

Docker and tcpdump

(Lab-09 2023/2024)

PART I

Summary

- *tcpdump*
- Installing *tcpdump* in a docker container
- Exercises

1. tcpdump command

The *tcpdump* is a command line utility that allows the capture of network traffic and traffic analysis in the machine it runs. See below some URLs with documentation about this program.

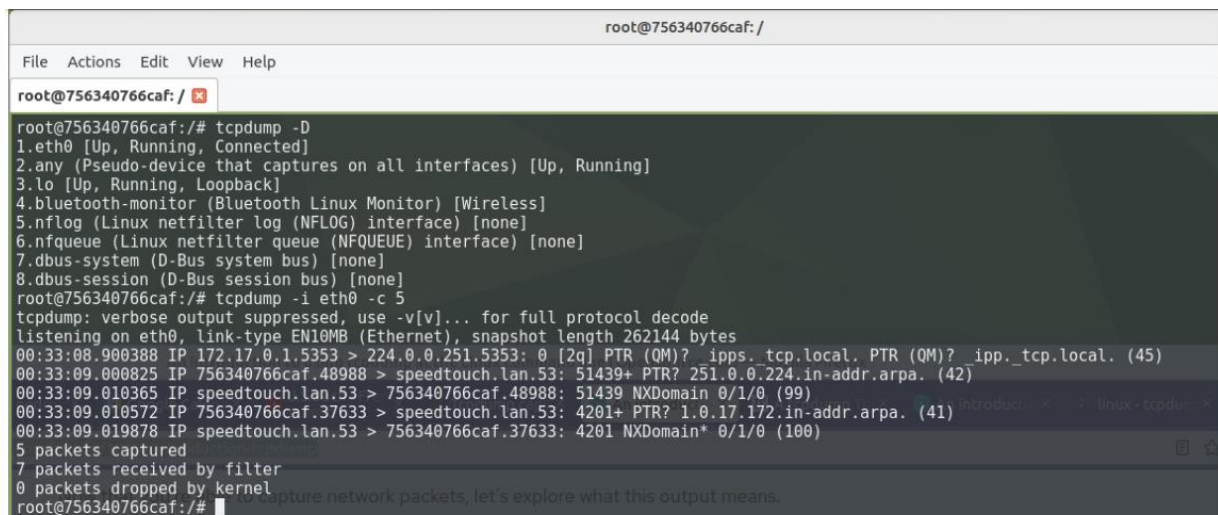
<https://www.tcpdump.org/index.html#documentation>

<https://danielmiessler.com/study/tcpdump/>

<http://www.alexonlinux.com/tcpdump-for-dummies>

<https://danielmiessler.com/study/tcpdump/>

An example of *tcpdump* use is below. In the example, packets are being captured from the Ethernet interface¹ *eth0* (-i eth0) and capture stops after 5 packets (-c 5).



```
root@756340766caf: /  
File Actions Edit View Help  
root@756340766caf: /  
root@756340766caf: /# tcpdump -D  
1.eth0 [Up, Running, Connected]  
2.any (Pseudo-device that captures on all interfaces) [Up, Running]  
3.lo [Up, Running, Loopback]  
4.bluetooth-monitor (Bluetooth Linux Monitor) [Wireless]  
5.nflog (Linux netfilter log (NFLOG) interface) [none]  
6.nfqueue (Linux netfilter queue (NFQUEUE) interface) [none]  
7.dbus-system (D-Bus system bus) [none]  
8.dbus-session (D-Bus session bus) [none]  
root@756340766caf: /# tcpdump -i eth0 -c 5  
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode  
listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes  
00:33:08.900388 IP 172.17.0.1.5353 > 224.0.0.251.5353: 0 [2q] PTR (QM)? _ipps._tcp.local. (45)  
00:33:09.000825 IP 756340766caf.48988 > speedtouch.lan.53: 51439+ PTR? 251.0.0.224.in-addr.arpa. (42)  
00:33:09.010365 IP speedtouch.lan.53 > 756340766caf.48988: 51439 NXDomain 0/1/0 (99)  
00:33:09.010572 IP 756340766caf.37633 > speedtouch.lan.53: 4201+ PTR? 1.0.17.172.in-addr.arpa. (41)  
00:33:09.019878 IP speedtouch.lan.53 > 756340766caf.37633: 4201 NXDomain* 0/1/0 (100)  
5 packets captured  
7 packets received by filter  
0 packets dropped by kernel  
root@756340766caf: /#
```

¹ In different machines the Ethernet interface can have different identifiers, ex: en0.

tcpdump needs privileged access to the network interfaces. In your desktop or laptop, you should run *tcpdump* using *sudo*. In the following, we will install and use *tcpdump* in a Docker container. In a Docker container most of time you are launching processes that run with the uid of privileged user root.

2. Use of tcpdump in a docker container

The following Docker file creates a container with *tcpdump* installed

```
FROM ubuntu
RUN apt-get update && \
apt-get install -y iputils-ping iproute2 python3 tcpdump
```

Generate the Docker image above. Run it and verify that *tcpdump* runs.

Note: If you already have the Docker image from Lab08, you can just install the *tcpdump* in such image if you prefer. Although, you will have to do that each time you launch a container from that image.

3. Send UDP datagrams and observe the traffic with tcpdump

Launch two Docker containers with the image created before. Obtain their IP and Ethernet addresses using the command *ip addr*

In one of the containers run *tcpdump*:

```
tcpdump -i eth0 udp port 9000
```

In the other container run the python3 interpreter and execute the following instructions, replacing 172.17.0.2 by the IP address of the machine where *tcpdump* runs in your case.

```
from socket import *
UDPsocket = socket(AF_INET, SOCK_DGRAM)
UDPsocket.sendto("bla bla bla\n".encode(), ("172.17.0.2", 9000))
```

Verify that *tcpdump* captured the sent datagram. Repeat the *sendto* several times. Each time stop *tcpdump* execution pressing CTRL-C and restart the command with different options, for example:

- To see Ethernet addresses (use *-e* option)
- To see the contents of the datagram (use *-X* option)

4. Send TCP packets and observe the traffic with tcpdump

Using now TCP Sockets, repeat the experiment of previous section. Use the files *TCPclient.py* and *TCPserver.py* from CLIP.

Launch two docker containers :

- In one container run the *TCPserver.py* in background (using *&*) and *tcpdump* with a command line like:

```
tcpdump -i eth0 tcp port 9000
```


(Note: Replace the port 9000 by the port number defined in your program)
- In the other container run the *TCPclient.py*.

5. Spying http

Reuse the programs of lab 8 where we used a python program implementing *http.server*, *launched in the following way*:

```
python3 -m http.server 9050
```

Launch two docker containers:

- In one container run the *http.server* in background (using *&*) and *tcpdump*;
- In the other container run the command *telnet*, to obtain pages from the http server in the same way we did in lab 8 and try to obtain (transfer) a file (ex., an HTML page)

Use *tcpdump* to observe the HTTP protocol (exchanged messages) in action.

Note: *telnet* has to be installed in the container: `apt-get install telnet`

6. Spying ftp

Launch two docker containers:

- In container1 install the *ftp* client:

```
apt-get install ftp
```
- In container2 install the *ftpd* daemon. This will require the following steps:

```
apt-get install ftpd  
apt-get install systemctl  
systemctl start inetd
```

After these commands *ftpd* should be running. Try it by running in container1, replacing 172.17.0.2 by the IP address of the machine where *ftpd* runs

```
ftp 172.17.0.2
```

Verify that a username and a password are required. Terminate *ftp* with CTRL-C. Now in container2 create a user named *rc* (*useradd*) with a password *rc2324*.

Start *tcpdump* with the following command line:

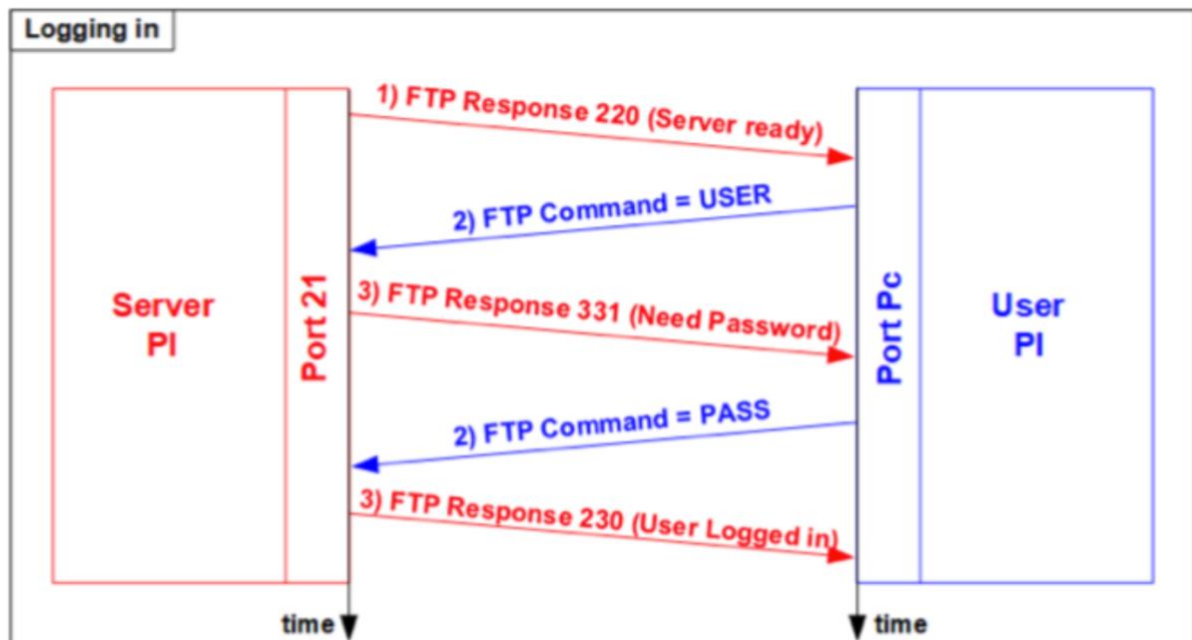
```
tcpdump -i eth0 tcp port 21
```

See in *tcpdump* manual the option that shows the contents of the packets exchanged. Notice that the use of this option allows to see the password of the user in clear.

FTP protocol - is a protocol for file transfer, it uses two TCP connections to transfer a file:

- a *control connection* on port 21, this connection is used for commands from the client to the server and for the server's replies and stays open for the entire time the client communicates with the server.
- a *data connection* on port 20, this connection is used for file transfer.

Observe the sniffed packets from *tcpdump* to trace the ftp protocol, as represented in the following pictures:



HTTP - Hyper Text Transfer Protocol (cont.)

(Lab-09 2023/2024)

PART II

Summary

- HTTP partial transfers (use of range)
- Exercise: HTTP file transfer
- Dynamic contents with parameters supplied in the browser

6. HTTP headers

HTTP request/response headers, respectively convey information about the client (request) and server (response).

HTTP request methods: GET, HEAD, PUT, POST, DELETE, PATCH, OPTIONS, CONNECT and TRACE (https://www.w3schools.com/tags/ref_httpmethods.asp)

The GET method requests data from a specific resource. The HEAD method asks for a response identical to a GET request, but without the body.

HEAD requests are useful for checking what a GET request will return before actually making a GET request, it can be used for example before downloading a large file or response body.

```
HEAD /earth.jpg HTTP/1.1\r\nHost: localhost:8080\r\nConnection: keep-alive\r\nUser-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) ... \r\nAccept: */*\r\nAccept-Encoding: gzip, deflate, br\r\nAccept-Language: en-US,en;q=0.9,de;q=0.8,es;q=0.7,pt;q=0.6\r\n\r\n
```

**HEAD
Request**

**HEAD
Response**

```
HTTP/1.1 200 OK\r\nDate: Wed, 27 Oct 2021 14:35:19 GMT\r\nServer: Apache/2.4.51 (Unix)\r\nLast-Modified: Tue, 05 Oct 2021 23:19:37 GMT\r\nETag: "fc85-5cda3435bb440"\r\nAccept-Ranges: bytes\r\nContent-Length: 64645\r\nKeep-Alive: timeout=5, max=100\r\nConnection: Keep-Alive\r\nContent-Type: image/jpeg\r\n\r\n
```

Based on the HTTPClientDraf.py of lab 8, build a small program to test the HEAD method or use the telnet (or python interpreter) as specified in part1 of this lab.

7. HTTP partial requests

HTTP protocol allows the retrieval of partial resources, for example it enables resuming a file download. If this functionality is supported by the server it is signified by the inclusion of the **Accept-Ranges: bytes** header in response to a **GET** or **HEAD** command.

The client performs a partial request by including a **Range: Bytes=x-[y]** header, where **x** and **y** are the initial and final offsets of the requested data range. If the **y** value is omitted, it will be interpreted as the length of the data resource.

Example of a partial request:

```
GET /earth.jpg HTTP/1.1\r\n
Host: localhost:8080\r\n
Connection: keep-alive\r\n
Range: bytes=0-49\r\n
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) ... \r\n
Accept: */*\r\n
Accept-Encoding: gzip, deflate, br\r\n
Accept-Language: en-US,en;q=0.9,de;q=0.8,es;q=0.7,pt;q=0.6\r\n
\r\n
```

HTTP Partial Request

```
HTTP/1.1 206 Partial Content\r\n
Date: Wed, 27 Oct 2021 15:02:01 GMT\r\n
Server: Apache/2.4.51 (Unix)\r\n
Last-Modified: Tue, 05 Oct 2021 23:19:37 GMT\r\n
Accept-Ranges: bytes\r\n
Connection: Keep-Alive\r\n
Content-Type: image/jpeg\r\n
Content-Range: bytes 0-49/64643\r\n
Content-Length: 50\r\n
\r\n
```

HTTP Partial Response

The **206 Partial Content** status code, indicates that the request has succeeded and the response body contains the requested ranges of data, as described in the **Range** header of the request. The **Content-Range** response HTTP header indicates where in a full body message a partial message belongs.

Content-Range: <unit> <range-start>-<range-end>/<size>

<unit> - the unit in which ranges are specified (usually bytes)

<range-start> - an integer indicating the start position of the requested range

<range-end> - an integer indicating the end position of the requested range

<size> - the total length of the requested data (or '*' if unknown)

8. Exercise: HTTP segmented downloader

Implement a segmented file transfer using HTTP partial requests.

The client command line: `python RangeHTTPclient.py name1 name2`

name1 – the complete URL `http://localhost:8000/resourceName`, corresponding to the file to be transferred.

name2 – name of the file where the client saves the body parts of the server replies.

HTTP Server

Instead of using the `http.server` you need to install and use the `RangeHTTPServer`. This is similar to the `http.server` but responds to partial requests.

Command line to install²: `pip install rangehttpserver`

Command line to run: `python -m RangeHTTPServer`

It will use the default port 8000, to change it to another port at the end of the command line indicate the port number you want to use.

For example: `python -m RangeHTTPServer 9000`

9. Dynamic contents with parameters supplied in the browser

To run the http server in the terminal, write the following line in a terminal:

```
$python -m http.server -d ./www -cgi 9000
```

This will start a webserver waiting connections on port 9000, with the document root defaulting to the subdirectory `www` of the current directory. You can start the server in a folder of your choice. The chosen folder will be populated with several HTML files and a folder called `cgi-bin`

You can use a program editor like *notepad++*, *geany* or *vscode* to create a file called *multiply.py* that will be stored in folder *cgi-bin*. The contents of the file should be:

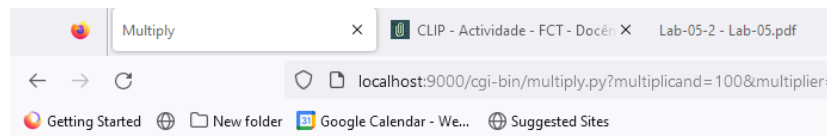
```
import sys
import codecs
import cgi

sys.stdout = codecs.getwriter("utf-8")(sys.stdout.detach())
print("""Content-type:text/html\n\n
<!DOCTYPE html>
<head>
    <title> Multiply </title>
</head>
<body>
    <h1> Multiply </h1>
    <h3> """)
form = cgi.FieldStorage()
multiplicand = int(form["multiplicand"].value)
multiplier = int(form["multiplier"].value)
result = multiplicand * multiplier
print(f"{ multiplicand } times {multiplier} equals {result} </p>")
print("""</h3>
</body> </html>""")
```

² When using docker container you will need to install the pip application.

You can invoke this very sophisticated multiply, using a browser and entering in the dialog box.

<http://localhost:9000/cgi-bin/multiply.py?multiplicand=100&multiplier=20>



Multiply

100 times 50 equals 5000

An alternative is creating the following file called *multiply.html* and store it in `.\www` (virtual root of the web server):

```
<html>
<head>
<title>Multiply</title>
</head>
<body>
<h2>Add employee</h2>
<form method="get" action="http://localhost:9000/cgi-bin/multiply.py">
Enter numbers to multiply<br /><br />
Multiplicand <input type="text" name="multiplicand" value="" /> <br><br>
Multiplier <input type="text" name="multiplier" value="" /> <br><br>
<input type="submit" value="Multiply" /> <br><br>
</form>
</body>
</html>
```

You can invoke this very sophisticated multiply, using a browser and entering in the dialog box <http://localhost:9000/multiply.html>

That will show the following page:

A screenshot of a web browser window. The title bar shows 'Multiply' and 'CLIP - Actividade - FCT - Docê...'. The address bar shows 'localhost:9000/multiply.html'. The browser's toolbar includes 'Getting Started', 'New folder', 'Google Calendar - We...', and 'Suggested Sites'. The main content area displays the text 'Enter numbers to multiply' in a bold, sans-serif font. Below this text are two input fields: 'Multiplicand' and 'Multiplier'. At the bottom of the form is a button labeled 'Multiply'.

Where you can introduce the two values to be multiplied. After pressing button “Multiply” the following will appear in the browser:

