HW 4: Packet Sniffing and Spoofing

Purpose

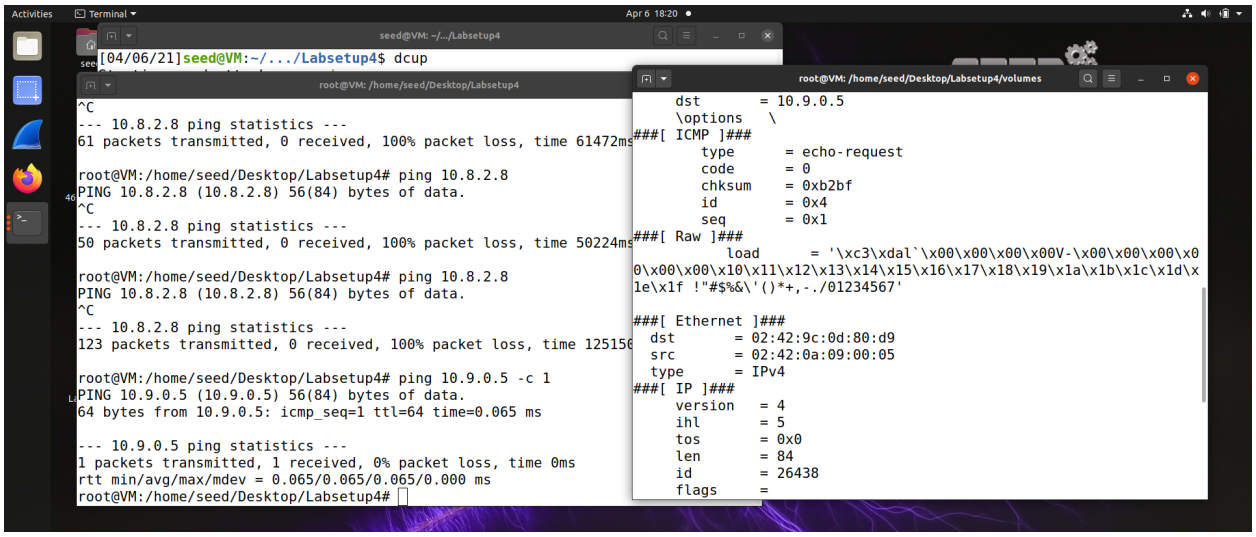
The purpose of this lab is to become familiar with how network packet sniffing and spoofing occurs. This lab uses two different languages, with the same approach in order to highlight the pros and cons of each language when it comes to spoof/sniffing packets. By creating docker containers on the VM, I was able to emulate a “victim” and “attacker” computer and send packets. By doing this, it eliminated the need for a separate computer outside of my VM. The first part of the lab consisted of packet sniffing/spoofing using Scapy.py. The second part of the lab used the pcap library to sniff/spoof packets in C. In order to make it easier to run attack programs on the “attacker” machine, a shared folder was created between that container and the main VM.

Tasks

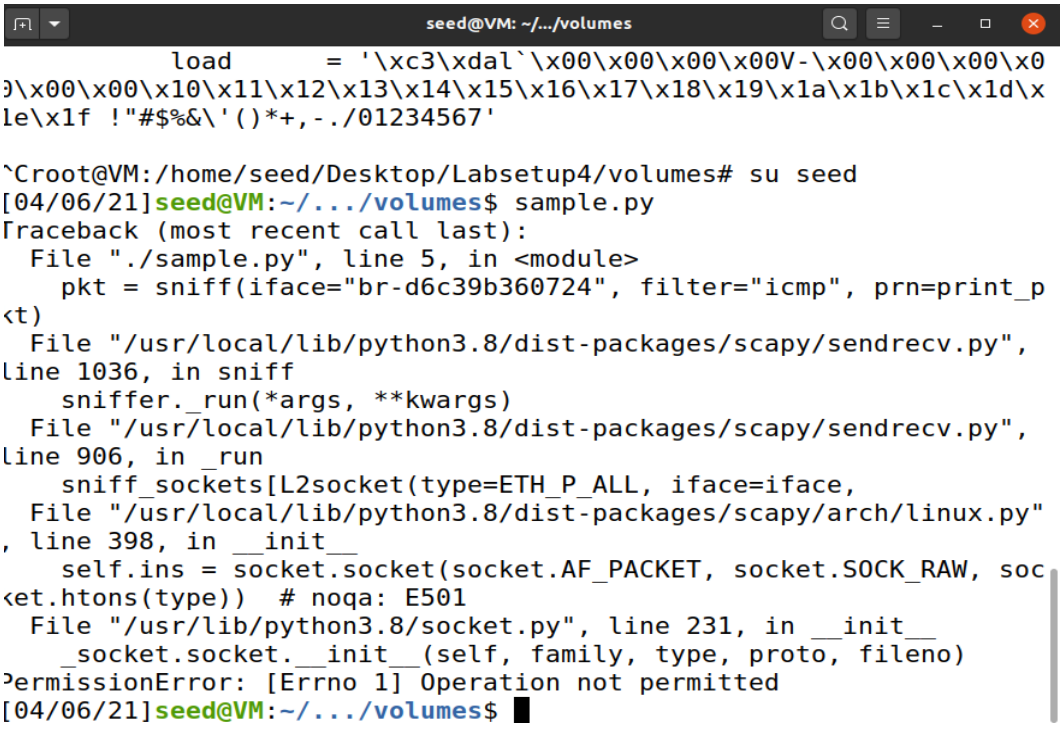
Task 1.1A

This task consisted of running a sniffer program to capture packets being sent from the host machine. After making the program executable and root privilege, the program worked correctly and printed out every sniffed packet that was sent through the host machine (see fig 1.1). I ran the same program but as a seed user. When I tried running the program, it failed and printed an error stating that the socket operation was not permitted. This socket operation is needed in order to successfully sniff/spoof packets. (see fig 1.2). This security measure is likely implemented to prevent users with regular permissions from seeing what packets are going to/from other applications that don’t belong to that user. Making it only root privilege prevents any user from seeing this network traffic. Nothing was changed from the code provided.

(fig 1.1)



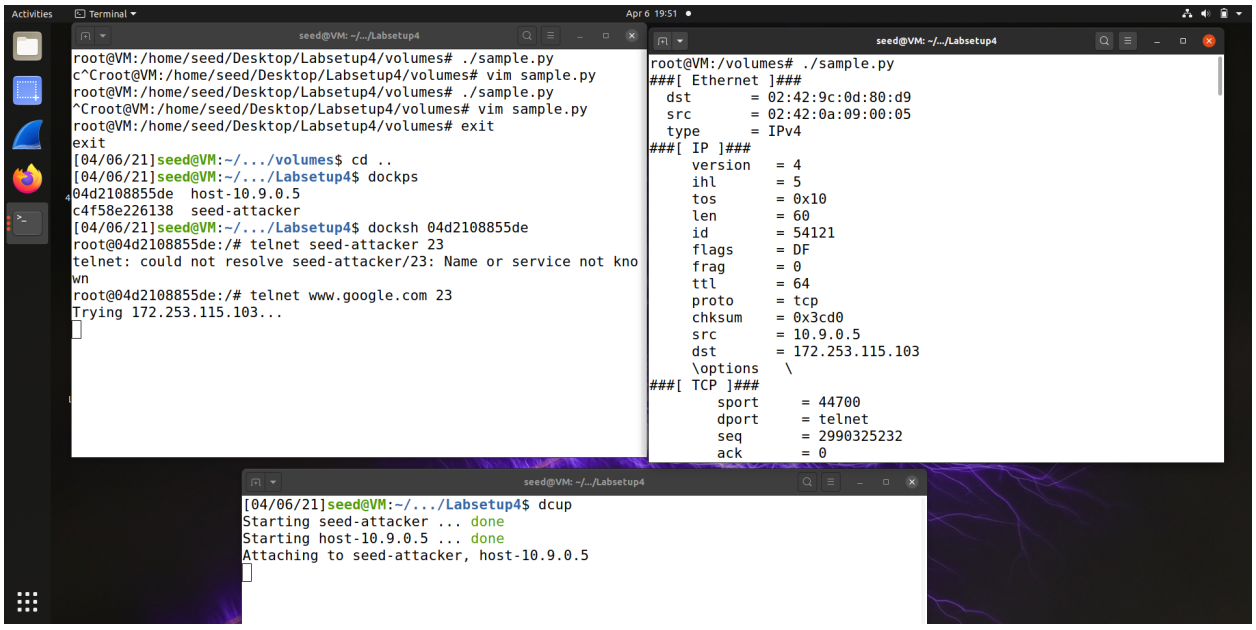
(fig 1.2)



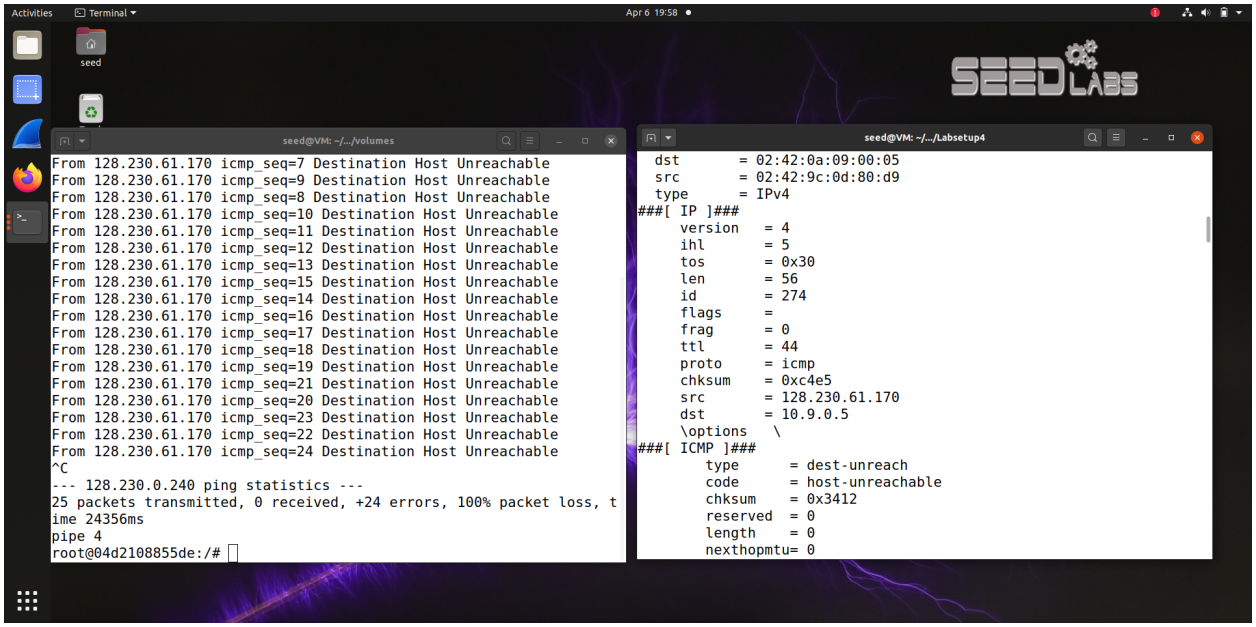
Task 1.1B

For this task, we did the same but applied filters to the sniffing program in order to get only the ICMP packets (fig 1.1), TCP packets from a specific IP and have a destination port of 23 (fig 1.5), and packets coming from or going to a particular subnet (fig 1.6). The subnet I chose to sniff belonged to my laptop. After calling each program, the desired results were achieved. Below are the results. For ICMP packets, the results are the same as in Task 1A since that was the original code for filters.

(fig 1.3, code modified by 'filter= tcp and (src host 10.9.0.5 and dst port 23')



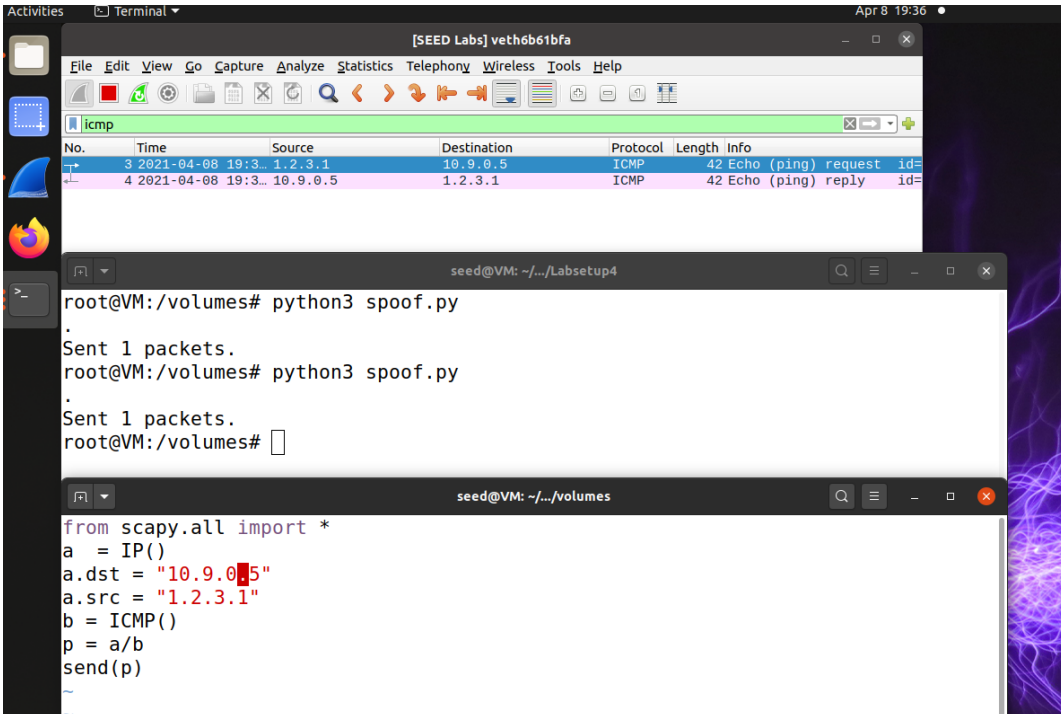
(fig 1.4 modified by setting 'filter= src net 169.254.51/16')



Task 1.2

This task consisted of spoofing ICMP packet requests and is achieved by setting every field of an ICMP header to a value that a normal ICMP packet header would have, despite having an arbitrary IP address. For this task, I used the IP address '1.2.3.1' to be the source and followed the model provided in the lab. Below are the code and resulting output.

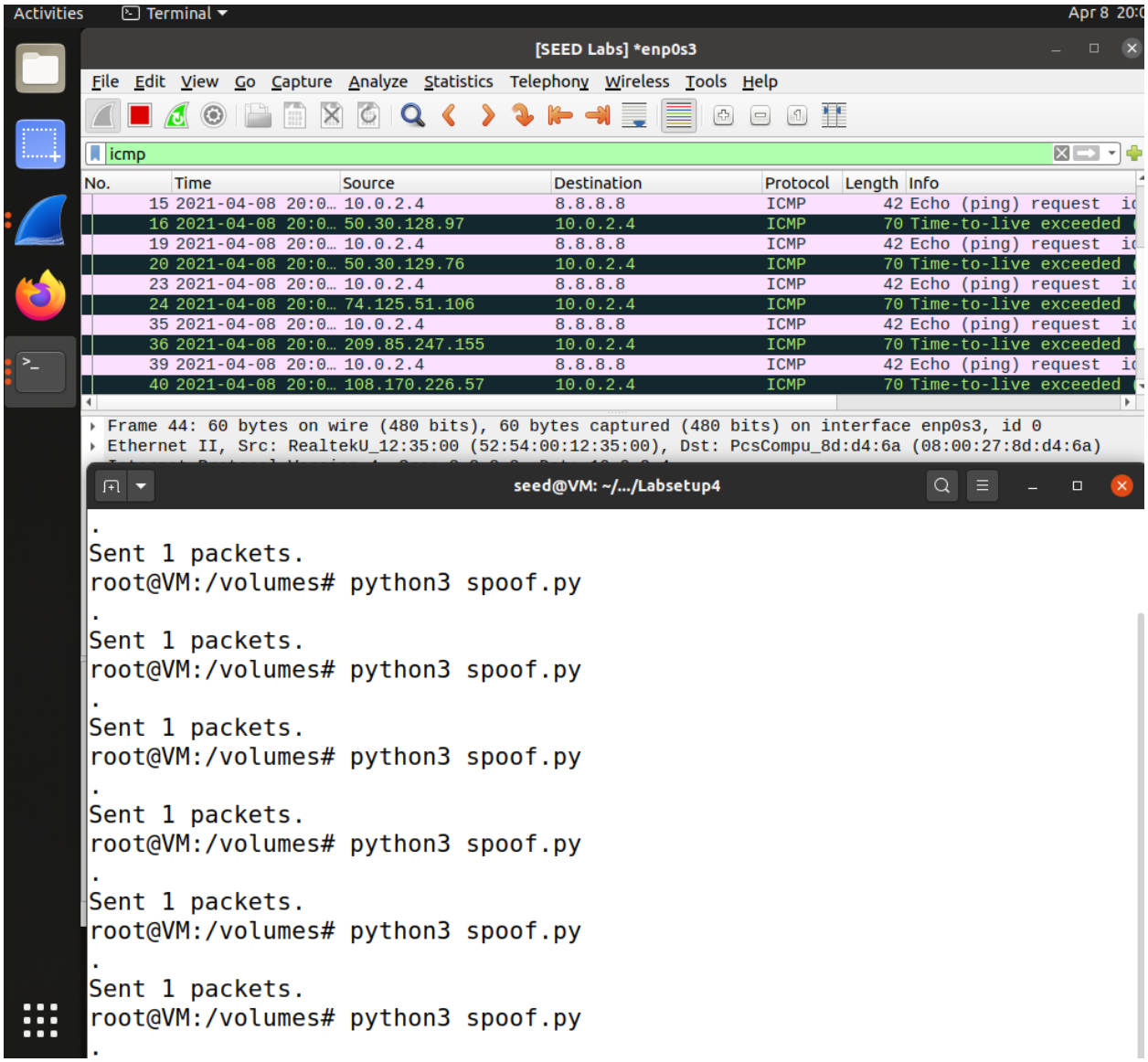
(fig 1.5)



Task 1.3

Task 1.3 consisted of providing a traceroute (estimation of how many routers away a destination is) gathered using scapy and slowly incrementing the packets 'TTL' until it reaches the right destination. The destination I chose was '8.8.8.8'. In order to help tell when a packet is failing, I used Wireshark to see packets being sent by the host machine (fig 1.6). Any dark highlighted line in (fig 1.6) signifies at what router the packet was dropped. By continuing this until it reaches the correct destination, I was able to determine that the program took 8 hops (8 routers, TTL=8) before it reached its destination. Below is a portion of the result and output. The code resembles the sample code and incrementing the TTL until it reaches its destination

(fig 1.6)

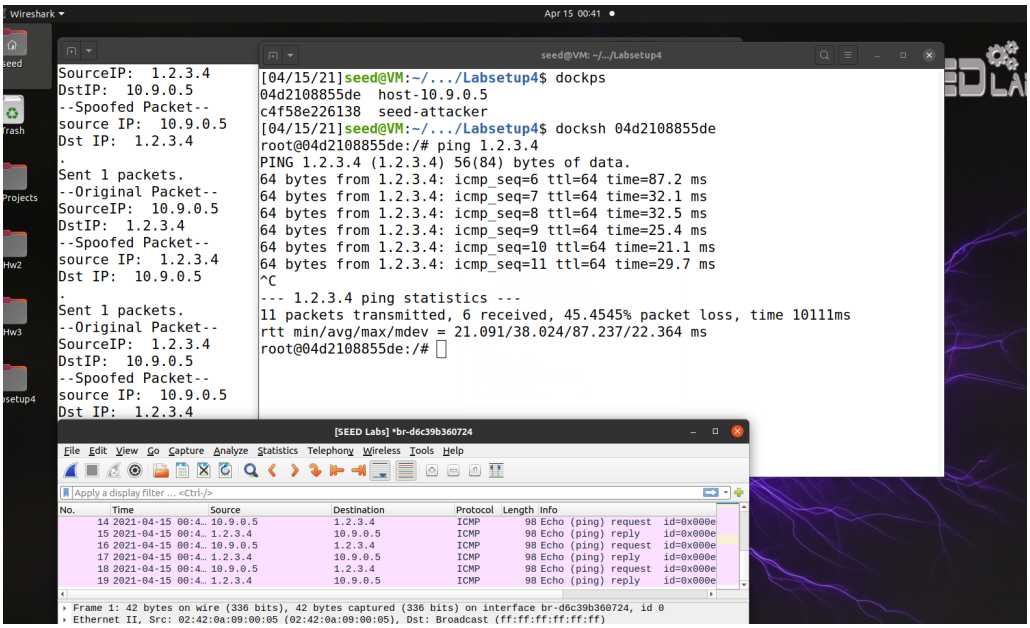


Task 1.4

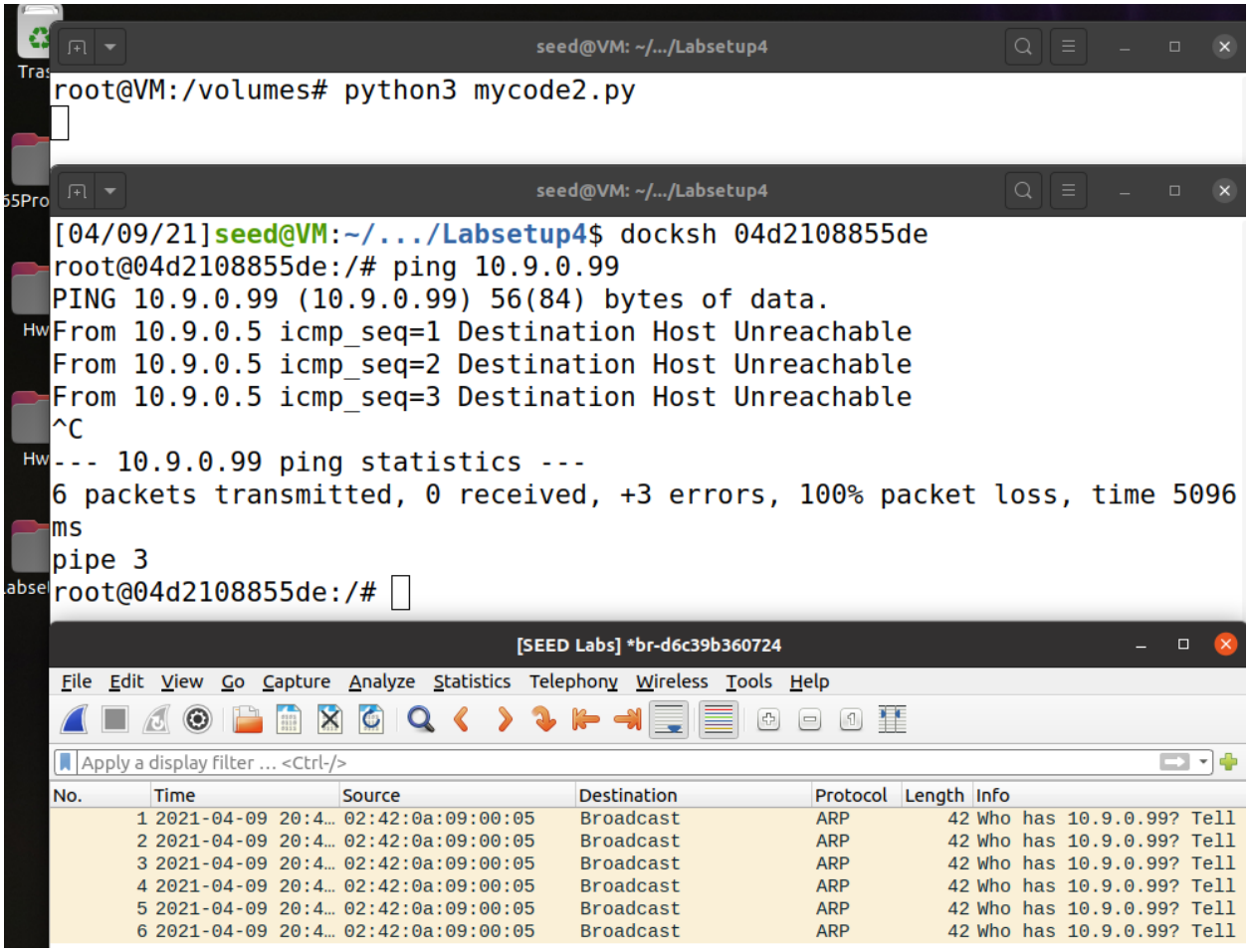
Task 1.4 required me to sniff a packet and then spoof it to the host machine. The first task (fig 1.7) shows this being done for a non-existing host but with the host machine getting a reply. The second task (fig 1.8) was from a non-existing host on the same LAN, and the last task was

spoofing a packet from a host on the internet (fig 1.9). The method was the same for all 3 tasks and further demonstrates how my sniff/spoof program is successful. Below also includes the code for this task (fig 1.10).

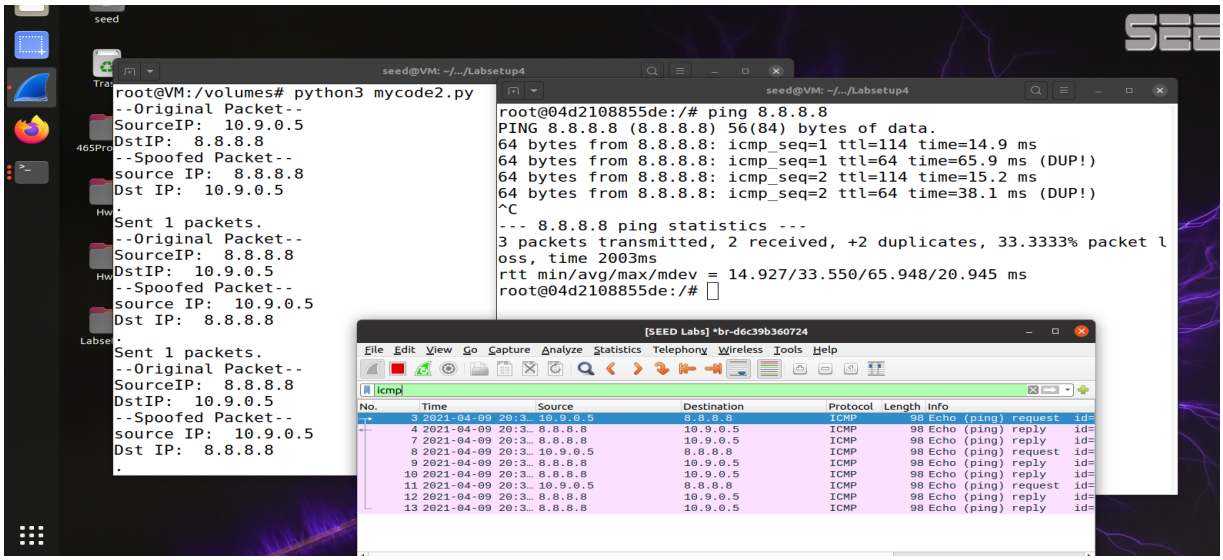
(fig 1.7)



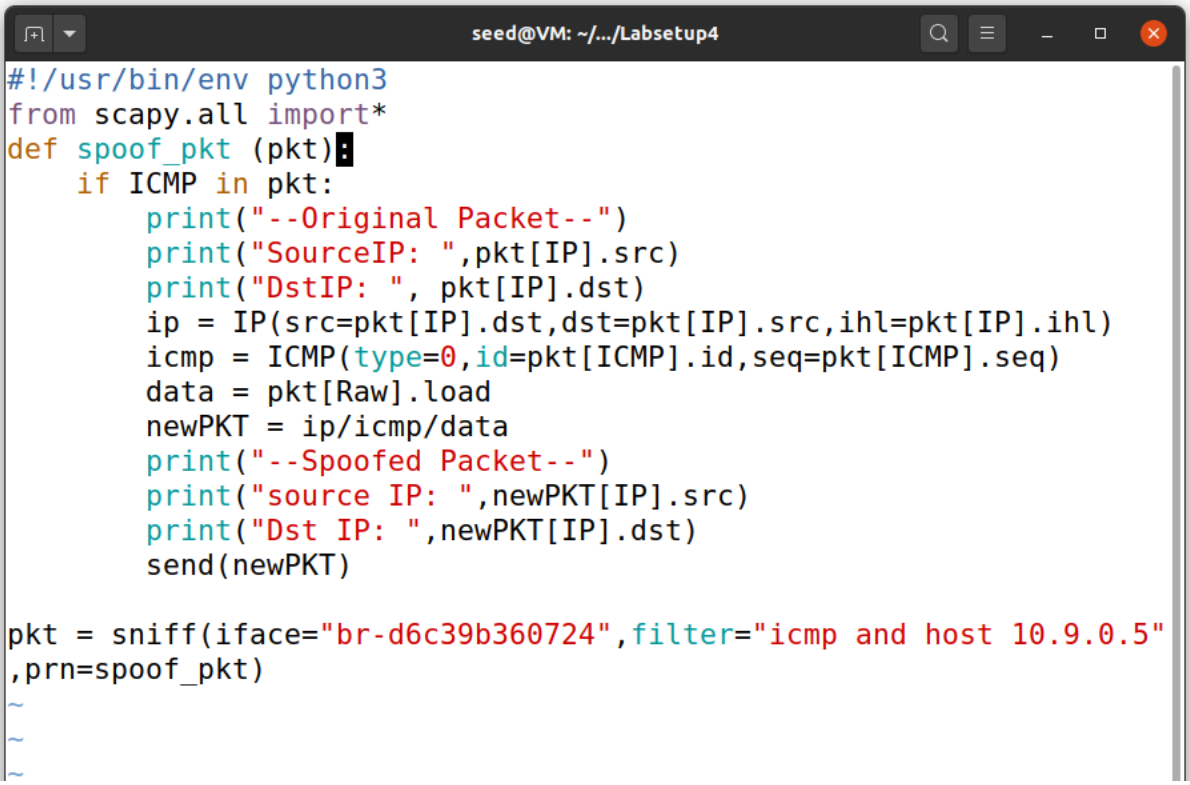
(fig 1.8)



(fig 1.9)



(fig 1.10)



Task 2.1A

Task 2.1 and all the tasks that follow are similar to the previous tasks but are written in C. Using the pseudocode provided in the lab, I was able to successfully sniff packets that were being sent to and from the host machine.

-Question 1: In order to run a sniffer program, pcap_lookupdev must first be called to find a capture device to sniff. The second library call is to pcap_lookupnet which returns the network number and mask for that capture device. The third library call is to pcap_open_live to start an active sniffing session on that capture device. The next library call is to pcap_datalink which returns what kind of device the program is capturing on. Pcap_compile creates the filter expression and stores it in a string so it can correctly set that filter in pcap_loop. With the pcap_loop library call, a callback is created (if the filter is matched with the incoming packet, it will go to a function defined with the 'pcap_loop' call). Pcap_dreecode frees up any allocated memory created, and pcap_close closes the sniffing session.

-Question 2: As mentioned in task 1.1, the root privilege is required because regular users aren't allowed to access network traffic.

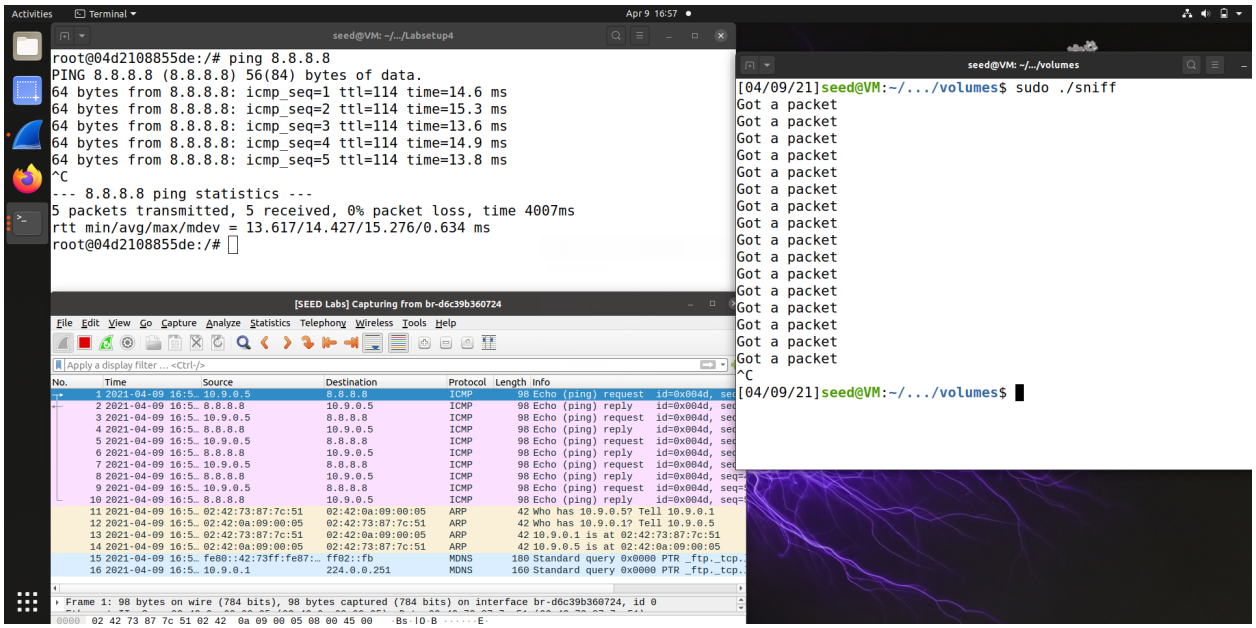
-Question 3: Promiscuous mode is a boolean parameter that when set to true, will only sniff packets until an error occurs. If promiscuous mode is set to false, it will sniff all traffic regardless of if there is an error or not. Despite this, there are some drawbacks to using promiscuous mode such as a host is able to tell if a sniffer is running in promiscuous mode. Promiscuous mode can also be unhelpful if you're trying to capture malformed packets or not sending malformed packets Figs 2.3 and 2.4 show the program running in promiscuous and not promiscuous, respectively.

(fig 2.1)

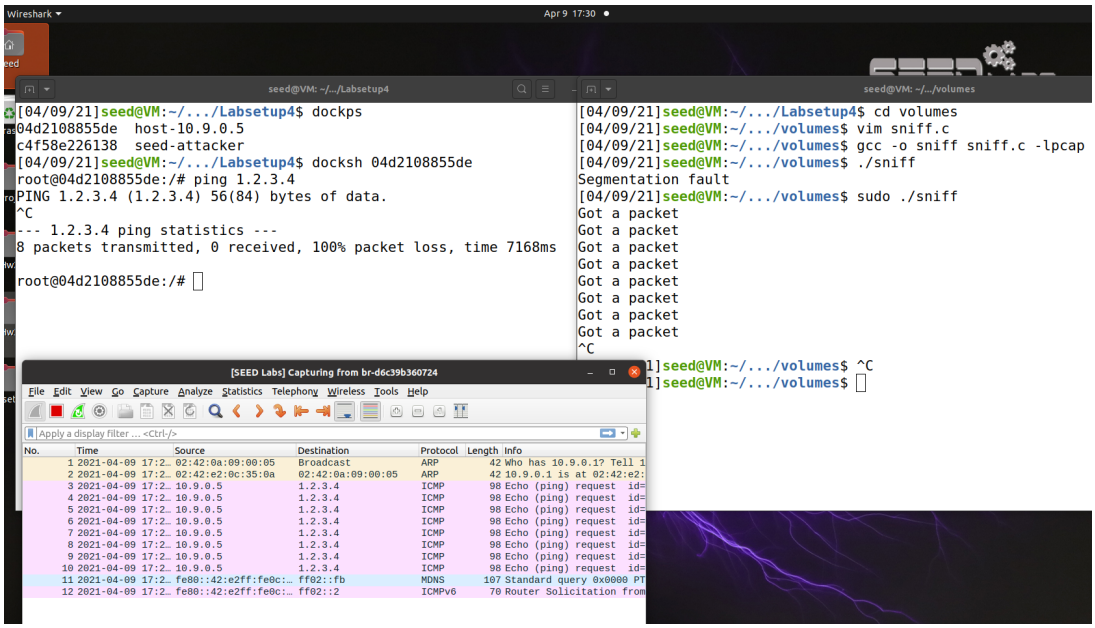

seed@VM: ~/.../Labsetup4
#define TH_CWR 0x80
#define TH_FLAGS (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
u_short th_win; /* window */
u_short th_sum; /* checksum */
u_short th_urp; /* urgent pointer */
};
void got_packet(u_char *args, const struct pcap_pkthdr *header,
const u_char *packet)
{
printf("Got a packet with data\n");
//Ethernet header is always 14 bytes
const char* payload;
const struct sniff_ip* iP = (struct sniff_ip*)(packet+14);
uint8_t size_ip = IP_HL(iP)*4;
const struct sniff_tcp* tcP = (struct sniff_tcp*)(packet+14+size_ip);
uint8_t size_tcp = TH_OFF(tcP)*4;
payload = (u_char*)(packet+14+size_ip+size_tcp);
uint8_t payloadSize = ntohs(iP->ip_len) - (size_ip + size_tcp);
printf("PayloadSize = %d\n",payloadSize);
//Print payload data - might be in binary
if(payloadSize > 0)
print_payload(payload,payloadSize);
}
void print_payload(const u_char* payload, int length){
int gap = 0;
const u_char* ch = payload;
for(int i = 0; i < length; i++){
if(isprint(*ch))
printf("%c", *ch);
else
printf(".");

(fig 2.2)

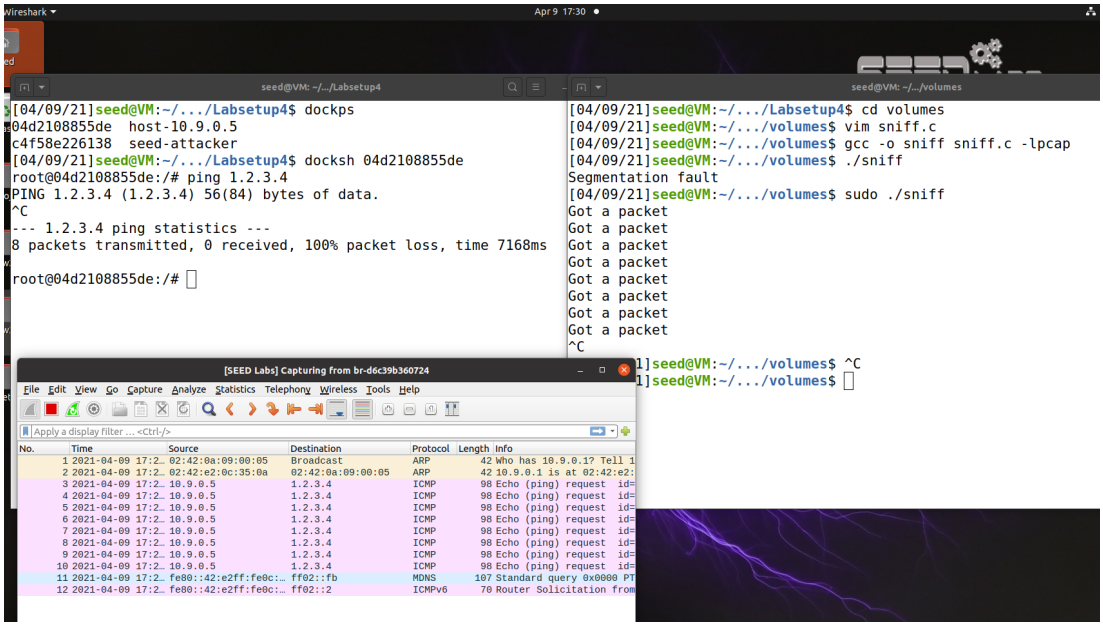
HW 4: Packet Sniffing and Spoofing



(fig 2.3)



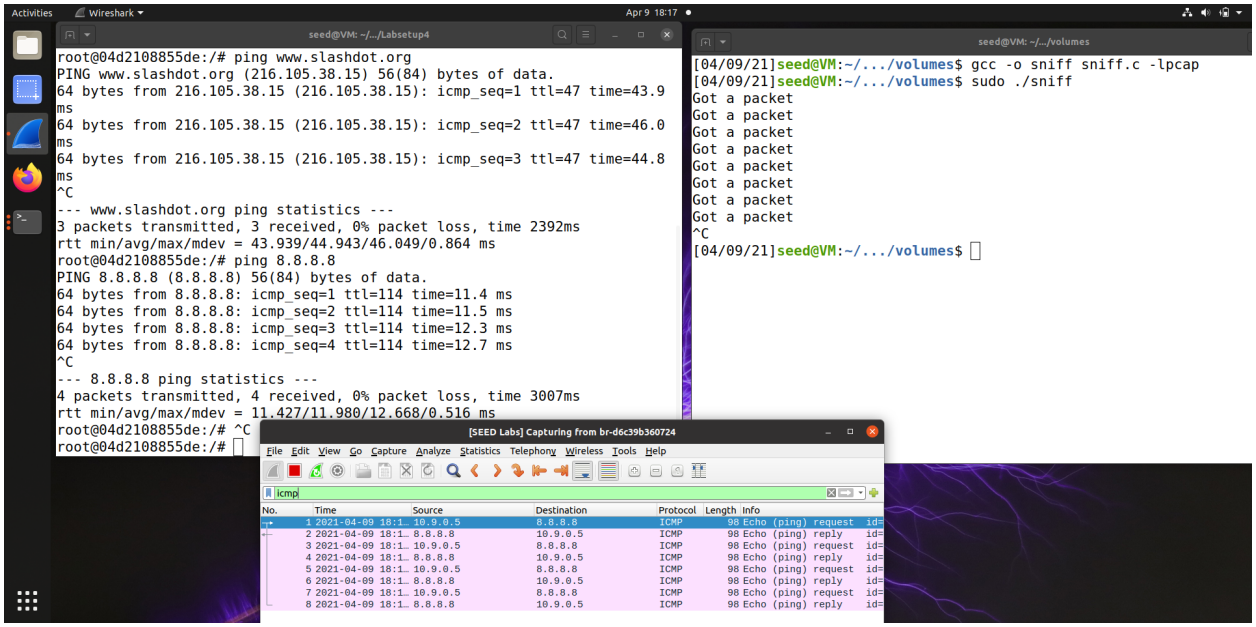
(fig 2.4)



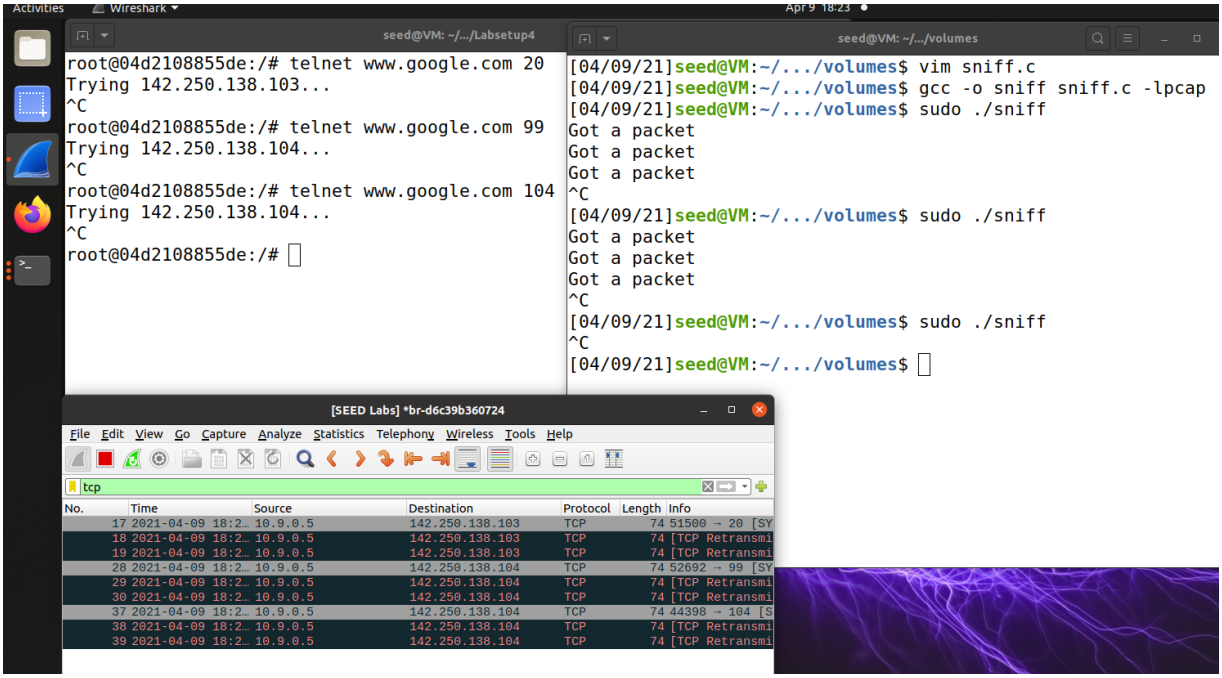
Task 2.1B

In this task, filters are set by changing the filter_expr variable to include the specific filters outlined by the lab. Fig 2.5 shows when an ICMP packet filter is set and fig 2.6 shows when TCP is set and a range of destination ports.

(fig 2.5)



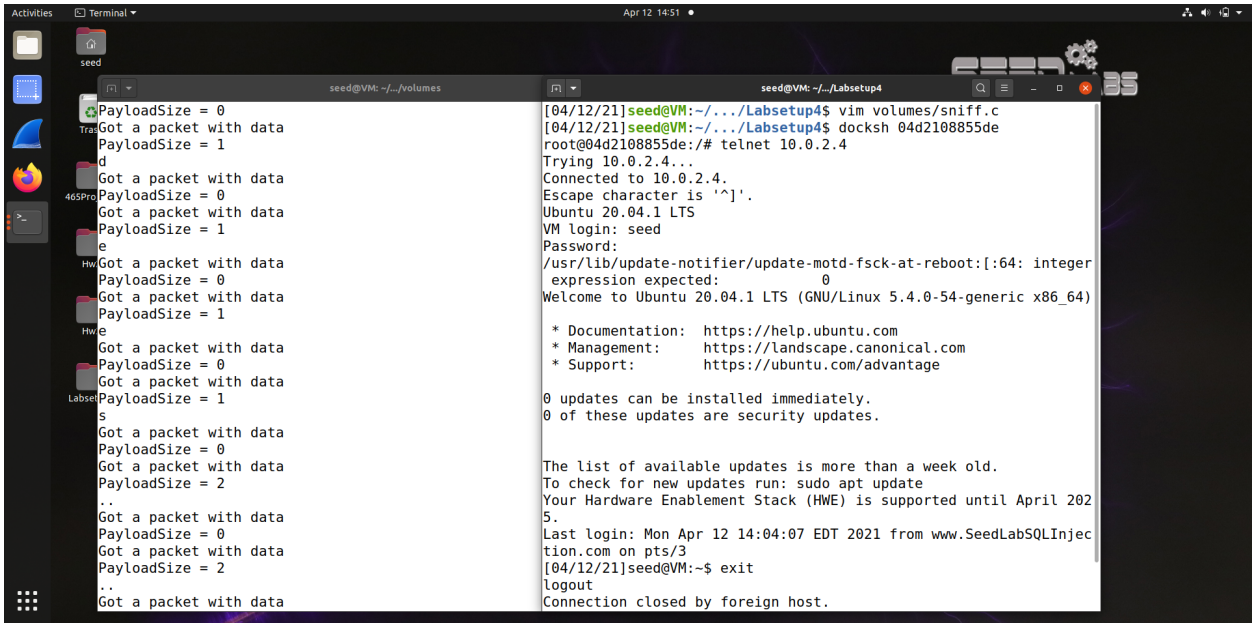
(fig 2.6)



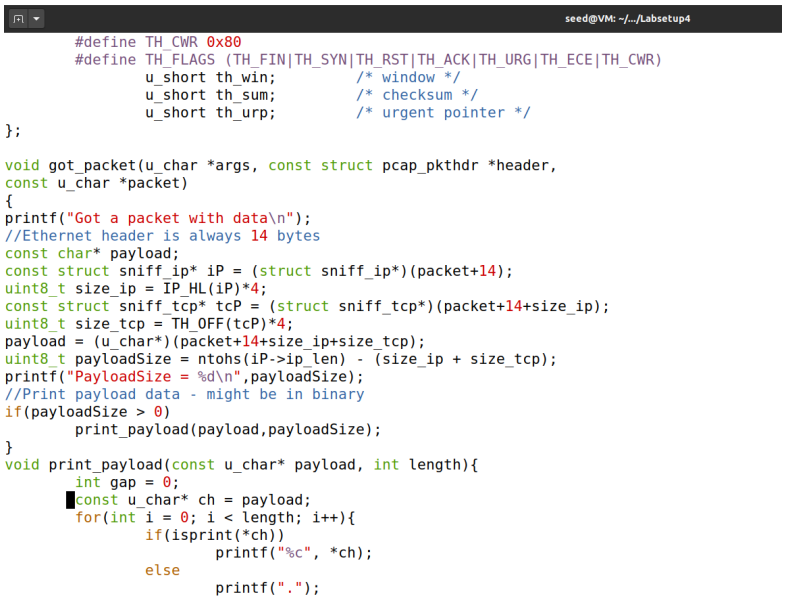
Task 2.1C

In task 2.1C I used the sniffer program and exploited a bug in the “telnet” network call. Telnet has a bug that will print the host user’s information in clear text (whatever that user types). For example, if one host is calling telnet to another host, that new host might ask the original host to log in to the shell. By doing this, the last bit of each telnet packet holds one bit of what the user is or has typed. By sniffing this network, one can easily gain the credentials of the user who called telnet. Below is the result that led to me gaining a password from telnet. The letters captured from the packet were converted from hex to ASCII (partially seen in fig 2.8) in order to create the letters seen underneath “payload size” in fig 2.7. The other terminal shows what the host machine has written.

(fig 2.7)



(fig 2.8)



Task 2.2 A & B

Task 2.2 consists of spoofing a request. The second part required spoofing an ICMP request packet. Because both parts of task 2.2 were very similar, I decided to combine them. In order to do this, all of the fields that aren't 0 must be set, and then checksum has to be determined before sending it through the socket and over the network. The checksum code was derived from the *Hands-on approach* textbook. After this, the socket must be successfully sent through the socket. Since this is an ICMP request packet, a payload doesn't have to be attached after the correct headers and fields are set. Fig 2.10 also shows the result. Although a reply isn't

seen, I was able to see that the packet was successfully sent and accepted as a correct ICMP request packet despite the source IP address being arbitrary. This is depicted by the Wireshark snippet.

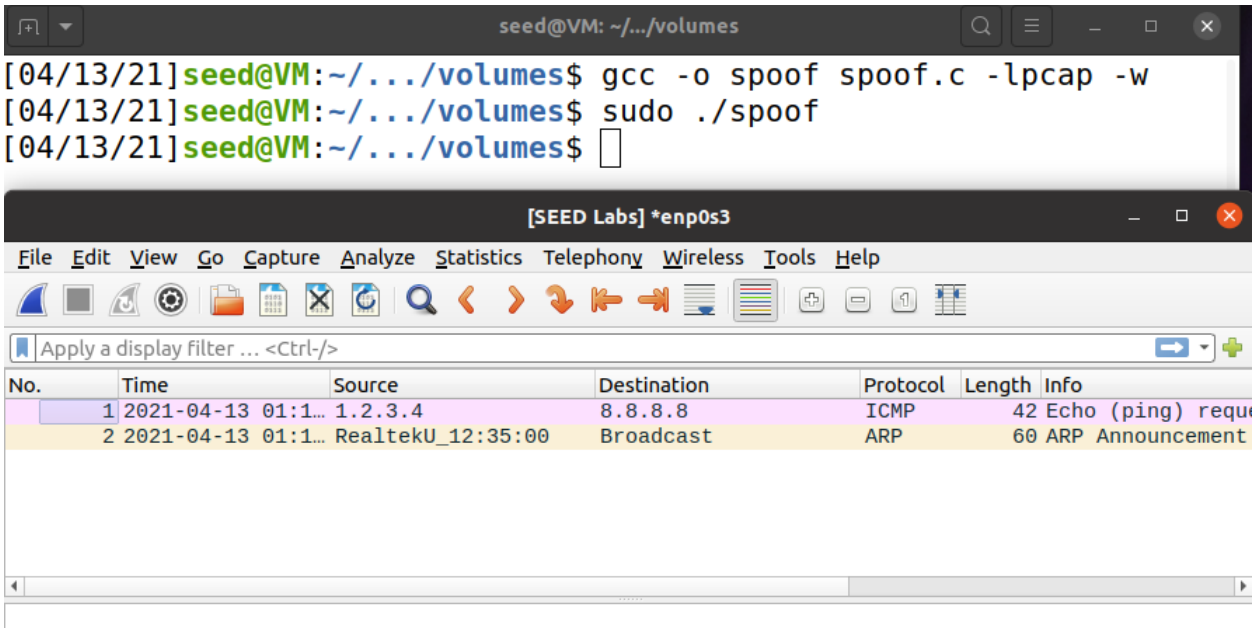
-Question 4: You can set the IP packet length to an arbitrary value because the excess length will be padded with zeros. If this is done, the total header length will also have to be adjusted.

-Question 5: With raw socket programming, you don't have to calculate the checksum for the IP header, the OS does that automatically. However, you do have to calculate the checksum for the ICMP header.

-Question 6: You need a root program because raw sockets allow a user to spoof custom packets. This can be dangerous if a malicious user is able to spoof packets. Additionally, spoofing custom packets might interfere with inbound traffic if not done correctly, and could break other network protocols and still send the socket leading to problems in the network.

(fig 2.9)

(fig 2.10)

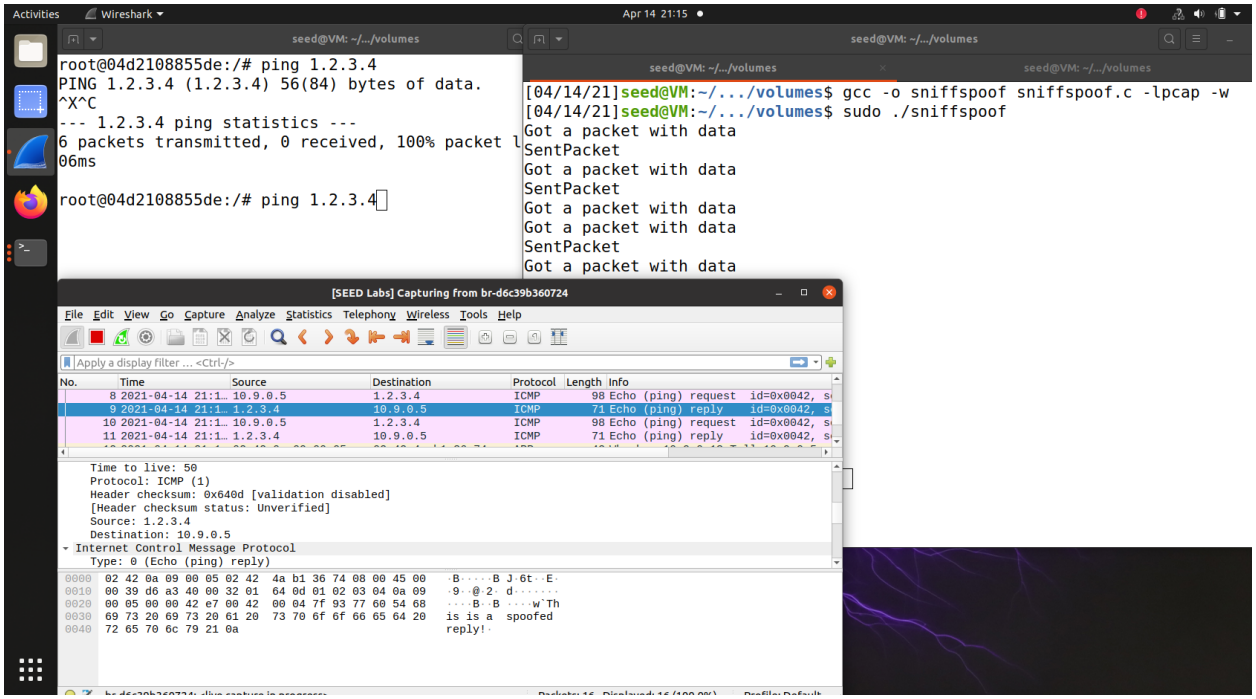


Task 2.3

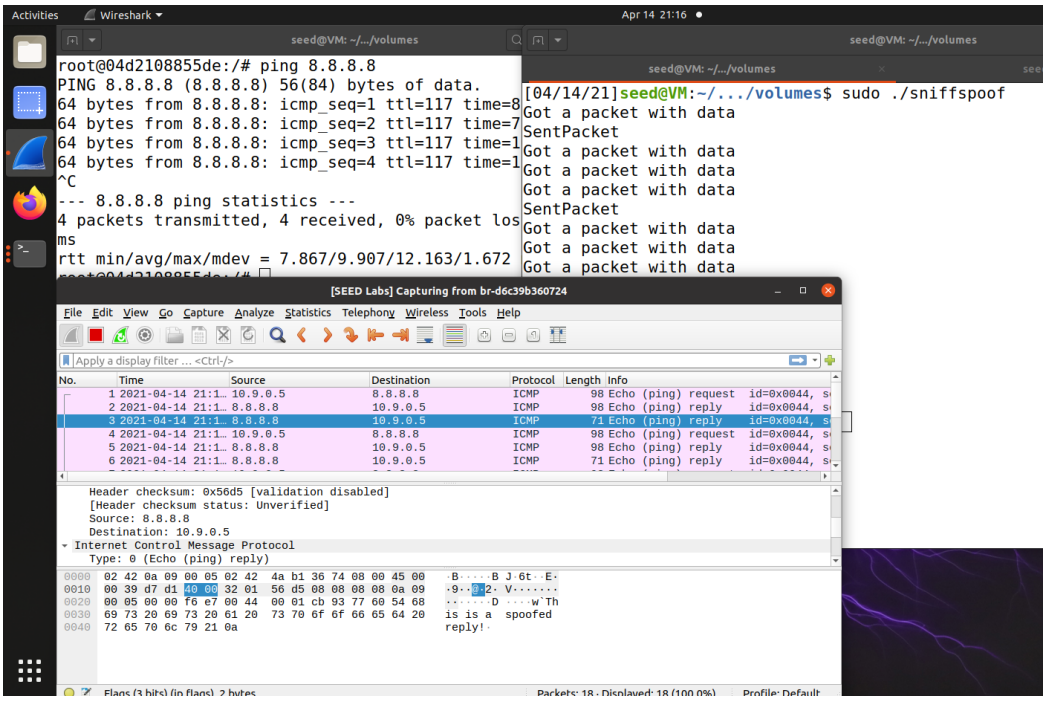
Task 2.3 is similar to task 1.4 where it is sniffing and spoofing an ICMP reply. I did this by first including code to sniff ICMP packets in the main function. Once a packet was sniffed and was an ICMP request, another function was called to actually spoof the packet and send it to the src host. Looking at fig 2.11, the wireshark trace shows that a successful reply was constructed, however the checksum is wrong, preventing the packet actually being sent to the host machine. This is because ICMP replies require the original data to be in front of the new data (right after the ICMP header). Through numerous tries of getting the original payload correctly placed into the spoofed packet, I wasn't able to achieve a correct spoof. This could be a result of how I'm passing in the ip which might not be including the data. If the data was correctly placed in front of the payload, the checksum would be correctly calculated, and the host machine would've gotten my spoofed packet response. I included fig 2.12 because it shows how the ICMP spoofed packet reached the host machine after the actual ICMP packet reached the host. This could be a result of the calculations/set up taking longer or the spoofed packet's checksum being wrong. One way to fix this would be to also sniff and block the connection to the live destination that way the only packets the source host sees is the spoofed packets. Fig 2.13-2.15 shows the modified code.

HW 4: Packet Sniffing and Spoofing

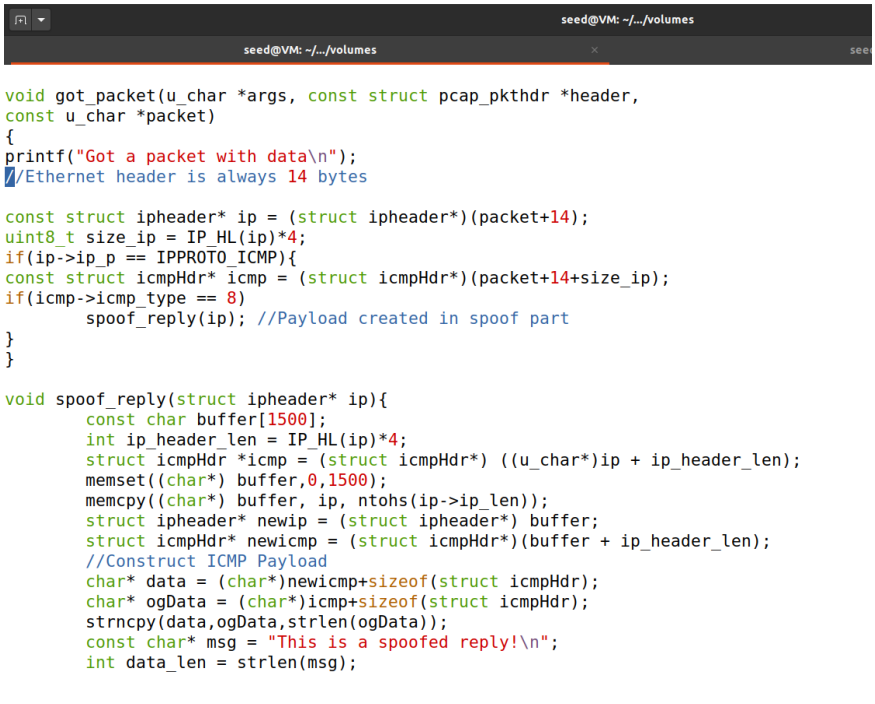
(fig 2.11)



(fig 2.12)



(fig 2.13)




```
void got_packet(u_char *args, const struct pcap_pkthdr *header,
const u_char *packet)
{
printf("Got a packet with data\n");
//Ethernet header is always 14 bytes

const struct ipheader* ip = (struct ipheader*)(packet+14);
uint8_t size_ip = IP_HL(ip)*4;
if(ip->ip_p == IPPROTO_ICMP){
const struct icmpHdr* icmp = (struct icmpHdr*)(packet+14+size_ip);
if(icmp->icmp_type == 8)
        spoof_reply(ip); //Payload created in spoof part
}
}

void spoof_reply(struct ipheader* ip){
        const char buffer[1500];
        int ip_header_len = IP_HL(ip)*4;
        struct icmpHdr *icmp = (struct icmpHdr*) ((u_char*)ip + ip_header_len);
        memset((char*) buffer,0,1500);
        memcpy((char*) buffer, ip, ntohs(ip->ip_len));
        struct ipheader* newip = (struct ipheader*) buffer;
        struct icmpHdr* newicmp = (struct icmpHdr*)(buffer + ip_header_len);
        //Construct ICMP Payload
        char* data = (char*)newicmp+sizeof(struct icmpHdr);
        char* ogData = (char*)icmp+sizeof(struct icmpHdr);
        strncpy(data,ogData,strlen(ogData));
        const char* msg = "This is a spoofed reply!\n";
        int data_len = strlen(msg);
```

(fig 2.14)

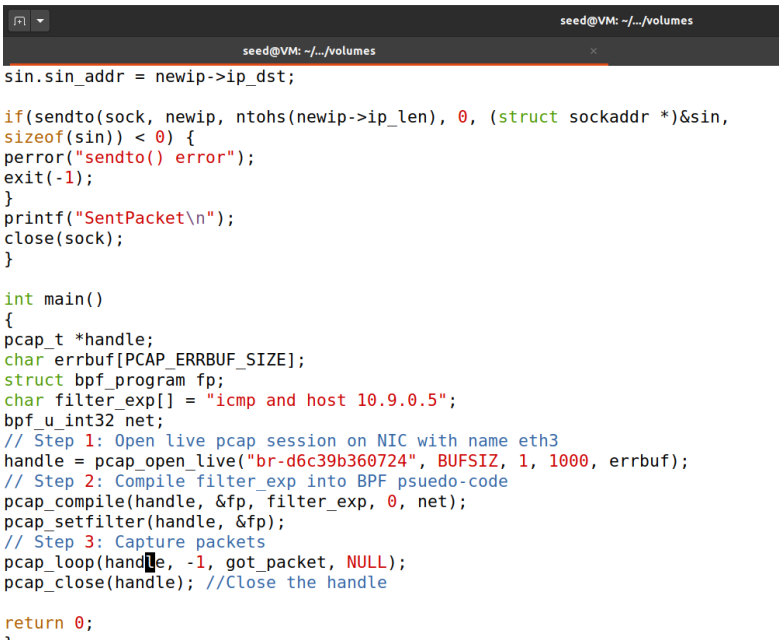

```
        strncpy(data,ogData,strlen(ogData));
        const char* msg = "This is a spoofed reply!\n";
        int data_len = strlen(msg);
        strncpy(data+strlen(ogData),msg,data_len);
        data_len = data_len + strlen(ogData);
        //Construct ICMP header
        newicmp->icmp_type = 0;
        newicmp->icmp_chksum = 0;
        newicmp->icmp_chksum = calc_checksum((u_short*)newicmp, sizeof(struct icmpHdr)+data_len);
        //Construct IP header
        newip->ip_src.s_addr = ip->ip_dst.s_addr;
        newip->ip_dst.s_addr = ip->ip_src.s_addr;
        newip->ip_ttl = 50;
        newip->ip_len = htons(sizeof(struct ipheader)+sizeof(struct icmpHdr)+data_len);
        //Sending out the packet
int sock;
struct sockaddr_in sin;

sock = socket(AF_INET,SOCK_RAW,IPPROTO_RAW);
if(sock < 0){
perror("socket() error");
exit(-1);
}
int enable = 1;
//setsockopt(sock,IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));

sin.sin_family = AF_INET;
sin.sin_addr = newip->ip_dst;

if(sendto(sock, newip, ntohs(newip->ip_len), 0, (struct sockaddr *)&sin,
```

(fig 2.15)


seed@VM: ~/.../volumes
seed@VM: ~/.../volumes
sin.sin_addr = newip->ip_dst;
if(sendto(sock, newip, ntohs(newip->ip_len), 0, (struct sockaddr *)&sin,
sizeof(sin)) < 0) {
perror("sendto() error");
exit(-1);
}
printf("SentPacket\n");
close(sock);
}
int main()
{
pcap_t *handle;
char errbuf[PCAP_ERRBUF_SIZE];
struct bpf_program fp;
char filter_exp[] = "icmp and host 10.9.0.5";
bpf_u_int32 net;
// Step 1: Open live pcap session on NIC with name eth3
handle = pcap_open_live("br-d6c39b360724", BUFSIZ, 1, 1000, errbuf);
// Step 2: Compile filter_exp into BPF psuedo-code
pcap_compile(handle, &fp, filter_exp, 0, net);
pcap_setfilter(handle, &fp);
// Step 3: Capture packets
pcap_loop(handle, -1, got_packet, NULL);
pcap_close(handle); //Close the handle
return 0;