# Packet Sniffing and Spoofing

Ethical Hacking Challenge #1

**Francesco Marchiori**

ID = 2020389

A.Y. 2021/2022

# Contents



# List of Figures

# 1 Introduction

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, Tcpdump, Netwox, Scapy, 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.

# 2 Environment Setup

The environment, as suggested in the instructions, consists in a virtual machine running `SEED-Ubuntu 20.04` on which are present three Docker containers, built and run using the `docker-compose.yml` file provided in the material.

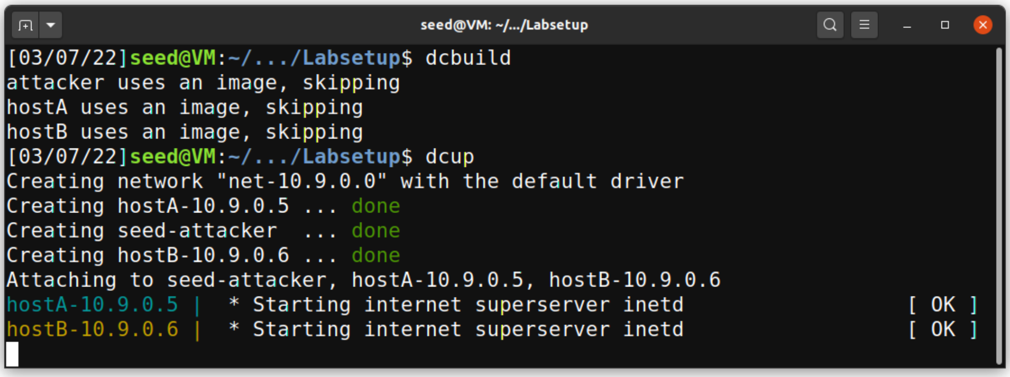


Figure 1: Setup of the Environment.

## 2.1 Network Interface Name

After launching two shells for each of the host, we now want to retrieve the network interface name, action that has been done by running the `docker network ls` command on the virtual machine shell.

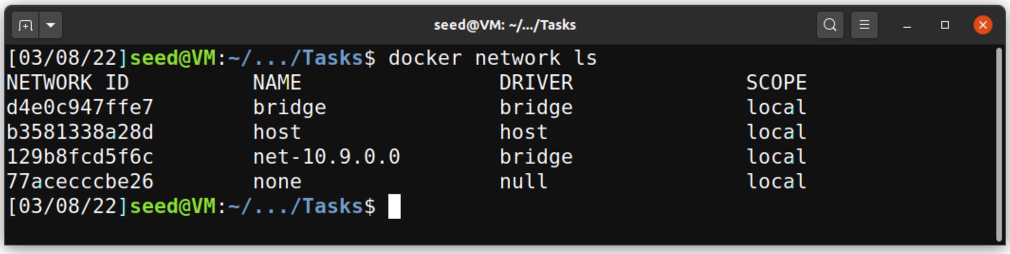


Figure 2: List of Docker network IDs.

As we can see, the name of the interface is `br-129b8fcd5f6c`.

# 3 Task 1

In this section, we will solve the given tasks regarding sniffing and spoofing.

## 3.1 Task 1.1: Using Scapy to Sniff and Spoof Packets

The code above will sniff the packets on the `br-129b8fcd5f6c` interface. For each captured packet, the callback function `print_pkt()` will be invoked; this function will print out some of the information about the packet.

```python
#!/usr/bin/env python3

from scapy.all import *

def print_pkt(pkt):
  print('[PACKET SNIFFED]')
  print('     [SOURCE]: {}'.format(pkt[IP].src))
  print('     [DESTINATION]: {}'.format(pkt[IP].dst))

print('[SNIFFING...]')
pkt = sniff(iface = 'br-129b8fcd5f6c', filter = 'icmp', prn = print_pkt)
```

We can now execute a `ping` command from the host A to the host B and see by running the sniffer on the virtual machine if we are able to see anything.

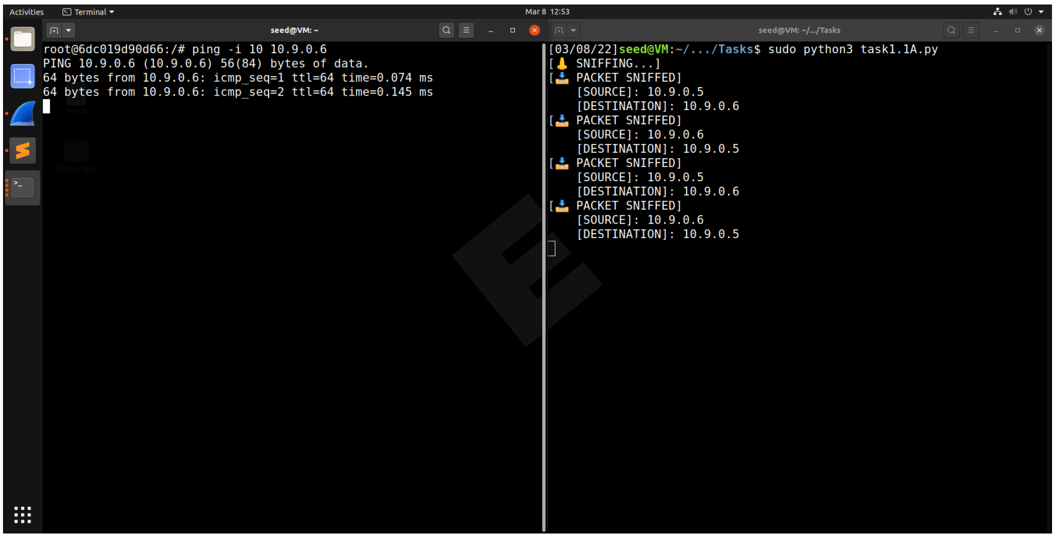


Figure 3: Sniffing on the interface.

As we can see, we are able to receive the packets. As a note, we used the `ping -i 10 10.9.0.6` just to be able to take a screenshot in time. Also, we are able to see both the ping and the reply.
Usually, when we sniff packets, we are only interested certain types of packets. We can do that by setting filters in sniffing. The code below allow to sniff only tcp packets on port 23 coming from the host with IP 10.9.0.5.

```python
#!/usr/bin/env python3

from scapy.all import *

def print_pkt(pkt):
  print('[PACKET SNIFFED]')
  print('     [SOURCE]: {}'.format(pkt[IP].src))
  print('     [DESTINATION]: {}'.format(pkt[IP].dst))

print('[SNIFFING...]')
pkt = sniff(iface='br-129b8fcd5f6c', filter='tcp port 23 and host 10.9.0.5', prn=print_pkt)
```

The results are shown in Fig. (4). As we can notice, we are able to see traffic, while with ping commands we wouldn't be able to see anything.

## 3.2 Task 1.2: Spoofing ICMP Packets

As a packet spoofing tool, Scapy allows us to set the fields of IP packets to arbitrary values. The objective of this task is to spoof IP packets with an arbitrary source IP address. We will spoof ICMP echo request packets,

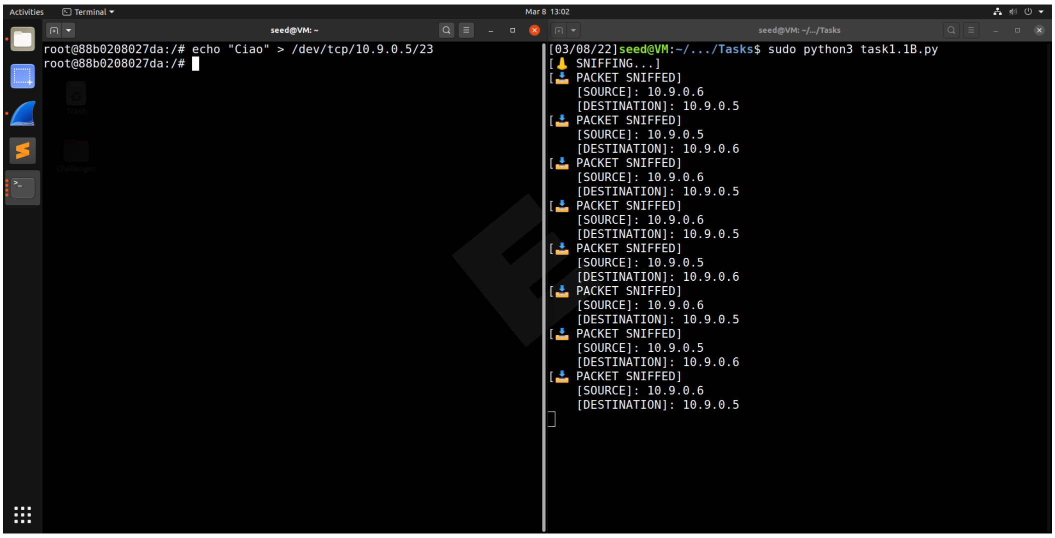


Figure 4: Sniffing on the interface with filters.

and send them to another VM on the same network. We will use Wireshark to observe whether our request will be accepted by the receiver. If it is accepted, an echo reply packet will be sent to the spoofed IP address. On the virtual machine, we are executing the following code:

```python
#!/usr/bin/env python3

from scapy.all import *

ip = IP()

src = '10.9.0.5'
dst = '10.9.0.6'

ip.src = src
ip.dst = dst

icmp = ip/ICMP()

print('[SENDING PACKET]')
print('    [SOURCE]: {}'.format(src))
print('    [DESTINATION]: {}'.format(dst))

send(icmp, verbose = 0)
print('[PACKET SENT]')
```

Since on Wireshark we are listening on the `br-129b8fcd5f6c` interface, we are using the IPs of the containers connected to it. Results are found in Fig. (5).

## 3.3 Task 1.3: Traceroute

The objective of this task is to use Scapy to estimate the distance, in terms of number of routers, between our VM and a selected destination. This is basically what is implemented by the traceroute tool. In this task, we will write our own tool. The idea is quite straightforward: just send an packet (any type) to the destination, with its Time-To-Live (TTL) field set to 1 first. This packet will be dropped by the first router, which will send us an ICMP error message, telling us that the time-to-live has exceeded. We can do this automatically using a simple loop in Python, as in the following code.

```python
#!/usr/bin/env python3

from scapy.all import *

dst = 'google.com'

# Starting from 1
ttl = 1

```

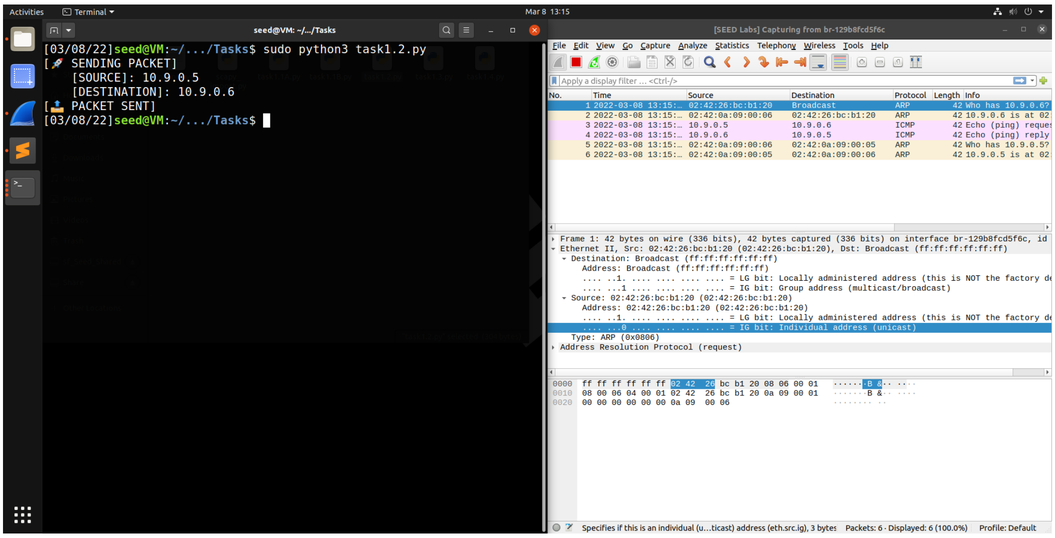


Figure 5: Spoofing ICMP Packets.

```python
while 1:
    pkt = sr1(IP(dst = dst, ttl = ttl)/ICMP(), verbose = 0) # Verbose handle the output
    # If the TTL is exceeded
    if pkt[ICMP].type == 11 and pkt[ICMP].code == 0:
        print('[TTL = {}] IP = {}'.format(ttl, pkt.src))
        ttl += 1
    # Otherwise we reached our destination
    elif pkt[ICMP].type == 0:
        print('[REACHED W/ TTL = {}] IP = {}'.format(ttl, pkt.src))
        break
```

The output is shown in Fig. (6) and as we can see we are able to determine the IP address of every machine inside the route between our virtual machine and the destination google.com, as well as the number of routers between them (in this case 10).

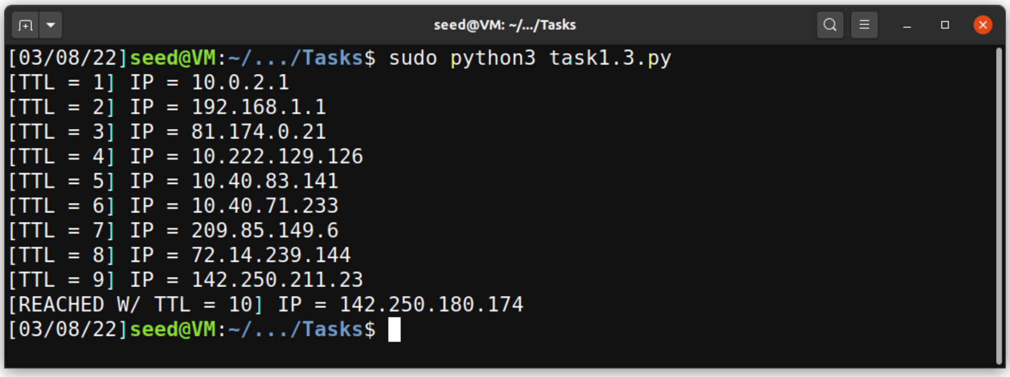


Figure 6: Traceroute with Scapy.

## 3.4 Task 1.4: Sniffing and-then Spoofing

In this task, we will combine the sniffing and spoofing techniques to implement the following sniff and then spoof program. From the user container, we ping an IP X. This will 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 VM, which monitors the LAN through packet sniffing. Whenever it sees an ICMP echo request, regardless of what the target IP address is, our program should immediately send out an echo reply using the packet spoofing technique. Therefore, regardless of whether machine X is alive or not, the ping program will always receive a reply, indicating that X is alive. The code used is the following:

```python
#!/usr/bin/env python3

from scapy.all import *

def spoof(pkt):
    """
    Given a received packet, spoof its address and send a reply
    """

    # Execute only if the packet is an echo request
    if pkt[ICMP].type != 8:
        return
    else:
        # Destination becomes source and viceversa
        spoof_ip = IP(src = pkt[IP].dst, dst = pkt[IP].src)

        spoof_icmp = ICMP(type = 0, id = pkt[ICMP].id, seq = pkt[ICMP].seq)

        spoof_data = pkt[Raw].load

        spoof_pkt = spoof_ip/spoof_icmp/spoof_data
        send(spoof_pkt, verbose = 0)

        print('[SPOOFED PACKET SENT]')
        return

print('[SNIFFING...]')
pkt = sniff(iface = 'br-129b8fcd5f6c', filter = 'icmp', prn = spoof)
```

We had to add the first `if pkt[ICMP].type != 8:` otherwise it would have spoofed also echo replies (if the type of ICMP is 8 then it's an echo request, else if it's 0 then it's an echo reply). When we spoof we IP the invert the source and the destination (since it's a reply) and we set the ICMP type to 0 (since it would be an echo reply). We also keep everything else untouched as the ICMP id and seq, and we further attach the data by calling `pkt[Raw].load`. First, let's see what happens when pinging a non-existing host on the Internet (1.2.3.4), a non-existing host on the LAN (10.9.0.99) and an existing host on the Internet (8.8.8.8).

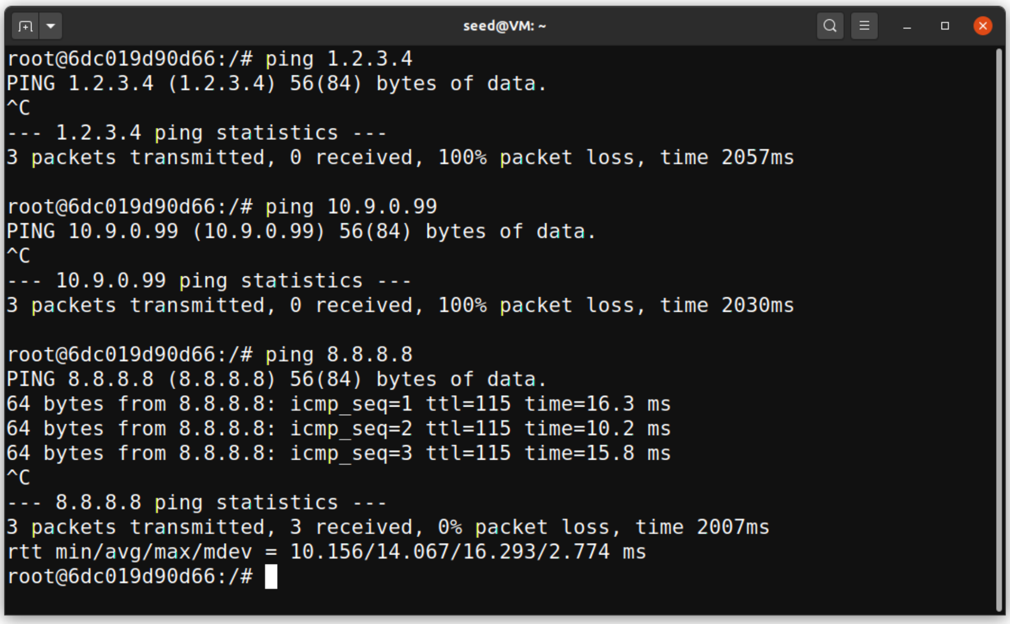


Figure 7: Examples of ping.

As we can expect, there are no packets received when we try to ping non existent IP (both on the internet or on the LAN), while we have responses when pinging and existing address. Let's know run our sniff-and-spoof program and see the output, which can be found in Fig. (8). We can see that when pinging a non-existing host on the Internet (1.2.3.4), out program is able to spoof the address and send replies which are correctly received by the host. When pinging a non-existing host on the LAN (10.9.0.99) however we have

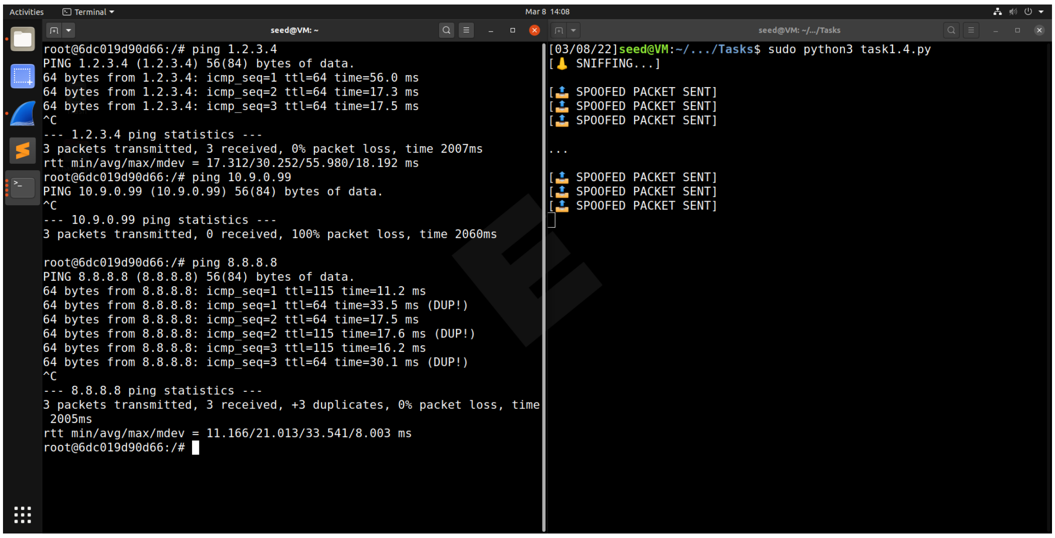


Figure 8: Examples of ping when running Sniffing and Spoofing.

no response: this happens because no ARP cache entry exists for a requested destination IP, and therefore the kernel will generate ARP requests until receiving an answer, thus failing our spoofing attack. Finally, when pinging an existing host on the Internet (8.8.8.8), we can see that the host receives a duplicate of the reply (marked with DUP!), since it's getting a reply both from the legitimate source and from our program.

## 4 Task 2

Even though I've little experience with the C language, I tried to complete the second task nonetheless. Using the `pcap` library, I tried to implement a sniffer that print out the source and destination IP addresses of each captured packet. This has been done by modifying the provided code in the following way.

```c
#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>

// Ethernet Header
struct ethheader {
    u_char  ether_dhost[6];    // Destination address
    u_char  ether_shost[6];    // Source address
    u_short ether_type;        // Type
};

// IP Header
struct ipheader {
  unsigned char      iph_ihl:4,      // IP header length
                     iph_ver:4;      // IP version
  unsigned char      iph_tos;        // Type of service
  unsigned short int iph_len;        // IP Packet length (data + header)
  unsigned short int iph_ident;      // Identification
  unsigned short int iph_flag:3,     // Fragmentation flags
                     iph_offset:13;  // Flags offset
  unsigned char      iph_ttl;        // Time to Live
  unsigned char      iph_protocol;   // Protocol type
  unsigned short int iph_chksum;     // IP datagram checksum
  struct  in_addr    iph_sourceip;   // Source IP address
  struct  in_addr    iph_destip;     // Destination IP address
};

/* This function will be invoked by pcap for each captured packet.
We can process each packet inside the function. */
void got_packet(u_char *args, const struct pcap_pkthdr *header,
const u_char *packet) {
    printf("[PACKET RECEIVED]\n");
    // Get the header information of Ethernet and IP
    struct ethheader *eth = (struct ethheader *)packet;
```

```c
    struct ipheader *ip_pkt = (struct ipheader *)(packet + sizeof(struct ethheader));
    printf("    [SOURCE IP]: %s \n", inet_ntoa(ip_pkt->iph_sourceip));
    printf("    [SOURCE IP]: %s \n", inet_ntoa(ip_pkt->iph_destip));
}

int main() {
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "icmp";
    bpf_u_int32 net;

    // Step 1: Open live pcap session on NIC with name br-129b8fcd5f6c.
    handle = pcap_open_live("br-129b8fcd5f6c", BUFSIZ, 1, 1000, errbuf);

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

    if (pcap_setfilter(handle, &fp) !=0) {
        pcap_perror(handle, "[ERROR]: ");
        exit(EXIT_FAILURE);
    }

    // Step 3: Capture packets
    pcap_loop(handle, -1, got_packet, NULL);
    pcap_close(handle);
    return 0;
    //Close the handle
}
// Note: don't forget to add "-lpcap" to the compilation command.
// For example: gcc -o sniff sniff.c -lpcap
```

The main contribution of this code is the addition of the first two struct, that define the different elements of the Ethernet and the IP header (even though we are only interested in the IP source address and the IP destination address). The proof of work is given in Fig. (9).

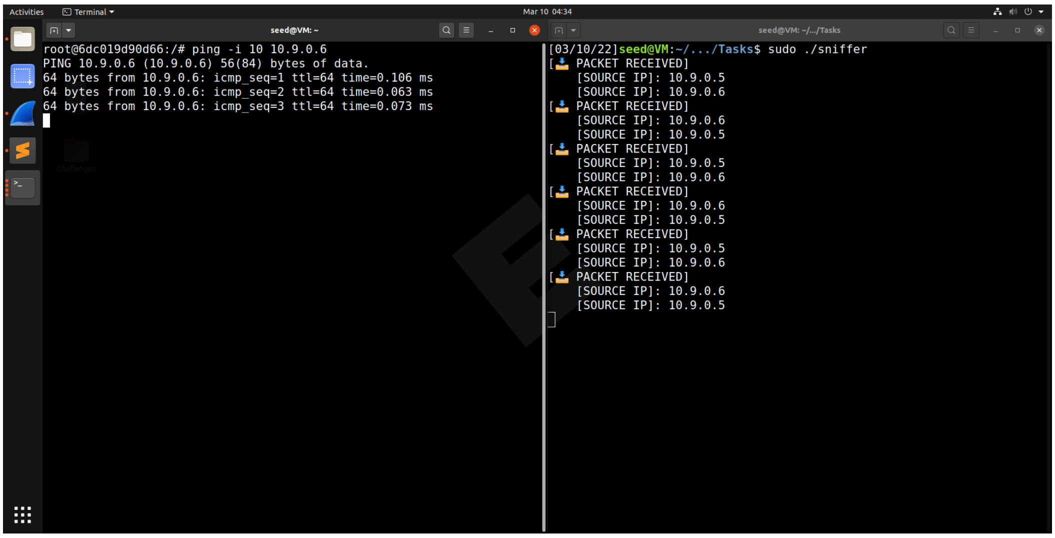


Figure 9: Sniffer in C.