

Assignment 6

By:

Yehonatan Amosi 209542349

Liav Levi 206603193

Task 1

A:

in this task we wrote a short program using scapy library and used the method show() so all the data of the packet we capture will be print.

Note: the code is add In the folder of the task.

Result:

```
seed@seed:~$ 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=114 time=56.8 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=114 time=55.6 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=114 time=56.0 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=114 time=56.8 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=114 time=56.7 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=114 time=56.1 ms
64 bytes from 8.8.8.8: icmp_seq=7 ttl=114 time=56.6 ms
64 bytes from 8.8.8.8: icmp_seq=8 ttl=114 time=58.2 ms
64 bytes from 8.8.8.8: icmp_seq=9 ttl=114 time=55.4 ms
64 bytes from 8.8.8.8: icmp_seq=10 ttl=114 time=57.2 ms
64 bytes from 8.8.8.8: icmp_seq=11 ttl=114 time=56.5 ms
64 bytes from 8.8.8.8: icmp_seq=12 ttl=114 time=59.4 ms
64 bytes from 8.8.8.8: icmp_seq=13 ttl=114 time=56.0 ms
64 bytes from 8.8.8.8: icmp_seq=14 ttl=114 time=55.7 ms
64 bytes from 8.8.8.8: icmp_seq=15 ttl=114 time=58.8 ms
64 bytes from 8.8.8.8: icmp_seq=16 ttl=114 time=56.7 ms
64 bytes from 8.8.8.8: icmp_seq=17 ttl=114 time=55.8 ms
^C
--- 8.8.8.8 ping statistics ---
17 packets transmitted, 17 received, 0% packet loss, time 16099 ms
rtt min/avg/max/mdev = 55.441/56.726/59.397/1.095 ms
[12/29/21]seed@VM:~$

seed@VM:~/Assignment_6$ sudo python3 main.py
[[[ Ethernet ]]]
dst      = 52:54:00:12:35:00
src      = 08:00:27:56:93:c1
type     = IPv4
[[[ IP ]]]
version  = 4
ihl      = 5
tos      = 0x0
len      = 84
id       = 48648
flags    = DF
frag     = 0
ttl      = 64
proto    = icmp
chksum   = 0x608d
src      = 10.0.2.4
dst      = 8.8.8.8
\options \
[[[ ICMP ]]]
type     = echo-request
code     = 0
chksum   = 0xfa01
id       = 0x1
seq      = 0x1
[[[ Raw ]]]
load     = '\xd8:\xcc\xa0\x00\x00\x00\x96\x8b\x04\x00\x00
\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
! "%&'()*+,-./01234567'
```

B:

a) Capture only ICMP packet

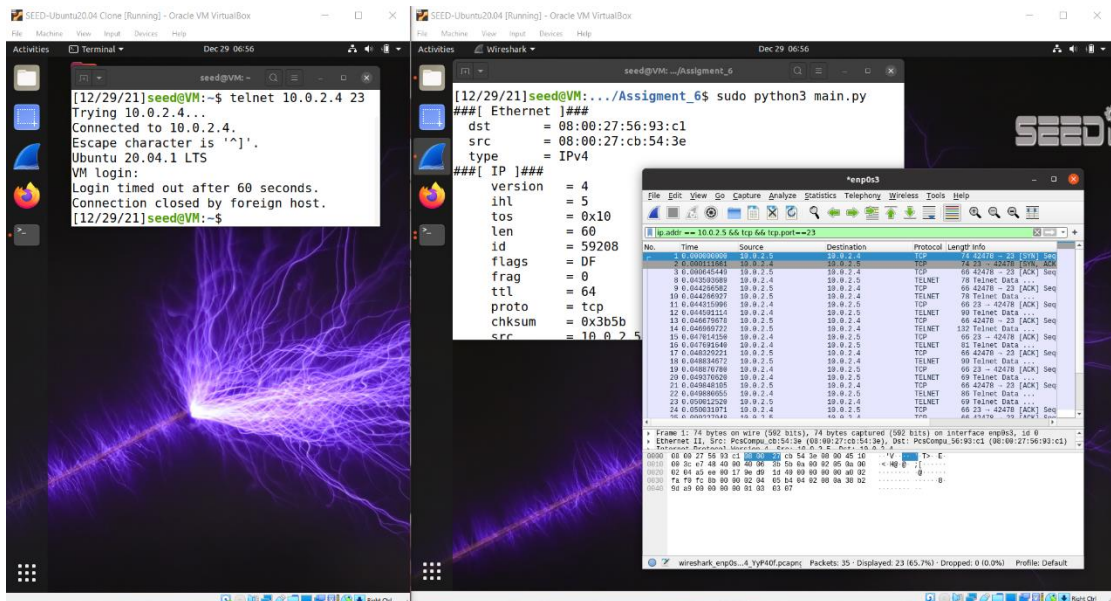
```
[01/03/22]seed@VM:~$ ping 10.0.2.4
PING 10.0.2.4 (10.0.2.4) 56(84) bytes of data:
64 bytes from 10.0.2.4: icmp_seq=1 ttl=64 time=1.27 ms
64 bytes from 10.0.2.4: icmp_seq=2 ttl=64 time=0.971 ms
64 bytes from 10.0.2.4: icmp_seq=3 ttl=64 time=0.670 ms
64 bytes from 10.0.2.4: icmp_seq=4 ttl=64 time=1.10 ms
64 bytes from 10.0.2.4: icmp_seq=5 ttl=64 time=0.720 ms
64 bytes from 10.0.2.4: icmp_seq=6 ttl=64 time=0.984 ms
^C
--- 10.0.2.4 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5036ms
rtt min/avg/max/mdev = 0.670/0.952/1.269/0.207 ms
[01/03/22]seed@VM:~$

seed@VM:~/Assignment_6$
64 bytes from 8.8.8.8: icmp_seq=1 ttl=117 time=73.5 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=117 time=59.0 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=117 time=88.0 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=117 time=46.4 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=117 time=61.6 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=117 time=406 ms
64 bytes from 8.8.8.8: icmp_seq=7 ttl=117 time=50.1 ms
64 bytes from 8.8.8.8: icmp_seq=8 ttl=117 time=51.5 ms
64 bytes from 8.8.8.8: icmp_seq=9 ttl=117 time=61.3 ms
^C
--- 8.8.8.8 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8029ms
rtt min/avg/max/mdev = 46.426/99.745/406.400/109.087 ms
[01/03/22]seed@VM:~/Assignment_6$

TYPE:      : 8
CODE:      : 0
Source IP  : 10.0.2.5
Destination IP : 10.0.2.4
Got a packet
TYPE:      : 8
CODE:      : 0
Source IP  : 10.0.2.5
Destination IP : 10.0.2.4
Got a packet
TYPE:      : 8
CODE:      : 0
Source IP  : 10.0.2.5
Destination IP : 10.0.2.4
Got a packet
TYPE:      : 8
```

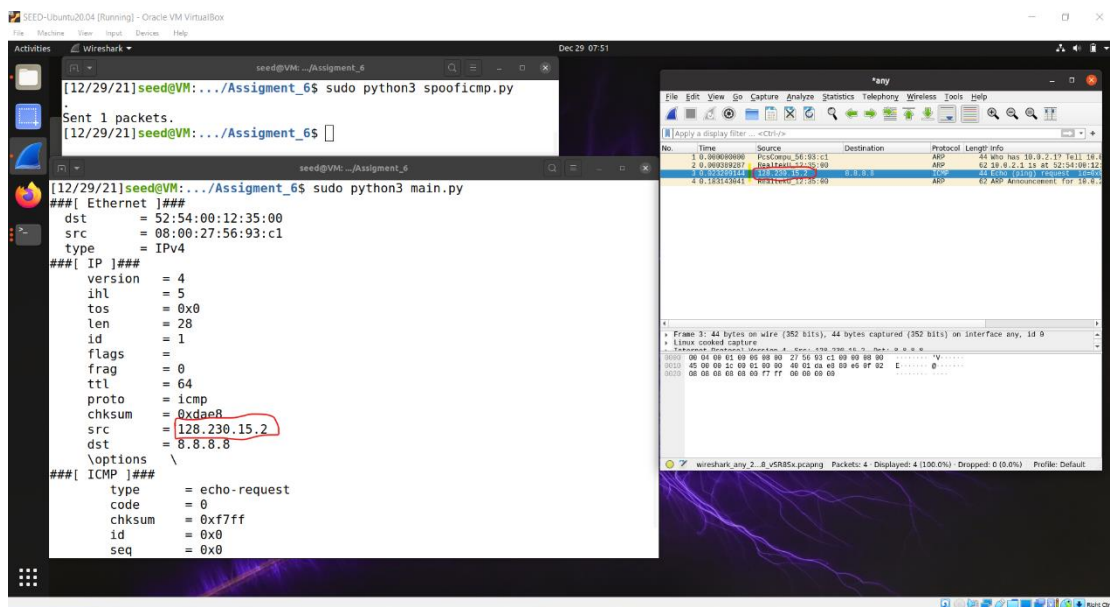
We send ping to my computer and we can see that we capture the packets by looking in the wireshark and for the fact that our sniffer print the data.

- b) Capture any TCP packet that comes from a particular IP and with a destination port number 23:



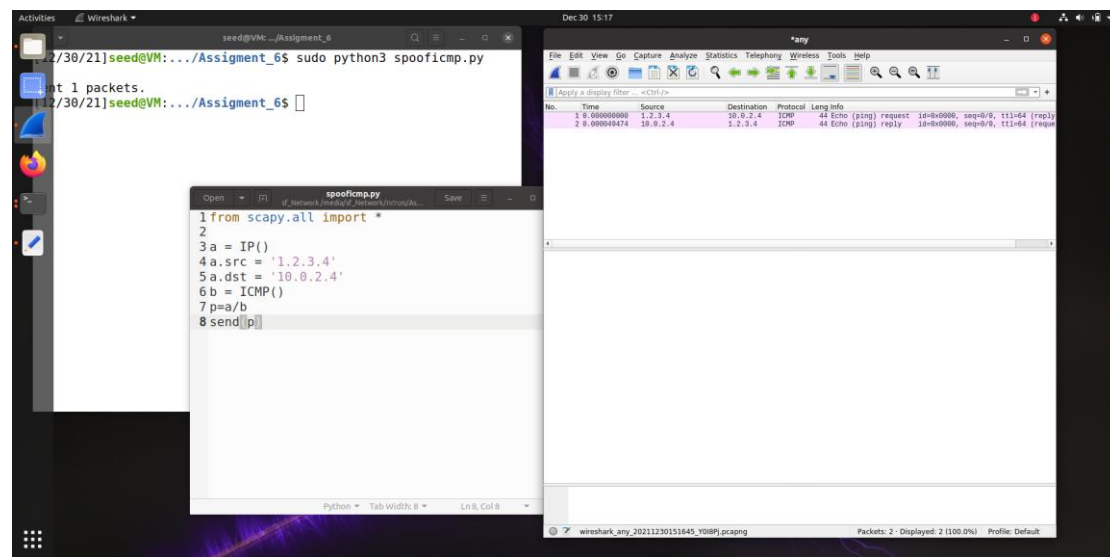
As we can see our sniffer capture only the packets that came from the left computer and print them.

- c) Capture packets comes from or to go to a particular subnet. You can pick any subnet, such as 128.230.0.0/16; you should not pick the subnet that your VM is attached to:



As we can see we capture the requested packets from the requested subnet.

Task 1 2: Spoofing ICMP Packets



as we can see we send packet from IP -1.2.3.4 to the IP – 10.0.2.4

the code:

```
1 from scapy.all import *
2
3 a = IP()
4 a.src = '1.2.3.4'
5 a.dst = '10.0.2.4'
6 b = ICMP()
7 p=a/b
8 send(p)
```

In the code above, Line 3 creates an IP object from the IP class. Line 3 shows how to set the destination IP address field. If a field is not set, a default value will be used.

Line 6 creates an ICMP object. The default type is echo request. In Line 7, we stack a and b together to form a new object. We can now send out this packet using `send()` in Line 8.

Task 1 3: Traceroute

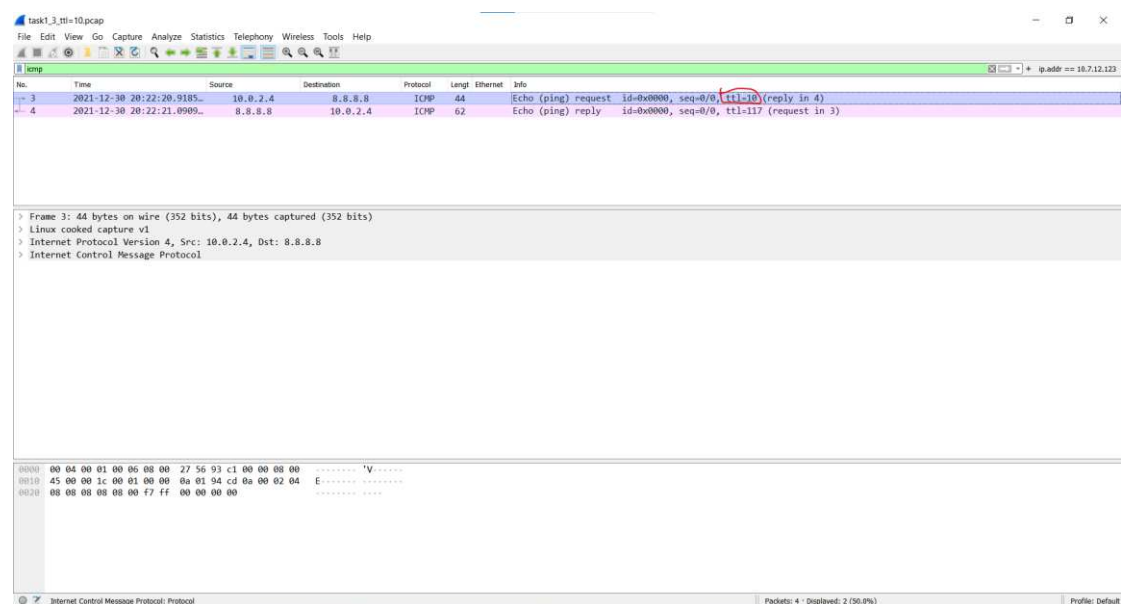
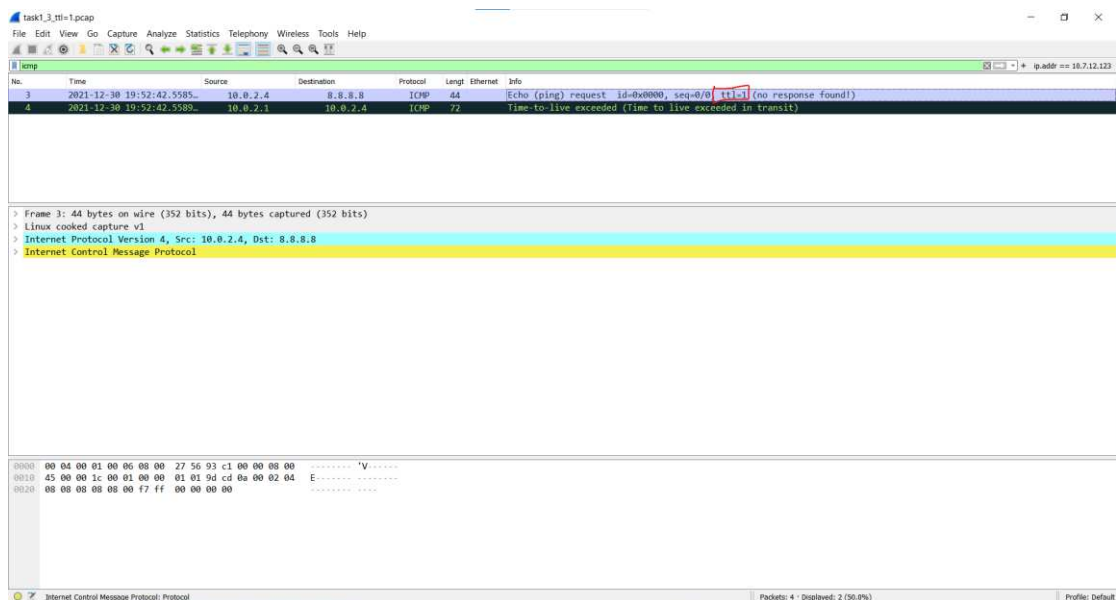
Code:

```
from scapy.all import *  
a = IP()  
a.dst = '8.8.8.8'  
a.ttl = 10 # change the number from 1 to 10  
b = ICMP()  
send(a/b)
```

We were change the ttl value from 1 to 10 and the we got the back echo respond.

To see the full result we add the pcap record and you can see there the full results.

We add the first record and the last one:



Task 1 4: Sniffing and-then Spoofing

Code:

```
#!/usr/bin/env python3
from scapy.all import *

def sniff_pkt(pkt):
    if pkt[ICMP].type != 8:
        return

    ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
    icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
    data = pkt[Raw].load
    new_pkt = ip / icmp / data
    send(new_pkt)

while 1:
    pkt = sniff(filter='icmp', prn=sniff_pkt)
```

In the method sniff_pkt() we make a fake packets and just switching between source and destination IP.

Ping to 1.2.3.4 :

The screenshot shows a virtual machine environment with two windows. The top window is a terminal titled 'seed@VM: ~' showing the execution of a ping command to 1.2.3.4. The output shows four successful ping requests and replies, with statistics indicating 4 packets transmitted, 4 received, 0% packet loss, and a total time of 3005ms. The bottom window is Wireshark, showing a packet capture titled 'task1_4_1-2-3-4.pcap'. The capture shows four ICMP Echo (ping) request and reply packets between source IP 192.168.1.15 and destination IP 1.2.3.4. The packets are numbered 14 through 17.

No.	Time	Source	Destination	Protocol	Length	Info
14	0.000000	192.168.1.15	1.2.3.4	ICMP	60	Echo (ping) request
15	0.000000	1.2.3.4	192.168.1.15	ICMP	60	Echo (ping) reply
16	0.000000	192.168.1.15	1.2.3.4	ICMP	60	Echo (ping) request
17	0.000000	1.2.3.4	192.168.1.15	ICMP	60	Echo (ping) reply

Ping to 8.8.8.8:

The screenshot displays a virtual machine environment with three main components:

- Terminal Window (Left):** Shows the command `sudo python3 sniff_and_spoofing.py` being executed. The output indicates that 1 packet was sent.
- Wireshark Window (Middle):** Displays a packet capture titled `task1_4-8-8-8.pcap`. The packet list shows 14 packets, all of which are ICMP Echo (ping) requests and replies between the source IP `10.0.2.6` and the destination IP `8.8.8.8`.
- Terminal Window (Right):** Shows the output of the `ping 8.8.8.8` command. The output indicates that 4 packets were transmitted, 4 were received, and there were 0% packet loss, with a total time of 3016ms.

Ping to 10.9.0.99:

The screenshot displays a virtual machine environment with three main components:

- Terminal Window (Left):** Shows the command `sudo python3 sniff_and_spoofing.py` being executed. The output indicates that 1 packet was sent.
- Wireshark Window (Middle):** Displays a packet capture titled `*esp03`. The packet list shows 14 packets, all of which are ICMP Echo (ping) requests and replies between the source IP `10.0.2.6` and the destination IP `10.9.0.99`.
- Terminal Window (Right):** Shows the output of the `ping 10.9.0.99` command. The output indicates that 9 packets were transmitted, 9 were received, and there were 0% packet loss, with a total time of 8018ms.

Task 2.1A: Understanding How a Sniffer Works- In this task, we wrote a sniffer program that use pcap library and print out the source and destination IP addresses of each captured packet.

The code:

```
C sniffer.c 2 X
Assignment_6 > task2_1 > C sniffer.c > main()
1  #include <pcap.h>
2  #include <stdio.h>
3  #include <arpa/inet.h>
4  #include <stdlib.h>
5  #include <ctype.h>
6
7  struct ethheader
8  {
9      u_char ether_dhost[6]; /* destination host address */
10     u_char ether_shost[6]; /* source host address */
11     u_short ether_type;    /* IP? ARP? RARP? etc */
12 };
13 struct ipheader
14 {
15     unsigned char iph_ihl : 4,      /*IP header length in byte
16         iph_ver : 4;                /*IP version
17     unsigned char iph_tos;           /*Type of service
18     unsigned short int iph_len;      /*IP Packet length (data + header)
19     unsigned short int iph_ident;    /*Identification
20     unsigned short int iph_flag : 3, /*Fragmentation flags
21     iph_offset : 13;                /*Flags offset
22     unsigned char iph_ttl;          /*Time to Live
23     unsigned char iph_protocol;     /*Protocol type
24     unsigned short int iph_chksum;   /*IP datagram checksum
25     struct in_addr iph_sourceip;    /*Source IP address
26     struct in_addr iph_destip;     /*Destination IP address
27 };
28 #define IP_HL(ip) (((ip)->iph_ihl) & 0x0f) // 1111 0101 1011 0010 AND 0000 1111 = 0000
29 #define IP_V(ip) (((ip)->ip_vhl) >> 4)
30
31 void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
32 {
33     struct ethheader *eth = (struct ethheader *)packet;
34     if (ntohs(eth->ether_type) == 0x0800) // check if its type IP
35     {
36         // copy the data to an ip header, the buffer includes all the data we need
37         struct ipheader *ipHeader = (struct ipheader *) (packet + sizeof(struct ethheader));
38         printf("got a packet\n");
39         printf("Source IP      : %s\n", inet_ntoa(ipHeader->iph_sourceip));
40         printf("Destination IP : %s\n", inet_ntoa(ipHeader->iph_destip));
41     }
42 }
43
44 int main()
45 {
46     pcap_t *handle;
47     char errbuf[PCAP_ERRBUF_SIZE];
48     struct bpf_program fp;
49     char filter_exp[] = "";
50     bpf_u_int32 net;
51
52     // Step 1: Open live pcap session on NIC with name eth3
53     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf); // can be change to lo so the myping.c file will worke
54
55     // Step 2: Compile filter_exp into BPF psuedo-code
56     pcap_compile(handle, &fp, filter_exp, 0, net);
57     pcap_setfilter(handle, &fp);
58
59     // Step 3: Capture packets
60     pcap_loop(handle, -1, got_packet, NULL);
61
62     pcap_close(handle); //Close the handle
63     return 0;
64 }
65
```

Question 1:

Please use your own words to describe the sequence of the library calls that are essential for sniffer programs. This is meant to be a summary, not detailed explanation like the one in the tutorial or book.

Answer:

pcap_open_live - this function used to open a listener socket to the network

pcap_compile and Pcap_setfilter - these functions used for making a filter so when we capture packets, we are not capturing all of them, we capture only the packets that we define on the filter like ICMP packets and more.

pcap_loop – this function used for capturing the packets the goes on the network after we define the filter we want.

pcap_close – this function closes the listener socket we opened.

Question 2:

Why do you need the root privilege to run a sniffer program? Where does the program fail if it is executed without the root privilege?

Answer:

because we are using raw socket and also use promiscuous mode the OS is blocking the access to the raw socket we are using to build the packets. The program fails at access the packets that goes on the network.

Question 3:

Please turn on and turn off the promiscuous mode in your sniffer program. The value 1 of the third parameter in pcap open live() turns on the promiscuous mode (use 0 to turn it off). Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this.

Answer:

when promiscuous mode is off we can see that the program sniffing packets that my IP was the source (including when someone is answering back to me and then my IP isn't the source IP) at the other hand when promiscuous is on we can see that we capturing packet that going from different computers. Here some screen shots to show that:

Promiscuous mode off:

The left screenshot shows a terminal window with a C program named 'sniffer.c' running. The program uses pcap_open_live with a third parameter of 0, indicating promiscuous mode is off. It captures packets from source IP 10.0.2.4 to destination IP 10.0.0.138. The right screenshot shows a Wireshark capture on interface 'enp0s3'. The capture shows various DNS and DHCP traffic from multiple source IPs (10.0.0.138, 10.0.2.1, 10.0.2.3, 10.0.2.4, 10.0.2.17, 10.0.2.11) to destination IP 10.0.0.138. A packet detail view for a DNS query from 10.0.2.1 to 10.0.0.138 is shown, confirming the source IP is not the local machine.

Promiscuous mode on:

The screenshot displays a virtual machine environment with three main windows:

- Wireshark:** Shows a list of captured packets. The first packet is an ICMP Echo (ping) request from 10.0.2.5 to 8.8.8.8. The second packet is an ICMP Echo (ping) reply from 8.8.8.8 to 10.0.2.5.
- Terminal:** Shows the execution of a script named `sniffnc`. The script performs the following steps:
 - Step 1: Open live pcap session on NIC with name `eth3`.
 - Step 2: Compile filter `exp` into BPF pseudo-code.
 - Step 3: Capture packets.
- Task2_1_AQ3_2_pcap:** Shows a list of captured packets. The first packet is an ICMP Echo (ping) request from 10.0.2.5 to 8.8.8.8. The second packet is an ICMP Echo (ping) reply from 8.8.8.8 to 10.0.2.5.

Here we send a ping message to 8.8.8.8 from another VM and her IP is 10.0.2.5

Task 2 1 B

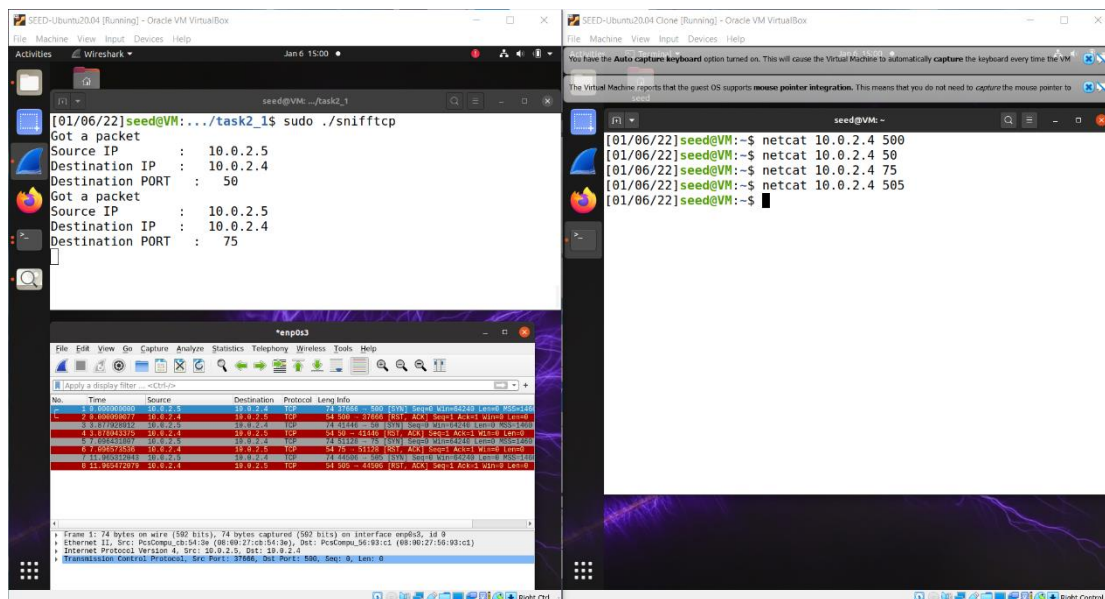
a) Capture the ICMP packets between two specific hosts

We send icmp packet from 10.0.2.5 to 10.0.2.4:

The screenshot displays a virtual machine environment with three main windows:

- Wireshark:** Shows a list of captured packets. The first packet is an ICMP Echo (ping) request from 10.0.2.5 to 10.0.2.4. The second packet is an ICMP Echo (ping) reply from 10.0.2.4 to 10.0.2.5.
- Terminal:** Shows the execution of a script named `sniffnc`. The script performs the following steps:
 - Step 1: Open live pcap session on NIC with name `eth3`.
 - Step 2: Compile filter `exp` into BPF pseudo-code.
 - Step 3: Capture packets.
- Task2_1_AQ3_2_pcap:** Shows a list of captured packets. The first packet is an ICMP Echo (ping) request from 10.0.2.5 to 10.0.2.4. The second packet is an ICMP Echo (ping) reply from 10.0.2.4 to 10.0.2.5.

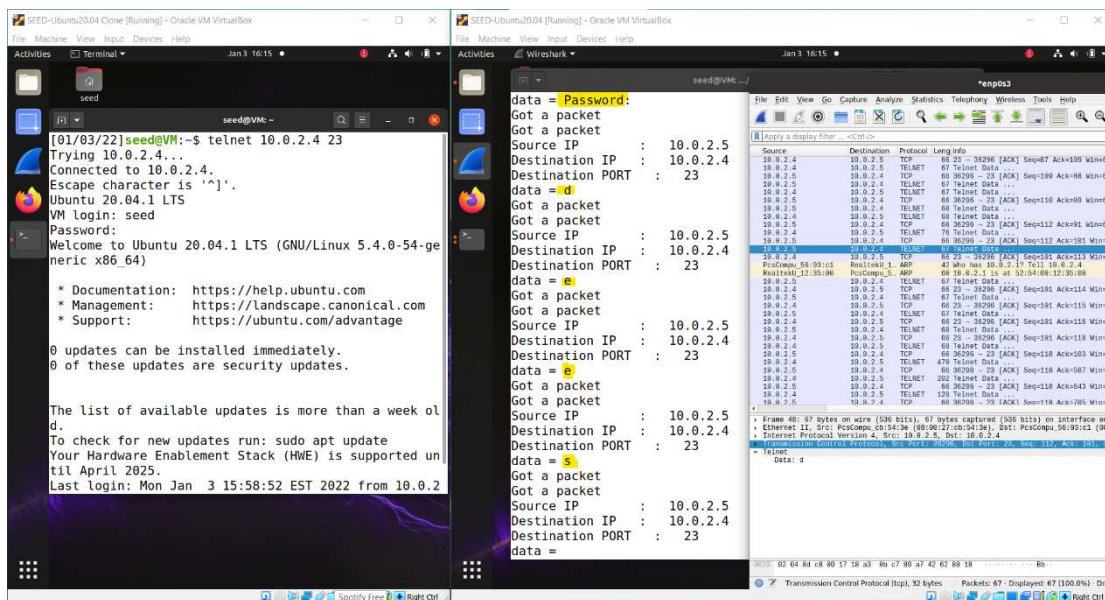
b) Capture the TCP packets with a destination port number in the range from 10 to 100



As we can see only the packets that goes between the asked range ports are capture in our sniffer and printed.

Task 2_1 C: Sniffing Passwords

We use our sniffer program to capture the password when somebody is using telnet on the network that we are monitoring:



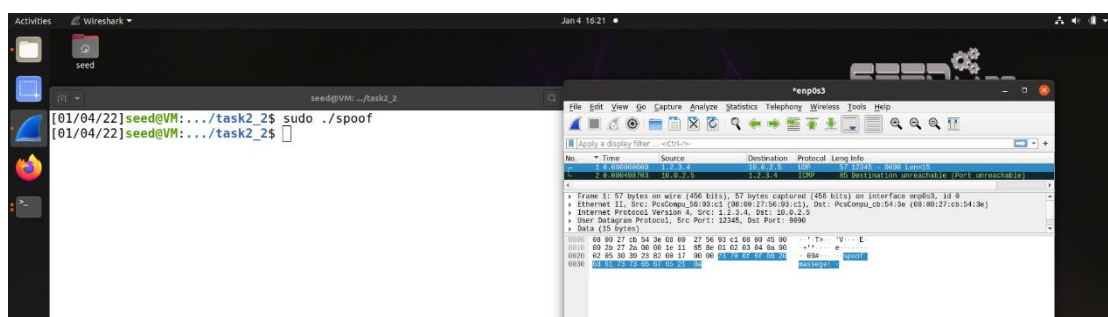
As we can the password is "dees".

Task 2 2:

A: Write a spoofing program. we write a packet spoofing program in C.

code:

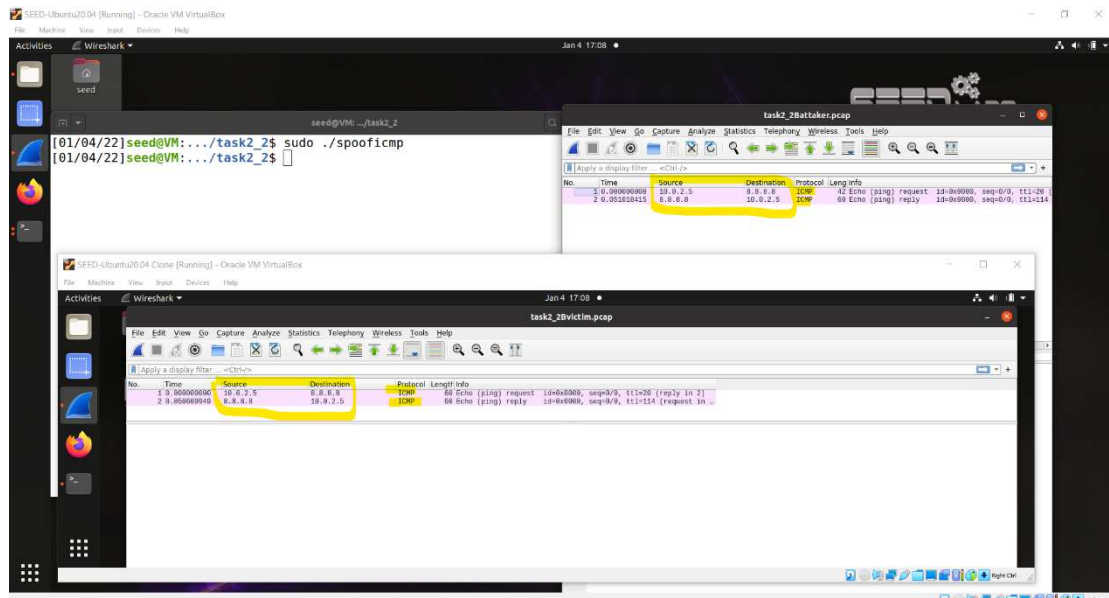
```
1 #include <pcap.h>
2 #include <stdio.h>
3 #include <arpa/inet.h>
4 #include <stdlib.h>
5 #include <ctype.h>
6
7 struct ethheader
8 {
9     u_char ether_dhost[6]; /* destination host address */
10    u_char ether_shost[6]; /* source host address */
11    u_short ether_type; /* IP? ARP? RARP? etc */
12 };
13
14 struct ipheader
15 {
16     unsigned char iph_ihl : 4, /*IP header length in byte
17     iph_ver : 4; /*IP version
18     unsigned char iph_tos; /*Type of service
19     unsigned short int iph_len; /*IP Packet length (data + header)
20     unsigned short int iph_ident; /*Identification
21     unsigned short int iph_flag : 3, /*Fragmentation flags
22     iph_offset : 13; /*Flags offset
23     unsigned char iph_ttl; /*Time to Live
24     unsigned char iph_protocol; /*Protocol type
25     unsigned short int iph_chksum; /*IP datagram checksum
26     struct in_addr iph_sourceip; /*Source IP address
27     struct in_addr iph_destip; /*Destination IP address
28 };
29
30 #define IP_HL(ip) (((ip)->iph_ihl) & 0x0f) // 1111 0101 1011 0010 AND 0000 1111 = 0000
31 #define IP_V(ip) (((ip)->ip_vhl) >> 4)
32
33 void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
34 {
35     struct ethheader *eth = (struct ethheader *)packet;
36     if (ntohs(eth->ether_type) == 0x0800) // check if its type IP
37     {
38         // copy the data to an ip header, the buffer includes all the data we need
39         struct ipheader *ipHeader = (struct ipheader *)(packet + sizeof(struct ethheader));
40         printf("Got a packet\n");
41         printf("Source IP : %s\n", inet_ntoa(ipHeader->iph_sourceip));
42         printf("Destination IP : %s\n", inet_ntoa(ipHeader->iph_destip));
43     }
44 }
45
46 int main()
47 {
48     pcap_t *handle;
49     char errbuf[PCAP_ERRBUF_SIZE];
50     struct bpf_program fp;
51     char filter_exp[] = "";
52     bpf_u_int32 net;
53
54     // Step 1: Open live pcap session on NIC with name eth3
55     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf); // can be change to lo so the myping.c file will worke
56
57     // Step 2: Compile filter_exp into BPF pseudo-code
58     pcap_compile(handle, &fp, filter_exp, 0, net);
59     pcap_setfilter(handle, &fp);
60
61     // Step 3: Capture packets
62     pcap_loop(handle, -1, got_packet, NULL);
63
64     pcap_close(handle); //Close the handle
65     return 0;
66 }
```



B: Spoof an ICMP Echo Request. Spoof an ICMP echo request packet on behalf of another machine (i.e., using another machine's IP address as its source IP address).

The code is add at the folder of Task2_2.

We send a spoofed packet from 10.0.2.5 to 8.8.8.8 but our real IP was 10.0.2.4:



- **Question 4:** Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?

Answers:

yes, the IP packet length can be any arbitrary value, although the packet's total length is overwritten to its original size when its sent.

- **Question 5:** Using the raw socket programming, do you have to calculate the checksum for the IP header?

Answers:

we don't have, the reason is that we can tell the kernel to calculate the checksum for the IP header. In IP header fields we have the parameter "ip_check", if we do "ip_check = 0" that will let the kernel to do the checksum by default unless we change it to different value but then we will have to use a checksum method.

- **Question 6:** Why do you need the root privilege to run the programs that use raw sockets? Where does the program fail if executed without the root privilege?

Answers:

we must have the root privileges and they are necessary to run the program that make use of raw socket. If you use non-privileges user you will not have the permissions to change all the fields in the protocol headers. The root privileges will give the ability to change and set any fields in the protocol headers and access to the socket and put the interface card in promiscuous mode. If we run the program without the root privileges, it will fail at socket setup.

Task 2 3:

Code:

```
1  #include <pcap.h>
2  #include <stdio.h>
3  #include <arpa/inet.h>
4  #include <stdlib.h>
5  #include <ctype.h>
6  #include <string.h>
7  #include <unistd.h>
8  #include <sys/socket.h>
9
10 struct icmpheader
11 {
12     unsigned char icmp_type; // ICMP message type
13     unsigned char icmp_code; // Error code
14     unsigned short int icmp_chksm;
15     unsigned short int icmp_id;
16     unsigned short int icmp_seq;
17 };
18 struct ethheader
19 {
20     u_char ether_dhost[6]; /* destination host address */
21     u_char ether_shost[6]; /* source host address */
22     u_short ether_type;    /* IP? ARP? RARP? etc */
23 };
24 struct ipheader
25 {
26     unsigned char iph_ihl : 4,      //IP header length in byte
27     | iph_ver : 4;                  //IP version
28     unsigned char iph_tos;           //Type of service
29     unsigned short int iph_len;      //IP Packet length (data + header)
30     unsigned short int iph_ident;    //Identification
31     unsigned short int iph_flag : 3, //Fragmentation flags
32     | iph_offset : 13;              //Flags offset
33     unsigned char iph_ttl;           //Time to Live
34     unsigned char iph_protocol;      //Protocol type
35     unsigned short int iph_chksm;    //IP datagram checksum
36     struct in_addr iph_sourceip;     //Source IP address
37     struct in_addr iph_destip;       //Destination IP address
38 };
```

```

39 #define IP_HL(ip) (((ip)->iph_lhl) & 0x0f) // 1111 0101 1011 0010 AND 0000 1111 = 0000
40 #define IP_V(ip) (((ip)->iph_ver) >> 4)
41 #define packet_size 512
42
43 unsigned short in_cksum(unsigned short *buf, int length)
44 {
45     unsigned short *w = buf;
46     int nleft = length;
47     int sum = 0;
48     unsigned short temp = 0;
49
50     /*
51      * The algorithm uses a 32 bit accumulator (sum), adds
52      * sequential 16 bit words to it, and at the end, folds back all
53      * the carry bits from the top 16 bits into the lower 16 bits.
54      */
55     while (nleft > 1)
56     {
57         sum += *w++;
58         nleft -= 2;
59     }
60
61     /* treat the odd byte at the end, if any */
62     if (nleft == 1)
63     {
64         *(u_char *)&temp = *(u_char *)w;
65         sum += temp;
66     }
67
68     /* add back carry outs from top 16 bits to low 16 bits */
69     sum = (sum >> 16) + (sum & 0xffff); // add hi 16 to low 16
70     sum += (sum >> 16); // add carry
71     return (unsigned short)(~sum);
72 }

```

```

74 void send_raw_ip_packet(struct ipheader *ip)
75 {
76     struct sockaddr_in dest_addr;
77     const int on = 1;
78     // Step 1: Create a raw network socket.
79     int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
80     if (sock < 0)
81     {
82         perror("socket() error");
83     }
84     // Step 2: Set socket option.
85     int se = setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));
86     if (se < 0)
87     {
88         perror("didn't allowed spoof");
89     }
90     // Step 3: Provide needed information about destination.
91     dest_addr.sin_family = AF_INET;
92     dest_addr.sin_addr = ip->iph_destip;
93
94     // Step 4: Send the packet out.
95     sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_addr, sizeof(dest_addr));
96     printf("spoofed packet sent :\n");
97     printf("    from : %s\n", inet_ntoa(ip->iph_sourceip));
98     printf("    to : %s\n", inet_ntoa(ip->iph_destip));
99     close(sock);
100    return;
101 }

```

```

103 void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
104 {
105     struct ethheader *eth = (struct ethheader *)packet;
106     if (ntohs(eth->ether_type) == 0x0800) // check if its type IP
107     {
108         // copy the data to an ip header, the buffer includes all the data we need
109         struct ipheader *ipHeader = (struct ipheader *) (packet + sizeof(struct ethheader));
110         // check the protocol
111         if (ipHeader->iph_protocol == IPPROTO_ICMP)
112         {
113             int size_ip = IP_HL(ipHeader) * 4;
114             struct icmpheader *newicmp = (struct icmpheader *) (packet + size_ip + sizeof(struct ethheader));
115             printf("%d \n", newicmp->icmp_type);
116             if (newicmp->icmp_type == 8)
117             {
118                 // Construct IP: swap src and dest in faked ICMP packet
119                 struct in_addr source = ipHeader->iph_sourceip;
120                 ipHeader->iph_sourceip = ipHeader->iph_destip;
121                 ipHeader->iph_destip = source;
122                 ipHeader->iph_ttl = 64;
123
124                 // Fill in all the needed ICMP header information.
125                 // ICMP Type: 8 is request, 0 is reply.
126                 newicmp->icmp_type = 0;
127                 newicmp->icmp_chksum = 0;
128
129                 int data_len = ntohs(ipHeader->iph_len) - sizeof(struct ipheader) - sizeof(struct icmpheader);
130                 printf("%d", data_len);
131                 newicmp->icmp_chksum = in_cksum((unsigned short *)newicmp, sizeof(struct icmpheader) + data_len);
132                 send_raw_ip_packet(ipHeader);
133                 return;
134             }
135         }
136     }
137     return;
138 }

```

```

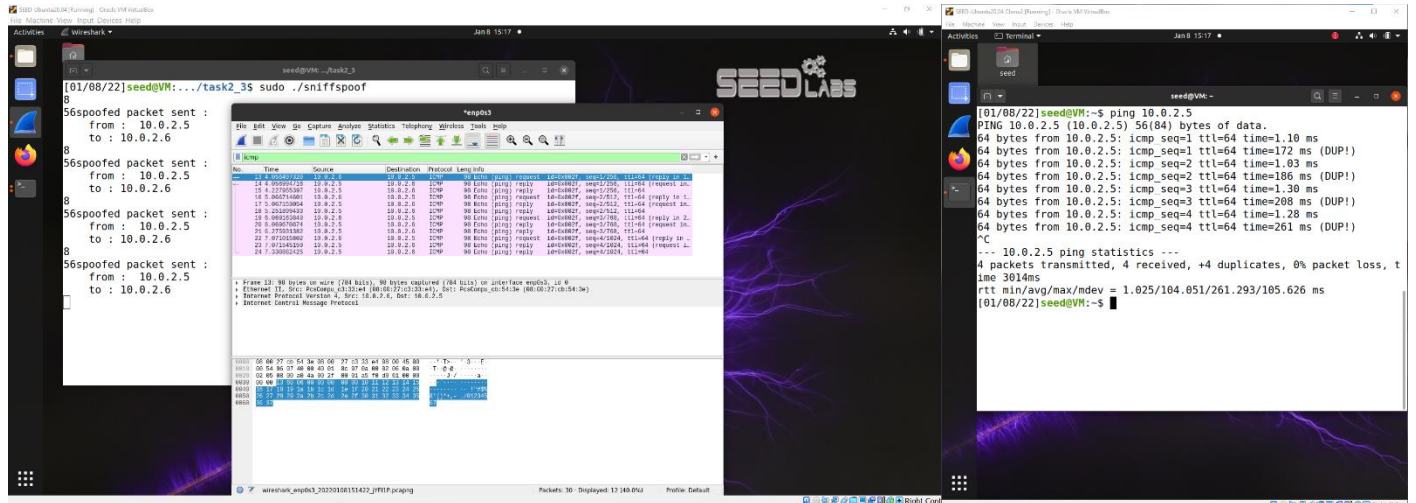
139
140 √ int main()
141 {
142     pcap_t *handle;
143     char errbuf[PCAP_ERRBUF_SIZE];
144     struct bpf_program fp;
145     char filter_exp[] = "icmp[icmptype] = icmp-echo";
146     bpf_u_int32 net;
147
148     // Step 1: Open live pcap session on NIC with name eth3
149     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf); // can be change to lo so the myping.c file will worke
150
151     // Step 2: Compile filter_exp into BPF psuedo-code
152     pcap_compile(handle, &fp, filter_exp, 0, net);
153     if (pcap_setfilter(handle, &fp) != 0)
154     {
155         pcap_perror(handle, "Error:");
156         exit(EXIT_FAILURE);
157     }
158
159     // Step 3: Capture packets
160     pcap_loop(handle, -1, got_packet, NULL);
161
162     pcap_close(handle); //Close the handle
163     return 0;
164 }

```

Explanation:

With the attacker machine that was in promiscuous we executed our sniff-spoof program.

The NIC captured all the packets that were relevant (decided by our filter), we modified the packets by swap between source and destination IP and send it buck to the original source as an echo replay.



```
[01/08/22]seed@VM:~/task2_3$ sudo ./sniffspoof
56spoofed packet sent :
from : 10.0.2.5
to : 10.0.2.6
56spoofed packet sent :
from : 10.0.2.5
to : 10.0.2.6
56spoofed packet sent :
from : 10.0.2.5
to : 10.0.2.6
56spoofed packet sent :
from : 10.0.2.5
to : 10.0.2.6
```

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.2.5	10.0.2.6	ICMP	64	Echo (ping) request 0x00000001 ttl=64
2	0.000000	10.0.2.6	10.0.2.5	ICMP	64	Echo (ping) reply 0x00000001 ttl=64
3	0.000000	10.0.2.5	10.0.2.6	ICMP	64	Echo (ping) request 0x00000002 ttl=64
4	0.000000	10.0.2.6	10.0.2.5	ICMP	64	Echo (ping) reply 0x00000002 ttl=64
5	0.000000	10.0.2.5	10.0.2.6	ICMP	64	Echo (ping) request 0x00000003 ttl=64
6	0.000000	10.0.2.6	10.0.2.5	ICMP	64	Echo (ping) reply 0x00000003 ttl=64
7	0.000000	10.0.2.5	10.0.2.6	ICMP	64	Echo (ping) request 0x00000004 ttl=64
8	0.000000	10.0.2.6	10.0.2.5	ICMP	64	Echo (ping) reply 0x00000004 ttl=64

```
Frame 1: 80 bytes on wire (640 bits), 80 bytes captured (640 bits) on interface vnet0, id 0
Ethernet II, Src: VxWorks (08:00:00:00:00:00), Dst: PcsNet (08:00:00:00:00:00)
Internet Protocol Version 4, Src: 10.0.2.5, Dst: 10.0.2.6
Internet Control Message Protocol
    Echo (ping) request 0x00000001 ttl=64
    0 60 70 00 54 3e 30 06 27 03 35 e4 30 00 45 50 7 7 5 1
    0010 02 05 80 00 00 00 00 00 00 00 00 00 00 00 00 00 1 2 8
    0020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 2 7
    0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 3 6
    0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 4 5
    0050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 5 4
    0060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 6 3
    0070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 7 2
    0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 8 1
```

```
[01/08/22]seed@VM:~$ ping 10.0.2.5
PING 10.0.2.5 (10.0.2.5) 56(84) bytes of data:
64 bytes from 10.0.2.5: icmp_seq=1 ttl=64 time=1.10 ms
64 bytes from 10.0.2.5: icmp_seq=2 ttl=64 time=1.03 ms
64 bytes from 10.0.2.5: icmp_seq=3 ttl=64 time=1.30 ms
64 bytes from 10.0.2.5: icmp_seq=4 ttl=64 time=208 ms
64 bytes from 10.0.2.5: icmp_seq=4 ttl=64 time=1.28 ms
64 bytes from 10.0.2.5: icmp_seq=4 ttl=64 time=261 ms (DUP!)
^C
--- 10.0.2.5 ping statistics ---
4 packets transmitted, 4 received, +4 duplicates, 0% packet loss, t
ime 3014ms
rtt min/avg/max/mdev = 1.025/104.051/261.293/105.626 ms
[01/08/22]seed@VM:~$
```