Task 1

Problem 1

```
yanbo@yanbo-ThinkPad-Edge:~/443/lab4$ gcc -Wall -o sniffex sniffex.c -lpcap
sniffex.c: In function ‘got_packet’:
sniffex.c:486:10: warning: pointer targets in assignment differ in signedness [-Wpointer-sign]
  payload = (u_char *)(packet + SIZE_ETHERNET + size_ip + size_tcp);
          ^
sniffex.c:497:17: warning: pointer targets in passing argument 1 of ‘print_payload’ differ in signedness [-Wpointer-sign]
   print_payload(payload, size_payload);
                 ^
sniffex.c:373:1: note: expected ‘const u_char * {aka const unsigned char *}’ but argument is of type ‘const char *’
 print_payload(const u_char *payload, int len)
 ^
sniffex.c:424:31: warning: variable ‘ethernet’ set but not used [-Wunused-but-set-variable]
  const struct sniff_ethernet *ethernet;  /* The ethernet header [1] */
                               ^
yanbo@yanbo-ThinkPad-Edge:~/443/lab4$ sudo ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: wlp3s0
Number of packets: 10
Filter expression: ip

Packet number 1:
       From: 172.217.8.166
         To: 104.38.187.117
   Protocol: TCP
   Src port: 443
   Dst port: 59778
   Payload (55 bytes):
00000   17 03 03 00 32 1e 34 93  d3 44 ab fd 96 91 9e 6f    ....2.4..D.....o
00016   1c 73 ec ff eb 3f 7c d1  e9 a2 b0 57 0b 19 66 73    .s...?|....W..fs
00032   ca dd db d9 ba 64 be ed  39 68 94 95 cc 28 81 5c    .....d..9h...(.\
00048   c2 ab 38 3a cf f9 3b                                ..8:..;

Packet number 2:
       From: 172.217.8.166
         To: 104.38.187.117
   Protocol: TCP
   Src port: 443
```

```
        To: 151.101.129.69
   Protocol: TCP
   Src port: 36778
   Dst port: 443

Packet number 6:
       From: 104.38.187.117
        To: 151.101.129.69
   Protocol: TCP
   Src port: 36780
   Dst port: 443

Packet number 7:
       From: 104.38.187.117
        To: 151.101.129.69
   Protocol: TCP
   Src port: 36782
   Dst port: 443

Packet number 8:
       From: 172.217.8.166
        To: 104.38.187.117
   Protocol: TCP
   Src port: 443
   Dst port: 59778

Packet number 9:
       From: 151.101.193.69
        To: 104.38.187.117
   Protocol: TCP
   Src port: 443
   Dst port: 45078

Packet number 10:
       From: 151.101.129.69
        To: 104.38.187.117
   Protocol: TCP
   Src port: 443
   Dst port: 36778

Capture complete.
```

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.

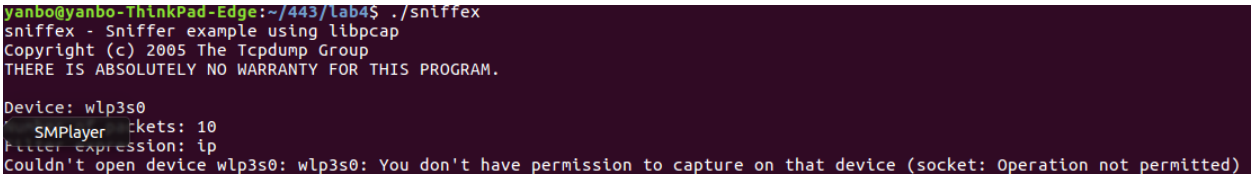
Pcap_compile compiles a filter expression stored in a regular string to set the filter.

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:

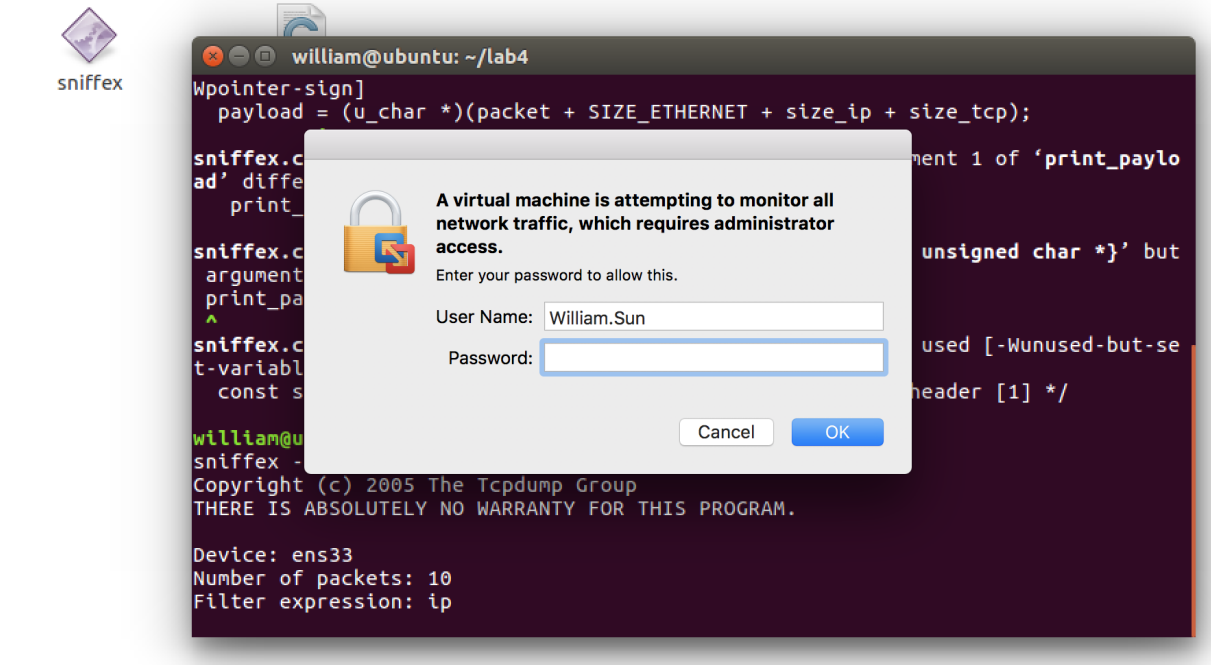
```
dev = pcap_lookupdev(errbuf);
if (dev == NULL) {
        fprintf(stderr, "Couldn't find default device: %s\n", errbuf);
        exit(EXIT_FAILURE);
}
```

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.

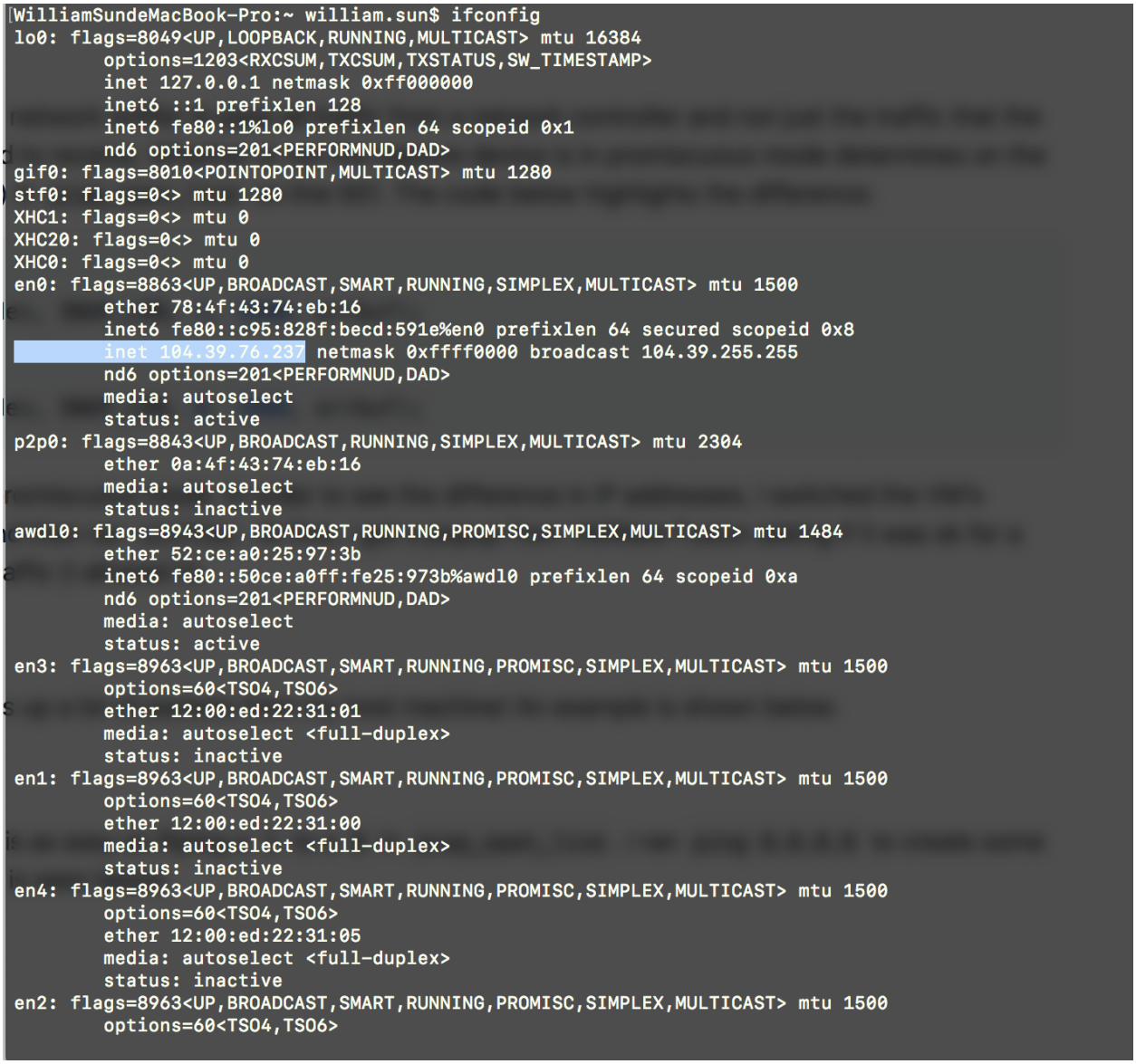
```
/* promisc mode on */
handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);

/* 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.


sniffex
william@ubuntu: ~/lab4
Wpointer-sign]
payload = (u_char *)(packet + SIZE_ETHERNET + size_ip + size_tcp);
sniffex.c
ment 1 of 'print_paylo
ad' diffe
print_
sniffex.c
unsigned char *}' but
argument
print_pa
sniffex.c
used [-Wunused-but-se
t-variabl
const s
header [1] */
william@u
sniffex -
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.
Device: ens33
Number of packets: 10
Filter expression: ip
A virtual machine is attempting to monitor all network traffic, which requires administrator access.
Enter your password to allow this.
User Name:
William.Sun
Password:
Cancel
OK

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

```
william@ubuntu:~/lab4$ ifconfig
ens33     Link encap:Ethernet  HWaddr 00:0c:29:f9:cc:40
          inet addr:104.39.142.37  Bcast:104.39.255.255  Mask:255.255.0.0
          inet6 addr: fe80::1a96:ed10:a66c:8f5/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:287610 errors:0 dropped:0 overruns:0 frame:0
          TX packets:120603 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:391601643 (391.6 MB)  TX bytes:6591329 (6.5 MB)
          Interrupt:19 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:431 errors:0 dropped:0 overruns:0 frame:0
          TX packets:431 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:34465 (34.4 KB)  TX bytes:34465 (34.4 KB)

william@ubuntu:~/lab4$ sudo ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: ens33
Number of packets: 10
Filter expression: ip

Packet number 1:
       From: 104.39.76.237
         To: 74.125.124.189
   Protocol: TCP
   Src port: 61442
   Dst port: 443

Packet number 2:
       From: 104.39.76.237
         To: 74.125.124.189
   Protocol: TCP
   Src port: 61442
   Dst port: 443

Packet number 3:
       From: 104.39.76.237
         To: 74.125.124.189
   Protocol: TCP
   Src port: 61442
```



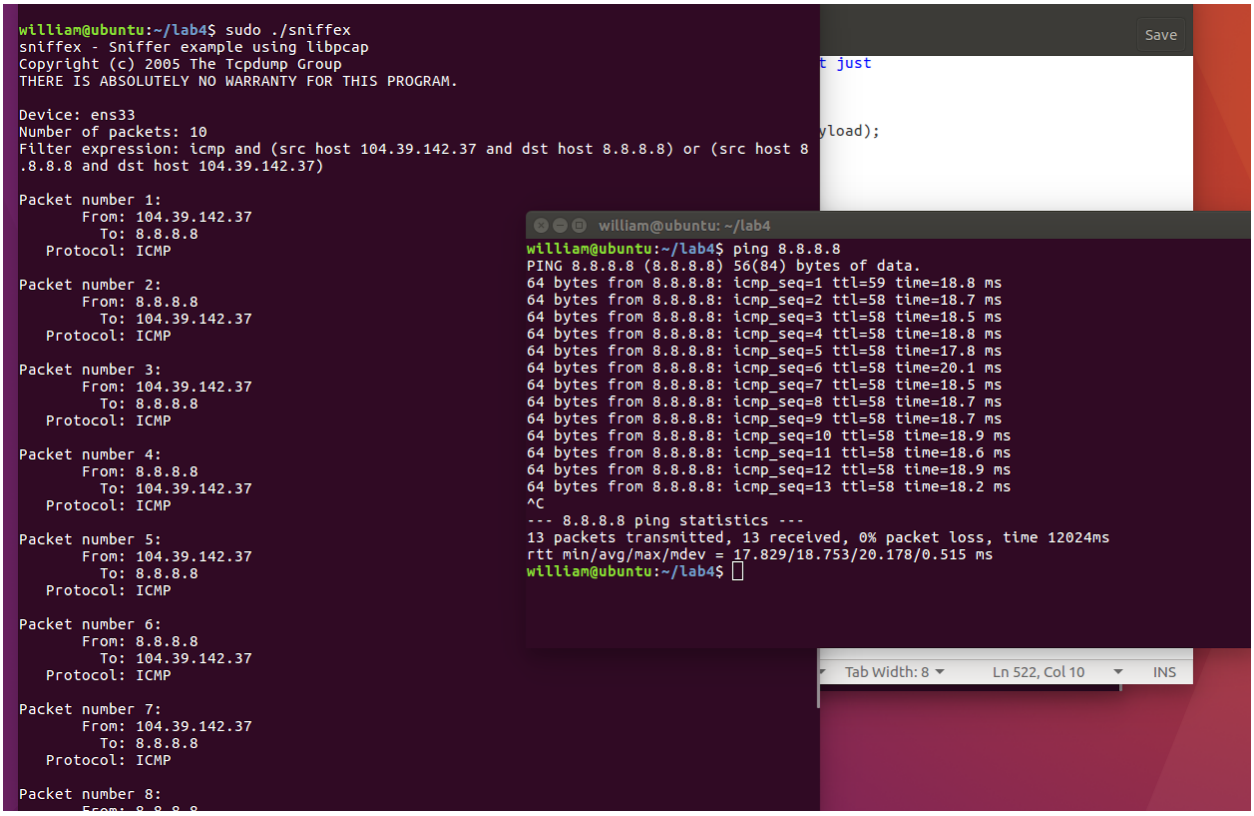
```
WilliamSundeMacBook-Pro:~ william.sun$ ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
        options=1203<RXCSUM,TXCSUM,TXSTATUS,SW_TIMESTAMP>
        inet 127.0.0.1 netmask 0xff000000
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
        nd6 options=201<PERFORMNUD,DAD>
gif0: flags=8010<POINTOPOINT,MULTICAST> mtu 1280
stf0: flags=0<> mtu 1280
XHC1: flags=0<> mtu 0
XHC20: flags=0<> mtu 0
XHC0: flags=0<> mtu 0
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        ether 78:4f:43:74:eb:16
        inet6 fe80::c95:828f:becd:591e%en0 prefixlen 64 secured scopeid 0x8
        inet 104.39.76.237 netmask 0xffff0000 broadcast 104.39.255.255
        nd6 options=201<PERFORMNUD,DAD>
        media: autoselect
        status: active
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
        ether 0a:4f:43:74:eb:16
        media: autoselect
        status: inactive
awdl0: flags=8943<UP,BROADCAST,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1484
        ether 52:ce:a0:25:97:3b
        inet6 fe80::50ce:a0ff:fe25:973b%awdl0 prefixlen 64 scopeid 0xa
        nd6 options=201<PERFORMNUD,DAD>
        media: autoselect
        status: active
en3: flags=8963<UP,BROADCAST,SMART,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1500
        options=60<TSO4,TSO6>
        ether 12:00:ed:22:31:01
        media: autoselect <full-duplex>
        status: inactive
en1: flags=8963<UP,BROADCAST,SMART,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1500
        options=60<TSO4,TSO6>
        ether 12:00:ed:22:31:00
        media: autoselect <full-duplex>
        status: inactive
en4: flags=8963<UP,BROADCAST,SMART,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1500
        options=60<TSO4,TSO6>
        ether 12:00:ed:22:31:05
        media: autoselect <full-duplex>
        status: inactive
en2: flags=8963<UP,BROADCAST,SMART,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1500
        options=60<TSO4,TSO6>
```

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:

```
william@ubuntu:~/lab4$ sudo ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: ens33
Number of packets: 10
Filter expression: ip

Packet number 1:
       From: 104.39.142.37
         To: 8.8.8.8
   Protocol: ICMP

Packet number 2:
       From: 8.8.8.8
         To: 104.39.142.37
   Protocol: ICMP

Packet number 3:
       From: 104.39.142.37
         To: 8.8.8.8
   Protocol: ICMP

Packet number 4:
       From: 8.8.8.8
         To: 104.39.142.37
   Protocol: ICMP

Packet number 5:
       From: 104.39.142.37
         To: 8.8.8.8
   Protocol: ICMP

Packet number 6:
       From: 8.8.8.8
         To: 104.39.142.37
   Protocol: ICMP

Packet number 7:
       From: 104.39.142.37
         To: 8.8.8.8
   Protocol: ICMP

Packet number 8:
       From: 8.8.8.8
         To: 104.39.142.37
   Protocol: ICMP
```

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:

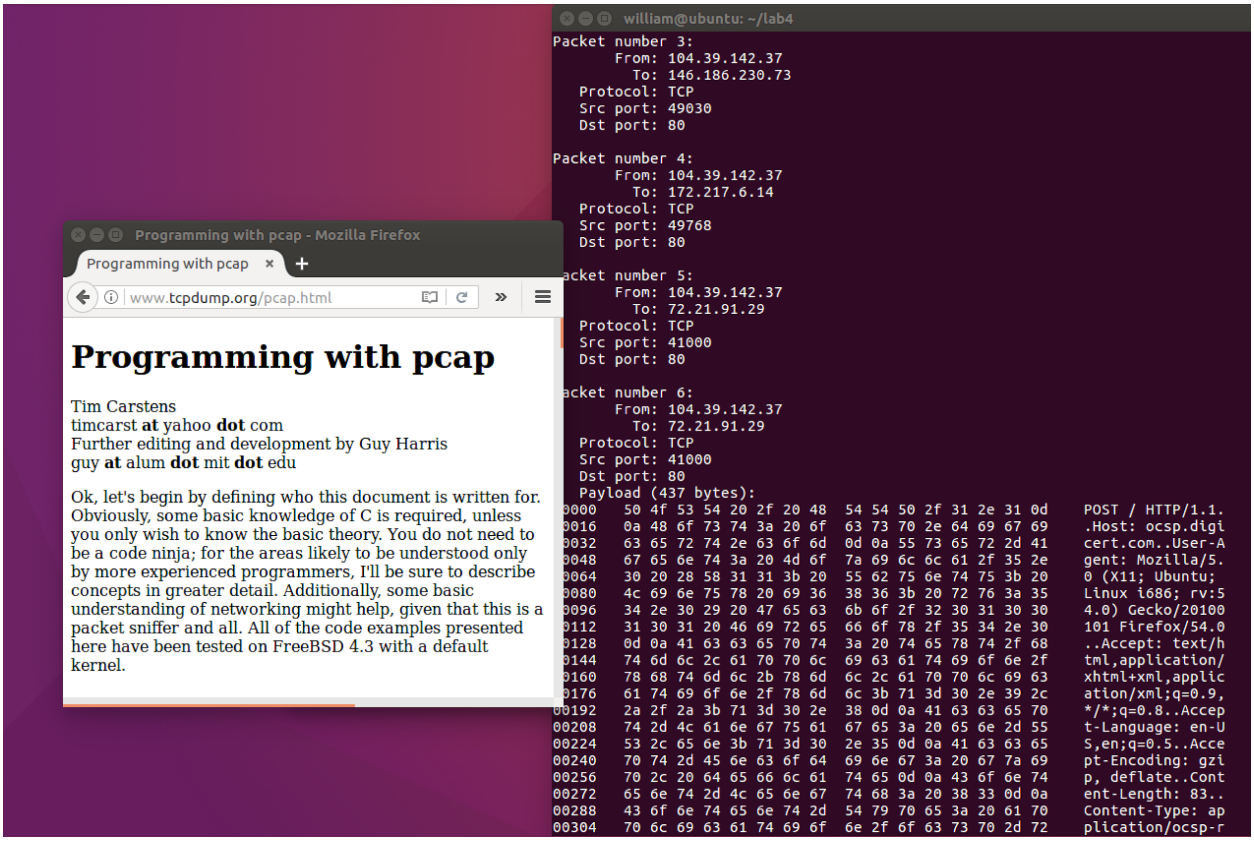


Task 1.b (2)

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:

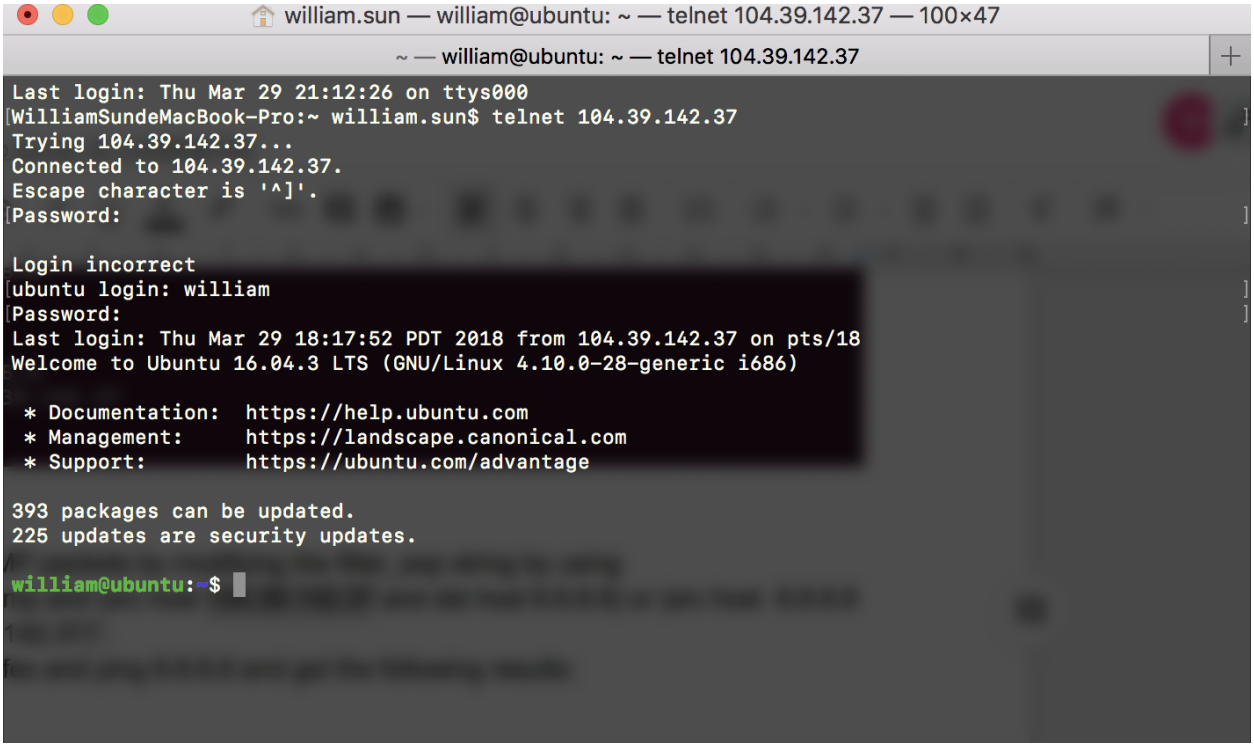


Task 1.c

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.



```
Device: ens33
Number of packets: 10
Filter expression: tcp port 23

Packet number 1:
       From: 104.39.76.237
         To: 104.39.142.37
   Protocol: TCP
   Src port: 49456
   Dst port: 23

Packet number 2:
       From: 104.39.142.37
         To: 104.39.76.237
   Protocol: TCP
   Src port: 23
   Dst port: 49456

Packet number 3:
       From: 104.39.76.237
         To: 104.39.142.37
   Protocol: TCP
   Src port: 49456
   Dst port: 23

Packet number 4:
       From: 104.39.76.237
         To: 104.39.142.37
   Protocol: TCP
   Src port: 49456
   Dst port: 23
   Payload (27 bytes):
00000   ff fb 25 ff fd 03 ff fb  18 ff fb 1f ff fb 20 ff    ..%.......... .
00016   fb 21 ff fb 22 ff fb 27  ff fd 05                   .!.."..'...

Packet number 5:
       From: 104.39.142.37
         To: 104.39.76.237
   Protocol: TCP
   Src port: 23
   Dst port: 49456

Packet number 6:
       From: 104.39.142.37
         To: 104.39.76.237
   Protocol: TCP
   Src port: 23
   Dst port: 49456
   Payload (12 bytes):
00000   ff fd 18 ff fd 20 ff fd  23 ff fd 27               ..... ..#..'
```

```
Packet number 6:
       From: 104.39.142.37
         To: 104.39.76.237
   Protocol: TCP
   Src port: 23
   Dst port: 49456
   Payload (12 bytes):
00000   ff fd 18 ff fd 20 ff fd  23 ff fd 27               ..... ..#..'

Packet number 7:
       From: 104.39.76.237
         To: 104.39.142.37
   Protocol: TCP
   Src port: 49456
   Dst port: 23

Packet number 8:
       From: 104.39.76.237
         To: 104.39.142.37
   Protocol: TCP
   Src port: 49456
   Dst port: 23
   Payload (3 bytes):
00000   ff fc 23                                           ..#

Packet number 9:
       From: 104.39.142.37
         To: 104.39.76.237
   Protocol: TCP
   Src port: 23
   Dst port: 49456
   Payload (36 bytes):
00000   ff fe 25 ff fb 03 ff fd  1f ff fd 21 ff fe 22 ff    ..%........!..".
00016   fb 05 ff fa 20 01 ff f0  ff fa 27 01 ff f0 ff fa    .... .....'.....
00032   18 01 ff f0                                        ....

Packet number 10:
       From: 104.39.76.237
         To: 104.39.142.37
   Protocol: TCP
   Src port: 49456
   Dst port: 23

Capture complete.
william@ubuntu:~/lab4$
```

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.

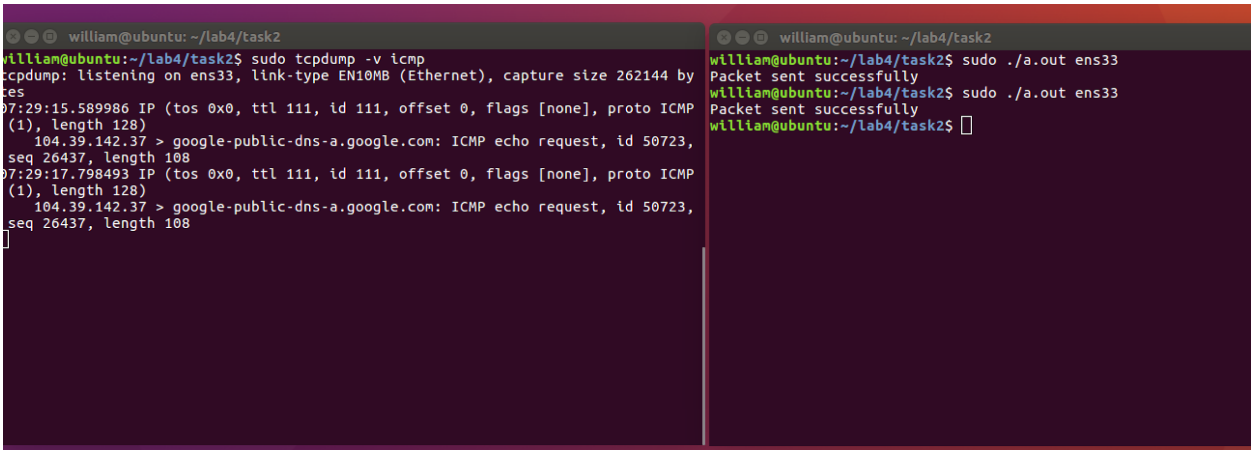
Question 5:
Yes since when we use the original socket programming in c, we must manually calculate the checksum, otherwise the error will be received. If we use the python scapy package, then it has a built-in function for us which is .chksum.

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:

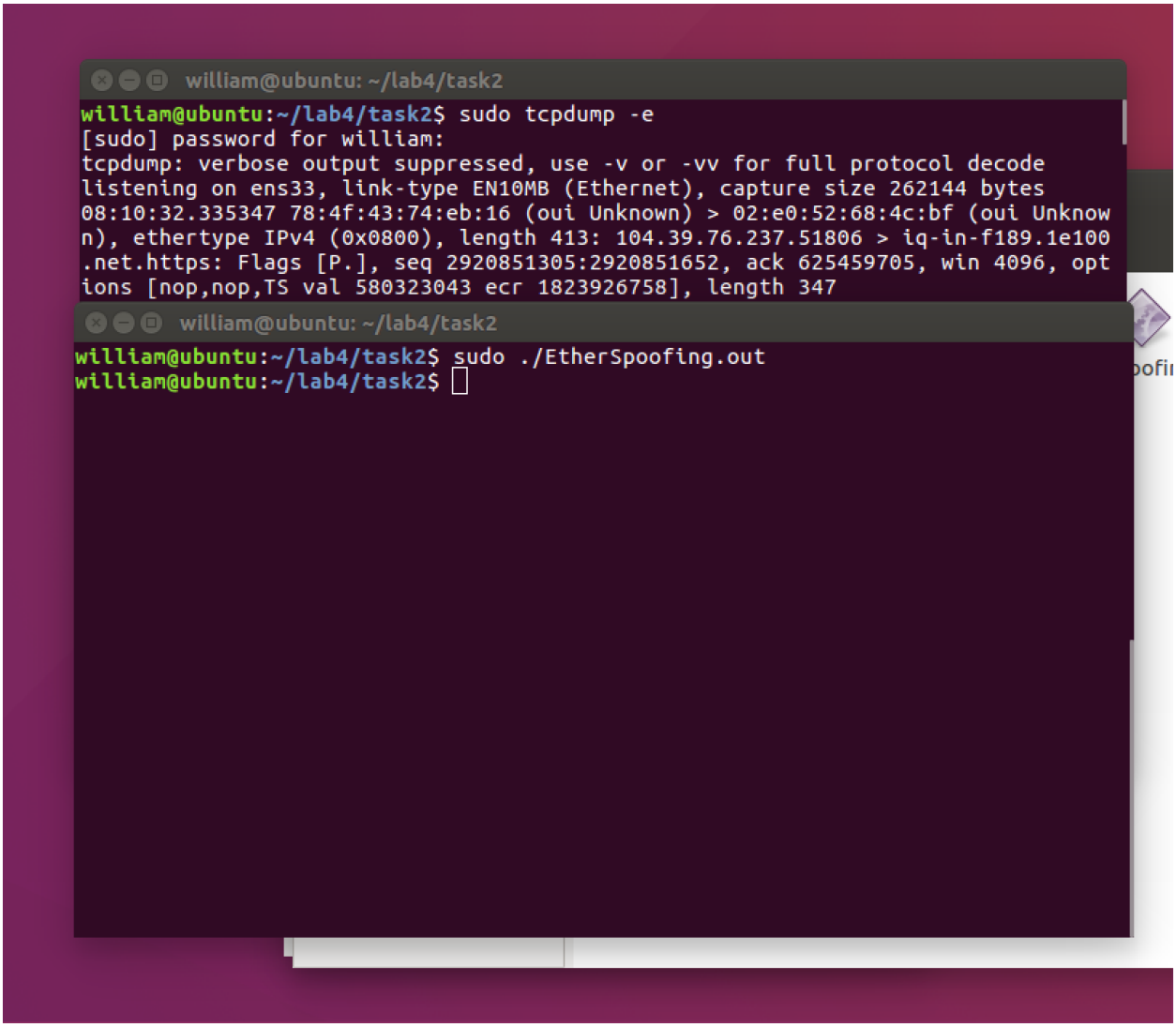


Task2 (b)

```
yanbo@yanbo-ThinkPad-Edge:~/443/lab4/task2$ sudo tcpdump -v icmp
tcpdump: listening on wlp3s0, link-type EN10MB (Ethernet), capture size 262144 bytes
11:01:17.969378 IP (tos 0x60, ttl 111, id 111, offset 0, flags [none], proto ICMP (1), length 128)
    104.39.142.37 > 104.39.120.152: ICMP echo request, id 50723, seq 26437, length 108
11:01:17.975274 IP (tos 0x60, ttl 64, id 19515, offset 0, flags [none], proto ICMP (1), length 128)
    104.39.120.152 > 104.39.142.37: ICMP echo reply, id 50723, seq 26437, length 108
11:01:37.790066 IP (tos 0x60, ttl 111, id 111, offset 0, flags [none], proto ICMP (1), length 128)
    104.39.142.37 > 104.39.120.152: ICMP echo request, id 50723, seq 26437, length 108
11:01:37.790117 IP (tos 0x60, ttl 64, id 21824, offset 0, flags [none], proto ICMP (1), length 128)
    104.39.120.152 > 104.39.142.37: ICMP echo reply, id 50723, seq 26437, length 108
11:01:39.732598 IP (tos 0x60, ttl 111, id 111, offset 0, flags [none], proto ICMP (1), length 128)
    104.39.142.37 > 104.39.120.152: ICMP echo request, id 50723, seq 26437, length 108
11:01:39.732645 IP (tos 0x60, ttl 64, id 21855, offset 0, flags [none], proto ICMP (1), length 128)
    104.39.120.152 > 104.39.142.37: ICMP echo reply, id 50723, seq 26437, length 108
11:01:40.887813 IP (tos 0x60, ttl 111, id 111, offset 0, flags [none], proto ICMP (1), length 128)
    104.39.142.37 > 104.39.120.152: ICMP echo request, id 50723, seq 26437, length 108
11:01:40.887858 IP (tos 0x60, ttl 64, id 22120, offset 0, flags [none], proto ICMP (1), length 128)
    104.39.120.152 > 104.39.142.37: ICMP echo reply, id 50723, seq 26437, length 108
```

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)



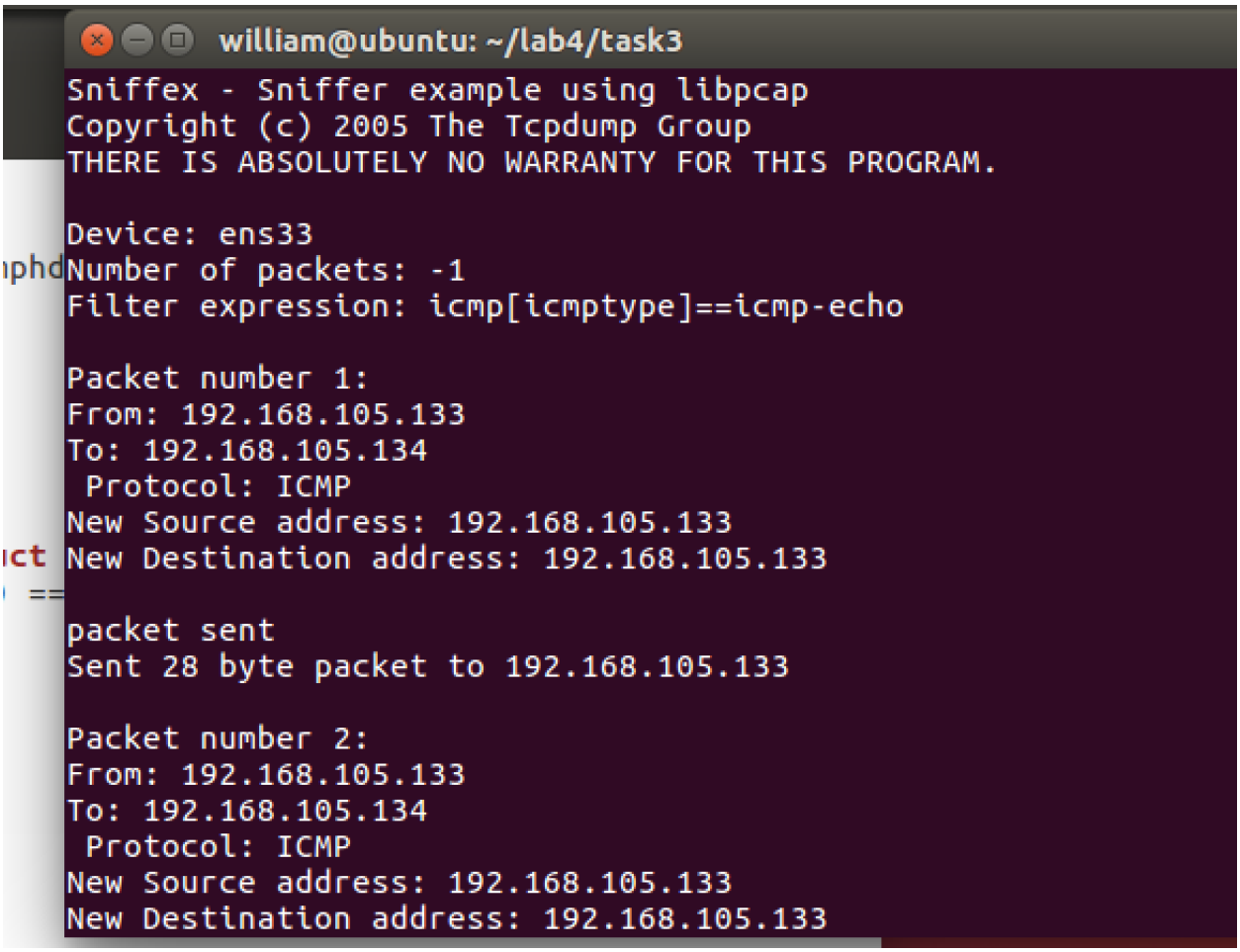
The source code is compiled and tested in the system. We successfully got the frames by tcpdump.

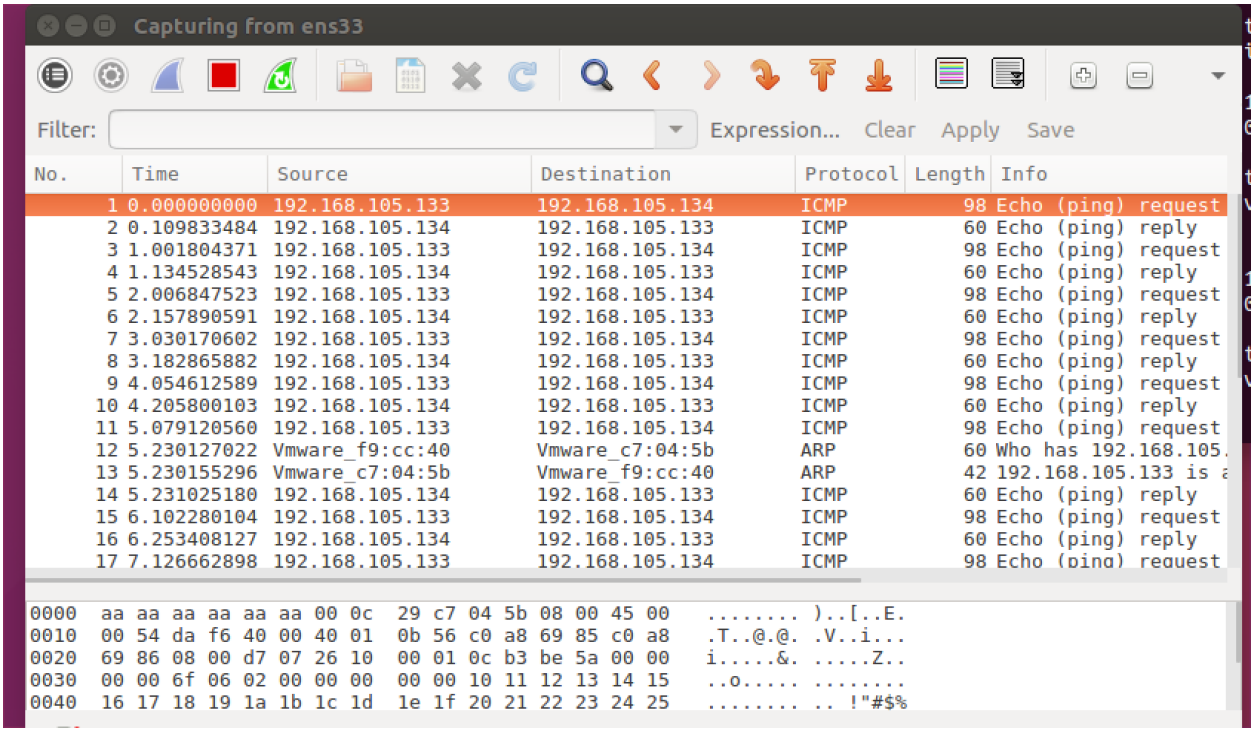
Task 3

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;
}
```