תקשורת

FINAL PROJECT

Packet Sniffing and Spoofing Lab

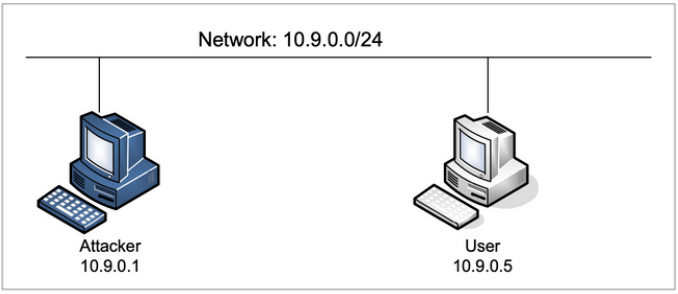
Nathanael Benichou-Jordan Perez
342769130 - 336165733

Packet Sniffing and Spoofing Lab

Packet sniffing and spoofing are the 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, Tcpdump, Netwox, 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, i.e., how packet sniffing and spoofing are implemented in software.

The objective of this lab is to master the technologies underlying most of the sniffing and spoofing tools. Students will play with some simple sniffer and spoofing programs, read their source code, modify them, perform a man in the middle attack, and eventually gain an in-depth understanding on the technical aspects of these programs. At the end of this lab, students should be able to write their own sniffing and spoofing programs.

Environment Setup using Container

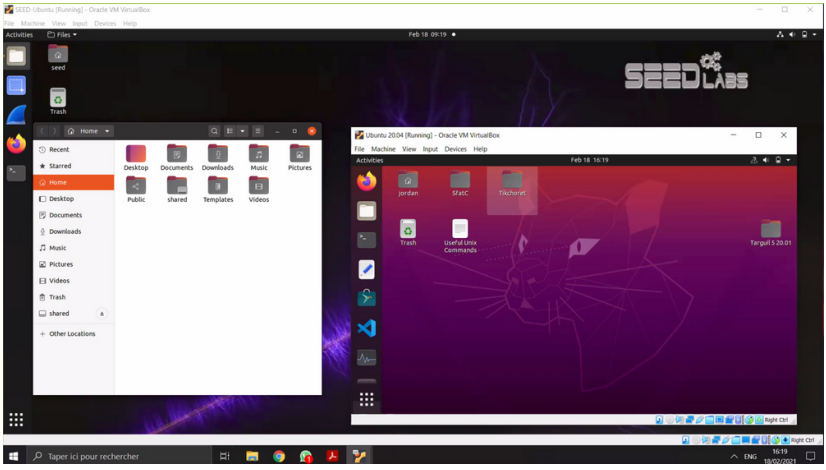


Lab Environment Setup:
Three Virtual Machines emulating Linux Ubuntu 20.04 (SEED Version)

1.VM (Seed attacker) - IP 10.0.2.7

2.VM SEED Ubuntu (Victim 1) - IP 10.0.2.8

3.VM SEED Ubuntu (Victim2) - IP 10.0.2.9



First of all we configured a shared folder for more convenience

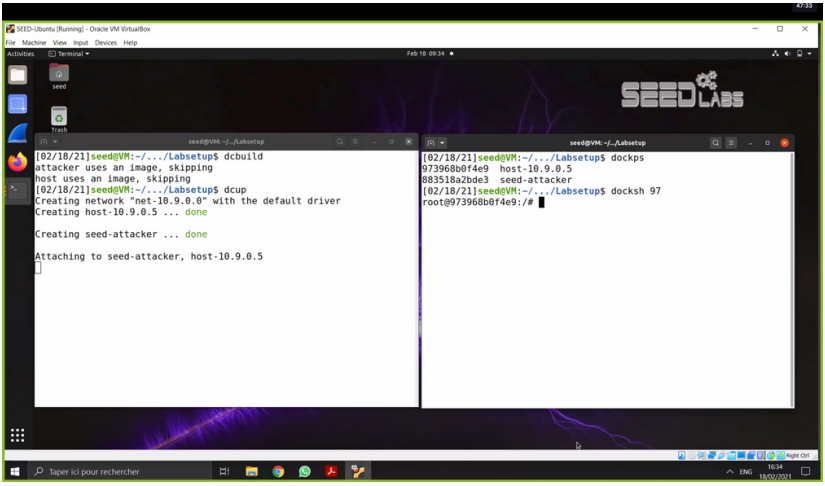
First contact with Dockers

Instead of using multiple VMs we could also use Dockers to emulate containers like this:

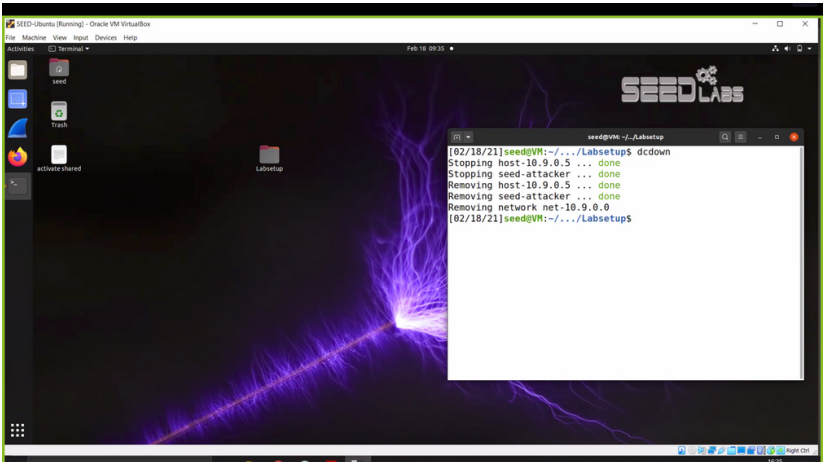
Commands used :

$ dockps

$ docksh <id>



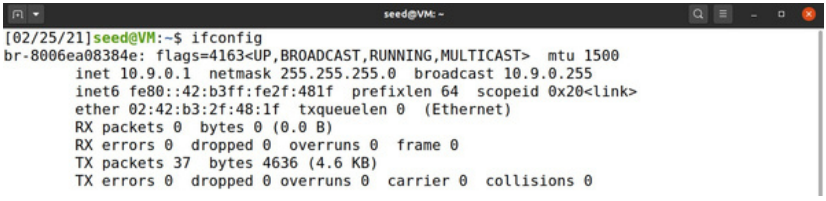
$ dcdown



3 Using Scapy to Sniff and Spoof Packets

3.1 Task 1.1. Sniffing packets

$ ifconfig



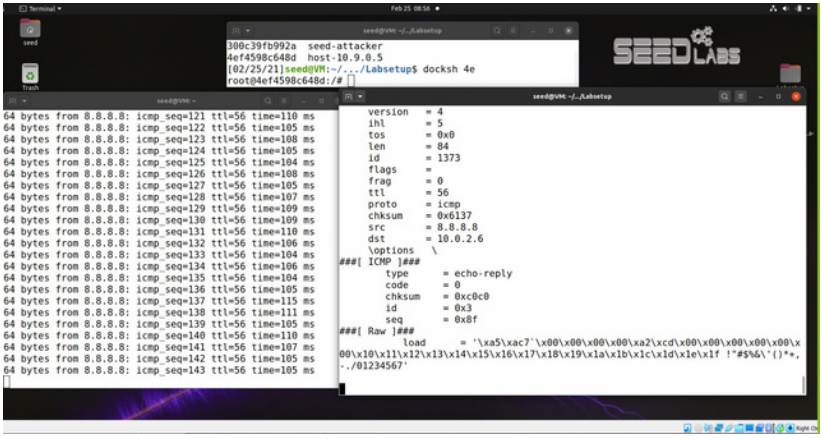
So the name of the Network is :

br-8006ea08384e

Python code to sniff icmp packets

Task 1.1.A

Running Scapy with sudo

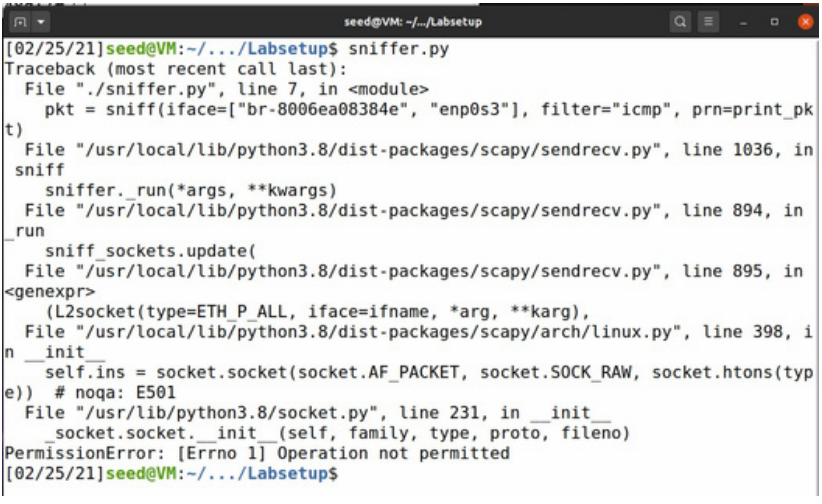


Running the python code of ICMP sniffing (using ID) in root

.

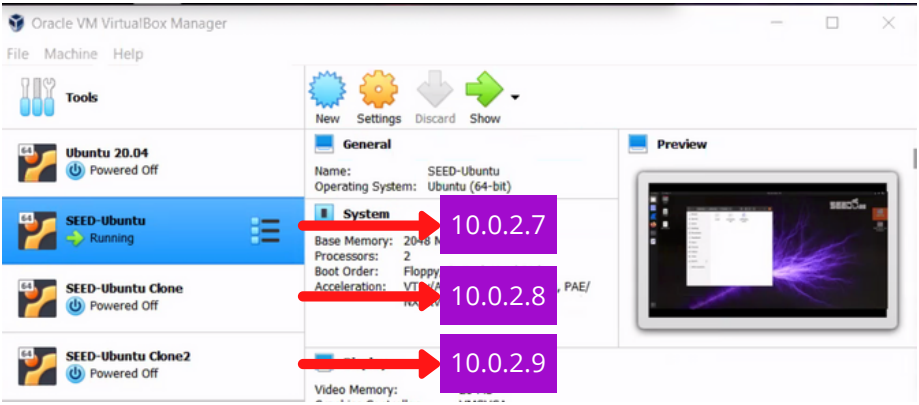
All the ICMP packets sent with the command ping 8.8.8.8 (Google DNS) are sniffed successfully

Running Scapy without sudo

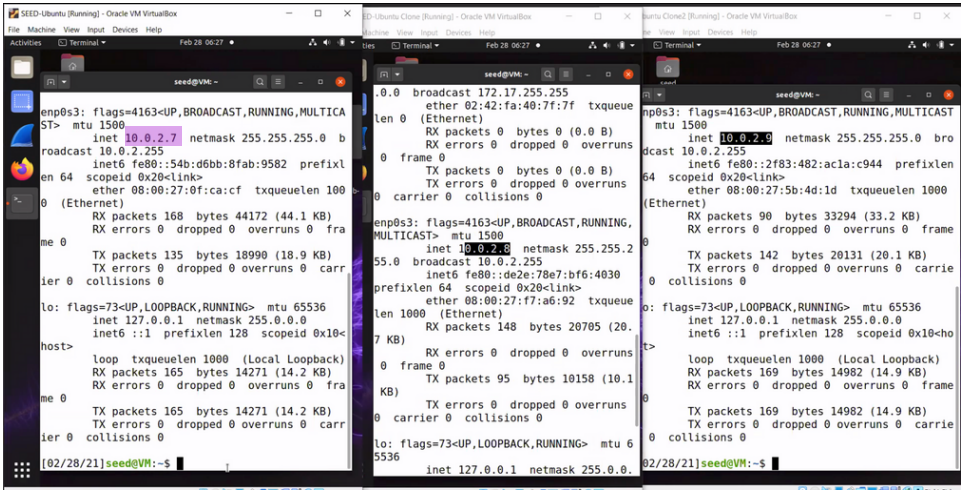


Core Dump/Segmentation fault is a specific kind of error caused by accessing memory that “does not belong to you.”

Using 3 VM (Seed) with different MAC Adresse Attached to NAT network



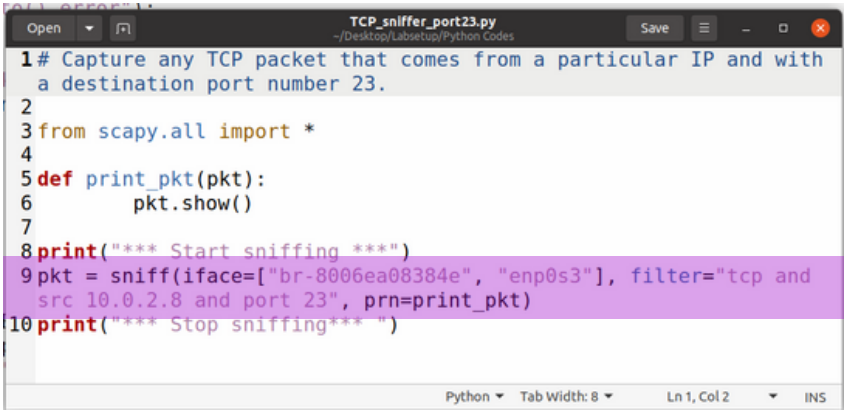
$ifconfig on the three VM



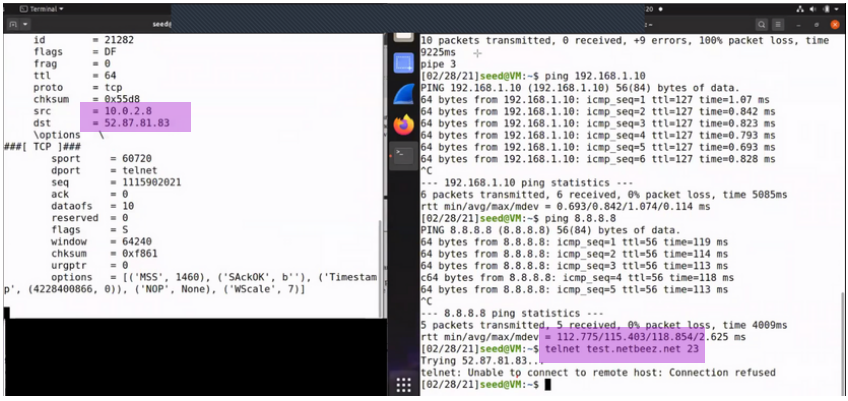
Task 1.1.B

- **Capturing only the ICMP packet: See 1.1.A**
- **Sniffing TCP from 10.0.2.8 with filter on port 23**
  (Using Telnet cmds to check it's working)

Code :



Result :



- **Capturing packets that comes from or that goes to a particular subnet.**
  You should not pick the subnet that your VM is attached to.

pcap filter host command:

Allowable primitives are:

dst host *host*
True if the IPv4/v6 destination field of the packet is *host*, which may be either an address or a name.
src host *host*
True if the IPv4/v6 source field of the packet is *host*.
host *host*
True if either the IPv4/v6 source or destination of the packet is *host*.
Any of the above host expressions can be prepended with the keywords, **ip**, **arp**, **rarp**, or **ip6** as in:

```
ip host host
```

which is equivalent to:

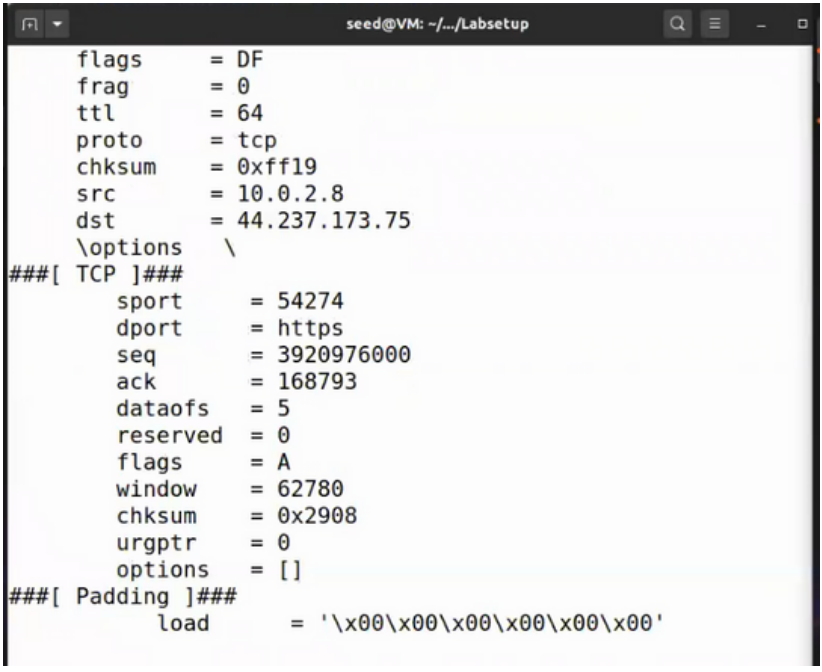
```
ether proto \ip and host host
```

If *host* is a name with multiple IPv4 addresses, each address will be checked for a match.

sudo python3 TCP_Sniffer_subnet.py

```
[02/28/21]seed@VM:~/.../Labsetup$ sudo python3 TCP_sniffer_subn
et.py
*** Start sniffing ***
```

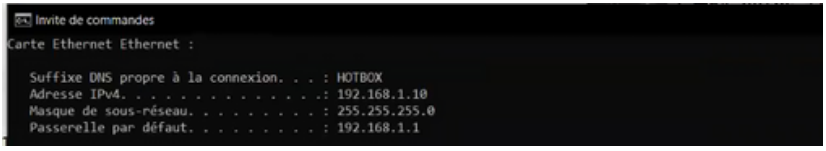
On VM 10.0.2.8 we open Firefox window and we can see that the packets were well sniffed



PS: *When we open a firefox window from VM 10.0.2.7 nothing happens as expected*

3.2 ICMP SPOOFER

Running Ipconfig on windows *(192.168.1.10)*



Task 1.2

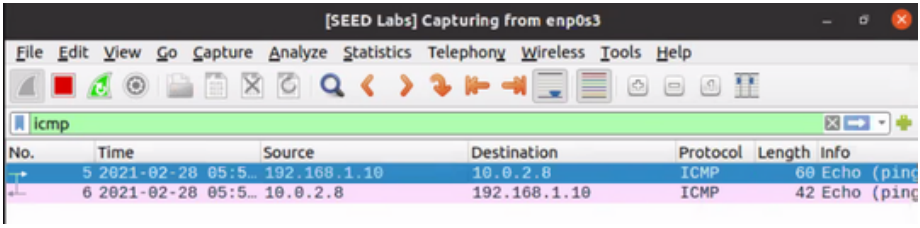
For this part we will need 3 VM.
We will run the following code on VM 10.0.2.7 which will be our man in the middle
The purpose is to spoof packets sent from one VM to another.
The ping will be issued by VM 10.0.2.8 to 10.0.2.9 using the command ping -c 1 10.0.2.9 which only issues a single ping request:

Open *ICMP_spoofer.py ~/Desktop/Labsetup Save

```
1 # ICMP Spoofer
2 from scapy.all import *
3
4 # Creates an IP object from the IP class
5 a = IP()
6
7 a.src = "10.0.2.8" # sets src address
8 a.dst = "192.168.1.10" # sets dst address
9 a.ttl = 10 # sets TTL
10
11 # Creates an ICMP object from the ICMP class
12 b = ICMP()
13
14 # The / operator is overloaded by the IP class, so
   it no longer represents division;instead, it means
   adding b as the payload field of a and modifying
   the fields of a accordingly. As a result, we get a
   new object that represent an ICMP packet
15 p = a/b
16
17 send(p)
18 ls(a)
```

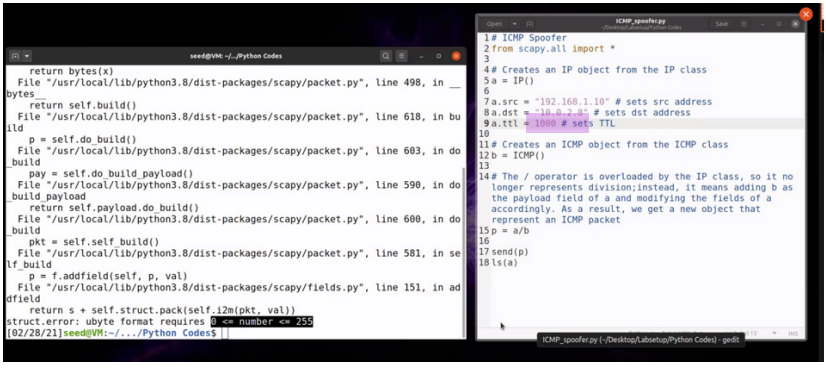
3.2 Task 1.2 Spoofing ICMP Packets

Using Wireshark on VM 10.0.2.9, we can see the ICMP request is successfully spoofed:
Although the request was sent by VM 10.0.2.7 it seems that it has been sent by 192.168.1.10 *(which is not true)*



Task 1.3: Traceroute

The objective of this task is to use Scapy to estimate the distance, in terms of number of routers, between your VM and a selected destination



We can see that with TTL=1000 we got an error message indicating that max TTL is 255

Task 1.4: Sniffing and-then Spoofing

For this part we will also need 3 VM.
We will run the following code on VM 10.0.2.7 which will be our man in the middle.
The purpose is to sniff and spoof packets sent from one VM to another.
The ping will be issued by VM 10.0.2.8 to 10.0.2.9 using the command
ping -c 1 10.0.2.9 which only issues a single ping request:

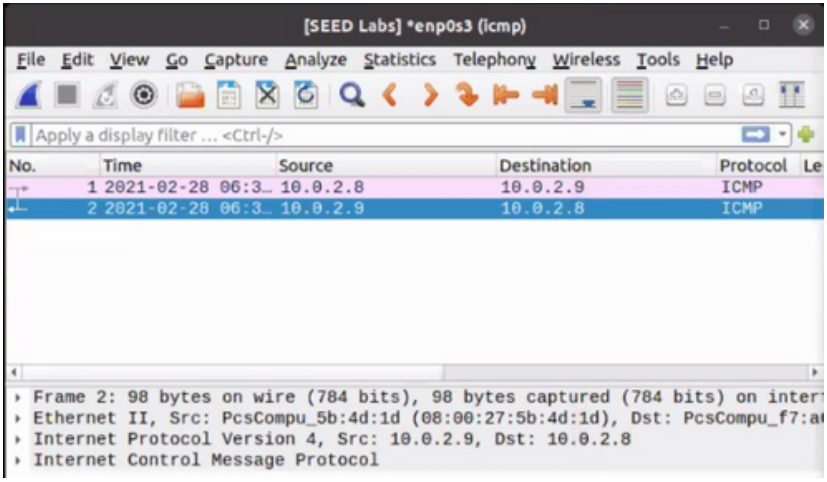
The following code is run on VM 10.0.2.7.
It sniffs the packet then spoofs it by swaping the source IP address with the dest IP address

Open ▾ sniffandspoof.py ~/Desktop/Labsetup/Python Codes Save

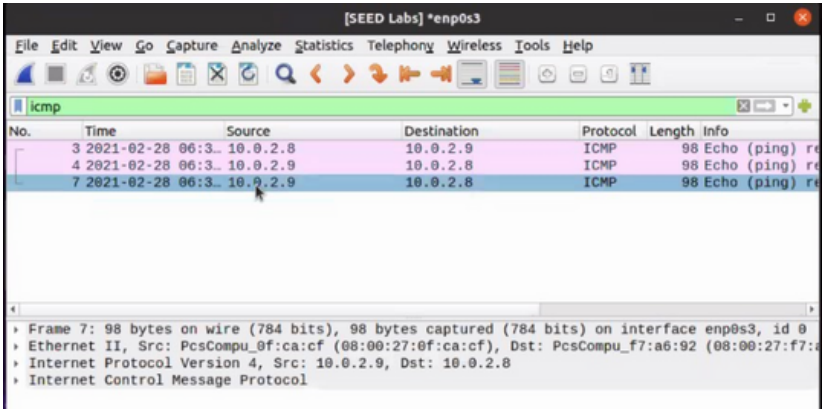
```
1 #!/usr/bin/python3
2 from scapy.all import *
3
4 def spoof_pkt(pkt):
5   if ICMP in pkt and pkt[ICMP].type == 8:
6       print("Original Packet.........")
7       print("Source IP : ", pkt[IP].src)
8       print("Destination IP :", pkt[IP].dst)
9
10      ip = IP(src=pkt[IP].dst, dst=pkt[IP].src,
    ihl=pkt[IP].ihl)
11      icmp = ICMP(type=0, id=pkt[ICMP].id,
    seq=pkt[ICMP].seq)
12      data = pkt[Raw].load
13      newpkt = ip/icmp/data
14
15      print("Spoofed Packet.........")
16      print("Source IP : ", newpkt[IP].src)
17      print("Destination IP :", newpkt[IP].dst)
18
19      send(newpkt,verbose=0)
20
21 pkt = sniff(filter='icmp and src host
    10.0.2.8',prn=spoof_pkt)
```

We are going to see the difference when we run a single ping request with and without spoofing ICMP:
*Using WireShark on VM **10.0.2.9***

Without spoofing



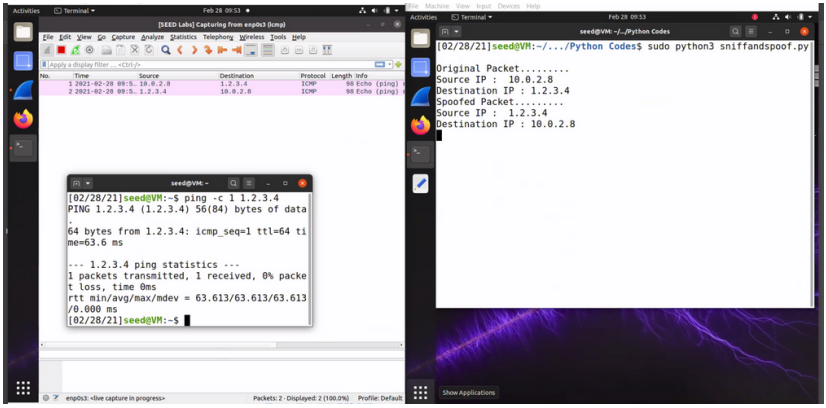
With spoofing



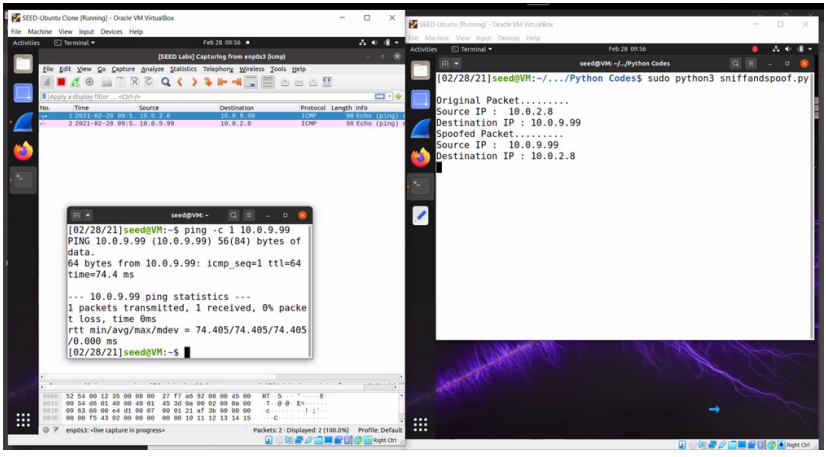
We can see that the sniff and spoof was successful: there is a third ICMP Echo Request from 10.0.2.9 to 10.0.2.8 as expected

```
ping 1.2.3.4      # a non-existing host on the Internet
ping 10.9.0.99    # a non-existing host on the LAN
ping 8.8.8.8      # an existing host on the Internet
```

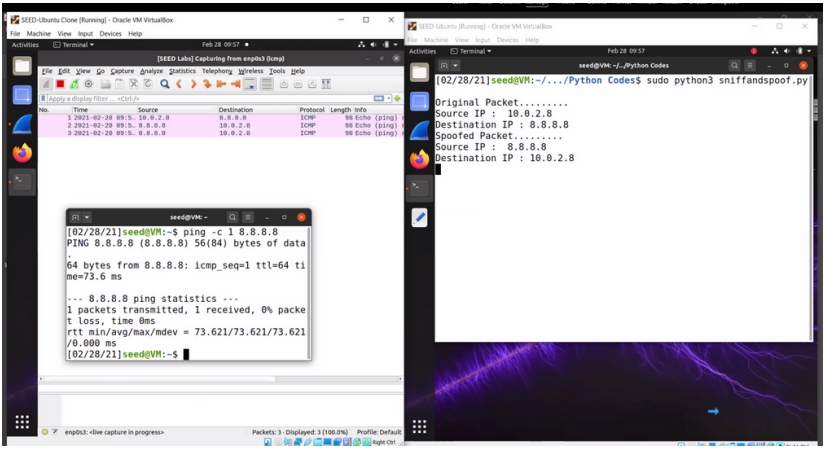
ping 1.2.3.4 *(non-existing host on the Internet)*



ping 10.0.9.99 *(non-existing host on the LAN)*



ping 8.8.8.8 *(an-existing host on the Internet)*

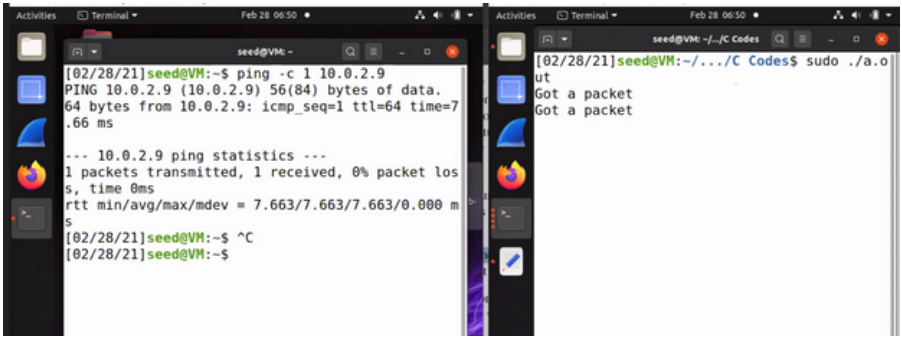


Conclusion: Even though ping requests were issued towards non-existing hosts, the victim still got answers (packets) from them.
It means that the spoof was successful: the attacker successfully altered the communications between the two parties and the victim sees a response from a host that truly doesn't exist.
And when the ping is issued towards an existing host on the Internet (here Google) the communication is also altered successfully (IP source and IP destination swapped)

4 Lab Task Set 2:
Writing Programs to Sniff and Spoof Packets

For this set up of tasks we will compile the C code inside the host VM

4.1 Task 2.1 Writing Packet Sniffing Program



Using the code of the pdf (sniff 4.1)

In this task, we need to write a sniffer program in C to print out the source and destination IP addresses of each captured packet

Task 2.1A: Understanding How a Sniffer Works

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.

The following calls are essential for this sniffer program:

- pcap_lookupdev : Find the default device on which to capture
- pcap_lookupnet : Is used to determine the IPv4 network number and mask associated with the network device.
- **pcap_open_live** : Open a device to start sniffing
- pcap_datalink : Get the link-layer header type to know what type
- **pcap_compile** : Compile a filter expression
- **pcap_setfilter** : Sets the compiled filter
- **pcap_loop** : Process packets from the capture
- pcap_freecode : Frees up allocated memory generated
- **pcap_close** : Closes the sniffing session

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?

We need root privilege to access to the Network Interface Card which needs admin authorization

For the following questions we are going to use the given main. We only changed eth3 to enp0s3:

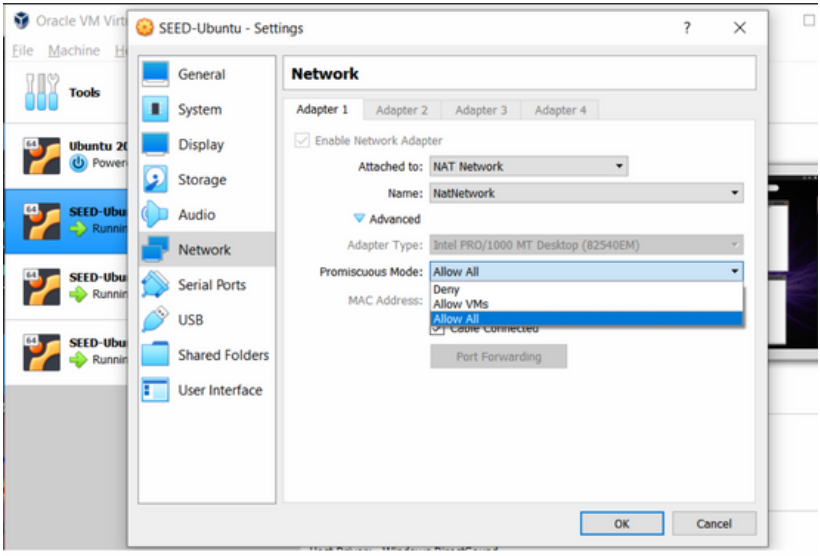
```c
int main()
        {
        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;
        char filter_exp[] = "ip proto icmp";
        bpf_u_int32 net;
        // Step 1: Open live pcap session on NIC with name eth3
        //Students needs to change "eth3" to the name
        //found on their own machines (using ifconfig).
        handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
        // Step 2: Compile filter_exp into BPF psuedo-code
        pcap_compile(handle, &fp, filter_exp, 0, net);
        pcap_setfilter(handle, &fp);
        // Step 3: Capture packets
        pcap_loop(handle, -1, got_packet, NULL);
        pcap_close(handle);
        return 0;
        //Close the handle
        }
```

<u>Quesiton 3. Please turn on and turn off the promiscuous mode in your sniffer program. Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this.</u>

For this part we will also need 3 VMs.

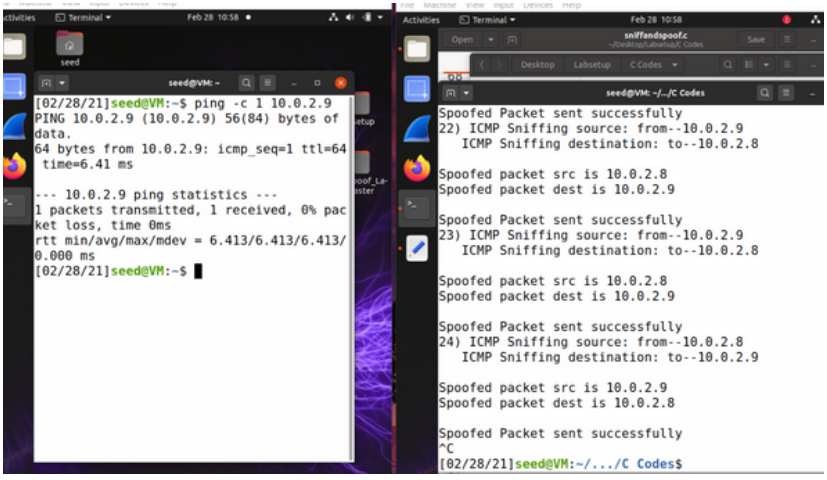
We will run the sniffandspoof.c code on VM 10.0.2.7 which will be our man in the middle but we will test the 3 different options we have for the promiscuous mode in VirtualBox (changes only for the VM being the man in the middle)

Ping requests are sent from VM 10.0.2.8 to VM 10.0.2.9):

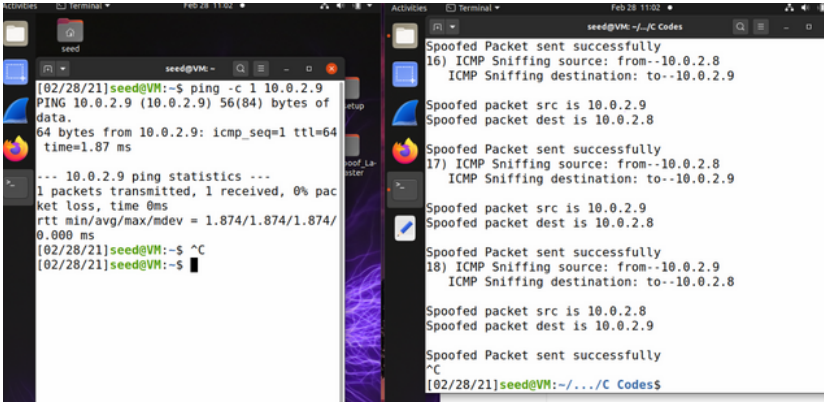


The available modes are: Deny, Allow VMs and Allow All

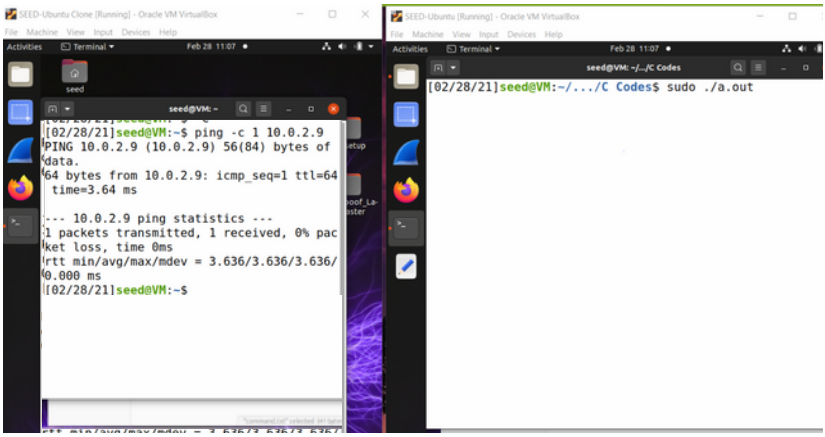
Running sniffandspoof on VM 10.0.2.7 with Promiscous mode Allow All



Running sniffandspoof on VM 10.0.2.7 with Promiscous mode Allow VMs



Running sniffandspoof on VM 10.0.2.7 with Promiscous mode Deny



We can see that in Deny mode absolutely no packets are sniffed nor spoofed.
The reason is because Promiscuous mode allows a network sniffer to pass all the traffic from a network controller and not just the traffic that the network controller was intended to receive.

PS: It's possible to manually switch between promiscuous modes directly in the code:

```
/* promisc mode on */
handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);

/* promisc mode off */
handle = pcap_open_live(dev, SNAP_LEN, 0, 1000, errbuf);
```

Task 2.1B: Writing Filters

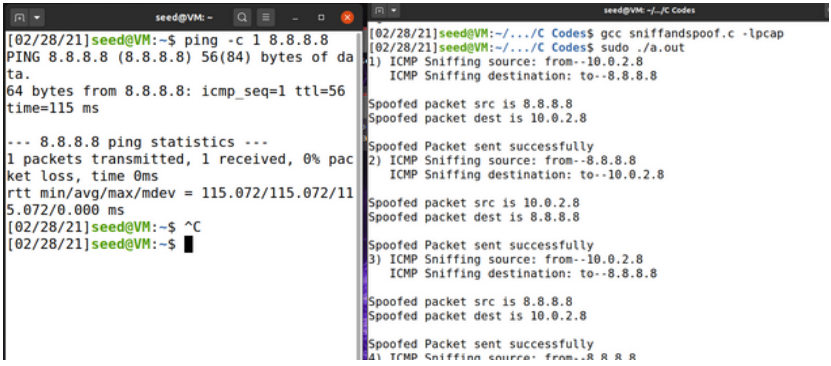
Writing filter expressions for the sniffer program to capture each of the followings:

- **Capture the ICMP packets between two specific hosts**

We can filter only the ICMP packets between two given hosts by modifying the filter_exp[] string this way:

```
int main()
        {
        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;
        // char filter_exp[] = "ip proto icmp";

        // ICMP packets between this host and 8.8.8.8
        char filter_exp[] = "icmp and (src host 10.0.2.8 and dst host 8.8.8.8) or (src host 8.8.8.8 and dst host 10.0.2.8)";

        bpf_u_int32 net;
        // Step 1: Open live pcap session on NIC with name eth3
        //Students needs to change "eth3" to the name
        //found on their own machines (using ifconfig).
        handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
        // Step 2: Compile filter_exp into BPF psuedo-code
        pcap_compile(handle, &fp, filter_exp, 0, net);
        pcap_setfilter(handle, &fp);
        // Step 3: Capture packets
        pcap_loop(handle, -1, got_packet, NULL);
        pcap_close(handle);
        return 0;
        //Close the handle
        }
```

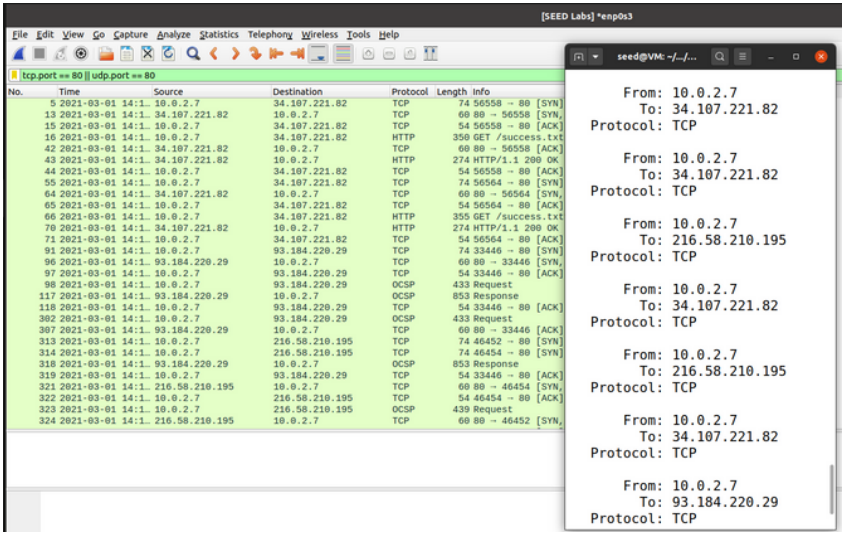
Spoofed succesfully with the filter



- **Capture the TCP packets with a destination port number in the range from 10 to 100.**

We can capture only the TCP packets with a destination port number in the given range from 10 to 100 by modifying the filter_exp[] string this way:

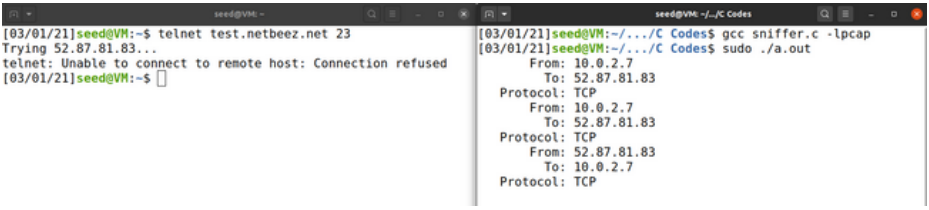
```
int main()
        {
        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;

        // TCP packets with dest port 10-100
        char filter_exp[] = "tcp dst portrange 10-100";


        bpf_u_int32 net;
        // Step 1: Open live pcap session on NIC with name eth3
        //Students needs to change "eth3" to the name
        //found on their own machines (using ifconfig).
        handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
        // Step 2: Compile filter_exp into BPF psuedo-code
        pcap_compile(handle, &fp, filter_exp, 0, net);
        pcap_setfilter(handle, &fp);
        // Step 3: Capture packets
        pcap_loop(handle, -1, got_packet, NULL);
        pcap_close(handle);
        return 0;
        //Close the handle
        }
```

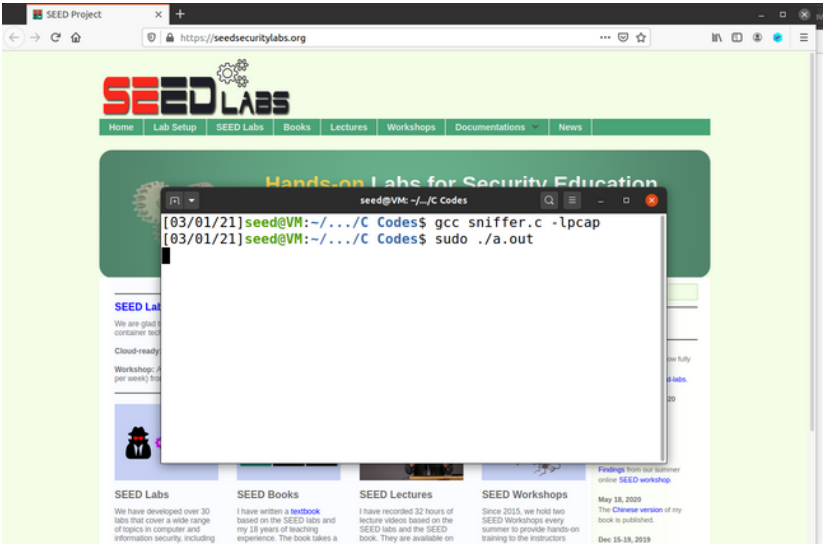


Task 2.1C: Sniffing Passwords

```c
int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;


    // TCP packets port 23
    char filter_exp[] = "tcp port 23";

    bpf_u_int32 net;

    // Step 1: Open live pcap session on NIC with name enp0s3
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

    // Step 2: Compile filter_exp into BPF pseudo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);
    pcap_setfilter(handle, &fp);

    // Step 3: Capture packets
    pcap_loop(handle, -1, got_packet, NULL);

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

Packets are well sniffed when using telnet (port 23) as expected





Nothing is captured when opening Firefox (since port 23 isn't used) as expected (Indeed, HTTP use port 80)

4.2 Task 2.2: Spoofing

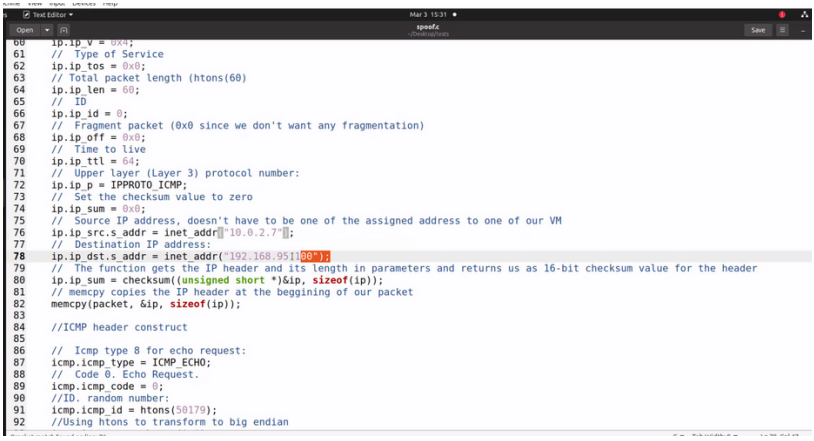
When a normal user sends out a packet, operating systems usually do not allow the user to set all the fields in the protocol headers (such as TCP, UDP, and IP headers). OSes will set most of the fields, while only allowing users to set a few fields, such as the destination IP address, the destination port number, etc.

However, if users have the root privilege, they can set any arbitrary field in the packet headers. This is called packet spoofing, and it can be done through raw sockets.

Raw sockets give programmers the absolute control over the packet construction, allowing programmers to construct any arbitrary packet, including setting the header fields and the payload. Using raw sockets is quite straightforward; it involves four steps: (1) create a raw socket, (2) set socket option, (3) construct the packet, and (4) send out the packet through the raw socket.

Task 2.2.A: Write a spoofing program.

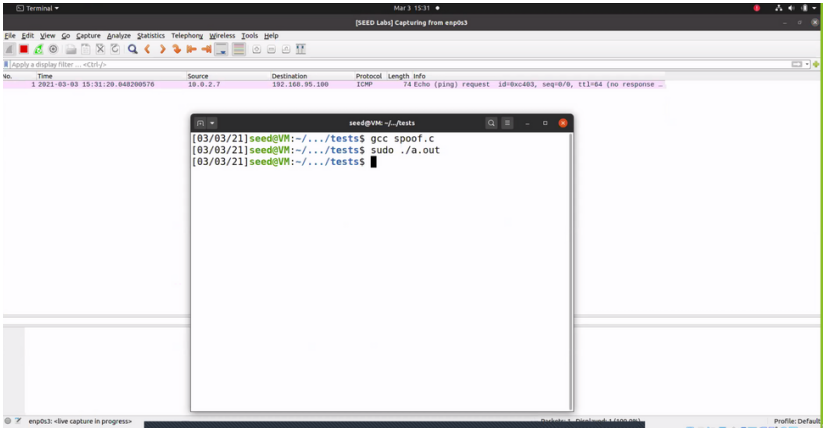
Spoof Code



```
60    ip.ip_v = 0x4;
61    //  Type of Service
62    ip.ip_tos = 0x0;
63    // Total packet length (htons(60)
64    ip.ip_len = 60;
65    //  ID
66    ip.ip_id = 0;
67    //  Fragment packet (0x0 since we don't want any fragmentation)
68    ip.ip_off = 0x0;
69    //  Time to live
70    ip.ip_ttl = 64;
71    //  Upper layer (Layer 3) protocol number:
72    ip.ip_p = IPPROTO_ICMP;
73    //  Set the checksum value to zero
74    ip.ip_sum = 0x0;
75    //  Source IP address, doesn't have to be one of the assigned address to one of our VM
76    ip.ip_src.s_addr = inet_addr("10.0.2.7");
77    //  Destination IP address:
78    ip.ip_dst.s_addr = inet_addr("192.168.95.100");
79    //  The function gets the IP header and its length in parameters and returns us as 16-bit checksum value for the header
80    ip.ip_sum = checksum((unsigned short *)&ip, sizeof(ip));
81    // memcpy copies the IP header at the beggining of our packet
82    memcpy(packet, &ip, sizeof(ip));
83
84    //ICMP header construct
85
86    //  Icmp type 8 for echo request:
87    icmp.icmp_type = ICMP_ECHO;
88    //  Code 0. Echo Request.
89    icmp.icmp_code = 0;
90    //ID. random number:
91    icmp.icmp_id = htons(50179);
92    //Using htons to transform to big endian
```

Task 2.2.B: Spoof an ICMP Echo Request.

Packet successfully spoofed:



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

The length field in this case should be the length of the IP packet, if not the sendto method (or function) will show an error 'Invalid Argument'. So yes, it is important that it is the length of the IP packet.

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

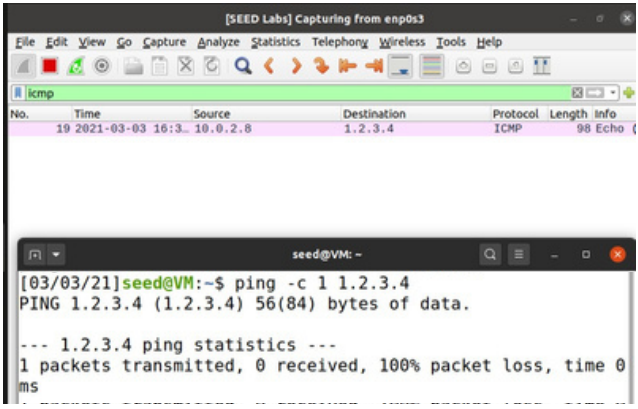
No you don't need to calculate the checksum, it will be filled by the system.

• **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?

It's restricted to root because it would break some rules for networking that are in place, without root you can't bind a port lower than 1024. With raw sockets you can simulate a server on any port and spoof custom packets which can interfere with inbound traffic

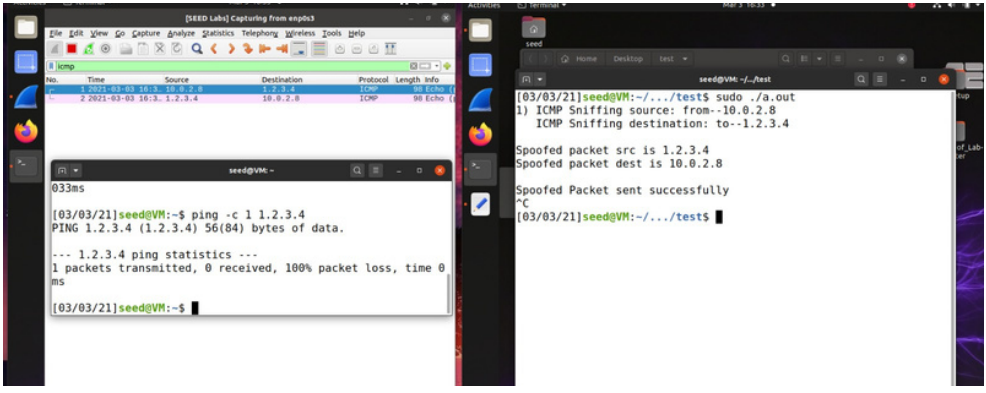
4.3 Task 2.3: Sniff and then Spoof

Sending a ping request to a non-existing host (1.2.3.4) before running sniffandspoof.c



no answer

Sending a ping request to a non-existing host (1.2.3.4) after running sniffandspoof.c



We can see that Wireshark displays a successful ping request even though no packet were really sent (100% packet loss), and in the attacker terminal we can also see that the source IP and the destination IP have been successfully swapped, meaning that the spoofing was successful.