Lab 1 Report Sample

Task 1

Since Task 1 is optional, I’ve only provided the code for each of the tasks.

Task 1.1A

The Python code to sniff for a packet is shown below:

#!/usr/bin/python

from scapy.all import \*

def print\_pkt(pkt):

pkt.show()

pkt = sniff(filter='icmp',prn=print\_pkt)

Task 1.1B

The Python code to sniff for an ICMP packet is shown below:

#!/usr/bin/python

from scapy.all import \*

def print\_pkt(pkt):

pkt.show()

pkt = sniff(filter='icmp',prn=print\_pkt)

The Python code to sniff for a TCP packet that comes from a particular IP and with a destination port number 23 is shown below:

#!/usr/bin/python

from scapy.all import \*

def print\_pkt(pkt):

pkt.show()

pkt = sniff(filter='tcp port telnet and src host 10.0.2.21',prn=print\_pkt)

The Python code to sniff for packets that come from or to go to a particular subnet is shown below:

#!/usr/bin/python

from scapy.all import \*

def print\_pkt(pkt):

pkt.show()

pkt = sniff(filter='net 128.230.0.0/16',prn=print\_pkt)

Task 1.2

The Python code to spoof ICMP packets is shown below:

#!/usr/bin/python

from scapy.all import \*

a = IP()

a.src = '1.2.3.4'

a.dst = '10.0.2.21'

b = ICMP()

p = a/b

send(p)

Task 1.3

The Python code for traceroute is shown below:

#!/usr/bin/python

from scapy.all import \*

i = 1

while i <= 30:

a = IP()

a.dst = '128.230.18.198'

a.ttl = i

b = ICMP()

reply = sr1(a/b,timeout=5,verbose=0)

if reply is None:

print "%2d \*" %i

elif reply.type == 0:

print "%2d " %i, reply.src

break

else:

print "%2d " %i, reply.src

i = i + 1

Task 1.4

The Python code to sniff and spoof is shown below:

#!/usr/bin/python

from scapy.all import \*

def print\_pkt(pkt):

p = copy.deepcopy(pkt[IP])

p.src = pkt[IP].dst

p.dst = pkt[IP].src

p[ICMP].type = 0

send(p)

pkt = sniff(filter='icmp[icmptype] = 8',prn=print\_pkt)

Task 2.1A

This task showed the creation of a packet sniffing program. A program was created to sniff all IP packets. The template from the lab was used as a starting point. The PCAP filter was set to the following to capture IP packets:

char filter\_exp[] = "ip";

The code for the callback function passed to the pcap\_loop function is shown below:

void got\_packet(u\_char \*args, const struct pcap\_pkthdr \*header,

const u\_char \*packet)

{

struct ipheader \* ip = (struct ipheader \*)(packet + sizeof(struct ethheader));

unsigned short pktlen = ntohs(ip->iph\_len);

printf(" SrcIP: %s,", inet\_ntoa(ip->iph\_sourceip));

printf(" DstIP: %s,", inet\_ntoa(ip->iph\_destip));

printf(" PktLenIP: %hu\n", pktlen);

/\* determine protocol \*/

switch(ip->iph\_protocol)

{

case IPPROTO\_TCP:

printf(" Protocol: TCP\n");

break;

case IPPROTO\_UDP:

printf(" Protocol: UDP\n");

break;

case IPPROTO\_ICMP:

printf(" Protocol: ICMP\n");

break;

default:

printf(" Protocol: others\n");

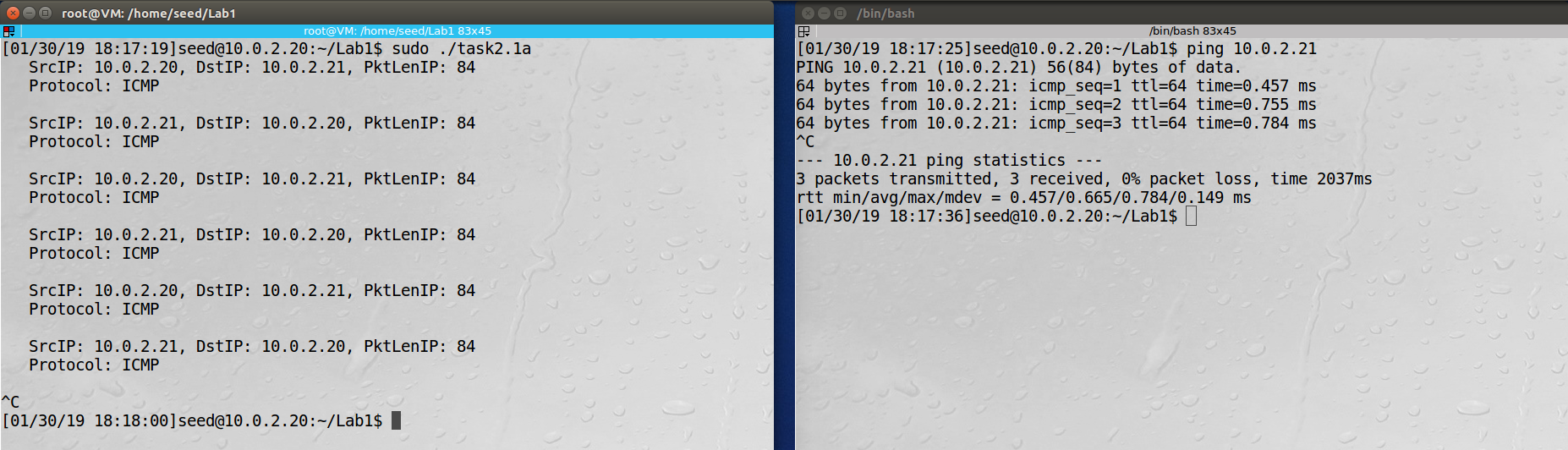
break;

}

printf("\n");

}

The pointer to the IP header was obtained by adding the size of the Ethernet header to the packet pointer. The resulting pointer was cast to a “struct ipheader” pointer and the source and destination IP addresses were printed. The IP packet length and protocol type were also printed. Note that ntohs and inet\_ntoa were used to convert from network to host byte order.



Observation: The screenshot above shows how the program captured the ICMP packets produced when the ping program was run. The sniffer program was run using the sudo command to provide root privilege.

Explanation: The PCAP library can be used to sniff for specific packets based on criteria specified in a filter. Captured packets are transferred to a callback routine that can print out details of the received packet.

Question 1

* You need to open a live session on the NIC. With the pcap library, the pcap\_open\_live function is used. It creates a raw socket, sets the device promiscuous mode based on an input parameter, and binds the socket.
* You need to create a filter that will be used to specify what packets you would like to capture. With the pcap library, the pcap\_compile and pcap\_setfilter functions are used to accomplish this step.
* You need to start capturing packets. With the pcap library, the pcap\_loop function is used to accomplish this step.
* You need to process packets that are captured. With the pcap library, the callback routine passed to the pcap\_loop function is used to process the captured packets.
* You need to close the capture device. With the pcap library, the pcap\_close function is used to accomplish this step.

Question 2

The root privilege is needed to set up a raw socket and use promiscuous mode. If the program is run without root privilege, the pcap\_open\_live function fails.

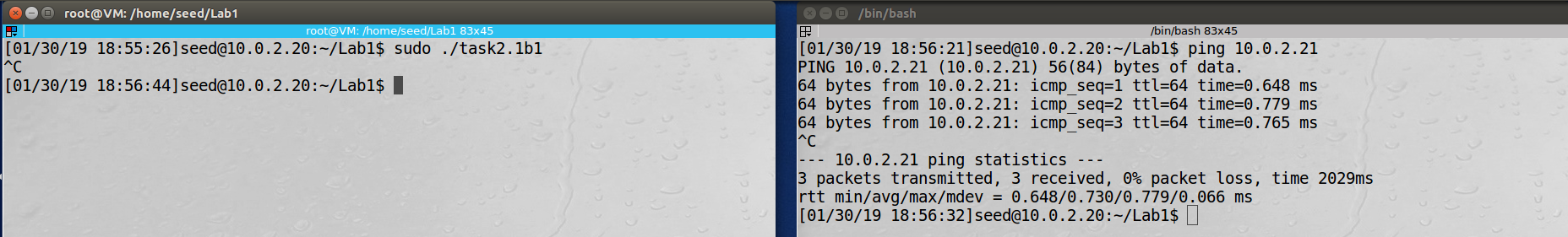
Question 3

The promiscuous mode is controlled by a parameter passed to the pcap\_open\_live function. If the promiscuous mode is turned off, the program only looks at packets that match the address of the network card. If promiscuous mode is turned on, the program looks at all packets seen by the network card. To demonstrate this, run the program on one machine and then use the ping program to send packets between two other machines. The program will only capture packets when the promiscuous mode is on.

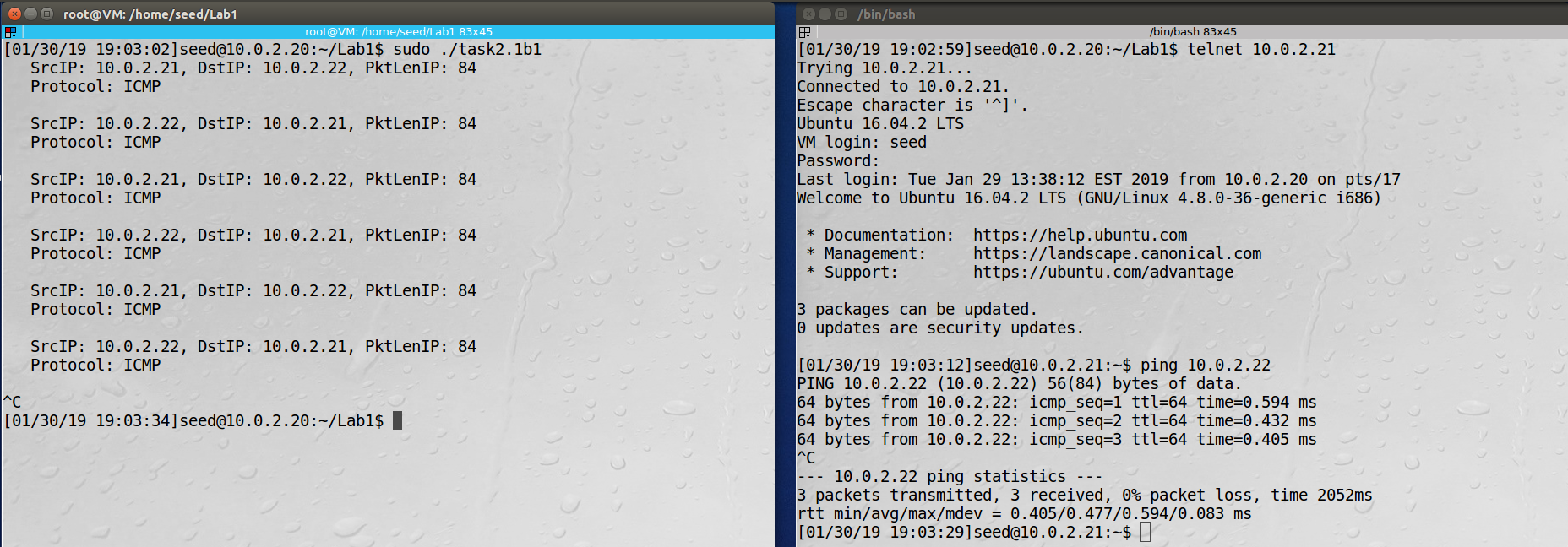
Task 2.1B

This task showed the use of different filters used to capture packets based on different criteria. The first filter was created to capture ICMP packets between the host with IP address 10.0.21 and the host with IP address 10.0.22:

char filter\_exp[] = "icmp and host 10.0.2.21 and host 10.0.2.22";



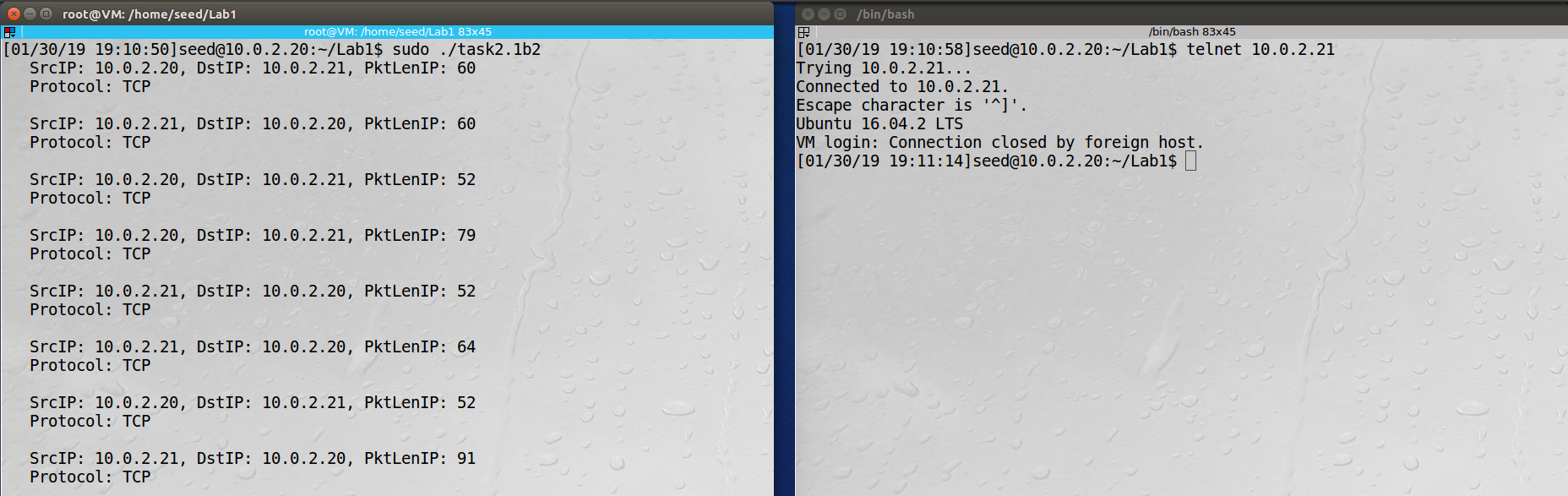
Observation: The screenshot above shows that the program did not capture packets when one of the hosts was set to 10.0.2.20.



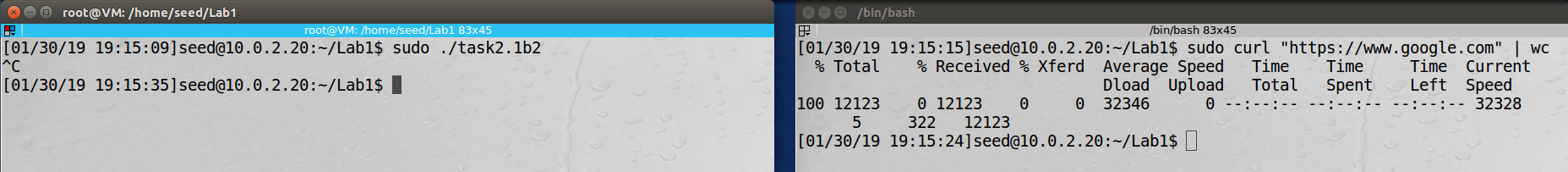
Observation: The screenshot above shows that the program was able to capture packets when the hosts were 10.0.2.21 and 10.0.22.

The second filter was created to capture TCP packets with a destination port number in the range 10 to 100:

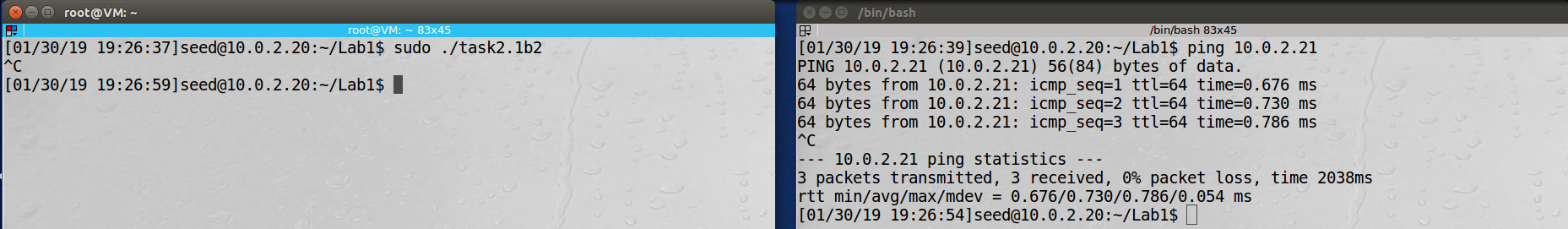
char filter\_exp[] = " tcp and dst portrange 10-100";



Observation: The screenshot above shows that the program was able to capture packets for the telnet program. The telnet program uses port 23.



Observation: The screenshot above shows that the program did not capture packets when curl was used to access a website using HTTPS. HTTPS uses port 443.



Observation: The screenshot above shows that the program did not capture ICMP packets.

Explanation: The PCAP library can be used to sniff for specific packets based on criteria specified in a filter. This task showed how different criteria were used to capture different sets of packets.

Task 2.1C

This task showed the use of the PCAP library to sniff the password used during a telnet session. The filter was set to capture telnet packets:

char filter\_exp[] = " tcp port telnet";

The code for the callback function passed to the pcap\_loop function is shown below:

void got\_packet(u\_char \*args, const struct pcap\_pkthdr \*header,

const u\_char \*packet)

{

struct ipheader \* ip = (struct ipheader \*)(packet + sizeof(struct ethheader));

unsigned short pktlen = ntohs(ip->iph\_len);

struct tcpheader \*tcp = (struct tcpheader \*)((u\_char\*)ip + sizeof(struct ipheader));

u\_char dataoffset = TH\_OFF(tcp) \* 4;

if ((pktlen - sizeof(struct ipheader)) > dataoffset)

{

printf(" SrcIP: %s,", inet\_ntoa(ip->iph\_sourceip));

printf(" DstIP: %s,", inet\_ntoa(ip->iph\_destip));

printf(" Data: ");

u\_char\* data = (u\_char\*)tcp + dataoffset;

for (unsigned short s = 0; s < (ntohs(ip->iph\_len) - (sizeof(struct ipheader) + dataoffset)); s++)

{

if (isprint(\*data) != 0)☺☺☺

{

printf("%c", \*data);

}

else

{

printf("\\%.3hho", \*data);

}

data++;

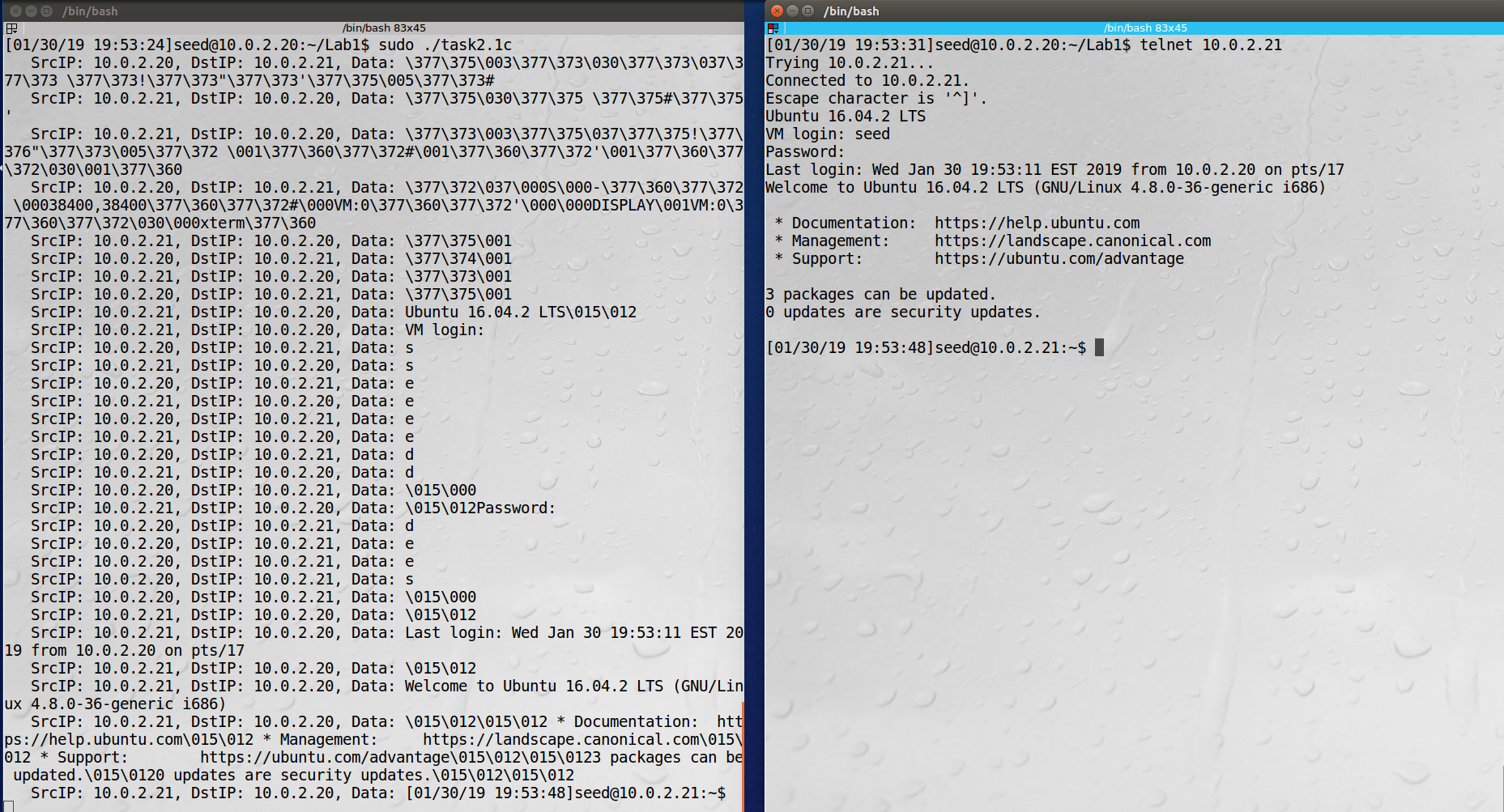
}

printf("\n");

}

}

The pointer to the IP header was obtained by adding the size of the Ethernet header to the packet pointer. The resulting pointer was cast to a “struct ipheader” pointer. The pointer to the TCP header was obtained by adding the size of the IP header to the IP header pointer. The resulting pointer was cast to a “struct tcpheader” pointer. The offset to the data section of the TCP packet was calculated. If there was any data, source and destination IP addresses were printed along with the data.



Observation: The screenshot above shows how the password was captured. The packets showing the password start after the packet containing “Password:” that is sent from 10.0.2.21 to 10.0.2.20. The characters for the password are shown in the packets from 10.0.2.20 to 10.0.2.21. The last character is in the packet before the character “\015” is sent from 10.0.2.20 to 10.0.2.21.

Explanation: The PCAP library can be used to sniff for specific packets based on criteria specified in a filter. Captured packets are transferred to a callback routine that can print out details of the received packet. This task showed how the data in a telnet packet was examined to capture the password that was used.

Task 2.2A

This task showed the ability to spoof packets. The code used to generate the spoofed packet is shown below:

int main()

{

char buffer[PACKET\_LEN];

memset(buffer, 0, PACKET\_LEN);

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; /\* Many OSes ignore this field, so we do not

calculate it. \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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("10.0.2.21");

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

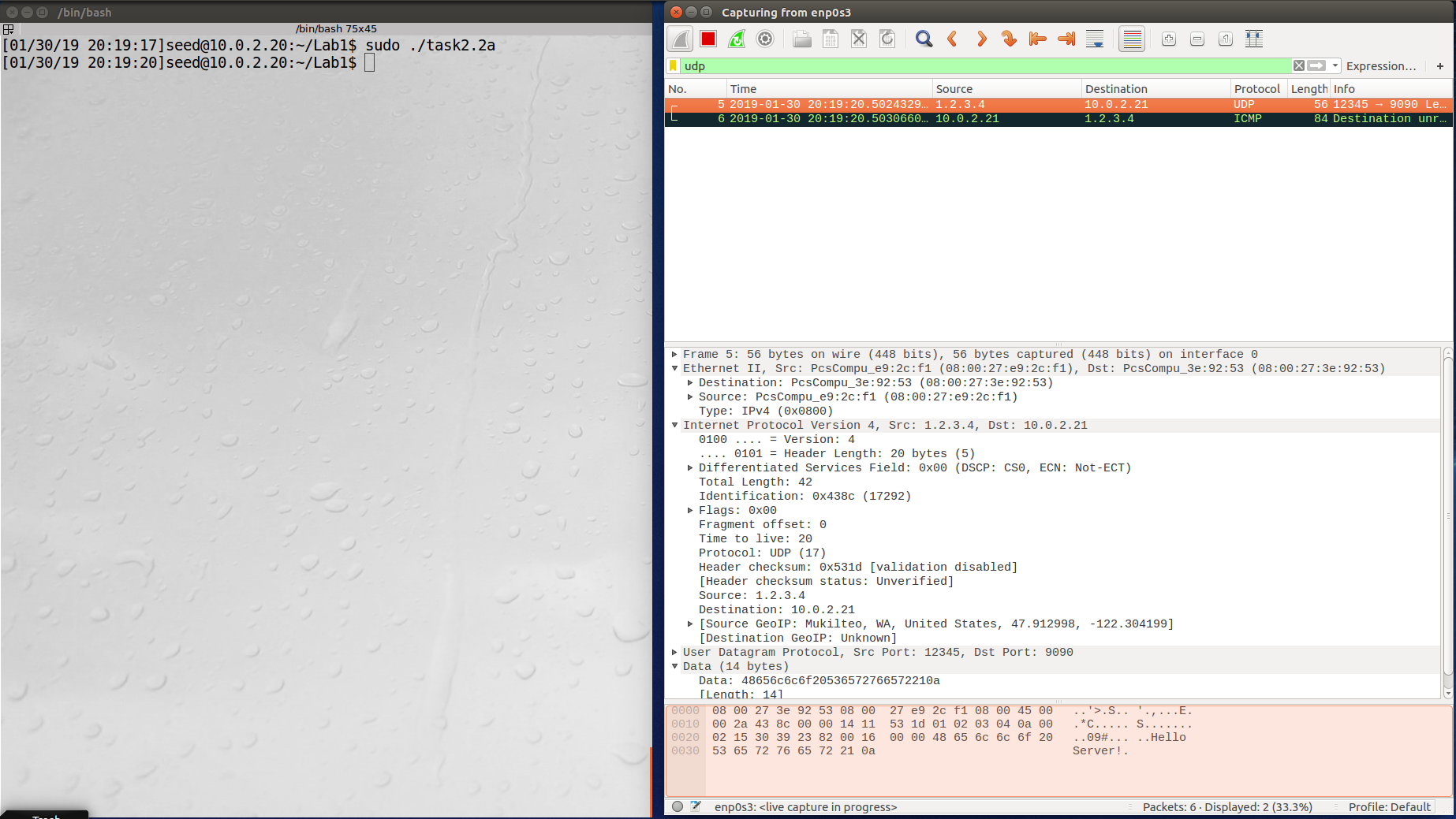
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

send\_raw\_ip\_packet (ip);

return 0;

}

A UDP packet is created with a spoofed source IP address of 1.2.3.4 and a destination IP address 10.0.2.21. The packet is sent using the send\_raw\_ip\_packet routine from the code on the class website.



Observation: The screenshot above shows the Wireshark trace when the program was run. It shows that a UDP packet was captured with a source address of 1.2.3.4, indicating the packet from the program was sent.

Explanation: Packets can be constructed and sent with spoofed IP addresses by using raw sockets. This task sent a UDP packet with a spoofed source IP address.

Task 2.2B

Tha task showed the ability to spoof an ICMP echo request. The code used to generate the spoofed packet is shown below:

int main()

{

char buffer[PACKET\_LEN];

memset(buffer, 0, PACKET\_LEN);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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("10.0.2.21");

ip->iph\_protocol = IPPROTO\_ICMP;

ip->iph\_len = htons(sizeof(struct ipheader) +

sizeof(struct icmpheader));

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Step 3: Finally, send the spoofed packet

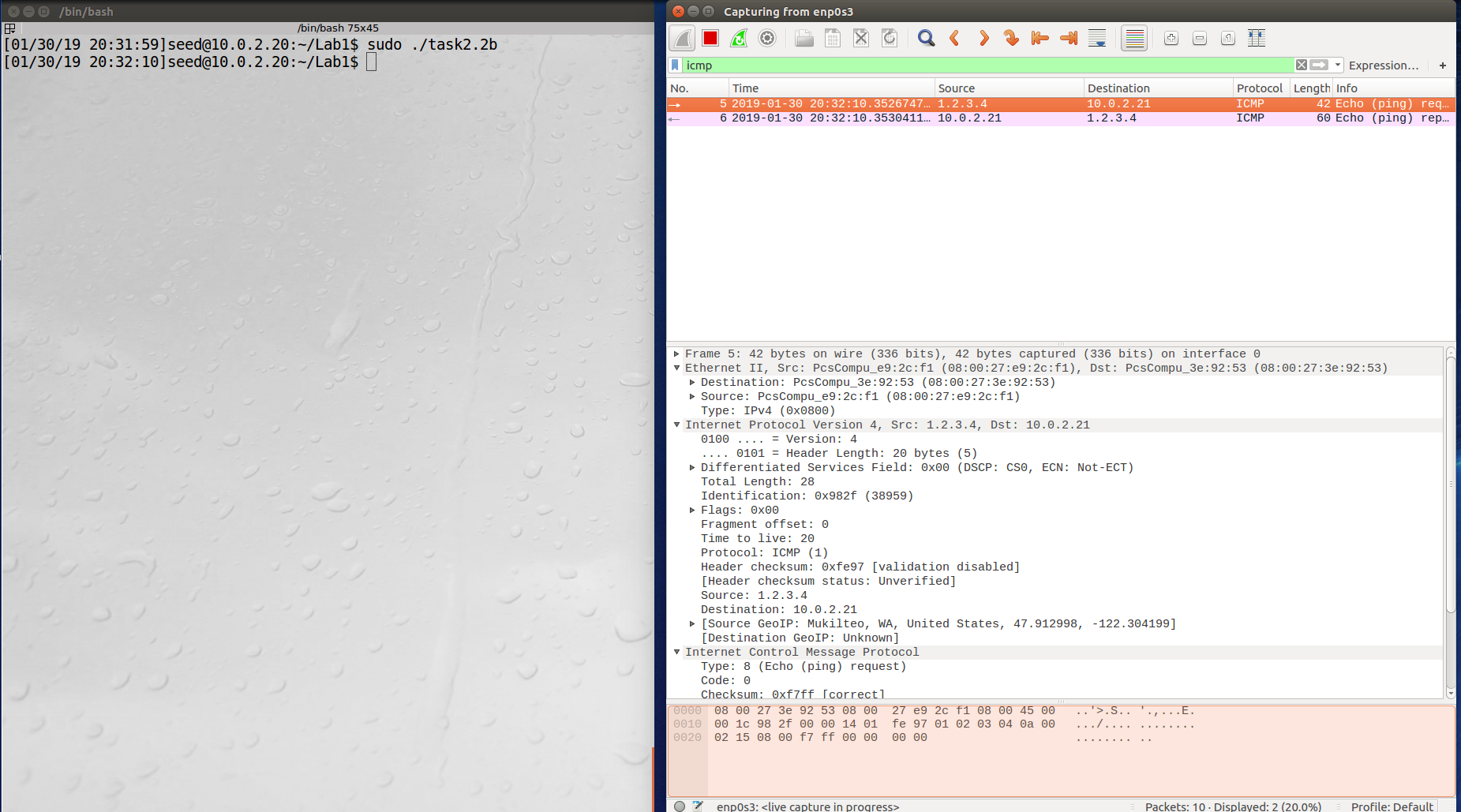
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

send\_raw\_ip\_packet (ip);

return 0;

}

An ICMP echo request packet is created with a spoofed source IP address of 1.2.3.4 and a destination IP address 10.0.2.21. The packet is sent using the send\_raw\_ip\_packet routine from the code on the class website.



Observation: The screenshot above shows the Wireshark trace when the program was run. It shows that an ICMP packet was captured with a source address of 1.2.3.4, indicating the packet from the program was sent.

Explanation: Packets can be constructed and sent with spoofed IP addresses by using raw sockets. This task sent an ICMP echo request packet with a spoofed source IP address.

Question 4

You can set the IP packet length to an arbitrary value. For example, my program that spoofs a UDP packet has a UDP data size of 14 bytes. I was able to set the IP packet length to 1000 bytes and still successfully send the packet.

Question 5

You do not need to calculate the checksum for the IP header.

Question 6

The root privilege is needed to set up a raw socket and use promiscuous mode. The program will fail on the call to the socket function.

Task 2.3

This task shows the ability to sniff and spoof. The code used to sniff and spoof is shown below:

void send\_echo\_reply(struct ipheader \* ip)

{

int ip\_header\_len = ip->iph\_ihl \* 4;

const char buffer[PACKET\_LEN];

// make a copy from original packet to buffer (faked packet)

memset((char\*)buffer, 0, PACKET\_LEN);

memcpy((char\*)buffer, ip, ntohs(ip->iph\_len));

struct ipheader\* newip = (struct ipheader\*)buffer;

struct icmpheader\* newicmp = (struct icmpheader\*)(buffer + ip\_header\_len);

// Construct IP: swap src and dest in faked ICMP packet

newip->iph\_sourceip = ip->iph\_destip;

newip->iph\_destip = ip->iph\_sourceip;

newip->iph\_ttl = 64;

// Fill in all the needed ICMP header information.

// ICMP Type: 8 is request, 0 is reply.

newicmp->icmp\_type = 0;

send\_raw\_ip\_packet (newip);

}

/\* This function will be invoked by pcap for each captured packet.

We can process each packet inside the function.

\*/

void got\_packet(u\_char \*args, const struct pcap\_pkthdr \*header,

const u\_char \*packet)

{

struct ipheader \* ip = (struct ipheader \*)(packet + sizeof(struct ethheader));

unsigned short pktlen = ntohs(ip->iph\_len);

printf(" SrcIP: %s,", inet\_ntoa(ip->iph\_sourceip));

printf(" DstIP: %s,", inet\_ntoa(ip->iph\_destip));

printf(" PktLenIP: %hu\n", pktlen);

/\* determine protocol \*/

switch(ip->iph\_protocol)

{

case IPPROTO\_TCP:

printf(" Protocol: TCP\n");

break;

case IPPROTO\_UDP:

printf(" Protocol: UDP\n");

break;

case IPPROTO\_ICMP:

printf(" Protocol: ICMP\n");

send\_echo\_reply(ip);

break;

default:

printf(" Protocol: others\n");

break;

}

printf("\n");

}

int main()

{

pcap\_t \*handle;

char errbuf[PCAP\_ERRBUF\_SIZE];

struct bpf\_program fp;

char filter\_exp[] = "icmp[icmptype] = 8";

bpf\_u\_int32 net;

// Step 1: Open live pcap session on NIC with name eth3

// Students needs to change "eth3" to the name

// found on their own machines (using ifconfig).

handle = pcap\_open\_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

if (handle == NULL)

{

printf("%s\n", errbuf);

exit(EXIT\_FAILURE);

}

// Step 2: Compile filter\_exp into BPF psuedo-code

int retvalue;

retvalue = pcap\_compile(handle, &fp, filter\_exp, 0, net);

if (retvalue == -1)

{

printf("%s\n", pcap\_geterr(handle));

exit(EXIT\_FAILURE);

}

retvalue = pcap\_setfilter(handle, &fp);

if (retvalue == -1)

{

printf("%s\n", pcap\_geterr(handle));

exit(EXIT\_FAILURE);

}

// Step 3: Capture packets

retvalue = pcap\_loop(handle, -1, got\_packet, NULL);

if (retvalue == -1)

{

printf("%s\n", pcap\_geterr(handle));

exit(EXIT\_FAILURE);

}

pcap\_close(handle); //Close the handle

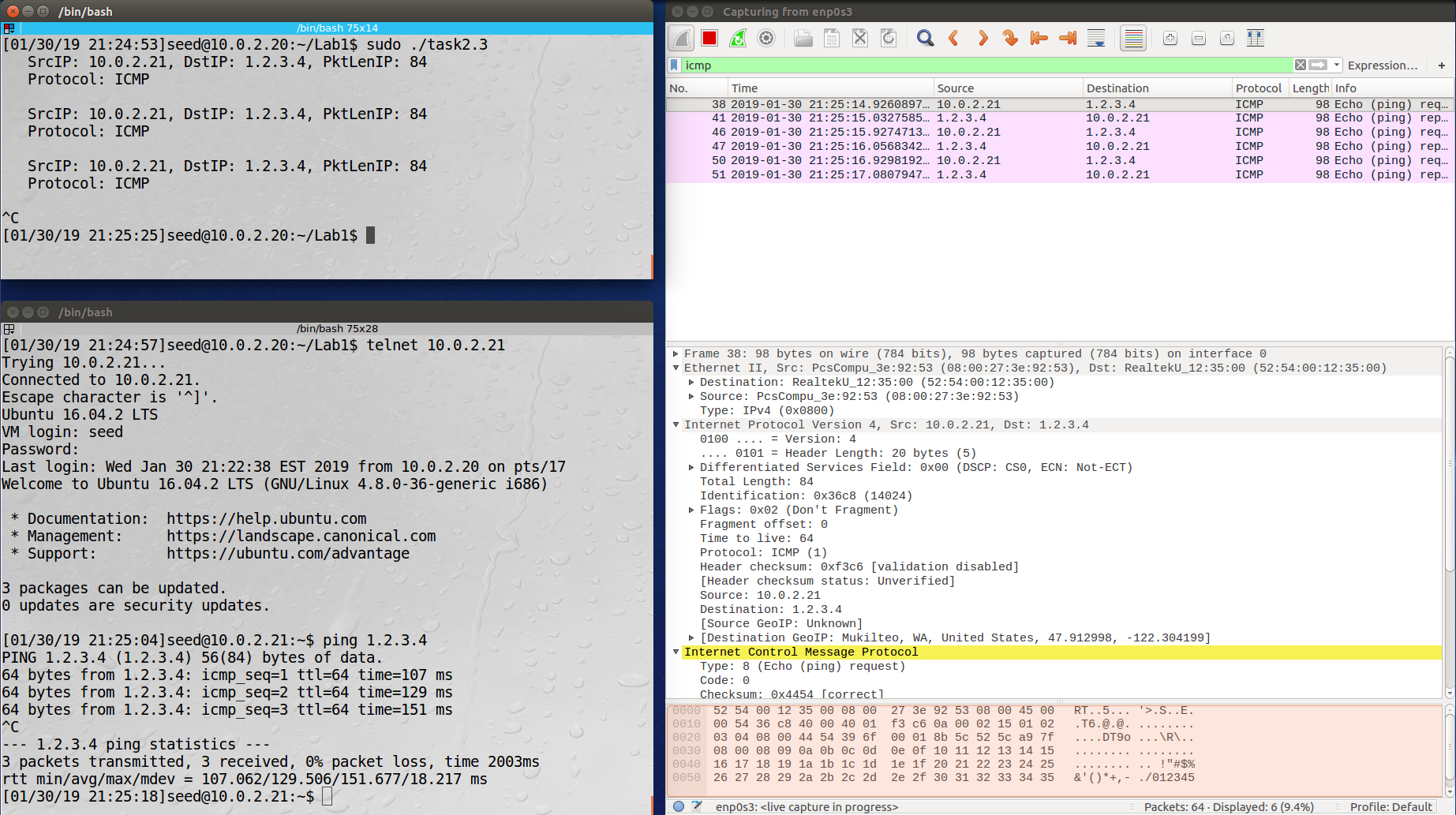
return 0;

}

The PCAP filter was set to the following to capture ICMP echo request packets:

char filter\_exp[] = "icmp[icmptype] = 8";

In the callback routine, the protocol is printed and the send\_echo\_reply function is called to send the reply. The send\_echo\_reply function copies the ICMP echo request packet and swaps the source and destination addresses in the new packet. It sets the ICMP type to 0 and sends the packet using the send\_raw\_ip\_packet routine from the code on the class website.



Observation: The screenshot above shows the program running on the machine with IP address 10.0.2.20. On the machine with IP address 10.0.2.21, an attempt is made to ping the arbitrary IP address of 1.2.3.4. The ping is successful from the point of view of the machine on IP address 10.0.2.21 as demonstrated by the received bytes. The Wireshark trace also shows a trace of the packets.

Explanation: The PCAP library can be used to sniff for specific packets based on criteria specified in a filter. Captured packets are transferred to a callback routine that can be used to spoof a packet based on the contents of the received packet. This task sent an ICMP echo reply packet with a spoofed source IP address set to the destination of the ICMP echo request.