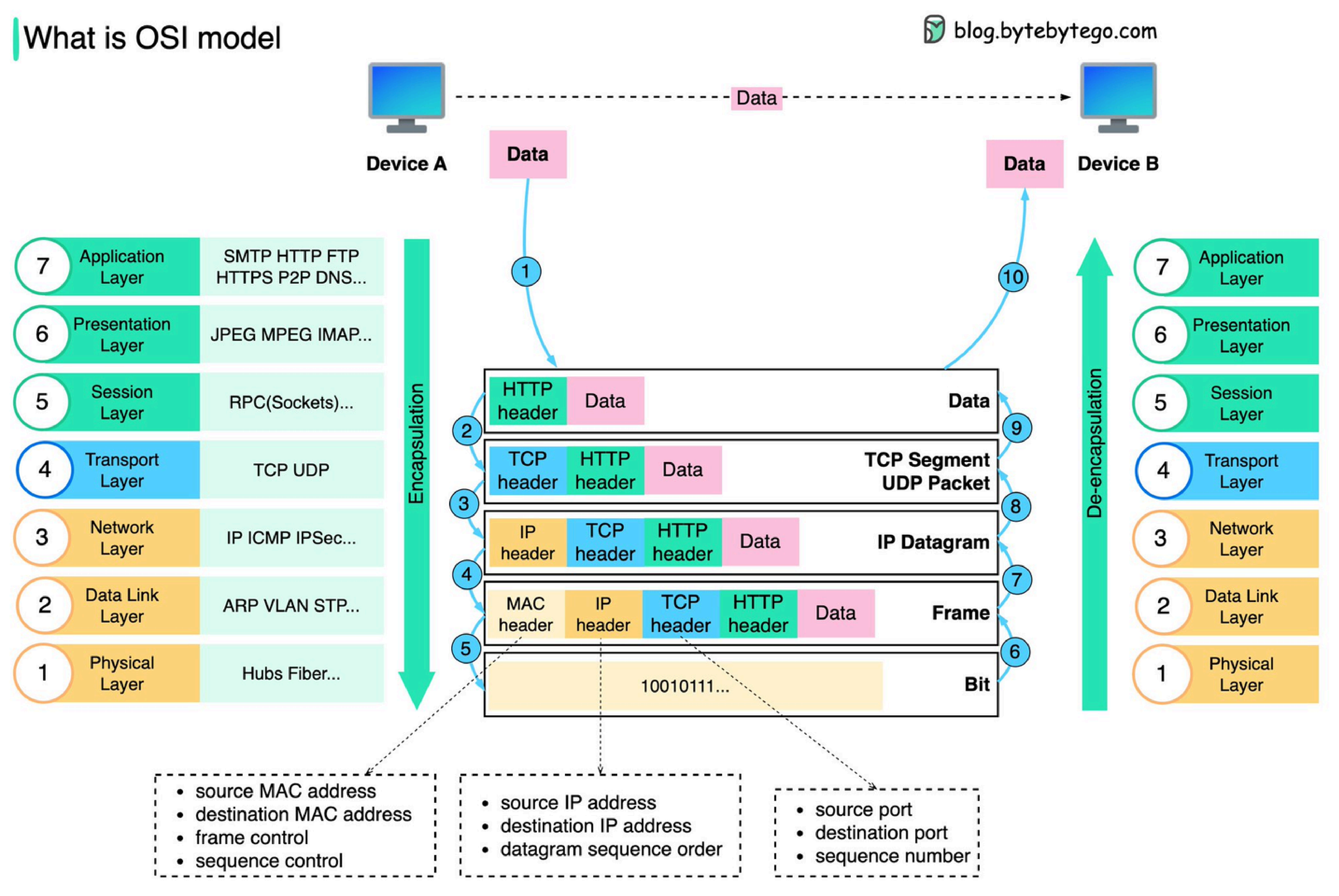


SCAPY

O Scapy é uma poderosa biblioteca interativa de manipulação de pacotes escrita em Python, sendo capaz de forjar, decodificar, enviar, capturar e combinar pacotes de diferentes protocolos.

scapy

RELEMBRANDO PACOTES



Fonte: bytebytego

MANIPULAÇÃO DE PACOTES

```python
from scapy.all import *
from scapy.layers.inet import IP, ICMP

src_ip = "192.168.2.113"
target_ip="172.23.228.217"

ip = IP(dst=target_ip, src=src_ip)

icmp = ICMP()
ping = sr1(ip / icmp)
ping.show()
```

Formação do pacote IP, definido IPs de origem e destino

Formação do pacote ICMP

Os valores não definidos serão gerados pelo próprio Scapy

Version	Length	Service type	Packet Length		
Identification			DF	MF	Fragment Offset
Time To Live		Transport	Header Checksum		
Source IP Address					
Destination IP Address					
Options					Padding

Type	Code	Checksum

MANIPULAÇÃO DE PACOTES

```python
from scapy.all import *
from scapy.layers.inet import IP, ICMP

src_ip = "192.168.2.113"
target_ip="172.23.228.217"

ip = IP(dst=target_ip, src=src_ip)

icmp = ICMP()
ping = sr1(ip / icmp)
ping.show()
```

A função sr1 é utilizada para enviar o pacote e receber a resposta
A função show é utilizada para fazer um print detalhado do pacote

Version	Length	Service type	Packet Length		
Identification			DF	MF	Fragment Offset
Time To Live		Transport	Header Checksum		
Source IP Address					
Destination IP Address					
Options					Padding

Type	Code	Checksum

MANIPULAÇÃO DE PACOTES

```
Begin emission:
WARNING: Mac address to reach destination not found. Using broadcast.
Finished sending 1 packets.
...*
Received 4 packets, got 1 answers, remaining 0 packets
###[ IP ]###
  version   = 4
  ihl       = 5
  tos       = 0x0
  len       = 28
  id        = 49096
  flags     =
  frag      = 0
  ttl       = 64
  proto     = icmp
  chksum    = 0x670e
  src       = 172.23.228.217
  dst       = 192.168.2.113
  \options   \
###[ ICMP ]###
     type      = echo-reply
     code      = 0
     chksum    = 0x0
     id        = 0x0
     seq       = 0x0
     unused    = ''
```

A função sr1 é utilizada para enviar o pacote e receber a resposta
A função show é utilizada para fazer um print detalhado do pacote

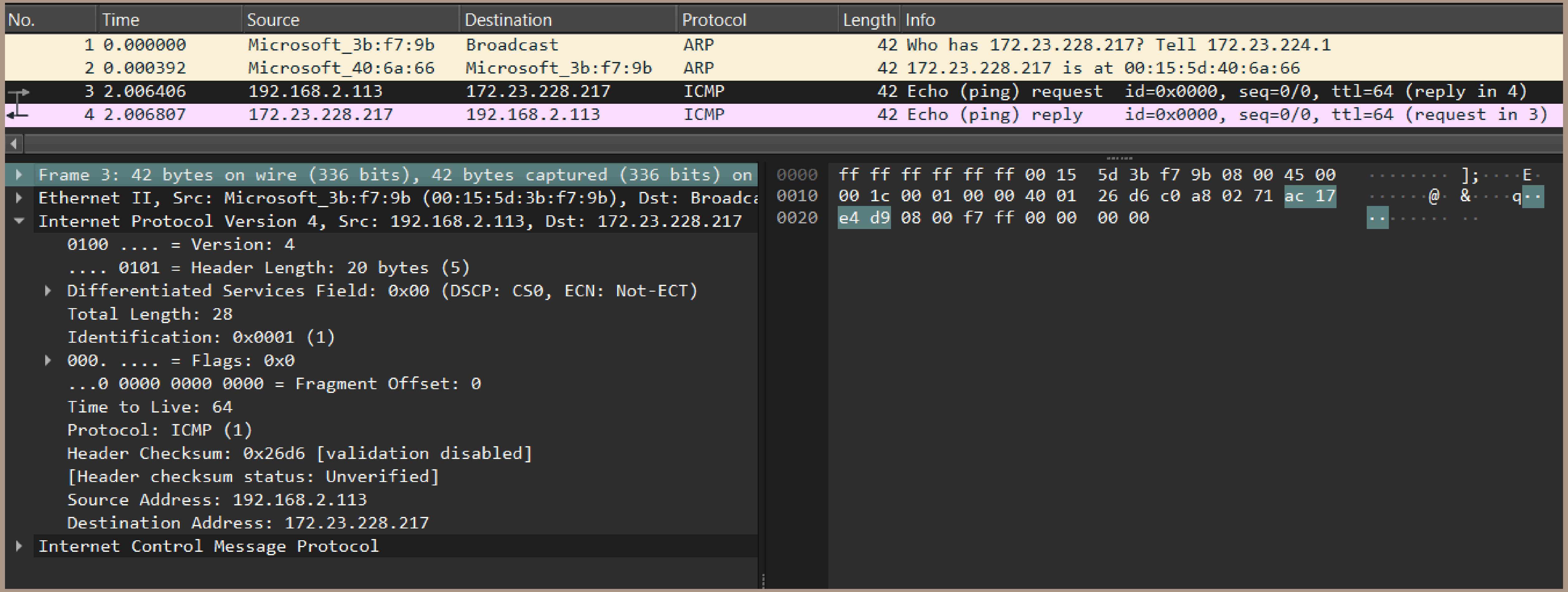
Version	Length	Service type	Packet Length		
Identification			DF	MF	Fragment Offset
Time To Live		Transport	Header Checksum		
Source IP Address					
Destination IP Address					
Options					Padding

Type	Code	Checksum

WIRESHARK

O wireshark é um sniffer que analisa tráfego de pacotes e organiza-o por protocolos e outras características.

SNIFFING DE PACOTES

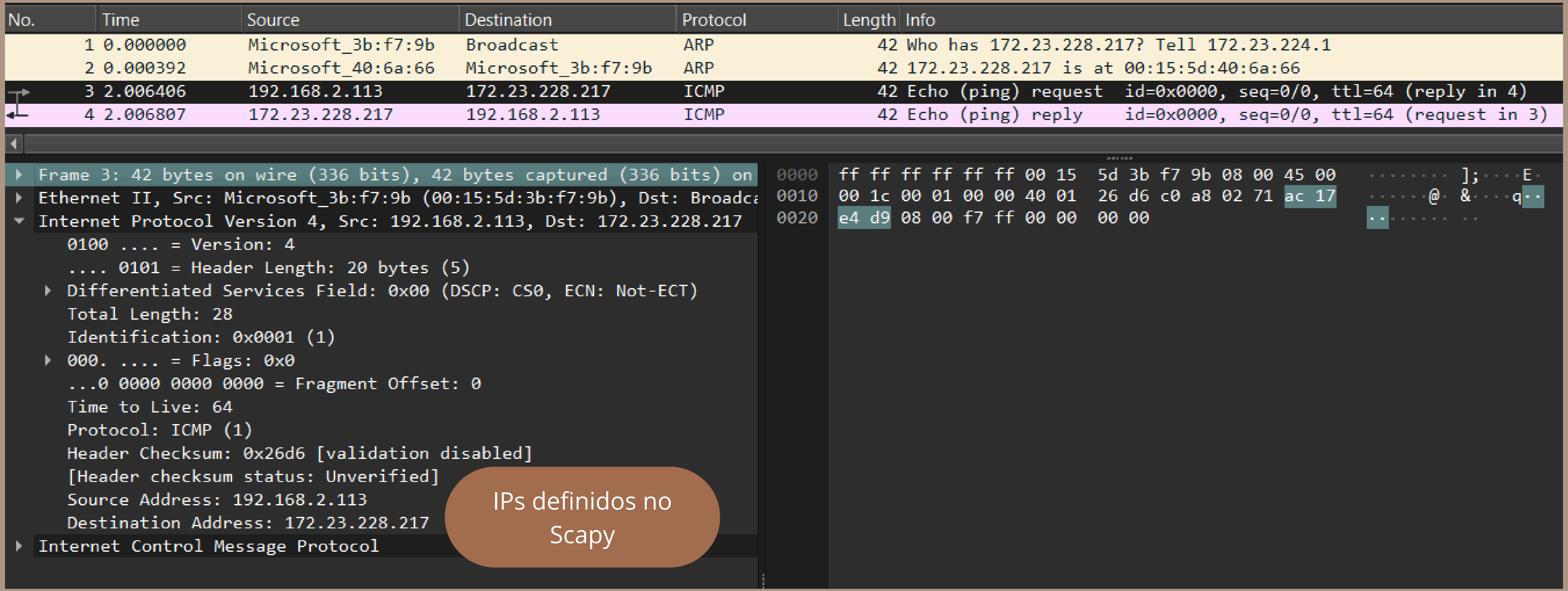



```python
src_ip = "192.168.2.113"
target_ip="172.23.228.217"

ip = IP(dst=target_ip, src=src_ip)

icmp = ICMP()
ping = sr1(ip / icmp)
ping.show()
```

SNIFFING DE PACOTES

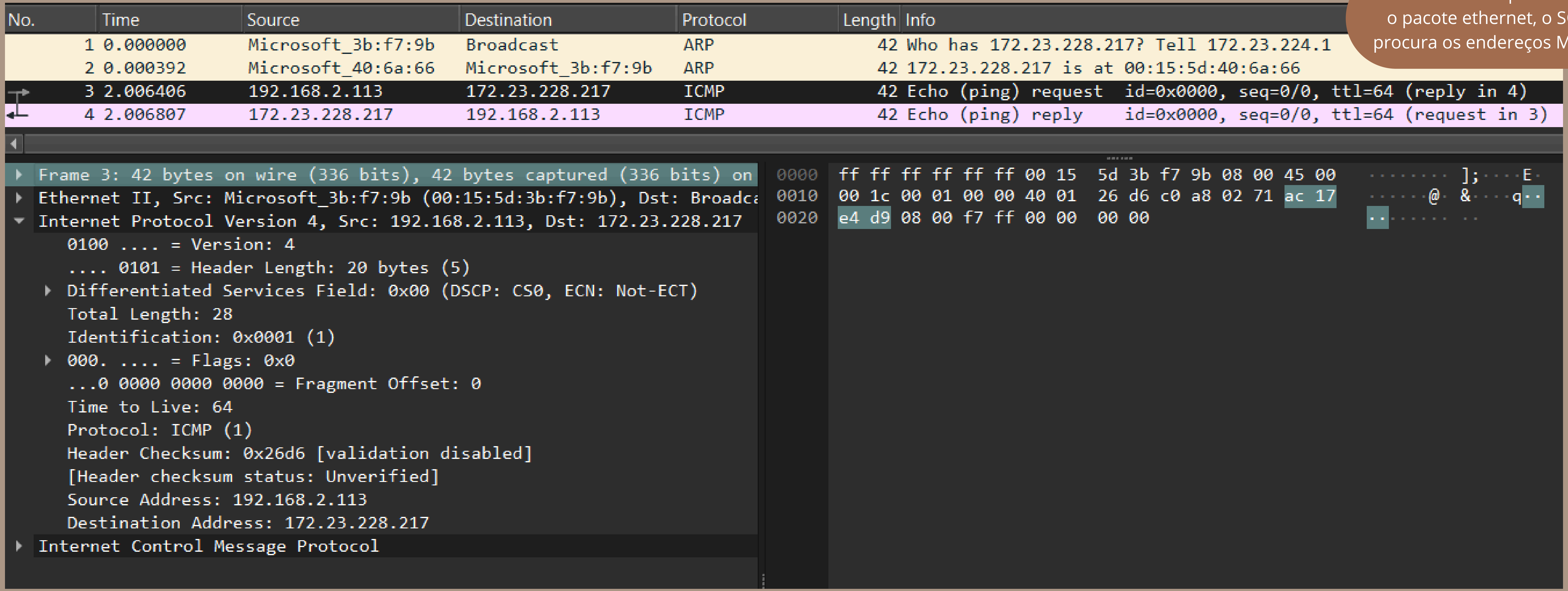



```python
src_ip = "192.168.2.113"
target_ip="172.23.228.217"

ip = IP(dst=target_ip, src=src_ip)

icmp = ICMP()
ping = sr1(ip / icmp)
ping.show()
```

SNIFFING DE PACOTES



Como não foi específicado o pacote ethernet, o SO procura os endereços MAC

```python
src_ip = "192.168.2.113"
target_ip="172.23.228.217"

ip = IP(dst=target_ip, src=src_ip)

icmp = ICMP()
ping = sr1(ip / icmp)
ping.show()
```

DEMO

SNIFFING TCP-HANDSHAKE

```
zak@DESKTOP-ZAK:/var/www/html$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1280
        inet 172.23.228.217  netmask 255.255.240.0  broadcast 172.23.239.255
        inet6 fe80::215:5dff:fe40:6a66  prefixlen 64  scopeid 0x20<link>
        ether 00:15:5d:40:6a:66  txqueuelen 1000  (Ethernet)
        RX packets 637  bytes 168539 (164.5 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 82  bytes 6848 (6.6 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 4  bytes 591 (591.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4  bytes 591 (591.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

zak@DESKTOP-ZAK:/var/www/html$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Utilizando o Python é possível abrir um servidor HTTP em sua máquina, no qual abrirá a porta 8000. O IP 0.0.0.0 indica que a porta será ouvida de todas interfaces da máquina

SNIFFING TCP-HANDSHAKE

O servidor aberto apontará para o diretório em que foi executado o comando.
Pela natureza do servidor http, ao acessa-lo ele irá buscar e retornar o arquivo index.html

```
zak@DESKTOP-ZAK:/var/www/html$ ls
index.html
zak@DESKTOP-ZAK:/var/www/html$ cat index.html
<!DOCTYPE html>
<html lang="pt-br">

<head>
    <title>Eu estou inseguro</title>
</head>

<body>
    <p>Minha senha é 123456wifi</p>
</body>
</html>
zak@DESKTOP-ZAK:/var/www/html$
```

SNIFFING TCP-HANDSHAKE

```python
from scapy.all import *
from scapy.layers.http import HTTP_Client
load_layer("http")

http_uri = "http://"
target_port=8000
target_ip="172.23.228.217"

client_url = http_uri+target_ip+":"+str(target_port)

client = HTTP_Client()
resp = client.request(client_url)
client.close()

```

O Scapy fornece as classes Client, que estabelecem a conexão TCP e enviam requisições, que, neste caso, faz uma requisição GET HTTP

SNIFFING TCP-HANDSHAKE

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.23.224.1	172.23.228.217	TCP	66	33835 → 8000 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
2	0.000448	172.23.228.217	172.23.224.1	TCP	66	8000 → 33835 [SYN, ACK] Seq=0 Ack=1 Win=64480 Len=0 MSS=1240 SACK_PERM WS=128
3	0.000539	172.23.224.1	172.23.228.217	TCP	54	33835 → 8000 [ACK] Seq=1 Ack=1 Win=262656 Len=0
4	0.021945	172.23.224.1	172.23.228.217	HTTP	212	GET / HTTP/1.1
5	0.022271	172.23.228.217	172.23.224.1	TCP	54	8000 → 33835 [ACK] Seq=1 Ack=159 Win=64384 Len=0
6	0.024918	172.23.228.217	172.23.224.1	TCP	239	8000 → 33835 [PSH, ACK] Seq=1 Ack=159 Win=64384 Len=185 [TCP segment of a reassembled PDU]
7	0.024984	172.23.228.217	172.23.224.1	HTTP	204	HTTP/1.0 200 OK (text/html)
8	0.025008	172.23.224.1	172.23.228.217	TCP	54	33835 → 8000 [ACK] Seq=159 Ack=337 Win=262400 Len=0
9	0.029365	172.23.224.1	172.23.228.217	TCP	54	33835 → 8000 [FIN, ACK] Seq=159 Ack=337 Win=262400 Len=0
10	0.029554	172.23.228.217	172.23.224.1	TCP	54	8000 → 33835 [ACK] Seq=337 Ack=160 Win=64384 Len=0

Com o HTTP Client e a requisição realizada podemos ver:

- TCP - Handshake
- Troca de dados HTTP em texto claro
- Finalização da conexão

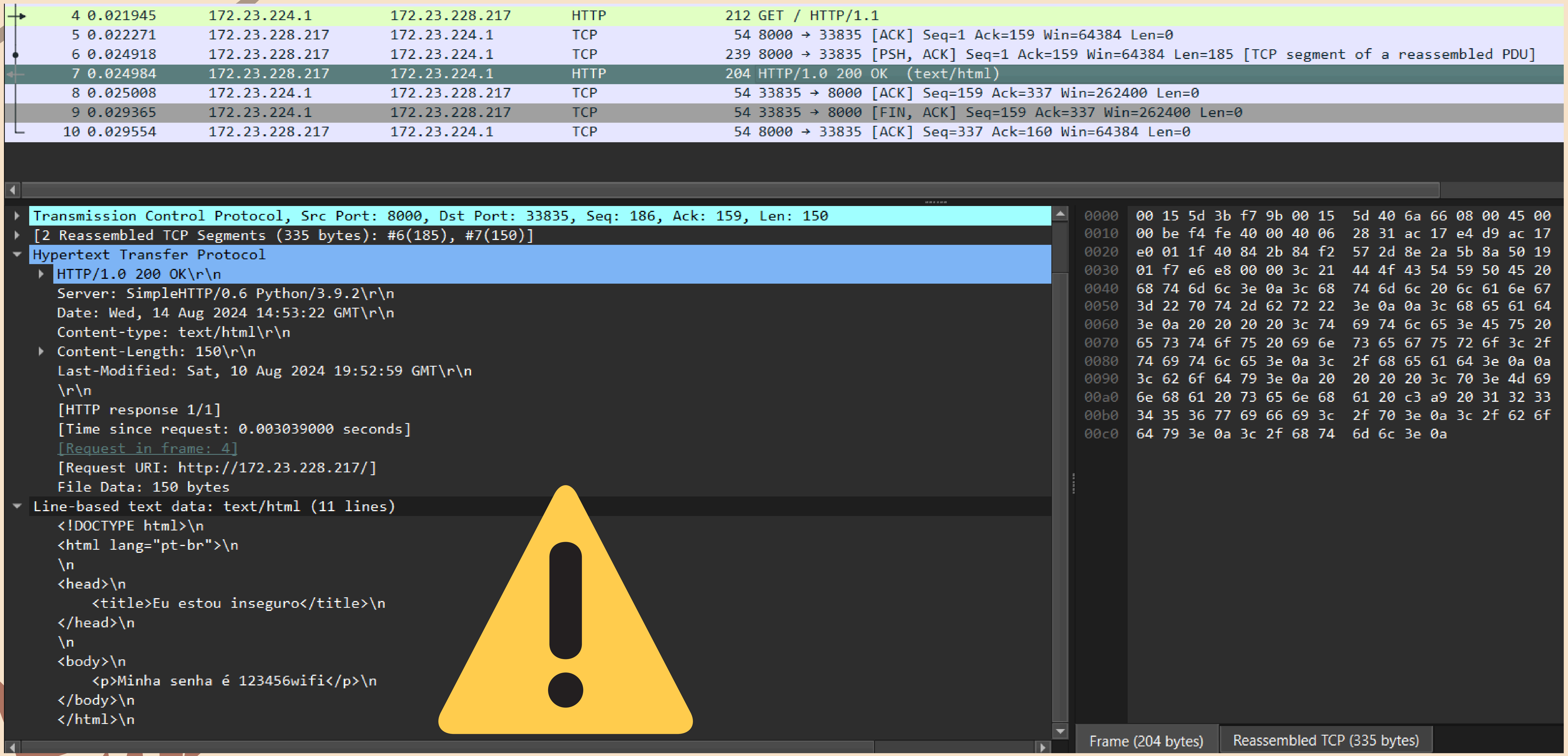
SNIFFING TCP-HANDSHAKE

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.23.224.1	172.23.228.217	TCP	66	33835 → 8000 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
2	0.000448	172.23.228.217	172.23.224.1	TCP	66	8000 → 33835 [SYN, ACK] Seq=0 Ack=1 Win=64480 Len=0 MSS=1240 SACK_PERM WS=128
3	0.000539	172.23.224.1	172.23.228.217	TCP	54	33835 → 8000 [ACK] Seq=1 Ack=1 Win=262656 Len=0

TCP Handshake

- O cliente envia a flag SYN sinalizando o desejo de estabelecer a conexão
- O servidor retorna com SYN, ACK, permitindo a conexão
- O cliente finaliza enviando um ACK

SNIFFING TCP-HANDSHAKE



A troca de dados HTTP a partir da requisição GET e resposta 200 OK se dá de forma não criptografada!

SNIFFING TLS-HANDSHAKE

```python
from scapy.all import *
from scapy.layers.tls.all import *

class ModifiedTLSClientAutomaton(TLSClientAutomaton):
    @ATMT.condition(TLSClientAutomaton.PREPARE_CLIENTFLIGHT1)
    def should_add_ClientHello(self):
        if self.client_hello:
            p = self.client_hello
        else:
            p = TLSClientHello()
        self.add_msg(p)
        raise self.ADDED_CLIENTHELLO()

target_domain = "142.250.219.4"
ciphers = [TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384]
ciphers += [TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256]
ciphers += [TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384]
ciphers += [TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256]
ciphers += [TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384]
ciphers += [TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256]
ciphers += [TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384]
ciphers += [TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256]
ciphers += [TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA]
ciphers += [TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA]
ciphers += [TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA]
ciphers += [TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA]

compression='null'
ext1 = TLS_Ext_ServerName(servernames=ServerName(servername=target_domain))
ext2 = TLS_Ext_CSR(stype='ocsp', req=OCSPStatusRequest())
ext3 = TLS_Ext_SupportedEllipticCurves(groups=['x25519', 'secp256r1', 'secp384r1'])
ext4 = TLS_Ext_SupportedPointFormat(ecpl='uncompressed')
ext5 = TLS_Ext_SignatureAlgorithms(sig_algs=['sha256+rsa', 'sha384+rsa', 'sha1+rsa', 'sha256+ecdsa', 'sha384+ecdsa', 'sha1+ecdsa', 'sha1+dsa', 'sha512+rsa', 'sha512+ecdsa'])
ext = [ext1, ext2, ext3, ext4, ext5]

ch = TLSClientHello(gmt_unix_time=10000, ciphers=ciphers, ext=ext, comp=compression)

t = ModifiedTLSClientAutomaton(client_hello=ch, server=target_domain, dport=443)
t.run()
```

SNIFFING TLS-HANDSHAKE

```python
import http.server
import ssl

server_ip = '0.0.0.0'
server_port = 443

server_address = (server_ip, server_port)

httpd = http.server.HTTPServer(server_address, http.server.SimpleHTTPRequestHandler)

httpd.socket = ssl.wrap_socket(httpd.socket,
                               keyfile="key.pem",
                               certfile="cert.pem",
                               server_side=True)

print(f"Serving on https://{server_ip}:{server_port}")
httpd.serve_forever()
```

Para abrir o servidor HTTPS é necessário envolver o servidor HTTP no SSL/TLS

SNIFFING TLS-HANDSHAKE

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.23.224.1	172.23.228.217	TCP	54	2148 → 443 [ACK] Seq=1 Ack=1 Win=1026 Len=0
2	0.000330	172.23.224.1	172.23.228.217	TCP	66	2149 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
3	0.000520	172.23.228.217	172.23.224.1	TCP	66	443 → 2149 [SYN, ACK] Seq=0 Ack=1 Win=64480 Len=0 MSS=1240 SACK_PERM WS=128
4	0.000569	172.23.224.1	172.23.228.217	TCP	54	2149 → 443 [ACK] Seq=1 Ack=1 Win=2097920 Len=0
5	0.000855	172.23.224.1	172.23.228.217	TLSv1.3	2122	Client Hello
6	0.000949	172.23.228.217	172.23.224.1	TCP	54	443 → 2149 [ACK] Seq=1 Ack=2069 Win=64128 Len=0
7	0.004534	172.23.228.217	172.23.224.1	TLSv1.3	2425	Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data
8	0.004626	172.23.224.1	172.23.228.217	TCP	54	2149 → 443 [ACK] Seq=2069 Ack=2372 Win=2097920 Len=0
9	0.004880	172.23.224.1	172.23.228.217	TLSv1.3	134	Change Cipher Spec, Application Data
10	0.005014	172.23.224.1	172.23.228.217	TLSv1.3	770	Application Data
11	0.005104	172.23.228.217	172.23.224.1	TLSv1.3	309	Application Data
12	0.008052	172.23.228.217	172.23.224.1	TLSv1.3	688	Application Data, Application Data, Application Data
13	0.008101	172.23.224.1	172.23.228.217	TCP	54	2149 → 443 [ACK] Seq=2865 Ack=3262 Win=2097152 Len=0
14	0.009507	172.23.224.1	172.23.228.217	TCP	54	2149 → 443 [FIN, ACK] Seq=2865 Ack=3262 Win=2097152 Len=0
15	0.009684	172.23.228.217	172.23.224.1	TCP	54	443 → 2149 [ACK] Seq=3262 Ack=2866 Win=64384 Len=0

Com a aplicação do TLS temos:

- TCP - Handshake
- TLS - Handshake
- Troca de dados HTTP criptografados (HTTPS)
- Finalização da conexão

SNIFFING TLS-HANDSHAKE

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.23.224.1	172.23.228.217	TCP	54	2148 → 443 [ACK] Seq=1 Ack=1 Win=1026 Len=0
2	0.000330	172.23.224.1	172.23.228.217	TCP	66	2149 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
3	0.000520	172.23.228.217	172.23.224.1	TCP	66	443 → 2149 [SYN, ACK] Seq=0 Ack=1 Win=64480 Len=0 MSS=1240 SACK_PERM WS=128
4	0.000569	172.23.224.1	172.23.228.217	TCP	54	2149 → 443 [ACK] Seq=1 Ack=1 Win=2097920 Len=0

TCP Handshake

TLS Handshake

5	0.000855	172.23.224.1	172.23.228.217	TLSv1.3	2122	Client Hello
6	0.000949	172.23.228.217	172.23.224.1	TCP	54	443 → 2149 [ACK] Seq=1 Ack=2069 Win=64128 Len=0
7	0.004534	172.23.228.217	172.23.224.1	TLSv1.3	2425	Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data
8	0.004626	172.23.224.1	172.23.228.217	TCP	54	2149 → 443 [ACK] Seq=2069 Ack=2372 Win=2097920 Len=0
9	0.004880	172.23.224.1	172.23.228.217	TLSv1.3	134	Change Cipher Spec, Application Data

```
Handshake Protocol: Client Hello
    Handshake Type: Client Hello (1)
    Length: 2059
    Version: TLS 1.2 (0x0303)
    Random: edcbc818633cd0f13d3bd18065e4966c89f893f358327c701a7bdcab966f6b1c
    Session ID Length: 32
    Session ID: 20c81609d41c2a2a61ed45ead68bbed1b742516beaf9d7870215681ad580821c
    Cipher Suites Length: 32
    Cipher Suites (16 suites)
        Cipher Suite: Reserved (GREASE) (0xcaca)
        Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
        Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
        Cipher Suite: TLS_CHACHA20_POLY1305_SHA256 (0x1303)
        Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
        Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
```

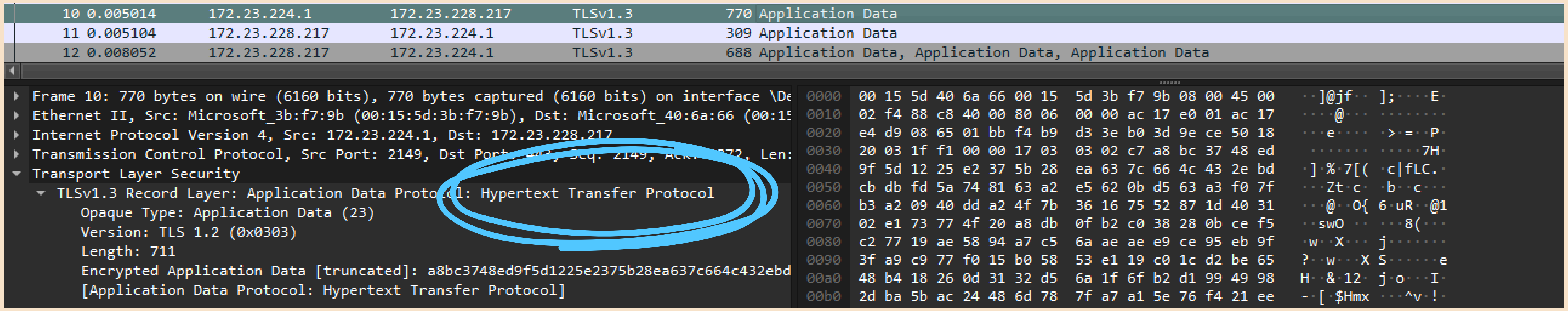
```
Handshake Protocol: Server Hello
    Handshake Type: Server Hello (2)
    Length: 118
    Version: TLS 1.2 (0x0303)
    Random: 2449dea3669b64f757608bbd841a8afd0fba2714543c053246689673dc86d1ce
    Session ID Length: 32
    Session ID: 20c81609d41c2a2a61ed45ead68bbed1b742516beaf9d7870215681ad580821c
    Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
```

```
Key Share Entry: Group: x25519, Key Exchange length: 32
    Group: x25519 (29)
    Key Exchange Length: 32
    Key Exchange: 31543ead09bfc760a246aaac926a373b908588f5d17999d3aa1a2caba9563e67
```

```
Key Share Entry: Group: x25519, Key Exchange length: 32
    Group: x25519 (29)
    Key Exchange Length: 32
    Key Exchange: 7e777f2844d23cd3d2612370031ea5b5e6dfaa91bd63f508b08d00b046b8621f
```

Chaves públicas

SNIFFING TLS-HANDSHAKE



Troca de dados HTTP encriptados

CRIPTOGRAFIA PÓS-QUÂNTICA

POR QUÊ

- Em 1994 o cientista da computação Peter Shor apresenta um algoritmo capaz de quebrar o RSA de forma polinomial com base na computação quântica
- Dada a importância do RSA, começou-se a implementar os algoritmos pós-quânticos (PQ)

COMO

- Os algoritimos PQ utilizam de problemas matemáticos considerados mais difíceis que a fatoração, no qual o RSA é baseado
- Serão utilizados na criptografia simétrica, durante o TLS-Handshake

QUANDO

- Desde 2016 há estudos para a padronização de algorítmos PQ nas trocas de dados em redes
- Em 13/08/2024 foram definidos pelo NIST 3 algoritmos para padronização: ML-KEM, ML-DSA e SLH-DSA.

IMPLEMENTAÇÃO DA CRIPTOGRAFIA PÓS-QUÂNTICA

- A implementação da criptografia PQ atráves do algoritmo ML-KEM se baseia nas padronizações do NIST.
- Uma biblioteca já vem desenvolvendo algoritmos para a padronização, a liboqs
- Utilizando a liboqs, é possível fazer uma implementação customizada do Scapy, para aplicar a criptografia PQ

- Todavia, sua utilização em troca de dados ainda não é possível, pois os valores dos campos de pacotes de dados são definidos em RFCs, como a 8446, do TLS 1.3.
- Mas podemos ir compreendendo como esta conexão será estabelecida!

```
Internet Engineering Task Force (IETF)                       E. Rescorla
Request for Comments: 8446                                       Mozilla
Obsoletes: 5077, 5246, 6961                                  August 2018
Updates: 5705, 6066
Category: Standards Track
ISSN: 2070-1721


        The Transport Layer Security (TLS) Protocol Version 1.3

Abstract

   This document specifies version 1.3 of the Transport Layer Security
   (TLS) protocol.  TLS allows client/server applications to communicate
   over the Internet in a way that is designed to prevent eavesdropping,
   tampering, and message forgery.

   This document updates RFCs 5705 and 6066, and obsoletes RFCs 5077,
   5246, and 6961.  This document also specifies new requirements for
   TLS 1.2 implementations.
```

ETAPAS

The Transport Layer Security (TLS) Protocol
Version 1.2

```
CipherSuite TLS_RSA_WITH_3DES_EDE_CBC_SHA          = { 0x00,0x0A };
```

RFC 5246

```python
class TLS_RSA_WITH_3DES_EDE_CBC_SHA(_GenericCipherSuite):
    val = 0x000A
```

github/scapy

```python
class RSAPrivateKey(metaclass=abc.ABCMeta): ...


RSAPrivateKeyWithSerialization = RSAPrivateKey
RSAPrivateKey.register(rust_openssl.rsa.RSAPrivateKey)


class RSAPublicKey(metaclass=abc.ABCMeta): ...


RSAPublicKeyWithSerialization = RSAPublicKey
RSAPublicKey.register(rust_openssl.rsa.RSAPublicKey)

RSAPrivateNumbers = rust_openssl.rsa.RSAPrivateNumbers
RSAPublicNumbers = rust_openssl.rsa.RSAPublicNumbers


def generate_private_key(
    public_exponent: int,
    key_size: int,
    backend: typing.Any = None,
) -> RSAPrivateKey:
    _verify_rsa_parameters(public_exponent, key_size)
    return rust_openssl.rsa.generate_private_key(public_exponent, key_size)
```

github/cryptography

ETAPAS

```python
class RSAPrivateKey(metaclass=abc.ABCMeta): ...


RSAPrivateKeyWithSerialization = RSAPrivateKey
RSAPrivateKey.register(rust_openssl.rsa.RSAPrivateKey)


class RSAPublicKey(metaclass=abc.ABCMeta): ...


RSAPublicKeyWithSerialization = RSAPublicKey
RSAPublicKey.register(rust_openssl.rsa.RSAPublicKey)

RSAPrivateNumbers = rust_openssl.rsa.RSAPrivateNumbers
RSAPublicNumbers = rust_openssl.rsa.RSAPublicNumbers


def generate_private_key(
    public_exponent: int,
    key_size: int,
    backend: typing.Any = None,
) -> RSAPrivateKey:
    _verify_rsa_parameters(public_exponent, key_size)
    return rust_openssl.rsa.generate_private_key(public_exponent, key_size)
```

github/cryptography

```c
int RSA_generate_key_ex(RSA *rsa, int bits, BIGNUM *e_value, BN_GENCB *cb)
{
    if (rsa->meth->rsa_keygen != NULL)
        return rsa->meth->rsa_keygen(rsa, bits, e_value, cb);

    return RSA_generate_multi_prime_key(rsa, bits, RSA_DEFAULT_PRIME_NUM,
                                        e_value, cb);
}
```

github/openssl

ETAPAS

```c
int RSA_generate_key_ex(RSA *rsa, int bits, BIGNUM *e_value, BN_GENCB *cb)
{
    if (rsa->meth->rsa_keygen != NULL)
        return rsa->meth->rsa_keygen(rsa, bits, e_value, cb);

    return RSA_generate_multi_prime_key(rsa, bits, RSA_DEFAULT_PRIME_NUM,
                                        e_value, cb);
}
```

github/openssl

```c
static apr_status_t ssl_init_ctx_cipher_suite(server_rec *s,
                                              apr_pool_t *p,
                                              apr_pool_t *ptemp,
                                              modssl_ctx_t *mctx)
{
    SSL_CTX *ctx = mctx->ssl_ctx;
    const char *suite;

    /*
     *  Configure SSL Cipher Suite. Always disable NULL and export ciphers,
     *  see also ssl_engine_config.c:ssl_cmd_SSLCipherSuite().
     *  OpenSSL's SSL_DEFAULT_CIPHER_LIST includes !aNULL:!eNULL from 0.9.8f,
     *  and !EXP from 0.9.8zf/1.0.1m/1.0.2a, so append them while we support
     *  earlier versions.
     */
    suite = mctx->auth.cipher_suite ? mctx->auth.cipher_suite :
            apr_pstrcat(ptemp, SSL_DEFAULT_CIPHER_LIST, ":!aNULL:!eNULL:!EXP",
                        NULL);

    ap_log_error(APLOG_MARK, APLOG_TRACE1, 0, s,
                 "Configuring permitted SSL ciphers [%s]",
                 suite);

    if (!SSL_CTX_set_cipher_list(ctx, suite)) {
        ap_log_error(APLOG_MARK, APLOG_EMERG, 0, s, APLOGNO(01898)
                "Unable to configure permitted SSL ciphers");
        ssl_log_ssl_error(SSLLOG_MARK, APLOG_EMERG, s);
        return ssl_die(s);
    }
```

github/httpd

OBRIGADO