Packet Sniffing and Spoofing Lab Report

Mudit Vats
mpvats@syr.edu
1/19/2019

Table of Contents

Overview

This lab report presents observations and explanations for the tasks described in the Packet Sniffing and Spoofing Lab.

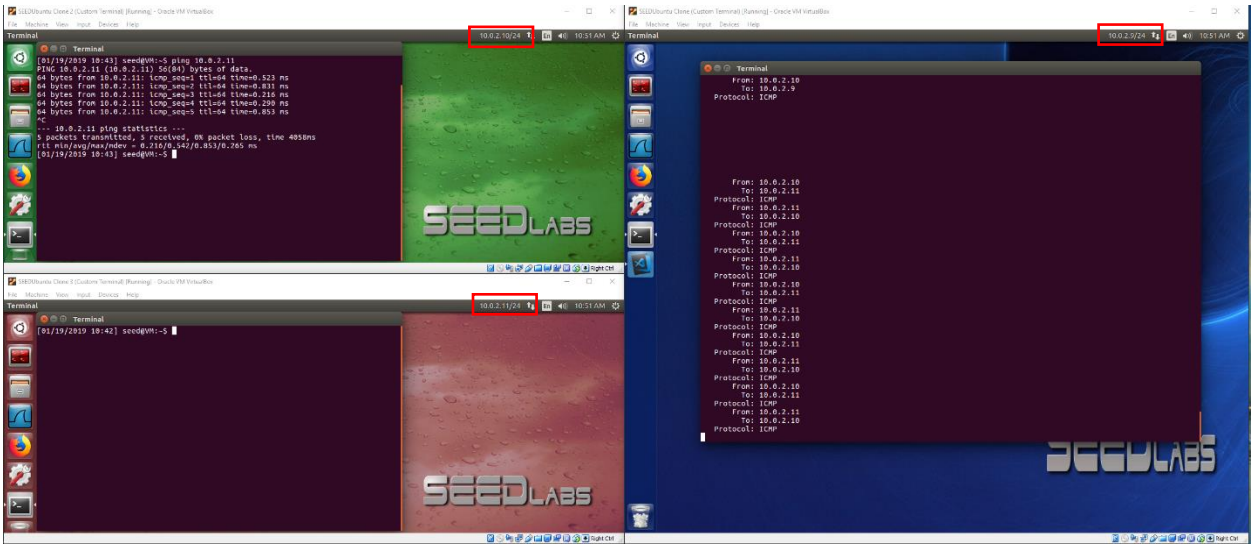
Task 2.1A: Understanding How a Sniffer Works

Students should provide screenshots as evidences to show that their sniffer program can run successfully and produces expected results.

In the screen shot below, you can see three VMs.

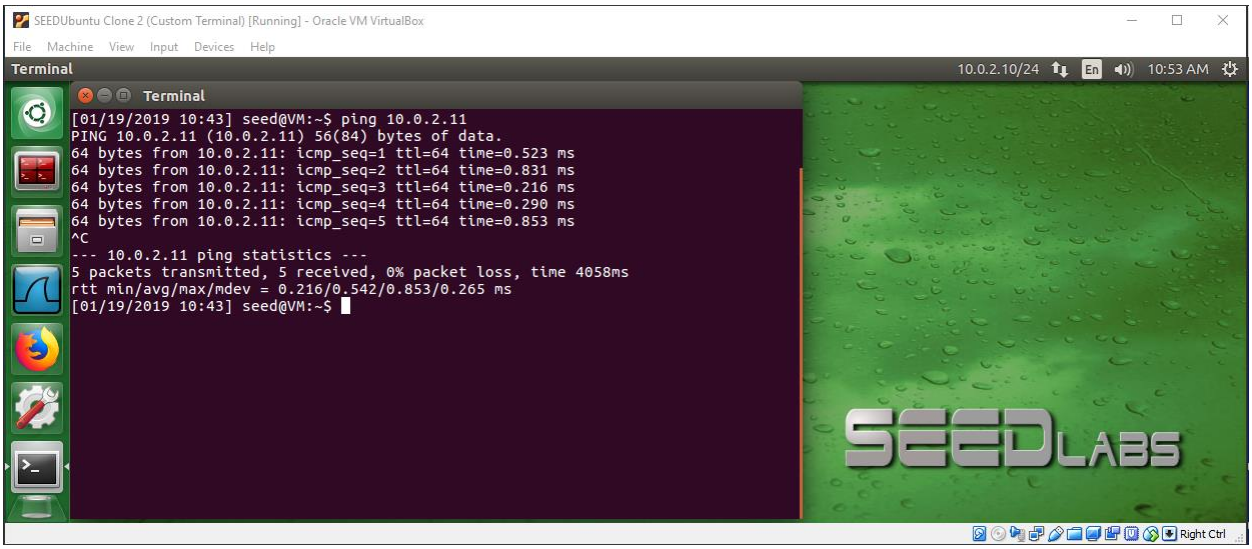
VM Background	IP Address
Blue	10.0.2.9
Green	10.0.2.10
Red	10.0.2.11

In the Blue VM, the sniffer program is running. I then run ping (ICMP) from the Green VM. This ping from Green VM is pinging the Red VM. So, 10.0.2.9 is sniffing ICMP packets on the network. It is displaying the To and From IP addresses of the hosts which are pinging each other: Green 10.0.2.10 and Red 10.0.2.11.

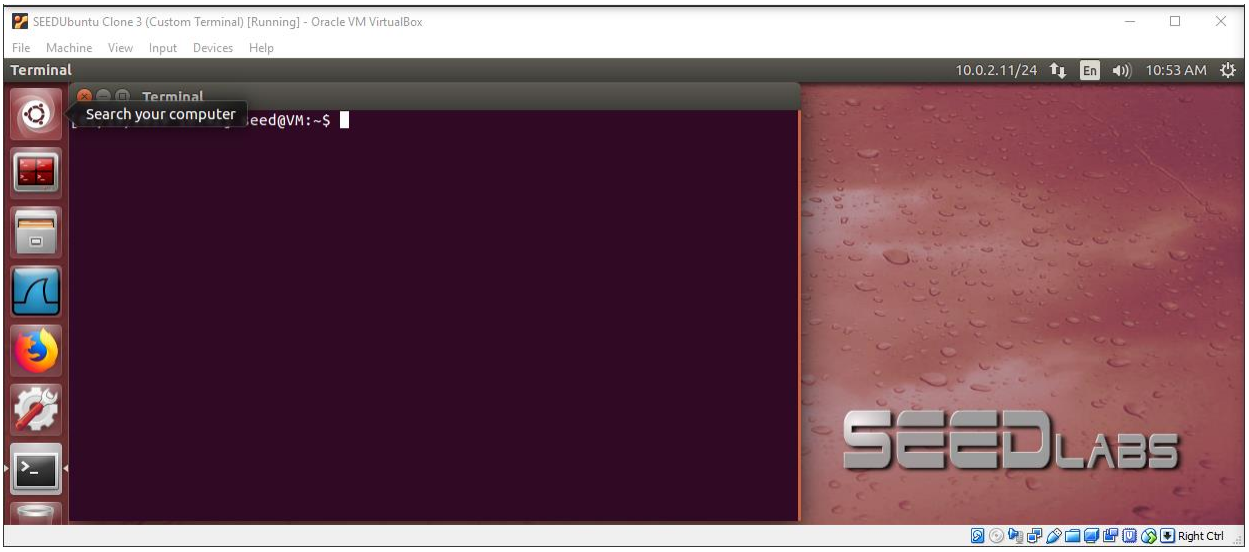


The figure below, shows a zoomed in view of the Green VM. This is the VM which is pinging the Red VM.

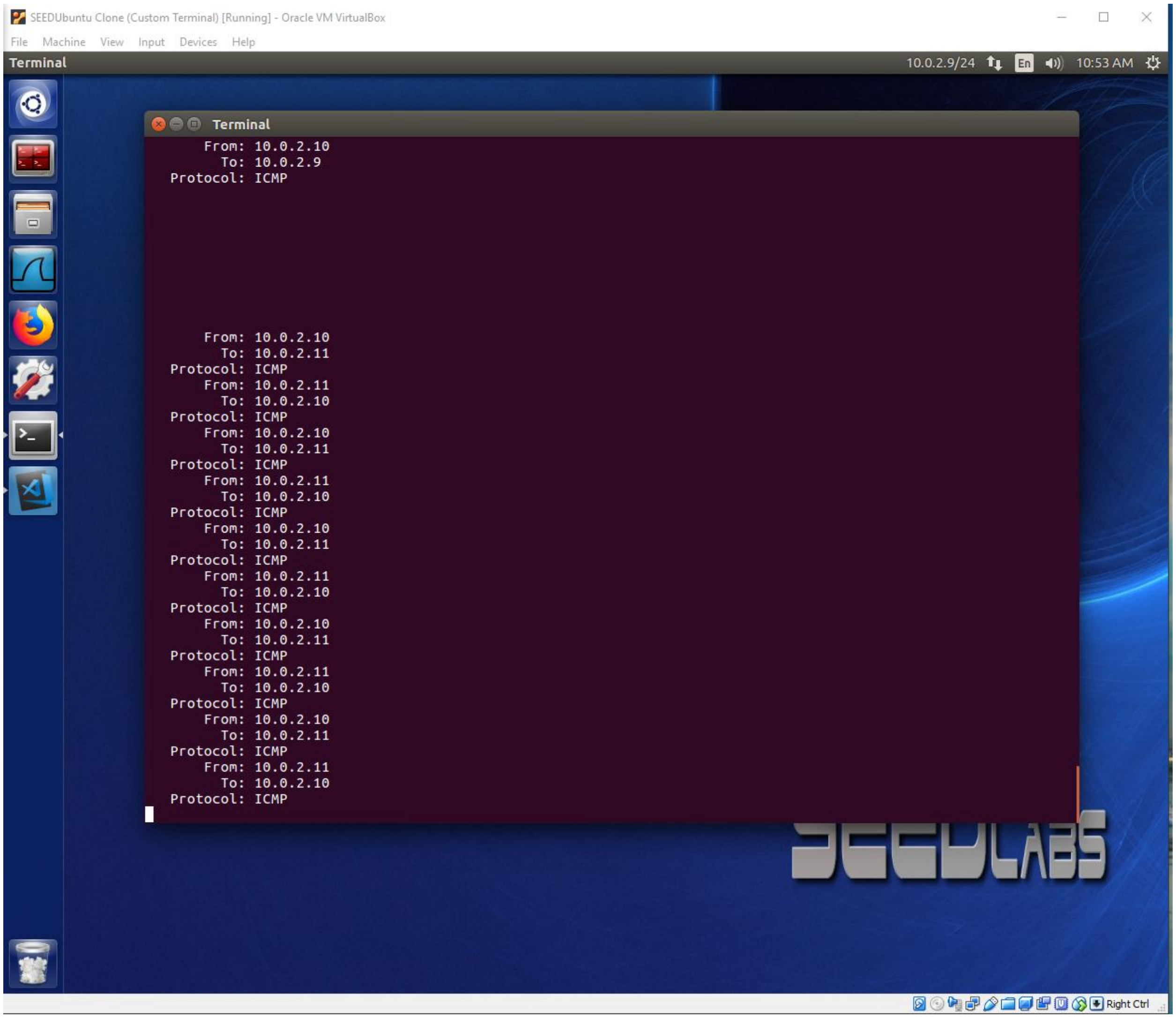
Please note the IP address on the system tray bar (upper right corner). I'm using indicator-ip to accomplish this.



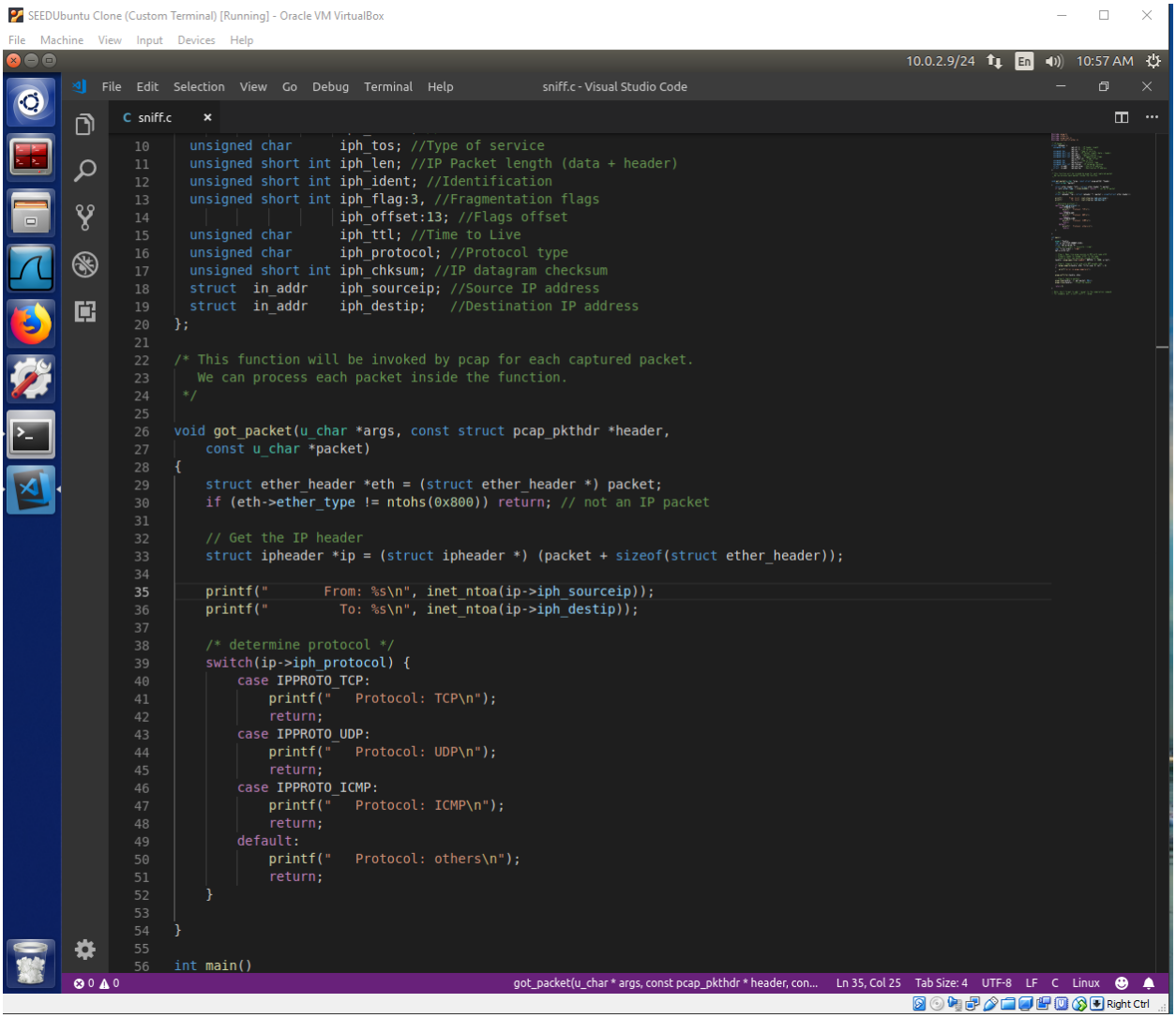
The figure below, shows a zoomed in view of the Red VM. This is the VM which receives the ping request from the Green VM and replies back.



The figure below shows the running ./sniff program.



The figure below show the code for the got_packet callback function which is called every time an ICMP packet is seen on the network.



Observations / Explanations

In this task, we used the PCAP API to create a sniffer program. We created a filter to sniff for ICMP packets. We accomplished this by specifying –

char filter_exp[] = "icmp";

This gets compiled into the pcap structure which is used to determine which packets our got_packet should respond to.

We ran the sniffer program as sudo -

[01/19/19]seed@VM:~/.../lab1$ sudo ./sniff

Since we set the filter to look for ICMP packets, we were able to observe the ping (ICMP) requests and replies from the other VMs (Green and Red) from our sniffing VM (Blue). This proves that the sniffing program is working.

Questions

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.

Open a live PCAP session with the specified NIC. In my case, I changed it to "enp0s3" since that is the ethernet device which I would like the sniffer program to sniff on. Then we specify the filter to use and compile it with pcap_compile. As mentioned, we are filtering for ICMP packets. We then set the filter and start the capture via pcap_loop. One of the parameters of pcap_loop is the callback function (got_packet) that is called when a packet that matches the filter criteria.

Finally, the got_packet gets called the filter-matching packet and we 1) ensure it's an IP packet and then parse the actual packet to print out the source / destination IP addresses.

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?

The program fails to execute with a segmentation fault at the pcap_compile stage. This is where the filter is compiled to the BPF format. It needs root here because we are setting a network filter in the kernel space right above the link level driver. PCAP makes creating BPF filters easy, but, once compiled, it's still a BPF filter in kernel space.

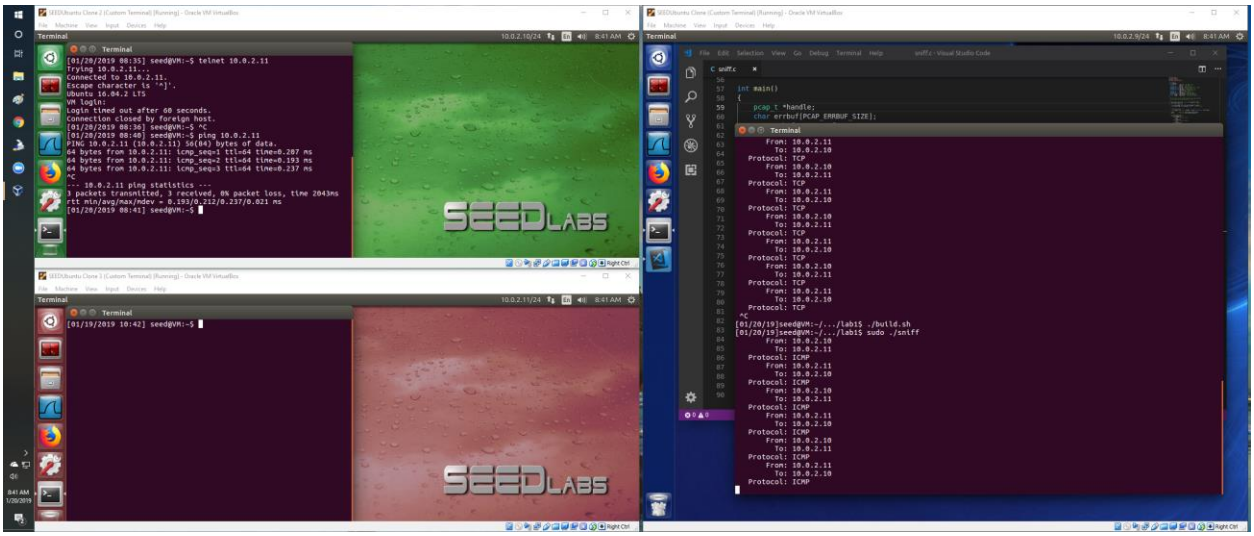
Question 3. Please turn on and turn off the promiscuous mode in your sniffer program. Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this.

I was able set promiscuous mode by toggling the third parameter of pcap_open_live(). When set to 0, promiscuous mode is off. When set to non-zero (eg. "1"), promiscuous mode is on.

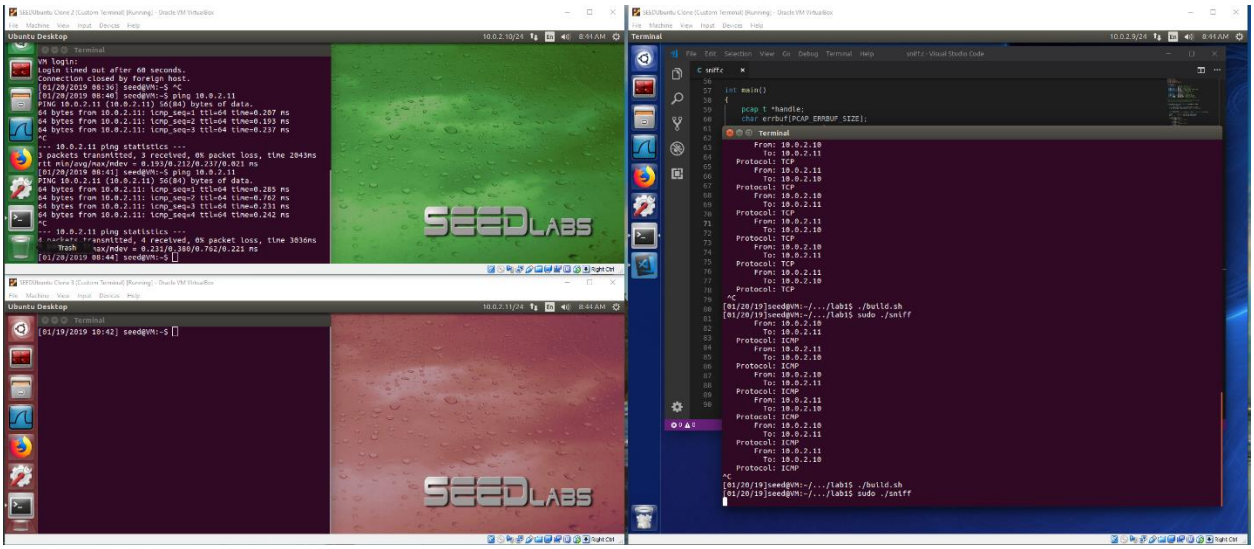
```
// Promiscuous mode off

handle = pcap_open_live("enp0s3", BUFSIZ, 0, 1000, errbuf);

// Promiscuous mode on

handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
```

I was able to prove that it works by using the same demonstration in this section. I had Green ping Red and with promiscuous mode off, I did not see any got_packet output in the sniffer program. When I turned it back on, the behavior was identical to what was documented above.

The figure below shows the ping (ICMP) from Green 10.0.2.10 packets being displayed via the sniffer program on Blue 10.0.2.9. This is with Promiscuous mode on. That is, we are sniffing all packets on the network and have a filter in place to give us the ICMP packets.



In the figure below, we see the recompile of the program with Promiscuous mode off (i.e. set to 0). Now we do not see any packets via the sniffer program.



Task 2.1B: Writing Filters

Please write filter expressions for your sniffer program to capture each of the followings.

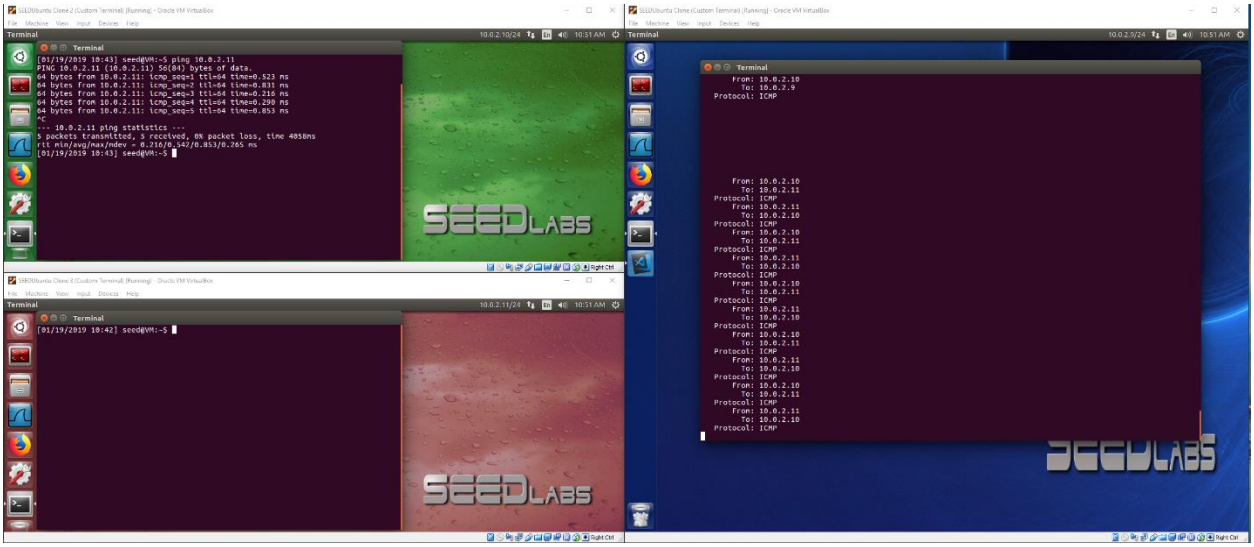
Capture the ICMP packets between two specific hosts

The filter that I used is –

- **char filter_exp[] = "icmp";**

The demonstration of this is in 2.1A since I used this filter to prove that sniffer program works and I also show the screen shots of the sniffer program capturing packets from two different VMs (Green 10.0.2.10 and Red 10.0.2.11). So, VM Blue 10.0.2.9, sees ping (ICMP) request / replies from VM Green 10.0.2.10 and VM Red 10.0.2.11.

The screen shot is below (please zoom in to see it) and/or please see zoomed screen shots of each VM in Task 2.1A.



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

The filter I used is –

- char filter_exp[] = "tcp portrange 10-100";

This filter simply filters all TCP packets in the range of 10 to 100. Either source or destination TCP ports.

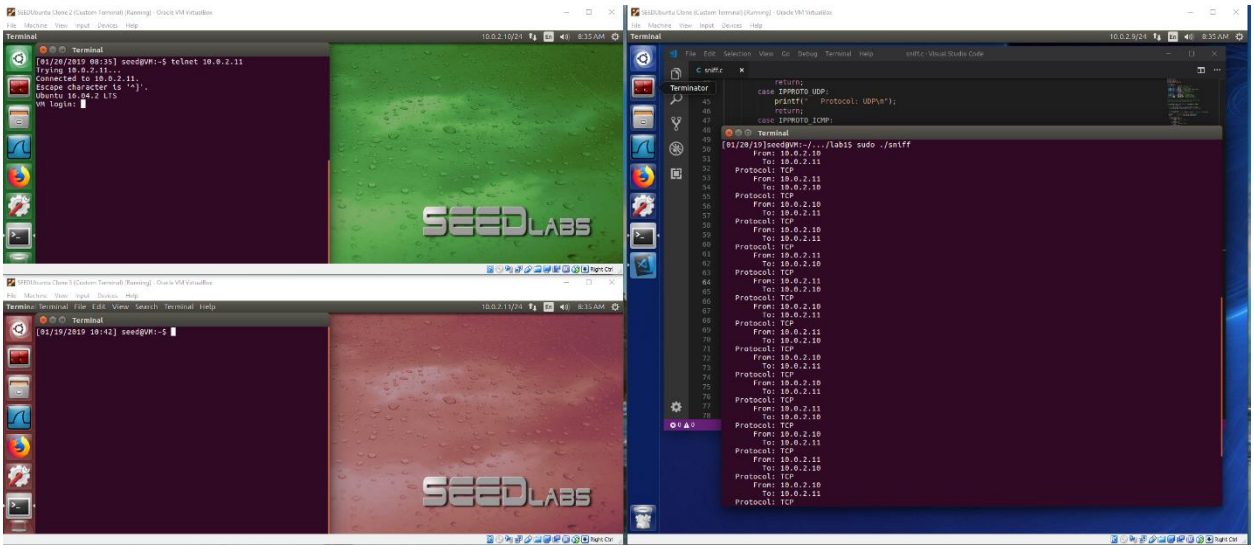
Also, I used tried this filter –

- char filter_exp[] = "(tcp[0:2] > 10 and tcp[0:2] < 100) or (tcp[2:2] > 10 and tcp[2:2] < 100)";

This filter does the same thing as the first filter, except is a more explicit syntax which looks at the TCP packet offset and length to get source port (tcp[0:2) and destination port (tcp[2:2]).

Both worked, but for the demonstration below, I used the first filter since it was a simpler syntax. In the figure below, you can seen the Green 10.0.2.10 VM telnet'ing to Red 10.0.2.11 VM. The Blue 10.0.2.9 VM is running the ./sniff program. We are capturing TCP packets in the range of 10-100. We can see the TCP packets via the sniffer program since we are printing out the protocol type and source/destination IP addresses.

Once again, please zoom in to view screenshot details.



Observations / Explanations

By setting the filters for filter for ICMP and port range 10-100, we were able to see Ping and Telnet traffic, respectively.

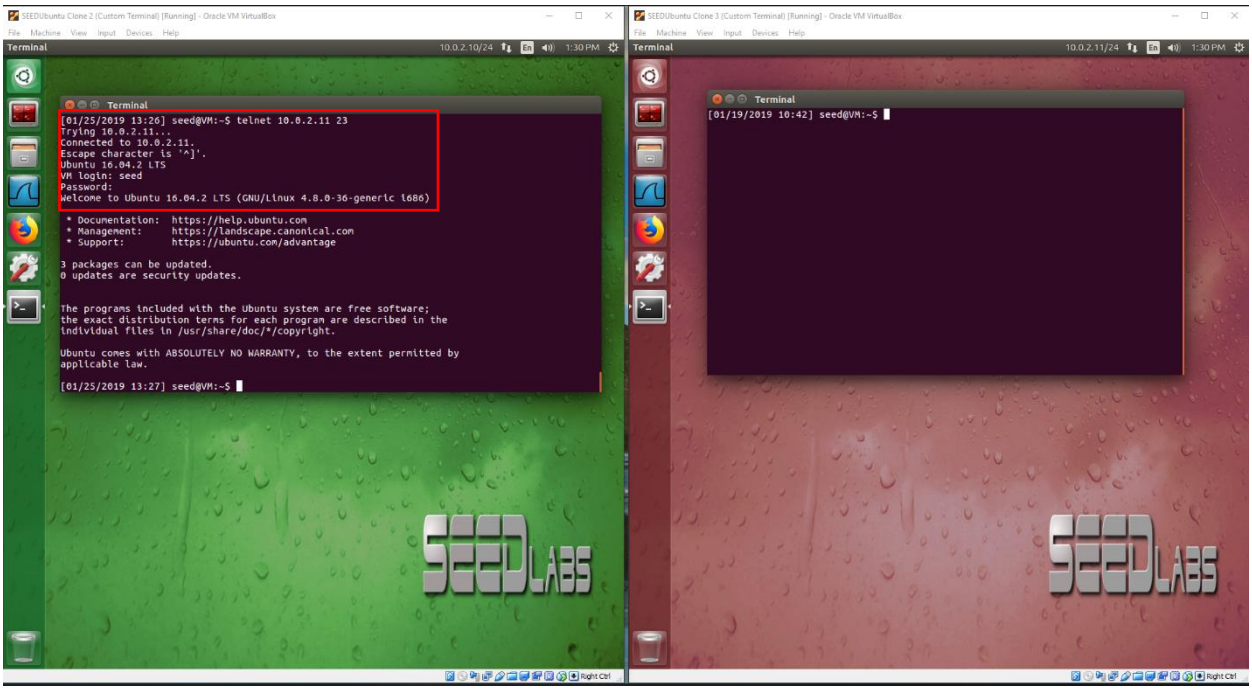
PCAP filters make it really easy to use a simple filtering "language" in a portable manner that can be compiled into native BPF network filters, which can otherwise be much more complicated and OS specific.

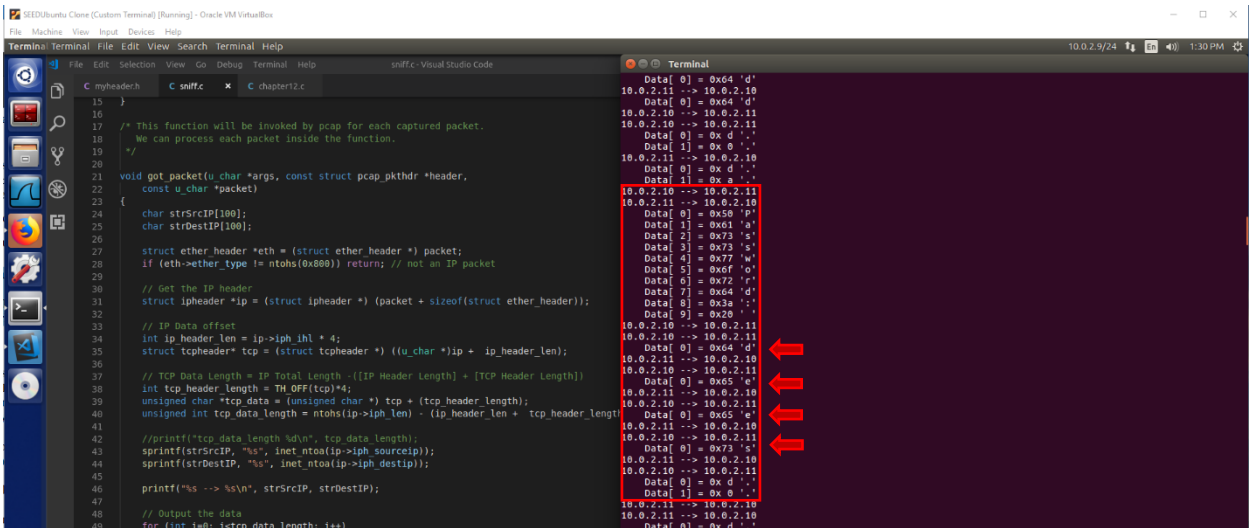
Task 2.1C: Sniffing Passwords

Please show how you can use your sniffer program to capture the password when somebody is using telnet on the network that you are monitoring.

In the figure below, we see our Green VM (10.0.2.10) and Red VM (10.0.2.11). Below that, we see our Blue VM (10.0.2.9). Our scenario is:

1. Run sniffer on the Blue VM (can be seen in the figure).
2. Run Telenet on the Green VM. This Green VM will telnet to the Red VM. The Sniffer program will be listening for "tcp port 23" messages (Telnet) and print out the printable characters from the packet payload.





Observations / Explanations

We created a sniffer program which listens for TCP port 23 / Telnet packets. When the Green VM telnet'ed to the Red VM, we can see the packet data in the console. Of particular interest is that the data for the login/password prompts and text entered into the telnet program on the Green VM are all "in the clear". As such, in the Blue VM we are able to see the username and password as typed when logging in.

More specifically, were able to see "Password:" which is printed by the Telnet program. We were also able to see "dees" that was typed in as the password. Below that text, we can start seeing the "Welcome" message which is shown upon successful login.

We had to update the sniffer code to get the TCP data and TCP data length. We see the code below and in the figure above –

```
void got_packet(u_char *args, const struct pcap_pkthdr *header,
    const u_char *packet)
{
    char strSrcIP[100];
    char strDestIP[100];

    struct ether_header *eth = (struct ether_header *) packet;
    if (eth->ether_type != ntohs(0x800)) return; // not an IP packet

    // Get the IP header
    struct ipheader *ip = (struct ipheader *) (packet + sizeof(struct ether_header));

    // IP Data offset
    int ip_header_len = ip->iph_ihl * 4;
    struct tcpheader* tcp = (struct tcpheader *) ((u_char *)ip +  ip_header_len);

    // TCP Data Length = IP Total Length -([IP Header Length] + [TCP Header Length])
    int tcp_header_length = TH_OFF(tcp)*4;
    unsigned char *tcp_data = (unsigned char *) tcp + (tcp_header_length);
    unsigned int tcp_data_length = ntohs(ip->iph_len) - (ip_header_len +  tcp_header_length);

    //printf("tcp_data_length %d\n", tcp_data_length);
    sprintf(strSrcIP, "%s", inet_ntoa(ip->iph_sourceip));
    sprintf(strDestIP, "%s", inet_ntoa(ip->iph_destip));

    printf("%s --> %s\n", strSrcIP, strDestIP);

    // Output the data
    for (int i=0; i<tcp_data_length; i++)
    {
        printf("    Data[%2d] = 0x%2x '%c'\n", i, (char) *(tcp_data + i) & 0xff, getPrintableAscii((char) *(tcp_data + i)) &
0xff);
    }
}
```

Basically, we get the TCP data by advancing the IP pointer past the IP header and TCP header. We get the TCP data length by getting the total length of the IP header and then subtracting the headers (IP and TCP). This leaves the TCP payload which is the data we printed out.

The main observation is that Telnet is NOT secure since its text is in the clear. Also, sniffing a program via PCAP did not take much effort (just create a PCAP session with the TCP port filter and capture the packet). While the sniffer can only be run as root, root is only a concern on the running platform so sniffing network traffic on the network is a very easy thing to do. So… be careful (use VPN on public networks!) 😊

Task 2.2A: Write a Spoofing Program

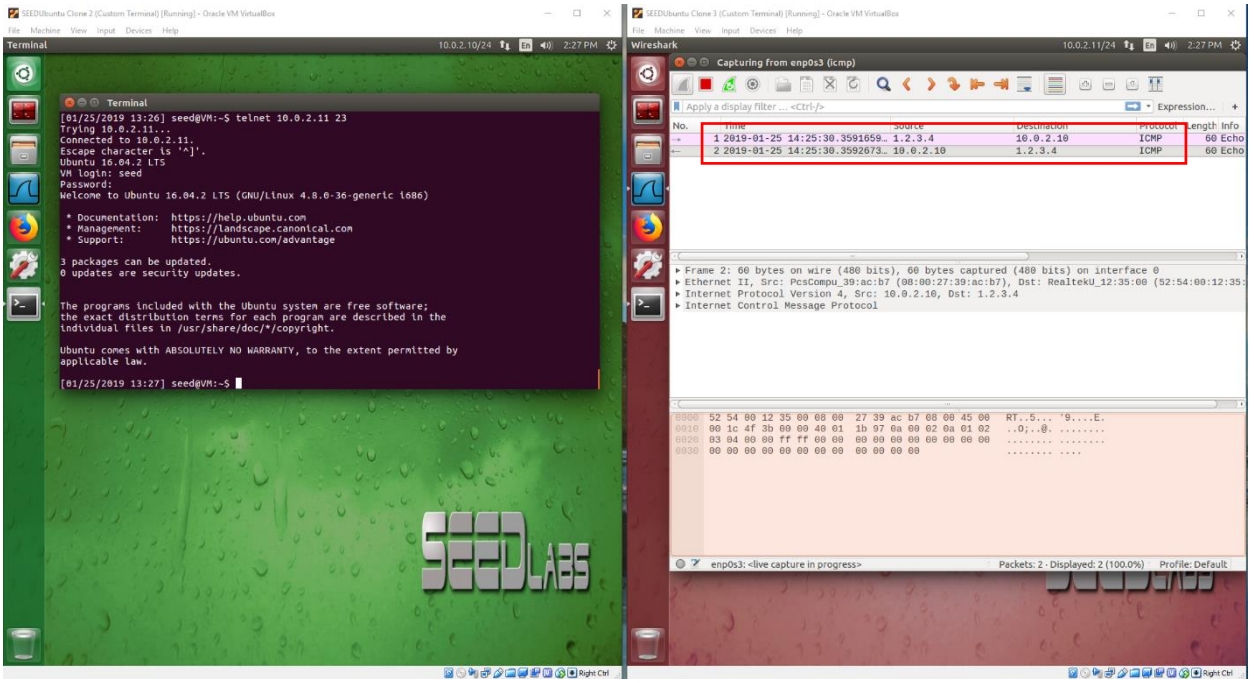
Per Professors: "you can combine them and do one task." This was in regards to Task 2.2A and 2.2B. Please see 2.2B for demonstrating a Spoofing program and demonstrating spoofing and ICMP Echo Request.

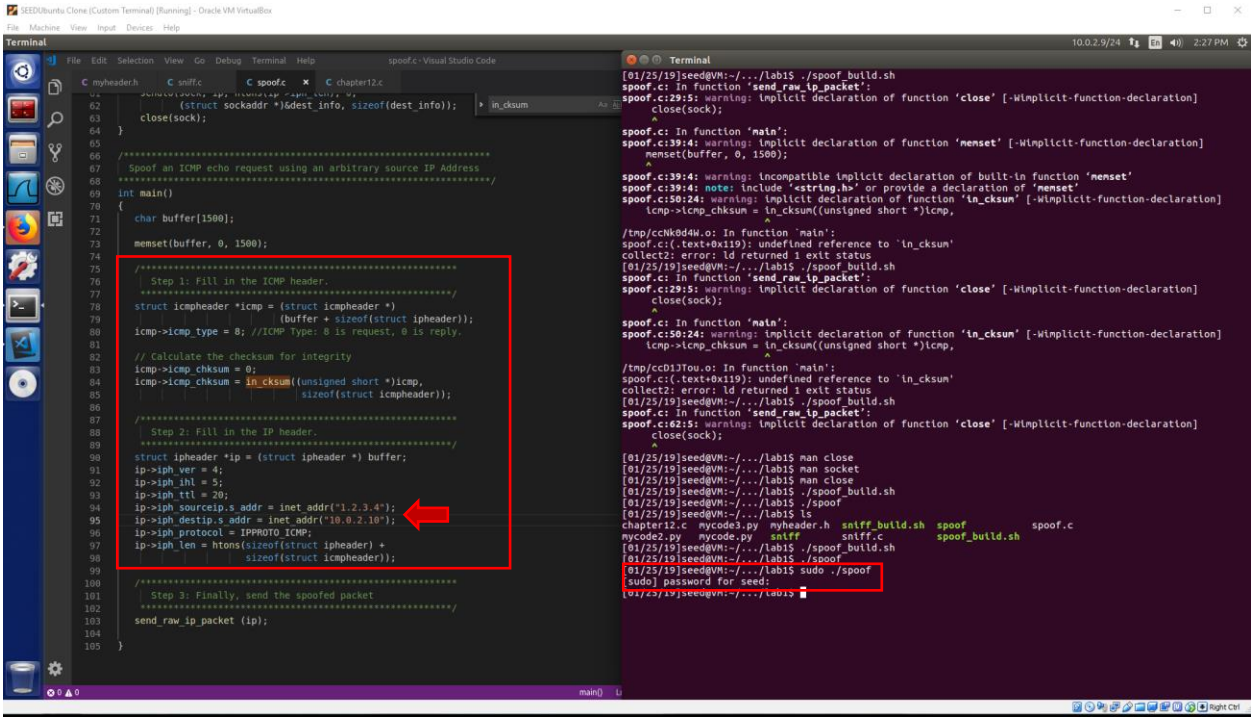
Task 2.2B: Spoof and ICMP Echo Request

In the two figures below, we see our three VMs – Green 10.0.2.10, Red 10.0.2.11, and Blue 10.0.2.9.

The Blue VM sends out a spoofed ICMP Echo Request to Green 10.0.2.10.

The Red VM is running Wireshark which is looking for all ICMP messages on the NAT network.





Observations / Explanations

We send out the spoofed packet from the Blue VM. We can see in the code in the screen capture of the Blue VM. Basically, we are created an ICMP Request packet (type 8) which is the payload of the IP packet. In the IP packet, we specify an IP address of a non-existent machine on the network: 1.2.3.4. We specify the destination IP to an

existing VM which is the Green VM 10.0.2.10. We need to send this to an existing VM / IP address since we need it to respond with a ICMP Echo Reply message. Additionally, we send the packet via RAW socket and sudo, since we need absolute control of the packet generation and being able to specify the details of the IP header.

We can see from the WireShark output that the ICMP Echo Request packet was sent from 1.2.3.4 to 10.0.2.10. Right below that, we see that an ICMP Echo Response was sent from 10.0.2.10 to 1.2.3.4.

By, observing the ICMP Echo Reply via WireShark, we can clearly see that our spoofed packet was valid such that the target Green VM 10.0.2.10, replied to the request; i.e. we successfully spoofed an ICMP Echo Request packet!

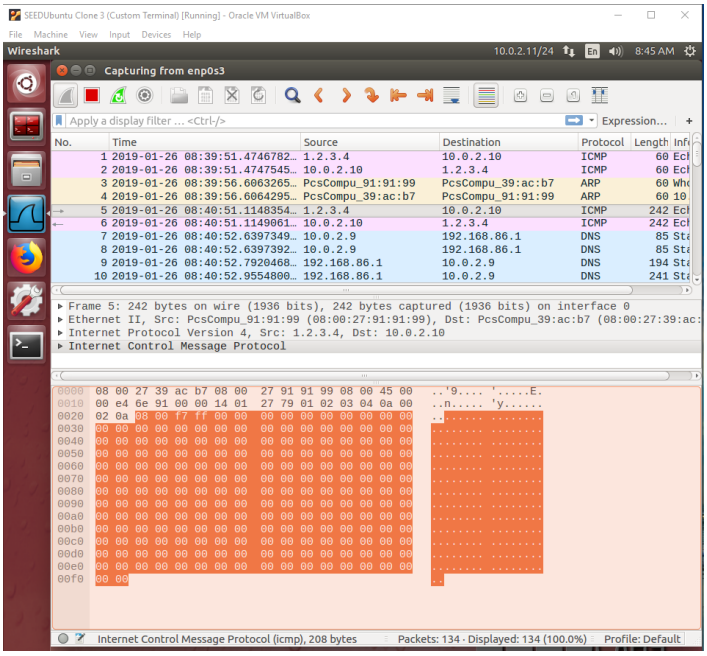
Questions

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

Yes. As an experiment, I set the IP total length to a larger value:

```
ip->iph_len = htons(sizeof(struct ipheader) +
                    sizeof(struct icmpheader) + 200);
```

When I looked at the packet via WireShark, this is what I saw:



The 200 additional bytes was increased the payload size of the ICMP Request. This didn't "seem" too cause any issues since the ICMP Reply still occurred. But, I think this may / may not work depending on how well the OS on the receiving end checks the integrity of the packet. For example, it may consider an ICMP with arbitrarily large payload suspicious and discard it.

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

No, it is being filled in by the OS when sending the packet.

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?

When I ran as non-root, the socket creation failed. So, "int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);" failed with "-1". The OS prevents a non-root user to create a RAW socket.

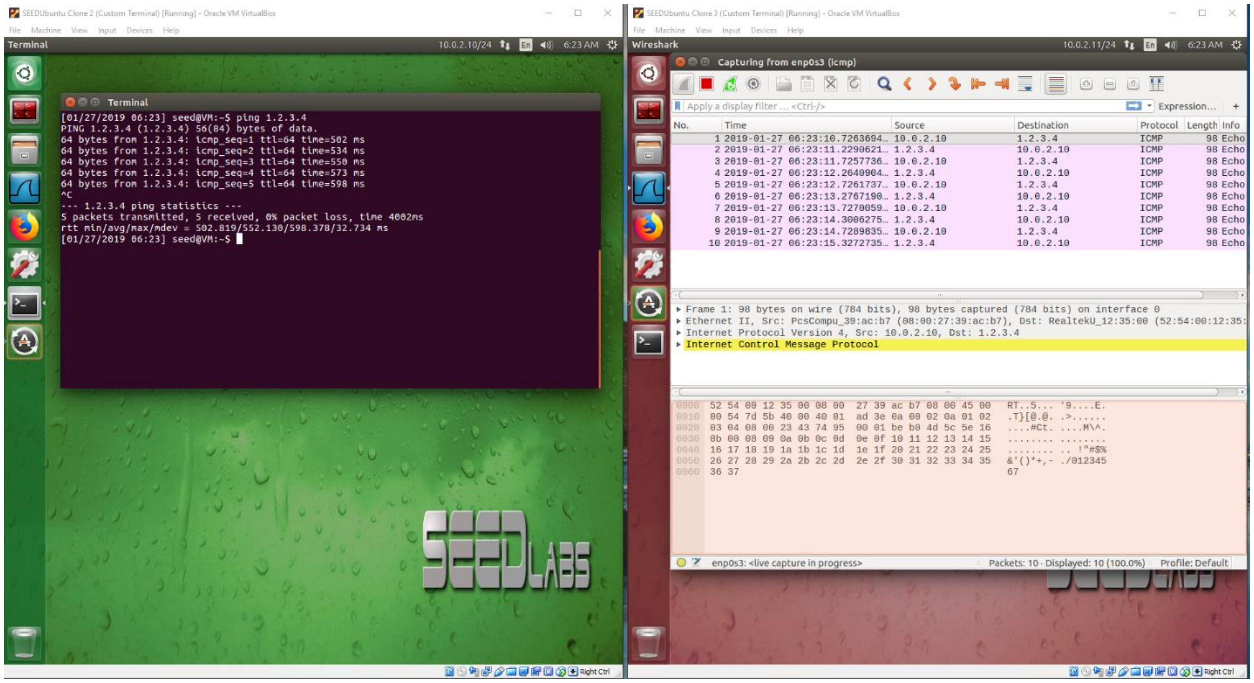
You need root privilege to run a program that uses RAW sockets because with RAW sockets you can create spoofed packets, like we just did, on the network which can be used for malicious attacks and or impersonation on the network.

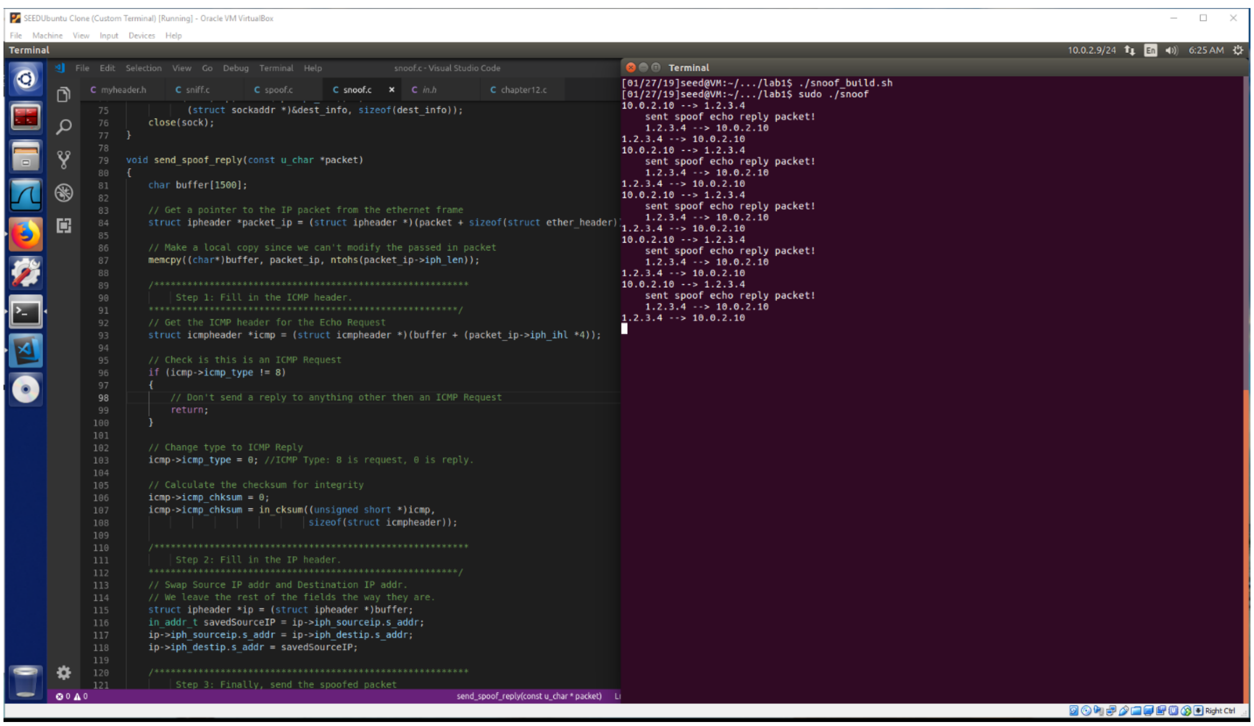
Task 2.3: Sniff and then Spoof

In this task, you will combine the sniffing and spoofing techniques to implement the following sniff-and then-spoof program.

The screen shots below show the following scenario:

1. Green VM 10.0.2.10 sends an ICMP Echo Request to 1.2.3.4. This is done via the "ping 1.2.3.4" command.
2. Blue VM 10.0.2.9 is sniffing for ICMP traffic and when it sees an ICMP Echo Request, it responds with an ICMP Echo Reply. Please note that the Blue VM will respond to *ALL ICMP Echo Requests* and spoof a reply on behalf of ICMP Echo Request's destination IP address.
3. Red VM 10.0.2.11 shows the WireShark sniffer which is sniffing for ICMP packets on the NAT network.





Observations / Explanations

In this lab, we combined Sniffing and Spoofing. We sniff for ICMP packets and then spoof an ICMP Echo Reply.

In our code, we set the filter to “icmp”. This is done by:

```
char filter_exp[] = "icmp";
```

When PCAP gets and ICMP packet, it will call our got_packet() function shown below:

```
void got_packet(u_char *args, const struct pcap_pkthdr *header,
                const u_char *packet)
{
    char strSrcIP[100];
    char strDestIP[100];

    struct ether_header *eth = (struct ether_header *)packet;
    if (eth->ether_type != ntohs(0x800))
    {
        return; // not an IP packet
    }

    // Get the IP header
    struct ipheader *ip = (struct ipheader *)(packet + sizeof(struct ether_header));

    // IP Data offset
    int ip_header_len = ip->iph_ihl * 4;
    struct tcpheader *tcp = (struct tcpheader *)((u_char *)ip + ip_header_len);

    // TCP Data Length = IP Total Length -([IP Header Length] + [TCP Header Length])
    int tcp_header_length = TH_OFF(tcp) * 4;
    unsigned char *tcp_data = (unsigned char *)tcp + (tcp_header_length);
    unsigned int tcp_data_length = ntohs(ip->iph_len) - (ip_header_len + tcp_header_length);

    sprintf(strSrcIP, "%s", inet_ntoa(ip->iph_sourceip));
    sprintf(strDestIP, "%s", inet_ntoa(ip->iph_destip));

    printf("%s --> %s\n", strSrcIP, strDestIP);
    send_spoof_reply(packet);
}
```

The main thing we do in this function is:

1. Check to make sure we're getting an IP packet.
2. Print out the Source IP and Destination IP addresses.
    a. The TCP Data Length code was from a previous part of this lab. We don't use it here since we do not need to output the data.
3. Call a function, send_spoof_reply(packet) to send a spoofed packet.

send_spoof_reply(packet) does the work of modifying the ICMP Echo Request and forging an ICMP Echo Reply. The code is below:

```
void send_spoof_reply(const u_char *packet)
{
    char buffer[1500];

    // Get a pointer to the IP packet from the ethernet frame
    struct ipheader *packet_ip = (struct ipheader *)(packet + sizeof(struct ether_header));

    // Make a local copy since we can't modify the passed in packet
    memcpy((char*)buffer, packet_ip, ntohs(packet_ip->iph_len));

    /*********************************************************
         Step 1: Fill in the ICMP header.
    *********************************************************/
    // Get the ICMP header for the Echo Request
    struct icmpheader *icmp = (struct icmpheader *)(buffer + (packet_ip->iph_ihl *4));

    // Check is this is an ICMP Request
    if (icmp->icmp_type != 8)
    {
        // Don't send a reply to anything other then an ICMP Request
        return;
    }

    // Change type to ICMP Reply
    icmp->icmp_type = 0; //ICMP Type: 8 is request, 0 is reply.

    // Calculate the checksum for integrity
    icmp->icmp_chksum = 0;
    icmp->icmp_chksum = in_cksum((unsigned short *)icmp,
                                 sizeof(struct icmpheader));

    /*********************************************************
```

```
        Step 2: Fill in the IP header.
    ******************************************************/
    // Swap Source IP addr and Destination IP addr.
    // We leave the rest of the fields the way they are.
    struct ipheader *ip = (struct ipheader *)buffer;
    in_addr_t savedSourceIP = ip->iph_sourceip.s_addr;
    ip->iph_sourceip.s_addr = ip->iph_destip.s_addr;
    ip->iph_destip.s_addr = savedSourceIP;

    /******************************************************
        Step 3: Finally, send the spoofed packet
    ******************************************************/
    send_raw_ip_packet(ip);

    char strSrcIP[100];
    char strDestIP[100];
    sprintf(strSrcIP, "%s", inet_ntoa(ip->iph_sourceip));
    sprintf(strDestIP, "%s", inet_ntoa(ip->iph_destip));

    printf("    sent spoof echo reply packet!\n");
    printf("    %s --> %s\n", strSrcIP, strDestIP);
}
```

While the code is commented at each step, the main tasks which this function executes are:

1. Makes a copy of the passed in packet. Please note that the passed in packet contains a "const" modifier so we cannot modify it. Therefore, we make a copy of it for our manipulation.
2. Check the ICMP Type to see if this is an ICMP Echo Request. We only send ICMP Echo Replies to ICMP Echo Requests, so we make this check.
3. Modify the copied packet so that the ICMP Type is 0, which is ICMP Echo Reply.
    a. Update the checksum for the ICMP header.
4. For the IP header, we simply swap the Source IP and Destination IP addresses. We leave the rest of the fields the same.
5. Finally, we send the RAW IP Packet.

By sniffing for ICMP Requests and spoofing ICMP Replies, we were able to successfully reply to an ICMP Echo Request to a non-existent (1.2.3.4) IP address.

Full code listing below.

Full Code Listings

Sniff.c

This code shows 2.1C Sniffing Password task. It also encompasses the code for the prior 2.1A Writing a Sniffer and 2.1B Writing Filters. I started Sniff.c for 2.1A and kept adding to it for the 2.1B and 2.1C.

```
#include "myheader.h"
#include <pcap.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>


char getPrintableAscii(char c)
{
    if (c<0x20) return '.';
    if (c>0x7e) return '.';

    return c;

}
```

```
/* 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)
{
    char strSrcIP[100];
    char strDestIP[100];

    struct ether_header *eth = (struct ether_header *) packet;
    if (eth->ether_type != ntohs(0x800)) return; // not an IP packet

    // Get the IP header
    struct ipheader *ip = (struct ipheader *) (packet + sizeof(struct ether_header));

    // IP Data offset
    int ip_header_len = ip->iph_ihl * 4;
    struct tcpheader* tcp = (struct tcpheader *) ((u_char *)ip +  ip_header_len);

    // TCP Data Length = IP Total Length -([IP Header Length] + [TCP Header Length])
    int tcp_header_length = TH_OFF(tcp)*4;
    unsigned char *tcp_data = (unsigned char *) tcp + (tcp_header_length);
    unsigned int tcp_data_length = ntohs(ip->iph_len) - (ip_header_len +  tcp_header_length);

    //printf("tcp_data_length %d\n", tcp_data_length);
    sprintf(strSrcIP, "%s", inet_ntoa(ip->iph_sourceip));
    sprintf(strDestIP, "%s", inet_ntoa(ip->iph_destip));

    printf("%s --> %s\n", strSrcIP, strDestIP);

    // Output the data
    for (int i=0; i<tcp_data_length; i++)
    {
        printf("    Data[%2d] = 0x%2x '%c'\n", i, (char) *(tcp_data + i) & 0xff, getPrintableAscii((char)
*(tcp_data + i)) & 0xff);
    }
}

int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    //char filter_exp[] = "ip proto \\icmp";
    //char filter_exp[] = "icmp";
    //char filter_exp[] = "(tcp[0:2] > 10 and tcp[0:2] < 100) or (tcp[2:2] > 10 and tcp[2:2] < 100)";
    //char filter_exp[] = "tcp portrange 10-100";
    char filter_exp[] = "tcp port 23";
    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
    if (pcap_compile(handle, &fp, filter_exp, 0, net) != 0)
    {
        printf("error in pcap_compile\n");
    }

    pcap_setfilter(handle, &fp);

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

    return 0;
}

// Note: don't forget to add "-lpcap" to the compilation command.
// For example: gcc -o sniff sniff.c -lpcap
```

Spoof.c

The listing below shows the code for 2.2B Spoof and ICMP Echo Request.

```
#include "myheader.h"
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>

/*****************************************
 * Listing 12.9: Calculating Internet Checksum
 *****************************************/

unsigned short in_cksum (unsigned short *buf, int length)
{
   unsigned short *w = buf;
   int nleft = length;
   int sum = 0;
   unsigned short temp=0;

   /*
    * The algorithm uses a 32 bit accumulator (sum), adds
    * sequential 16 bit words to it, and at the end, folds back all
    * the carry bits from the top 16 bits into the lower 16 bits.
    */
   while (nleft > 1)  {
       sum += *w++;
       nleft -= 2;
   }

   /* treat the odd byte at the end, if any */
   if (nleft == 1) {
        *(u_char *)(&temp) = *(u_char *)w ;
        sum += temp;
   }

   /* add back carry outs from top 16 bits to low 16 bits */
   sum = (sum >> 16) + (sum & 0xffff);  // add hi 16 to low 16
   sum += (sum >> 16);                  // add carry
   return (unsigned short)(~sum);
}

/*************************************************************
  Given an IP packet, send it out using a raw socket.
*************************************************************/
void send_raw_ip_packet(struct ipheader* ip)
{
    struct sockaddr_in dest_info;
    int enable = 1;

    // Step 1: Create a raw network socket.
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
    printf("sock %d\n", sock);

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

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

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

/*****************************************************************
  Spoof an ICMP echo request using an arbitrary source IP Address
*****************************************************************/
int main()
{
   char buffer[1500];

   memset(buffer, 0, 1500);

   /*********************************************************
      Step 1: Fill in the ICMP header.
```

```
    ****************************************************/
   struct icmpheader *icmp = (struct icmpheader *)
                             (buffer + sizeof(struct ipheader));
   icmp->icmp_type = 8; //ICMP Type: 8 is request, 0 is reply.

   // Calculate the checksum for integrity
   icmp->icmp_chksum = 0;
   icmp->icmp_chksum = in_cksum((unsigned short *)icmp,
                                sizeof(struct icmpheader));

   /*****************************************************
      Step 2: Fill in the IP header.
    *****************************************************/
   struct ipheader *ip = (struct ipheader *) buffer;
   ip->iph_ver = 4;
   ip->iph_ihl = 5;
   ip->iph_ttl = 20;
   ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
   ip->iph_destip.s_addr = inet_addr("10.0.2.10");
   ip->iph_protocol = IPPROTO_ICMP;
   ip->iph_len = htons(sizeof(struct ipheader) +
                       sizeof(struct icmpheader));

   /*****************************************************
      Step 3: Finally, send the spoofed packet
    *****************************************************/
   send_raw_ip_packet (ip);

}
```

Snoof.c

The code below is the listing for 2.3 Sniff and then Spoof.

```
#include "myheader.h"
#include <pcap.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <netinet/if_ether.h>

char getPrintableAscii(char c)
{
    if (c < 0x20)
        return '.';
    if (c > 0x7e)
        return '.';

    return c;
}

/*********************************************
 * Listing 12.9: Calculating Internet Checksum
 *********************************************/

unsigned short in_cksum(unsigned short *buf, int length)
{
    unsigned short *w = buf;
    int nleft = length;
    int sum = 0;
    unsigned short temp = 0;

    /*
    * The algorithm uses a 32 bit accumulator (sum), adds
    * sequential 16 bit words to it, and at the end, folds back all
    * the carry bits from the top 16 bits into the lower 16 bits.
    */
    while (nleft > 1)
    {
        sum += *w++;
        nleft -= 2;
    }

    /* treat the odd byte at the end, if any */
    if (nleft == 1)
    {
        *(u_char *)(&temp) = *(u_char *)w;
        sum += temp;
```

```
    }

    /* add back carry outs from top 16 bits to low 16 bits */
    sum = (sum >> 16) + (sum & 0xffff); // add hi 16 to low 16
    sum += (sum >> 16);                 // add carry
    return (unsigned short)(~sum);
}

/*************************************************************
  Given an IP packet, send it out using a raw socket.
*************************************************************/
void send_raw_ip_packet(struct ipheader *ip)
{
    struct sockaddr_in dest_info;
    int enable = 1;

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

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

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

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

void send_spoof_reply(const u_char *packet)
{
    char buffer[1500];

    // Get a pointer to the IP packet from the ethernet frame
    struct ipheader *packet_ip = (struct ipheader *)(packet + sizeof(struct ether_header));

    // Make a local copy since we can't modify the passed in packet
    memcpy((char*)buffer, packet_ip, ntohs(packet_ip->iph_len));

    /*************************************************************
         Step 1: Fill in the ICMP header.
    *************************************************************/
    // Get the ICMP header for the Echo Request
    struct icmpheader *icmp = (struct icmpheader *)(buffer + (packet_ip->iph_ihl *4));

    // Check is this is an ICMP Request
    if (icmp->icmp_type != 8)
    {
        // Don't send a reply to anything other then an ICMP Request
        return;
    }

    // Change type to ICMP Reply
    icmp->icmp_type = 0; //ICMP Type: 8 is request, 0 is reply.

    // Calculate the checksum for integrity
    icmp->icmp_chksum = 0;
    icmp->icmp_chksum = in_cksum((unsigned short *)icmp,
                                 sizeof(struct icmpheader));

    /*************************************************************
         Step 2: Fill in the IP header.
    *************************************************************/
    // Swap Source IP addr and Destination IP addr.
    // We leave the rest of the fields the way they are.
    struct ipheader *ip = (struct ipheader *)buffer;
    in_addr_t savedSourceIP = ip->iph_sourceip.s_addr;
    ip->iph_sourceip.s_addr = ip->iph_destip.s_addr;
    ip->iph_destip.s_addr = savedSourceIP;

    /*************************************************************
         Step 3: Finally, send the spoofed packet
    *************************************************************/
    send_raw_ip_packet(ip);

    char strSrcIP[100];
    char strDestIP[100];
```

```
    sprintf(strSrcIP, "%s", inet_ntoa(ip->iph_sourceip));
    sprintf(strDestIP, "%s", inet_ntoa(ip->iph_destip));

    printf("    sent spoof echo reply packet!\n");
    printf("    %s --> %s\n", strSrcIP, strDestIP);
}

/* 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)
{
    char strSrcIP[100];
    char strDestIP[100];

    struct ether_header *eth = (struct ether_header *)packet;
    if (eth->ether_type != ntohs(0x800))
    {
        return; // not an IP packet
    }

    // Get the IP header
    struct ipheader *ip = (struct ipheader *)(packet + sizeof(struct ether_header));

    // IP Data offset
    int ip_header_len = ip->iph_ihl * 4;
    struct tcpheader *tcp = (struct tcpheader *)((u_char *)ip + ip_header_len);

    // TCP Data Length = IP Total Length -([IP Header Length] + [TCP Header Length])
    int tcp_header_length = TH_OFF(tcp) * 4;
    unsigned char *tcp_data = (unsigned char *)tcp + (tcp_header_length);
    unsigned int tcp_data_length = ntohs(ip->iph_len) - (ip_header_len + tcp_header_length);

    sprintf(strSrcIP, "%s", inet_ntoa(ip->iph_sourceip));
    sprintf(strDestIP, "%s", inet_ntoa(ip->iph_destip));

    printf("%s --> %s\n", strSrcIP, strDestIP);
    send_spoof_reply(packet);
}

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 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
    if (pcap_compile(handle, &fp, filter_exp, 0, net) != 0)
    {
        printf("error in pcap_compile\n");
    }

    pcap_setfilter(handle, &fp);

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

    return 0;
}

// Note: don't forget to add "-lpcap" to the compilation command.
// For example: gcc -o sniff sniff.c -lpcap
```