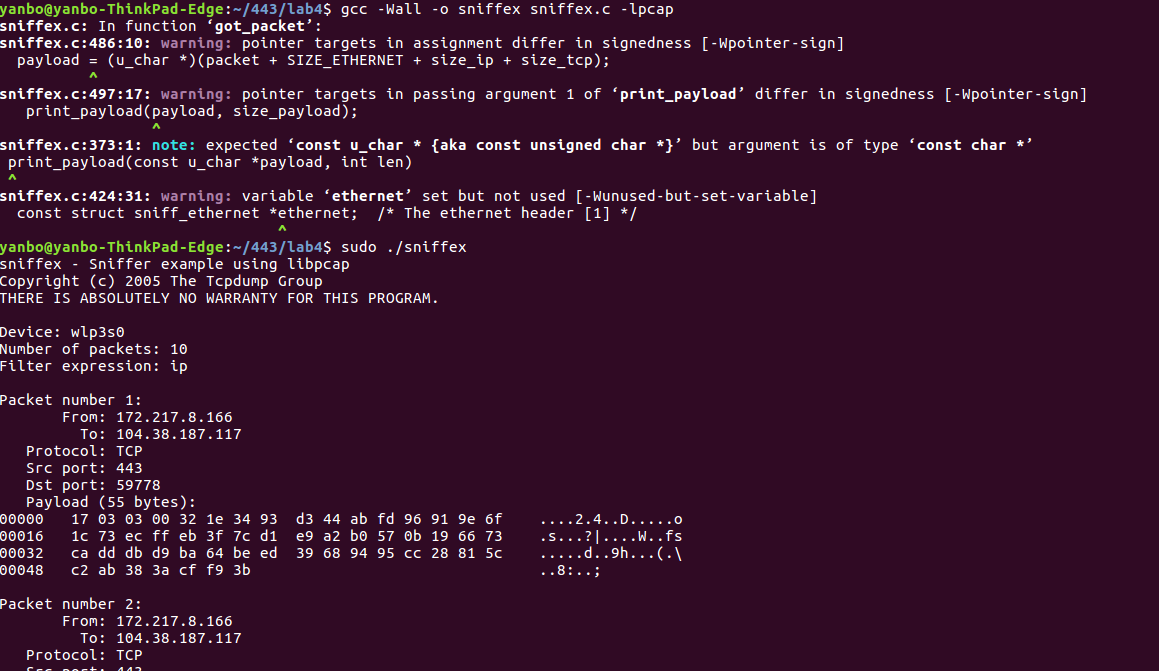
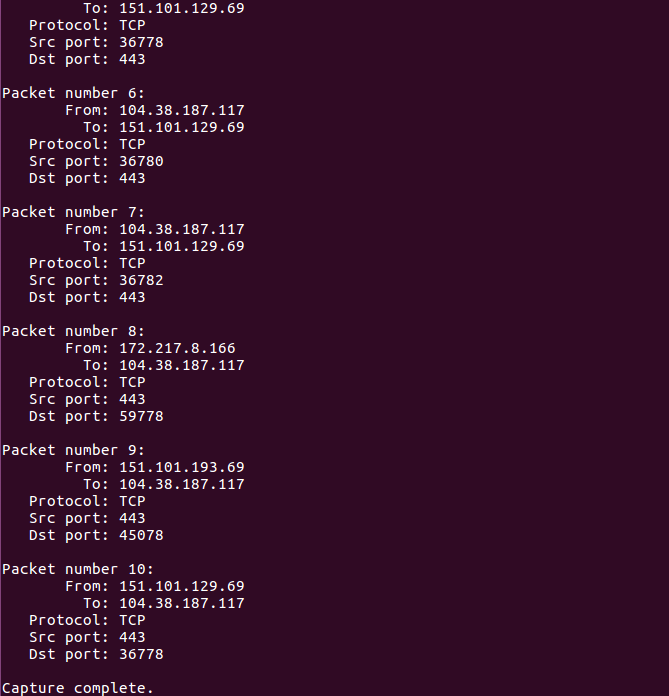
**Task 1**

Problem 1



  
The following calls are functions of sniffex.c:  
  
Pcap\_lookupdev is to find a capture device to sniff.

Pcap\_lookupnet returns the network number and mask of the capture device.

Pcap\_open\_live starts sniffing on the capture device.

Pcap\_datalink returns the device we are capturing.

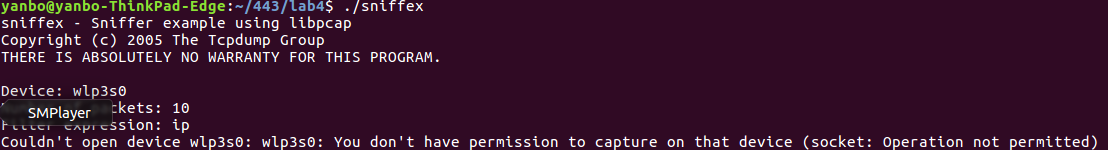
Pcap\_compile compiles a filter expression stored in a regular string to set the filter.

Pcap\_setfilter sets the compiled filter. We can sniff a packet (pcap\_next) or continuously sniff (pcap\_loop). We can either sniff one packet at a time (pcap\_next) or continuously sniff (pcap\_loop). We will continue with pcap\_loop: Sets callback function for new (filtered!) packets

Pcap\_freecode frees the memory generated by pcap\_compile.

Pcap\_close closes the sniffer session.

Problem 2



We need root for sniffex to run because sniffex will need to access a network device and non root user cannot do that.

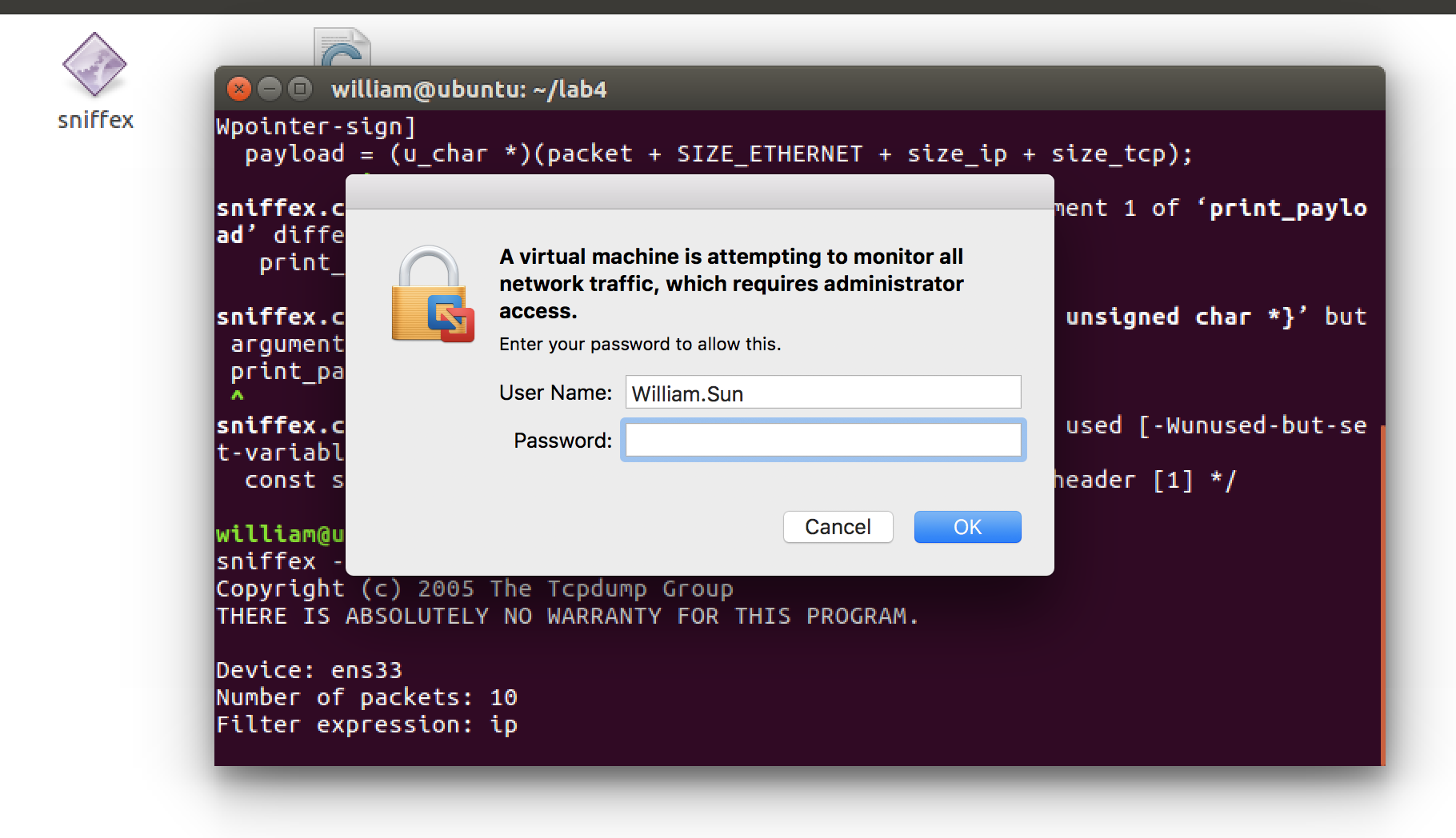
The code causing failure is:

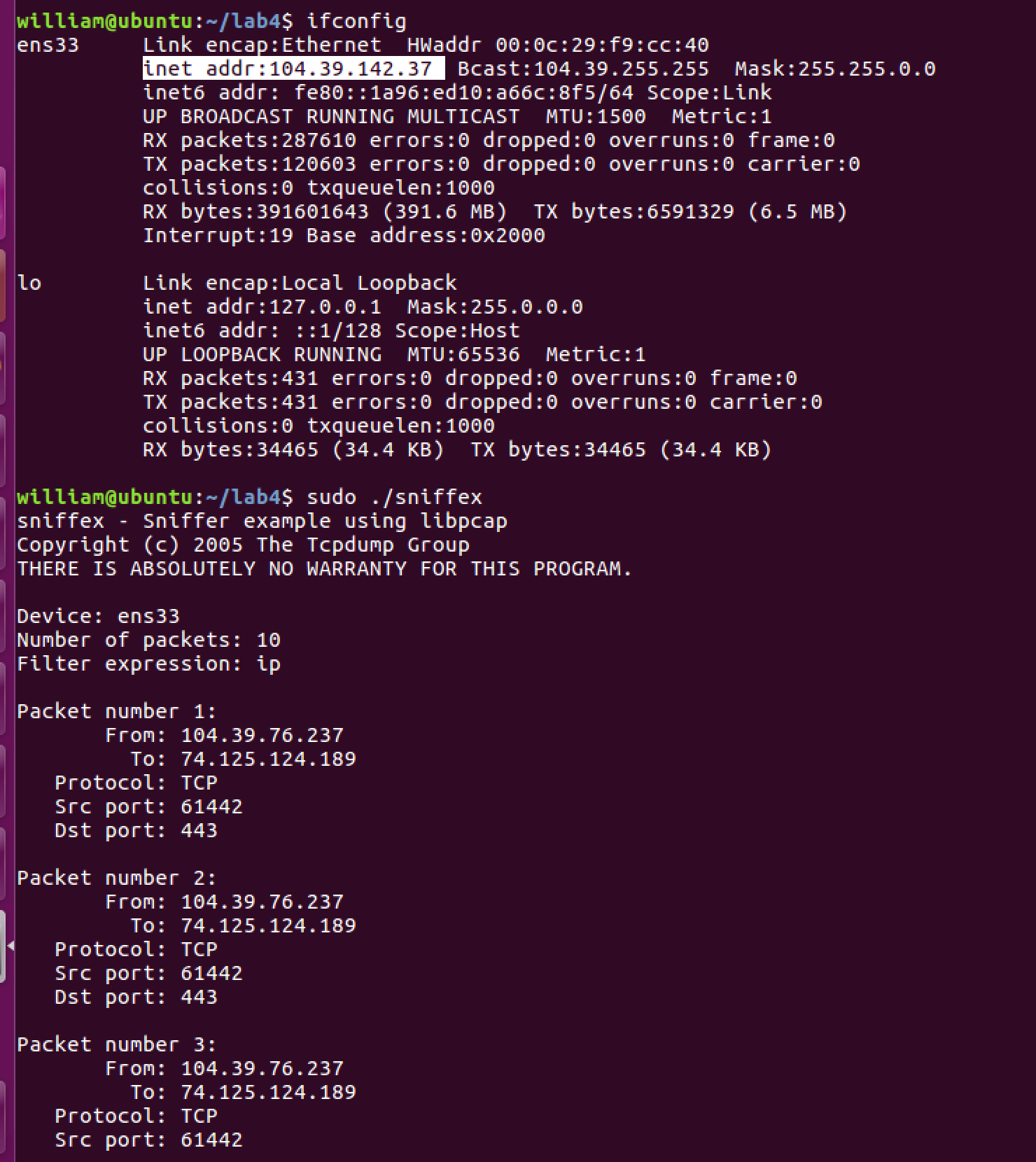
dev = pcap\_lookupdev(errbuf);  
if (dev == NULL) {  
 fprintf(stderr, "Couldn't find default device: %s\n", errbuf);  
 exit(EXIT\_FAILURE);  
}

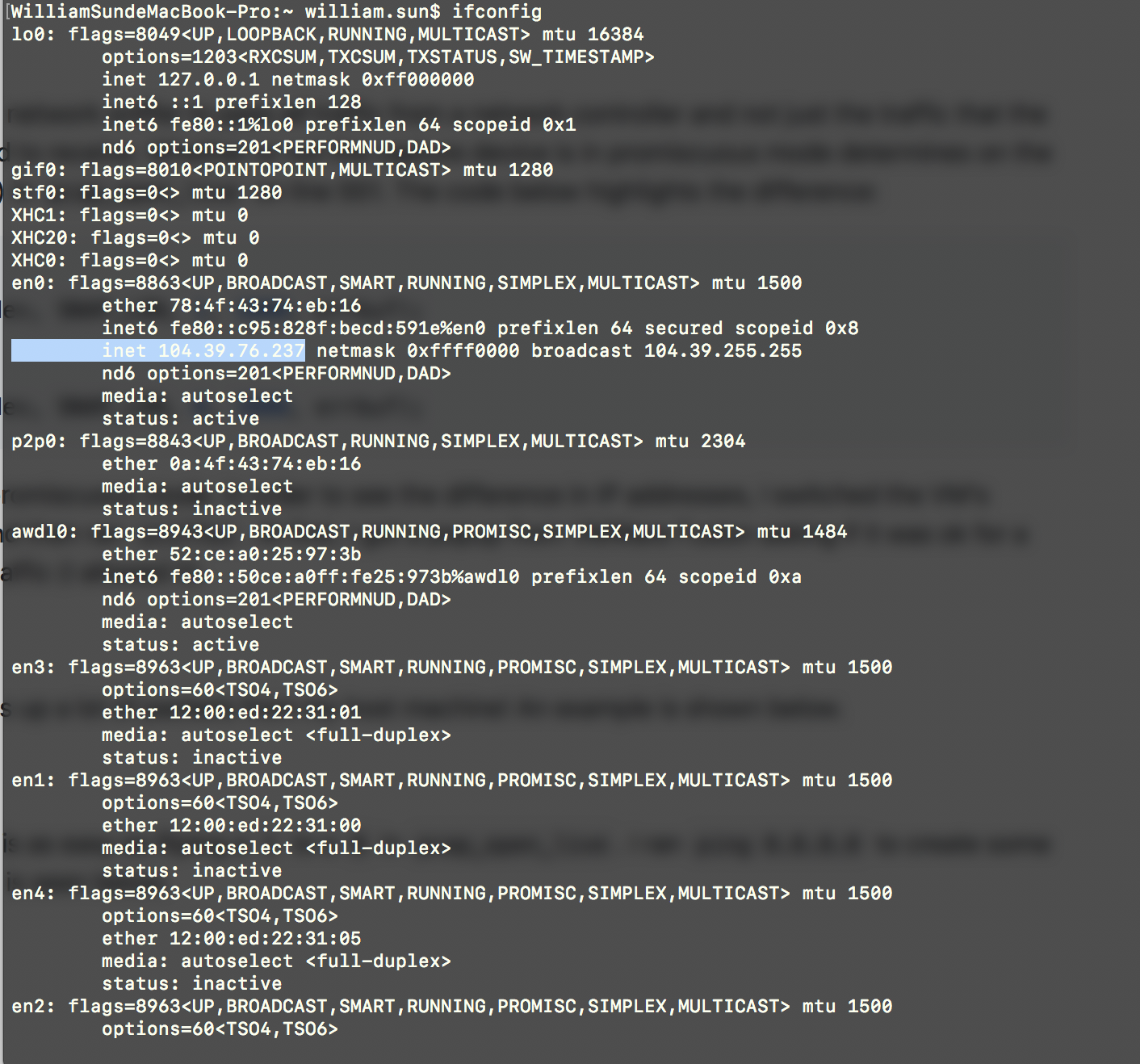
Problem 3

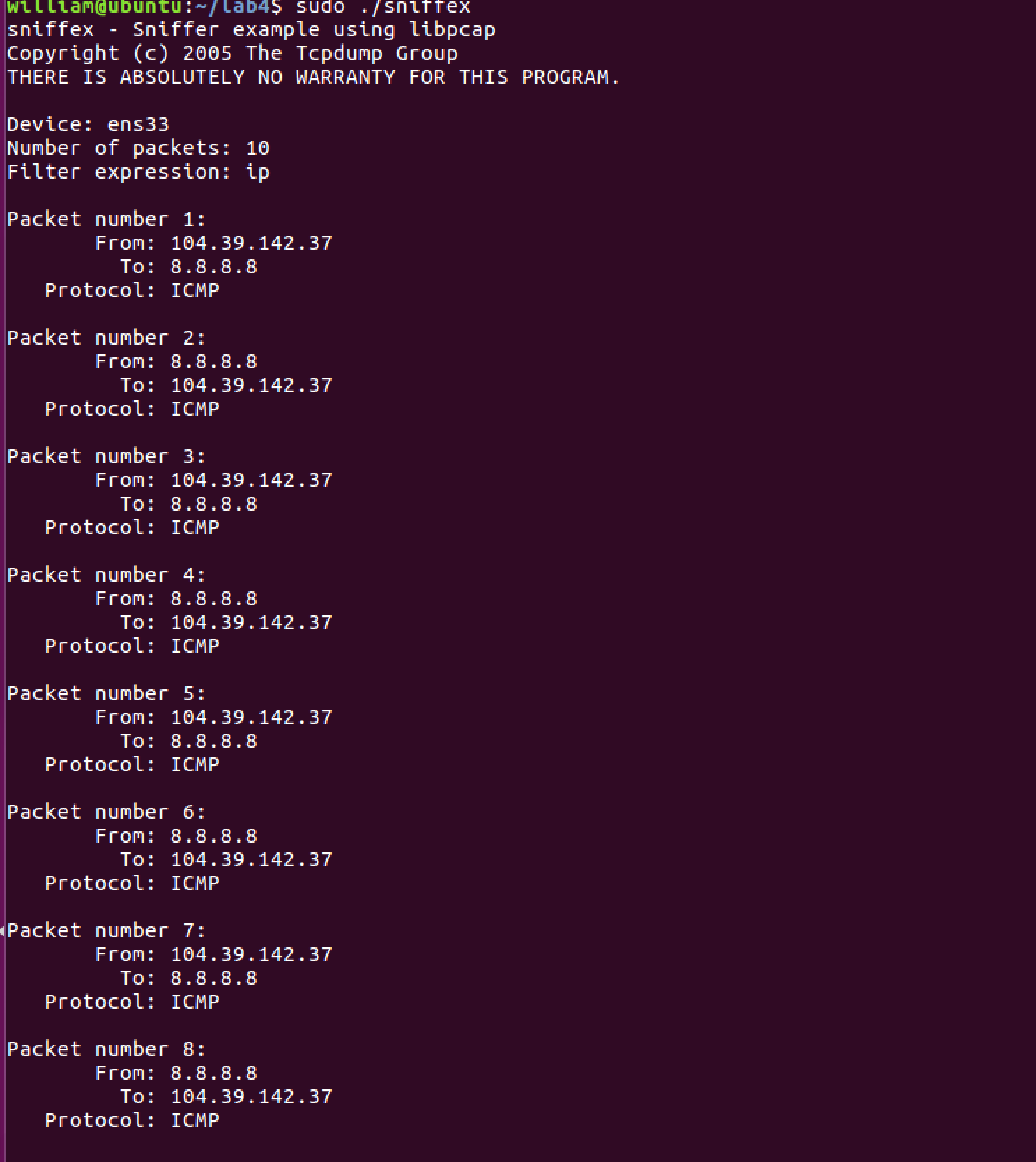
Hybrid mode allows network sniffers to transfer all traffic from network controllers, not just traffic to the network controller. On pcap\_open\_live, the capture device determines the third parameter (a Boolean int) in the mixed mode in line 551. The following code shows the differences.  
  
/\* promisc mode on \*/  
handle = pcap\_open\_live(dev, SNAP\_LEN, 1, 1000, errbuf);  
  
/\* promisc mode off \*/  
handle = pcap\_open\_live(dev, SNAP\_LEN, 0, 1000, errbuf);

By default, sniffex scoffs at hybrid models. To see the difference in IP addresses, we switched the VM's network adapter to "bridge" and then ran sniffex. First, we get a pop-up window from VMWare Fusion to ask if the virtual machine can monitor all traffic which we allow it.



As we allow it, sniffex picks up more packets from the host machine. It is shown below:  


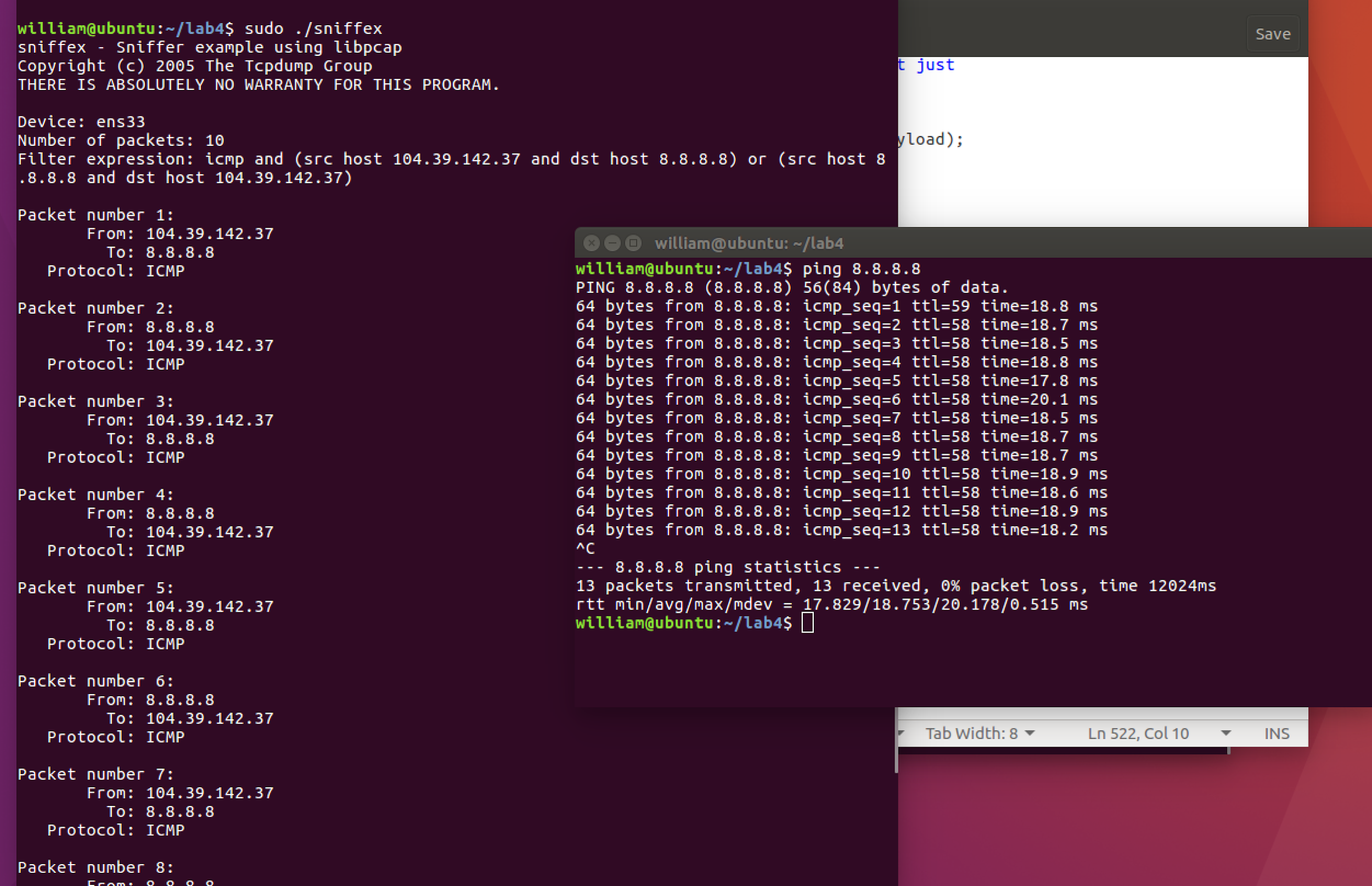
  
Then we turn off promiscuous mode from 1 to 0 in pcap\_open\_live. And we run ping 8.8.8.8 and create some network traffic and the results are:



Task 1.b(1)

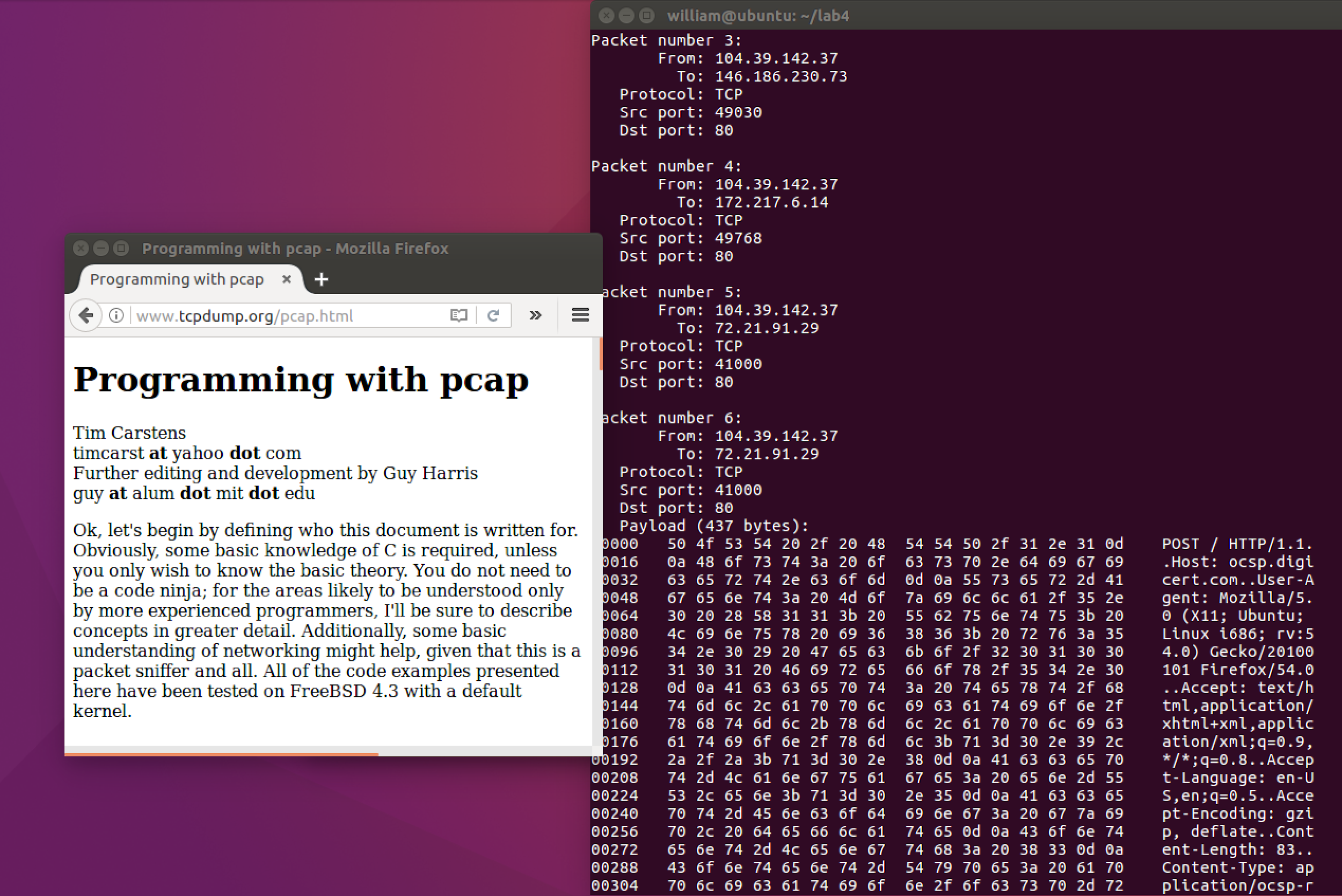
We filter only the ICMP packets by modifying the filter\_exp string by using  
char filter\_exp[] = "icmp and (src host 104.39.142.37 and dst host 8.8.8.8) or (src host 8.8.8.8 and dst host 104.39.142.37)";

Then we run the sniffex and ping 8.8.8.8 and got the following results:



Task 1.b (2)

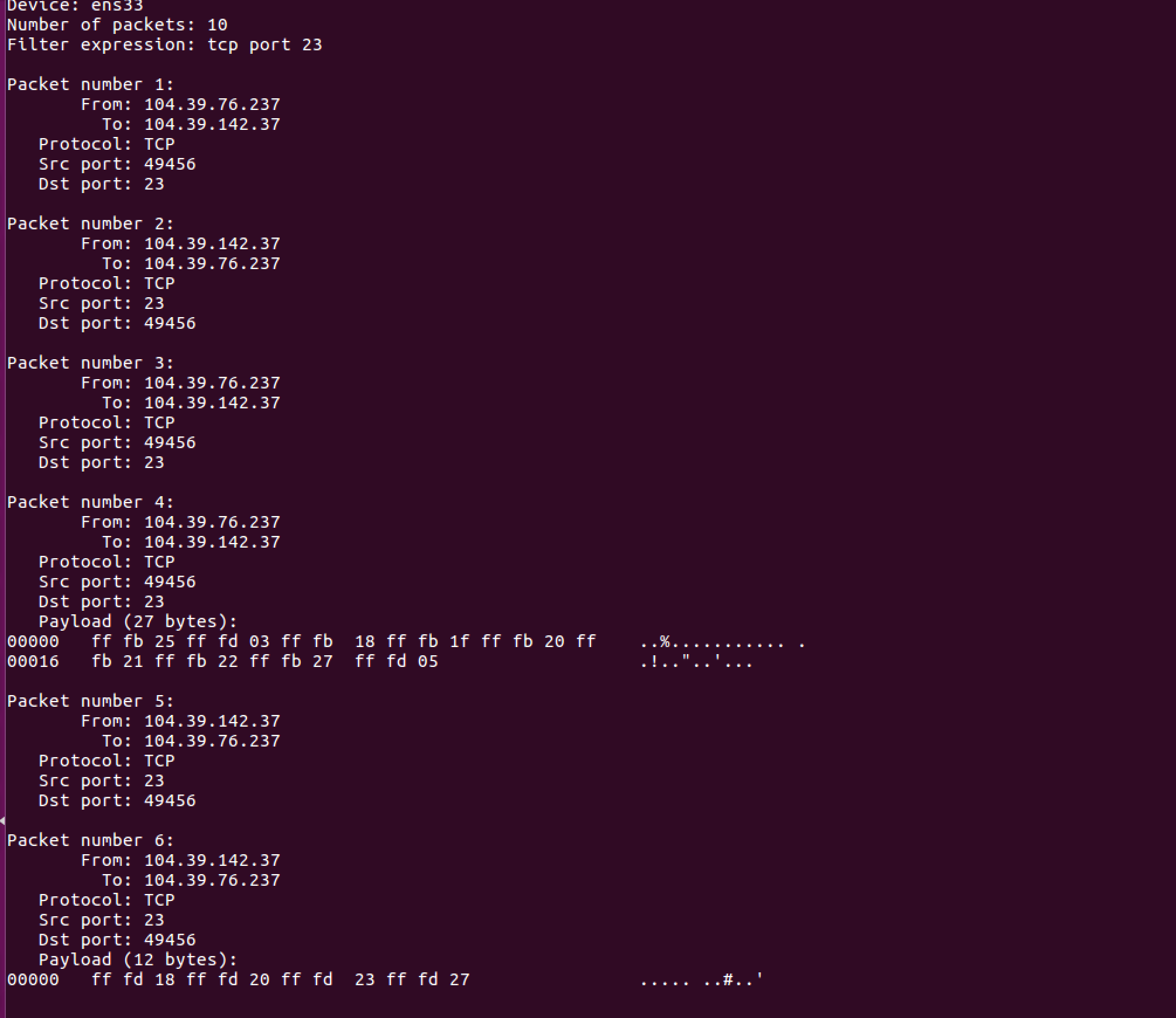
We change the filter\_exp variable and only sniff TCP packets with a destination port from 10 to 100.  
char filter\_exp[] = "tcp dst portrange 10-100";

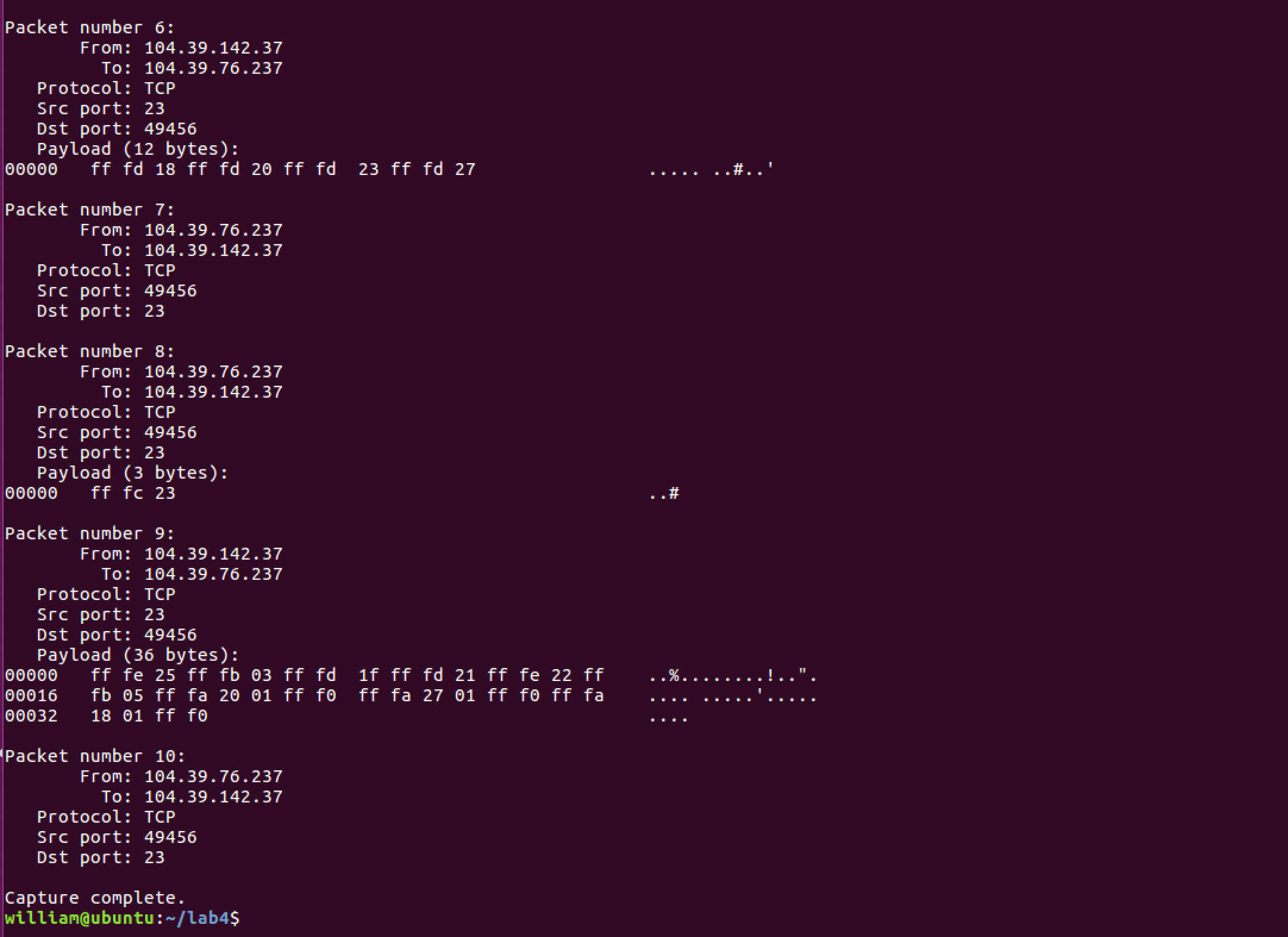
Then we navigate to a webpage in browser and got the following:  


Task 1.c

We have sniffed telnet passwords, we can just look for tcp packets on port 23.  
char filter\_exp[] = "tcp port 23";

Then we use the host machine and telnet into the VM.  



Sniffex will need some modification for us to sniff passwords. However, for short passwords like those password to the vm, there is no need for us to just output the payload.

Questions:

Question 4:

Yes since we can set the package length field to any value. But when we receive the package, it is always added to the zero that is not specified as packet size. Therefore, the payload is larger than the load and it will result in zero padding until the total length of the package is completed.

Question 5:

Yes since when we use the original socket programming in c, we must manually calculate the checksum, otherwise the error will be received. If we use the python scapy package, then it has a built-in function for us which is .chksum.

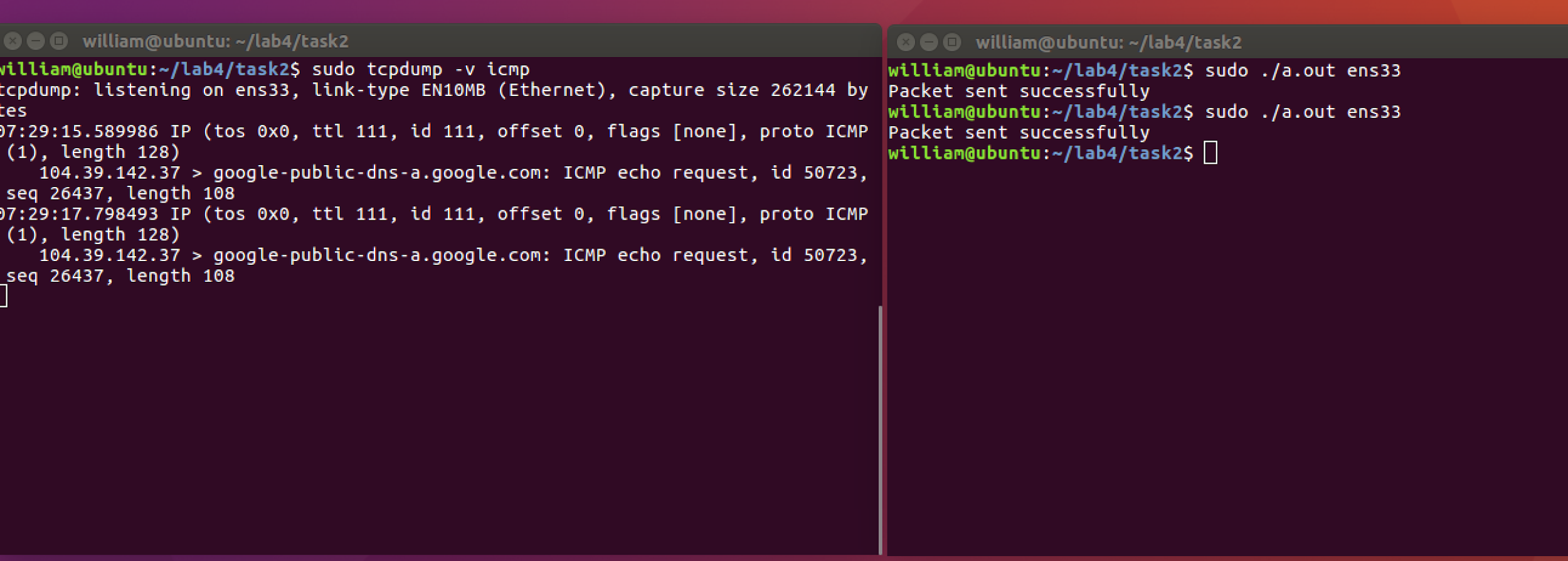
Question 6:

The original socket requires privileges that are not present in the normal user so it must be the root privilege user to run the command to start the original socket. It failed when it requested kernel access to create the socket.

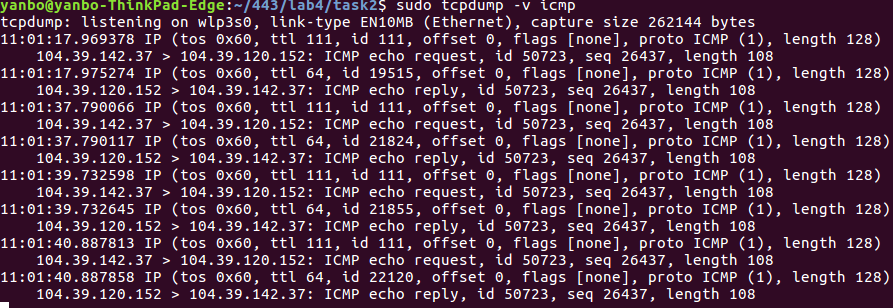
**Task 2**

Task 2 (a)

We find that the packets that has been caught by tcpdump. We use a spoofing program and changed the ip address and results are shown below:

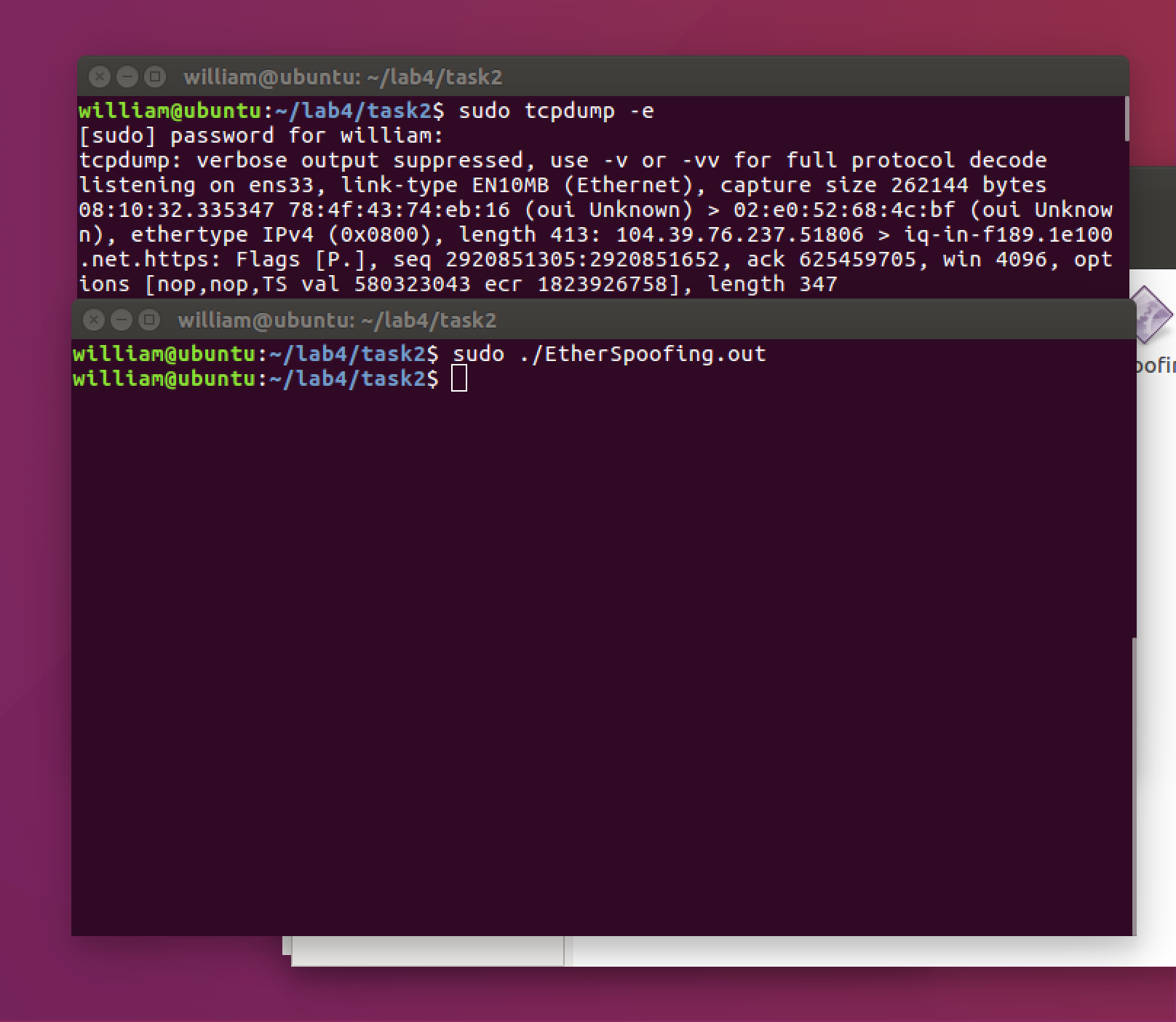


Task2 (b)



We spoofed an ICMP echo request packet on behalf of another machine and the packet was sent to the ip address of 104.39.120.152. The tcpdump caught requested packets and thus replied packets.

Task 2(c)



The source code is compiled and tested in the system. We successfully got the frames by tcpdump.

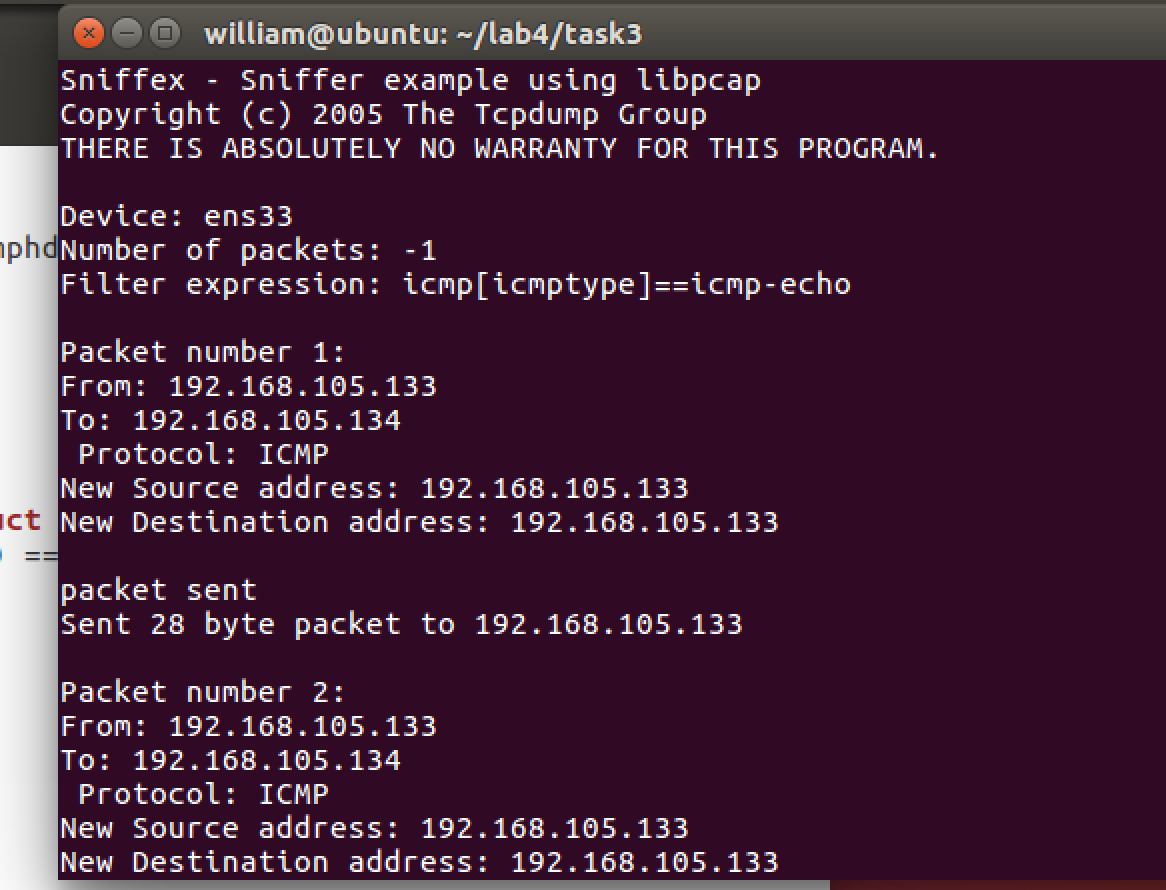
**Task 3**

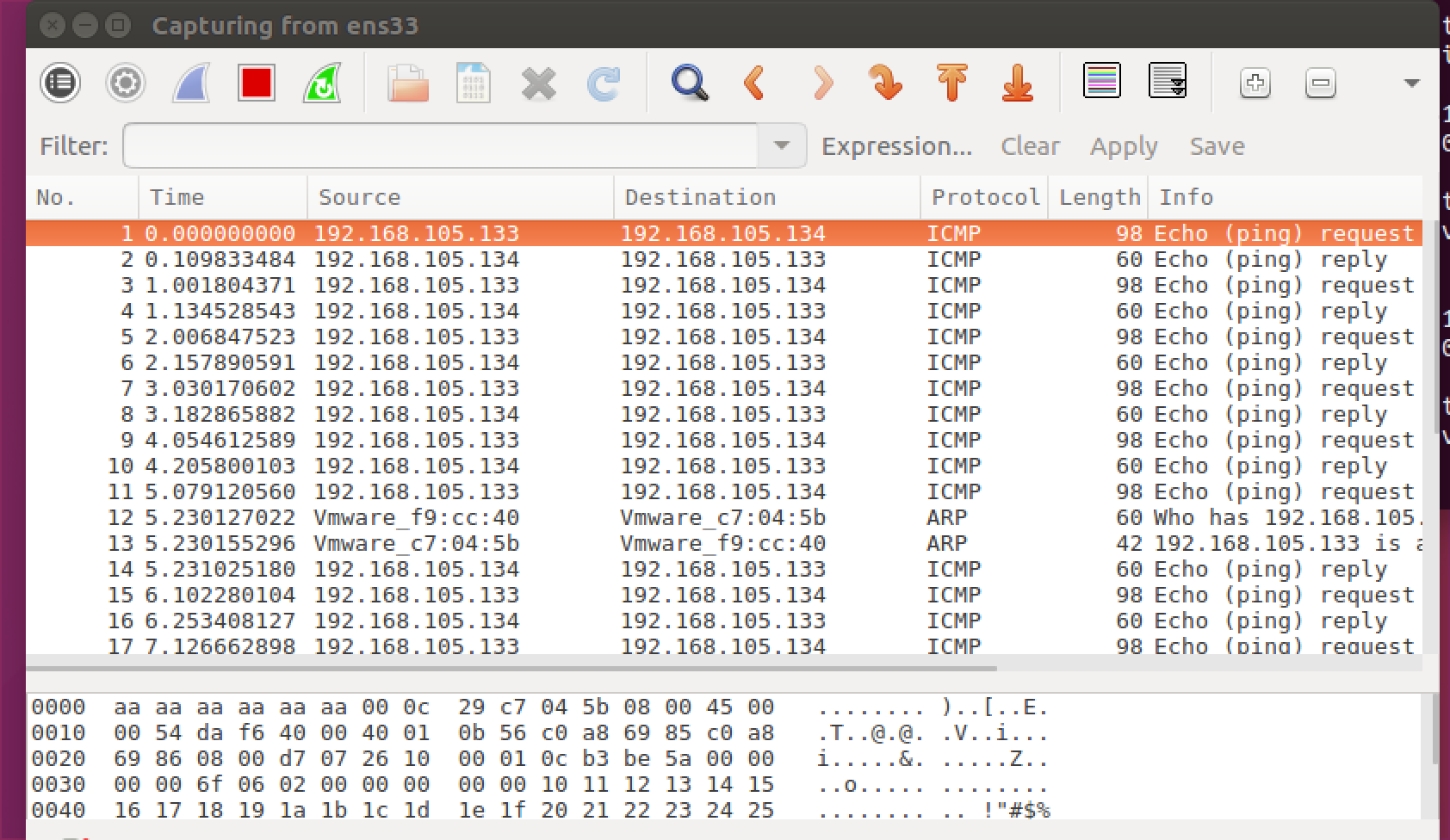
In this task, we use inet addr(), inet network(), inet ntoa(), inet aton() to  
convert IP addresses from the dotted decimal form (a string) to a 32-bit integer of network/host byte order.

The ip address of virtual machine A is 192.168.105.133. The ip address of virtual machine B is 192.168.105.132. Non-existing ip address is 192.168.105.134. When we ping a non-existing IP in the same LAN, an ARP request will be sent first. If there is no reply, the ICMP requests will not be sent later. So in order to avoid that ARP request which will stop the ICMP packet, we change the ARP cache of the victim VM by adding another MAC to the IP address mapping entry.

In this program, whenever it sees an ICMP echo request packet, it spoofs an ICMP echo reply packet. Therefore, even if the victim machine pings a non-existing machine, it will always see that the machine is alive.

In the program shown below, the ping's host and a sniffing program with spoofing facility. Basically my spoof program sniff an ICMP request packet on the network and make a ICMP reply packet and send it to the source of the ICMP request packet. When we ping the link, we are able to see the packets sent and received from VM A to VM B in wireshark and terminal, thus prove that our program can let VM B(192.168.105.132) relies VM A(192.168.105.133) by using non-existing host (192.168.105.134).





Our code as below:

#define APP\_NAME "Sniffex"

#define APP\_DESC "Sniffer example using libpcap"

#define APP\_COPYRIGHT "Copyright (c) 2005 The Tcpdump Group"

#define APP\_DISCLAIMER "THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM."

#include <pcap.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <ctype.h>

#include <errno.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <netdb.h>

#include <linux/ip.h>

#include <linux/icmp.h>

#include <string.h>

#include <unistd.h>

unsigned short in\_cksum(unsigned short \*, int);

void parse\_argvs(char\*\*, char\*, char\* );

void usage();

char\* getip();

char\* toip(char\*);

void sendicmp(char \*src\_addr,char \* dst\_addr)

{

struct iphdr\* ip;

struct iphdr\* ip\_reply;

struct icmphdr\* icmp;

struct sockaddr\_in connection;

char\* packet;

char\* buffer;

int sockfd;

int one=1;

int \*optval=&one;

int addrlen;

int siz;

/\*

\* allocate all necessary memory

//

\*/

printf("New Source address: %s\n", dst\_addr);

printf("New Destination address: %s\n", src\_addr);

packet = malloc(sizeof(struct iphdr) + sizeof(struct icmphdr));

buffer = malloc(sizeof(struct iphdr) + sizeof(struct icmphdr));

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

ip = (struct iphdr\*) packet;

icmp = (struct icmphdr\*) (packet + sizeof(struct iphdr));

/\*\*

here the ip packet is set up

\*/

ip->ihl = 5;

ip->version = 4;

ip->tos = 0;

ip->tot\_len = sizeof(struct iphdr) + sizeof(struct icmphdr);

ip->id = htons(0);

ip->frag\_off= 0;

ip->ttl= 64;

ip->protocol= IPPROTO\_ICMP;

ip->saddr = inet\_addr("10.20.103.90");

ip->daddr = inet\_addr(src\_addr);

ip->check =0;

ip->check = in\_cksum((unsigned short \*)ip, sizeof(struct iphdr));

if ((sockfd = socket(AF\_INET, SOCK\_RAW, IPPROTO\_ICMP)) == -1)

{

perror("socket");

exit(EXIT\_FAILURE);

}

/\*

\* IP\_HDRINCL must be set on the socket so that

\* the kernel does not attempt to automatically add

\* a default ip header to the packet

\*/

//

optval=1;

setsockopt(sockfd, IPPROTO\_IP, IP\_HDRINCL, &optval, sizeof(int));

/\*

\* here the icmp packet is created\* also the ip checksum is generated

\*/

icmp->type = 0;

icmp->code = 0;

icmp->un.echo.id = random();

icmp->un.echo.sequence= 0;

icmp->checksum=0;

icmp-> checksum= in\_cksum((unsigned short \*)icmp, sizeof(struct icmphdr));

ip->check= in\_cksum((unsigned short \*)packet, sizeof(struct iphdr)+sizeof(struct icmphdr));

connection.sin\_family = AF\_INET;

connection.sin\_addr.s\_addr = ip->daddr;

//connection.sin\_addr.\_addr = inet\_addr(src\_addr);

printf("\npacket sent\n");

/\*

now the packet is sent

\*/

//memcpy(packet+20,&icmp,8);

if( (sendto(sockfd, packet, ip->tot\_len, 0, (struct sockaddr \*)&connection, sizeof(struct sockaddr)))==-1)

{

perror("socket");

exit(EXIT\_FAILURE);

}

printf("Sent %d byte packet to %s\n", ip->tot\_len, src\_addr);

/\*

\*

now we listen for responses

\*/

free(packet);

free(buffer);

close(sockfd);

//return 0;

}

void usage()

{

fprintf(stderr, "\nUsage: pinger [destination] <-s [source]>\n");

fprintf(stderr, "Destination must be provided\n");

fprintf(stderr, "Source is optional\n\n");

}

/\*

\* return the ip address if host provided by DNS name

\*/

char\* toip(char\* address)

{

struct hostent\* h;

h = gethostbyname(address);

return inet\_ntoa(\*(struct in\_addr \*)h->h\_addr);

}

/\*

\* in\_cksum --

\* Checksum routine for Internet Protocol

\* family headers (C Version)

\*/

unsigned short in\_cksum(unsigned short \*addr, int len)

{

register int sum = 0;

u\_short answer = 0;

register u\_short \*w = addr;

register int nleft = len;

/\*\* Our algorithm is simple, using a 32 bit accumulator (sum), we add

\* sequential 16 bit words to it, and at the end, fold back all the

\* carry bits from the top 16 bits into the lower 16 bits.

\*/

while (nleft > 1)

{

sum += \*w++;

nleft -= 2;

}

/\* mop up an odd byte, if necessary \*/

if (nleft == 1)

{

\*(u\_char \*) (&answer) = \*(u\_char \*) w;

sum += answer;

}

/\* 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 \*/

answer = ~sum; /\* truncate to 16 bits \*/

return (answer);

}

/\* default snap length (maximum bytes per packet to capture) \*/

#define SNAP\_LEN 1518

/\* ethernet headers are always exactly 14 bytes [1] \*/

#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 {

u\_char ip\_vhl;

/\* version << 4 | header length >> 2 \*/

u\_char ip\_tos;

/\* type of service \*/

u\_short ip\_len;

/\* total length \*/

u\_short ip\_id;

/\* identification \*/

u\_short ip\_off;

/\* fragment offset field \*/

#define IP\_RF 0x8000

/\* reserved fragment flag \*/

#define IP\_DF 0x4000

/\* dont fragment flag \*/

#define IP\_MF 0x2000

/\* more fragments flag \*/

#define IP\_OFFMASK 0x1fff

/\* mask for fragmenting bits \*/

u\_char ip\_ttl;

/\* time to live \*/

u\_char ip\_p;

/\* protocol \*/

u\_short ip\_sum;

/\* checksum \*/

struct in\_addr ip\_src,ip\_dst; /\* source and dest address \*/

};

#define IP\_HL(ip) ((ip->ip\_vhl) & 0x0f)

#define IP\_V(ip) ((ip->ip\_vhl) >> 4)

/\* TCP header \*/

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);

//void print\_payload(const u\_char \*payload, int len);

//void print\_hex\_ascii\_line(const u\_char \*payload, int len, int offset);

//void print\_app\_banner(void);

//void print\_app\_usage(void);

/\*

\* app name/banner

\*/

void print\_app\_banner(void)

{

printf("%s - %s\n", APP\_NAME, APP\_DESC);

printf("%s\n", APP\_COPYRIGHT);

printf("%s\n", APP\_DISCLAIMER);

printf("\n");

return;

}

/\*

\* print help text

\*/

void print\_app\_usage(void)

{

printf("Usage: %s [interface]\n", APP\_NAME);

printf("\n");

printf("Options:\n");

printf(" interface Listen on <interface> for packets.\n");

printf("\n");

return;

}

/\*

\* print data in rows of 16 bytes: offset hex ascii

\*

\* 00000 47 45 54 20 2f 20 48 54 54 50 2f 31 2e 31 0d 0a GET / HTTP/1.1..

\*/

void print\_hex\_ascii\_line(const u\_char \*payload, int len, int offset)

{

int i;

int gap;

const u\_char \*ch;

/\* offset \*/

printf("%05d ", offset);

/\* hex \*/

ch = payload;

for(i = 0; i < len; i++) {

printf("%02x ", \*ch);

ch++;

/\* print extra space after 8th byte for visual aid \*/

if (i == 7)

printf(" ");

}

/\* print space to handle line less than 8 bytes \*/

if (len < 8)

printf(" ");

/\* fill hex gap with spaces if not full line \*/

if (len < 16) {

gap = 16 - len;

for (i = 0; i < gap; i++) {

printf(" ");

}

}

printf(" ");

/\* ascii (if printable) \*/

ch = payload;

for(i = 0; i < len; i++) {

if (isprint(\*ch))

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

else

printf(".");

ch++;

}

printf("\n");

return;

}

/\*

\* print packet payload data (avoid printing binary data)

\*/

void print\_payload(const u\_char \*payload, int len)

{

int len\_rem = len;

int line\_width = 16;

int line\_len;

int offset = 0;

/\* number of bytes per line \*/

/\* zero-based offset counter \*/

const u\_char \*ch = payload;

if (len <= 0)

return;

/\* data fits on one line \*/

if (len <= line\_width) {

print\_hex\_ascii\_line(ch, len, offset);

return;

}

/\* data spans multiple lines \*/

for ( ;; ) {

/\* compute current line length \*/

line\_len = line\_width % len\_rem;

/\* print line \*/

print\_hex\_ascii\_line(ch, line\_len, offset);

/\* compute total remaining \*/

len\_rem = len\_rem - line\_len;

/\* shift pointer to remaining bytes to print \*/

ch = ch + line\_len;

/\* add offset \*/

offset = offset + line\_width;

/\* check if we have line width chars or less \*/

if (len\_rem <= line\_width) {

/\* print last line and get out \*/

print\_hex\_ascii\_line(ch, len\_rem, offset);

break;

}

}

return;

}

/\*

\* dissect/print packet

\*/

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

{

//sendicmp(inet\_ntoa(ip->ip\_src),inet\_ntoa(ip->ip\_dst));

static int count = 1;

/\* packet counter \*/

/\* declare pointers to packet headers \*/

const struct sniff\_ethernet \*ethernet; /\* The ethernet header [1] \*/

const struct sniff\_ip \*ip;

/\* The IP header \*/

const struct sniff\_tcp \*tcp;

/\* The TCP header \*/

const char \*payload;

/\* Packet payload \*/

int size\_ip;

int size\_tcp;

int size\_payload;

//sendicmp(inet\_ntoa(ip->ip\_src),inet\_ntoa(ip->ip\_dst));

printf("\nPacket number %d:\n", count);

count++;

/\* define ethernet header \*/

ethernet = (struct sniff\_ethernet\*)(packet);

/\* define/compute ip header offset \*/

ip = (struct sniff\_ip\*)(packet + SIZE\_ETHERNET);

size\_ip = IP\_HL(ip)\*4;

if (size\_ip < 20)

{

printf(" \* Invalid IP header length: %u bytes\n", size\_ip);

return;

}

/\* print source and destination IP addresses \*/

printf("From: %s\n", inet\_ntoa(ip->ip\_src));

printf("To: %s\n", inet\_ntoa(ip->ip\_dst));

/\* determine protocol \*/

switch(ip->ip\_p) {

case IPPROTO\_TCP:

printf(" Protocol: TCP\n");

break;

case IPPROTO\_UDP:

printf(" Protocol: UDP\n");

return;

case IPPROTO\_ICMP:

printf(" Protocol: ICMP\n");

sendicmp(inet\_ntoa(ip->ip\_src),inet\_ntoa(ip->ip\_dst));

return;

case IPPROTO\_IP:

printf(" Protocol: IP\n");

return;

default:

printf(" Protocol: unknown\n");

return;

}

/\*

\* OK, this packet is TCP.

\*/

/\* define/compute tcp header offset \*/

tcp = (struct sniff\_tcp\*)(packet + SIZE\_ETHERNET + size\_ip);

size\_tcp = TH\_OFF(tcp)\*4;

if (size\_tcp < 20) {

printf(" \* Invalid TCP header length: %u bytes\n", size\_tcp);

return;

}

printf(" Src port: %d\n", ntohs(tcp->th\_sport));

printf(" Dst port: %d\n", ntohs(tcp->th\_dport));

/\* define/compute tcp payload (segment) offset \*/

payload = (u\_char \*)(packet + SIZE\_ETHERNET + size\_ip + size\_tcp);

/\* compute tcp payload (segment) size \*/

size\_payload = ntohs(ip->ip\_len) - (size\_ip + size\_tcp);

/\*

\* Print payload data; it might be binary, so don't just

\* treat it as a string.

\*/

if (size\_payload > 0) {

printf(" Payload (%d bytes):\n", size\_payload);

print\_payload(payload, size\_payload);

}

//sendicmp(inet\_ntoa(ip->ip\_src),inet\_ntoa(ip->ip\_dst));

return;

}

int main(int argc, char \*\*argv)

{

char \*dev = NULL; /\* capture device name \*/

char errbuf[PCAP\_ERRBUF\_SIZE]; /\* error buffer \*/

pcap\_t \*handle; /\* packet capture handle \*/

char \*filter\_exp=argv[1]; /\* filter expression [3] \*/

struct bpf\_program fp; /\* compiled filter program (expression) \*/

bpf\_u\_int32 mask; /\* subnet mask \*/

bpf\_u\_int32 net; /\* ip \*/

int num\_packets = 10; /\* number of packets to capture \*/

print\_app\_banner();

/\* check for capture device name on command-line

if (argc == 2) {

dev = argv[1];

}

else if (argc > 2) {

fprintf(stderr, "error: unrecognized command-line options\n\n");

print\_app\_usage();

exit(EXIT\_FAILURE);

}

else {

find a capture device if not specified on command-line \*/

dev = pcap\_lookupdev(errbuf);

if (dev == NULL) {

fprintf(stderr, "Couldn't find default device: %s\n",

errbuf);

exit(EXIT\_FAILURE);

}

/\* get network number and mask associated with capture device \*/

if (pcap\_lookupnet(dev, &net, &mask, errbuf) == -1) {

fprintf(stderr, "Couldn't get netmask for device %s: %s\n",

dev, errbuf);

net = 0;

mask = 0;

}

/\* print capture info \*/

printf("Device: %s\n", dev);

printf("Number of packets: %d\n", num\_packets);

printf("Filter expression: %s\n", filter\_exp);

/\* open capture device \*/

handle = pcap\_open\_live(dev, SNAP\_LEN, 1, 1000, errbuf);

if (handle == NULL) {

fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);

exit(EXIT\_FAILURE);

}

/\* make sure we're capturing on an Ethernet device [2] \*/

if (pcap\_datalink(handle) != DLT\_EN10MB) {

fprintf(stderr, "%s is not an Ethernet\n", dev);

exit(EXIT\_FAILURE);

}

/\* compile the filter expression \*/

if (pcap\_compile(handle, &fp, filter\_exp, 0, net) == -1) {

fprintf(stderr, "Couldn't parse filter %s: %s\n",

filter\_exp, pcap\_geterr(handle));

exit(EXIT\_FAILURE);

}

/\* apply the compiled filter \*/

if (pcap\_setfilter(handle, &fp) == -1) {

fprintf(stderr, "Couldn't install filter %s: %s\n",

filter\_exp, pcap\_geterr(handle));

exit(EXIT\_FAILURE);

}

/\* now we can set our callback function \*/

pcap\_loop(handle, num\_packets, got\_packet, NULL);

/\* cleanup \*/

pcap\_freecode(&fp);

pcap\_close(handle);

printf("\nCapture complete.\n");

return 0;

}