Moshe Ofer & Matanya Brazili

208821652 208982991

A sniffer is a type of software tool that is designed to capture and analyze network traffic.
Sniffers are used for network troubleshooting, security analysis, and monitoring network activity for unauthorized behavior.

Sniffers work by capturing network packets using Promiscuous mode, which give us the ability to capture all network traffic, rather than just traffic that is addressed to our host. Then the sniffer can analyze the packets which we've captured and can include information such as IP addresses, port numbers, and payload data. Some sniffers can capture and decode other types of data, such as email or web traffic.

Sniffers have some **limitations**.

- Encryption - can make it difficult or impossible to capture and analyze data.
- Detection - can alert administrators to the presence of a sniffer on the network

In this task we focused on building our sniffer in C language, the sniffer can capture and analyze network traffic using a pcap library.

Our sniffer extracts the Ethernet, IP and App headers in each of the packet and analyzes them.
It uses BPF (Berkeley Packet Filter) to filter certain types of packets for example based on their port or protocol etc. It uses a structs to define the format of the headers, and it prints the main data to the screen. In addition, the sniffer writes most of the following information from the packet to the file according to the assignment requires:

Makefile instructions:

We joined a simple Makefile to this assignment in order to make life easier. Here is the instruction:

- Make sure you have make and GCC installed on your system.
- If needed run:

  `$ sudo apt-get install libpcap-dev`

  to install pcap library.
- Open terminal and navigate to the directory where the Makefile and the source files are located.
- Run `$ make all` to build all the executables.
- You can run `$ make <executable_name>` to build one executable at a time.
- Run `./<executable_name>` to run the executables.
- Run `$ make clean` to remove the executables.

Task A

Q1.1: *Why do you need the root privilege to run a sniffer program?*

In order to capture the all traffic, even if it is not belong to us, the sniffer must run a "Promiscuous mode" which is mode that brings us the ability to make the ethernet or wlan card, pass all the traffic it received to the kernel. In other words, if you have this ability, you can read all the messages and communications of everyone else on the network, therefore the sniffer runs on low level, which requires a root privilege, to avoid unauthorized users to get access to the kernel.

Q1.2: *Where does the program fail if it is executed without the root privilege?*

The program runs a function called 'pcap_live_open', part of the input argument is '1' which indicate the program to run with "Promiscuous mode".

So, if we will run without the root privilege the "Promiscuous mode" will not work, and we will not be able to bypass the kernel filter for traffic who don't intendent to us.

Abilities:

1. Capturing: Sniffers can capture and analyze all the packets that are transmitted over a network. This allows us an overall view of network traffic and can be useful for identifying patterns or exception.

2. Analysis: Sniffers can decode and analyze various network protocols, such as TCP, UDP, and HTTP, which allows us deeper understanding of network traffic.

3. Filtering: Sniffers can filter captured packets based on various rules, such as IP address or port number, which allows for a more targeted analysis of network traffic.

Limitations:

1. Encryption: Many modern networks use encryption to protect their traffic, which makes it difficult for sniffers to analyze that traffic without the encryption keys.

2. Detection: Sniffers can be detected by various security measures, such as intrusion detection systems, which can make it difficult for an attacker to use a sniffer to gain unauthorized access to a network.

Sniffer:

To run the "sniffer" executable, use the following command:

```
$ sudo ./sniff <filter> <interface>
```

Replace <filter> with a BPF filter to capture specific packets (e.g. "tcp","udp").

Replace <interface> with the network interface you want to capture on (e.g. eth0, wlan0).

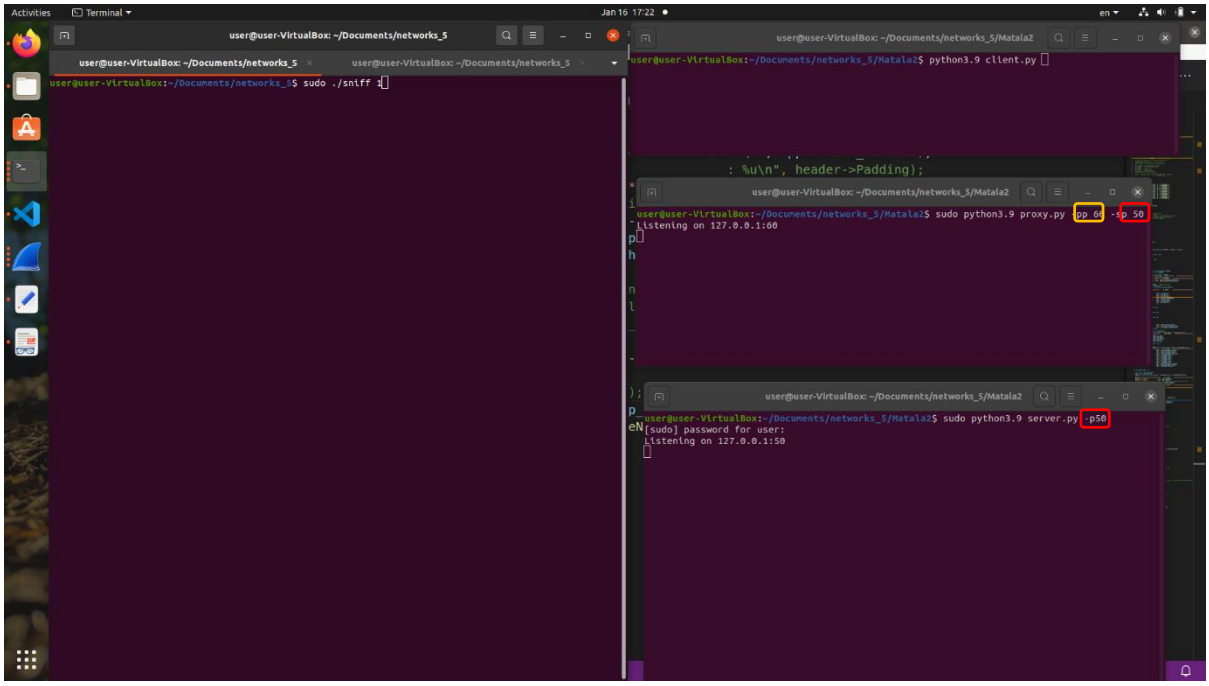
Example:

```
$ sudo ./sniff  tcp eth0
```

This command captures all the tcp packets on the eth0 interface.

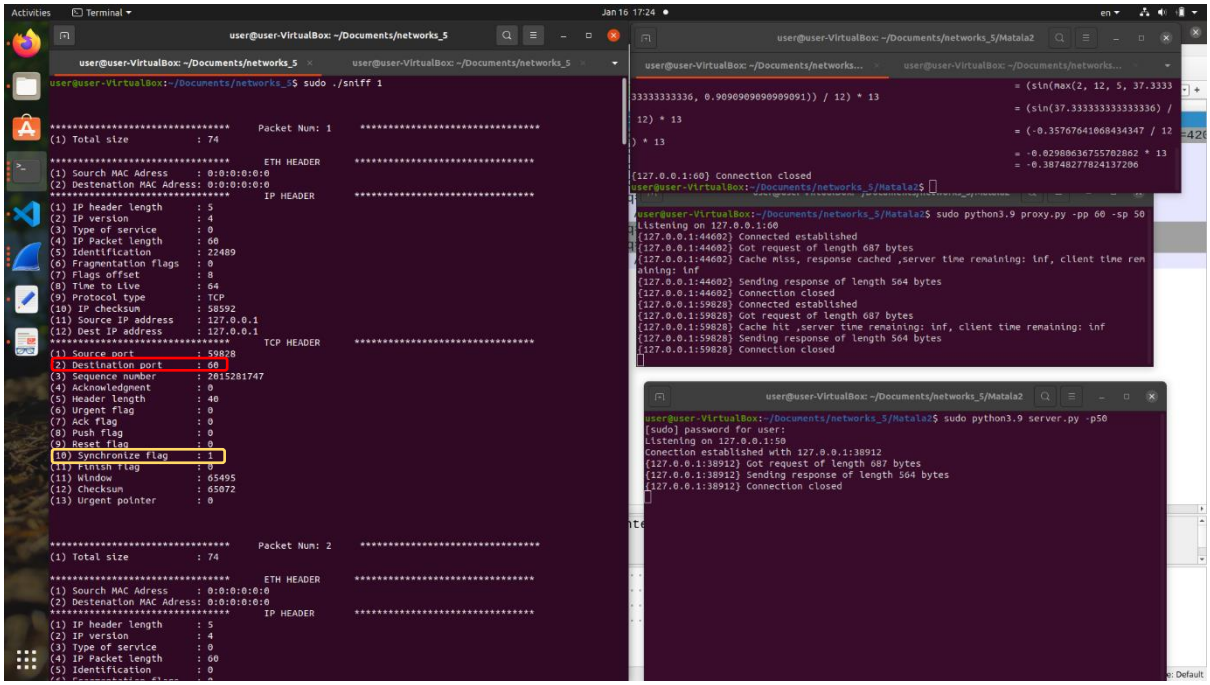
Then it will generate a txt file named "208821652_208982991.txt" with all the **Push** packet only. We run the Matala 2 files, with all the server.py, proxy.py, client.py etc. we have captured all the packet which contains the data and print in the txt file in the following format:

"source_ip: %s, dest_ip: %s, source_port: %d, dest_port: %d, timestamp: %u, total_length: %d, cache_flag: %d, steps_flag: %d, type_flag: %d, status_code: %d, cache_control: %d Data: \n"

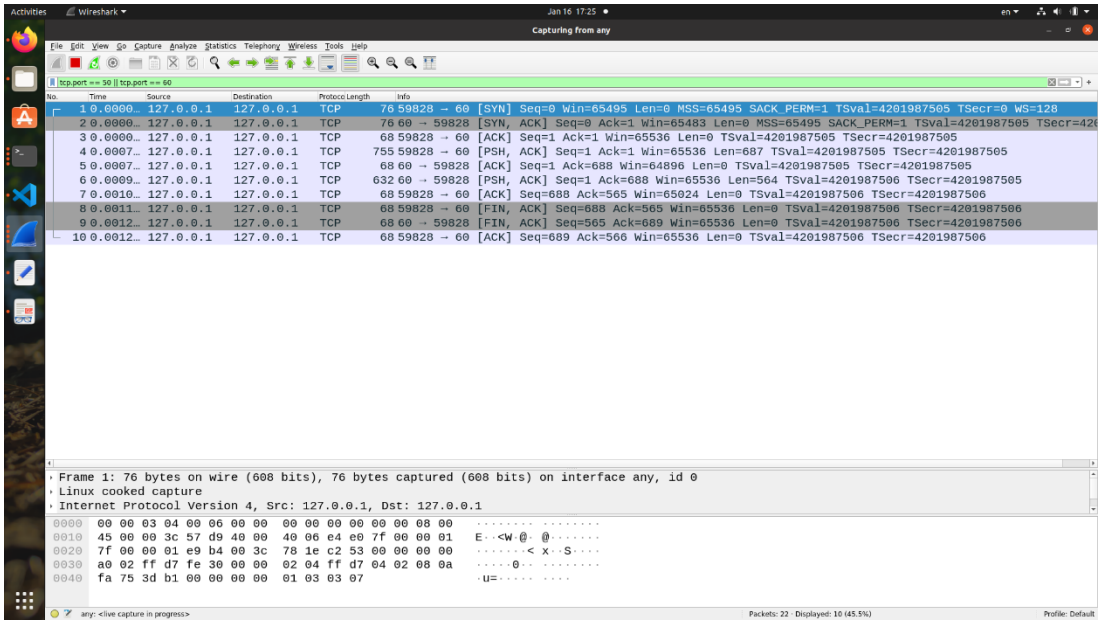


In the pic you can see that we run

1. The server.py on port 50
2. The proxy.py on port 60
3. The client on port 60 to proxy
4. The sniffer on "icmp" mode

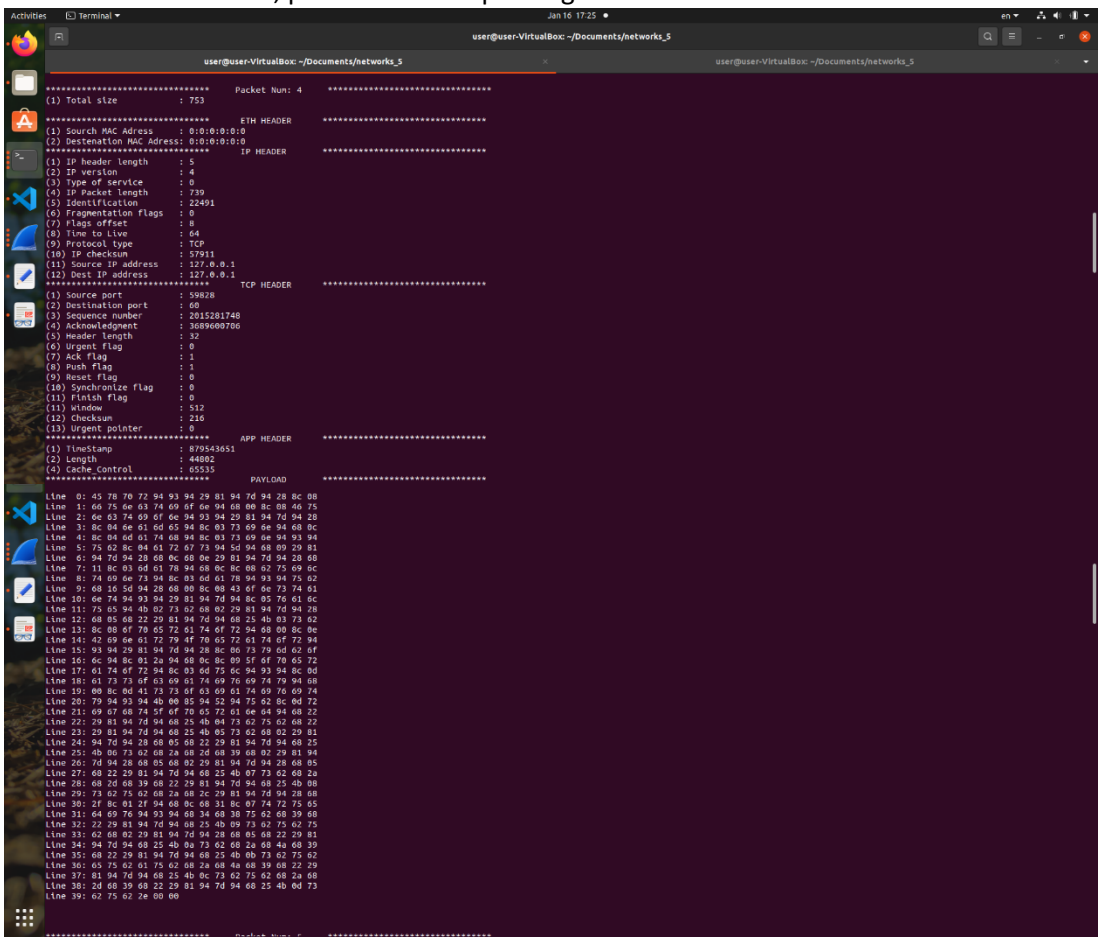


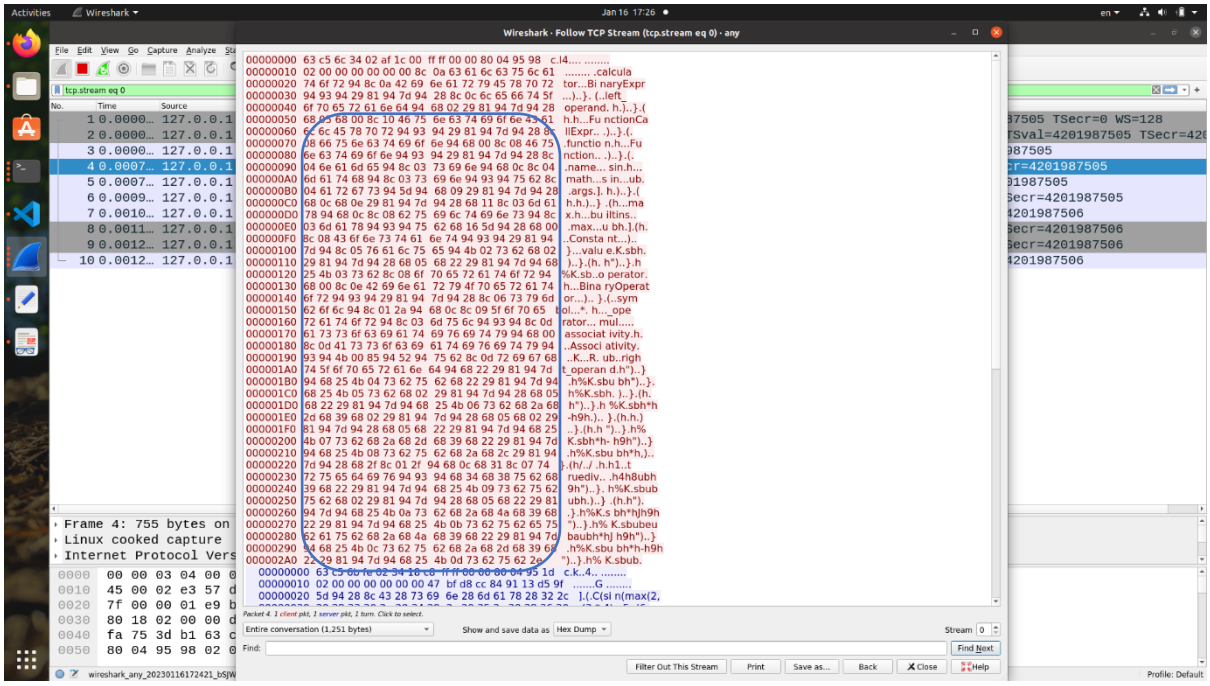
As you can see the sniffer succeed to sniff the packet, on the left, notice to the **"Destination Port : 60"(mark red)** this is the first Syn(look at the flag mark gold), from client to proxy.



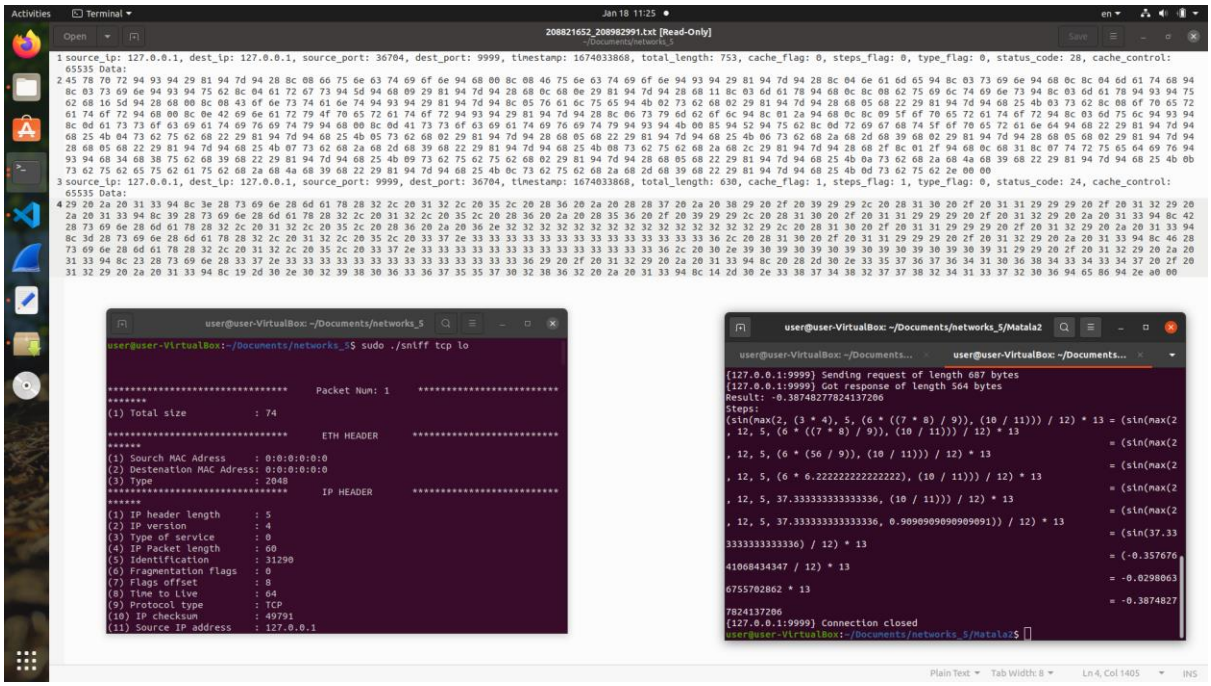
Parallelly we captured it on Wireshark,(filter by the tcp.port) notice that it's the same packets.

After the three hand shake, packet num 4 is pushing the data:





Notice that it can be compared to the Wireshark captured packet, the data of packet number 4 (Mark blue) (Please see the pcap files attached).



In this pic you can see two terminals, one for the server.py and client.py and one for the Sniffer.

On the back, look at the txt file that log the 2 relevant push packets with the data. We choose to log just the push packet because those are the ones who have the payload.

Task B

Spoofing, in general, refers to the act of pretending to be another host as trusted source. There are many different types of spoofing, including email spoofing, phone call spoofing, and website spoofing. These attacks can be used for a variety of purposes, such as phishing scams, identity theft, and spreading malware.

The **limitation** of spoof is that many spoofing techniques can be detected by using network monitoring tools and identified by detection systems.

In addition, Spoofing requires a certain level of technical expertise, and not everyone has the knowledge to do so.

For this part of the assignment, we build in c a file called spoffer.c. This spoofer will be able to send an ICMP package to a known destination, in the name of another host, that may not be really exist. We have built our code such that, it can be modified by other programmers in the future easily, to support any kind of protocols.

Q1: *Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?*

Yes, but it's definitely not recommended.
The "spoofer" program can set the IP packet length field to an arbitrary value, but it might cause issues with how the packet is handled by host and can lead to packet loss or errors.

Q2: *Using the raw socket programming, do you have to calculate the checksum for the IP header?*

Yes, when using raw sockets to send an IP packets, checksum field is can be either handled automatically by operating system or handled manually .
However, it depends on the type of communication, in some communication protocols the checksum calculation is made by the operating system (TCP etc).
and with some other protocols it will not (UDP etc.).
So, when we are creating our packets, it will be better to calculate the checksum yourself and avoid troubles.

Spoffer

To run the "spoffer" executable, use the following command:

```
$ sudo ./spoff <src_ip> <dst_ip> [Data][Len of data]
```

Replace <src_ip> with the source IP address you want to use for the ICMP packet.

Replace <dst_ip> with the destination IP address you want to send the packet to.

Optional: [Data] We added the ability to implant the data in the fake ping.

[Len of data] The length of the data to take. Default as strlen(Data).

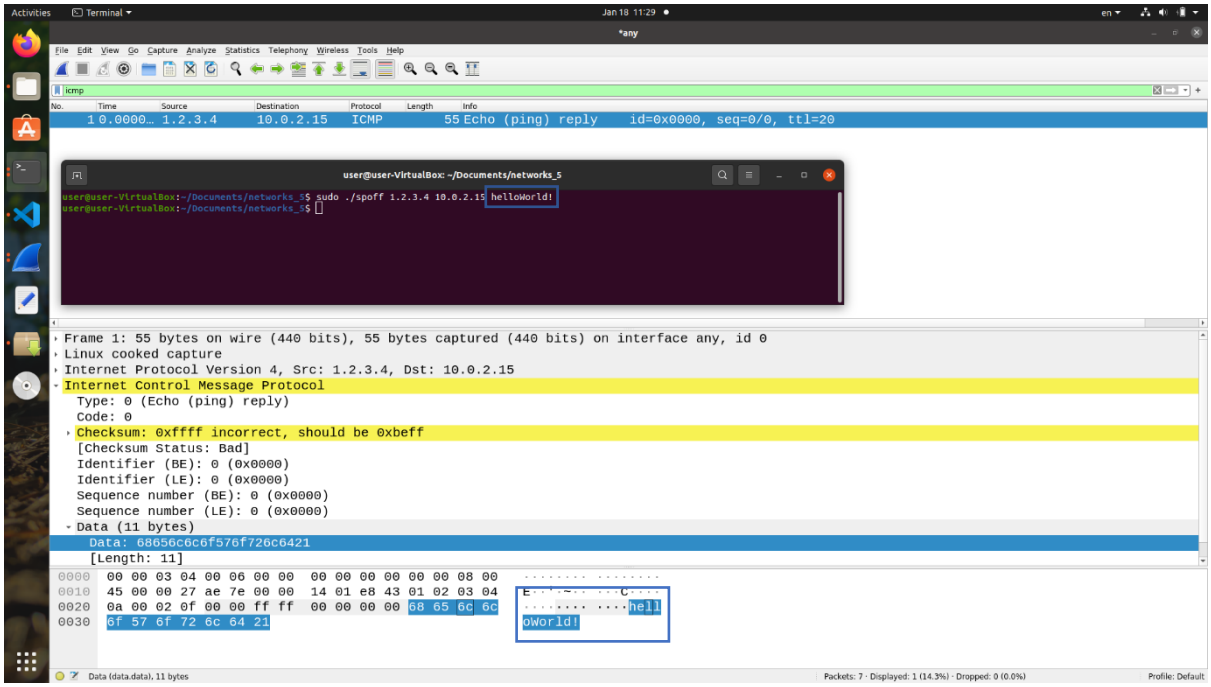
Example:

```
$ sudo ./spoff 10.0.0.1 8.8.8.8
```

This command sends an ICMP echo request using 10.0.0.1 as the source IP address to 8.8.8.8 (Google DNS) as the destination IP address.

It should be noted that this code uses raw sockets to send the packet, which requires root privileges. That's why it's necessary to use "sudo" before running the command.

Screen shots for Spoofer



In the pic, we see a terminal that we run the following command:

```
$ sudo ./spoff 1.2.3.4 10.0.2.15 helloWorld!
```

In addition, we run the Wireshark that captured the fake ICMP packet from fake IP 1.2.3.4 to us.

Please notice that we have implanted the data in the packet. (Mark on blue).

Task C

Sniff-and-then-spoof program = Snoffer

For the third task we wrote the "Snoffer", a combination of the previous two tasks. To let the seed-attacker fake a Replay ICMP to Host A he sniffs the packets on-Host A interfaces "vethcb02347", and then at the moment he sees an ICMP Request, he sends him a fake Reply.

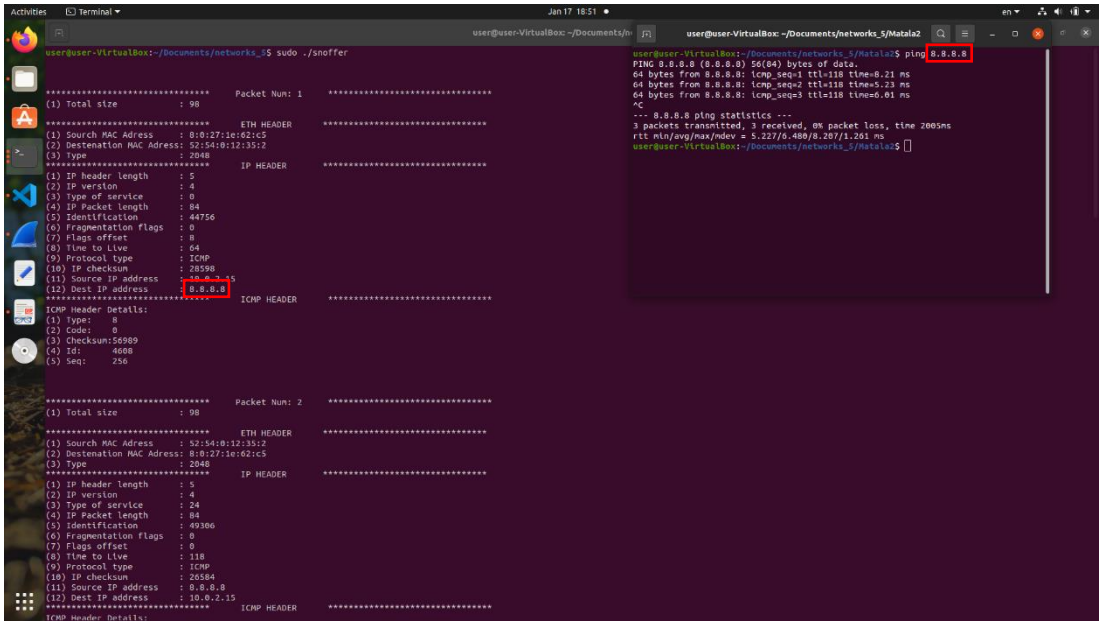
To run the Snoffer it's done similarly to the Sniffer. For our use we'll need to do something like this:

```
./snoffer icmp vethcb02347
```

"vethcb02347" is the interface that Host A runs on.

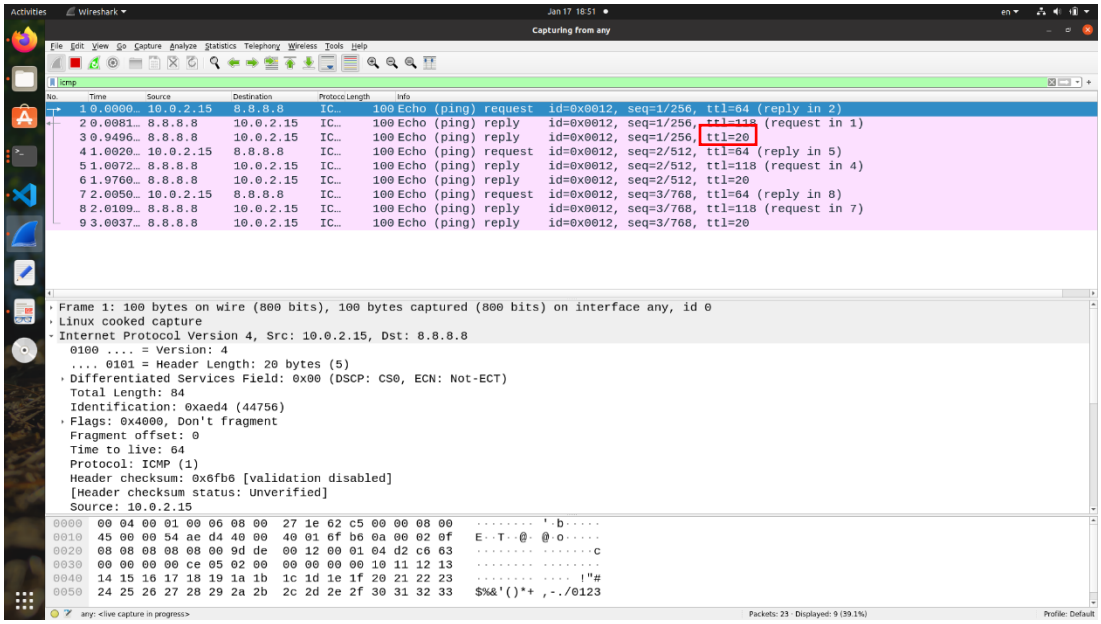
First Step

We run the Snoffer on the regular ubuntu without the docker in order to test the ability to sniff and spoff an icmp packet with the regular ping command.



As you can see the snoffer indeed capture the packets. See the destination for example

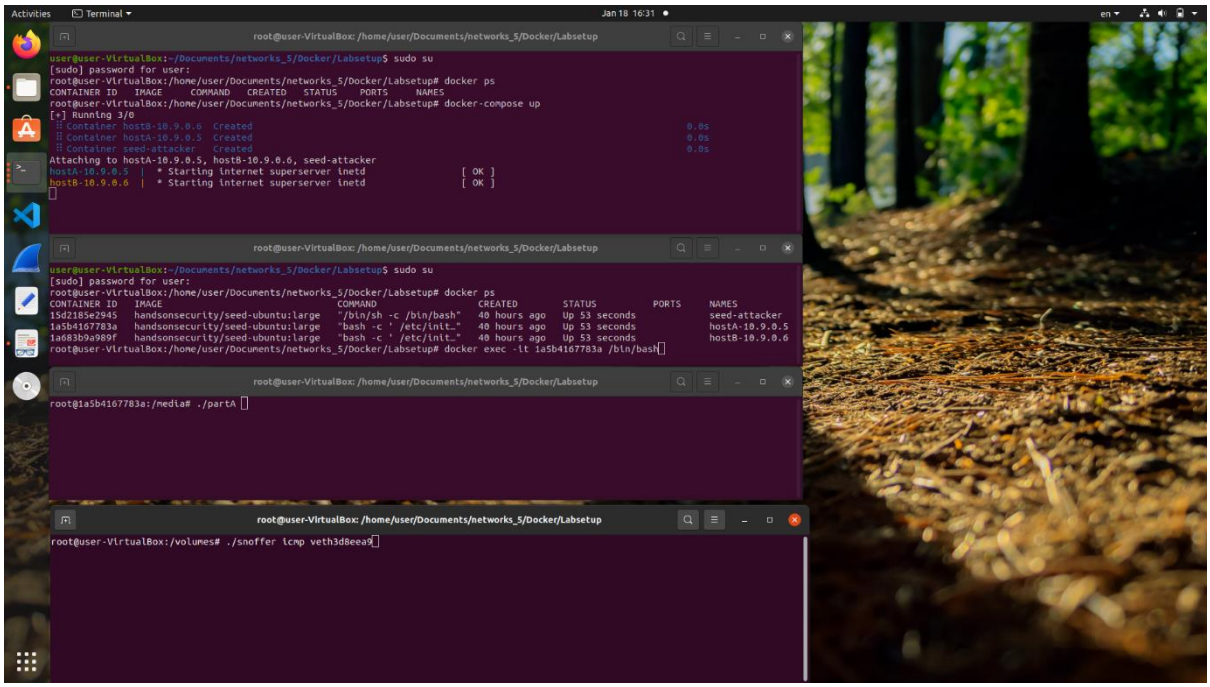
Now let's have a look at the Wireshark:



Something amazing append. We captured the Requests and for each one we've got 2 Replies.

The first one is the real one, and the second is the fake one. It can be noticed by the **TTL** value, the fake one as an arbitrary value of 20.

Second Step – **Docker**



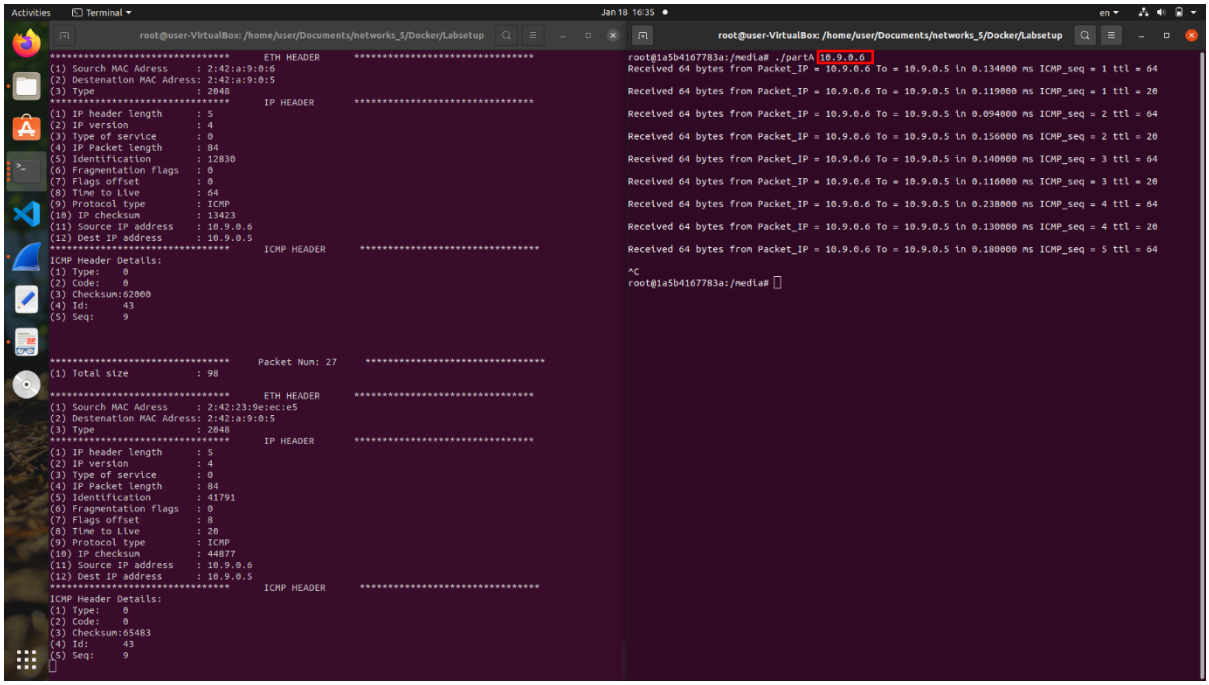
- The top terminal shows all the containers.
- The second terminal shows the IDs of all containers.
- The third terminal shows the bin\bash\ inside **Host-A**. We run the ping from Matala 4 there.
- The fourth terminal shows the bin\bash\ inside **Attacker**. We run our Snoffer with the suitable arguments.

We have set up all the containers that we got from the Ta's and copy from outside the Host-A the PartA from Matala 4. We also put the Snoffer in the volumes folder to let the seed-attacker to run Snoffer.

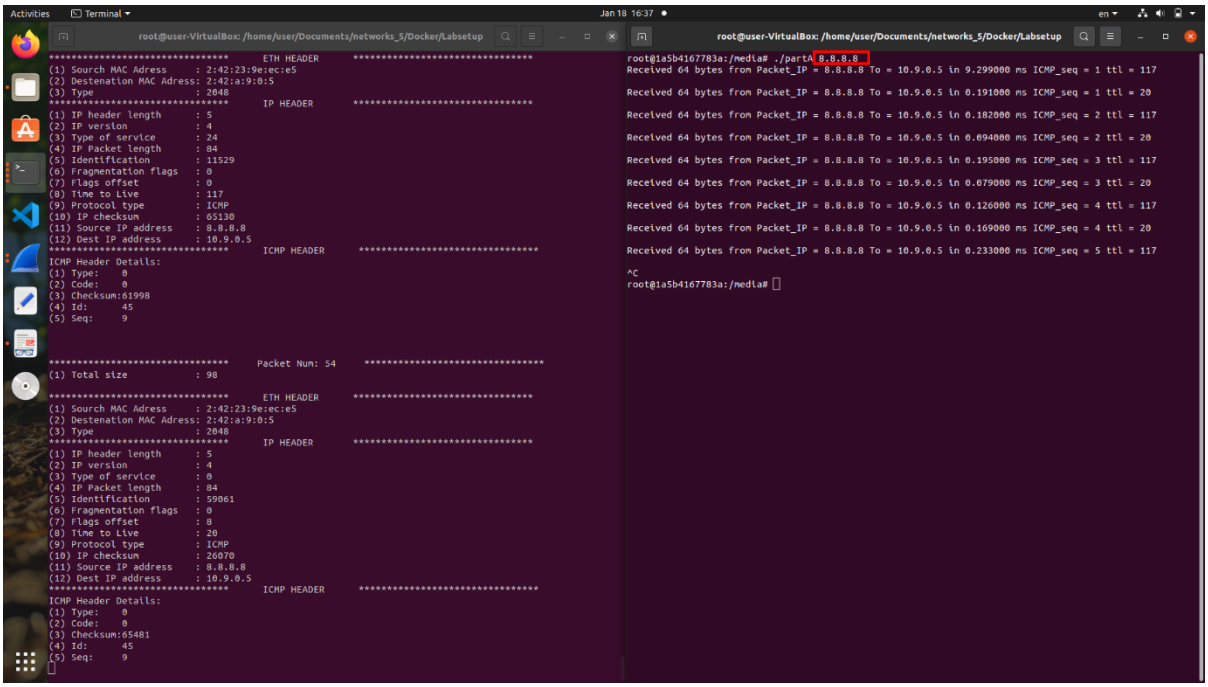
In the above pic, we run the following commend from the **Attacker bin\bash\:**
$ ./snoffer icmp veth3d8eea9
That opened the Snoffer on protocol ICMP on Host-A Interface's.

After that, we run 3 times the following commends from **Host-A bin\bash**:

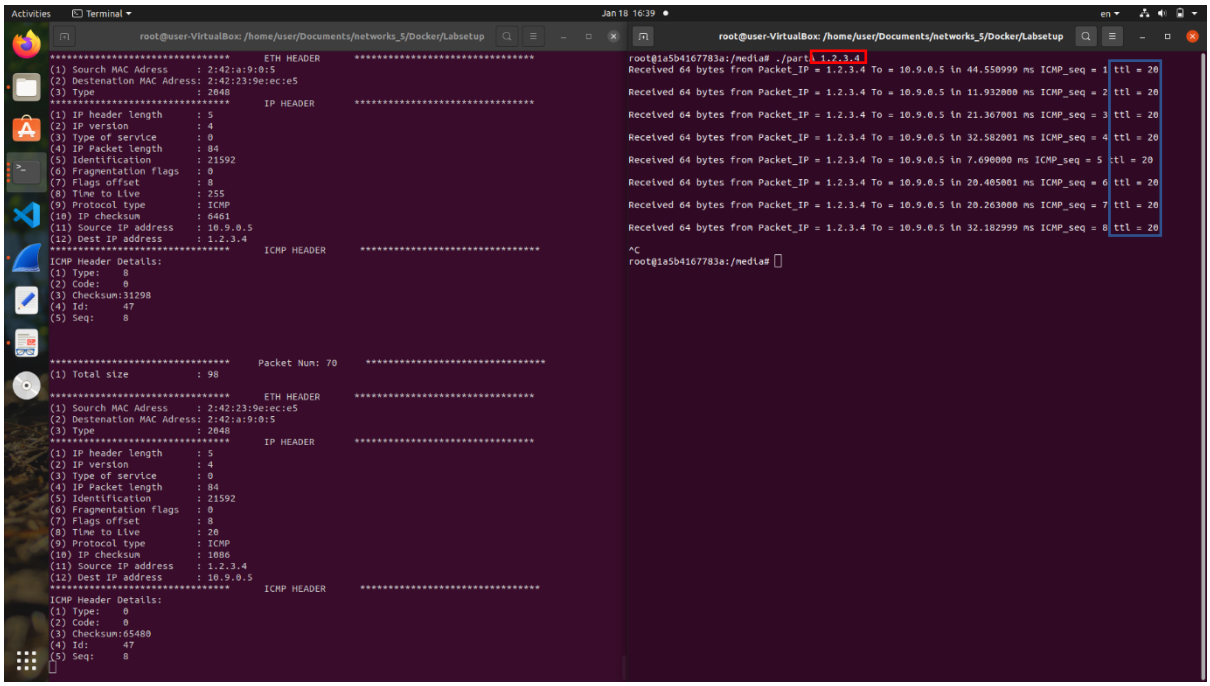
- $ ./PartA 10.9.0.6
  Ping to Host-B IP:



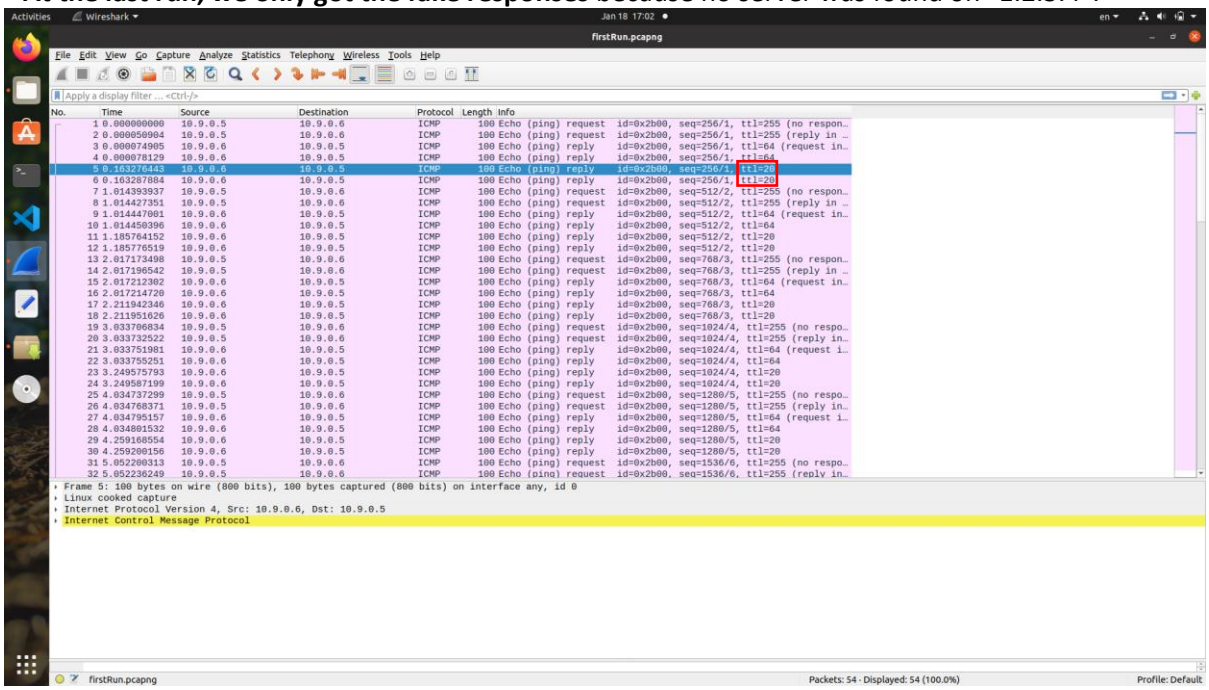
- $ ./PartA 8.8.8.8
  Ping to Google DNS IP.



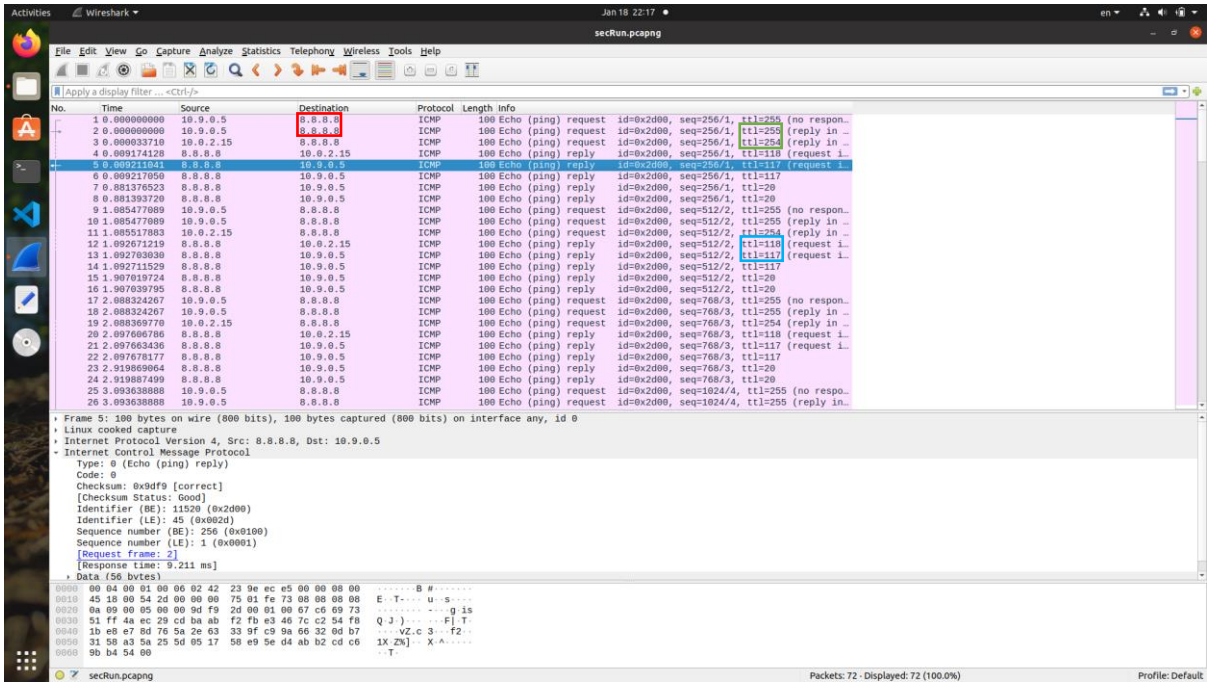
- $ ./PartA 1.2.3.4
  Ping to fake and not exist IP:



As you can notice**, at the first 2 runs** Host A got for each ping two responses, regardless to the destination. It can be easily distinguished between the real and the fake pongs thanks to the TTL field. The fake one has always 20, and real ones have higher.
At the last run, we only got the fake responses because no server was found on "1.2.3.4".

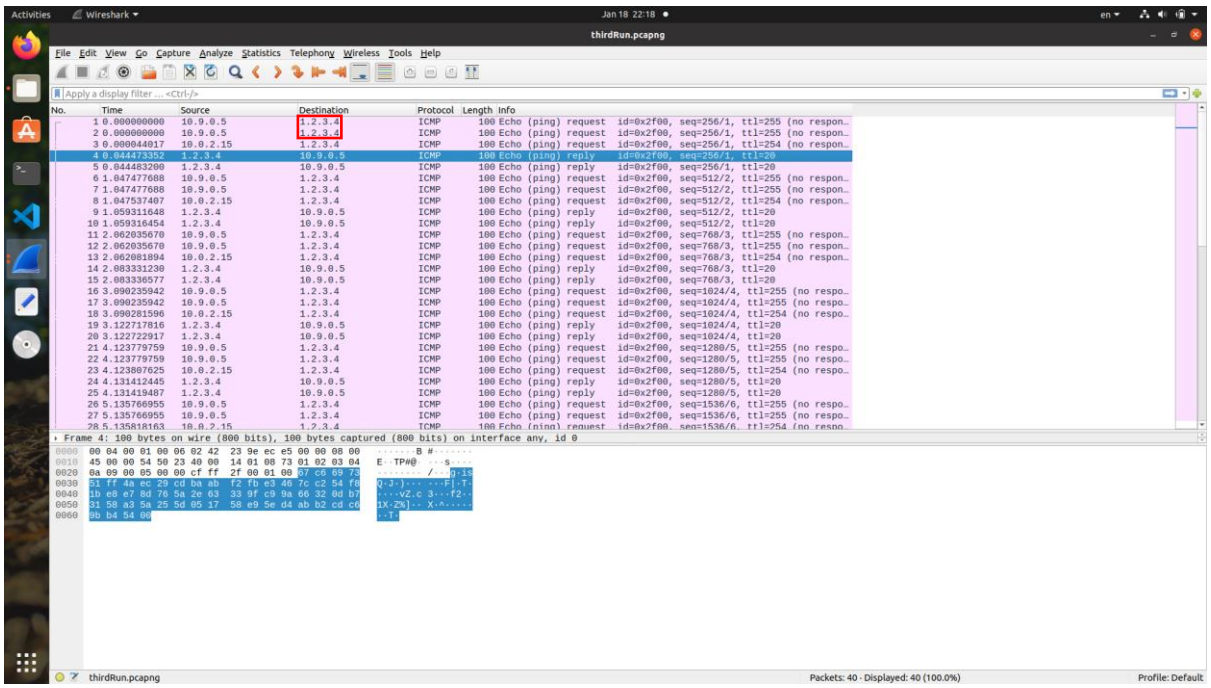


On the Wireshark we can see all the ICMP between Host A (10.9.0.5) to Host B (10.9.0.6), the fake packet can be easily seen thanks to the TTL=20. (The rest of the pcap records are attached named "firstRun" etc). Notice that we see each packet twice, our assumption is it's due to the fact that the container is using the host interface to ping outside the LAN.



Second run to 8.8.8.8.

Notice that we can see the TTL reduced by 1. This is because, in this case **our machine is functioned as router!** The first intercept is with 225 from the host, and the second one is 224 from the docker. The same happens with 118 and 117.



Third run to 1.2.3.4

Task D

To run the Gateway program using the Makefile, use the command "make gateway" in the terminal. This will compile the Gateway.c file and create an executable file named "gate".

You can then run the program by typing:

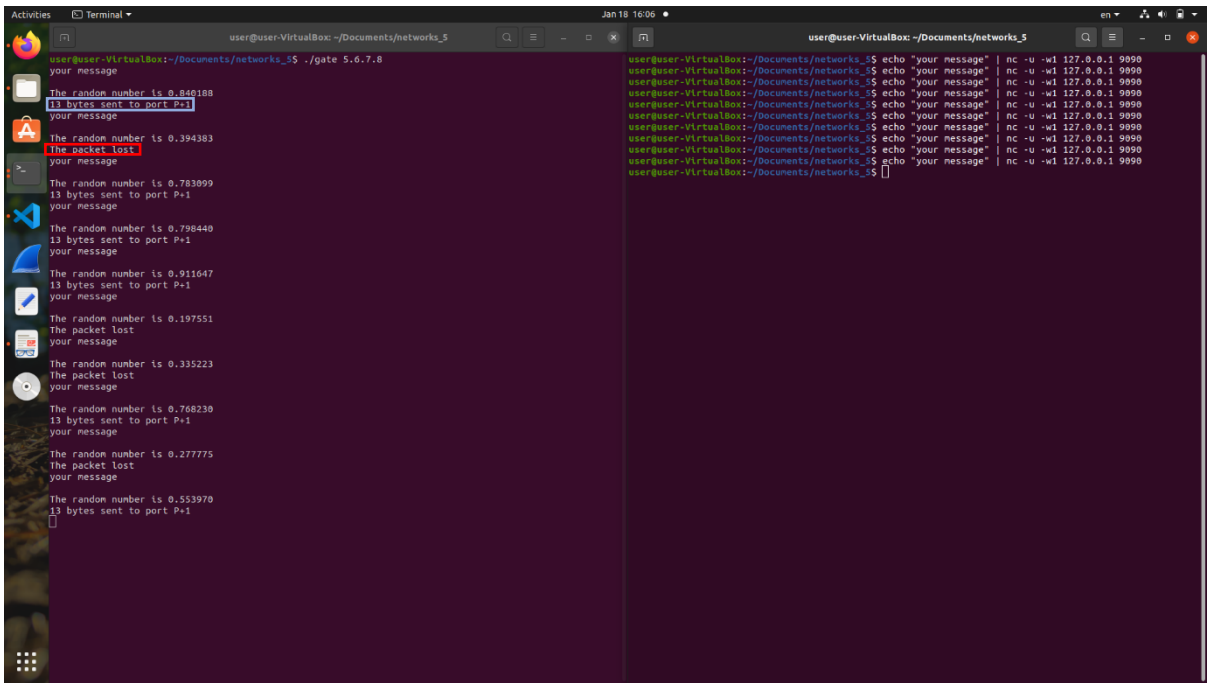
```
./gate <Host>
```

Replace <Host> with the destination.

The gateway.c program is designed to simulate bad network conditions by randomly deciding whether to send incoming messages to their intended destination.

The program creates two sockets, one for listening for incoming messages and another for sending messages. It then binds the listening socket to a specific port and address and waits for incoming messages. When a message is received, the program generates a random number between 0 and 1. If the number is greater than 0.5, the program sends the message to its intended destination with a probability of 50%. If the number is less than or equal to 0.5, the message is not sent and is dropped.

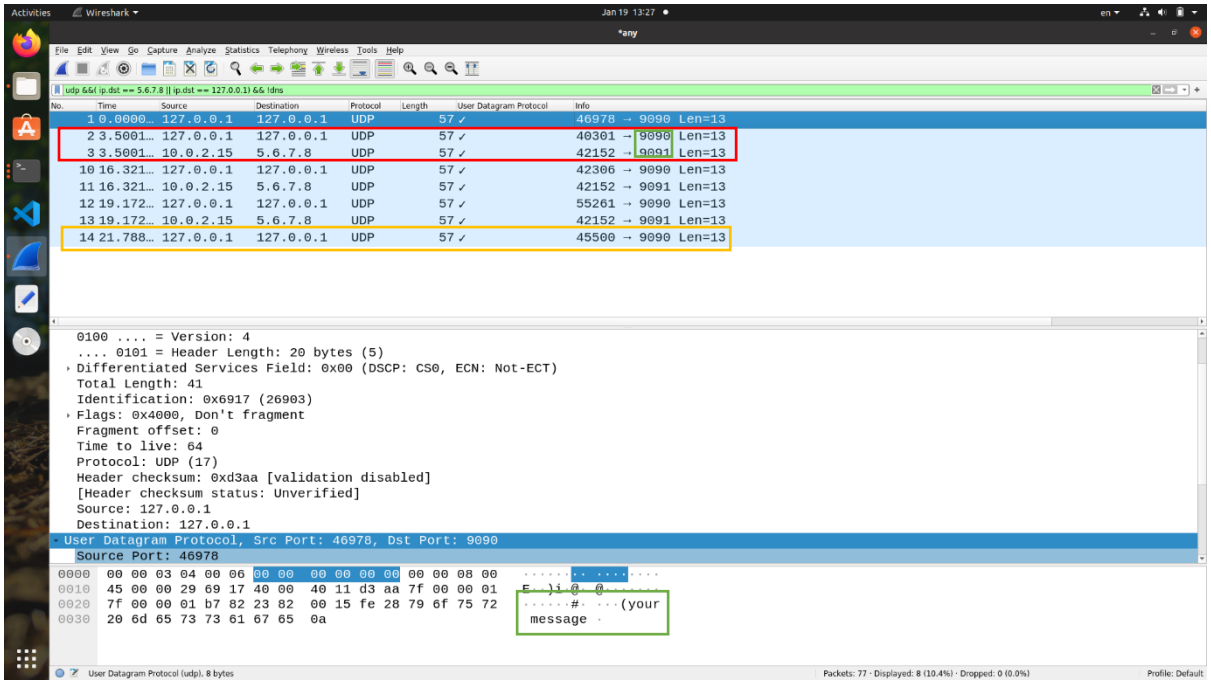
Example:



In the above pic, we see the right terminal runs net cat in order to test our gateway.

It's send a message to our local IP on UDP Protocol on port 9090 (P).

The left terminal runs the gateway and we can that with probability of 50% it passes it to "5.6.7.8" on port 9091. For example, the first try succeeds because 0.84 was picked and the second try failed though because 0.39 picked.



In this run we tried to send with the netcat tool 5 times.

1. First time failed. Mark blue.
2. Second time succeed, so we can see that we passed the msg to "5.6.7.8" on port P+1 = 9091.

   Mark red.
3. Third time succeed as well.
4. Succeed.
5. Failed. Mark gold.

Notice to the msg: "your message".

Bibliography

- https://www.geeksforgeeks.org/header-files-in-c-cpp-and-its-uses/
- https://www.binarytides.com/packet-sniffer-code-c-libpcap-linux-sockets/
- https://www.hackingloops.com/how-to-spoof-mac-address-using-c-program/
- Exercise 9 from CS moodle .