PCAP Traffic Analysis with Wireshark

Taha Zouggari

July 2025

Contents

1 Introduction

Network traffic analysis is a core component of cybersecurity. It allows analysts to inspect communication between systems to detect intrusions, investigate incidents, or extract transferred data. One of the most powerful tools for this task is **Wireshark**.

2 What is Wireshark?

Wireshark is an open-source network protocol analyzer that lets you capture and inspect the details of network traffic in real-time or from a previously recorded file (`.pcap`). It supports hundreds of protocols and offers advanced filtering, protocol decoding, and stream reassembly features.

3 Protocol Overview

In this report, we analyze two insecure legacy protocols captured in packet dumps:

- **Telnet:** A remote shell protocol where all traffic, including credentials, is sent in plaintext.
- **FTP (File Transfer Protocol):** Used to transfer files. We focus on the `PASV` mode (passive mode), which is commonly exploited due to lack of encryption.

4 Challenges

4.1 Challenge 1 – Telnet Credentials and Commands

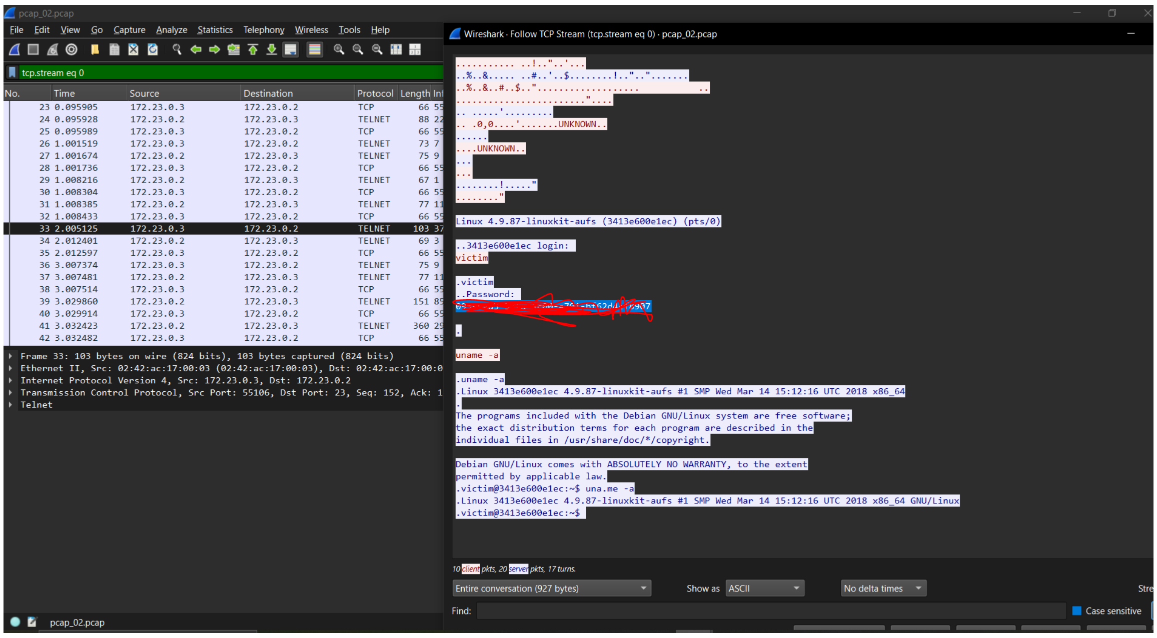
This challenge provides a PCAP file containing a single Telnet session between a client and a server. The objective is to extract sensitive information transmitted over this insecure protocol.

Objective

Retrieve the username, password, and executed command.

Steps

1. Open `pcap_02.pcap` in Wireshark.
2. Right-click a packet and choose "Follow TCP Stream".
3. Inspect the full Telnet session shown in ASCII format.



```
No.  Time       Source       Destination  Protocol  Length Inf
  23 0.095905   172.23.0.3   172.23.0.2   TCP        66 55
  24 0.095928   172.23.0.2   172.23.0.3   TELNET     88 22
  25 0.095989   172.23.0.3   172.23.0.2   TCP        66 55
  26 1.001519   172.23.0.3   172.23.0.2   TELNET     73 7
  27 1.001674   172.23.0.2   172.23.0.3   TELNET     75 9
  28 1.001736   172.23.0.3   172.23.0.2   TCP        66 55
  29 1.008216   172.23.0.2   172.23.0.3   TELNET     67 1
  30 1.008304   172.23.0.3   172.23.0.2   TCP        66 55
  31 1.008385   172.23.0.2   172.23.0.3   TELNET     77 11
  32 1.008433   172.23.0.3   172.23.0.2   TCP        66 55
  33 2.005125   172.23.0.3   172.23.0.2   TELNET    103 37
  34 2.012401   172.23.0.2   172.23.0.3   TELNET     69 3
  35 2.012597   172.23.0.3   172.23.0.2   TCP        66 55
  36 3.007374   172.23.0.3   172.23.0.2   TELNET     75 9
  37 3.007481   172.23.0.2   172.23.0.3   TELNET     77 11
  38 3.007514   172.23.0.3   172.23.0.2   TCP        66 55
  39 3.029860   172.23.0.2   172.23.0.3   TELNET    151 85
  40 3.029914   172.23.0.3   172.23.0.2   TCP        66 55
  41 3.032423   172.23.0.2   172.23.0.3   TELNET    360 29
  42 3.032482   172.23.0.3   172.23.0.2   TCP        66 55

> Frame 33: 103 bytes on wire (824 bits), 103 bytes captured (824 bits)
> Ethernet II, Src: 02:42:ac:17:00:03 (02:42:ac:17:00:03), Dst: 02:42:ac:17:00:0
> Internet Protocol Version 4, Src: 172.23.0.3, Dst: 172.23.0.2
> Transmission Control Protocol, Src Port: 55106, Dst Port: 23, Seq: 152, Ack: 1
> Telnet
```

Wireshark · Follow TCP Stream (tcp.stream eq 0) · pcap_02.pcap

```
........... ..!.."..'...
..%..&..... ..#..'..$........!.."..".......
..%..&..#..$.."..................           ..
........................."....
.. .....'.........
.. .0,0....'.......UNKNOWN..
......
....UNKNOWN..
...
...
........!....."
......."

Linux 4.9.87-linuxkit-aufs (3413e600e1ec) (pts/0)

..3413e600e1ec login: 
victim

.victim
..Password: 
[redacted]

.

uname -a

.uname -a
.Linux 3413e600e1ec 4.9.87-linuxkit-aufs #1 SMP Wed Mar 14 15:12:16 UTC 2018 x86_64
.
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
.victim@3413e600e1ec:~$ una.me -a
.Linux 3413e600e1ec 4.9.87-linuxkit-aufs #1 SMP Wed Mar 14 15:12:16 UTC 2018 x86_64 GNU/Linux
.victim@3413e600e1ec:~$ 
```

10 client pkts, 20 server pkts, 17 turns.

Entire conversation (927 bytes) — Show as: ASCII — No delta times

Figure 1: Telnet stream showing login credentials and a command

Results

- **Username:** `victim`
- **Password:** `0959ead3-52fa-4c00-a701-bf624dcf8907`
- **Command:** `uname -a`

Analysis

Because Telnet does not encrypt its traffic, all transmitted data (credentials and commands) can be intercepted easily. This highlights why Telnet should not be used in modern systems.

4.2 Challenge 2 – FTP Passive Mode File Retrieval

This challenge involves analyzing an FTP session in passive mode. The goal is to extract a file transmitted during the session and recover a key from it.

Objective

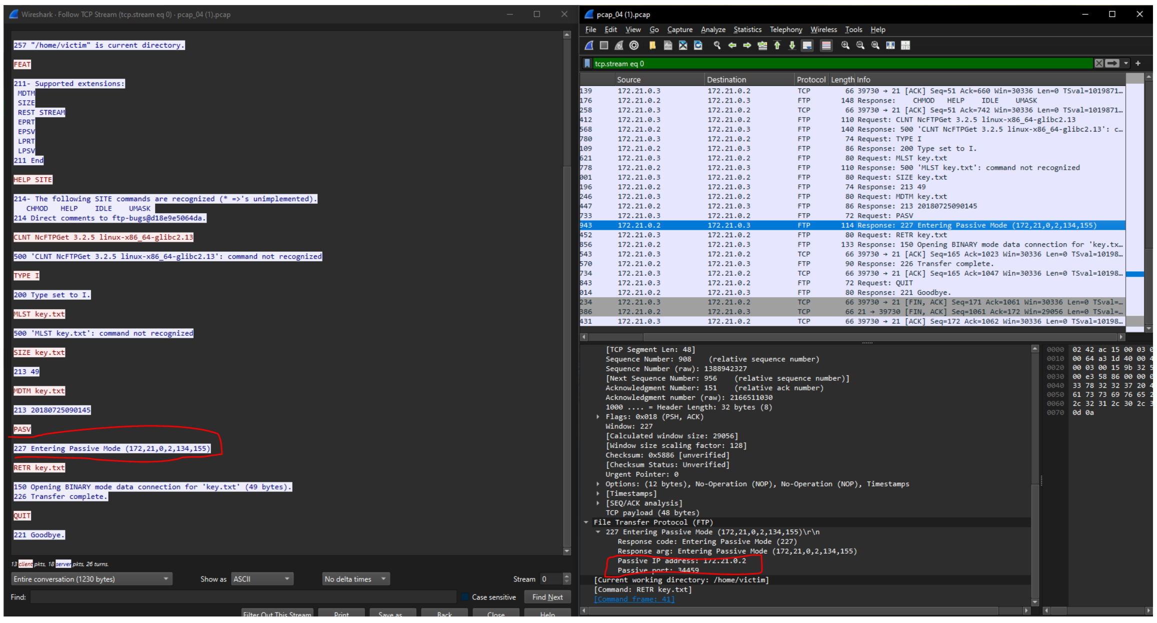
Find the IP and port communicated via the `PASV` command, connect to the data stream, and extract the file.

Steps

1. Open `pcap_04.pcap` in Wireshark.
2. Initially, we use `tcp.stream eq 0` to analyze the FTP control connection, where we observe the PASV command and its response.
3. After decoding the port number (`34459`), we must switch to the actual data transfer stream.
4. Since the data is transferred over a new TCP connection (to the specified port), we identify that connection in Wireshark.
5. We apply `tcp.stream eq 1` to view this new TCP session that carries the file contents.
6. Look for the line:

   ```
   227 Entering Passive Mode (172,21,0,2,134,155)
   ```

7. Decode IP and port:
   - IP: `172.21.0.2`
   - Port $= 134 \times 256 + 155 = 34459$
8. Apply filter `tcp.stream eq 1` to view the data transfer.
9. Follow TCP Stream and extract the key.



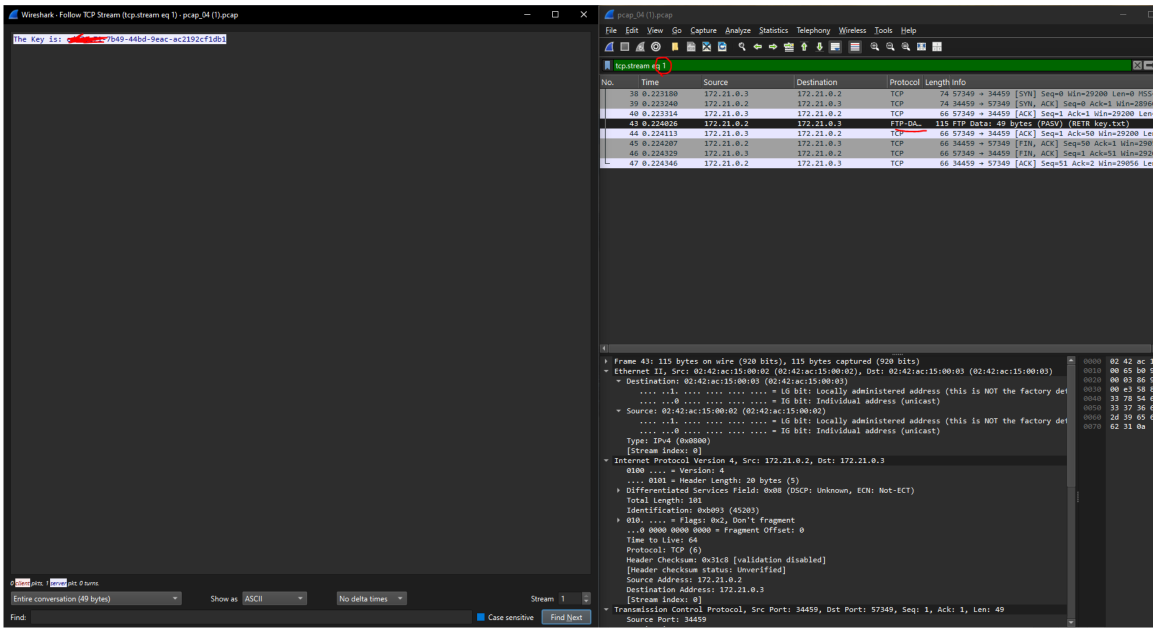
```
Wireshark · Follow TCP Stream (tcp.stream eq 0) · pcap_04 (1).pcap

257 "/home/victim" is current directory.

FEAT

211- Supported extensions:
 MDTM
 SIZE
 REST STREAM
 EPRT
 EPSV
 LPRT
 LPSV
211 End

HELP SITE

214- The following SITE commands are recognized (* =>'s unimplemented).
   CHMOD   HELP    IDLE    UMASK
214 Direct comments to ftp-bugs@d18e9e5064da.

CLNT NcFTPGet 3.2.5 linux-x86_64-glibc2.13

500 'CLNT NcFTPGet 3.2.5 linux-x86_64-glibc2.13': command not recognized

TYPE I

200 Type set to I.

MLST key.txt

500 'MLST key.txt': command not recognized

SIZE key.txt

213 49

MDTM key.txt

213 20180725090145

PASV

227 Entering Passive Mode (172,21,0,2,134,155)

RETR key.txt

150 Opening BINARY mode data connection for 'key.txt' (49 bytes).
226 Transfer complete.

QUIT

221 Goodbye.

13 client pkts, 18 server pkts, 26 turns.
Entire conversation (1230 bytes)   Show as ASCII   No delta times   Stream 0
Find:   Case sensitive
```

```
pcap_04 (1).pcap
tcp.stream eq 0

     Source       Destination  Protocol Length Info
139  172.21.0.3   172.21.0.2   TCP      66 39730 → 21 [ACK] Seq=51 Ack=660 Win=30336 Len=0 TSval=1019871…
176  172.21.0.2   172.21.0.3   FTP      148 Response:    CHMOD   HELP    IDLE    UMASK
258  172.21.0.3   172.21.0.2   TCP      66 39730 → 21 [ACK] Seq=51 Ack=742 Win=30336 Len=0 TSval=1019871…
412  172.21.0.3   172.21.0.2   FTP      110 Request: CLNT NcFTPGet 3.2.5 linux-x86_64-glibc2.13
568  172.21.0.2   172.21.0.3   FTP      140 Response: 500 'CLNT NcFTPGet 3.2.5 linux-x86_64-glibc2.13': c…
780  172.21.0.3   172.21.0.2   FTP      74 Request: TYPE I
109  172.21.0.2   172.21.0.3   FTP      86 Response: 200 Type set to I.
621  172.21.0.3   172.21.0.2   FTP      80 Request: MLST key.txt
778  172.21.0.2   172.21.0.3   FTP      110 Response: 500 'MLST key.txt': command not recognized
001  172.21.0.3   172.21.0.2   FTP      80 Request: SIZE key.txt
196  172.21.0.2   172.21.0.3   FTP      74 Response: 213 49
246  172.21.0.3   172.21.0.2   FTP      80 Request: MDTM key.txt
447  172.21.0.2   172.21.0.3   FTP      86 Response: 213 20180725090145
733  172.21.0.3   172.21.0.2   FTP      72 Request: PASV
943  172.21.0.2   172.21.0.3   FTP      114 Response: 227 Entering Passive Mode (172,21,0,2,134,155)
452  172.21.0.3   172.21.0.2   FTP      80 Request: RETR key.txt
856  172.21.0.2   172.21.0.3   FTP      133 Response: 150 Opening BINARY mode data connection for 'key.tx…
543  172.21.0.3   172.21.0.2   TCP      66 39730 → 21 [ACK] Seq=165 Ack=1023 Win=30336 Len=0 TSval=10198…
570  172.21.0.2   172.21.0.3   FTP      90 Response: 226 Transfer complete.
734  172.21.0.3   172.21.0.2   TCP      66 39730 → 21 [ACK] Seq=165 Ack=1047 Win=30336 Len=0 TSval=10198…
843  172.21.0.3   172.21.0.2   FTP      72 Request: QUIT
014  172.21.0.2   172.21.0.3   FTP      80 Response: 221 Goodbye.
234  172.21.0.3   172.21.0.2   TCP      66 39730 → 21 [FIN, ACK] Seq=171 Ack=1061 Win=30336 Len=0 TSval=…
386  172.21.0.2   172.21.0.3   TCP      66 21 → 39730 [FIN, ACK] Seq=1061 Ack=172 Win=29056 Len=0 TSval=…
431  172.21.0.3   172.21.0.2   TCP      66 39730 → 21 [ACK] Seq=172 Ack=1062 Win=30336 Len=0 TSval=10198…

    [TCP Segment Len: 48]
    Sequence Number: 908    (relative sequence number)
    Sequence Number (raw): 1388942327
    [Next Sequence Number: 956    (relative sequence number)]
    Acknowledgment Number: 151    (relative ack number)
    Acknowledgment number (raw): 2166511030
    1000 .... = Header Length: 32 bytes (8)
  ▸ Flags: 0x018 (PSH, ACK)
    Window: 227
    [Calculated window size: 29056]
    [Window size scaling factor: 128]
    Checksum: 0x5886 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  ▸ Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
  ▸ [Timestamps]
  ▸ [SEQ/ACK analysis]
    TCP payload (48 bytes)
▾ File Transfer Protocol (FTP)
  ▾ 227 Entering Passive Mode (172,21,0,2,134,155)\r\n
      Response code: Entering Passive Mode (227)
      Response arg: Entering Passive Mode (172,21,0,2,134,155)
      Passive IP address: 172.21.0.2
      Passive port: 34459
  [Current working directory: /home/victim]
  [Command: RETR key.txt]
  [Command frame: 41]

0000  02 42 ac 15 00 03
0010  00 64 a3 1d 40 00
0020  00 03 00 15 9b 32
0030  00 e3 58 86 00 00
0040  33 78 32 32 37 20
0050  61 73 73 69 76 65
0060  2c 32 31 2c 30 2c
0070  0d 0a
```

Figure 2: FTP control connection: PASV response with IP and port

Decoded Info:

- IP Address: `172.21.0.2`
- Port: 34459



```
Wireshark · Follow TCP Stream (tcp.stream eq 1) · pcap_04 (1).pcap

The Key is: [redacted]-7b49-44bd-9eac-ac2192cf1db1

0 client pkts, 1 server pkt, 0 turns.
Entire conversation (49 bytes)   Show as ASCII   No delta times   Stream 1
Find:   Case sensitive
```

```
pcap_04 (1).pcap
tcp.stream eq 1

No. Time      Source      Destination Protocol Length Info
 38 0.223180  172.21.0.3  172.21.0.2  TCP      74 57349 → 34459 [SYN] Seq=0 Win=29200 Len=0 MSS
 39 0.223240  172.21.0.2  172.21.0.3  TCP      74 34459 → 57349 [SYN, ACK] Seq=0 Ack=1 Win=2896
 40 0.223314  172.21.0.3  172.21.0.2  TCP      66 57349 → 34459 [ACK] Seq=1 Ack=1 Win=29200 Len
 43 0.224026  172.21.0.2  172.21.0.3  FTP-DA…  115 FTP Data: 49 bytes (PASV) (RETR key.txt)
 44 0.224113  172.21.0.3  172.21.0.2  TCP      66 57349 → 34459 [ACK] Seq=1 Ack=50 Win=29200 Le
 45 0.224207  172.21.0.2  172.21.0.3  TCP      66 34459 → 57349 [FIN, ACK] Seq=50 Ack=1 Win=290
 46 0.224329  172.21.0.3  172.21.0.2  TCP      66 57349 → 34459 [FIN, ACK] Seq=1 Ack=51 Win=292
 47 0.224346  172.21.0.2  172.21.0.3  TCP      66 34459 → 57349 [ACK] Seq=51 Ack=2 Win=29056 Le

▸ Frame 43: 115 bytes on wire (920 bits), 115 bytes captured (920 bits)
▾ Ethernet II, Src: 02:42:ac:15:00:02 (02:42:ac:15:00:02), Dst: 02:42:ac:15:00:03 (02:42:ac:15:00:03)
  ▾ Destination: 02:42:ac:15:00:03 (02:42:ac:15:00:03)
      .... ..1. .... .... .... .... = LG bit: Locally administered address (this is NOT the factory det
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
  ▾ Source: 02:42:ac:15:00:02 (02:42:ac:15:00:02)
      .... ..1. .... .... .... .... = LG bit: Locally administered address (this is NOT the factory det
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Type: IPv4 (0x0800)
    [Stream index: 0]
▾ Internet Protocol Version 4, Src: 172.21.0.2, Dst: 172.21.0.3
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x08 (DSCP: Unknown, ECN: Not-ECT)
    Total Length: 101
    Identification: 0xb093 (45203)
  ▸ 010. .... = Flags: 0x2, Don't fragment
    ...0 0000 0000 0000 = Fragment Offset: 0
    Time to Live: 64
    Protocol: TCP (6)
    Header Checksum: 0x31c8 [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 172.21.0.2
    Destination Address: 172.21.0.3
    [Stream index: 0]
▾ Transmission Control Protocol, Src Port: 34459, Dst Port: 57349, Seq: 1, Ack: 1, Len: 49
    Source Port: 34459

0000  02 42 ac
0010  00 65 b0
0020  00 03 86
0030  00 e3 58
0040  33 78 54
0050  33 37 36
0060  2d 39 65
0070  62 31 0a
```

Figure 3: FTP data stream: Key content received from the file

Analysis

The FTP protocol does not encrypt its data, and even the file transfer session can be easily reconstructed and inspected. Passive mode makes this easier by exposing the IP and port directly to the client — and any observer.

4.3 Challenge 3 – RSH Connection and Retrieved Key

This challenge demonstrates the use of the **RSH (Remote Shell)** protocol, a legacy remote command execution service similar to Telnet. The goal is to analyze the RSH traffic in a PCAP file and extract the file transferred during the session.

What is RSH?

RSH is a protocol that allows a user to execute shell commands on a remote machine without logging in interactively. It relies on a trust-based system using the `.rhosts` file, which lists trusted client IP addresses or usernames.

Security Issue: RSH is extremely insecure:

- It sends all commands and output in plaintext.
- Trust is based only on IP address and username.
- There is no encryption or authentication beyond IP/username matching.

This makes RSH vulnerable to IP spoofing and network sniffing attacks, and it should never be used on modern networks.

Challenge Description

In this challenge, the root user connects to a remote server via RSH. The trust relationship is configured using a `.rhosts` file, allowing the client to access the server without a password.

Objective: Analyze the connection and extract the retrieved file (the key).

Note: This challenge is intentionally simple and designed to familiarize you with RSH's risks and packet visibility.

Steps

1. Open `pcap_06.pcap` in Wireshark.
2. Use filter: `tcp.stream eq 0` (or browse TCP streams to find the one containing RSH).
3. Right-click a packet and choose **"Follow TCP Stream"**.
4. The full plaintext connection is displayed, including the file retrieved.

erciement

Analysis

This challenge emphasizes how RSH exposes all activity on the network. The attacker or analyst doesn't need any credentials — only access to the PCAP file. Even root-level operations are visible to anyone capturing traffic.

4.4 Challenge 4 – Rlogin Connection and Password Recovery

This challenge involves analyzing an insecure remote login session using the **rlogin** protocol. Your task is to inspect the connection and retrieve the password sent over the network.

What is Rlogin?

Rlogin (Remote Login) is a legacy Unix protocol used to log into another host over a TCP/IP network. Like Telnet and RSH, it is insecure by design:

- The login process transmits the username and password in plaintext.
- It can be used with `.rhosts` trust relationships.
- There is no encryption or protection against sniffing or spoofing.

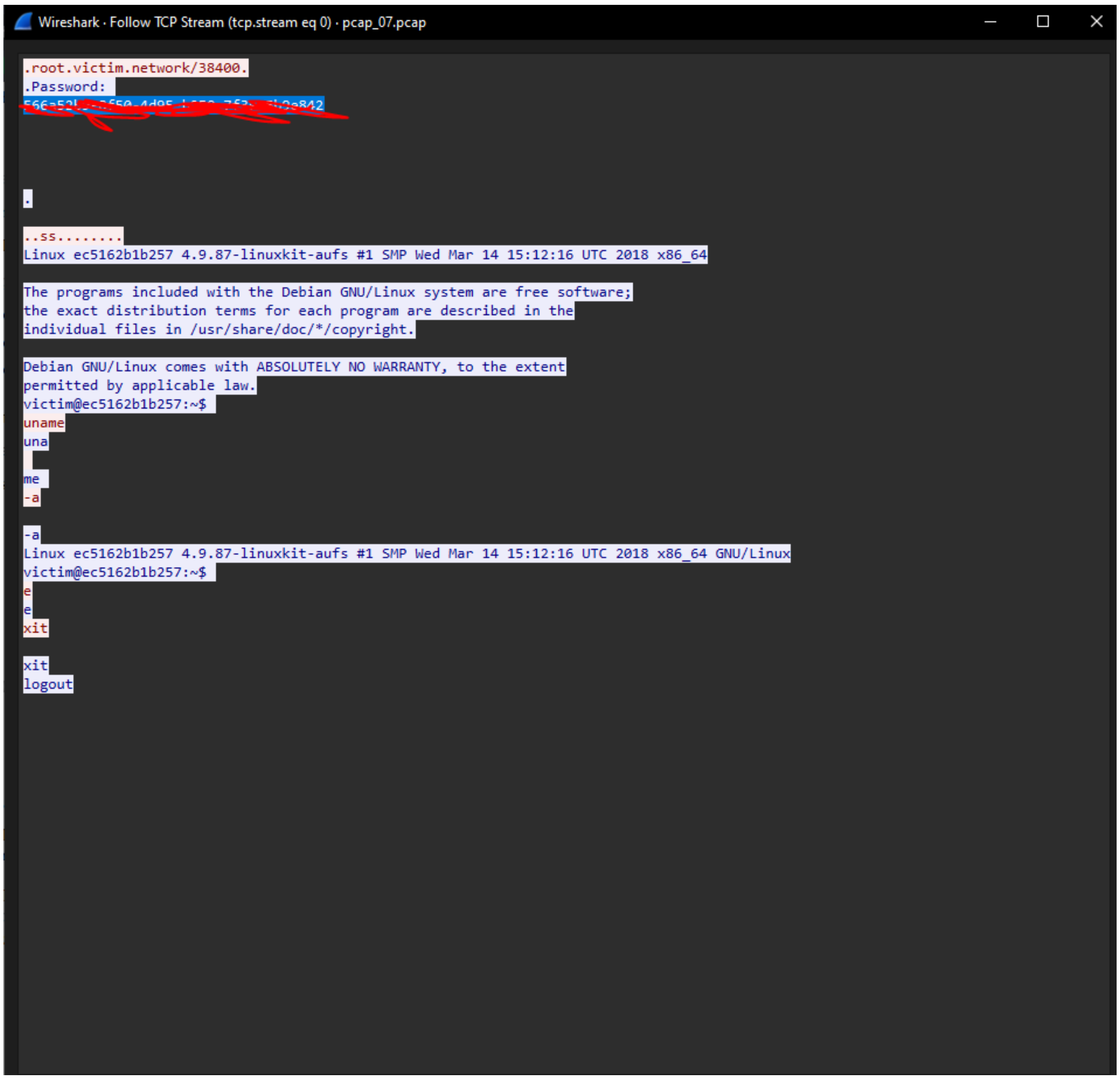
Because of these flaws, rlogin has been deprecated and replaced by secure alternatives like **SSH**.

Challenge Description

In this PCAP challenge, a user connects to a remote server using rlogin. Your objective is to extract the **username** and **password** from the packet stream — the password is the key that solves this exercise.

Steps

1. Open `pcap_07.pcap` in Wireshark.
2. Identify the correct TCP stream using `tcp.stream eq 0`, or browse available streams.
3. Right-click a packet and select **"Follow TCP Stream"**.
4. Inspect the beginning of the stream — you will find:
    - The client username
    - The remote username
    - The password sent in cleartext


Wireshark · Follow TCP Stream (tcp.stream eq 0) · pcap_07.pcap

```
.root.victim.network/38400.
.Password:
[redacted]




.

..ss.......
Linux ec5162b1b257 4.9.87-linuxkit-aufs #1 SMP Wed Mar 14 15:12:16 UTC 2018 x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
victim@ec5162b1b257:~$
uname
una

me
-a

-a
Linux ec5162b1b257 4.9.87-linuxkit-aufs #1 SMP Wed Mar 14 15:12:16 UTC 2018 x86_64 GNU/Linux
victim@ec5162b1b257:~$
e
e
xit

xit
logout
```

Figure 4: Rlogin TCP stream showing cleartext credentials

Analysis

This challenge clearly demonstrates the inherent insecurity of rlogin. Anyone with access to network traffic can retrieve not only the login credentials but also the full session content.

4.5 Challenge 5 – SMTP Authentication via Base64 (Email)

This challenge involves analyzing a packet capture of a user sending an email using the **SMTP (Simple Mail Transfer Protocol)**. The authentication process for SMTP often uses base64-encoded credentials, which can be easily decoded if captured over an unencrypted connection.

What is SMTP?

SMTP is the standard protocol used for sending emails across the Internet. While secure versions exist (like SMTPS and STARTTLS), basic SMTP transmits credentials and email contents in plaintext unless encryption is explicitly enabled.

Challenge Description

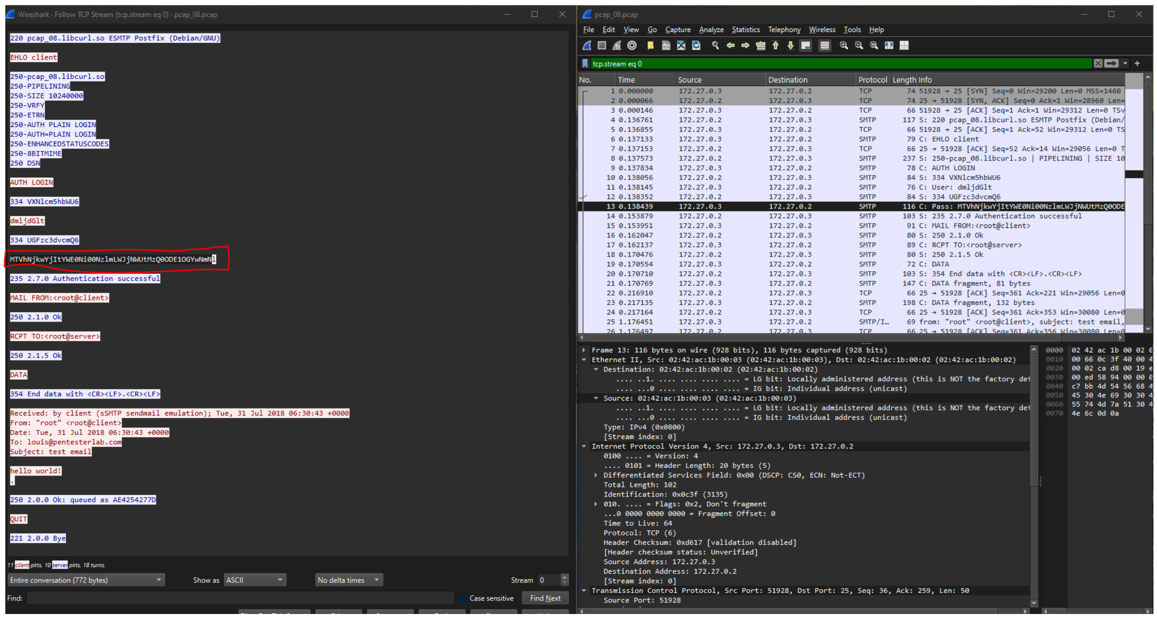
The provided PCAP file contains a captured SMTP session where the client authenticates with the server using the `AUTH LOGIN` method. This method prompts the client to send:

- A base64-encoded username
- A base64-encoded password

The server responds with `334 VXNlcm5hbWU6` and `334 UGFzc3dvcmQ6`, which are the base64-encoded versions of `Username:` and `Password:`, respectively.

Steps

1. Open `pcap_08.pcap` in Wireshark.
2. Use the filter `tcp.stream eq 0` to isolate the SMTP session.
3. Right-click a packet and select **"Follow TCP Stream"**.
4. You will see the following sequence:
   - `AUTH LOGIN`
   - `334 VXNlcm5hbWU6` – base64 for `Username:`
   - `<base64-username>`
   - `334 UGFzc3dvcmQ6` – base64 for `Password:`
   - `<base64-password>`


Wireshark · Follow TCP Stream (tcp.stream eq 0) · pcap_08.pcap
pcap_08.pcap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Show as
Find:
Case sensitive
Find Next

```
220 pcap_08.libcurl.so ESMTP Postfix (Debian/GNU)

EHLO client

250-pcap_08.libcurl.so
250-PIPELINING
250-SIZE 10240000
250-VRFY
250-ETRN
250-AUTH PLAIN LOGIN
250-AUTH=PLAIN LOGIN
250-ENHANCEDSTATUSCODES
250-8BITMIME
250 DSN

AUTH LOGIN

334 VXNlcm5hbWU6

dmljdGlt

334 UGFzc3dvcmQ6

MTVhNjkwYjItYWE0Ni00NzlmLWJjNWUtMzQ0ODE1OGYwNmNl

235 2.7.0 Authentication successful

MAIL FROM:<root@client>

250 2.1.0 Ok

RCPT TO:<root@server>

250 2.1.5 Ok

DATA

354 End data with <CR><LF>.<CR><LF>

Received: by client (sSMTP sendmail emulation); Tue, 31 Jul 2018 06:30:43 +0000
From: "root" <root@client>
Date: Tue, 31 Jul 2018 06:30:43 +0000
To: louis@pentesterlab.com
Subject: test email

hello world!
.

250 2.0.0 Ok: queued as AE4254277D

QUIT

221 2.0.0 Bye
```

11 client pkts, 10 server pkts, 18 turns.

Entire conversation (772 bytes) Show as ASCII No delta times Stream 0

tcp.stream eq 0

```
No.  Time      Source      Destination  Protocol Length Info
  1 0.000000   172.27.0.3  172.27.0.2   TCP        74 51928 → 25 [SYN] Seq=0 Win=29200 Len=0 MSS=1460
  2 0.000066   172.27.0.2  172.27.0.3   TCP        74 25 → 51928 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=
  3 0.000146   172.27.0.3  172.27.0.2   TCP        66 51928 → 25 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSv
  4 0.136761   172.27.0.2  172.27.0.3   SMTP      117 S: 220 pcap_08.libcurl.so ESMTP Postfix (Debian/
  5 0.136855   172.27.0.3  172.27.0.2   TCP        66 51928 → 25 [ACK] Seq=1 Ack=52 Win=29312 Len=0 TS
  6 0.137133   172.27.0.3  172.27.0.2   SMTP       79 C: EHLO client
  7 0.137153   172.27.0.2  172.27.0.3   TCP        66 25 → 51928 [ACK] Seq=52 Ack=14 Win=29056 Len=0 T
  8 0.137573   172.27.0.2  172.27.0.3   SMTP      237 S: 250-pcap_08.libcurl.so | PIPELINING | SIZE 10
  9 0.137834   172.27.0.3  172.27.0.2   SMTP       78 C: AUTH LOGIN
 10 0.138056   172.27.0.2  172.27.0.3   SMTP       84 S: 334 VXNlcm5hbWU6
 11 0.138145   172.27.0.3  172.27.0.2   SMTP       76 C: User: dmljdGlt
 12 0.138352   172.27.0.2  172.27.0.3   SMTP       84 S: 334 UGFzc3dvcmQ6
 13 0.138439   172.27.0.3  172.27.0.2   SMTP      116 C: Pass: MTVhNjkwYjItYWE0Ni00NzlmLWJjNWUtMzQ0ODE
 14 0.153879   172.27.0.2  172.27.0.3   SMTP      103 S: 235 2.7.0 Authentication successful
 15 0.153951   172.27.0.3  172.27.0.2   SMTP       91 C: MAIL FROM:<root@client>
 16 0.162047   172.27.0.2  172.27.0.3   SMTP       80 S: 250 2.1.0 Ok
 17 0.162137   172.27.0.3  172.27.0.2   SMTP       89 C: RCPT TO:<root@server>
 18 0.170476   172.27.0.2  172.27.0.3   SMTP       80 S: 250 2.1.5 Ok
 19 0.170554   172.27.0.3  172.27.0.2   SMTP       72 C: DATA
 20 0.170710   172.27.0.2  172.27.0.3   SMTP      103 S: 354 End data with <CR><LF>.<CR><LF>
 21 0.170769   172.27.0.3  172.27.0.2   SMTP      147 C: DATA fragment, 81 bytes
 22 0.216910   172.27.0.2  172.27.0.3   TCP        66 25 → 51928 [ACK] Seq=361 Ack=221 Win=29056 Len=0
 23 0.217135   172.27.0.3  172.27.0.2   SMTP      198 C: DATA fragment, 132 bytes
 24 0.217164   172.27.0.2  172.27.0.3   TCP        66 25 → 51928 [ACK] Seq=361 Ack=353 Win=30080 Len=0
 25 1.176451   172.27.0.3  172.27.0.2   SMTP/I…    69 from: "root" <root@client>, subject: test email,
 26 1.176497   172.27.0.2  172.27.0.3   TCP        66 25 → 51928 [ACK] Seq=361 Ack=356 Win=30080 Len=0
```

```
▸ Frame 13: 116 bytes on wire (928 bits), 116 bytes captured (928 bits)
▾ Ethernet II, Src: 02:42:ac:1b:00:03 (02:42:ac:1b:00:03), Dst: 02:42:ac:1b:00:02 (02:42:ac:1b:00:02)
  ▾ Destination: 02:42:ac:1b:00:02 (02:42:ac:1b:00:02)
      .... ..1. .... .... .... .... = LG bit: Locally administered address (this is NOT the factory def
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
  ▾ Source: 02:42:ac:1b:00:03 (02:42:ac:1b:00:03)
      .... ..1. .... .... .... .... = LG bit: Locally administered address (this is NOT the factory def
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Type: IPv4 (0x0800)
    [Stream index: 0]
▾ Internet Protocol Version 4, Src: 172.27.0.3, Dst: 172.27.0.2
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 102
    Identification: 0x0c3f (3135)
  ▸ 010. .... = Flags: 0x2, Don't fragment
    ...0 0000 0000 0000 = Fragment Offset: 0
    Time to Live: 64
    Protocol: TCP (6)
    Header Checksum: 0xd617 [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 172.27.0.3
    Destination Address: 172.27.0.2
    [Stream index: 0]
▾ Transmission Control Protocol, Src Port: 51928, Dst Port: 25, Seq: 36, Ack: 259, Len: 50
    Source Port: 51928
```

```
0000  02 42 ac 1b 00 02 0
0010  00 66 0c 3f 40 00 4
0020  00 02 ca d8 00 19 e
0030  00 ed 58 94 00 00 0
0040  c7 bb 4d 54 56 68 4
0050  45 30 4e 69 30 30 4
0060  55 74 4d 7a 51 30 4
0070  4e 6c 0d 0a
```

Figure 5: SMTP authentication with base64-encoded credentials

Analysis

The captured exchange shows the credentials being transmitted in an easily reversible format (base64). Even though base64 encodes the data, it does not provide any encryption or protection — anyone with access to the packet capture can decode the credentials.

This highlights the importance of using secure versions of SMTP (e.g., with STARTTLS or SMTPS), which encrypt the communication between the client and server to prevent credential leakage.

Note: This challenge does not require decoding the credentials to understand the vulnerability. Simply observing the base64-encoded login process is enough to recognize the risk.

4.6 Challenge 6 – SMTP Email Recipient Extraction

This challenge focuses on inspecting an SMTP session to extract the recipient of an email. The captured communication contains the full process of sending an email to an external address.

Objective

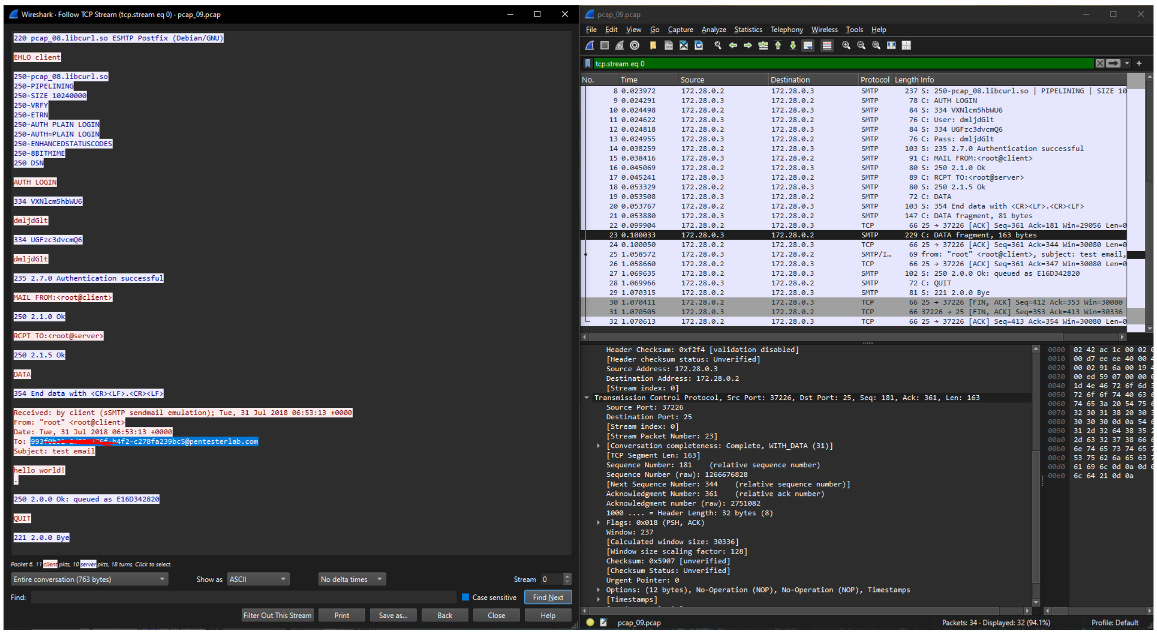
Identify the recipient of an email sent to a `@pentesterlab.com` address. The local part of this email address (i.e., everything before the `@`) is the key to solving this challenge.

Challenge Description

You are provided with a PCAP file that captures a full SMTP session where an email is transmitted from a client to a remote server. The email is sent using plaintext SMTP, meaning the message content and headers — including the recipient — are visible in the captured packets.

Steps

1. Open `pcap_09.pcap` in Wireshark.
2. Use the display filter: `tcp.stream eq 0` to isolate the email session.
3. Right-click a packet and select **"Follow TCP Stream"**.
4. In the stream view, locate the following:
   - 'MAIL FROM:' and 'RCPT TO:' headers
   - The full email being sent, including 'To:' and 'Subject:' headers
5. Identify the recipient email (e.g., `xyz@pentesterlab.com`) and extract the part before the `@` symbol.



```
Wireshark · Follow TCP Stream (tcp.stream eq 0) · pcap_09.pcap

220 pcap_08.libcurl.so ESMTP Postfix (Debian/GNU)

EHLO client

250-pcap_08.libcurl.so
250-PIPELINING
250-SIZE 10240000
250-VRFY
250-ETRN
250-AUTH PLAIN LOGIN
250-AUTH=PLAIN LOGIN
250-ENHANCEDSTATUSCODES
250-8BITMIME
250 DSN

AUTH LOGIN

334 VXNlcm5hbWU6

dmljdGlt

334 UGFzc3dvcmQ6

dmljdGlt

235 2.7.0 Authentication successful

MAIL FROM:<root@client>

250 2.1.0 Ok

RCPT TO:<root@server>

250 2.1.5 Ok

DATA

354 End data with <CR><LF>.<CR><LF>

Received: by client (sSMTP sendmail emulation); Tue, 31 Jul 2018 06:53:13 +0000
From: "root" <root@client>
Date: Tue, 31 Jul 2018 06:53:13 +0000
To: 993f[redacted]b4f2-c278fa239bc5@pentesterlab.com
Subject: test email

hello world!
.

250 2.0.0 Ok: queued as E16D342820

QUIT

221 2.0.0 Bye

Packet 6. 11 client pkts, 10 server pkts, 18 turns. Click to select.
Entire conversation (763 bytes)    Show as ASCII    No delta times    Stream 0
Find:    Case sensitive
```

```
pcap_09.pcap
tcp.stream eq 0

No.  Time     Source      Destination Protocol Length Info
 8 0.023972 172.28.0.2  172.28.0.3  SMTP     237 S: 250-pcap_08.libcurl.so | PIPELINING | SIZE 10
 9 0.024291 172.28.0.3  172.28.0.2  SMTP      78 C: AUTH LOGIN
10 0.024498 172.28.0.2  172.28.0.3  SMTP      84 S: 334 VXNlcm5hbWU6
11 0.024622 172.28.0.3  172.28.0.2  SMTP      76 C: User: dmljdGlt
12 0.024818 172.28.0.2  172.28.0.3  SMTP      84 S: 334 UGFzc3dvcmQ6
13 0.024955 172.28.0.3  172.28.0.2  SMTP      76 C: Pass: dmljdGlt
14 0.038259 172.28.0.2  172.28.0.3  SMTP     103 S: 235 2.7.0 Authentication successful
15 0.038416 172.28.0.3  172.28.0.2  SMTP      91 C: MAIL FROM:<root@client>
16 0.045069 172.28.0.2  172.28.0.3  SMTP      80 S: 250 2.1.0 Ok
17 0.045241 172.28.0.3  172.28.0.2  SMTP      89 C: RCPT TO:<root@server>
18 0.053329 172.28.0.2  172.28.0.3  SMTP      80 S: 250 2.1.5 Ok
19 0.053508 172.28.0.3  172.28.0.2  SMTP      72 C: DATA
20 0.053767 172.28.0.2  172.28.0.3  SMTP     103 S: 354 End data with <CR><LF>.<CR><LF>
21 0.053880 172.28.0.3  172.28.0.2  SMTP     147 C: DATA fragment, 81 bytes
22 0.099904 172.28.0.2  172.28.0.3  TCP       66 25 → 37226 [ACK] Seq=361 Ack=181 Win=29056 Len=0
23 0.100033 172.28.0.3  172.28.0.2  SMTP     229 C: DATA fragment, 163 bytes
24 0.100050 172.28.0.2  172.28.0.3  TCP       66 25 → 37226 [ACK] Seq=361 Ack=344 Win=30080 Len=0
25 1.058572 172.28.0.3  172.28.0.2  SMTP/I…   69 from: "root" <root@client>, subject: test email,
26 1.058660 172.28.0.2  172.28.0.3  TCP       66 25 → 37226 [ACK] Seq=361 Ack=347 Win=30080 Len=0
27 1.069635 172.28.0.2  172.28.0.3  SMTP     102 S: 250 2.0.0 Ok: queued as E16D342820
28 1.069966 172.28.0.3  172.28.0.2  SMTP      72 C: QUIT
29 1.070315 172.28.0.2  172.28.0.3  SMTP      81 S: 221 2.0.0 Bye
30 1.070411 172.28.0.2  172.28.0.3  TCP       66 25 → 37226 [FIN, ACK] Seq=412 Ack=353 Win=30080
31 1.070505 172.28.0.3  172.28.0.2  TCP       66 37226 → 25 [FIN, ACK] Seq=353 Ack=413 Win=30336
32 1.070613 172.28.0.2  172.28.0.3  TCP       66 25 → 37226 [ACK] Seq=413 Ack=354 Win=30080 Len=0

    Header Checksum: 0xf2f4 [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 172.28.0.3
    Destination Address: 172.28.0.2
    [Stream index: 0]
  Transmission Control Protocol, Src Port: 37226, Dst Port: 25, Seq: 181, Ack: 361, Len: 163
    Source Port: 37226
    Destination Port: 25
    [Stream index: 0]
    [Stream Packet Number: 23]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 163]
    Sequence Number: 181    (relative sequence number)
    Sequence Number (raw): 1266676828
    [Next Sequence Number: 344    (relative sequence number)]
    Acknowledgment Number: 361    (relative ack number)
    Acknowledgment number (raw): 2751082
    1000 .... = Header Length: 32 bytes (8)
    Flags: 0x018 (PSH, ACK)
    Window: 237
    [Calculated window size: 30336]
    [Window size scaling factor: 128]
    Checksum: 0x5907 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
    Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
    [Timestamps]

0000  02 42 ac 1c 00 02
0010  00 d7 ee ee 40 00
0020  00 02 91 6a 00 19
0030  00 ed 59 07 00 00
0040  1d 4e 46 72 6f 6d
0050  72 6f 6f 74 40 63
0060  74 65 3a 20 54 75
0070  32 30 31 38 20 30
0080  30 30 30 0d 0a 54
0090  31 2d 32 64 38 35
00a0  2d 63 32 37 38 66
00b0  6e 74 65 73 74 65
00c0  53 75 62 6a 65 63
00d0  61 69 6c 0d 0a 0d
00e0  6c 64 21 0d 0a

pcap_09.pcap    Packets: 34 · Displayed: 32 (94.1%)    Profile: Default
```

Figure 6: SMTP stream showing plaintext email with recipient

Analysis

Since SMTP traffic in this scenario is unencrypted, both the sender and recipient addresses, as well as the full email content, are exposed in plaintext. This means anyone monitoring the network or analyzing a PCAP can extract sensitive metadata and potentially private communication content.

The challenge is a simple reminder of the importance of encrypting email traffic using secure alternatives such as:

- SMTPS (SMTP over TLS/SSL)
- STARTTLS (upgraded encrypted SMTP connection)

Important: In this exercise, you do not need to decode anything. The key is directly visible in the recipient email address — just remove `@pentesterlab.com` and submit the remaining part.

4.7 Challenge 7 – SMTP Email with Attachment (UUEncoded Zip)

This challenge demonstrates how attachments can be transferred over SMTP using a legacy encoding mechanism: **UUEncode**. The objective is to retrieve an email that includes a file attachment encoded in the body of the message and understand how it could be extracted.

Challenge Description

You are provided with a PCAP file that captures the entire process of sending an email via SMTP. The email contains an attached ZIP file encoded using the traditional `uuencode` format — a common method for sending binary files over text-based protocols.

Objective

Observe the email content in the SMTP stream and extract the block starting with:

```
begin 644 the_key.zip
...
end
```

This block contains a `uuencoded` ZIP file. The final goal is to extract the file and decompress it using:

- `uudecode` to decode the UUencoded block into a binary '.zip' file.
- `unzip` to extract its contents and recover the key.

Steps

1. Open `pcap_10.pcap` in Wireshark.
2. Use the filter: `tcp.stream eq 0` to isolate the SMTP session.
3. Right-click a packet and select **"Follow TCP Stream"**.
4. Locate the section of the email body that includes:
   - 'begin 644 the$_key.zip$'$Severallinesof UUencodedcontent$

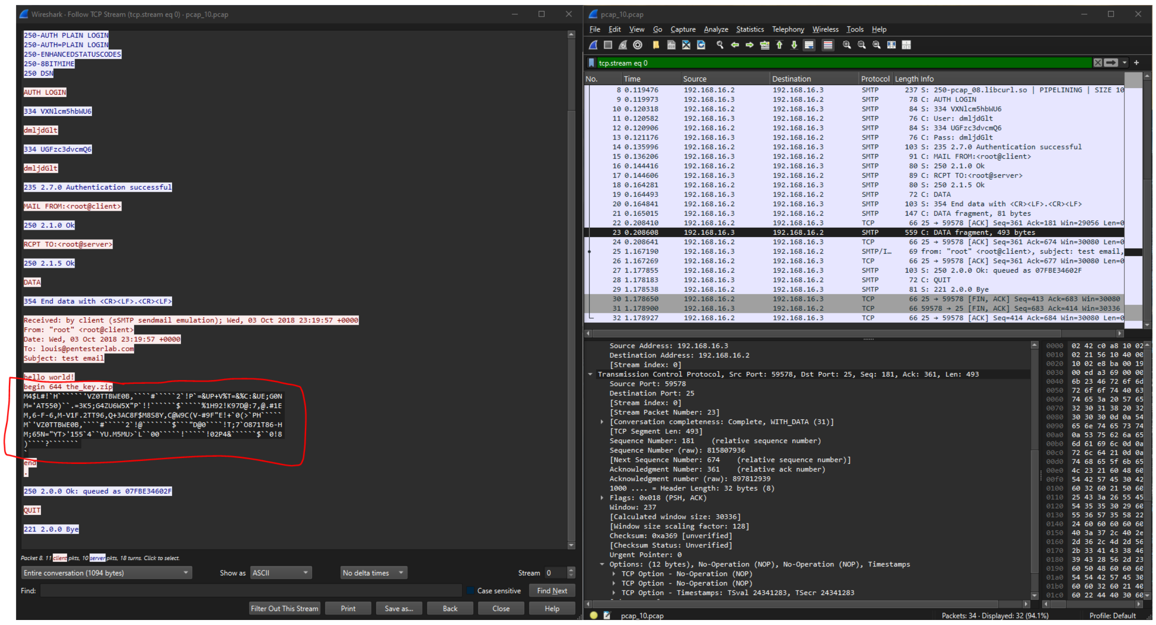
- Ending with 'end'

5. Copy this block to a file (e.g., `key.uue`).
6. Use a tool like `uudecode` to decode the file:

   ```
   uudecode key.uue
   ```

7. Then extract the archive:

   ```
   unzip the_key.zip
   ```



```
250-AUTH PLAIN LOGIN
250-AUTH=PLAIN LOGIN
250-ENHANCEDSTATUSCODES
250-8BITMIME
250 DSN

AUTH LOGIN

334 VXNlcm5hbWU6

dmljdGlt

334 UGFzc3dvcmQ6

dmljdGlt

235 2.7.0 Authentication successful

MAIL FROM:<root@client>

250 2.1.0 Ok

RCPT TO:<root@server>

250 2.1.5 Ok

DATA

354 End data with <CR><LF>.<CR><LF>

Received: by client (sSMTP sendmail emulation); Wed, 03 Oct 2018 23:19:57 +0000
From: "root" <root@client>
Date: Wed, 03 Oct 2018 23:19:57 +0000
To: louis@pentesterlab.com
Subject: test email

hello world!
begin 644 the_key.zip
M4$L#!`H``````'VZ0TTBWE0B,````#``````2`!P`=&UP+V%T=&%%C:&UE;G0N
M='AT550)``.=3K5;G4ZU6W5X"P`!!``````$``````%1H92!K97D@:7,@.#1E
M,6-F-6,M-V1F.2TT96,Q+3AC8F$M8S8Y,C@W9C(V-#9F"E!+`0(>`PH`````
M`'VZ0TTBWE0B,````#``````2`!@```````$````D@0````!T;7`O871T86-H
M;65N="YT>'155`4``YU.M5MU>`L``00``````!``````!02P4&``````$``0!8
)````?```````
`
end
.

250 2.0.0 Ok: queued as 07FBE34602F

QUIT

221 2.0.0 Bye
```

Packet 8. 11 client pkts, 10 server pkts, 18 turns. Click to select.

Entire conversation (1094 bytes)

```
No. Time      Source        Destination   Protocol Length Info
  8 0.119476  192.168.16.2  192.168.16.3  SMTP     237 S: 250-pcap_08.libcurl.so | PIPELINING | SIZE 10
  9 0.119973  192.168.16.3  192.168.16.2  SMTP      78 C: AUTH LOGIN
 10 0.120318  192.168.16.2  192.168.16.3  SMTP      84 S: 334 VXNlcm5hbWU6
 11 0.120582  192.168.16.3  192.168.16.2  SMTP      76 C: User: dmljdGlt
 12 0.120906  192.168.16.2  192.168.16.3  SMTP      84 S: 334 UGFzc3dvcmQ6
 13 0.121176  192.168.16.3  192.168.16.2  SMTP      76 C: Pass: dmljdGlt
 14 0.135996  192.168.16.2  192.168.16.3  SMTP     103 S: 235 2.7.0 Authentication successful
 15 0.136206  192.168.16.3  192.168.16.2  SMTP      91 C: MAIL FROM:<root@client>
 16 0.144416  192.168.16.2  192.168.16.3  SMTP      80 S: 250 2.1.0 Ok
 17 0.144606  192.168.16.3  192.168.16.2  SMTP      89 C: RCPT TO:<root@server>
 18 0.164281  192.168.16.2  192.168.16.3  SMTP      80 S: 250 2.1.5 Ok
 19 0.164493  192.168.16.3  192.168.16.2  SMTP      72 C: DATA
 20 0.164841  192.168.16.2  192.168.16.3  SMTP     103 S: 354 End data with <CR><LF>.<CR><LF>
 21 0.165015  192.168.16.3  192.168.16.2  SMTP     147 C: DATA fragment, 81 bytes
 22 0.208410  192.168.16.2  192.168.16.3  TCP       66 25 → 59578 [ACK] Seq=361 Ack=181 Win=29056 Len=0
 23 0.208608  192.168.16.3  192.168.16.2  SMTP     559 C: DATA fragment, 493 bytes
 24 0.208641  192.168.16.2  192.168.16.3  TCP       66 25 → 59578 [ACK] Seq=361 Ack=674 Win=30080 Len=0
 25 1.167190  192.168.16.3  192.168.16.2  SMTP/I…   69 from: "root" <root@client>, subject: test email,
 26 1.167269  192.168.16.2  192.168.16.3  TCP       66 25 → 59578 [ACK] Seq=361 Ack=677 Win=30080 Len=0
 27 1.177855  192.168.16.2  192.168.16.3  SMTP     103 S: 250 2.0.0 Ok: queued as 07FBE34602F
 28 1.178183  192.168.16.3  192.168.16.2  SMTP      72 C: QUIT
 29 1.178538  192.168.16.2  192.168.16.3  SMTP      81 S: 221 2.0.0 Bye
 30 1.178650  192.168.16.2  192.168.16.3  TCP       66 25 → 59578 [FIN, ACK] Seq=413 Ack=683 Win=30080
 31 1.178900  192.168.16.3  192.168.16.2  TCP       66 59578 → 25 [FIN, ACK] Seq=683 Ack=414 Win=30336
 32 1.178927  192.168.16.2  192.168.16.3  TCP       66 25 → 59578 [ACK] Seq=414 Ack=684 Win=30080 Len=0
```

```
    Source Address: 192.168.16.3
    Destination Address: 192.168.16.2
    [Stream index: 0]
  Transmission Control Protocol, Src Port: 59578, Dst Port: 25, Seq: 181, Ack: 361, Len: 493
    Source Port: 59578
    Destination Port: 25
    [Stream index: 0]
    [Stream Packet Number: 23]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 493]
    Sequence Number: 181    (relative sequence number)
    Sequence Number (raw): 815807936
    [Next Sequence Number: 674    (relative sequence number)]
    Acknowledgment Number: 361    (relative ack number)
    Acknowledgment number (raw): 897812939
    1000 .... = Header Length: 32 bytes (8)
    Flags: 0x018 (PSH, ACK)
    Window: 237
    [Calculated window size: 30336]
    [Window size scaling factor: 128]
    Checksum: 0xa369 [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
    Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
      TCP Option - No-Operation (NOP)
      TCP Option - No-Operation (NOP)
      TCP Option - Timestamps: TSval 24341283, TSecr 24341283
```

```
0000  02 42 c0 a8 10 02
0010  02 21 56 10 40 00
0020  10 02 e8 ba 00 19
0030  00 ed a3 69 00 00
0040  6b 23 46 72 6f 6d
0050  72 6f 6f 74 40 63
0060  74 65 3a 20 57 65
0070  32 30 31 38 20 32
0080  30 30 30 0d 0a 54
0090  65 6e 74 65 73 74
00a0  0a 53 75 62 6a 65
00b0  6d 61 69 6c 0d 0a
00c0  72 6c 64 21 0d 0a
00d0  74 68 65 5f 6b 65
00e0  4c 23 21 60 48 60
00f0  54 42 57 45 30 42
0100  60 32 60 21 50 60
0110  25 43 3a 26 55 45
0120  54 35 35 30 29 60
0130  55 36 57 35 58 22
0140  24 60 60 60 60 60
0150  40 3a 37 2c 40 2e
0160  2d 36 2c 4d 2d 56
0170  2b 33 41 43 38 46
0180  39 43 28 56 2d 23
0190  60 50 48 60 60 60
01a0  54 54 42 57 45 30
01b0  60 60 32 60 21 40
01c0  60 22 44 40 30 60
```

pcap_10.pcap   Packets: 34 · Displayed: 32 (94.1%)

Figure 7: SMTP stream showing email body with uuencoded ZIP attachment

Analysis

This challenge shows that even binary file attachments, when sent over insecure protocols like SMTP (without TLS), are fully visible in traffic captures. Tools like Wireshark allow complete recovery of email content — including attachments — if no encryption is applied.

Note: This challenge also demonstrates how legacy encoding formats (like UUEncode) are still relevant when dealing with raw traffic analysis, especially in penetration testing and forensic contexts.

Tip: Make sure to extract only the UUEncode block (without leading or trailing headers) to avoid decoding errors.

4.8 Challenge 8 – POP3 Login Credentials in Cleartext

This challenge demonstrates how insecure email retrieval protocols like **POP3 (Post Office Protocol v3)** expose user credentials in plaintext. The goal is to extract the username and password from a captured POP3 session.

What is POP3?

POP3 is a protocol used by email clients to retrieve messages from a mail server. In its default form (on port 110), POP3 transmits both usernames and passwords in plaintext — making it extremely vulnerable to interception unless secured by TLS.

Challenge Description

You are given a PCAP file containing a POP3 session. The email client (user `victim`) logs in to the server using basic authentication.

Objective

Retrieve the POP3 login credentials sent by the client. The password is the key that solves the challenge.

Note: This challenge is intentionally easy and designed to illustrate how visible and extractable email credentials are when unencrypted POP3 is used.

Steps

1. Open `pcap_11.pcap` in Wireshark.
2. Use the display filter: `tcp.stream eq 0` or browse TCP conversations.
3. Right-click a packet and choose **"Follow TCP Stream"**.
4. Look for the lines similar to:

   ```
   USER victim
   PASS [password]
   ```

5. The password string sent by the client is the solution key.

Analysis

This challenge reinforces the importance of encrypting email credentials. Without protection (e.g., via STARTTLS or SSL), any observer can extract user passwords from network traffic.

4.9 Challenge 9 – IMAP Login Credentials in Cleartext

This challenge illustrates how the **IMAP** protocol, when used without encryption, can expose user credentials directly in the captured traffic. The goal is to extract the password sent by the email client during the login phase.

What is IMAP?

IMAP (Internet Message Access Protocol) is a standard protocol used by email clients to retrieve and manage messages from a mail server. Like POP3, if IMAP is used without encryption (e.g., without STARTTLS or SSL), the username and password are transmitted in plaintext and are fully visible to an attacker or analyst capturing the traffic.

Challenge Description

The provided PCAP file captures a user logging into their email account via IMAP. The login command includes the username `victim` and a plaintext password, which serves as the solution key for the challenge.

Objective

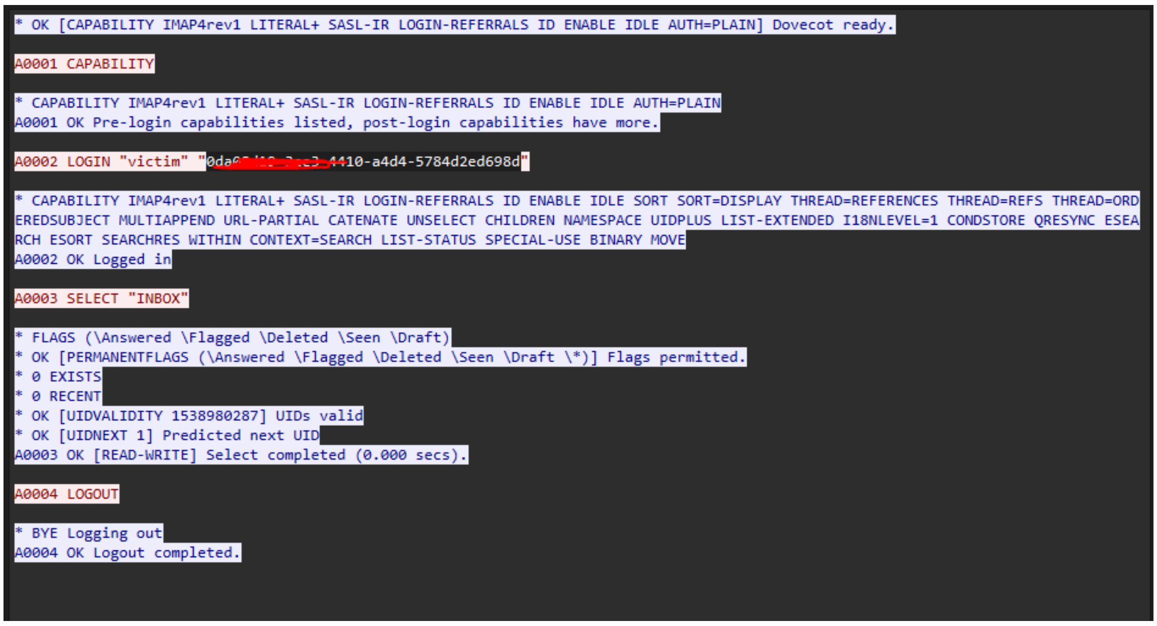
Use Wireshark to extract the plaintext password submitted during the IMAP login.

Steps

1. Open `pcap_12.pcap` in Wireshark.
2. Apply the filter: `tcp.stream eq 0` (or identify the correct TCP stream).
3. Right-click any packet and choose **"Follow TCP Stream"**.
4. Locate the IMAP command:

   ```
   A0002 LOGIN "victim" "[password]"
   ```

5. The password sent in cleartext is the key for this exercise.



```
* OK [CAPABILITY IMAP4rev1 LITERAL+ SASL-IR LOGIN-REFERRALS ID ENABLE IDLE AUTH=PLAIN] Dovecot ready.

A0001 CAPABILITY

* CAPABILITY IMAP4rev1 LITERAL+ SASL-IR LOGIN-REFERRALS ID ENABLE IDLE AUTH=PLAIN
A0001 OK Pre-login capabilities listed, post-login capabilities have more.

A0002 LOGIN "victim" "0da[redacted]4410-a4d4-5784d2ed698d"

* CAPABILITY IMAP4rev1 LITERAL+ SASL-IR LOGIN-REFERRALS ID ENABLE IDLE SORT SORT=DISPLAY THREAD=REFERENCES THREAD=REFS THREAD=ORD
EREDSUBJECT MULTIAPPEND URL-PARTIAL CATENATE UNSELECT CHILDREN NAMESPACE UIDPLUS LIST-EXTENDED I18NLEVEL=1 CONDSTORE QRESYNC ESEA
RCH ESORT SEARCHRES WITHIN CONTEXT=SEARCH LIST-STATUS SPECIAL-USE BINARY MOVE
A0002 OK Logged in

A0003 SELECT "INBOX"

* FLAGS (\Answered \Flagged \Deleted \Seen \Draft)
* OK [PERMANENTFLAGS (\Answered \Flagged \Deleted \Seen \Draft \*)] Flags permitted.
* 0 EXISTS
* 0 RECENT
* OK [UIDVALIDITY 1538980287] UIDs valid
* OK [UIDNEXT 1] Predicted next UID
A0003 OK [READ-WRITE] Select completed (0.000 secs).

A0004 LOGOUT

* BYE Logging out
A0004 OK Logout completed.
```

Figure 8: IMAP login command showing plaintext credentials in captured traffic

Analysis

This challenge once again highlights the dangers of using unencrypted legacy protocols. If IMAP is not secured (e.g., via STARTTLS or SSL/TLS), anyone with access to the network traffic can extract user credentials with minimal effort.

4.10 Challenge 10 – HTTP GET Parameter Leakage

This challenge demonstrates how sensitive data passed in HTTP GET parameters can be intercepted and exposed in plaintext when transmitted over unencrypted HTTP connections.

Challenge Description

The PCAP file provided contains a single HTTP request made using the `curl` command-line tool. The request includes a GET parameter named `key` that contains the solution to the challenge. The goal is to inspect this request and extract the key from the URL.

Objective

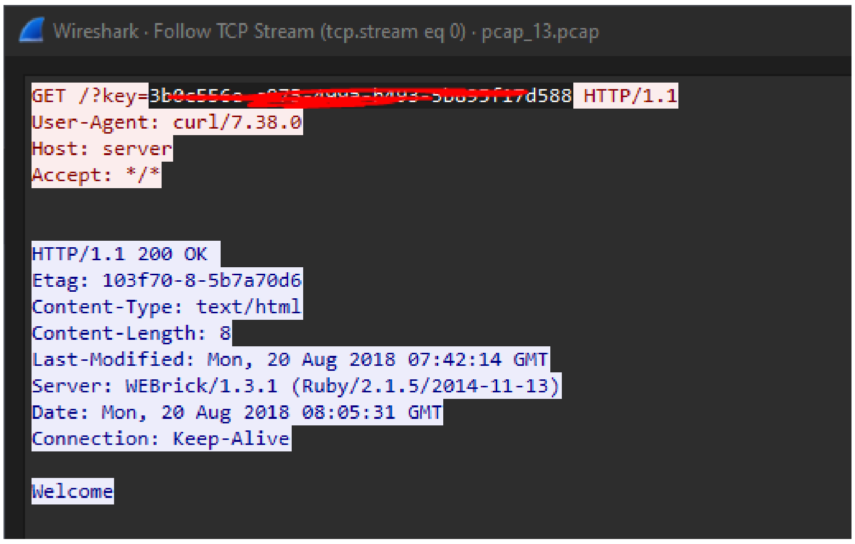
Locate the HTTP GET request and identify the value of the `key` parameter. This value is the flag or solution for the challenge.

Steps

1. Open `pcap_13.pcap` in Wireshark.
2. Use the display filter `tcp.stream eq 0` to isolate the TCP stream.
3. Right-click a packet and select **”Follow TCP Stream”**.
4. Review the full HTTP request shown in ASCII.
5. Locate the line similar to:

   ```
   GET /?key=3b0c55cc... HTTP/1.1
   ```

6. The value of the `key` parameter is the solution.


Wireshark · Follow TCP Stream (tcp.stream eq 0) · pcap_13.pcap

```
GET /?key=[redacted]d588 HTTP/1.1
User-Agent: curl/7.38.0
Host: server
Accept: */*


HTTP/1.1 200 OK
Etag: 103f70-8-5b7a70d6
Content-Type: text/html
Content-Length: 8
Last-Modified: Mon, 20 Aug 2018 07:42:14 GMT
Server: WEBrick/1.3.1 (Ruby/2.1.5/2014-11-13)
Date: Mon, 20 Aug 2018 08:05:31 GMT
Connection: Keep-Alive

Welcome
```

Figure 9: GET request with key exposed in plaintext over HTTP

Analysis

This challenge highlights a fundamental issue in insecure web applications: placing sensitive data (like tokens, passwords, or API keys) in the URL. When HTTP is used without encryption (no HTTPS), the entire request — including headers, URL, and query parameters — is transmitted in plaintext.

Security Recommendations:

- Avoid sending sensitive data in GET parameters.
- Always use HTTPS to encrypt HTTP traffic.
- Use POST requests for confidential data when possible.

4.11 Challenge 11 – HTTP POST Parameter Exposure

This challenge focuses on analyzing an HTTP request where a secret key is sent in the body of a POST request. The capture shows how data submitted through POST can also be intercepted in plaintext if no encryption is used.

Challenge Description

The PCAP file provided captures a single HTTP connection initiated using the `curl` command-line tool. In this session, a secret key is submitted via a `POST` request. The goal is to locate the value of the parameter named `key` in the request body.

Objective

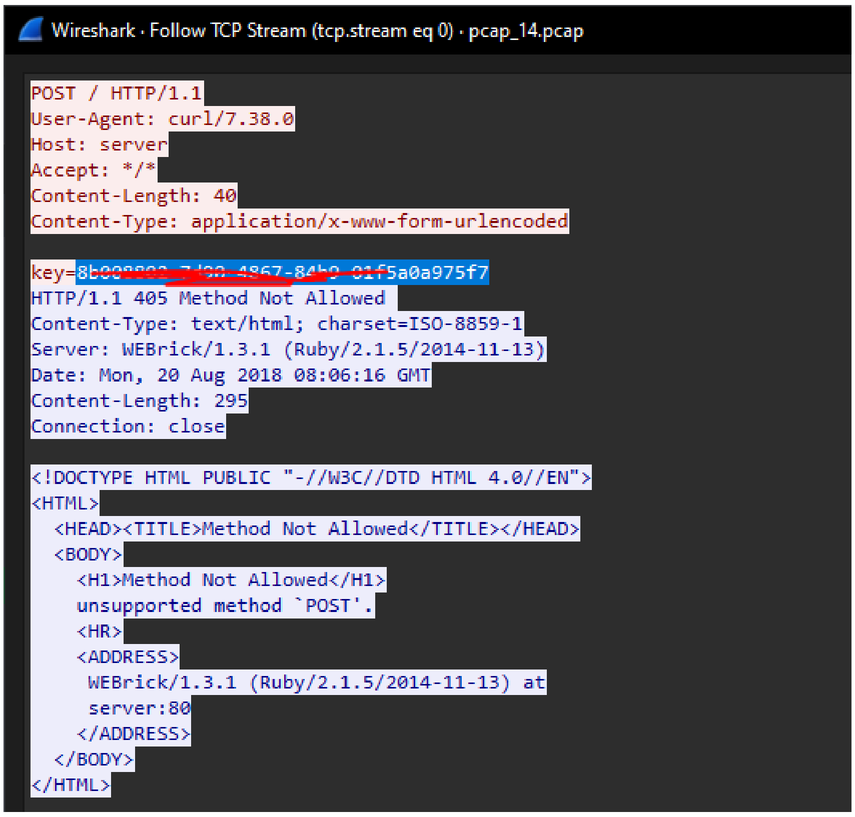
Extract the value of the POST parameter `key` from the HTTP request body. This value serves as the key for solving the challenge.

Steps

1. Open `pcap_14.pcap` in Wireshark.
2. Use the display filter `tcp.stream eq 0` to isolate the connection.
3. Right-click any packet and choose "**Follow TCP Stream**".
4. Observe the POST request with the following structure:

```
POST / HTTP/1.1
...
Content-Length: 40
Content-Type: application/x-www-form-urlencoded

key=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

5. The string after `key=` is the value you are asked to extract.



Wireshark · Follow TCP Stream (tcp.stream eq 0) · pcap_14.pcap

```
POST / HTTP/1.1
User-Agent: curl/7.38.0
Host: server
Accept: */*
Content-Length: 40
Content-Type: application/x-www-form-urlencoded

key=[redacted]5a0a975f7
HTTP/1.1 405 Method Not Allowed
Content-Type: text/html; charset=ISO-8859-1
Server: WEBrick/1.3.1 (Ruby/2.1.5/2014-11-13)
Date: Mon, 20 Aug 2018 08:06:16 GMT
Content-Length: 295
Connection: close

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
  <HEAD><TITLE>Method Not Allowed</TITLE></HEAD>
  <BODY>
    <H1>Method Not Allowed</H1>
    unsupported method `POST'.
    <HR>
    <ADDRESS>
     WEBrick/1.3.1 (Ruby/2.1.5/2014-11-13) at
     server:80
    </ADDRESS>
  </BODY>
</HTML>
```

Figure 10: HTTP POST request with key submitted in the request body

Interesting Technical Note

The HTTP header `Content-Length: 40` informs the server of the exact size of the request body (40 bytes). This is essential for the server to know how much data to read from the TCP socket, especially when no terminating boundary is used (unlike multipart forms or chunked transfer encoding).

Security Considerations

Although POST is commonly preferred over GET for submitting sensitive data (as it avoids exposing it in the URL), the content is still visible to anyone with access to the network traffic if HTTPS is not used.

Conclusion: This challenge emphasizes that POST does not equal secure — encryption (HTTPS) must still be enforced to ensure privacy of submitted data.

4.12 Challenge 12 – HTTP Cookie Parameter Exposure

This challenge highlights how cookies, when transmitted over an unencrypted HTTP connection, can be intercepted and analyzed using packet capture tools like Wireshark.

Challenge Description

The provided PCAP file captures a simple HTTP request in which a secret key is transmitted as part of the `Cookie` header. The task is to identify this value by examining the HTTP headers.

Objective

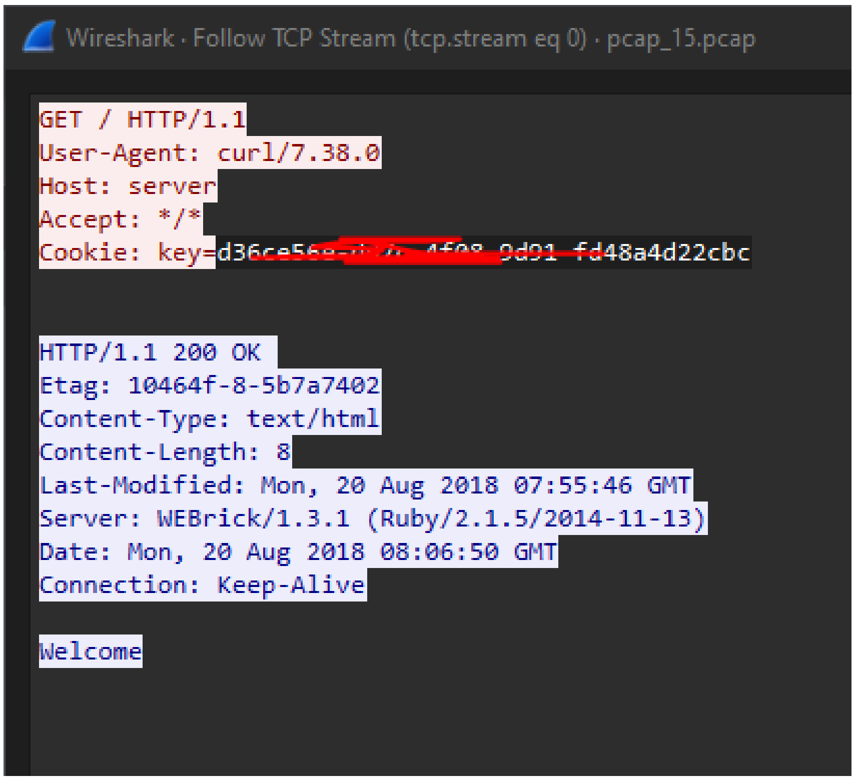
Locate the value associated with the cookie parameter `key` in the HTTP request. This value is the solution to the challenge.

Steps

1. Open `pcap_15.pcap` in Wireshark.
2. Use the display filter: `tcp.stream eq 0` to isolate the HTTP request.
3. Right-click a packet and select **"Follow TCP Stream"**.
4. Look for the header:

   ```
   Cookie: key=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
   ```

5. Extract the value assigned to `key`.


Wireshark · Follow TCP Stream (tcp.stream eq 0) · pcap_15.pcap

```
GET / HTTP/1.1
User-Agent: curl/7.38.0
Host: server
Accept: */*
Cookie: key=d36c[redacted]fd48a4d22cbc


HTTP/1.1 200 OK
Etag: 10464f-8-5b7a7402
Content-Type: text/html
Content-Length: 8
Last-Modified: Mon, 20 Aug 2018 07:55:46 GMT
Server: WEBrick/1.3.1 (Ruby/2.1.5/2014-11-13)
Date: Mon, 20 Aug 2018 08:06:50 GMT
Connection: Keep-Alive

Welcome
```

Figure 11: HTTP GET request containing a key as a cookie

Technical Insight

Cookies are often used for session management, authentication, or tracking. However, when HTTP (instead of HTTPS) is used, these cookies are transmitted in plaintext and can be captured by any party with access to the network.

Cookie format:

```
Cookie: key1=value1; key2=value2
```

In this case, only a single cookie named `key` is present.

Security Considerations

This challenge demonstrates the critical importance of always using HTTPS when transmitting session cookies or other sensitive tokens.

Best Practices:

- Use the `Secure` and `HttpOnly` flags on cookies.
- Enforce HTTPS across the entire application.
- Avoid sending sensitive data in cookies unless absolutely necessary.

4.13 Challenge 13 – HTTP Response Body Exposure (HTML)

This challenge demonstrates how sensitive information can be revealed in the body of an HTTP response. In this case, the server embeds a secret key directly into a web page returned to the client.

Challenge Description

The provided PCAP file contains a single HTTP GET request and the corresponding response. The response body is an HTML page that includes the key inside an HTML tag.

Objective

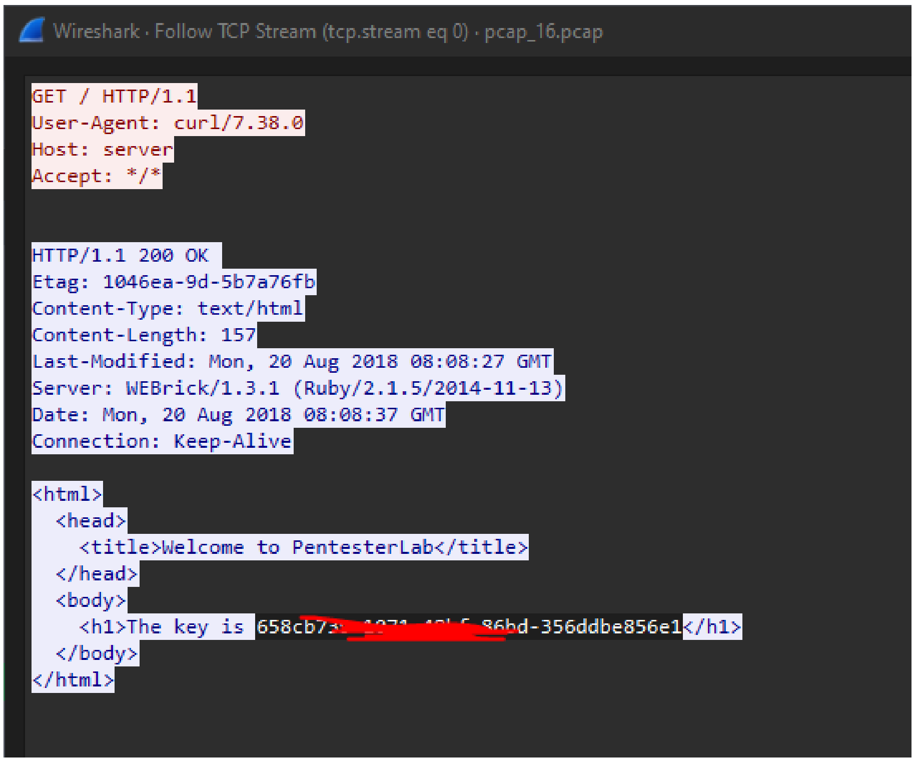
Retrieve the key printed inside the HTML page. This key is located between HTML tags in the body of the HTTP response.

Steps

1. Open `pcap_16.pcap` in Wireshark.
2. Apply the display filter: `tcp.stream eq 0`.
3. Right-click any packet and choose "**Follow TCP Stream**".
4. Inspect the HTTP response body. You will find:

```
<h1>The key is 658cb7...-856e1</h1>
```

5. Extract the key from between the tags.


Wireshark · Follow TCP Stream (tcp.stream eq 0) · pcap_16.pcap

```
GET / HTTP/1.1
User-Agent: curl/7.38.0
Host: server
Accept: */*


HTTP/1.1 200 OK
Etag: 1046ea-9d-5b7a76fb
Content-Type: text/html
Content-Length: 157
Last-Modified: Mon, 20 Aug 2018 08:08:27 GMT
Server: WEBrick/1.3.1 (Ruby/2.1.5/2014-11-13)
Date: Mon, 20 Aug 2018 08:08:37 GMT
Connection: Keep-Alive

<html>
  <head>
    <title>Welcome to PentesterLab</title>
  </head>
  <body>
    <h1>The key is 658cb7[redacted]6bd-356ddbe856e1</h1>
  </body>
</html>
```

Figure 12: HTML body in HTTP response containing the key

Analysis

While this challenge is simple, it emphasizes that sensitive values hardcoded in responses can be intercepted when not protected by encryption. An attacker monitoring HTTP traffic can easily extract HTML contents, including messages, credentials, or tokens.

Recommendations:

- Avoid embedding sensitive information in frontend responses.
- Serve all web content over HTTPS.
- Sanitize output to ensure private content is not unintentionally exposed.

4.14 Challenge 14 – HTTP Set-Cookie Header Exposure

This challenge focuses on retrieving a secret key embedded in the `Set-Cookie` HTTP header of a server response. Although the page content appears empty of sensitive information, the key is included in the HTTP response headers.

Challenge Description

The provided PCAP file captures a basic HTTP GET request made by a client. The server responds with a cookie using the `Set-Cookie` header, and the key is stored as the value of the cookie named `key`.

Objective

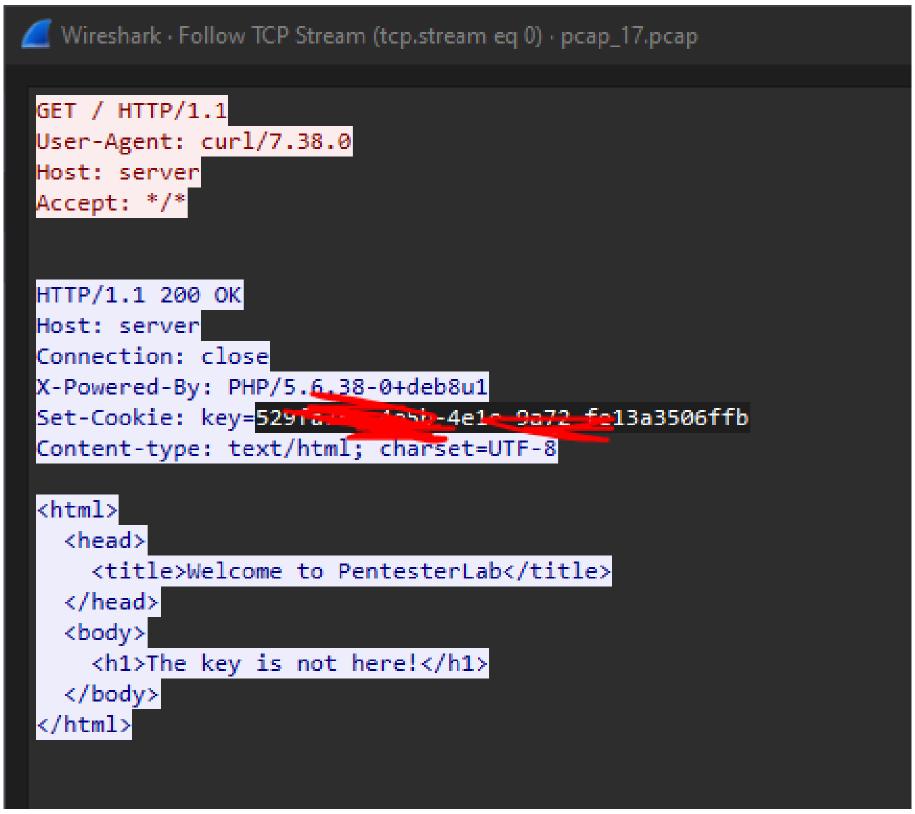
Extract the value of the cookie named `key` from the HTTP response header `Set-Cookie`.

Steps

1. Open `pcap_17.pcap` in Wireshark.
2. Use the filter: `tcp.stream eq 0`.
3. Right-click any packet and choose **"Follow TCP Stream"**.
4. In the HTTP response, locate the header:

```
Set-Cookie: key=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
```

5. Extract the value associated with the cookie named `key`.


Wireshark · Follow TCP Stream (tcp.stream eq 0) · pcap_17.pcap

```
GET / HTTP/1.1
User-Agent: curl/7.38.0
Host: server
Accept: */*


HTTP/1.1 200 OK
Host: server
Connection: close
X-Powered-By: PHP/5.6.38-0+deb8u1
Set-Cookie: key=[redacted]
Content-type: text/html; charset=UTF-8

<html>
  <head>
    <title>Welcome to PentesterLab</title>
  </head>
  <body>
    <h1>The key is not here!</h1>
  </body>
</html>
```

Figure 13: HTTP response with the key inside the Set-Cookie header

Analysis

This challenge shows that sensitive information is not always in the body of an HTTP response — headers can also reveal important data. If cookies are not properly protected, they can be intercepted and reused by attackers, especially over HTTP.

Recommendations:

- Always serve cookies over `HTTPS`.
- Use the `Secure` and `HttpOnly` attributes in cookies.
- Avoid placing critical secrets in cookies unless necessary and well protected.

4.15 Challenge 15 – HTTP Basic Authentication Header

This challenge demonstrates how HTTP Basic Authentication works and how easily it can be compromised when transmitted over an unencrypted connection. The credentials (username and password) are base64 encoded and passed in the `Authorization` header.

Challenge Description

The PCAP file contains a single HTTP request sent using the `curl` tool. The request includes an `Authorization: Basic` header that contains the username and password encoded in base64.

Objective

Extract the password used in the Basic Authentication header. The value of the header follows this format:

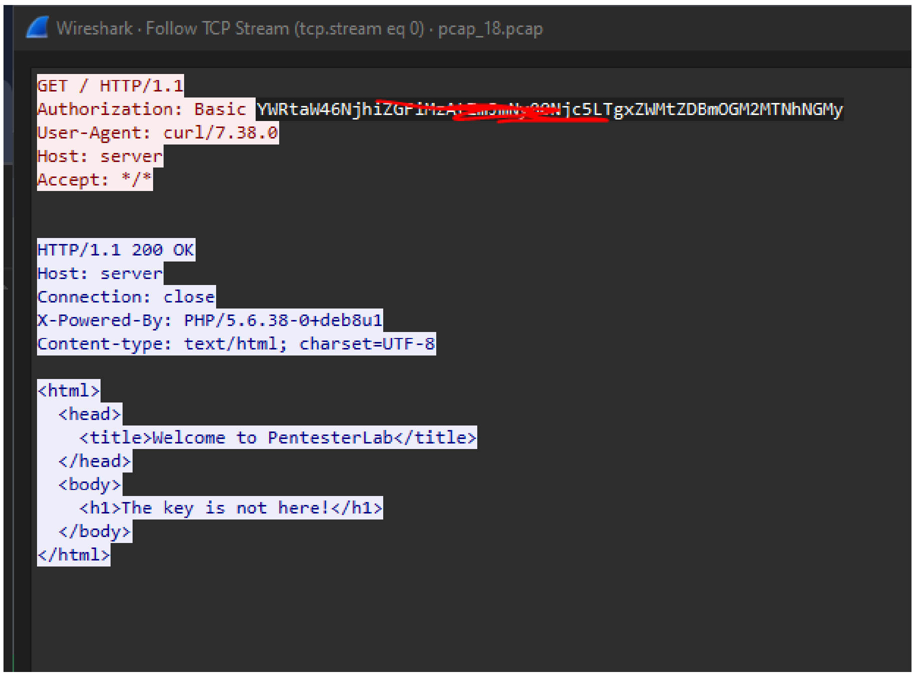
```
Authorization: Basic base64(username:password)
```

Steps

1. Open `pcap_18.pcap` in Wireshark.
2. Use the filter: `tcp.stream eq 0`.
3. Right-click any packet and choose **"Follow TCP Stream"**.
4. Look for the line:

   ```
   Authorization: Basic YWRtaW46cGFzc3dvcmQ=
   ```

5. Decode the base64 string (e.g., with `echo` or `base64 -d`) to get the format `username:password`.
6. The key is the password portion of the decoded value.


Wireshark · Follow TCP Stream (tcp.stream eq 0) · pcap_18.pcap

```
GET / HTTP/1.1
Authorization: Basic YWRtaW46NjhiZ[redacted]jc5LTgxZWMtZDBmOGM2MTNhNGMy
User-Agent: curl/7.38.0
Host: server
Accept: */*


HTTP/1.1 200 OK
Host: server
Connection: close
X-Powered-By: PHP/5.6.38-0+deb8u1
Content-type: text/html; charset=UTF-8

<html>
  <head>
    <title>Welcome to PentesterLab</title>
  </head>
  <body>
    <h1>The key is not here!</h1>
  </body>
</html>
```

Figure 14: HTTP request with Authorization header using Basic Authentication

Analysis

HTTP Basic Authentication is often misunderstood as secure, but it simply encodes the credentials in base64 — which is easily reversible. If HTTP is used without TLS, these credentials are fully exposed on the network.

Security Recommendations:

- Always use HTTPS when transmitting credentials.
- Prefer more secure authentication mechanisms such as token-based authentication.
- Avoid using Basic Authentication in production unless wrapped in strict TLS enforcement.

4.16 Challenge 16 – JWT Authorization Token (pcap_19.pcap)

Objective: This challenge requires us to extract a secret key embedded in a JWT (JSON Web Token) sent through an HTTP request's `Authorization` header.

Protocol Overview:

- **HTTP**: Used for communication between client and server.
- **JWT (JSON Web Token)**: A token format that encodes three components: header, payload, and signature, separated by dots (`.`). Format:

  ```
  header.payload.signature
  ```

- The payload is base64-encoded JSON and may contain sensitive data like the key we need.

Steps:

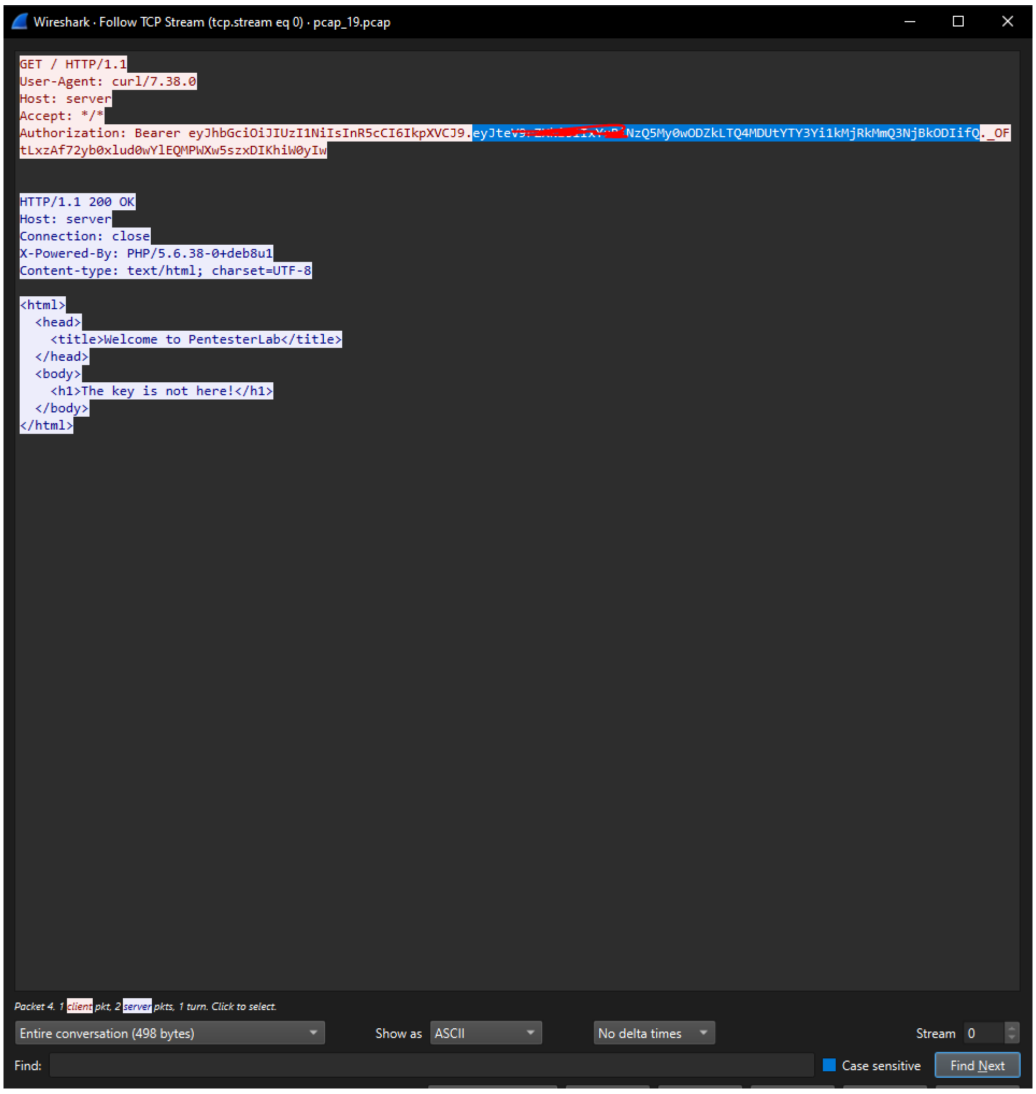
1. Open the capture file `pcap_19.pcap` in Wireshark.
2. Use the filter: `tcp.stream eq 0` to isolate the TCP session.
3. Right-click any packet in the stream and select `Follow` → `TCP Stream`.
4. You will see an HTTP GET request with an `Authorization: Bearer` header containing a JWT token:

   ```
   Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJteV9zZWNyZXQiOiJzdX
   ```

5. Copy the JWT and decode the payload (the middle part between the dots) using a decoder like `https://jwt.io`.
6. The payload typically contains JSON, such as:

   ```
   {
     "my_secret": "super_secret_key"
   }
   ```

7. The `my_secret` field holds the flag or key.



```
GET / HTTP/1.1
User-Agent: curl/7.38.0
Host: server
Accept: */*
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJteV9[redacted]NzQ5My0wODZkLTQ4MDUtYTY3Yi1kMjRkMmQ3NjBkODIifQ._OF
tLxzAf72yb0xlud0wYlEQMPWXw5szxDIKhiW0yIw


HTTP/1.1 200 OK
Host: server
Connection: close
X-Powered-By: PHP/5.6.38-0+deb8u1
Content-type: text/html; charset=UTF-8

<html>
  <head>
    <title>Welcome to PentesterLab</title>
  </head>
  <body>
    <h1>The key is not here!</h1>
  </body>
</html>
```

Packet 4. 1 client pkt, 2 server pkts, 1 turn. Click to select.

Entire conversation (498 bytes) Show as ASCII No delta times Stream 0

Find: Case sensitive Find Next

Wireshark · Follow TCP Stream (tcp.stream eq 0) · pcap_19.pcap

Figure 15: JWT Bearer Token seen in HTTP Authorization Header

Screenshot:

4.17 Challenge 17 – Gzipped HTTP Response Body (pcap_20.pcap)

Objective: In this challenge, we aim to extract a key found in the body of an HTTP response. Unlike previous challenges, the response body is compressed using GZIP, requiring additional decoding.

Protocol Overview:

- **HTTP**: Standard protocol for web communication.
- **GZIP**: The response body is compressed to reduce data transfer size.

Procedure:

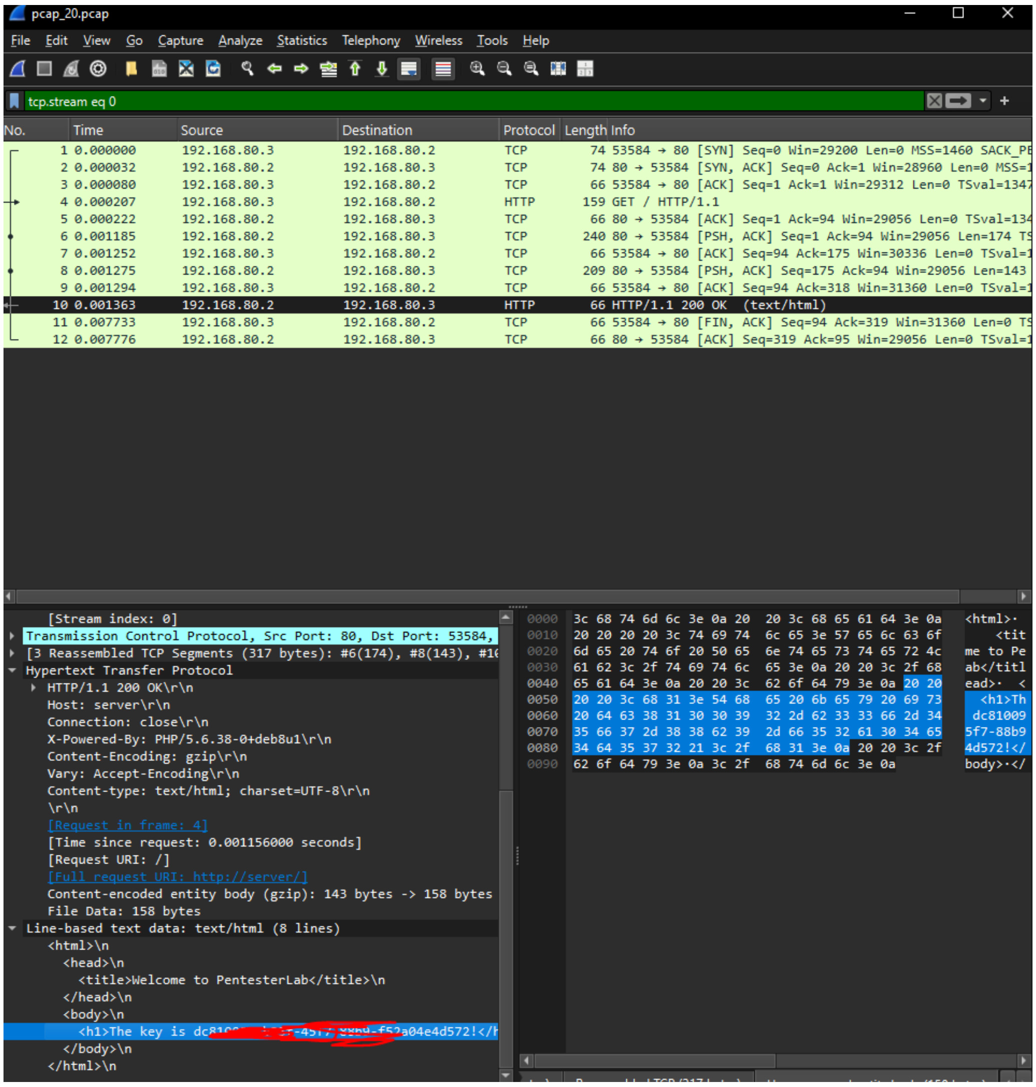
1. Open the file `pcap_20.pcap` in Wireshark.
2. Apply the display filter: `tcp.stream eq 0` to isolate the relevant TCP session.

3. Right-click on any packet in the stream and choose: `Follow` → `TCP Stream`.
4. In the "Follow TCP Stream" window:
   - Select `Show data as: RAW`
   - Click on `Save as...` to store the raw HTTP stream locally.
5. Open the saved file in a text editor.
   - Delete all HTTP headers (everything before the empty line following the headers).
   - Save the remaining binary data.
6. Decompress the file using the command:

   ```
   gunzip -c saved_file > decompressed.html
   ```

7. Open `decompressed.html` and locate the key in the body:

   ```
   <h1>The key is de6f1f12-...-04e4d5721</h1>
   ```



```
pcap_20.pcap
File  Edit  View  Go  Capture  Analyze  Statistics  Telephony  Wireless  Tools  Help

tcp.stream eq 0

No.  Time       Source         Destination    Protocol Length Info
   1 0.000000   192.168.80.3   192.168.80.2   TCP       74 53584 → 80 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PE
   2 0.000032   192.168.80.2   192.168.80.3   TCP       74 80 → 53584 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1
   3 0.000080   192.168.80.3   192.168.80.2   TCP       66 53584 → 80 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=1347
   4 0.000207   192.168.80.3   192.168.80.2   HTTP     159 GET / HTTP/1.1
   5 0.000222   192.168.80.2   192.168.80.3   TCP       66 80 → 53584 [ACK] Seq=1 Ack=94 Win=29056 Len=0 TSval=134
   6 0.001185   192.168.80.2   192.168.80.3   TCP      240 80 → 53584 [PSH, ACK] Seq=1 Ack=94 Win=29056 Len=174 TS
   7 0.001252   192.168.80.3   192.168.80.2   TCP       66 53584 → 80 [ACK] Seq=94 Ack=175 Win=30336 Len=0 TSval=1
   8 0.001275   192.168.80.2   192.168.80.3   TCP      209 80 → 53584 [PSH, ACK] Seq=175 Ack=94 Win=29056 Len=143
   9 0.001294   192.168.80.3   192.168.80.2   TCP       66 53584 → 80 [ACK] Seq=94 Ack=318 Win=31360 Len=0 TSval=1
  10 0.001363   192.168.80.2   192.168.80.3   HTTP      66 HTTP/1.1 200 OK  (text/html)
  11 0.007733   192.168.80.3   192.168.80.2   TCP       66 53584 → 80 [FIN, ACK] Seq=94 Ack=319 Win=31360 Len=0 TS
  12 0.007776   192.168.80.2   192.168.80.3   TCP       66 80 → 53584 [ACK] Seq=319 Ack=95 Win=29056 Len=0 TSval=1
```

```
    [Stream index: 0]
> Transmission Control Protocol, Src Port: 80, Dst Port: 53584,
> [3 Reassembled TCP Segments (317 bytes): #6(174), #8(143), #1
v Hypertext Transfer Protocol
  > HTTP/1.1 200 OK\r\n
    Host: server\r\n
    Connection: close\r\n
    X-Powered-By: PHP/5.6.38-0+deb8u1\r\n
    Content-Encoding: gzip\r\n
    Vary: Accept-Encoding\r\n
    Content-type: text/html; charset=UTF-8\r\n
    \r\n
    [Request in frame: 4]
    [Time since request: 0.001156000 seconds]
    [Request URI: /]
    [Full request URI: http://server/]
    Content-encoded entity body (gzip): 143 bytes -> 158 bytes
    File Data: 158 bytes
v Line-based text data: text/html (8 lines)
    <html>\n
      <head>\n
        <title>Welcome to PentesterLab</title>\n
      </head>\n
      <body>\n
        <h1>The key is dc81[redacted]a04e4d572!</h
      </body>\n
    </html>\n
```

```
0000  3c 68 74 6d 6c 3e 0a 20  20 3c 68 65 61 64 3e 0a   <html>·
0010  20 20 20 20 3c 74 69 74  6c 65 3e 57 65 6c 63 6f       <tit
0020  6d 65 20 74 6f 20 50 65  6e 74 65 73 74 65 72 4c   me to Pe
0030  61 62 3c 2f 74 69 74 6c  65 3e 0a 20 20 3c 2f 68   ab</titl
0040  65 61 64 3e 0a 20 20 3c  62 6f 64 79 3e 0a 20 20   ead>·  <
0050  20 20 3c 68 31 3e 54 68  65 20 6b 65 79 20 69 73     <h1>Th
0060  20 64 63 38 31 30 30 39  32 2d 62 33 33 66 2d 34    dc81009
0070  35 66 37 2d 38 38 62 39  2d 66 35 32 61 30 34 65   5f7-88b9
0080  34 64 35 37 32 21 3c 2f  68 31 3e 0a 20 20 3c 2f   4d572!</
0090  62 6f 64 79 3e 0a 3c 2f  68 74 6d 6c 3e 0a         body>·</
```

Figure 16: Gzipped HTTP Response containing the key in HTML body

Screenshot:

4.18 Challenge 18 – DNS over TCP (pcap_25.pcap)

Objective: The goal of this challenge is to retrieve the key embedded within a DNS query and its corresponding response. This challenge highlights an important networking detail: while DNS typically uses UDP, it can also operate over TCP.

Protocol Overview:

- **DNS (Domain Name System)**: Resolves human-readable domain names to IP addresses.
- **TCP (Transmission Control Protocol)**: Although DNS commonly uses UDP (port 53), TCP is used for zone transfers or large queries.

Procedure:

1. Open `pcap_25.pcap` using Wireshark.

2. Apply the display filter: `tcp.stream eq 0` to isolate the relevant TCP session.
3. Right-click a packet and select: `Follow` → `TCP Stream`.
4. The key will appear embedded in the domain name queried or in the DNS answer section.
5. You can also inspect DNS fields directly by expanding the DNS protocol section in the packet details pane.

Insight: It is a common misconception that DNS only uses UDP. In this example, DNS is performed over TCP. This can be triggered using the `+tcp` option in tools such as `dig`:

```
dig +tcp example.com
```

4.19 Challenge 19 – DNS over TCP (pcap_25.pcap)

Getting the PCAP File You can download the file using the following name: `pcap_25.pcap`.

Challenge Description This capture contains a single DNS query and the corresponding response. Unlike the typical case, the DNS communication in this challenge is conducted over **TCP** rather than UDP.

This is important because many believe DNS always operates over UDP. However, DNS can also use TCP—especially for larger responses or zone transfers.

Steps to Solve:

1. Open `pcap_25.pcap` using Wireshark.
2. Apply the filter: `tcp.stream eq 0`.
3. Right-click one of the filtered packets and select `Follow → TCP Stream`.
4. Look inside both the DNS query and DNS answer sections to locate the **key**.

Tip: To test this behavior manually using a DNS client like `dig`, use:

```
dig +tcp example.com
```

4.20 Challenge 20 – DNS with Predictable Transaction ID (pcap_27.pcap)

Challenge Description This capture illustrates a common vulnerability found in many Internet of Things (IoT) devices: the use of a predictable or fixed DNS transaction ID. The DNS transaction ID is a 16-bit value used to match a DNS request to its corresponding response. When a client sends a DNS query, it includes a transaction ID, and any valid response must contain the same ID. This is a basic mechanism to prevent spoofed responses.

Vulnerability Overview If the transaction ID is:

- **Fixed** (e.g., always 0), or
- **Predictable** (e.g., incremented with every query),

then an attacker can easily guess it. Even without intercepting the request (i.e., not being in the middle), an attacker on the same network can race to inject a spoofed DNS response with a guessed transaction ID. If successful, the victim will accept the malicious DNS answer, allowing redirection to an attacker-controlled domain.

Observed Behavior in the PCAP In this PCAP file:

- All DNS packets use a transaction ID of `0x0000`.
- This behavior is extremely insecure and facilitates DNS spoofing attacks.
- The key to solve the challenge is the **hostname queried**, but **without the domain part**.

From the DNS query packet:

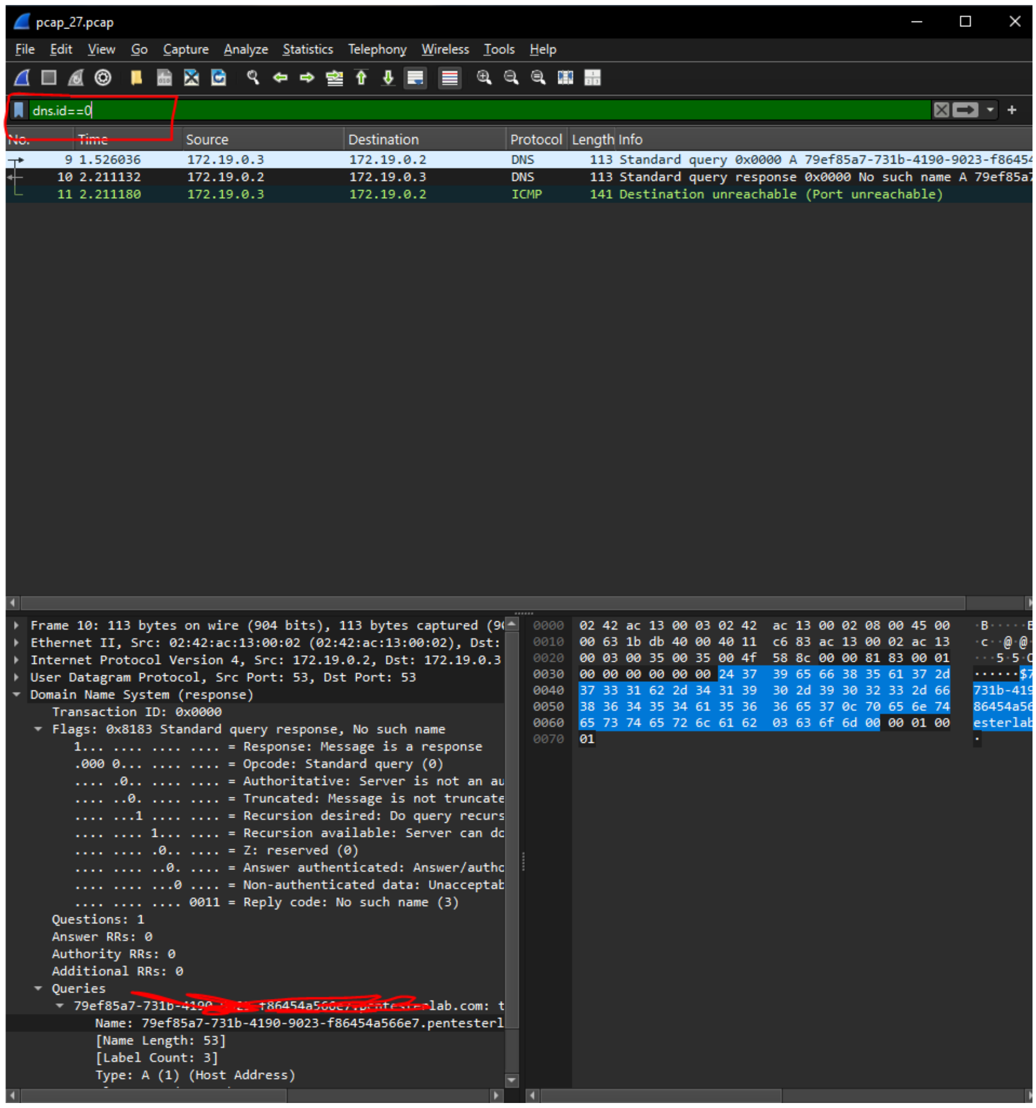
```
Query Name: xxxxxxxxxxx.pentesterlab.com
```

The key is the hostname part:

```
xxxxxxxxxxxxxxxxxxxxxx
```

Security Best Practices To mitigate this vulnerability, devices and applications must:

- Use **randomized transaction IDs**.
- Validate that the **transaction ID in the response matches the request**.
- Consider using DNS over TLS or DNSSEC for additional layers of protection.



```
pcap_27.pcap
dns.id==0

No.   Time       Source       Destination  Protocol Length Info
    9 1.526036   172.19.0.3   172.19.0.2   DNS      113 Standard query 0x0000 A 79ef85a7-731b-4190-9023-f86454
   10 2.211132   172.19.0.2   172.19.0.3   DNS      113 Standard query response 0x0000 No such name A 79ef85a7
   11 2.211180   172.19.0.3   172.19.0.2   ICMP     141 Destination unreachable (Port unreachable)
```

```
> Frame 10: 113 bytes on wire (904 bits), 113 bytes captured (90
> Ethernet II, Src: 02:42:ac:13:00:02 (02:42:ac:13:00:02), Dst:
> Internet Protocol Version 4, Src: 172.19.0.2, Dst: 172.19.0.3
> User Datagram Protocol, Src Port: 53, Dst Port: 53
v Domain Name System (response)
    Transaction ID: 0x0000
  v Flags: 0x8183 Standard query response, No such name
      1... .... .... .... = Response: Message is a response
      .000 0... .... .... = Opcode: Standard query (0)
      .... .0.. .... .... = Authoritative: Server is not an au
      .... ..0. .... .... = Truncated: Message is not truncate
      .... ...1 .... .... = Recursion desired: Do query recurs
      .... .... 1... .... = Recursion available: Server can do
      .... .... .0.. .... = Z: reserved (0)
      .... .... ..0. .... = Answer authenticated: Answer/autho
      .... .... ...0 .... = Non-authenticated data: Unacceptab
      .... .... .... 0011 = Reply code: No such name (3)
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 0
  v Queries
    v 79ef85a7-731b-41[redacted]lab.com: t
        Name: 79ef85a7-731b-4190-9023-f86454a566e7.pentesterl
        [Name Length: 53]
        [Label Count: 3]
        Type: A (1) (Host Address)
```

```
0000  02 42 ac 13 00 03 02 42  ac 13 00 02 08 00 45 00   ·B·····B
0010  00 63 1b db 40 00 40 11  c6 83 ac 13 00 02 ac 13   ·c··@·@·
0020  00 03 00 35 00 35 00 4f  58 8c 00 00 81 83 00 01   ···5·5·O
0030  00 00 00 00 00 00 24 37  39 65 66 38 35 61 37 2d   ······$7
0040  37 33 31 62 2d 34 31 39  30 2d 39 30 32 33 2d 66   731b-419
0050  38 36 34 35 34 61 35 36  36 65 37 0c 70 65 6e 74   86454a56
0060  65 73 74 65 72 6c 61 62  03 63 6f 6d 00 00 01 00   esterlab
0070  01                                                 ·
```

Figure 17: DNS query using fixed transaction ID (0x0000) leaking hostname

Screenshot:

4.21 Challenge 21 – Decrypting TLS Traffic Without Forward Secrecy

Challenge Concept This challenge demonstrates how encrypted network traffic can still be decrypted under certain conditions — specifically, when the server does **not** use a feature called *Forward Secrecy* (FS). It is a great opportunity to learn how TLS encryption works and why FS is critical for secure communication.

What You Are Given You are provided with:

- A network capture file containing a TLS-encrypted session between a client and a server.
- The server's **private key**, which is the secret used by the server to decrypt the session.

What You Need to Do

1. Open the capture file in **Wireshark**.
2. Locate a packet related to the TLS handshake (usually "Client Hello" or "Server Hello").
3. Right-click and go to: `Protocol Preferences → RSA Keys List`.
4. Add an entry containing:
   - The server's IP address,
   - The port number used (typically 443 for HTTPS),
   - The protocol (use "http" or "tcp"),
   - And finally, the path to the provided **private key file**.
5. Once confirmed, Wireshark will use the server's private key to decrypt the TLS session, and you will be able to view the plaintext HTTP traffic — including the secret key used in the exercise.

Why This Works (No Forward Secrecy) This decryption is only possible because the server is using a form of TLS that **does not implement Forward Secrecy (FS)**. In such configurations:

- The same long-term private key is used to establish all TLS sessions.
- If someone obtains this key (e.g., from a compromised server), they can decrypt any past or future sessions that were recorded.

Security Lesson Using TLS without Forward Secrecy is risky. If an attacker records encrypted traffic today and later gains access to the private key, they can decrypt all that past communication. In modern secure systems, FS is strongly recommended or even required. With FS, even if the server's private key is compromised, past sessions remain secure.

Conclusion This challenge is a practical exercise in decrypting TLS traffic when the server is poorly configured. It shows how access to the private key alone can expose sensitive information and emphasizes the importance of using Forward Secrecy to protect encrypted communications.