Lab 15: Packet Sniffing and Spoofing Lab

By Michael Minogue

4/14/20

—

CSP 544

—

A20388449

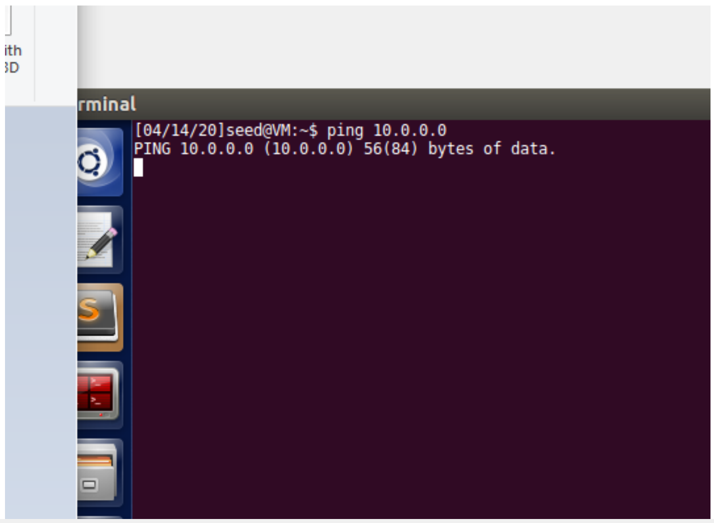
Lab Task Set 1: Using Tools to Sniff and Spoof Packets

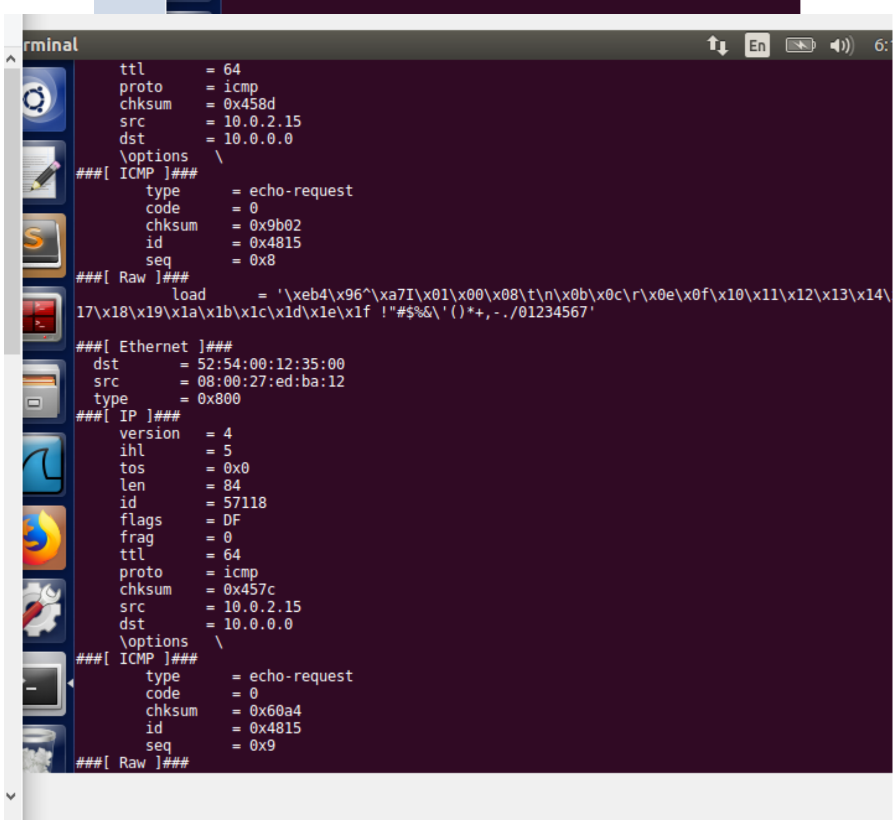
Task 1.1: Sniffing Packets

Task 1.1A:

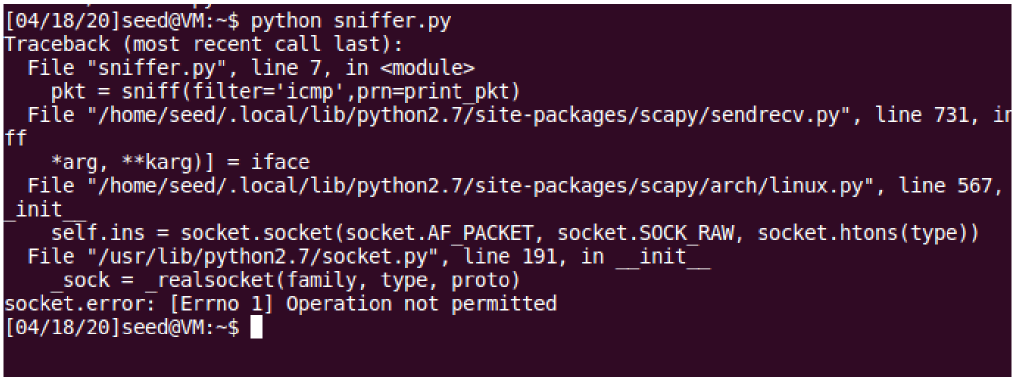
We begin by running the provided sample code to sniff packets. This will print the packet information to the console using the print_pkt() function, allowing us to gather valuable data.

I set up one VM with the ping command, sending a ping with 10.0.0.0 as the destination I then set up another VM to sniff the ping with root privilege, which it successfully does.





When attempting to run the program without root access, we are given the error message "operation not permitted". This is because the system seems to need direct access to the network device to do sniffing, which requires root privilege.



Task 1.1B:

a. Capture only the ICMP packet

Thankfully our code actually already does this. If you look at the provided sample code, it has the line "pkt = sniff(filter='icmp', prn=print_pkt)". The BPF filter is already set to only capture ICMP packets, so you can see Task 1.1A for an example of ICMP capturing.

b. Capture any TCP packet that comes from a particular IP and with a destination port number 23.

sniffer2.py   spoofer.py

```python
#!/usr/bin/python
from scapy.all import *

def print_pkt(pkt):
    pkt.show()

pkt = sniff(filter='tcp and dst 12.12.12.12/23',prn=print_pkt)
```

The filter section of our code is altered so that TCP packets are received, with 12.12.12.12 as the source IP and port 23 the destination port.

We then use the spoof program (given later in the lab) to create a test TCP packet with destination 12.12.12.12/23. This is sent in a separate console window. After checking back up on our sniffer program, we see it successfully intercepted the TCP message!

```
###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:ed:ba:12
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 40
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x55b6
     src       = 10.0.2.15
     dst       = 12.12.12.255
     \options   \
###[ TCP ]###
        sport     = ftp_data
        dport     = http
        seq       = 0
        ack       = 0
        dataofs   = 5
        reserved  = 0
        flags     = S
        window    = 8192
        chksum    = 0x6a65
        urgptr    = 0
        options   = []

###[ Ethernet ]###
  dst       = 52:54:00:12:35:00
  src       = 08:00:27:ed:ba:12
  type      = 0x800
```

```
[04/18/20]seed@VM:~$ sudo spoofer.py
sudo: spoofer.py: command not found
[04/18/20]seed@VM:~$ sudo python spoofer.py
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
.....................................................................................
...............................................................
Sent 512 packets.
[04/18/20]seed@VM:~$
```

c. Capture packets to or from a subnet

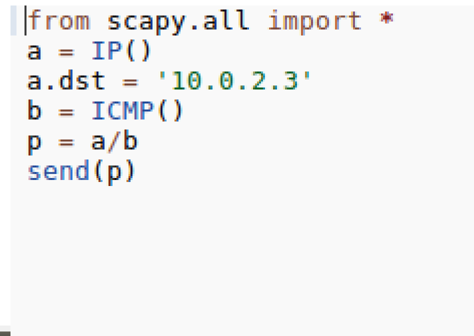
```
sniffer3.py

#!/usr/bin/python
from scapy.all import *

def print_pkt(pkt):
    pkt.show()

pkt = sniff(filter='tcp and src 128.230.0.0/16',prn=print_pkt)
```

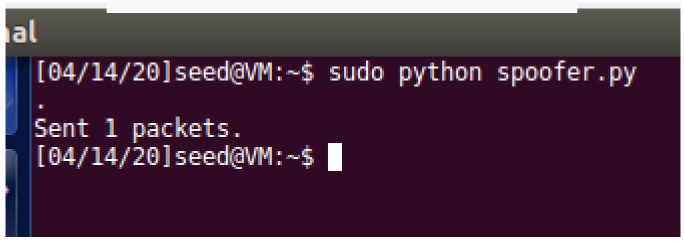
As you can see, we set the filter format to tcp and the src address as 128.230.0.0/16, the one required in the book. I saved the file as sniffer3.py and executed it, generating a successful stream of sniffing.

The sniffer was still gathering UDP messages as well, which seemed incorrect, but overall the program functioned well and at least sorted the destinations correctly.

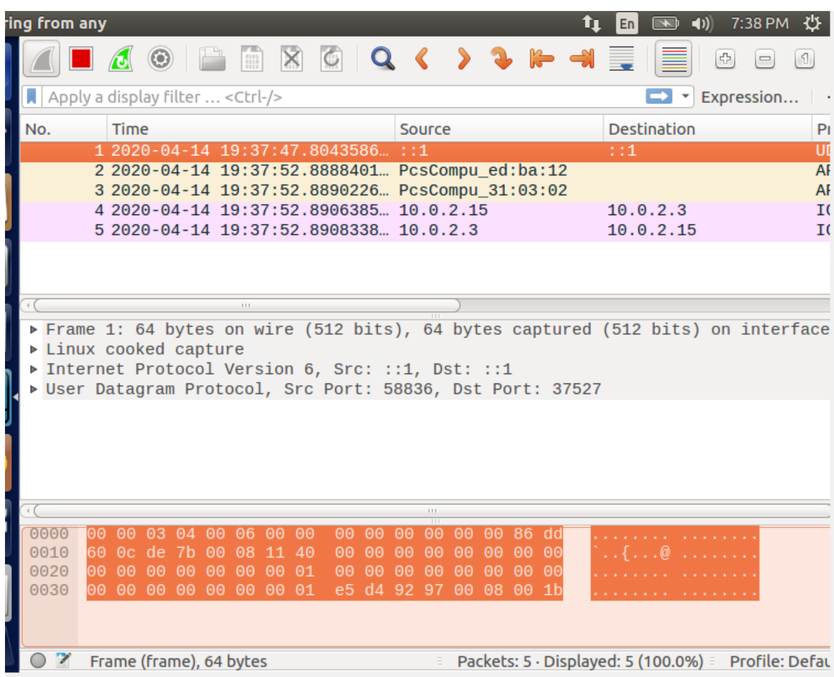
```
rminal
[04/14/20]seed@VM:~$ sudo python sniffer3.py
tcpdump: Mask syntax for networks only
###[ Ethernet ]###
  dst       = 00:00:00:00:00:00
  src       = 00:00:00:00:00:00
  type      = 0x86dd
###[ IPv6 ]###
     version   = 6
     tc        = 0
     fl        = 843387
     plen      = 8
     nh        = UDP
     hlim      = 64
     src       = 128.230.0.0
     dst       = 10.0.2.15
###[ UDP ]###
        sport     = 58836
        dport     = 37527
        len       = 8
        chksum    = 0x1b

###[ Ethernet ]###
  dst       = 00:00:00:00:00:00
  src       = 00:00:00:00:00:00
  type      = 0x86dd
###[ IPv6 ]###
     version   = 6
     tc        = 0
     fl        = 843387
     plen      = 8
     nh        = UDP
     hlim      = 64
     src       = 128.230.0.0
     dst       = 10.0.2.15
###[ UDP ]###
```

Task 1.2: Spoofing ICMP Packets





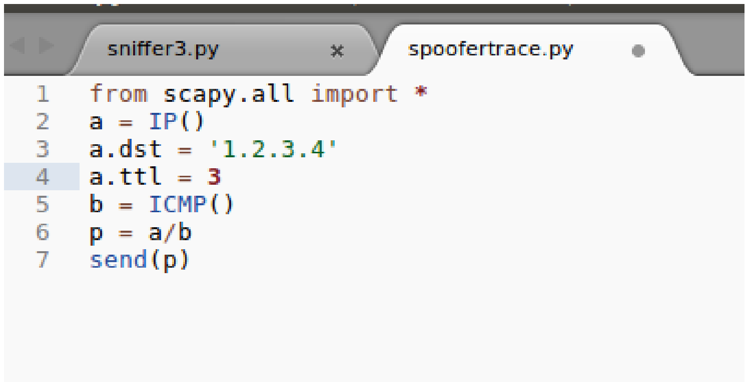
Using scapy, this code (provided on the lab document) allows us to send packets, which you can see was done above. Using Wireshark I could verify that the packet was actually sent, which you can see by the source/destination arraignment. This shows a packet was sent, then a reply was sent back.

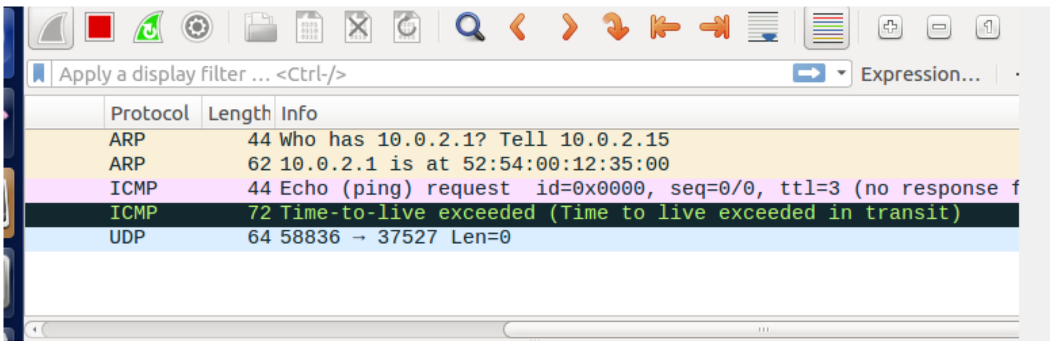


Task 1.3: Traceroute

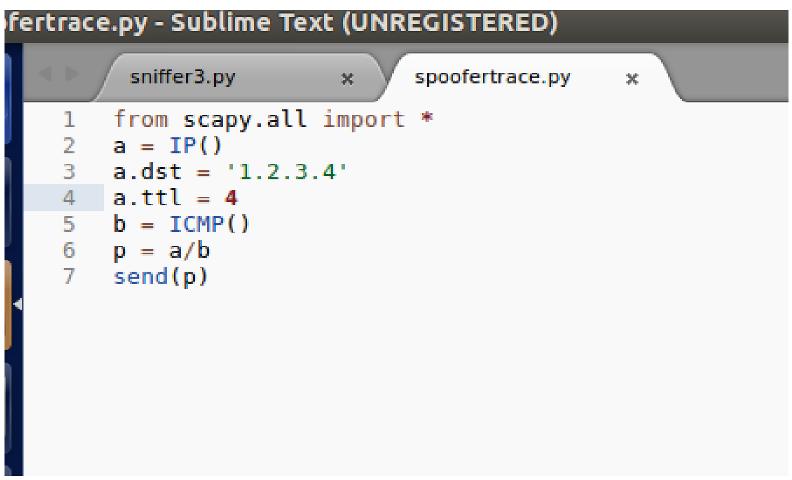
For simplicities sake, I decided to maintain sample code as close to the original lab as possible, and so decided to find the traceroute to 1.2.3.4 using Wireshark.

I initially used the exact same ttl setting as the sample code, to see whether I needed to go higher or possibly lower to find the number of hops. Running the code, the packet failed to arrive, meaning higher we go.

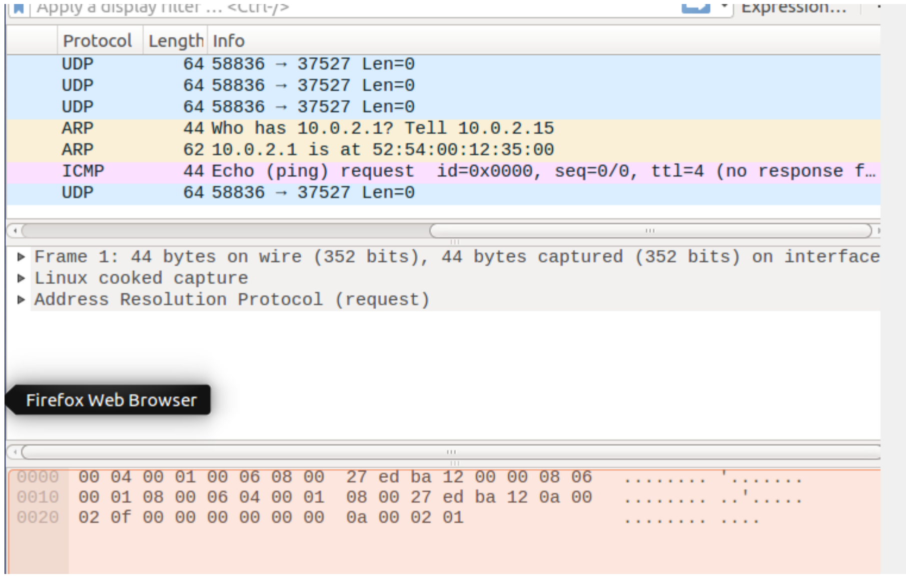




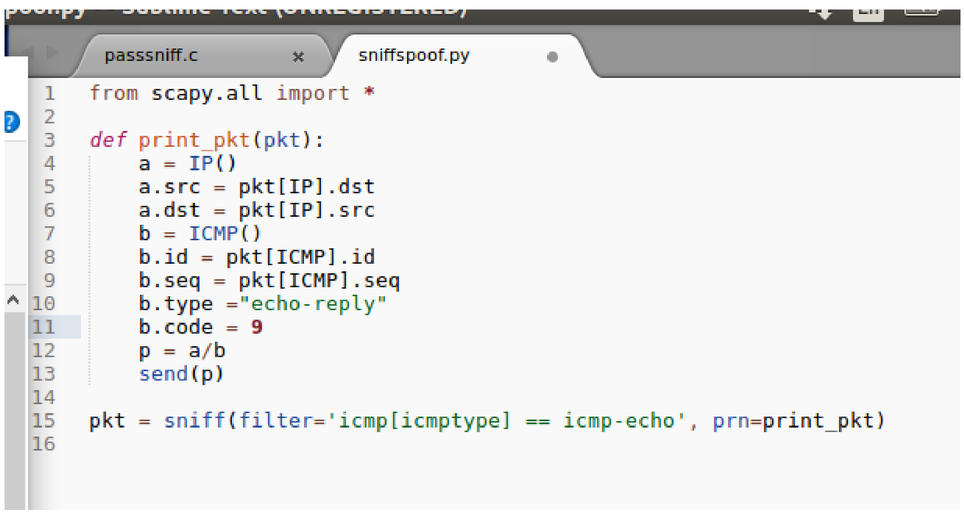
Following this I updated my code to a ttl of 4, to try one more hop. This seems to have actually been the correct number of hops, making my job much easier for this task.



As you will see, the request was echoed at a ttl of 4, meaning that this is the ultimate destination our ping request arrives at. There was no response from the destination, but that makes sense as we are simply pinging them, attempting to reach their destination. I'm not sure what is really at 1.2.3.4 to be honest, but this shows it is fairly close.

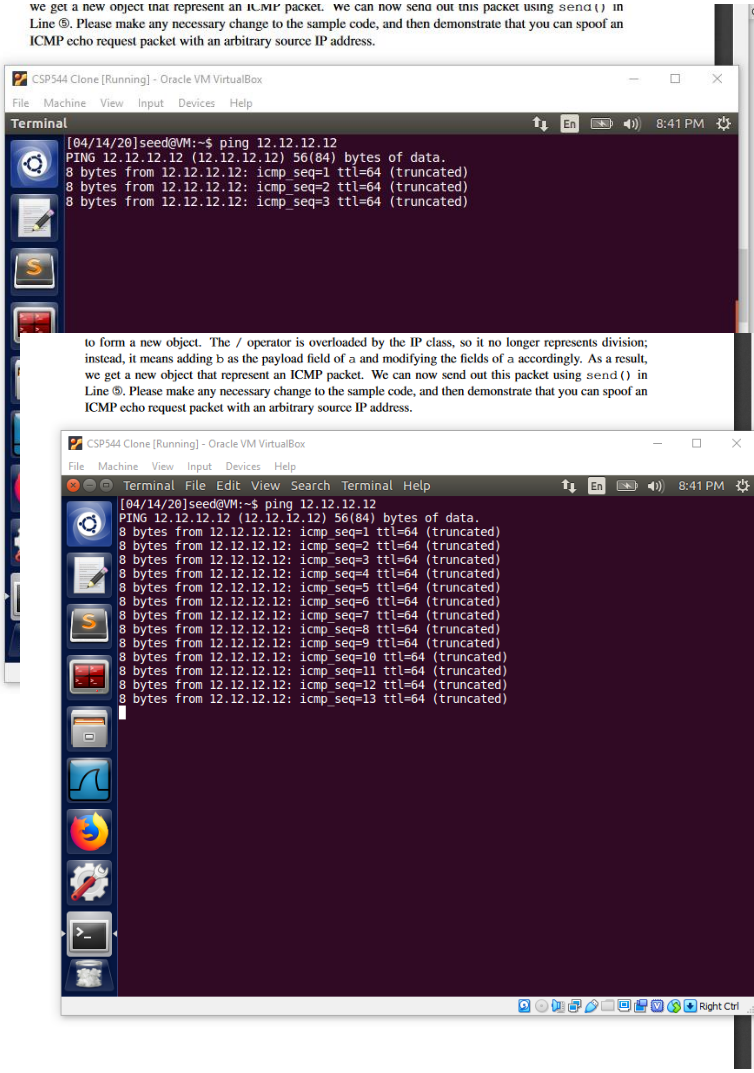


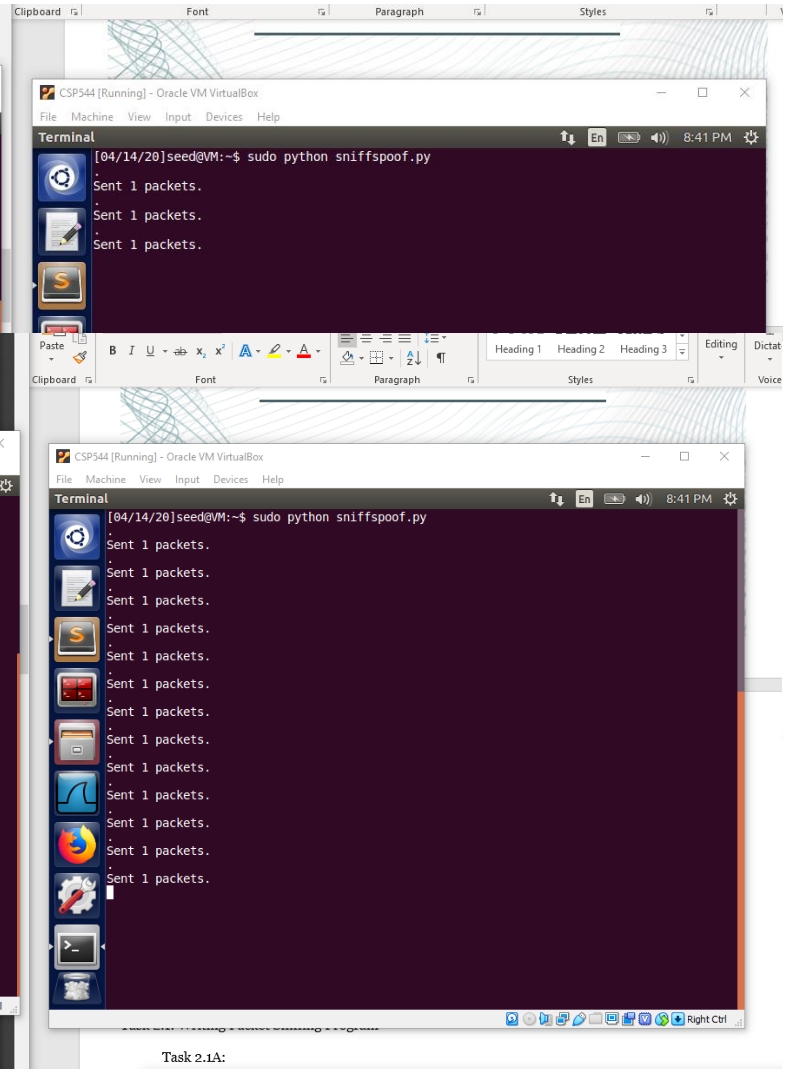
Task 1.4: Sniffing and-then Spoofing

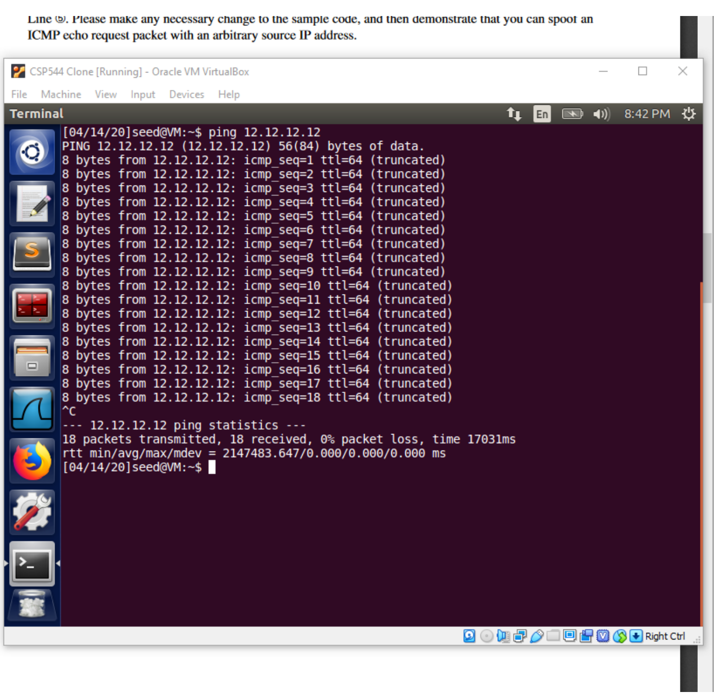


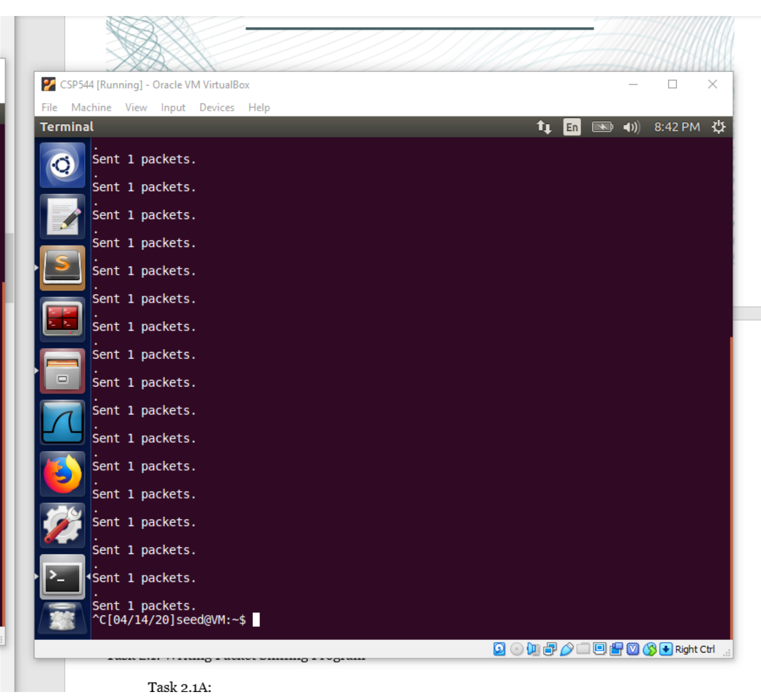
My code is shown above. What we are attempting to do with this code is to sniff out a packet that is being sent, flip its destination and source address, and send that new packet out as if it is in response from the destination server. You can see above that the original id and seq of each packet is maintained the same, its just the destination and source that are flipping.

I set this up being two VMs as described in the lab document, and tested the code, generating a 100% success rate. This is shown in action on the following page.


we get a new object that represent an ICMP packet. We can now send out this packet using send() in Line ⑤. Please make any necessary change to the sample code, and then demonstrate that you can spoof an ICMP echo request packet with an arbitrary source IP address.
CSP544 Clone [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal 8:41 PM
[04/14/20]seed@VM:~$ ping 12.12.12.12
PING 12.12.12.12 (12.12.12.12) 56(84) bytes of data.
8 bytes from 12.12.12.12: icmp_seq=1 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=2 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=3 ttl=64 (truncated)
to form a new object. The / operator is overloaded by the IP class, so it no longer represents division; instead, it means adding b as the payload field of a and modifying the fields of a accordingly. As a result, we get a new object that represent an ICMP packet. We can now send out this packet using send() in Line ⑤. Please make any necessary change to the sample code, and then demonstrate that you can spoof an ICMP echo request packet with an arbitrary source IP address.
CSP544 Clone [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal File Edit View Search Terminal Help 8:41 PM
[04/14/20]seed@VM:~$ ping 12.12.12.12
PING 12.12.12.12 (12.12.12.12) 56(84) bytes of data.
8 bytes from 12.12.12.12: icmp_seq=1 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=2 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=3 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=4 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=5 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=6 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=7 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=8 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=9 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=10 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=11 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=12 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=13 ttl=64 (truncated)
Right Ctrl


Clipboard Font Paragraph Styles
CSP544 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal 8:41 PM
[04/14/20]seed@VM:~$ sudo python sniffspoof.py
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Paste Heading 1 Heading 2 Heading 3 Editing Dictat
Clipboard Font Paragraph Styles Voice
CSP544 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal 8:41 PM
[04/14/20]seed@VM:~$ sudo python sniffspoof.py
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Right Ctrl
Task 2.1A:


Line ⑤. Please make any necessary change to the sample code, and then demonstrate that you can spoof an ICMP echo request packet with an arbitrary source IP address.
CSP544 Clone [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal 8:42 PM
[04/14/20]seed@VM:~$ ping 12.12.12.12
PING 12.12.12.12 (12.12.12.12) 56(84) bytes of data.
8 bytes from 12.12.12.12: icmp_seq=1 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=2 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=3 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=4 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=5 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=6 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=7 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=8 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=9 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=10 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=11 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=12 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=13 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=14 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=15 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=16 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=17 ttl=64 (truncated)
8 bytes from 12.12.12.12: icmp_seq=18 ttl=64 (truncated)
^C
--- 12.12.12.12 ping statistics ---
18 packets transmitted, 18 received, 0% packet loss, time 17031ms
rtt min/avg/max/mdev = 2147483.647/0.000/0.000/0.000 ms
[04/14/20]seed@VM:~$
Right Ctrl


CSP544 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal 8:42 PM
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
Sent 1 packets.
^C[04/14/20]seed@VM:~$
Right Ctrl
Task 2.1A:

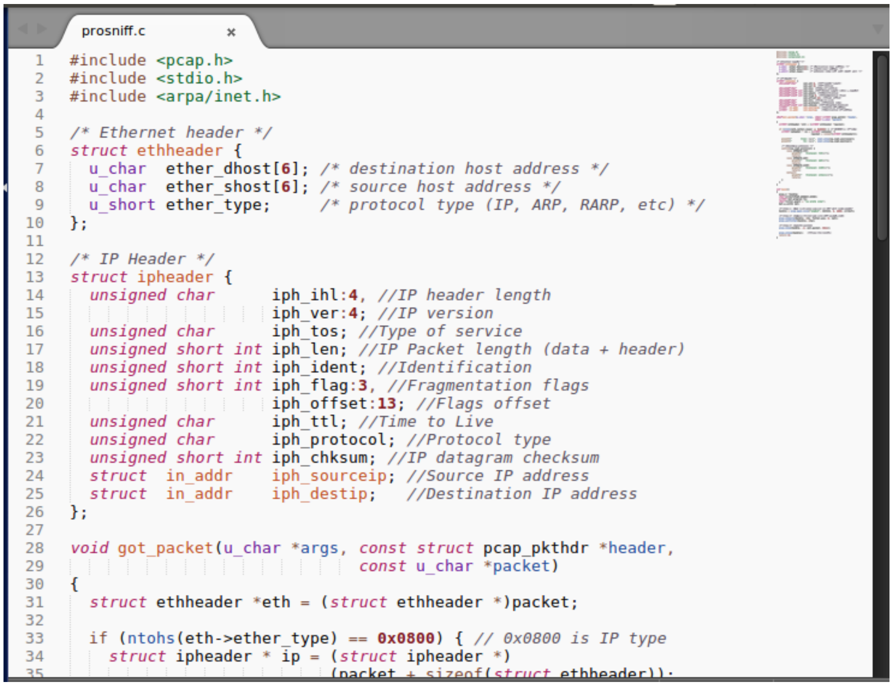
Lab Task Set 2: Writing Programs to Sniff and Spoof Packets

Task 2.1: Writing Packet Sniffing Program

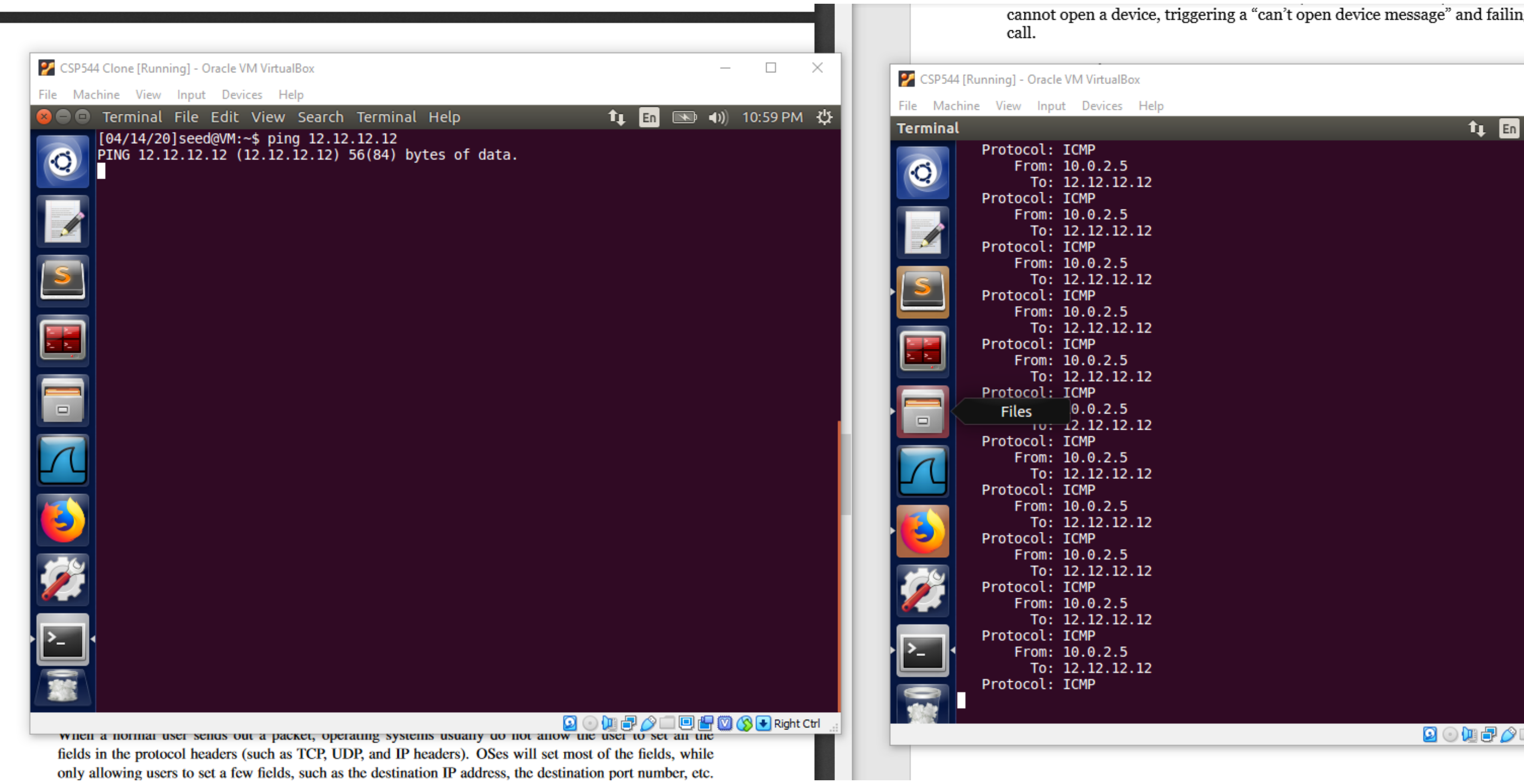
Task 2.1A:

I obtained my code for this section from the official textbook github, which is also located in the textbook. I decided to use sniff_improved.c as it seemed like a much more robust implementation. As the lab document said using provided sample code by the author is a-okay, this seemed like a solid way to do things.

As you can use below, the code worked just fine. I set one VM containing SEED to ping 12.12.12.12, and used the other to sniff the packets out successfully.




```c
#include <pcap.h>
#include <stdio.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));
```



This code will be manipulated as we go on to allow me to progress with these tasks.

Question 1.

In general, packet sniffing programs follow a standard sequence of library calls to gather information. This begins with "Setting the device", where we designate a BPF device to bring in the information. Next we "Open for sniffing", where we configure the device so that it is open to data traffic. We then use this open port to intake packets on the network, the "actual sniffing", and finally we "filter traffic" so that we are shown only the packets that are relevant to our (malicious) interests. Without filtering traffic, we might be overwhelmed, and without a device we could not listen, and therefore not sniff at all.

Question 2.

You need root privilege because sniffer programs invoke the systems API, which requires root privilege to access. You need to invoke the system API to access any devices on the network, as dangerous agents could use access information for dangerous purposes.

When invoked without root access, as shown in Task 1, we will fail when the system cannot open a device, triggering a "operation not permitted" message and failing at the first library call.

Question 3.

With promiscuous mode on, every broadcast packet on the local network will be received, even if their intended destination was not us. This is because the local IP was set as a target, but not necessarily a specific subnet. As a result I can see the broadcast packets even while just on ethernet.

Task 2.1B:

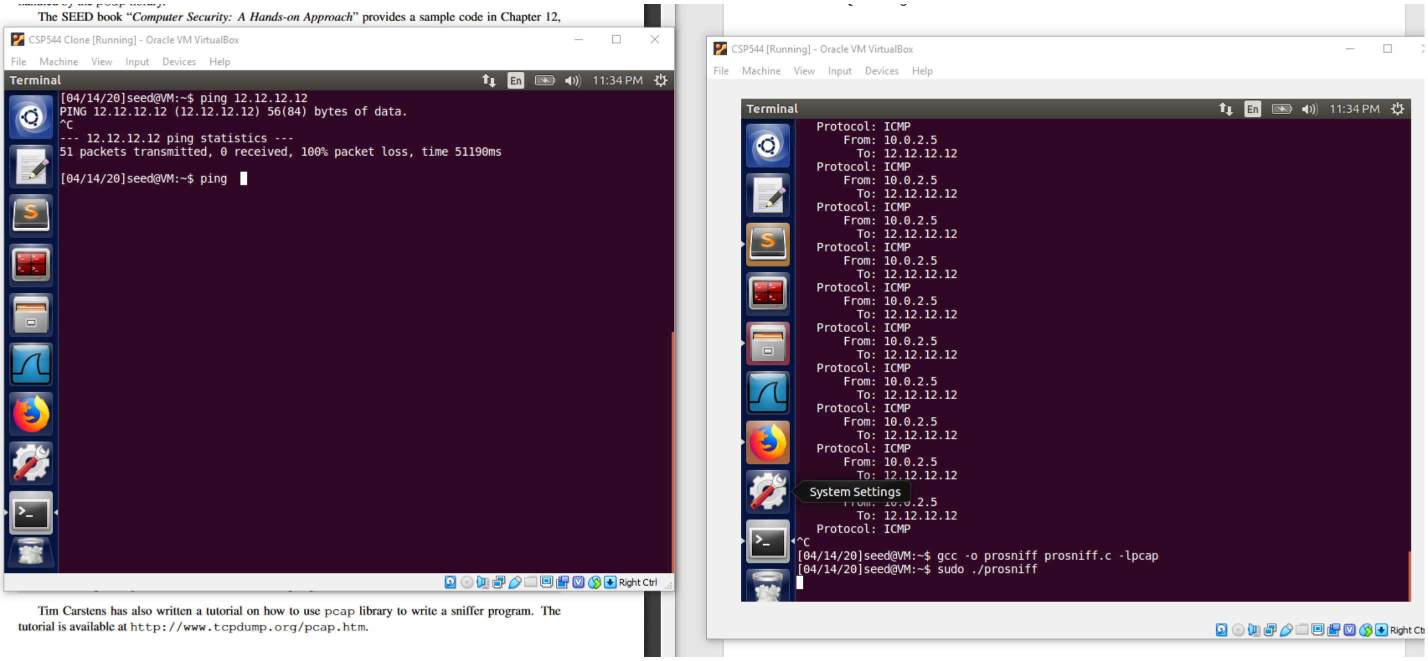
a. Capture the ICMP packets between two specific hosts

```
}

int main()
{
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;
  char filter_exp[] = "ip proto 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);

  // 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;
}
```

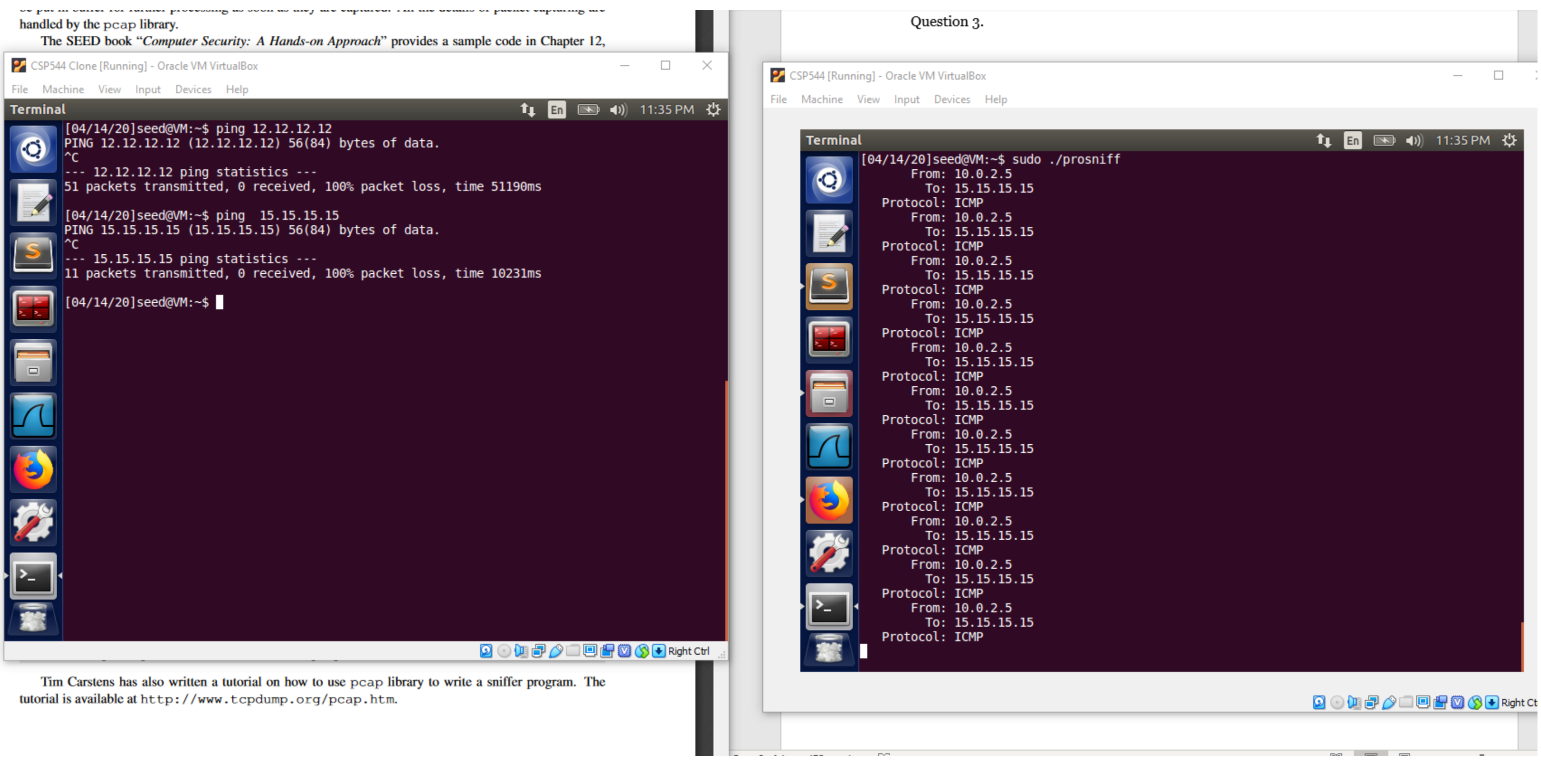
```
t main()

pcap_t *handle;
char errbuf[PCAP_ERRBUF_SIZE];
struct bpf_program fp;
char filter_exp[] = "icmp and (src host 10.0.2.5 and dst host 15.15.15.15)
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);
pcap_setfilter(handle, &fp);

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

pcap_close(handle);   //Close the handle
return 0;
```

I realized through observing the code that I could use the filter_exp array to pass pcap filters to the sniffer function. As requested I set the format to ICMP and set the source and destination addresses to specific hosts, with 10.0.2.5 being the address of my other VM in this simulation.

After running the code, I captured the resulting captures, which appeared to work successfully. I've printed these below.

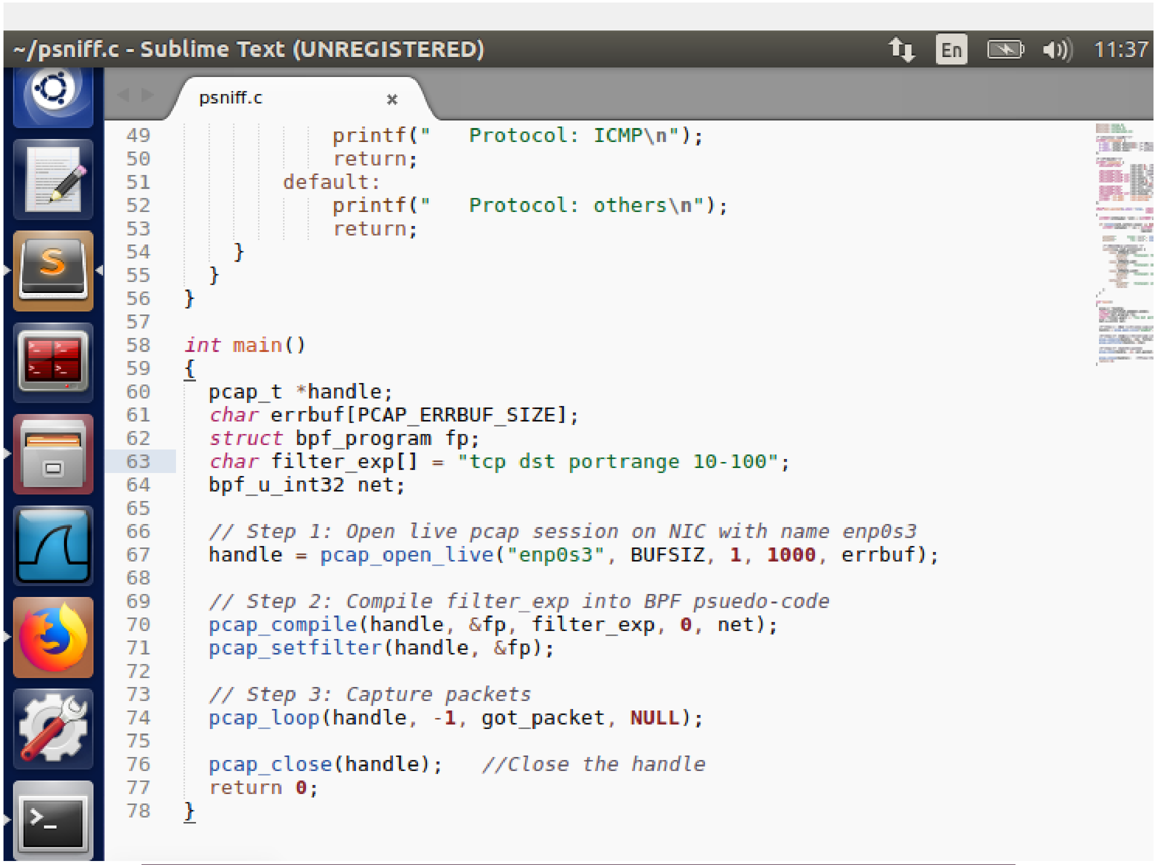


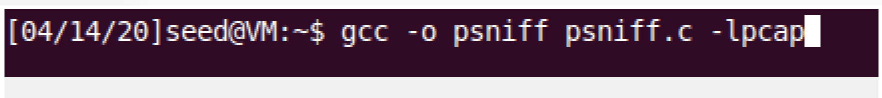
Above you can see above, if I attempt to ping 12.12.12.12, the sniffer doesn't receive me as the destination address doesn't match. However, if I ping 15.15.15.15, it successfully parses and prints those packets!



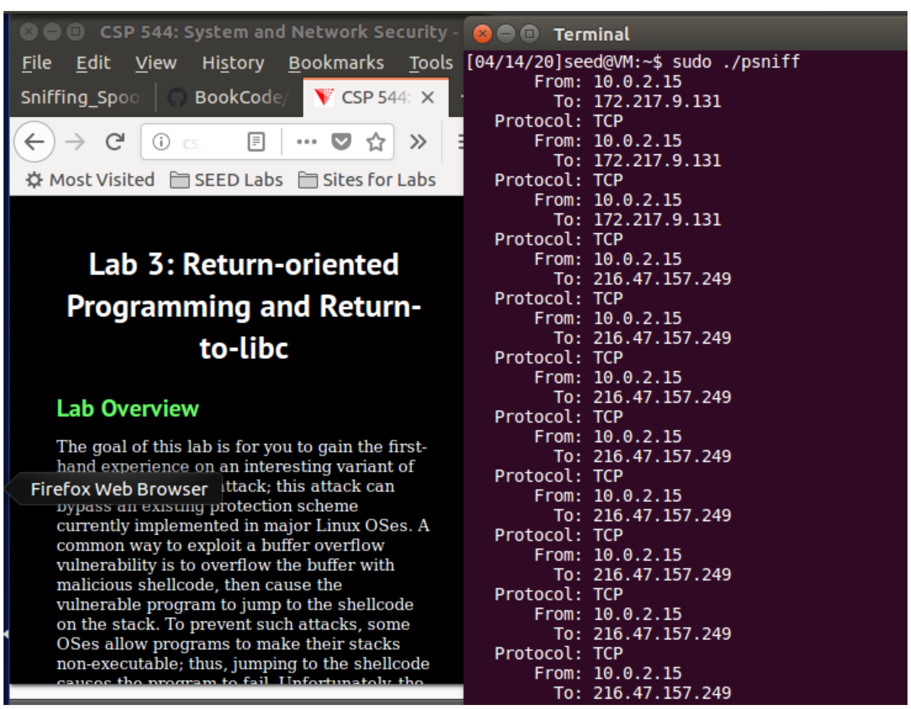
b. Capture the TCP packets with a destination port number in the range from 10 to 100

As before, I use the improved sniffer code from the lab document for my base. This time I alter filter_exp[] to indicate 2 main things: we want to capture TCP packets, and their destination port range should be 10 to 100. I've implemented these in filter code.





To test that the sniffer was running, I decided to go to an HTTP based website, as I know that HTTP data is sent over port 80, which falls within our 10-100 range. Specifically, I decided to use our course website, to sniff out some learning opportunities I guess.



As you can clearly see, the sniff and spoof program is working correctly!

Task 2.1C:

I had significant trouble with this one, particularly in deciphering the password.

While I have no trouble capturing packets sent over telnet, I'm having a lot of trouble getting the specific data location to print. To prove that I did try, however, I'll show some of my work.

I first found out that telnet is TCP and goes through port 23, so I made those adjustments under filter_exp[].

```c
int main()
{
  pcap_t *handle;
  char errbuf[PCAP_ERRBUF_SIZE];
  struct bpf_program fp;
  char filter_exp[] = "tcp 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);
  pcap_setfilter(handle, &fp);

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

  pcap_close(handle);   //Close the handle
  return 0;
}
```

I then logged into telnet, into the user account seed:dees.

```
[04/18/20]seed@VM:~$ telnet 10.0.2.5
Trying 10.0.2.5...
Connected to 10.0.2.5.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Tue Apr 14 23:50:08 EDT 2020 from 10.0.2.5 on pts/6
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

1 package can be updated.
0 updates are security updates.

[04/18/20]seed@VM:~$
```

I then checked on my sniffing program, which had indeed gotten the payload data. The sheer quantity of payload data, however, caused me to give up looking, though you can see if is clearly printing it (even certain notifications are shown).

```
.....:.......H`..G.P
00048   17 40 00 40 06 04 8a 0a  00 02 0f 0a 00 02 05 97 be 00 17 1b f6 8f b2 55 59 7f 58
80 10 00 ed 18 3a 00 00 01 01 08 0a 00 48 60 e7 00 47 04 50    .@.@..................UY.X
.....:.......H`..G.P
00064   17 40 00 40 06 04 8a 0a  00 02 0f 0a 00 02 05 97 be 00 17 1b f6 8f b2 55 59 7f 58
80 10 00 ed 18 3a 00 00 01 01 08 0a 00 48 60 e7 00 47 04 50    .@.@..................UY.X
.....:.......H`..G.P
       From: 10.0.2.5
         To: 10.0.2.15
   Protocol: TCP
00000   2a 40 00 40 06 d6 53 0a  00 02 05 0a 00 02 0f 00 17 97 be 55 59 7f 58 1b f6 8f b2
80 18 00 e3 b9 17 00 00 01 01 08 0a 00 47 04 50 00 48 60 e7 0d 0a 45 73 63 61 70 65 20 63
68 61 72 61 63 74 65 72 20 69 73 20 27 5e 5d 27 2e 0d 0a ff fa 21 00 ff f0    *@.@..S.....
.......UY.X................G.P.H`...Escape character is '^]'.....!...
00016   2a 40 00 40 06 d6 53 0a  00 02 05 0a 00 02 0f 00 17 97 be 55 59 7f 58 1b f6 8f b2
80 18 00 e3 b9 17 00 00 01 01 08 0a 00 47 04 50 00 48 60 e7 0d 0a 45 73 63 61 70 65 20 63
```

More code:

```
    //Find protocol
    switch(ip->iph_protocol) {
        case IPPROTO_TCP:
            printf("   Protocol: TCP\n");
            break;
        case IPPROTO_UDP:
            printf("   Protocol: UDP\n");
            return;
        case IPPROTO_ICMP:
            printf("   Protocol: ICMP\n");
            return;
        default:
            printf("   Protocol: others\n");
            return;
    }
    // Find offset
  struct tcpheader * tcp = (struct tcpheader
  size_tcp = TH_OFF(tcp)*4;

  payload = (u_char *)(packet + sizeof(struc
  size_payload = ntohs(ip->iph_len)- (size_i

  print_hex(payload, size_payload, 0);
  print_hex(payload, size_payload, 16);
  print_hex(payload, size_payload, 32);
  print_hex(payload, size_payload, 48);
  print_hex(payload, size_payload, 64);
  }
}
```

Task 2.2: Spoofing

Task 2.2A:

At first I found these next few tasks very difficult, as I realized I would need to implement a checksum and another validation measures. However, I eventually realized the book had some example code to implement the skeletons of these, which made things significantly easier to understand.

2.2A and B are essentially the same, as I just decided to try and construct an echo request packet program from the get-go, knowing that I would ultimately need to discover it either way. I figured this would be okay, as task A technically just says write any packet spoofing program.

Based on examples in the book and online, I constructed 3 main files for our spoofing program:
a. icmp_spoofer.c : This contains the guts of the socket work, and ultimately sends the packet.
b. checksum.c : This performs the checksum for the ICMP packet.
c. ipheaders.h : This contains the general structure of our IP and ICMP headers.

I’ve shown these below. Together I compiled these to spoof ICMP packets, and shown is my success.

```c
    icmp->icmp_type = 8; //8 = request, 0 = reply


    // Calculate checksum
    icmp->icmp_check = 0;
    icmp->icmp_check = checksum((unsigned short *)icmp,
                                sizeof(struct icmpheader));

    //We must have an ip header for each packet
    struct ipheader *ip = (struct ipheader *) buffer;

    ip->iph_ihl = 5;

    ip->iph_ver = 4;

    ip->iph_ttl = 90;

    ip->iph_destip.s_addr = inet_addr("8.8.8.8");

    ip->iph_sourceip.s_addr = inet_addr("15.15.15.15");

    ip->iph_protocol = IPPROTO_ICMP;
    ip->iph_len = htons(sizeof(struct ipheader) +
                        sizeof(struct icmpheader));
//This sends the packet
    send_ip_packet (ip);

    return 0;
}
```

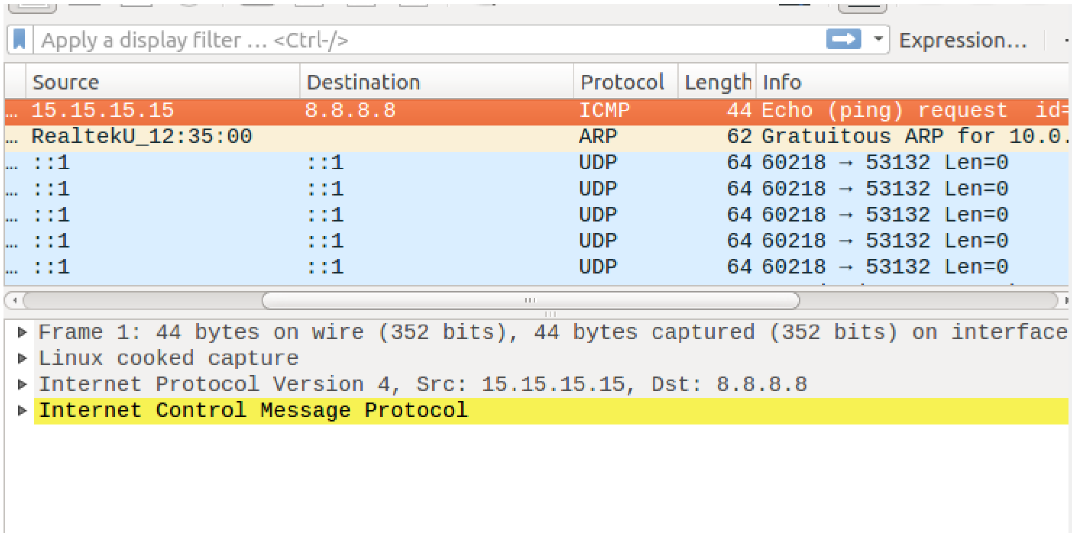
checksum.c × icmp_spoofer.c × ipheaders.h ×

```c

//IP Header
struct ipheader {
  unsigned char      iph_ihl:4, //IP header length
                     iph_ver:4; //IP version

  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_check; //IP datagram checksum
  unsigned char      iph_tos; //Type of service
  unsigned short int iph_len; //IP Packet length (data + header)
  unsigned short int iph_ident; //Identification

  struct  in_addr    iph_sourceip; //Source IP address
  struct  in_addr    iph_destip;   //Destination IP address
};

//ICMP Header
struct icmpheader {

  unsigned short int icmp_check; //Checksum for ICMP Header and data
  unsigned short int icmp_id;     //Used for identifying request
  unsigned short int icmp_seq;    //Sequence number

  unsigned char icmp_type; // ICMP message type
  unsigned char icmp_code; // Error code
};
```

```c
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>

#include <ipheaders.h>


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

   /*

   Inspired by the book, this is a 32 bit accumulator
   utilizing 16 bit words. Ultimately it folds the carry bits
   from the upper to the lower bits respectively.
   This allows us to perform an ICMP checksum.

    */
   while (nleft > 1)  {
       sum += *w++;
       nleft -= 2;
   }
   if (nleft == 1) {
        *(u_char *)(&temp) = *(u_char *)w ;
        sum += temp;
   }
```

I will go into more detail on the checksum under 2.2B. But you can see the IP header structure, which will be used in icmp_spoofer.c to create the spoof.

```
[04/18/20]seed@VM:~$ gcc -o icmp_spoofer icmp_spoofer.c checksum.c
[04/18/20]seed@VM:~$ sudo ./icmp_spoofer
[04/18/20]seed@VM:~$ 
```

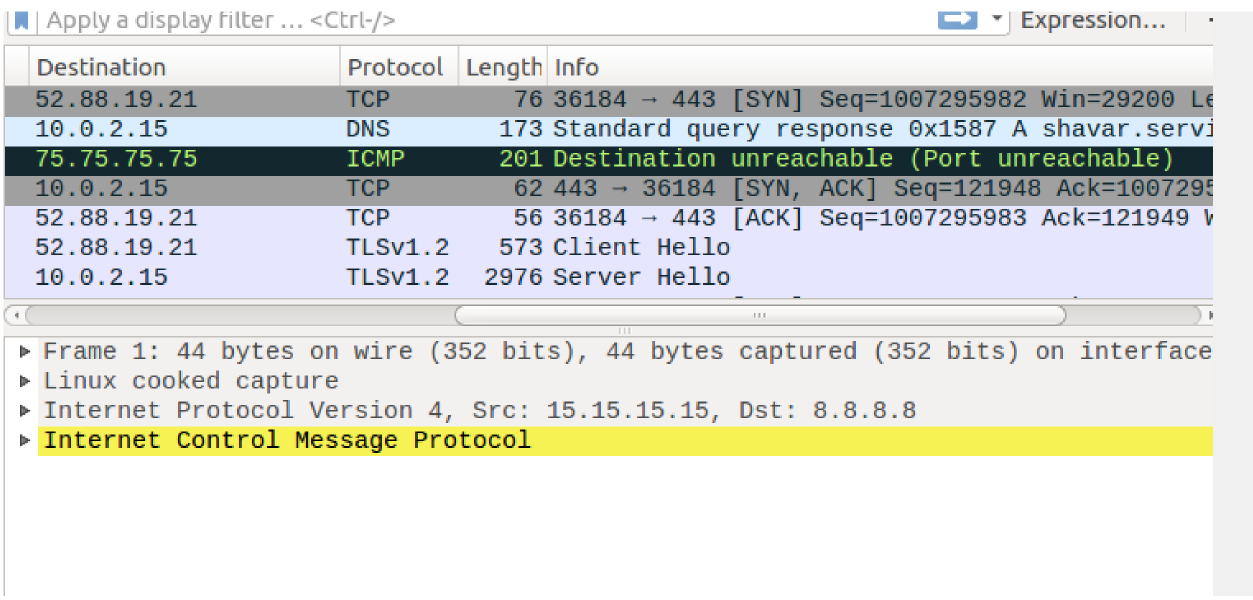


It was sent! Spoofed with a false address (as you will remember from earlier, mine is 10.0.5.2)

Task 2.2B:

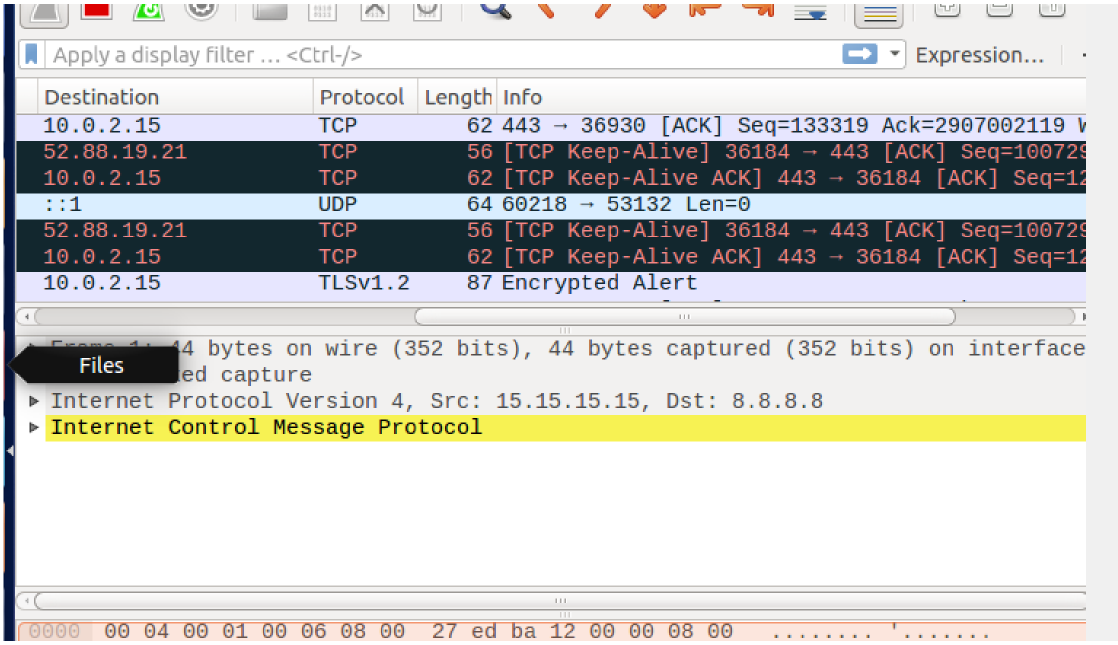
As I'm sure you have noticed, my message above used ICMP protocol, and is sent as a request. This demonstrates that I have correctly sent an ICMP Echo Request packet via my program. But to make sure, I must get an echo reply yes?

One thing I forgot to mention is that 8.8.8.8, the destination of the spoof, is Google's DNS test servers. So they are configured to respond to ICMP spoofing in a rather unique way.



You can see I get a "port unreachable" ICMP response, and am matched as a client up to some random server. This fascinated me so I thought I would share.

Ultimately, however, I ended up getting hit by a bunch of encrypted messages and errors, so I decided to exercise caution and close the connection.



More on the checksum: The checksum uses an algorithm I found in the book, which uses 16 bit words in a 32 bit space. This allows us to perform the ICMP checksum, which unlike the IP header is not performed automatically.

Question 4.

You cannot. Attempting to do so will cause an invalid argument error. This is because the function requires the length to be the length of the IP Packet.

Question 5.

No, you do not. Assuming your programming is implemented correctly, the system should automatic calculate it for you.

Question 6.

This is for very similar reasons to question 3. Raw sockets allow you to create communication systems on the network, just like we have done in this lab. Because of this, allowing any user to utilize it is a massive safety hazard.

The system fails at step 1 when it tries to create a raw network socket. The system will not allow network socket access without root.

Task 2.3: Sniff and then Spoof

For this task we will be combining our the previous programs, taking the information from the sent packet and sending back a spoof reply.

This code was very arduous, though ultimately a simple integration of my existing code worked best.

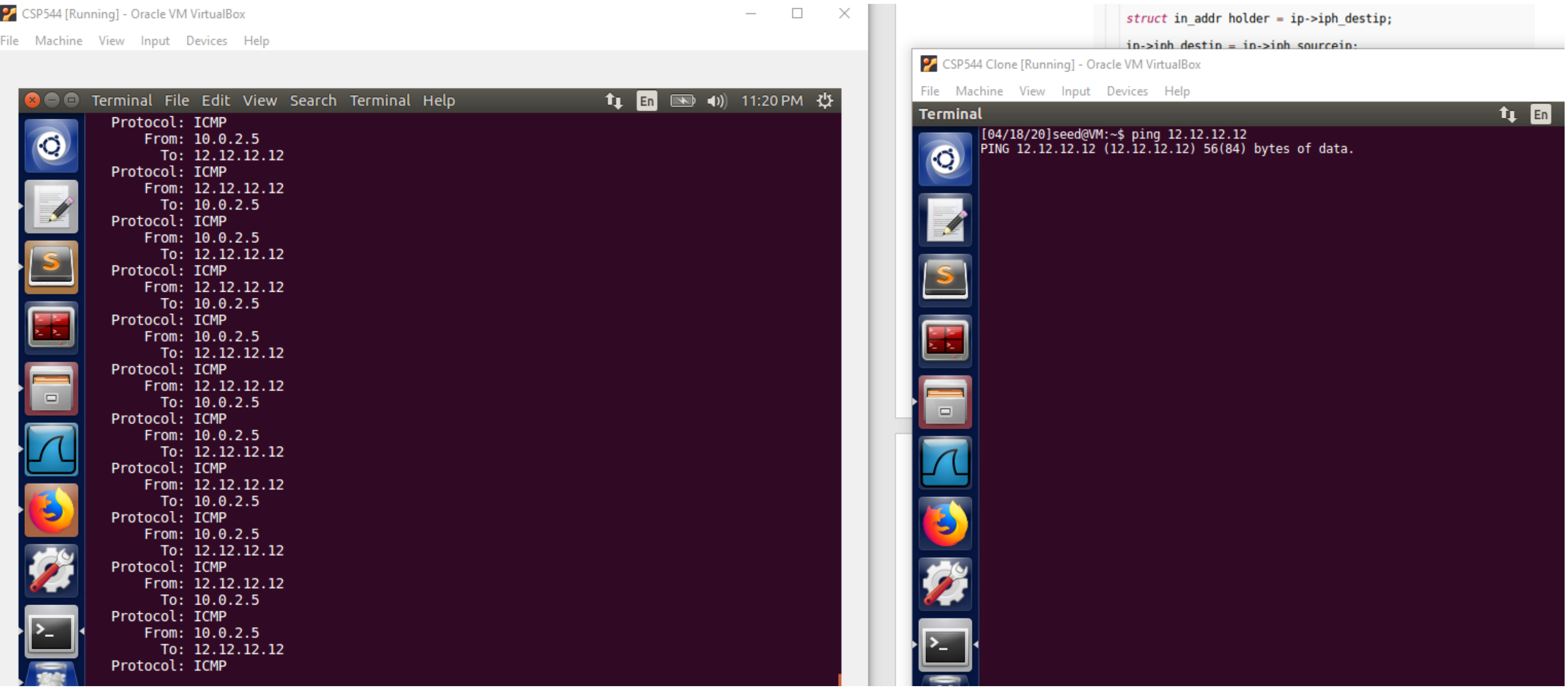
Looking below, you will see my new function, “procombo.c”, which I use to perform my sniff-and-spoof. It compiles the same as my old spoofing program, via linkages with ipheaders.h and checksum.c.

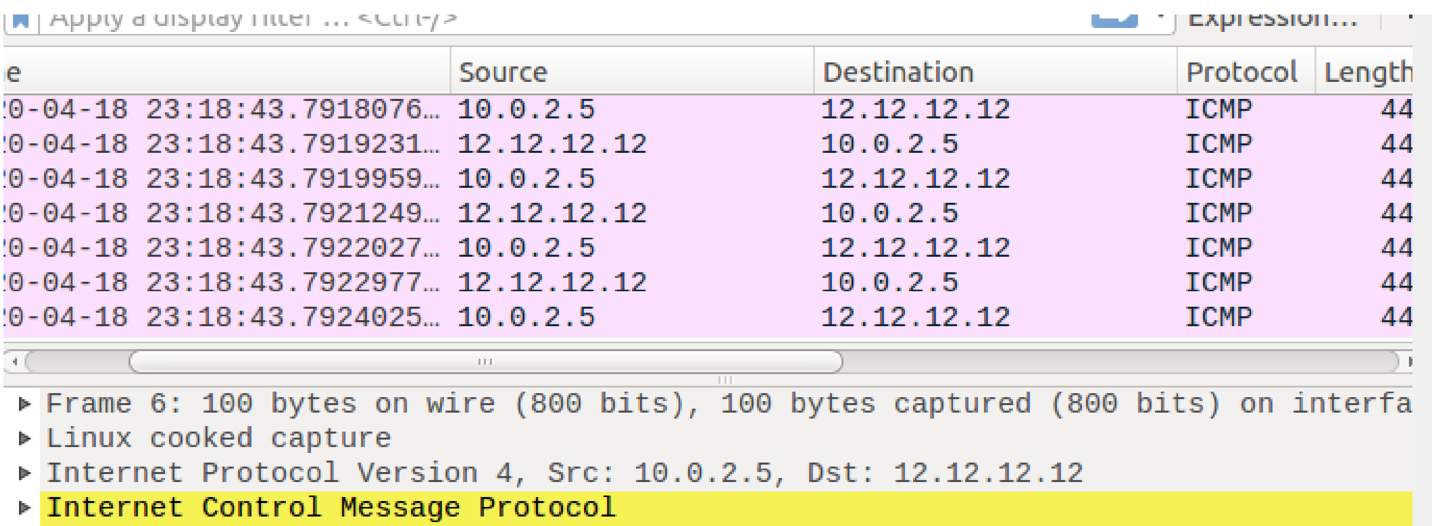
Below you can see the merged code:

```
void got_packet(u_char *args, const struct pcap_pkthdr *header,
                              const u_char *packet)
{
  struct ethheader *eth = (struct ethheader *)packet;
  char buffer[1500];

   memset(buffer, 0, 1500);

  if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
    struct ipheader * ip = (struct ipheader *)
                           (packet + sizeof(struct ethheader));

   struct icmpheader *icmp = (struct icmpheader *)
                             (buffer + sizeof(struct ipheader));

    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");
            break;
        default:
            printf("   Protocol: others\n");
            return;
    }
    icmp->icmp_type = 0; //8 = request, 0 = reply

   // Calculate checksum
   icmp->icmp_check = 0;
   icmp->icmp_check = checksum((unsigned short *)icmp,
                               sizeof(struct icmpheader));

   struct in_addr holder = ip->iph_destip;

   ip->iph_destip = ip->iph_sourceip;

   ip->iph_sourceip = holder;

   ip->iph_len = htons(sizeof(struct ipheader) +
                       sizeof(struct icmpheader));
//This sends the packet
   send_ip_packet (ip);
  }
}
```

Essentially, I have combined the old main into the “got_packet” function. The new spoof header will be exactly the same type as the packet sniffed, except the source and destinations are reversed.

This was ran in one VM while the other pinged 12.12.12.12. The results are shown below.





Boom! Success! As you can see, I have 2 VMs set up, one pinging 12.12.12.12, and the other running procombo.c. When you see Wireshark, it becomes clear that ping is getting an ICMP response, demonstrating that our program is both sniffing and spoofing correctly!