Packet Sniffing and Spoofing

CN\_Ex5

Ariel University

Connection networking

Maor Saadon 318532421

Duvi Amiram 305677494

**Contents**

1.1 About the System

1.1.1 Project overview…………………………………………………  
1.1.2 Sniffing……………………………………………………………………..4-6  
1.1.3 Spoofing……………………………………………………………………..4-6  
1.1.4 How to Install and Run the Program………………………….

1.2 Task A

1.2.1 Code Description……………………………………………………..…………8-19  
1.2.2 Flowchart  
1.2.3 Question………………………………………………………………………….22-31  
1.2.4 Output&Wireshark………………………………………………………

1.3 Task B

1.3.1 Code Description……………………………………………………..…………8-19  
1.3.2 Flowchart……………………………………………………………………  
1.3.3 Question………………………………………………………………………….22-31  
1.3.4 Output…………………………………………………………………………………..21  
1.3.5 Wireshark……………………………………………………………

1.4 Task C

1.4.1 Code Description……………………………………………………..…………8-19  
1.4.2 Flowchart……………………………………………………………..  
1.4.3 Output…………………………………………………………………………………..21  
1.4.4 Wireshark……………………………………………………………………….32-33

1.5 Task D

1.5.1 Code Description……………………………………………………..…………8-19  
1.5.2 Flowchart………………………………..  
1.5.3 Output…………………………………………………………………………………..21  
1.5.4 Wireshark……………………………………………………………………….32-33

1.6 Bibliography………………………………………………………………………..……………..48

**1.1 About the System**

1.1.1 Project overview

Packet sniffing and spoofing are two important concepts in network security; they are two major threats in network communication.

Being able to understand these two threats is essential for understanding security measures in networking.

There are many packet sniffing and spoofing tools, such as Wireshark, Tcp dump, etc.

Some of these tools are widely used by security experts, as well as by attackers.

Being able to use these tools is important for students, but what is more important for students in a network security course is to understand how these tools work, for example - how packet sniffing and spoofing are implemented in software.

* The objective of this Exe is two-fold:   
  learning to use the tools and understanding the technologies underlying these tools. For the second object, students will write simple sniffer and spoofing programs,
* and gain an in-depth understanding of the technical aspects of these programs.

This lab covers the following topics:

1. How the sniffing and spoofing work
2. Packet sniffing using the pcap library and Scapy
3. Packet spoofing using raw socket and Scapy
4. Manipulating packets using Scapy

1.1.2 Sniffing

What are packet sniffers?

Packet sniffers are applications or utilities that read data packets traversing the network within the Transmission Control Protocol/Internet Protocol (TCP/IP) layer. When in the hands of network administrators, these tools “sniff” internet traffic in real-time, monitoring the data, which can then be interpreted to evaluate and diagnose performance problems within servers, networks, hubs and applications.

When packet sniffing is used by hackers to conduct unauthorized monitoring of internet activity, network administrators can use one of several methods for detecting sniffers on the network. Armed with this early warning, they can take steps to protect data from illicit sniffers.

1.1.3 Spoofing

Packet spoofing or IP spoofing is the creation of Internet Protocol (IP) packets having a source IP address with the purpose of concealing the identity of the sender or impersonating another computing system. A spoofing attack occurs when a malicious party impersonates another device or user on a network in order to launch attacks against network hosts, steal data, spread malware, or bypass access controls.

The attacker creates an IP packet and sends it to the server, which is known as an SYN (synchronize) request. The attacker puts own source address as another computer’s IP address in the newly created IP packet. The server responds back with a SYN ACK response, which travels to the forged IP address. The attacker receives this SYN ACK response sent by the server and acknowledges it so as to complete a connection with the server. Once this is done the attacker can try various commands on the server computer. The most common methods include IP address spoofing attacks, ARP spoofing attacks, and DNS server spoofing attacks. Common measures that organizations can take for spoofing attack prevention include packet filtering, using spoofing detection software, and cryptographic network protocols.

1.1.4 How to Install and Run the Program

1. Requirements: Linux environment.
2. Download save in the same director.:
   1. Sniffer.c
   2. Spoofer.c
   3. sniff\_spoof.c
   4. Gateway.c
   5. header.h
3. Open the directory in the Linux terminal.
4. To compile and run the sniffing program use the commands:
   1. gcc Sniffer.c -o sniffer -lpcap
   2. sudo ./sniffer
5. To compile and run the spoofing program use the commands:
   1. gcc Spoofer.c -o sniffer -lpcap
   2. sudo ./ Spoofer
6. To compile and run the sniffing and spoofing program use the commands:
   1. gcc sniff\_spoof.c -o sniffer -lpcap
   2. sudo ./ sniff\_spoof
7. To compile and run the gateway program use the commands:
   1. gcc Gateway.c -o sniffer -lpcap
   2. sudo ./ Gateway

\*\*\*notice before you are running the code check that the

device are matches to your computer\*\*\*

**1.2 Task A**

1.2.1 Code Description

Sniffer.c

The sniffer output is a txt file ("id\_id.txt") of the network TCP packets traffic (like in the Wireshark), with the data that by your request about every TCP packet.

**Main() function:**

opens a live pcap session on the loopback interface with the name "lo" using pcap\_open\_live() func.

 Call the pcap\_compile() func. to compile the filter expression into a BPF (Berkeley Packet Filter) program.

The compiled filter is then set on the session with pcap\_setfilter() func. This filter is used to only capture packets that use TCP protocol and have a dst or src port of 9999 or 9998.

Text

Description automatically generated

call pcap\_loop() function to start capturing packets with got\_packet() as callback func.

Text

Description automatically generated

**got\_packet() func:**

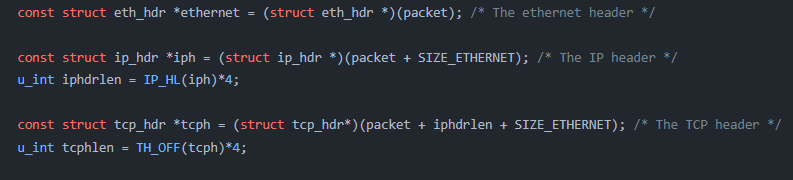
the function is called for each captured packet and is responsible for processing and printing the packet information.

Open file with the name of our ID's: "318532421\_305677494.txt" in append mode.

Text

Description automatically generated

Retrieve header info from packet:



Print packet info to file:



Close file:



1.2.2 Flowchart

**Sniffer**Sniffing the packets that transfer between the client and the proxy. Extract information from the packet and convey the packets layout into a file.

Graphical user interface, diagram, application, Word

Description automatically generated

1.2.3 Question

A sniffer program typically needs to be run with root (or administrator) privilege because it needs access to the network interface in order to capture and analyze network traffic. Without this access, the program would not be able to intercept and read the packets of data traveling over the network.

When a program is executed without root privilege, it may fail to capture network traffic because it does not have the necessary permissions to access the network interface. Additionally, the program may also be unable to access certain system-level functions or resources that it needs in order to perform its intended tasks.

Alternatively, some sniffer programs can be run with normal user privilege if they are run in promiscuous mode on a switched network, but they may only see the traffic that is directed to them by the switch.

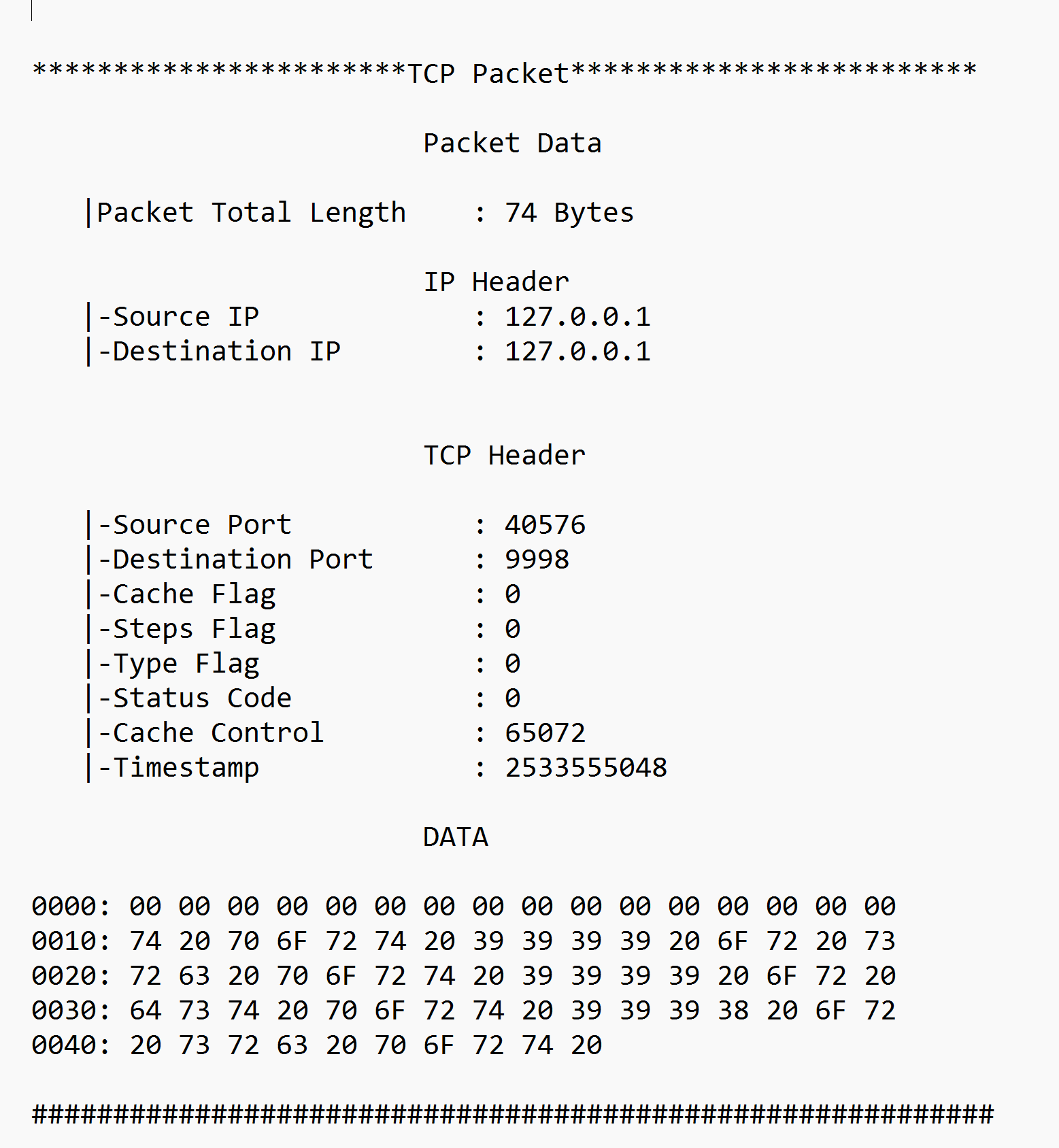
1.2.4 Output & Wireshark

To make sure that our sniffer is reliable and good, we ran the sniffer and the Wireshark together and then run Ex2 as required.

In the screenshots, you can see the information by your requested that we print. in addition, note that the information that we sniffed is the same as the information that Wireshark sniffed.

**Text

Description automatically generated with medium confidence**

**zoom in (line 1) :**

**Text

Description automatically generated**

**1.3 Task B**

1.3.1 Code Description

Spoofer.c

In this task we create spoofer for spoofing packets. Our spoofer can spoof ICMP\* packets. The spoofer fakes the sender’s IP and has a valid response. Packet spoofing or IP spoofing is the creation of Internet Protocol (IP) packets having a source IP address with the purpose of concealing the identity of the sender or impersonating another computing system. The attacker creates an IP packet and sends it to the server.

**Main() function:**

Ip verification using the check\_ip() func. (see below):

Text

Description automatically generated

Then we create an ICMP packet with ICMP and IP headers using the custom IP stored in argv[1] (in this case: "1.1.1.1"). all of the above using the icmp\_hdr and ip\_hdr structs.

Text

Description automatically generated

Printing information about the IP and ICMP headers, such as the source and destination IPs, packet length, and ICMP type:

Text

Description automatically generated

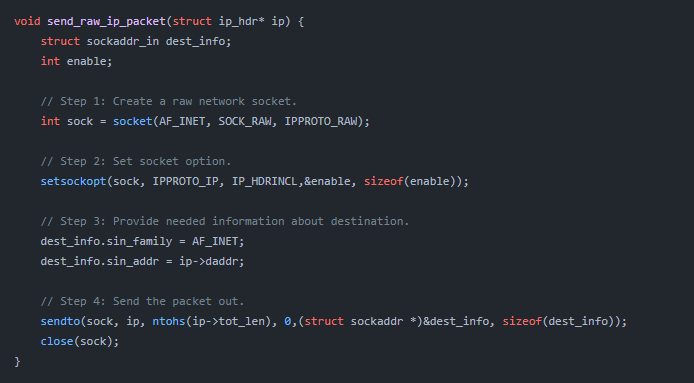
Finally, the packet is sent to the destination IP address using the send\_raw\_ip\_packet() func.

Text

Description automatically generated

**send\_raw\_ip\_packet() function:**

the function creates a raw socket and sends the inputed ip header using the sendto() function.



Check\_ip() function:

Returns false if and only if the Ip isn't valid (using the inet\_pton() func.):

Text

Description automatically generated

1.3.2 Flowchart

Spoofed packet to 1.2.3.4 from (IP)

Default IP – 1.1.1.1

Put IP to fake.

1.3.3 Question

Question 1.  
Can you set the IP packet length field to an arbitrary value, regardless of how big  
the actual packet is?

Answers:

The IP packet length field is used to indicate the total length of an IP packet, including both the header and the data payload. The value of this field is typically set automatically by the operating system or network device when the packet is created.

It is not generally recommended to set the IP packet length field to an arbitrary value, as doing so could lead to malformed packets and result in communication errors. The actual size of the packet should match the value in the length field.

However, it's possible that some network equipment or software may allow for the modification of the IP packet length field, but it's not a common practice and it could be considered as a malicious activity, as it may cause problems for network communication.

Question 2.   
Using the raw socket programming, do you have to calculate the checksum for the IP header?

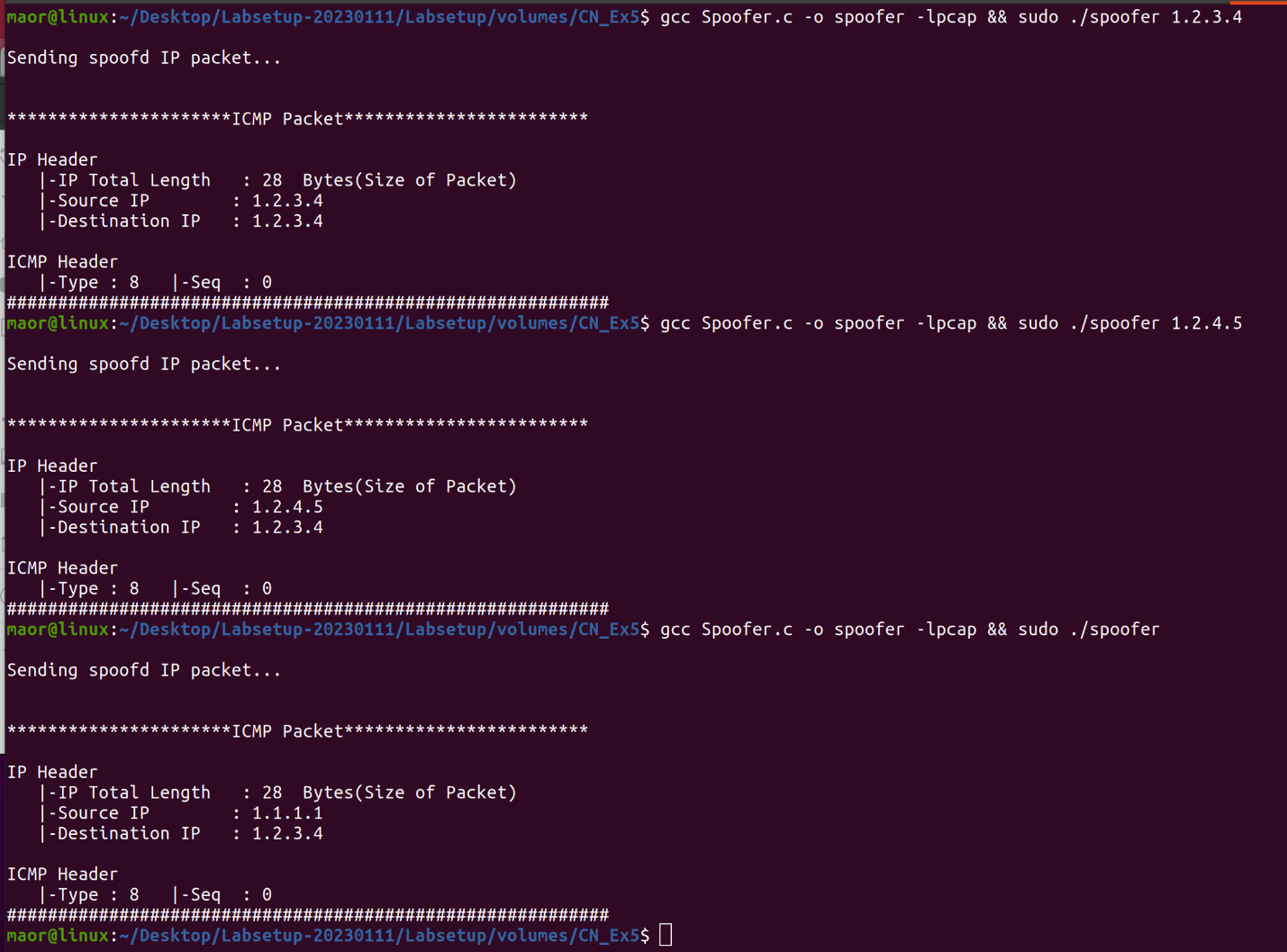
Answer:

When using raw socket programming to create and send IP packets, the checksum for the IP header is typically calculated automatically by the operating system or network device. This is done to ensure that the packet is not corrupted during transmission.

However, it is possible to manually calculate and set the IP header checksum when using raw socket programming, but it depends on the programming language and Operating system you are using. For example, in C/C++ on Linux, you can use the "IP\_HDRINCL" option to tell the operating system that you will be including your own IP header and not to calculate the checksum.

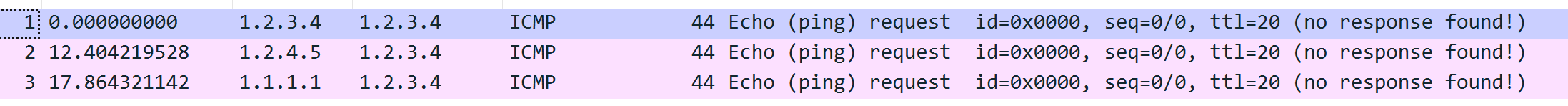
It's important to note that, if you choose to manually calculate the IP header checksum, it must be done correctly, and the checksum value must be updated if the header is modified in any way.

1.3.3 Output



In the first spoof we sent a ping (ICMP packet) from 1.2.3.4 to 1.2.3.4(default).  
In the second spoof we sent a ping from 1.2.4.5 to 1.2.3.4 (default).  
In the third spoof we didn’t put an IP so the program sent an ping from a default IP – 1.1.1.1 to 1.2.3.4 (default).

1.3.4 Wireshark



we can see in line 1 the first ping to 1.2.3.4 to 1.2.3.4, in line 2 the ping from 1.2.4.5 to 1.2.3.4 and the third ping from 1.1.1.1 to 1.2.3.4.  
we can see that the ping is an ICMP request (by your request). Also we can that there is no ping replay because we sent the ICMP request to IP address that not exist.

**1.4 Task C**

1.4.1 Code Description

In this task, we combine the sniffing and spoofing techniques to implement the following sniff-and-then-spoof program. We use two machines on the same LAN. From machine A, we ping an IP X. This generate an ICMP echo request packet. If X is alive, the ping program will receive an echo reply, and print out the response. Our sniff-and-then-spoof program runs on the attacker machine, which monitors the LAN through packet sniffing. Whenever it sees an ICMP echo request, regardless of what the target IP address is, our program immediately send out an echo reply using the packet spoofing technique.

In order to implements this program, we will use the docker. Docker is a platform for developers and sysadmins to build, ship, and run distributed applications, whether on laptops, data center VMs, or the cloud. It uses containerization technology to package and isolate applications, making them more portable and easier to manage. With Docker, developers can work in a consistent and isolated environment, and sysadmins can deploy and scale applications more efficiently.

Sniff\_spoof.c:

This is a c program designed to impersonate any server/host. It sniffs out ICMP requests and sends back and ICMP response (from the attacker side) regardless of the status of the server that was pinged.

This c program utilizes the exact same tools that we already implemented in sniffer.c and spoof.c (which is why we won't repeat the code description for these parts) with the exception that when spoofing, we do not send an ICMP packet with a custom previously determined source IP address. Instead we set the source address in the ICMP response packet, to be the destination address of the ICMP  
request that we have sniffed:

Text

Description automatically generated

1.4.2 Flowchart

HOST B / GOOGLE / SFAKE IP

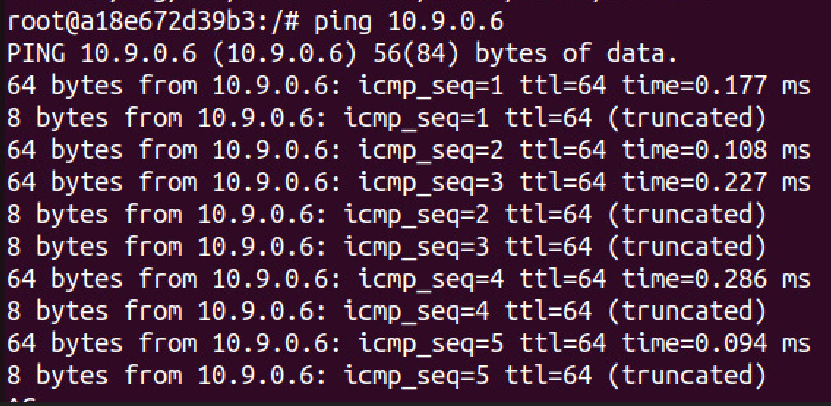
HOST A

**mohahahaha**

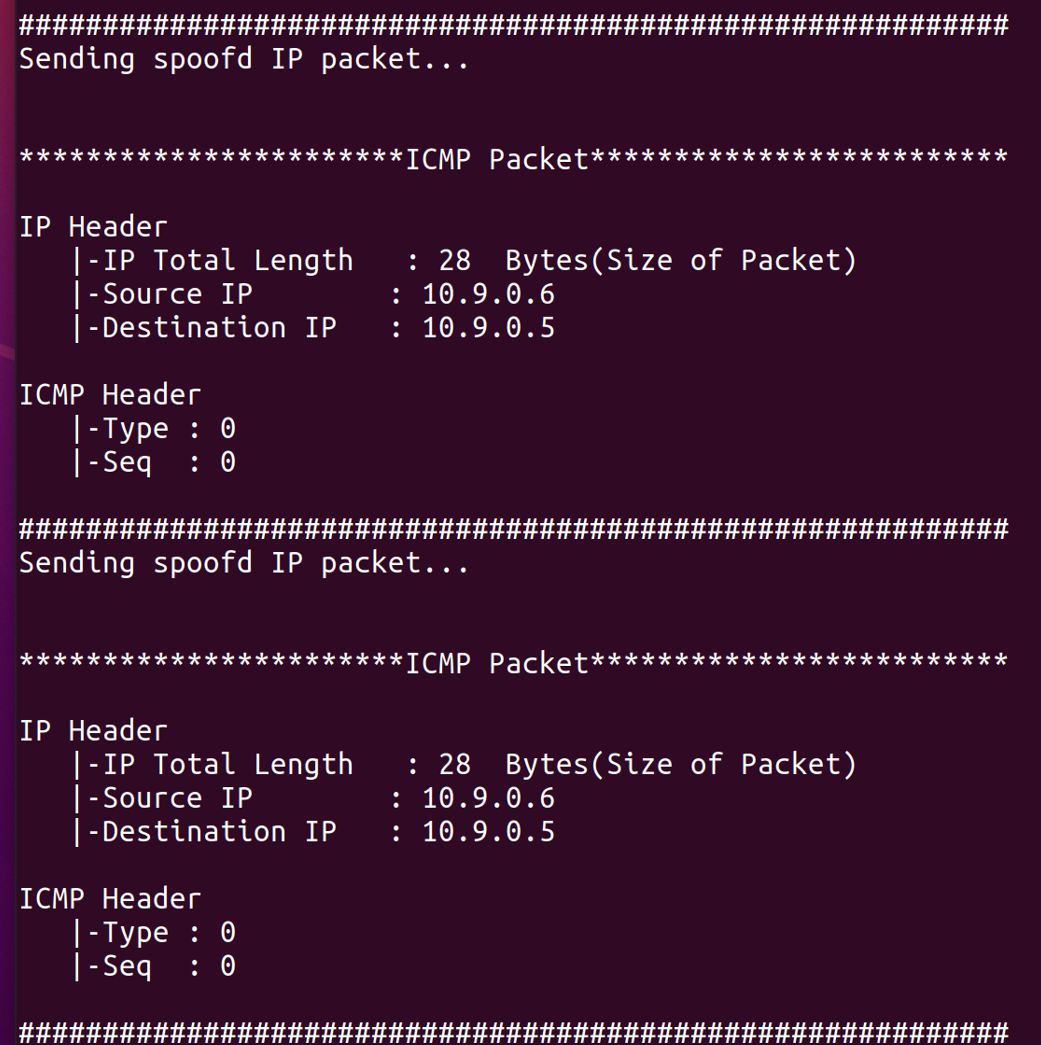
ATTACKER

1.4.3 Output

1.Host A 🡪 Host B

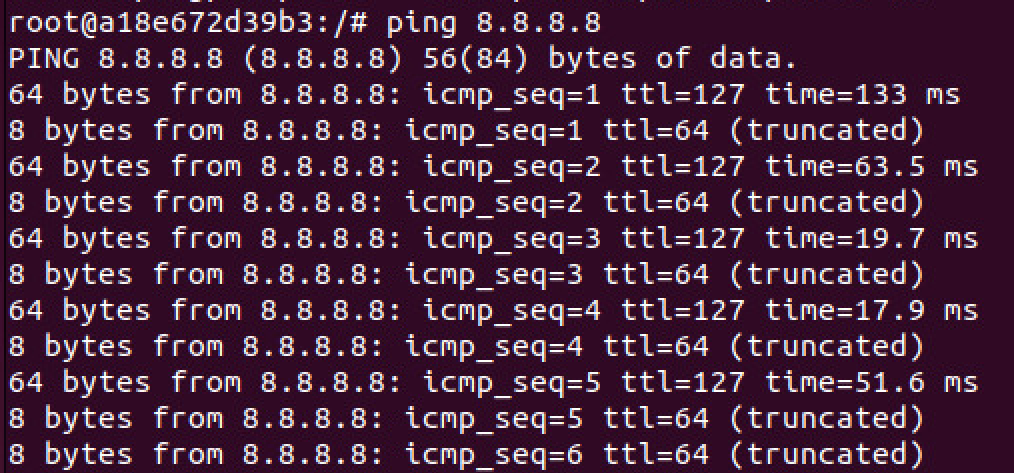


**HOST A SEND   
PING TO HOST B**



**TERMINAL –   
THE FAKE PING**

2.Host A 🡪 GOOGLE



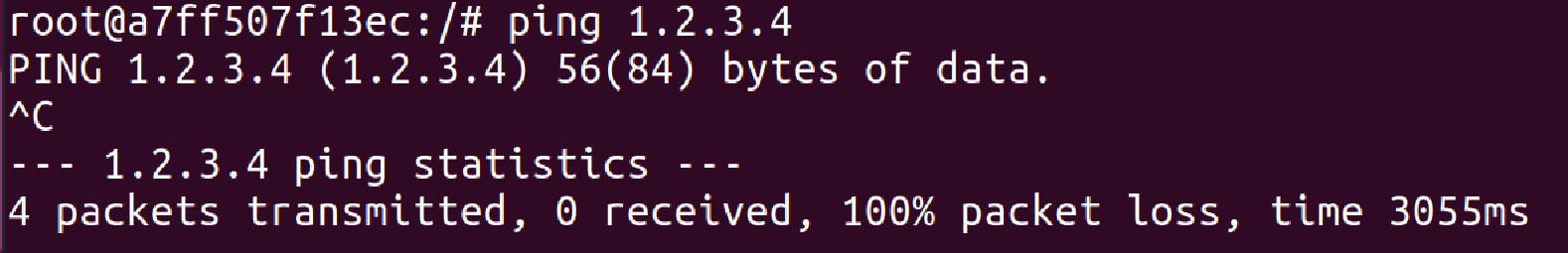
**HOST A SEND   
PING TO 8.8.8.8**

Text

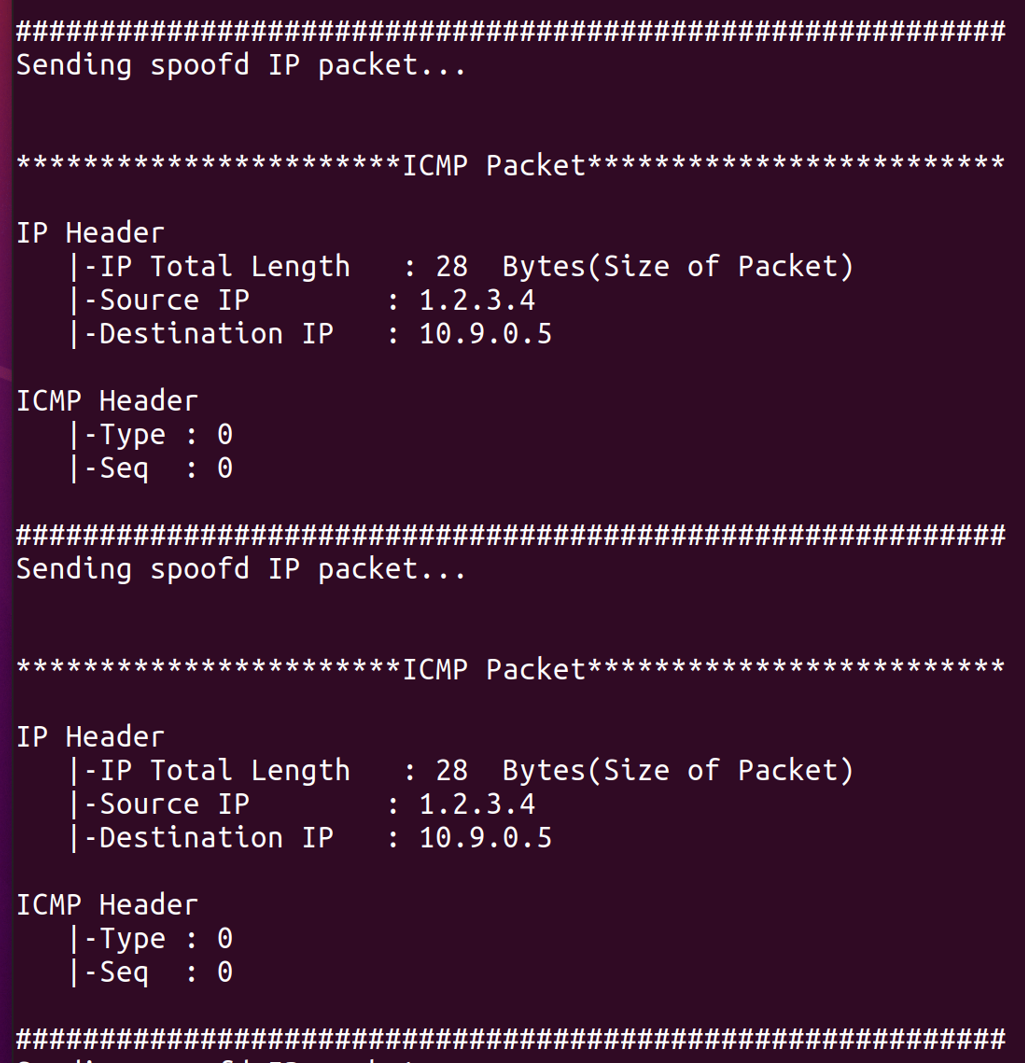
Description automatically generated

**TERMINAL –   
THE FAKE PING**

3.Host A 🡪 FAKE IP



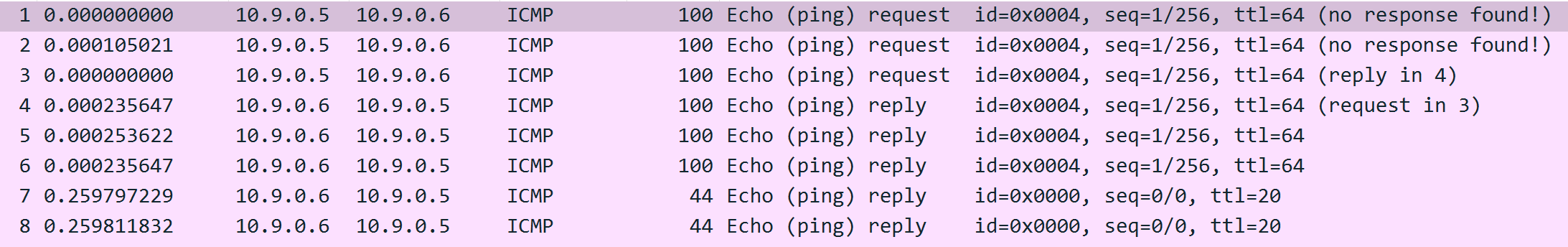
**HOST A SEND   
PING TO 1.2.3.4**



**TERMINAL –   
THE FAKE PING**

1.4.4 Wireshark

1.Host A 🡪 Host B



1. Duplicate PING request with no response from host A to host B

2. “\_\_\_\_\_\_\_\_\_”

3. Ping request from A to B.

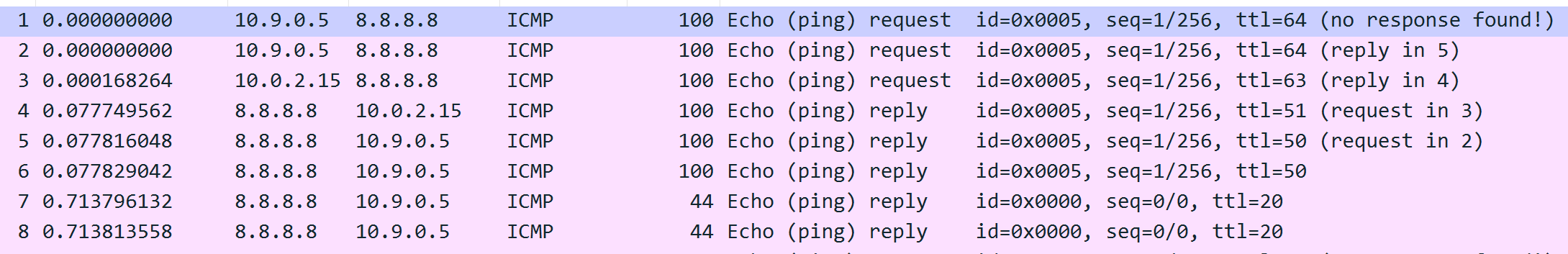
4. Ping reply from the real B.

5. “\_\_\_\_\_\_\_\_\_”

6. “\_\_\_\_\_\_\_\_\_”

7. Our fake ping reply to host A we can see that its our ping because it 20 ttl like we define.

2.Host A 🡪 GOOGLE



Duplicate with no response from host A to GOOGLE (8.8.8.8)

1. Ping request from A to google.

2. “\_\_\_\_\_\_\_\_\_”

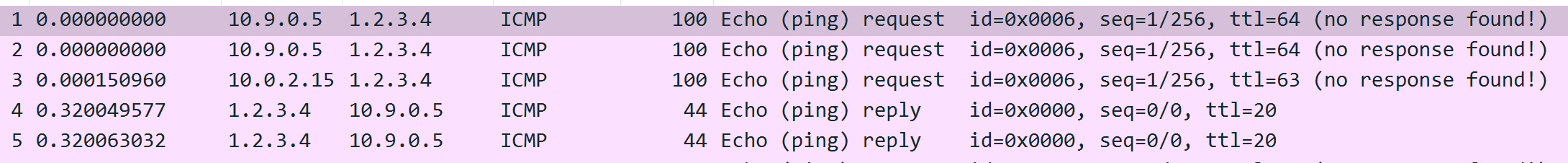
3. Ping reply from the real GOOGLE.

4. “\_\_\_\_\_\_\_\_\_”

6. “\_\_\_\_\_\_\_\_\_”

7. Our fake ping reply to host A (20 ttl).

3.Host A 🡪 FAKE IP



1. Ping request from A to fake IP (1.2.3.4).

2. “\_\_\_\_\_\_\_\_\_” .

4. “\_\_\_\_\_\_\_\_\_”

6. Our fake ping reply to host A (20 ttl).

7. “\_\_\_\_\_\_\_\_\_”

6. “\_\_\_\_\_\_\_\_\_”

**1.5 Task D**

1.5.1 Code Description

This task takes the name of a host on the command line and creates a datagram socket to that host (using port number P+1), it also creates another datagram socket where it can receive datagrams from any host on port number P; next, it enters an infinite loop in each iteration of which it receives a datagram from port P, then samples a random number using if the number obtained is greater than 0.5, the datagram received is forwarded onto the outgoing socket to port P+1, otherwise the datagram is discarded and the process goes back to waiting for another incoming datagram. Note that this gateway will simulate an unreliable network that loses datagrams with 50% probability. In order to check our Gateway we made another task that sent UDP packet through the Gate.

Gateway.c:

Declaring 2 ints for sending socket (sockIP) and receiving socket (sockAny).

Both are datagram sockets (for UDP traffic)

Then declaring 2 sockadrr\_in structs for the details (ports to bind and so forth) of the two sockets.

A screenshot of a computer

Description automatically generated with medium confidence

Main():

Retrieve IP from user:

Text

Description automatically generated

Create sending socket:

Text

Description automatically generated

Create receiving port and binding to port 50000:

Text

Description automatically generated

Text

Description automatically generatedInfinite loop for sending received packets:

The loop continuously receives UDP packets using the recvfrom function and checks the random number generated, if the number is greater than 0.5 it calls send\_udp\_packet() func. to send the packet to the specified IP address on port 50001. If the number is less than or equal to 0.5, the gateway discards the packet.

Text

Description automatically generated

Close the sockets:

Text

Description automatically generated  
  
  
Function for sending the UDP packets:

Text

Description automatically generated

This function takes the packet, its size, and the destination IP address as input. It sets the destination address and port number, then it sends the packet using the sendto function. Finally, it prints a message indicating whether the packet was sent successfully or not.

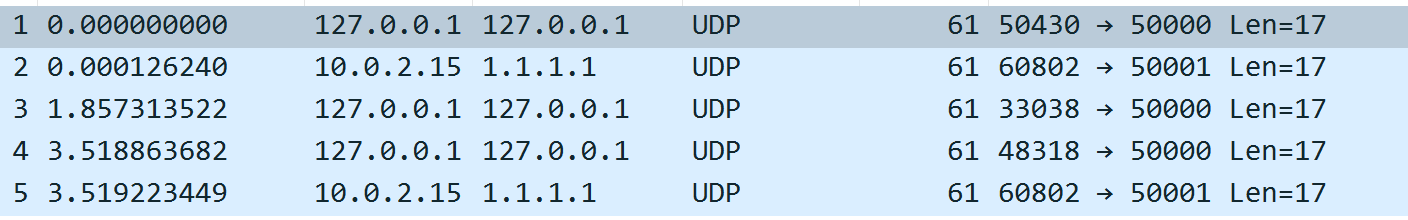
1.5.2 Flowchart

1.5.3 Output



We sent three UDP packet, notice that we mast put an IP in order to make the program work.

1.5.4 Wireshark

****

1. Send the packet to IP (1.1.1) .

2. Receive UDP packet on Port 5001 .

3. Send the packet to IP (1.1.1.1) . – didn’t succeed

4. Send the packet to IP (1.1.1.1) .

5. Receive UDP packet on Port 5001 .

**1.6 Header.h**

This header file defines several structs that represent different parts of a network packet.

1. The struct "eth\_hdr" represents an Ethernet header, which contains the destination and source host addresses (6 bytes each) and the type of packet (IP, ARP, RARP, etc.).
2. The struct "ip\_hdr" represents an IP header, which contains information such as the IP version, header length, time to live, protocol, and source and destination addresses.
3. The struct "tcp\_hdr" represents a TCP header, which contains information such as the source and destination ports, sequence and acknowledgement numbers, and flags (such as SYN, ACK, and RST).
4. The struct "icmp\_hdr" represents an ICMP header, which contains information such as the message type, error code, checksum, and identification and sequence numbers.
5. The struct "app\_hdr" represents an application header, which contains information such as a timestamp, total length, flags, cache control and padding.

**1.7 Bibliography**

<https://sphirewall.sourceforge.net/data/docs/structsniff__ip.html#d2db4a1d3fbfb1bcc44e5a26d6c28c2e>

<https://elf11.github.io/2017/01/22/libpcap-in-C.html>

<https://www.iana.org/assignments/icmp-parameters/icmp-parameters.xhtml#icmp-parameters-codes-1>

<https://cpp.hotexamples.com/examples/-/-/pcap_compile/cpp-pcap_compile-function-examples.html>

google

man in wsl