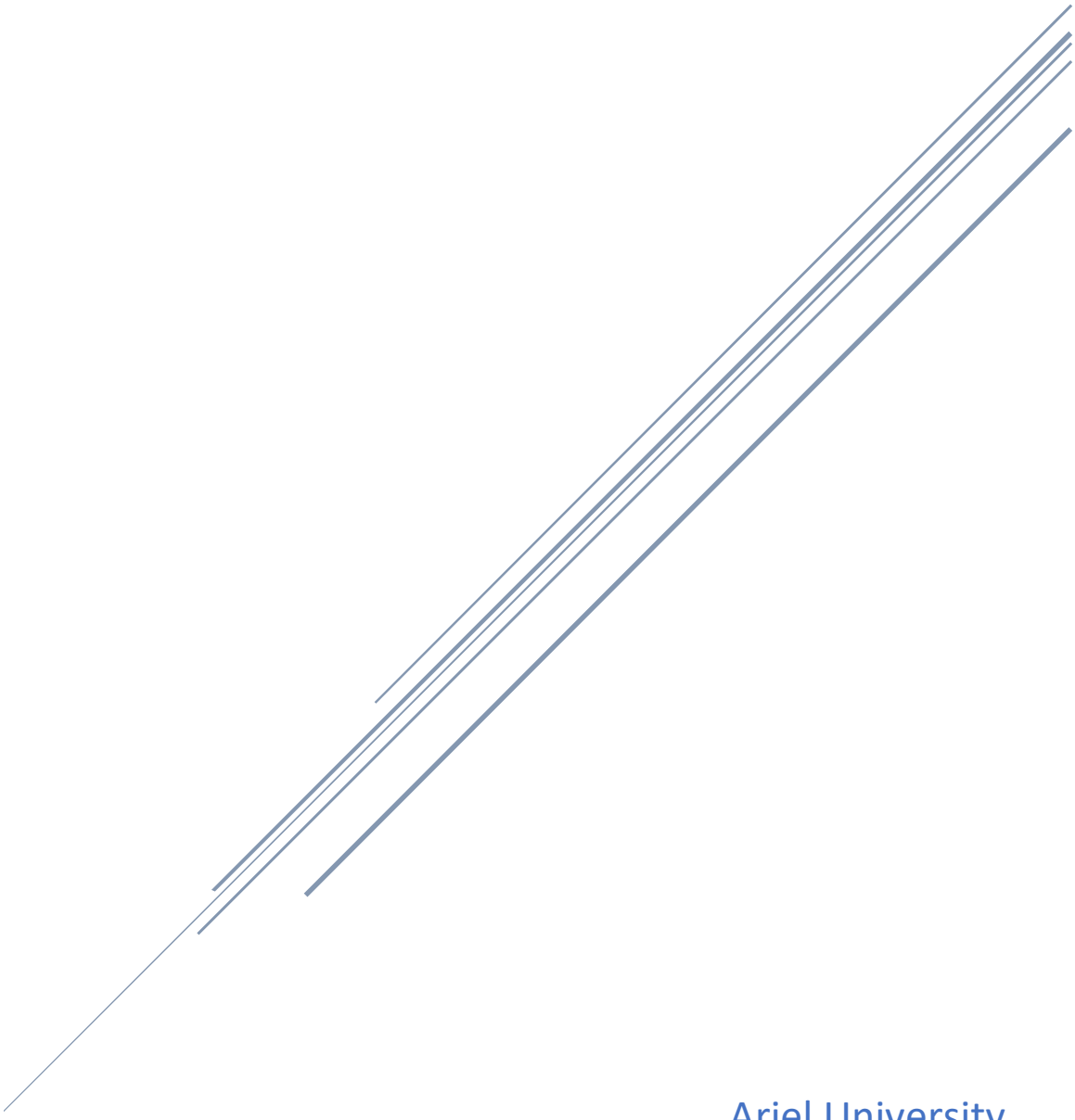
PACKET SNIFFING AND SPOOFING

CN_Ex5

Maor Saadon 318532421

Duvi Amiram 305677494



Ariel University
Connection networking

Contents

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.:
    a. Sniffer.c
    b. Spoofer.c
    c. sniff_spoof.c
    d. Gateway.c
    e. header.h

3. Open the directory in the Linux terminal.
4. To compile and run the sniffing program use the commands:
    a. gcc Sniffer.c -o sniffer -lpcap
    b. sudo ./sniffer
5. To compile and run the spoofing program use the commands:
    a. gcc Spoofer.c -o sniffer -lpcap
    b. sudo ./ Spoofer
6. To compile and run the sniffing and spoofing program use the commands:
    a. gcc sniff_spoof.c -o sniffer -lpcap
    b. sudo ./ sniff_spoof
7. To compile and run the gateway program use the commands:
    a. gcc Gateway.c -o sniffer -lpcap
    b. 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.

```
//Open live pcap session on NIC with name lo
handle = pcap_open_live(nameDevice, BUFSIZ, 1, 1000, errbuf);
```

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.

```
//Compile filter_exp into BPF psuedo-code
pcap_compile(handle, &fp, filter_exp, 0, net);
pcap_setfilter(handle, &fp);
```

call pcap_loop() function to start capturing packets with got_packet() as callback func.

```
//Capture packets
pcap_loop(handle, -1, got_packet, NULL);

//Close the handle
pcap_close(handle);
return 0;
```

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.

```
FILE *output;
output = fopen("318532421_305677494.txt","a");
if(output==NULL)
{
    printf("Unable to create file.\n");
}
```

Retrieve header info from packet:

```
const struct eth_hdr *ethernet = (struct eth_hdr *)(packet); /* The ethernet header */

const struct ip_hdr *iph = (struct ip_hdr *)(packet + SIZE_ETHERNET); /* The IP header */
u_int iphdrlen = IP_HL(iph)*4;

const struct tcp_hdr *tcph = (struct tcp_hdr*)(packet + iphdrlen + SIZE_ETHERNET); /* The TCP header */
u_int tcphlen = TH_OFF(tcph)*4;
```

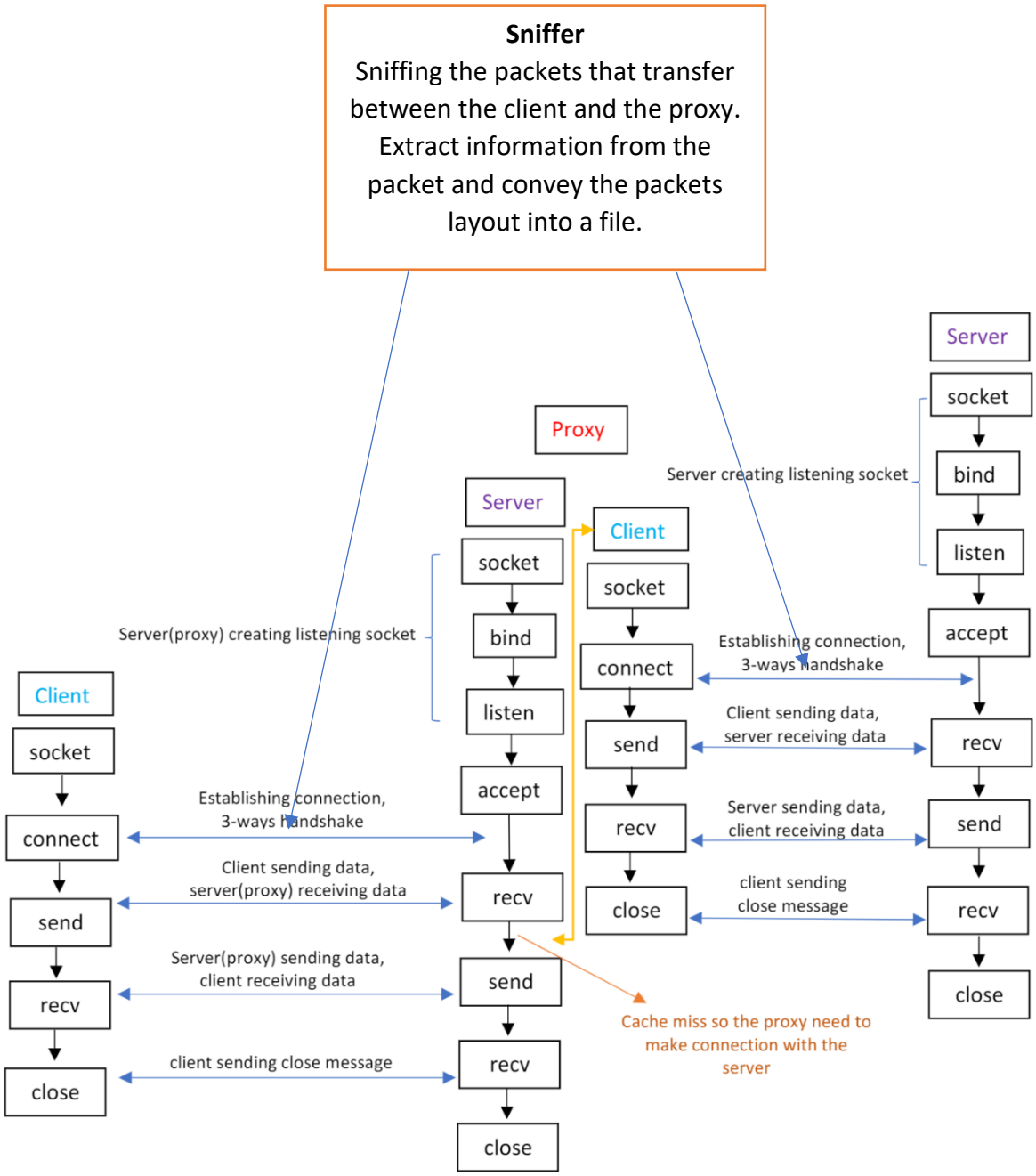
Print packet info to file:

```
fprintf(output , "\n\n**********************TCP Packet************************\n");

fprintf(output , "\n                        Packet Data                          \n\n");
fprintf(output , "   |Packet Total Length    : %u Bytes\n",header->len);

//IP Header
fprintf(output , "\n                        IP Header                          \n");
fprintf(output , "   |-Source IP             : %s\n" , inet_ntoa(iph->saddr) );
fprintf(output , "   |-Destination IP        : %s\n" , inet_ntoa(iph->daddr) );
fprintf(output , "\n");

//TCP Header
fprintf(output , "\n                        TCP Header                          \n\n");
fprintf(output , "   |-Source Port           : %u\n",ntohs(tcph->source));
fprintf(output , "   |-Destination Port      : %u\n",ntohs(tcph->dest));
fprintf(output , "   |-Cache Flag            : %u\n",tcph->cache_flag);
fprintf(output , "   |-Steps Flag            : %u\n",tcph->steps_flag);
fprintf(output , "   |-Type Flag             : %u\n",tcph->type_flag);
fprintf(output , "   |-Status Code           : %u\n",tcph->status_code);
fprintf(output , "   |-Cache Control         : %u\n",ntohs(tcph->check));
fprintf(output , "   |-Timestamp             : %u\n", ntohl(tcph->timestamp));
fprintf(output , "\n                        DATA                          \n");

uint8_t *data = (uint8_t *)malloc(sizeof(uint8_t) * header->len);
if (!data)
{
    return;
}

for (int i = 0; i < header->len; i++)
{
    if (!(i & 15)){
        fprintf(output, "\n%04X: ", i);
    }
    fprintf(output, "%02X ", (uint8_t)data[i]);
}

fprintf(output, "\n");

fprintf(output , "\n###########################################################");
```

Close file:

```
fclose(output);
```

1.2.2 Flowchart



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.

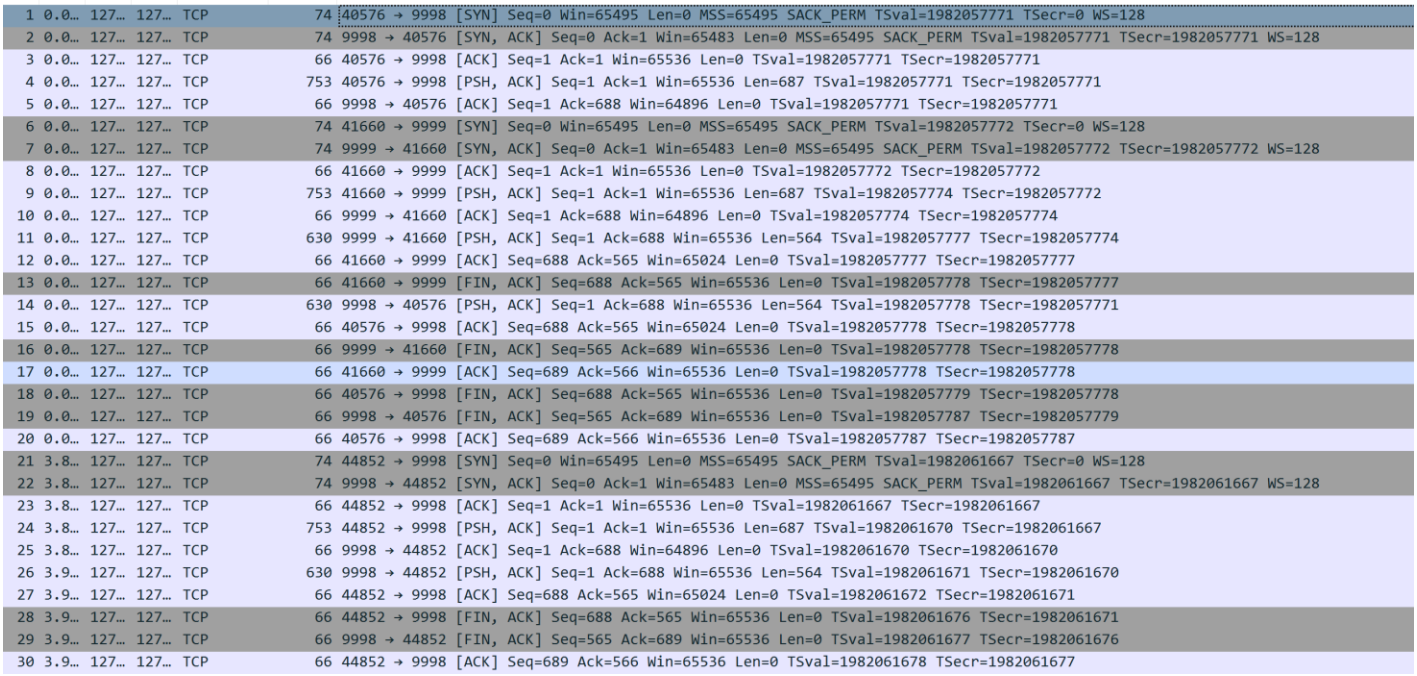
When a program is executed without root privilege, it may fail to capture network traffic because it does not have the necessary permissions to access the network interface. Additionally, the program may also be unable to access certain system-level functions or resources that it needs in order to perform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

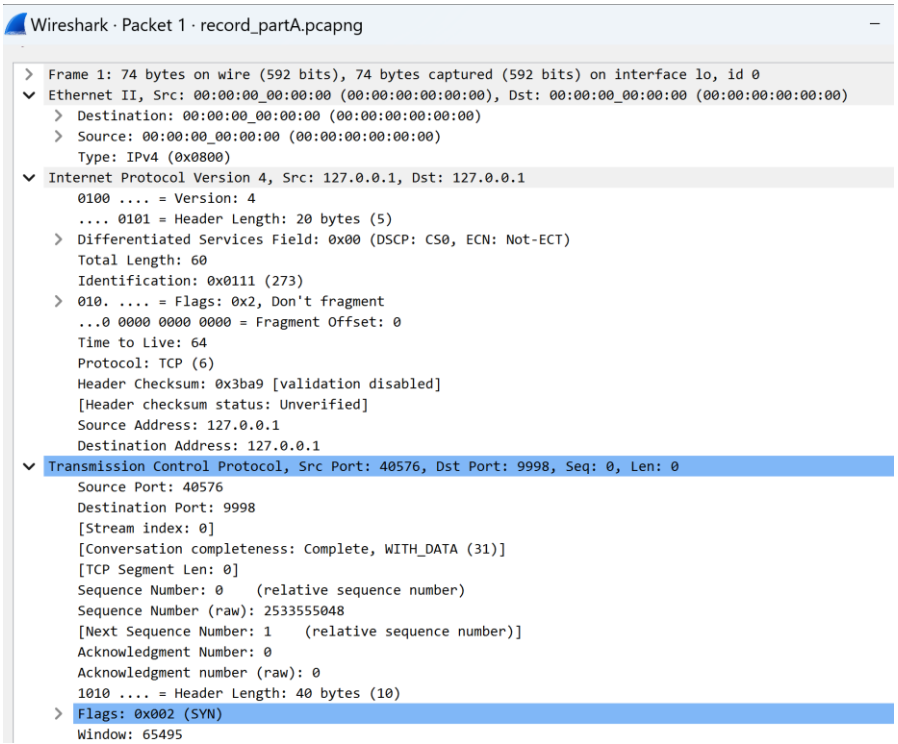
To make sure that our sniffer is reliable and good, we ran the sniffer and the Wireshark together and then run Ex2 as required.

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.



zoom in (line 1) :

```
**********************TCP Packet*************************

                        Packet Data

   |Packet Total Length    : 74 Bytes

                        IP Header
   |-Source IP             : 127.0.0.1
   |-Destination IP        : 127.0.0.1


                        TCP Header

   |-Source Port           : 40576
   |-Destination Port      : 9998
   |-Cache Flag            : 0
   |-Steps Flag            : 0
   |-Type Flag             : 0
   |-Status Code           : 0
   |-Cache Control         : 65072
   |-Timestamp             : 2533555048

                        DATA

0000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0010: 74 20 70 6F 72 74 20 39 39 39 39 20 6F 72 20 73
0020: 72 63 20 70 6F 72 74 20 39 39 39 39 20 6F 72 20
0030: 64 73 74 20 70 6F 72 74 20 39 39 39 38 20 6F 72
0040: 20 73 72 63 20 70 6F 72 74 20

###########################################################
```



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):

```
if(!check_ip(argv[1]))
{
    printf(" Sorry but this is not a valid ip please try again. \n");
    exit(1);

}
```

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.

```
struct icmp_hdr *icmp = (struct icmp_hdr *)(buffer + sizeof(struct ip_hdr));
icmp->type = 8;
icmp->checksum = calculate_checksum((unsigned short *)icmp,sizeof(struct icmp_hdr));

struct ip_hdr *ip = (struct ip_hdr *) buffer;
ip->version = 4;
ip->ihl = 5;
ip->ttl = 20;
ip->saddr.s_addr = inet_addr(argv[1]);
ip->daddr.s_addr = inet_addr("1.2.3.4");
ip->protocol = IPPROTO_ICMP;
ip->tot_len= htons(sizeof(struct ip_hdr) + sizeof(struct icmp_hdr));
```

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

```
printf("\nSending spoofd IP packet...\n");
printf("\n\n**********************ICMP Packet************************\n");

printf("\nIP Header\n");
printf("   |-IP Total Length   : %d  Bytes(Size of Packet)\n", ntohs(ip->tot_len));
printf("   |-Source IP         : %s\n", inet_ntoa(ip->saddr));
printf("   |-Destination IP    : %s\n", inet_ntoa(ip->daddr));

printf("\nICMP Header\n");
printf("   |-Type : %d", (unsigned int) (icmp->type));
printf("   |-Seq  : %d", (unsigned int) (icmp->seq));

printf("\n############################################################\n");
```

Finally, the packet is sent to the destination IP address using the send_raw_ip_packet() func.

```
  printf("\n############################################################\n");

  send_raw_ip_packet(ip);

  return 0;
```

send_raw_ip_packet() function:

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

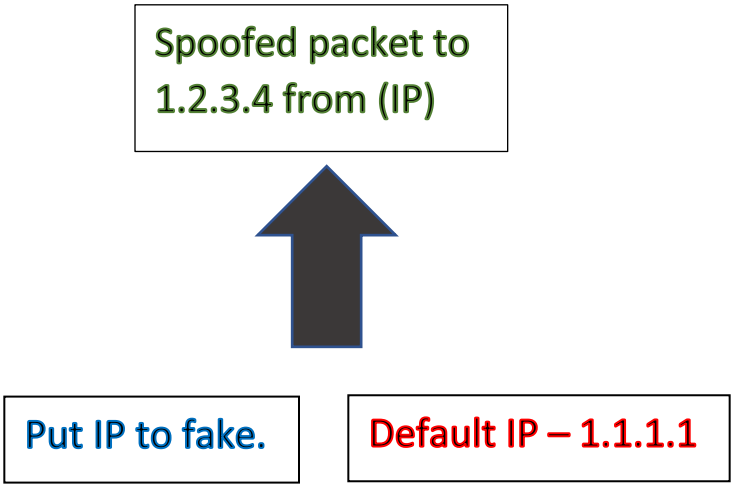
```
void send_raw_ip_packet(struct ip_hdr* ip) {
    struct sockaddr_in dest_info;
    int enable;

    // Step 1: Create a raw network socket.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

    // Step 2: Set socket option.
    setsockopt(sock, IPPROTO_IP, IP_HDRINCL,&enable, sizeof(enable));

    // Step 3: Provide needed information about destination.
    dest_info.sin_family = AF_INET;
    dest_info.sin_addr = ip->daddr;

    // Step 4: Send the packet out.
    sendto(sock, ip, ntohs(ip->tot_len), 0,(struct sockaddr *)&dest_info, sizeof(dest_info));
    close(sock);
}
```

Check_ip() function:

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

```
int check_ip(char *ip){
    struct sockaddr_in sock_in;
    int check = inet_pton(AF_INET , ip , &(sock_in.sin_addr));
    return check!=0;
}
```

1.3.2 Flowchart



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?

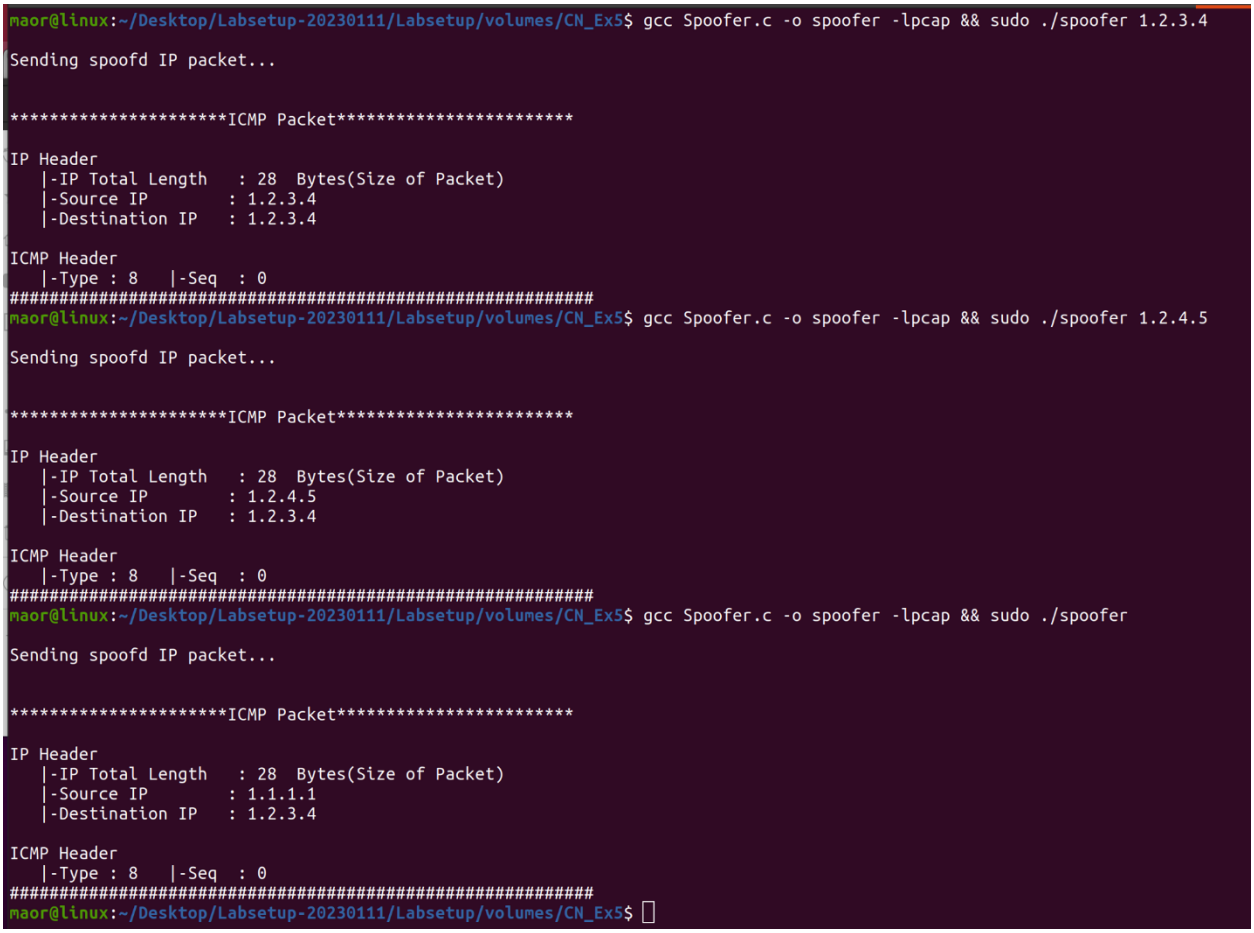
Answer:

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

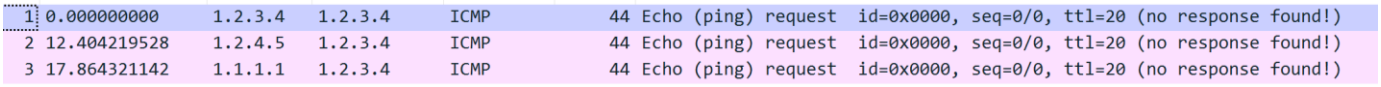


In the first spoof we sent a ping (ICMP packet) from 1.2.3.4 to 1.2.3.4(default).

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



1	0.000000000	1.2.3.4	1.2.3.4	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=20 (no response found!)
2	12.404219528	1.2.4.5	1.2.3.4	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=20 (no response found!)
3	17.864321142	1.1.1.1	1.2.3.4	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=20 (no response found!)

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.

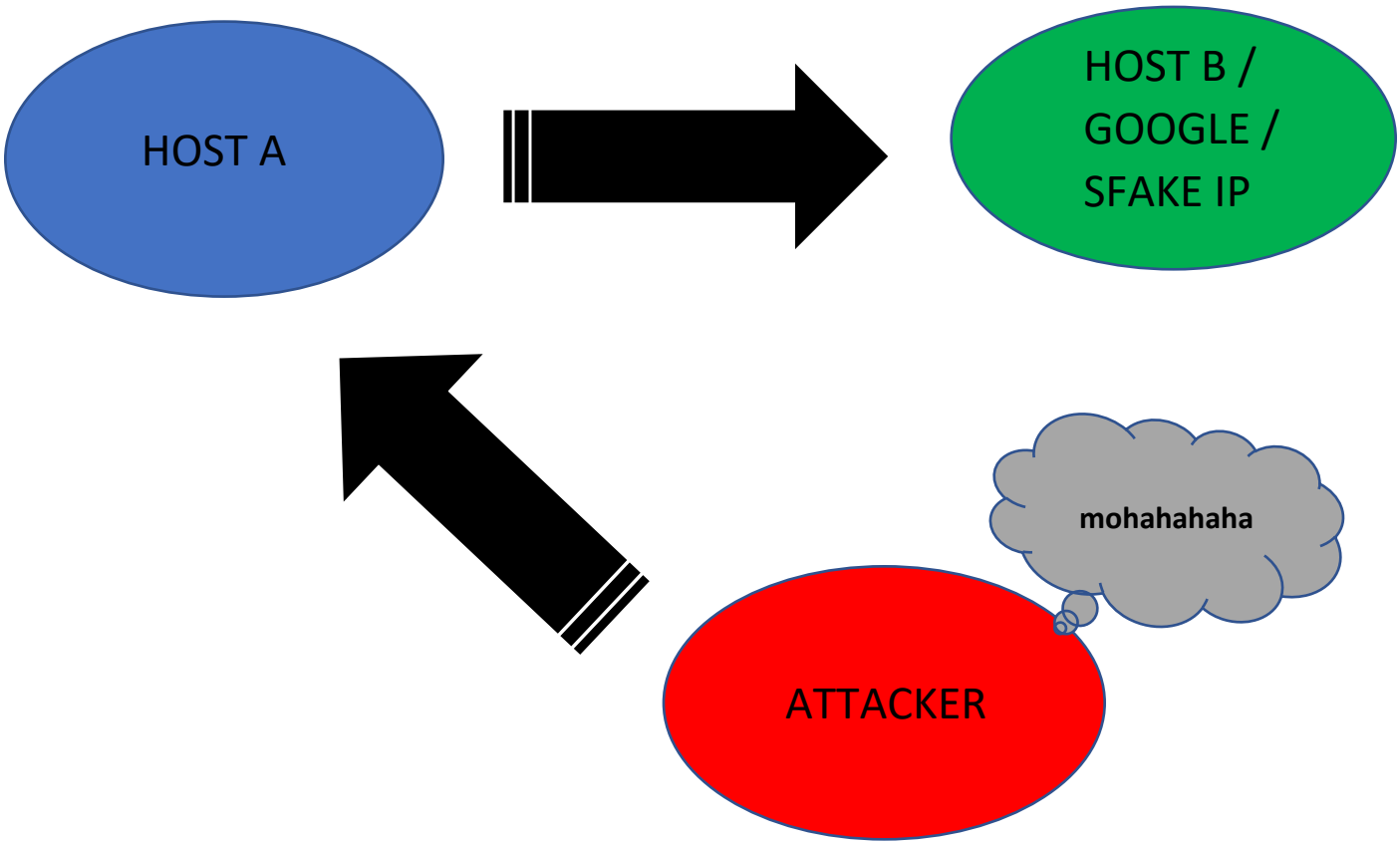
Sniff_spoof.c:

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:

```
void send_echo_reply(struct ip_hdr * ip) {

    const char buffer[PACKET_LEN];
    memset((char *) buffer, 0, PACKET_LEN);


    struct icmp_hdr *new_icmp = (struct icmp_hdr *)(buffer + sizeof(struct ip_hdr));
    new_icmp->type = 0;
    new_icmp->checksum = calculate_checksum((unsigned short *)new_icmp,sizeof(struct icmp_hdr));

    struct ip_hdr *new_ip = (struct ip_hdr *) buffer;
    new_ip->version = 4;
    new_ip->ihl = 5;
    new_ip->ttl = 20;
    new_ip->saddr.s_addr = inet_addr(inet_ntoa(ip->daddr));
    new_ip->daddr.s_addr= inet_addr(inet_ntoa(ip->saddr));
    new_ip->protocol = IPPROTO_ICMP;
    new_ip->tot_len= htons(sizeof(struct ip_hdr) + sizeof(struct icmp_hdr));

    // Fill in the ICMP header
    //ICMP type 8 for request and 0 for replay
    new_icmp->type = 0;

    // Calculate checksum
    new_icmp->checksum = 0;
    new_icmp->checksum = calculate_checksum((unsigned short *) new_icmp, sizeof(struct icmp_hdr));


    send_raw_ip_packet (new_ip, new_icmp);
}
```

1.4.2 Flowchart



1.4.3 Output

1.Host A → Host B

```
root@a18e672d39b3:/# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.177 ms
8 bytes from 10.9.0.6: icmp_seq=1 ttl=64 (truncated)
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.108 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.227 ms
8 bytes from 10.9.0.6: icmp_seq=2 ttl=64 (truncated)
8 bytes from 10.9.0.6: icmp_seq=3 ttl=64 (truncated)
64 bytes from 10.9.0.6: icmp_seq=4 ttl=64 time=0.286 ms
8 bytes from 10.9.0.6: icmp_seq=4 ttl=64 (truncated)
64 bytes from 10.9.0.6: icmp_seq=5 ttl=64 time=0.094 ms
8 bytes from 10.9.0.6: icmp_seq=5 ttl=64 (truncated)
```

HOST A SEND PING TO HOST B

TERMINAL – THE FAKE PING

```
############################################################
Sending spoofd IP packet...


**********************ICMP Packet************************

IP Header
   |-IP Total Length   : 28  Bytes(Size of Packet)
   |-Source IP         : 10.9.0.6
   |-Destination IP    : 10.9.0.5

ICMP Header
   |-Type : 0
   |-Seq  : 0

############################################################
Sending spoofd IP packet...


**********************ICMP Packet************************

IP Header
   |-IP Total Length   : 28  Bytes(Size of Packet)
   |-Source IP         : 10.9.0.6
   |-Destination IP    : 10.9.0.5

ICMP Header
   |-Type : 0
   |-Seq  : 0

############################################################
```

2.Host A → GOOGLE

```
root@a18e672d39b3:/# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=127 time=133 ms
8 bytes from 8.8.8.8: icmp_seq=1 ttl=64 (truncated)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=127 time=63.5 ms
8 bytes from 8.8.8.8: icmp_seq=2 ttl=64 (truncated)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=127 time=19.7 ms
8 bytes from 8.8.8.8: icmp_seq=3 ttl=64 (truncated)
64 bytes from 8.8.8.8: icmp_seq=4 ttl=127 time=17.9 ms
8 bytes from 8.8.8.8: icmp_seq=4 ttl=64 (truncated)
64 bytes from 8.8.8.8: icmp_seq=5 ttl=127 time=51.6 ms
8 bytes from 8.8.8.8: icmp_seq=5 ttl=64 (truncated)
8 bytes from 8.8.8.8: icmp_seq=6 ttl=64 (truncated)
```

HOST A SEND PING TO 8.8.8.8

TERMINAL – THE FAKE PING

```
############################################################
Sending spoofd IP packet...


**********************ICMP Packet************************

IP Header
   |-IP Total Length   : 28  Bytes(Size of Packet)
   |-Source IP         : 8.8.8.8
   |-Destination IP    : 10.9.0.5

ICMP Header
   |-Type : 0
   |-Seq  : 0

############################################################
Sending spoofd IP packet...


**********************ICMP Packet************************

IP Header
   |-IP Total Length   : 28  Bytes(Size of Packet)
   |-Source IP         : 8.8.8.8
   |-Destination IP    : 10.9.0.5

ICMP Header
   |-Type : 0
   |-Seq  : 0

############################################################
Sending spoofd IP packet...
```

3.Host A → FAKE IP

```
root@a7ff507f13ec:/# ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
^C
--- 1.2.3.4 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3055ms
```

HOST A SEND PING TO 1.2.3.4

TERMINAL – THE FAKE PING

```
############################################################
Sending spoofd IP packet...


**********************ICMP Packet************************

IP Header
   |-IP Total Length   : 28  Bytes(Size of Packet)
   |-Source IP         : 1.2.3.4
   |-Destination IP    : 10.9.0.5

ICMP Header
   |-Type : 0
   |-Seq  : 0

############################################################
Sending spoofd IP packet...


**********************ICMP Packet************************

IP Header
   |-IP Total Length   : 28  Bytes(Size of Packet)
   |-Source IP         : 1.2.3.4
   |-Destination IP    : 10.9.0.5

ICMP Header
   |-Type : 0
   |-Seq  : 0

############################################################
```

1.4.4 Wireshark

1.Host A → Host B

1	0.000000000	10.9.0.5	10.9.0.6	ICMP	100	Echo (ping) request id=0x0004, seq=1/256, ttl=64 (no response found!)
2	0.000105021	10.9.0.5	10.9.0.6	ICMP	100	Echo (ping) request id=0x0004, seq=1/256, ttl=64 (no response found!)
3	0.000000000	10.9.0.5	10.9.0.6	ICMP	100	Echo (ping) request id=0x0004, seq=1/256, ttl=64 (reply in 4)
4	0.000235647	10.9.0.6	10.9.0.5	ICMP	100	Echo (ping) reply id=0x0004, seq=1/256, ttl=64 (request in 3)
5	0.000253622	10.9.0.6	10.9.0.5	ICMP	100	Echo (ping) reply id=0x0004, seq=1/256, ttl=64
6	0.000235647	10.9.0.6	10.9.0.5	ICMP	100	Echo (ping) reply id=0x0004, seq=1/256, ttl=64
7	0.259797229	10.9.0.6	10.9.0.5	ICMP	44	Echo (ping) reply id=0x0000, seq=0/0, ttl=20
8	0.259811832	10.9.0.6	10.9.0.5	ICMP	44	Echo (ping) reply id=0x0000, seq=0/0, ttl=20

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

2. “_____”

3. Ping request from A to B.

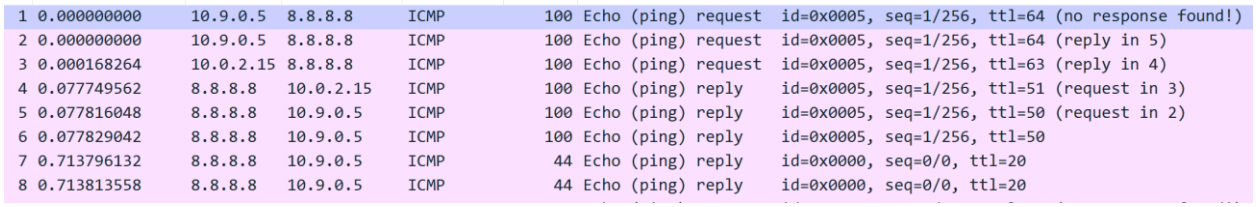
4. Ping reply from the real B.

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



1	0.000000000	10.9.0.5	8.8.8.8	ICMP	100	Echo (ping) request id=0x0005, seq=1/256, ttl=64 (no response found!)
2	0.000000000	10.9.0.5	8.8.8.8	ICMP	100	Echo (ping) request id=0x0005, seq=1/256, ttl=64 (reply in 5)
3	0.000168264	10.0.2.15	8.8.8.8	ICMP	100	Echo (ping) request id=0x0005, seq=1/256, ttl=63 (reply in 4)
4	0.077749562	8.8.8.8	10.0.2.15	ICMP	100	Echo (ping) reply id=0x0005, seq=1/256, ttl=51 (request in 3)
5	0.077816048	8.8.8.8	10.9.0.5	ICMP	100	Echo (ping) reply id=0x0005, seq=1/256, ttl=50 (request in 2)
6	0.077829042	8.8.8.8	10.9.0.5	ICMP	100	Echo (ping) reply id=0x0005, seq=1/256, ttl=50
7	0.713796132	8.8.8.8	10.9.0.5	ICMP	44	Echo (ping) reply id=0x0000, seq=0/0, ttl=20
8	0.713813558	8.8.8.8	10.9.0.5	ICMP	44	Echo (ping) reply id=0x0000, seq=0/0, ttl=20

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	0.000000000	10.9.0.5	1.2.3.4	ICMP	100	Echo (ping) request id=0x0006, seq=1/256, ttl=64 (no response found!)
2	0.000000000	10.9.0.5	1.2.3.4	ICMP	100	Echo (ping) request id=0x0006, seq=1/256, ttl=64 (no response found!)
3	0.000150960	10.0.2.15	1.2.3.4	ICMP	100	Echo (ping) request id=0x0006, seq=1/256, ttl=63 (no response found!)
4	0.320049577	1.2.3.4	10.9.0.5	ICMP	44	Echo (ping) reply id=0x0000, seq=0/0, ttl=20
5	0.320063032	1.2.3.4	10.9.0.5	ICMP	44	Echo (ping) reply id=0x0000, seq=0/0, ttl=20

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.

```
int sockAny , sockIp;
struct sockaddr_in receiver,sender;
```

Main():

Retrieve IP from user:

```
int main(int argc , char *argv[])
{
    if (argc < 2) {
        printf("please put IP !\n");
        exit(1);
    }
```

Create sending socket:

```
// Create socket
sockIp = socket(AF_INET, SOCK_DGRAM, 0);
if (sockIp < 0) {
    perror("Error creating socket");
    return 1;
}
```

Create receiving port and binding to port 50000:

```
printf("Starting the gateway process..\n");
//Create a raw socket that shall sniff
sockAny = socket(AF_INET , SOCK_DGRAM , 0);
if(sockAny < 0)
{
    printf("Socket Error\n");
    return 1;
}

receiver.sin_port = htons(PORT);
receiver.sin_addr.s_addr = INADDR_ANY;
receiver.sin_family = AF_INET;
receiverSize = sizeof receiver;

if (bind(sockAny, (struct sockaddr *) &receiver, receiverSize) < 0) {
    perror("Error binding socket to IP and port");
    return 1;
}
```

Infinite loop for sending received packets:

```
while(1)
{

    printf("Receiving UDP packets\n");
    //Receive a packet
    dataSize = recvfrom(sockAny , buffer , 65536 , 0 , NULL , 0);
    if(dataSize <0 )
    {
        printf("Recvfrom error , failed to get packets\n");
        return 1;
    }

    printf("let's see if you have lucky today ..........\n");
    double randomNum = ((float)random())/((float)RAND_MAX);
    if (randomNum > 0.5)
    {
        //Now process the packet
        send_udp_packet(buffer , dataSize , argv[1]);
        printf("The number is : %f ,congratulation! you packet will be transfer through the Gateway..\n" , randomNum);
        printf("The gateway transfer the packet to %s.\n", argv[1]);
    }
    else{
        printf("The number is : %f ,sorry luck is not with you today, try again later!\n" , randomNum);
    }

}
```

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.

```
void send_udp_packet(unsigned char *packet, int size , char *ip)
{

    printf("\n\n******UDP Packet******\n\n");

    // Set the destination address
    memset(&sender, 0, sizeof(sender));
    sender.sin_family = AF_INET;
    sender.sin_port = htons(PORT_1);
    sender.sin_addr.s_addr = inet_addr(ip);


    if(sendto(sockIp, packet, size , 0, (struct sockaddr *) &sender, sizeof(sender)) < 0){
        perror("Error with sendto() the packet\n");

    }
    printf("Succeed with send the packet!\n");
    return;
}
```

Close the sockets:

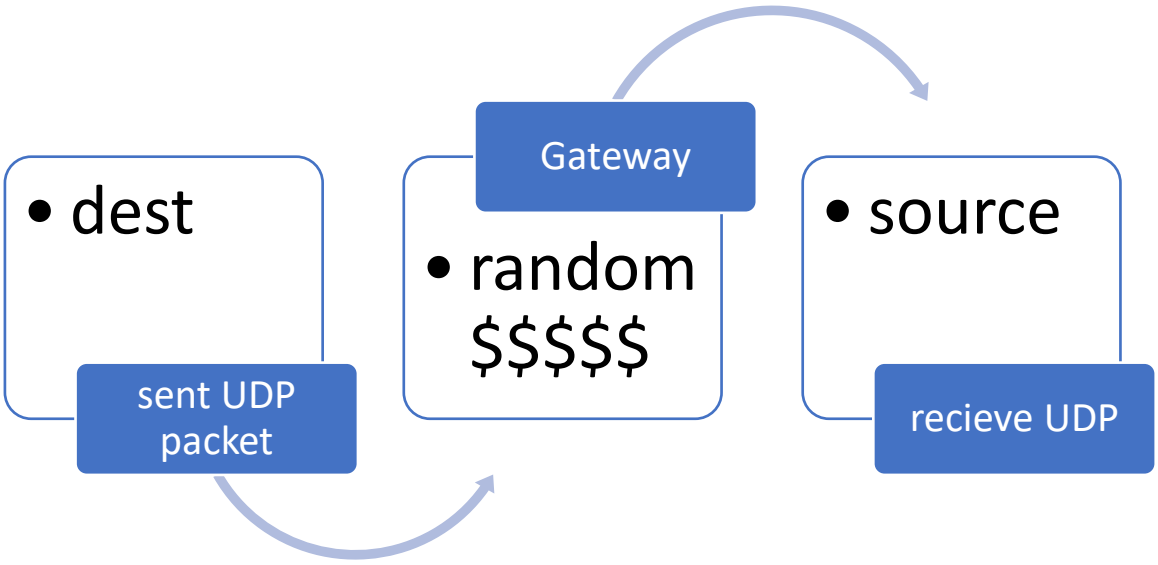
```
    }
    close(sockAny);
    close(sockIp);
    printf("Done");
    return 0;
}
```

Function for sending the UDP packets:

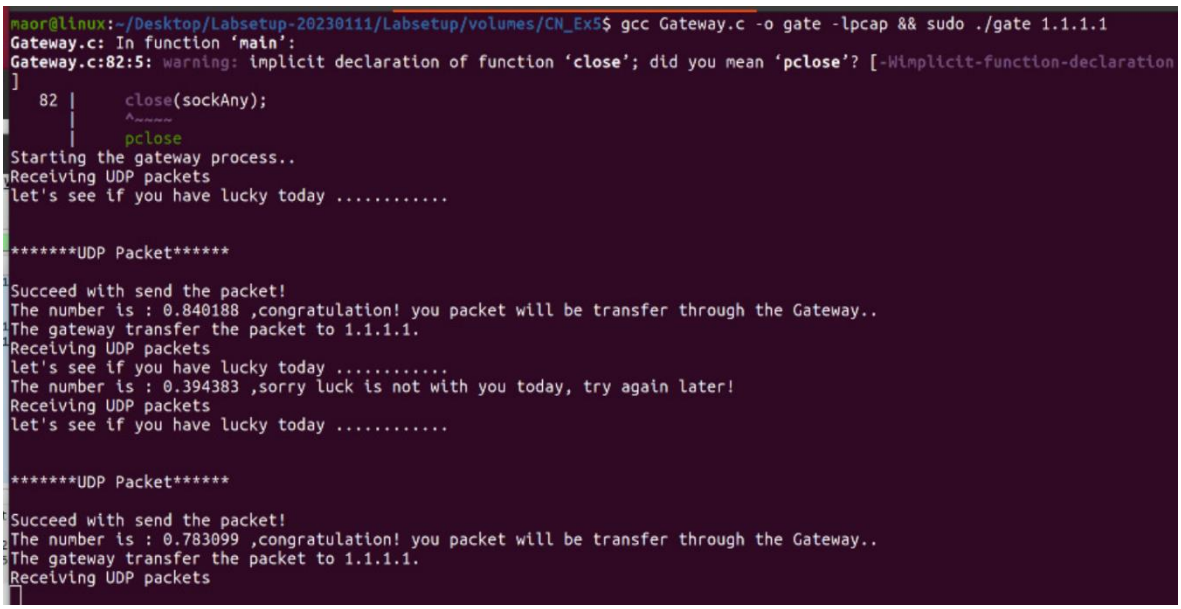
```
void send_udp_packet(unsigned char *packet, int size , char *ip)
{

    printf("\n\n*******UDP Packet******\n\n");

    // Set the destination address
    memset(&sender, 0, sizeof(sender));
    sender.sin_family = AF_INET;
    sender.sin_port = htons(PORT_1);
    sender.sin_addr.s_addr = inet_addr(ip);


    if(sendto(sockIp, packet, size , 0, (struct sockaddr *) &sender, sizeof(sender)) < 0){
        perror("Error with sendto() the packet\n");

    }
    printf("Succeed with send the packet!\n");
    return;
}
```

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	0.000000000	127.0.0.1	127.0.0.1	UDP	61	50430 → 50000 Len=17
2	0.000126240	10.0.2.15	1.1.1.1	UDP	61	60802 → 50001 Len=17
3	1.857313522	127.0.0.1	127.0.0.1	UDP	61	33038 → 50000 Len=17
4	3.518863682	127.0.0.1	127.0.0.1	UDP	61	48318 → 50000 Len=17
5	3.519223449	10.0.2.15	1.1.1.1	UDP	61	60802 → 50001 Len=17

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