Homework 1: Packet Sniffing and Spoofing Lab

1 Overview

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`, `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.

The objective of this lab is for students 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, 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.

2 Initial Lab Setup

Your need to first install either VirtualBox (`https://www.virtualbox.org/wiki/Download_Old_Builds_6_0`, please use version 6.0.4) or VMware player (available at `http://www.vmware.com/products/player/playerpro-evaluation.html`). Next you need to install `Ubuntu` Linux (any recent version) as a guest operating system on VMware virtual machine. You can download this OS (or pre-built image for VMware) from the Internet. You will execute the lab tasks using `Ubuntu` system (although the lab should work for many Linux distributions, we recommend you to use `Ubuntu` as we have tested the lab on it). For your convenience, we will provide a pre-built `Ubuntu` 16.04 Linux image for VMware. It is available at `https://drive.google.com/file/d/12l8OO3PXHjUsf9vfjkAf7-I6bsixvMUa/view?usp=sharing` (warning: it is large 3.3G). In this image, you can login using two accounts:

- User ID: root, Password: seedubuntu
- User ID: seed, Password: dees

3 Lab Tasks

3.1 Task 1: Writing Packet Sniffing Program

Sniffer programs can be easily written using the `pcap` library. With `pcap`, the task of sniffers becomes invoking a simple sequence of procedures in the `pcap` library. At the end of the sequence, packets will be put in buffer for further processing as soon as they are captured. All the details of packet capturing are handled by the `pcap` library. Tim Carstens has written a tutorial on how to use `pcap` library to write a sniffer program. The tutorial is available at `http://www.tcpdump.org/pcap.htm`. In this task, you need to read the tutorial, play with the program `sniffex` included in the tutorial, read the source code `sniffex.c`, and solve the following problems:

Understanding `sniffex`**.** Please download the `sniffex.c` program from the tutorial mentioned above, compile and run it. You should provide screendump evidence to show that your program runs successfully and produces expected results.

Problem 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.

1. Create a session handler. This session will specify what device (ethernet, in this case) to sniff on and whether promiscuous mode is activated. Use pcap_open_live() function to do that.
2. Create a network filter. The option filters are stored in struct bpf_program fp and compiled using pcap_compile() function.
3. After compilation, we will apply the options using pcap_setfilter() function.
4. There are 2 ways to sniff packets: capture a single packet or using loops to capture multiple packets
    a. Capturing single packets: we use pcap_next() function
    b. Capturing in loops: we use either pcap_loop() or pcap_dispatch(). In the sniffex example, he used pcap_loop. The difference between the 2 function is that pcap_dispatch() will only process the first batch of packets that it receives, while pcap_loop() will continue to process the rest of the packets.

Problem 2: Why do you need the root privilege to run `sniffex`? Where does the program fail if executed without the root privilege?

1. It fails at pcap_open_live() function. This function is probably a system call function that requires an escalated root privilege to execute.

Problem 3: Please turn on and turn off the promiscuous mode in the sniffer program. Can you demonstrate the difference when this mode is on and off? Please describe how you demonstrate this.

With promiscuous mode, running `ping 8.8.8.8` on another VM_B allows VM_A to sniff the packets made by VM_B.

An example of this can be seen in the image below:

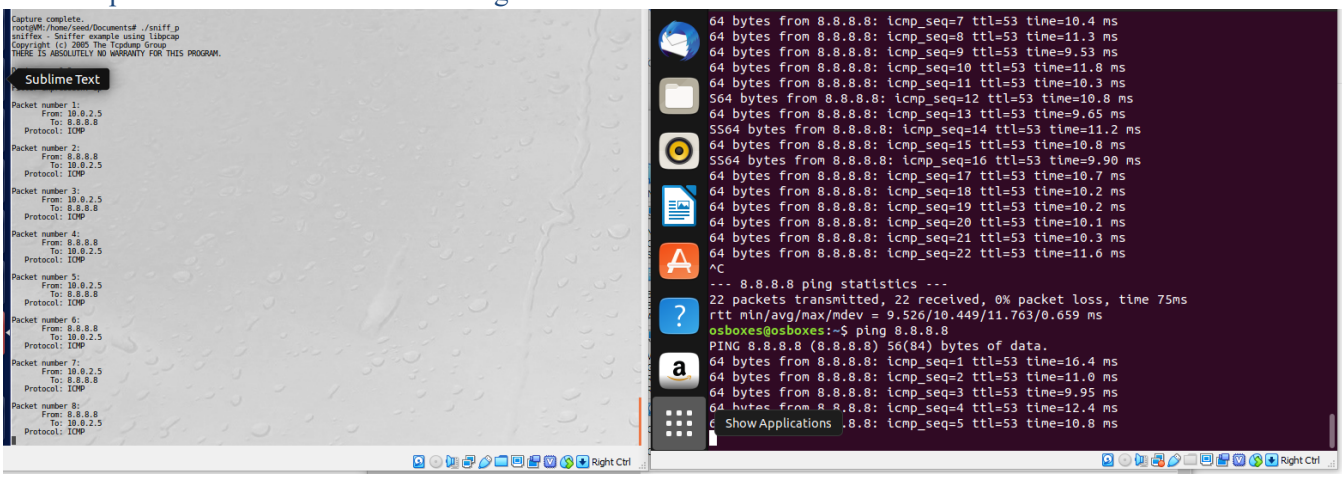

Figure 1: Results from sniffing in Promiscuous mode

The window on the left (VM_A) is sniffing in promiscuous mode. The window on the right (VM_B) is pinging 8.8.8.8. The result of the sniff can be seen in VM_A windows.

On the other hand, if non-promiscuous mode:

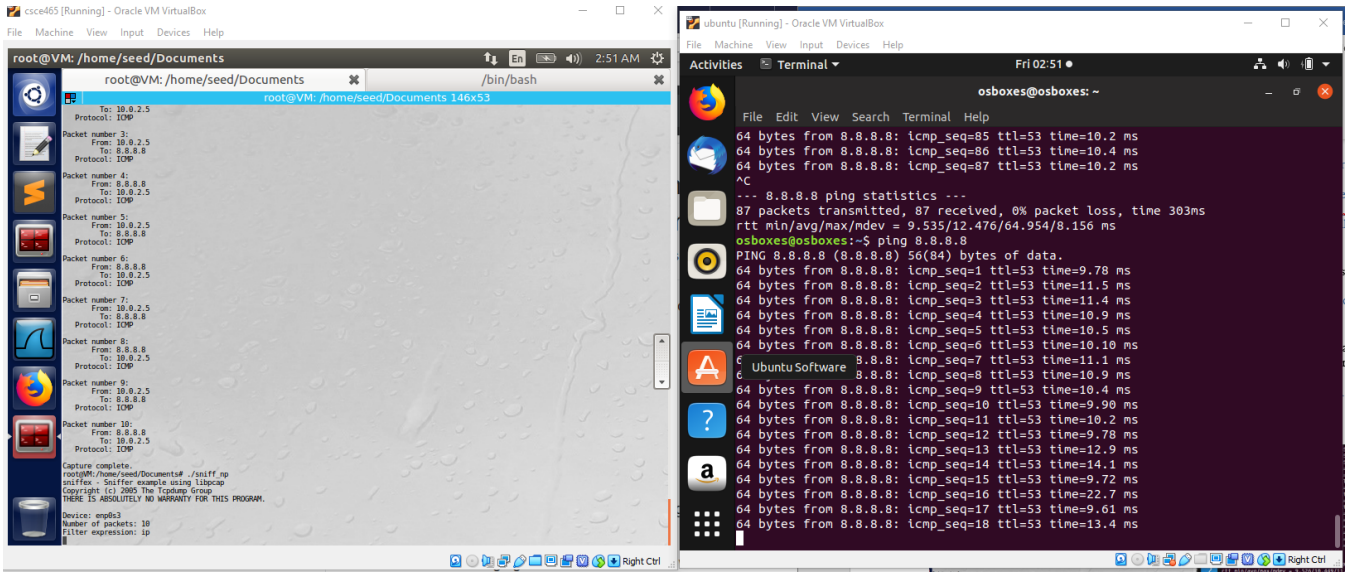

Figure 2: Results from sniffing in non-Promiscuous mode

The image above shows that even after several pings made by VM_B, there are 0 results made in VM_A's window.

Problem 4: Writing Filters Please write filter expressions to capture each of the followings. In your lab reports, you need to include screendumps to show the results of applying each of these filters.

- Capture the ICMP packets between two specific hosts.

```
4    char filter_exp[] = "ip";        /* filter expression [3] */
```

When the filter is "ip", browsing www.google.com from VM_B will give the result:

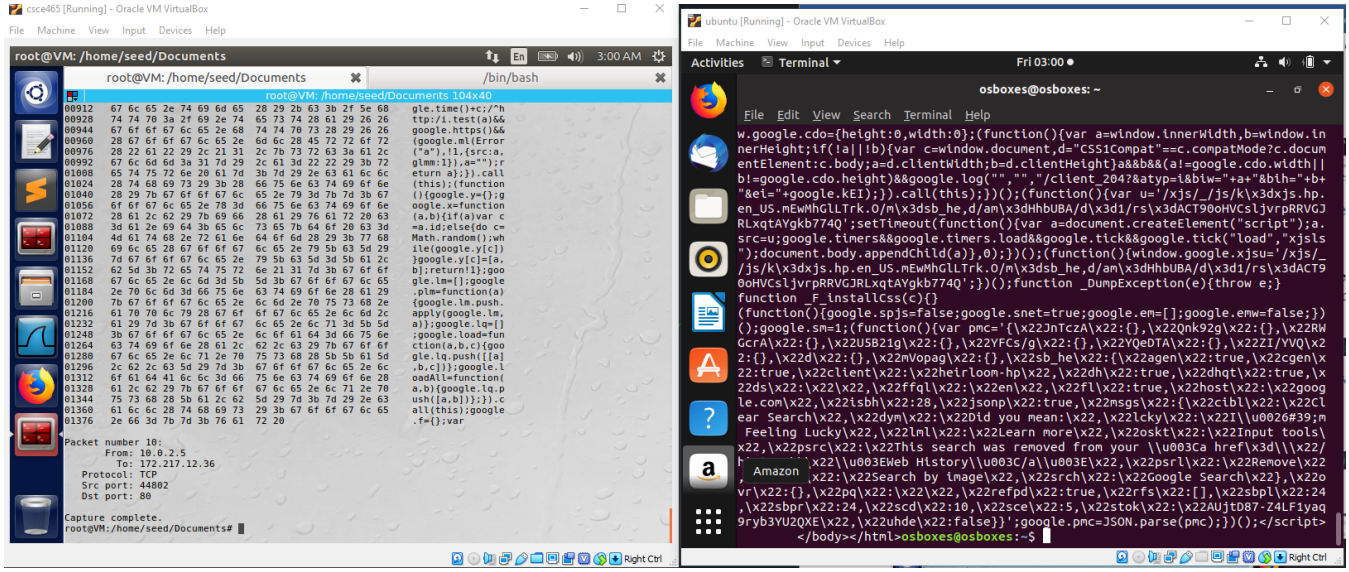

Figure 3: Sniff result when filter option = "ip"

Change the above code to

```
char filter_exp[] = "icmp";        /* filter expression [3] */
```

The result of the sniff can be seen in the image below:

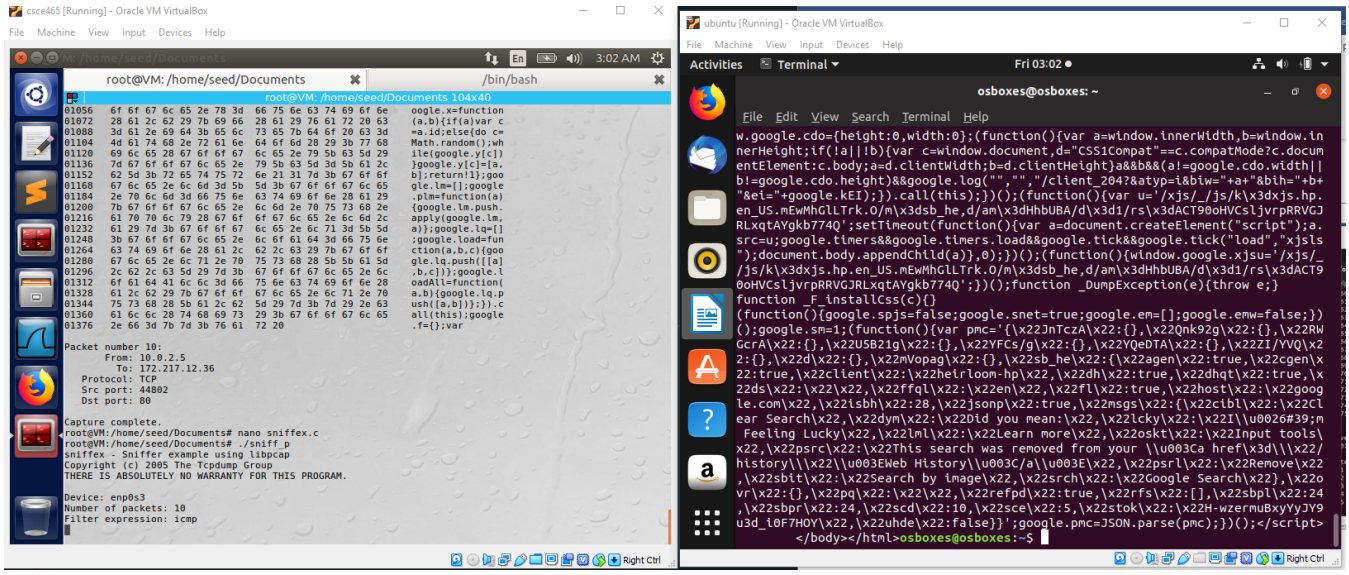

Figure 4: No packets sniffed even though `curl www.google.com` was performed

From Figure 4, it can be seen that despite performing `curl www.google.com` on VM_B, there is no packets being sniffed in VM_A.

However, once VM_B begins to `ping 8.8.8.8`, the VM_A starts to sniff ICMP packets made by VM_B, as seen in Figure 5 image below.

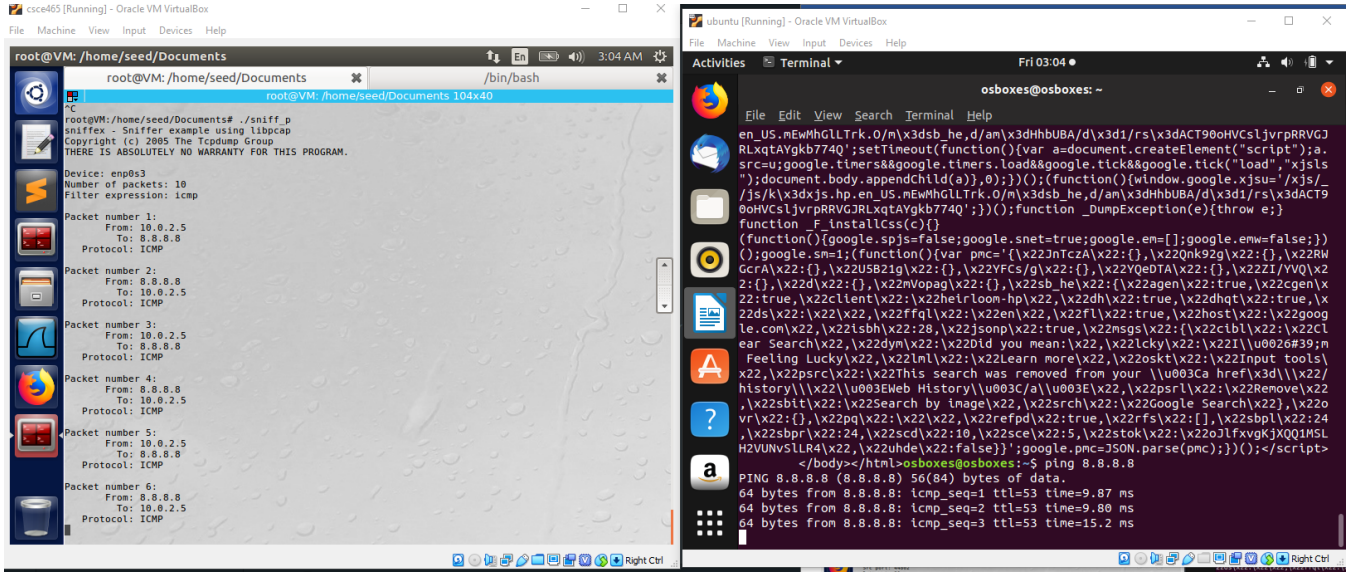

Figure 5: ICMP-only packets are traced when pinged

- Capture the TCP packets that have a destination port range from to port 10 - 100.

Change the code:

```
char filter_exp[] = "ip";        /* filter expression [3] */
```

to:

```
char filter_exp[] = "portrange 10-100";        /* filter expression [3] */
```

After which, we can test the sniffing similar to the above methods.

For VM_B, we will run ` nmap -p **10-100** 8.8.8.8` instead of the usual ping. This command basically tells the system to ping 8.8.8.8 from port 10 to port 100.

The image below, figure 6, depicts the results from sniffing VM_B.

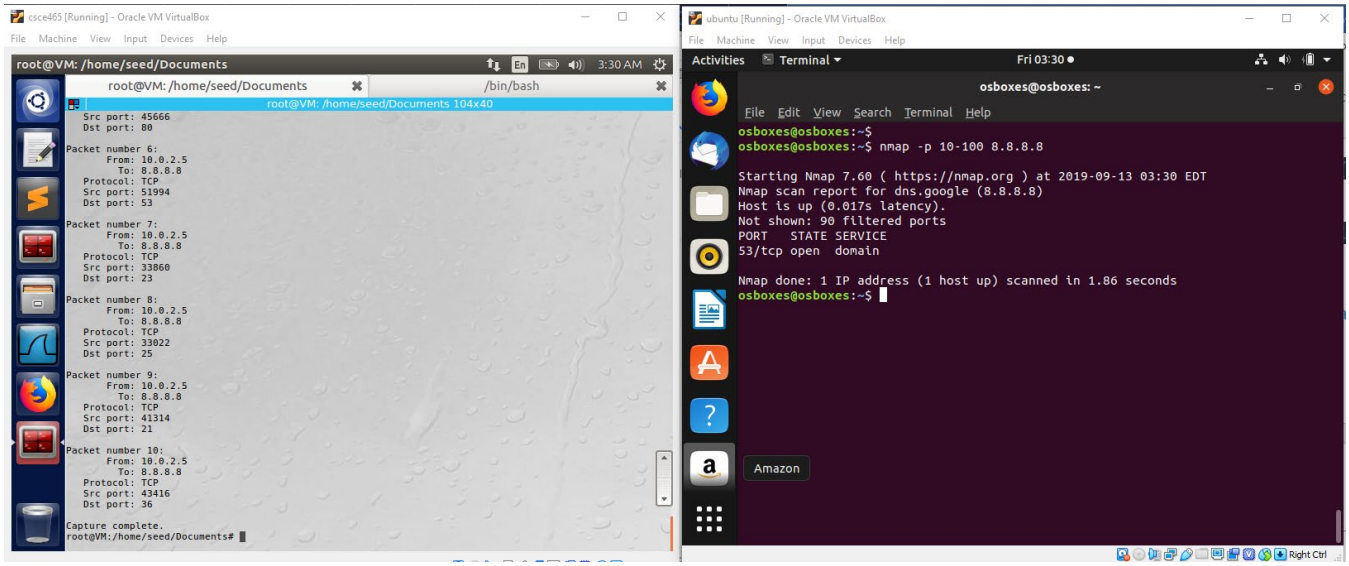

Figure 6: Sniffing with filter option: 'portrange 10-100'

Problem 5: Sniffing Passwords Please show how you can use `sniffex` to capture the password(s) when somebody is using `telnet` on the network that you are monitoring. You can start from modifying `sniffex.c` to implement the function. You also need to start the `telnetd` server on your VM. If you are using our pre-built VM, the `telnetd` server is already installed; just type the following command to start it.

```
% sudo service openbsd-inetd start
```

In addition to live traffic, your program should also support read from an offline trace file. For your convenience, we also provide a network trace that contains Telnet traffic for your test. It is available at `http://faculty.cse.tamu.edu/guofei/download/tfsession.pcap`.

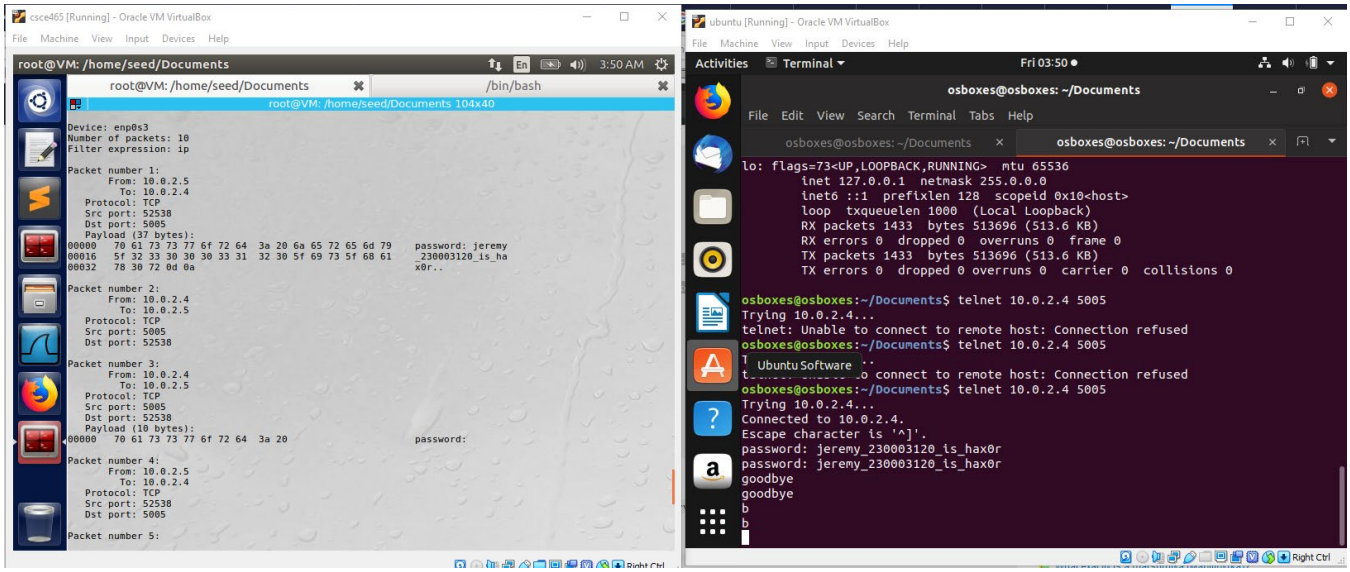

Figure 7: Sending password to another server

A python server was hosted to allow a user to connect using telnet. The source code of the python server can be found here (credits: https://apple.stackexchange.com/questions/14515/how-to-send-text-with-telnet-in-terminal):

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

import socket


TCP_IP = '127.0.0.1'
TCP_PORT = 5005
BUFFER_SIZE = 10  # Normally 1024, but we want fast response

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((TCP_IP, TCP_PORT))
s.listen(1)

conn, addr = s.accept()
print 'Connection address:', addr
while 1:
    data = conn.recv(BUFFER_SIZE)
    if not data: break
    print "received data:", data
    conn.send(data)  # echo
conn.close()
```

From Figure 7, VM_B attempts to connect to the python server, hosted by another VM, VM_C. VM_A was sniffing packets when VM_B sends its credentials (password and name) to the python server. The packet received by VM_A contains the password and name in plaintext, as seen in image.

4.2 Task 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, and the destination port number, etc. However, if the user has the root privilege, he/she can set any arbitary field in the packet headers. This is essentially 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. There are many online tutorials that can teach you how to use raw sockets in C programming. In this task, we will not focus on any specific tutorial. Your task is to read some of these tutorials, then either write your own packet spoofing program.

For your convenience, we show a simple skeleton of such a program.

```
int sd;
struct sockaddr_in sin;
char buffer[1024]; // You can change the buffer size

/* Create a raw socket with IP protocol. The IPPROTO_RAW parameter
 * tells the sytem that the IP header is already included;
 * this prevents the OS from adding another IP header.  */
sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
if(sd < 0) {
    perror("socket() error"); exit(-1);
}

/* This data structure is needed when sending the packets
 * using sockets. Normally, we need to fill out several
 * fields, but for raw sockets, we only need to fill out
 * this one field */
sin.sin_family = AF_INET;

// Here you can construct the IP packet using buffer[]
//    - construct the IP header ...
//    - construct the TCP/UDP/ICMP header ...
//    - fill in the data part if needed ...
// Note: you should pay attention to the network/host byte order.


/* Send out the IP packet.
 * ip_len is the actual size of the packet. */
if(sendto(sd, buffer, ip_len, 0, (struct sockaddr *)&sin,
             sizeof(sin)) < 0) {
     perror("sendto() error"); exit(-1);
}
```

Task 2.a: Write a spoofing program. Please write *your own* packet spoofing program in C. You need to provide evidences (e.g., Wireshark packet trace) to show us that your program successfully sends out spoofed IP packets.

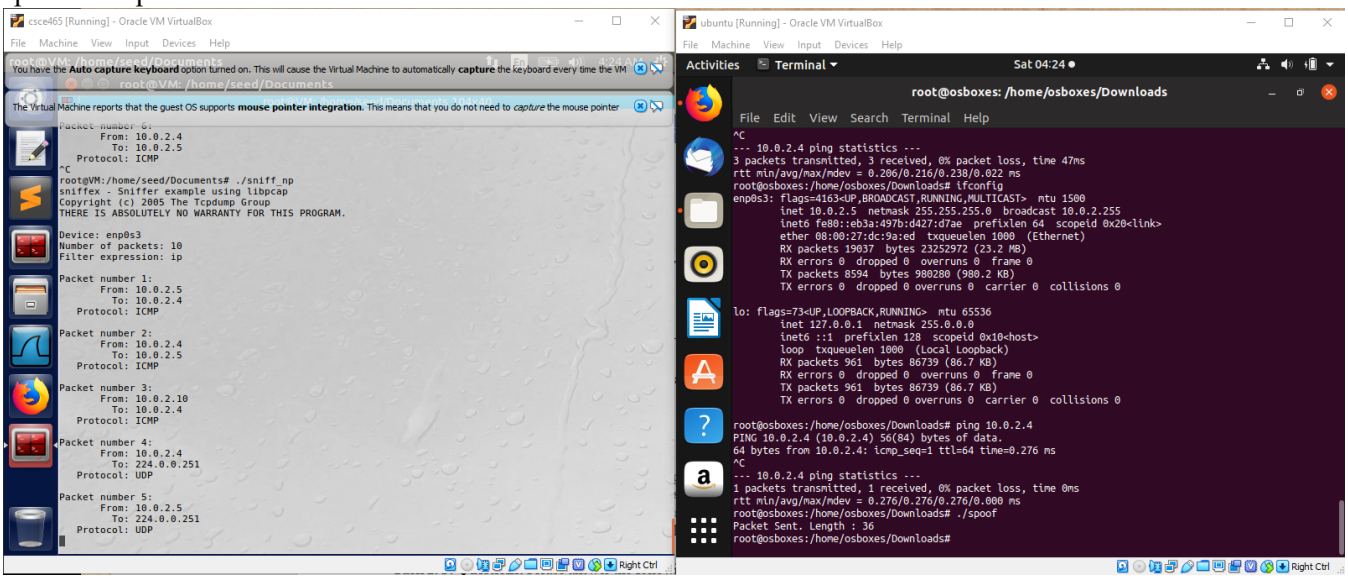

Figure 8: VM_A received packet from normal and spoofed user

Refer to Figure 8 to see that VM_A (left window) initially received ICMP packets from user with 10.0.2.5. This user can be seen in VM_B (right window), where the command `ping 10.0.2.4` command was executed.

After running the `spoof` program, it sends a packet of length 36 bits to VM_A from VM_B and a spoof IP address, 10.0.2.10. Ignore packet number 4 and 5 in VM_A.

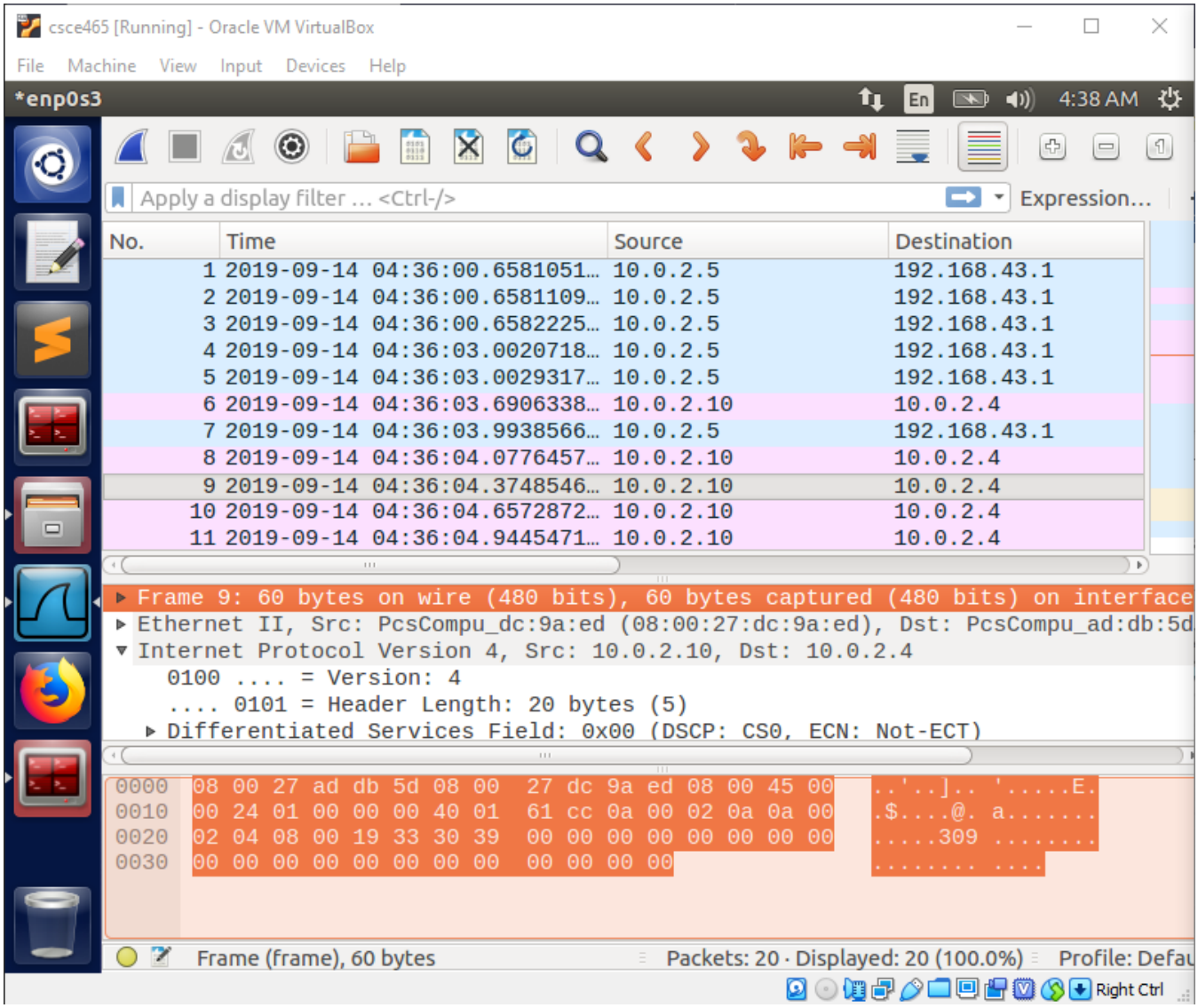

Figure 9: Wireshark result from spoofing

Figure 9 shows the result from spoofing caught in Wireshark.

Task 2.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). This packet should be sent to a remote machine on the Internet (the machine must be alieve). You should turn on your Wireshark, so if your spoofing is successful, you can see the echo reply coming back from the remote machine.

The image below shows the IP address of my Host Device, which is 192.168.1.79:

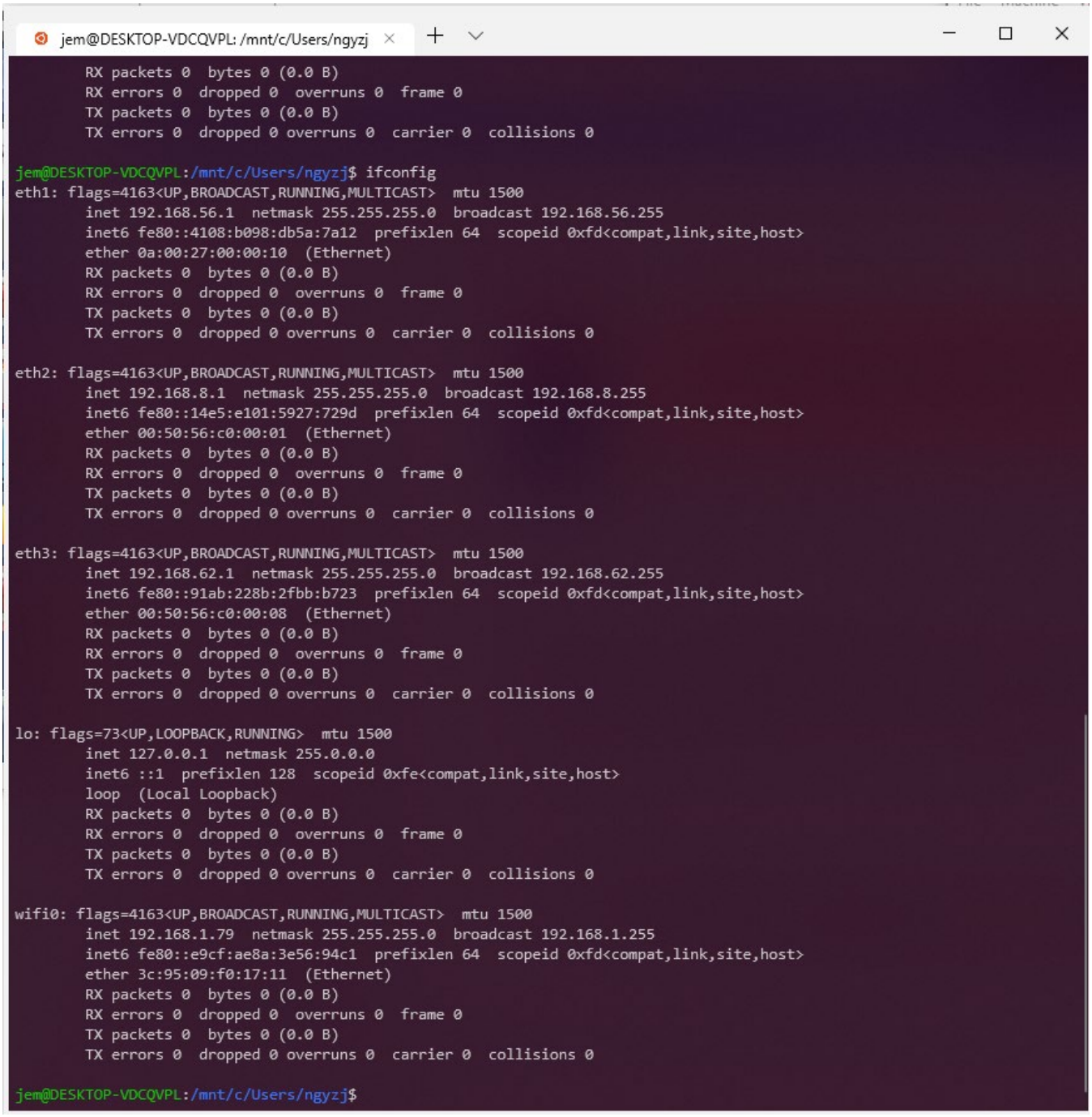

Figure 10: IP Address of Host Machine

I will use one of the VM to spoof my Host Machine's IP Address and display the results in image below:

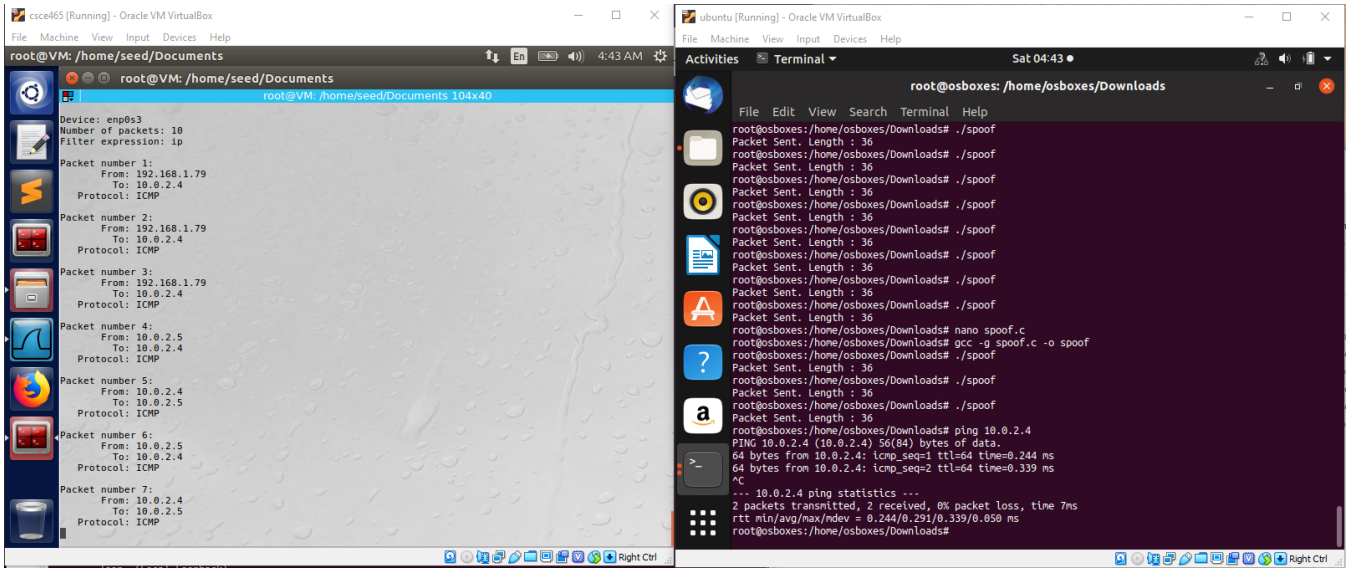

Figure 11: Spoofing Host Machine's IP Address result

On VM_B, it can be seen that it runs 'spoof' program 3 times. This is reflected in VM_A where is sniffed 3 packets from 192.168.1.79. To prove that VM_B is not actually ran on the host machine, I later ran `ping 10.0.2.4` command to show that the VM_B's IP address is actually 10.0.2.5!

Task 2.C: Questions. Please answer the following questions.

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

Yes, you may, since it is not important to the other libraries.

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

Yes, of course.

3. 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?

Because the socket libraries require system call functions to have access to lower level protocols (Transport, Network, Data Link, and Physical). System calls require admin/root privilege since no access to kernel space is given to user space execution.

It fails at socket() function, as seen in the image below:

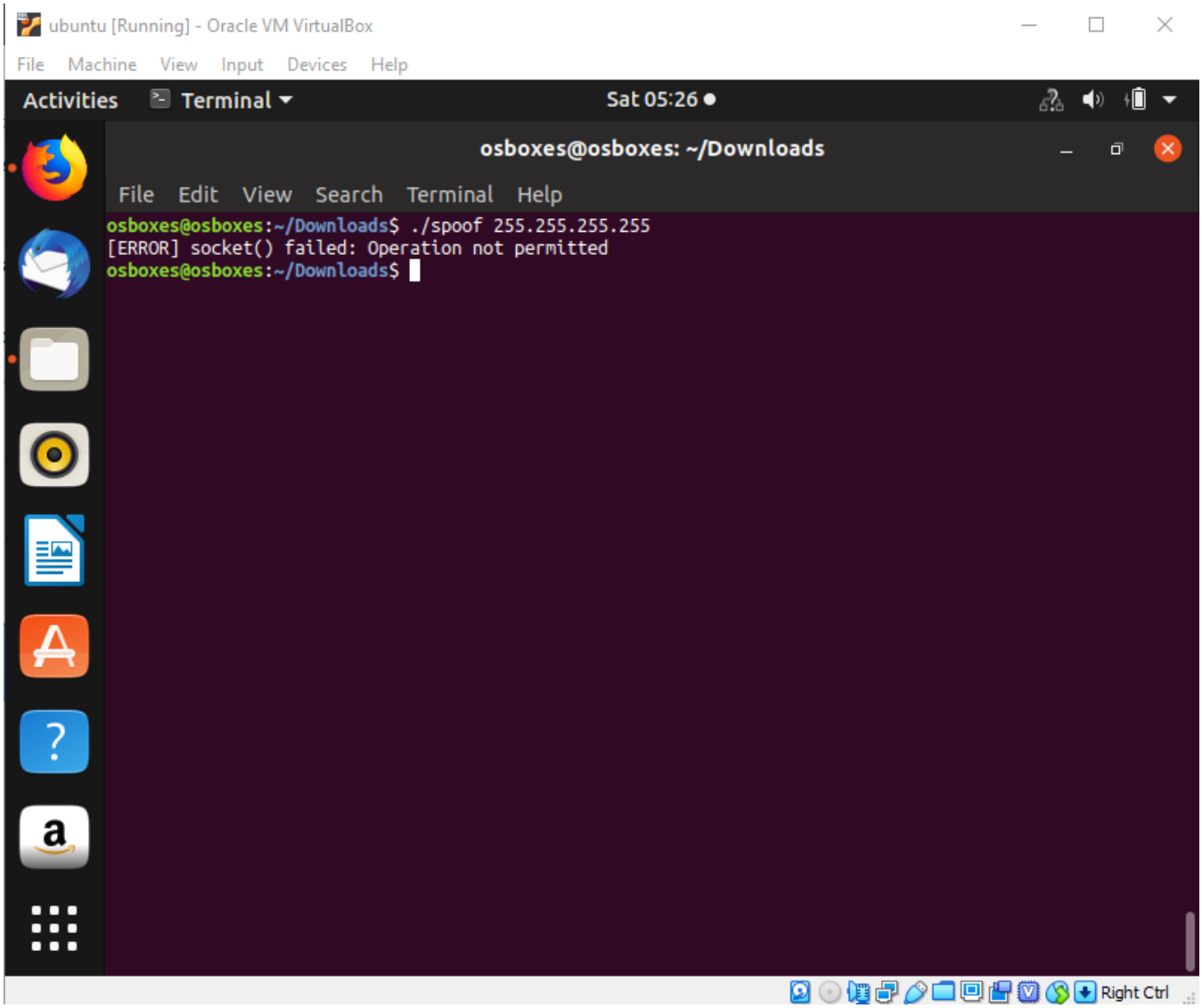

Figure 12: socket() function not accessible from user space execution

4.3 Task 3: Sniff and then Spoof

In this task, you will combine the sniffing and spoofing techniques to implement the following sniff-and-then-spoof program. You need two VMs on the same LAN. From VM A, you `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. Your sniff-and-then-spoof program runs on VM B, which monitors the LAN through packet sniffing. Whenever it sees an ICMP echo request, regardless of what the target IP address is, your 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. You need to write such a program, and include screendumps in your report to show that your program works. Please also attach the code (with adequate amount of comments) in your report.

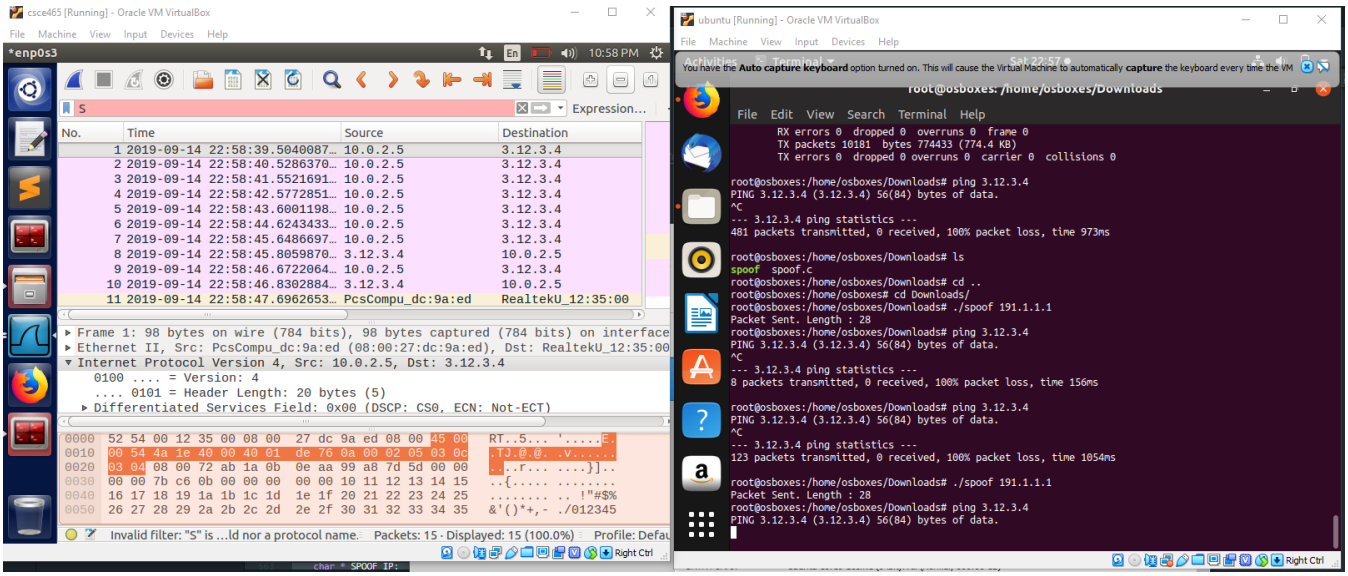

Figure 13: ICMP Echo Reply from VM_A (left window)

The image above shows how the VM_A (left window) spoofs the ping that VM_B (right window) was trying to ping to. The image below (Figure 14) shows how the program sniffs both the ICMP request and the ICMP reply packets.

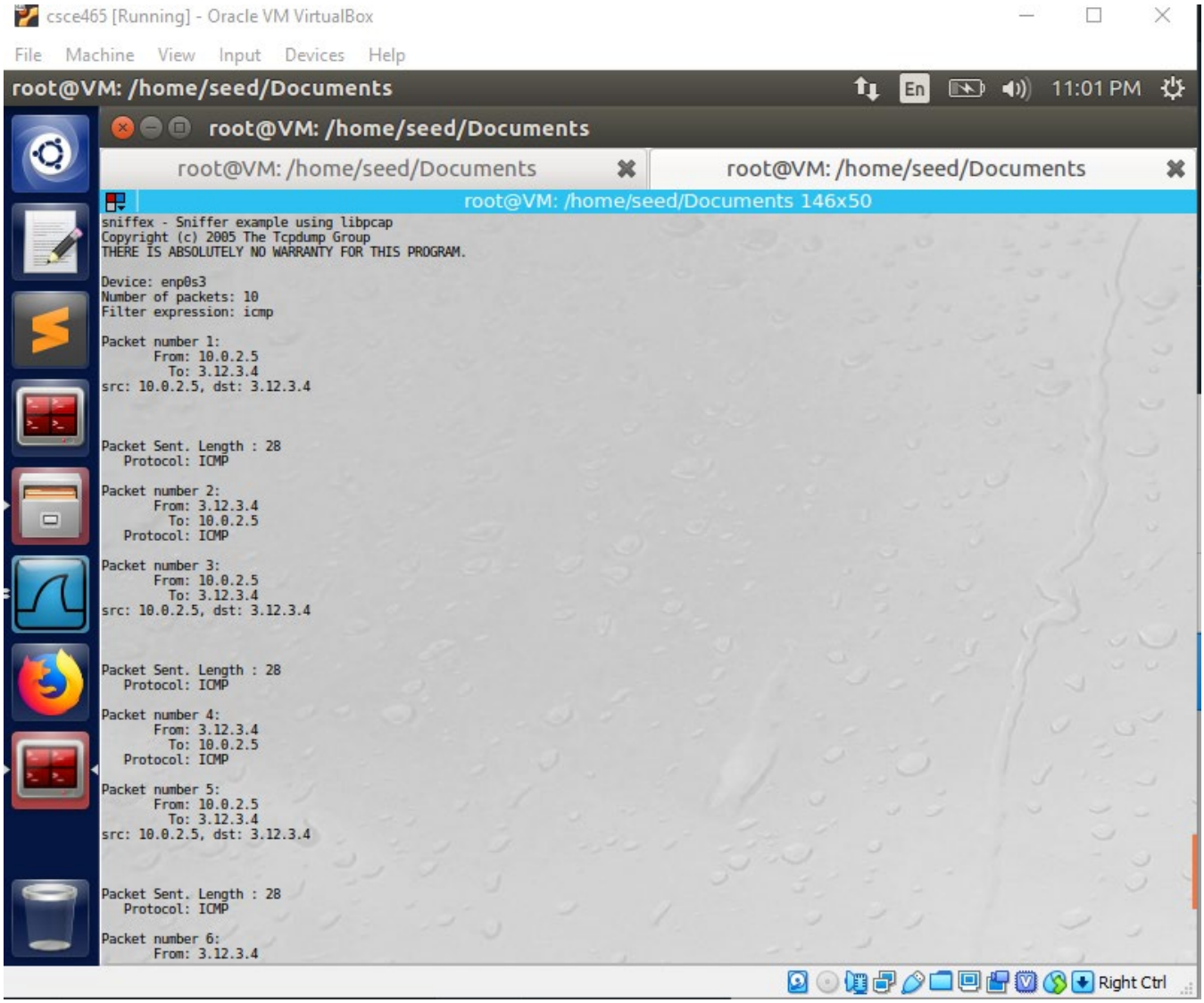

Figure 14: ICMP Echo Request and ICMP Echo Reply packets found through sniffing

In the program, the got_packet callback function will call the function spoof_ip() when it sniffed an ICMP packet where the ICMP source address matches our target address (i.e. targeted attacks), as seen by the code below:

```
if (strcmp(src,dst) != 0 && strcmp(src,TARGET_IP) == 0){
        printf("src: %s, dst: %s\n\n\n\n", src,dst);
        spoof_ip(packet);
    }
```

The following lines of code depict how the program will reply the target user with ICMP reply packet with duplicated information, drawn from struct *packet derived from got_packet():

```
    /* --== FILL UP IP HEADER (LAYER II)==-- */
   iph->ihl=0x5;               /* header length = 5 */
   iph->version=0x4;           /* IPv4 */
   iph->tos= 0x0;              /* typ of service */
   iph->tot_len=SIZEOF_PACKET;
   iph->id=htonl(ip->ip_id);                /* unique field identify each data sent by this host */
   iph->frag_off=0;            /* 0 = off fragment */
   iph->ttl=htonl(ip->ip_ttl);              /* time to live */
   iph->protocol=IPPROTO_ICMP;
   iph->check=0x0;             /* this checksum is only checked after the IP header is done */
   iph->saddr=inet_addr(d_addr);
   iph->daddr=inet_addr(TARGET_IP);

   /* get new checksum after instantiating the ipheader */
   iph->check = chksum((unsigned short *) &pkt, iph->tot_len >> 1);

   /* --== FILL UP IP HEADER (LAYER III)==-- */
   icmph->type = ICMP_ECHOREPLY;               /* ECHO ICMP type */
   icmph->code = 0;                        /* ECHO Request code */
   icmph->checksum = 0;                    /* same as IP header, initiate at 0 first */
   icmph->un.echo.id = htons(icmp_->un.echo.id);       /* can be a random number. htons() is used to transform to big endian */
   icmph->un.echo.sequence = htons(icmp_->un.echo.sequence);
   icmph->checksum = chksum((unsigned short *) &icmph, sizeof(struct icmphdr) >> 1);

   sin.sin_addr.s_addr = iph->daddr;
```

5 Guidelines

5.1 Filling in Data in Raw Packets

When you send out a packet using raw sockets, you basically construct the packet inside a buffer, so when you need to send it out, you simply give the operating system the buffer and the size of the packet. Working directly on the buffer is not easy, so a common way is to typecast the buffer (or part of the buffer) into structures, such as IP header structure, so you can refer to the elements of the buffer using the fields of those structures. You can define the IP, ICMP, TCP, UDP and other header structres in your program. The following example show how you can construct an UDP packet:

```
struct ipheader {
   type  field;
   .....
}

struct udpheader {
   type field;
   ......
}

// This buffer will be used to construct raw packet.
char buffer[1024];

// Typecasting the buffer to the IP header structure
struct ipheader *ip = (struct ipheader *) buffer;

// Typecasting the buffer to the UDP header structure
struct udpheader *udp = (struct udpheader *) (buffer
                            + sizeof(struct ipheader));

// Assign value to the IP and UDP header fields.
```

```
ip->field = ...;
udp->field = ...;
```

5.2 Network/Host Byte Order and the Conversions

You need to pay attention to the network and host byte orders. If you use x86 CPU, your host byte order uses *Little Endian*, while the network byte order uses *Big Endian*. Whatever the data you put into the packet buffer has to use the network byte order; if you do not do that, your packet will not be correct. You actually do not need to worry about what kind of Endian your machine is using, and you actually should not worry about if you want your program to be portable.

What you need to do is to always remember to convert your data to the network byte order when you place the data into the buffer, and convert them to the host byte order when you copy the data from the buffer to a data structure on your computer. If the data is a single byte, you do not need to worry about the order, but if the data is a `short`, `int`, `long`, or a data type that consists of more than one byte, you need to call one of the following functions to convert the data:

```
htonl(): convert unsigned int from host to network byte order.
ntohl(): reverse of htonl().
htons(): convert unsigned short int from host to network byte order.
ntohs(): reverse of htons().
```

You may also need to use `inet_addr()`, `inet_network()`, `inet_ntoa()`, `inet_aton()` to convert IP addresses from the dotted decimal form (a string) to a 32-bit integer of network/host byte order. You can get their manuals from the Internet.

5.3 VirtualBox Network Configuration for Problem 8

The configuration mainly depends on the IP address you ping. Please make sure finish the configuration before the task. The details are attached as follows:

- When you ping a non-existing IP in the same LAN, an ARP request will be sent first. If there is no reply, the ICMP requests will not be sent later. So in order to avoid that ARP request which will stop the ICMP packet, you need to change the ARP cache of the victim VM by adding another MAC to the IP address mapping entry. The command is as follows:

  ```
  % sudo arp -s 192.168.56.1 AA:AA:AA:AA:AA:AA
  ```

  IP `192.168.56.1` is the non-existing IP on the same LAN. `AA:AA:AA:AA:AA:AA` is a spoofed MAC address.

- When you ping a non-existing IP outside the LAN, the Network settings will make a difference.

  If you are using the new "NAT Network", no further changes are needed. If your VMs are connected to two virtual networks (one through the "NAT" adapter, and the other through the "Host-only" adapter), there are two things you need to keep in mind to finish the task:

  – If the victim user is trying to ping a non-existing IP outside the local network, the traffic will go to the NAT adapter. VirtualBox permanently disables the promiscuous mode for NAT by default.

Here we explain how to address these issues:

Redirect the traffic of the victim VM to the gateway of your "Host-only" network by changing the routing table in the victim VM. You can use the command `route` to configure the routing table. Here is an example:

```
% sudo route add -net 1.1.2.0 netmask 255.255.255.0 gw 192.168.56.1
```

The subnet `1.1.2.0/24` is where the non-existing IP sits. The traffic is redirected to `192.168.56.1`, which is on the same LAN of the victim but does not exist. Before sending out the ICMP request, again the victim VM will send an ARP request to find the gateway. Similarly, you should change the ARP cache as stated in the previous example.

Alternatively, you can setup your two VMs using "Bridged Network" in VirtualBox to do the task. "Bridged Network" in VirtualBox allows you to turn on the promiscuous mode. The following procedure on the VirtualBox achieves it:

```
Settings -> Network -> Adapter 1 tab -> "Attach to" section:
Select "Bridged Adapter"

In "Advanced" section -> "Promiscuous Mode":
Select "Allow All"
```

6 Submission

You need to submit your programs and a detailed lab report to describe what you have done and what you have observed; you also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.

References

[1] Programming with pcap. `http://www.tcpdump.org/pcap.htm`

[2] Programming with Libcap - Sniffing the network from our own applicaiton by Luis Martin Garcia. `http://recursos.aldabaknocking.com/libpcapHakin9LuisMartinGarcia.pdf`

[3] Advanced TCP/IP - The Raw Socket Program Examples. `http://www.tenouk.com/Module43a.html`

[4] A brief programming tutorial in C for raw sockets. `http://courses.cse.tamu.edu/guofei/csce465/rawip.txt`