Shreyas Prabhakar
15th February 2017

CPSC 8570: SECURITY IN ADVANCED NETWORKING TECHNOLOGIES

PROJECT 1

SNIFFING AND SPOOFING

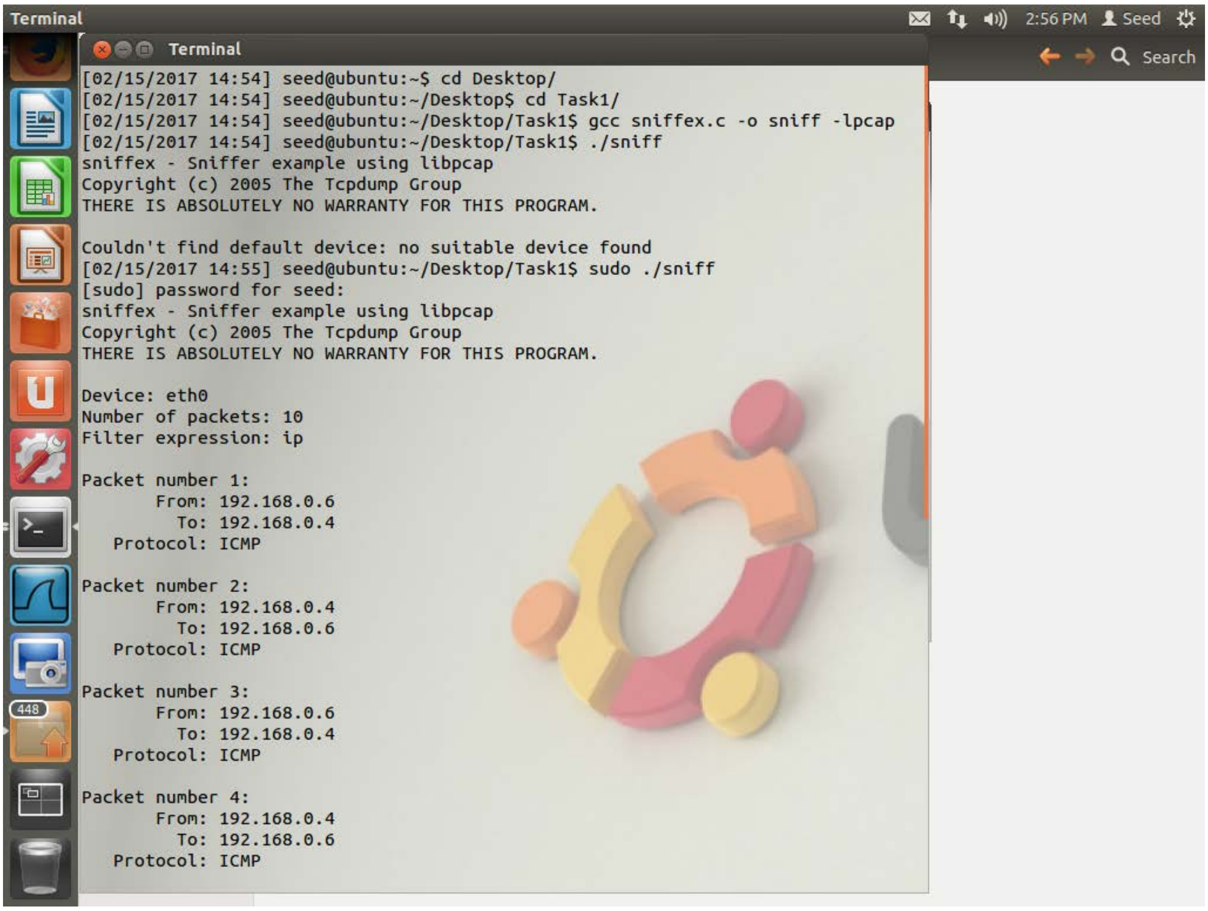
TASK 1 – SNIFFING

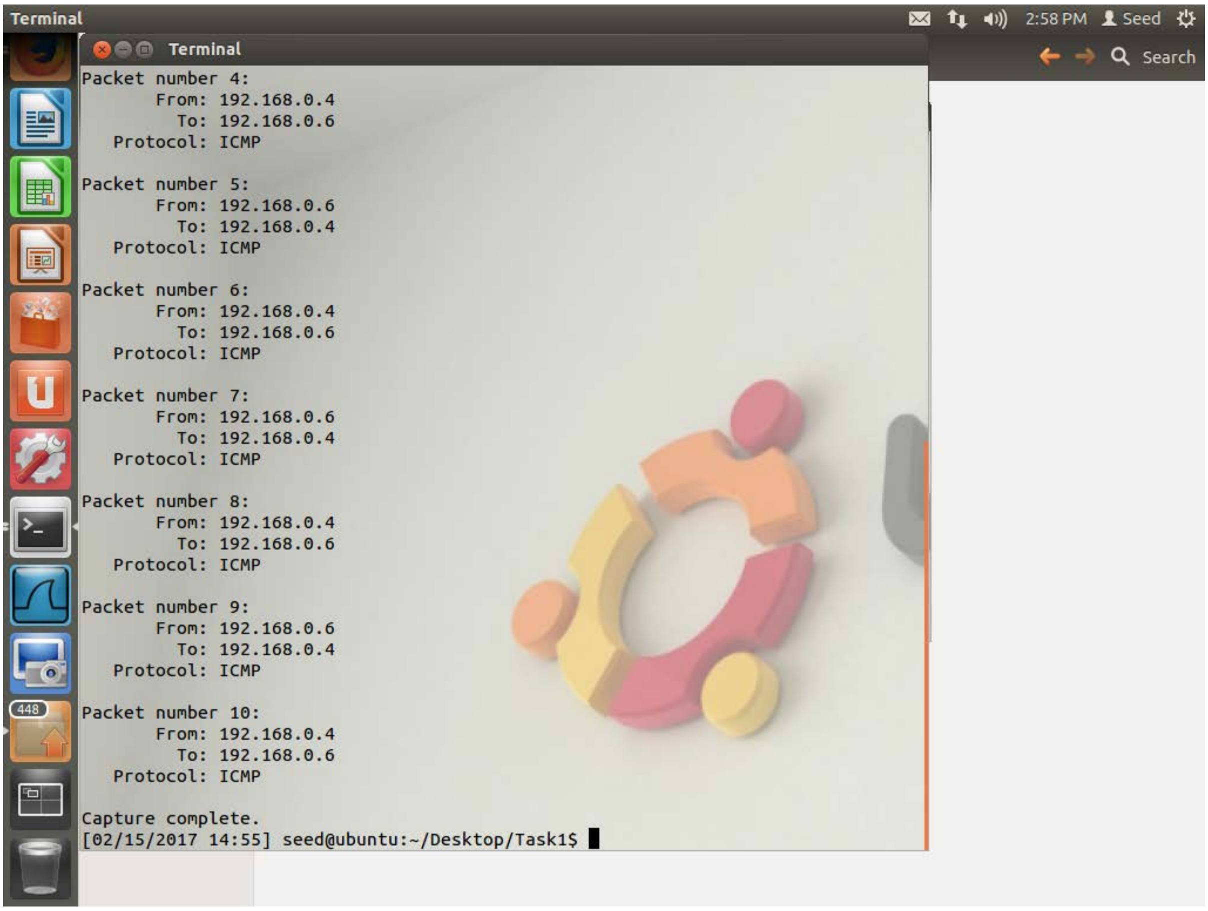
Task 1. a. Understanding Sniffex

The code was compiled using the command line code

gcc sniffex.c -o sniff -lpcap

When we try to run the executable file by giving ./sniff on the command line, the program return with an error – "Couldn't find default device: no suitable device found". We need to rum the program with root privileges.





From the above screen dumps we can see that the sniffex.c compiles successfully. The program successfully runs with root privileges but fails to run without the root privileges.

Problem 1: Please use your own words to describe the sequence of the library calls that are essential for sniffer programs. This is meant to be a summary, not detailed explanation like the one in the tutorial.

1. Setting Up the device - We have two techniques for setting up the device
    1. The user specifies the device by passing the name of the device as the first argument to the program. The character string 'dev' holds the name of the interface that we want to sniff on.
    2. pcap sets the device on its own.
2. Opening the device for sniffing

    We use pcap_open_live() to create a sniffing session.

    *pcap_t *pcap_open_live(char *device, int snaplen, int promisc, int to_ms, char *ebuf)*

    - device is the device that we have set up
    - snaplen defines the maximum number of bytes to be captured by pcap.
    - promisc sets the interface to promiscuous mode, if this field is set to true.
    - to_ms is the read time out in milliseconds. We should use a non-zero timeout, so that it doesn't wait until sufficient number of packets are received before seeing at other packets.
    - ebuf can store any error messages.
    - This function returns our session handler

    Not all devices provide the same type of link-layer headers in the packets we read. We need to determine the type of link-layer headers the device provides, and use that type when processing the packet contents. The pcap_datalink() routine returns a value indicating the type of link-layer headers.

3. Filtering Traffic
   Often times our sniffer may be interested in only specific type of traffic. After we have already called the pcap_open_live() and have a working sniffing session, we can apply our filter.
   Before we apply our filter, we must 'compile' it. To compile the program, we call pcap_compile()
   *int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize, bpf_u_int32 netmask)*
      - p is the session handle which was returned by pcap_open_live() routine
      - fp is a reference to the place we will store the compiled version of our filter
      - str is the filter expression
      - optimize decides if the expression should be optimized or not. 0 is false. 1 is true
      - netmask specifies the network mask of the network the filter applies to.
      - Function return -1 on failure. All other values imply success

   After the expression has been compiled, we apply it using pcap_setfilter()
   int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
      - p is the session handle which was returned by pcap_open_live() routine
      - fp is a reference to the place we will store the compiled version of our filter
4. The actual sniffing - We have two techniques for capturing packets
   1. Capture a single packet at a time. For this we use pcap_next()
      u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
         - p is the session handle which was returned by pcap_open_live() routine
         - h is a pointer to a structure that holds general information about the packet, specifically the time in which it was sniffed, the length of this packet, and the length of its specific portion
   2. Enter a loop that waits for n number of packets to be sniffed before being done. This can be done by using call back functions pcap_loop() and pcap_dispatch().
      int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
         - p is the session handle which was returned by pcap_open_live() routine
         - cnt tells pcap_loop() how many packets it should sniff for before returning.
         - Callback is the name of the callback function
         - user is useful in some applications, but many times is simply set as NULL. Suppose we have arguments of our own that we wish to send to our callback function, in addition to the arguments that pcap_loop() sends. This is where we do it.

      pcap_dispatch() is almost identical in usage. The only difference between pcap_dispatch() and pcap_loop() is that pcap_dispatch() will only process the first batch of packets that it receives from the system, while pcap_loop() will continue processing packets or batches of packets until the count of packets runs out.

Problem 2: Why do you need the root privilege to run sniffex? Where does the program fail if executed without the root privilege?

When it comes to accessing devices, files and other resources provided by the operating system, the access models are implemented in the operating system and APIs are provided for the users. The users are expected to use whatever is made available to the them by the operating system and accept any error condition that is thrown at them when they try to access the resources that are restricted by the operating system.

For the sniffex.c program, it compiles successfully, but it fails to run successfully without the administrative privileges, as the part of code which is required for the setting up of the interface device is blocked by the operating system as this operation is allowed only for those with administrative access

The program fails at dev = pcap_loopupdev(errbuf); in the following excerpt of the sniffex.c code

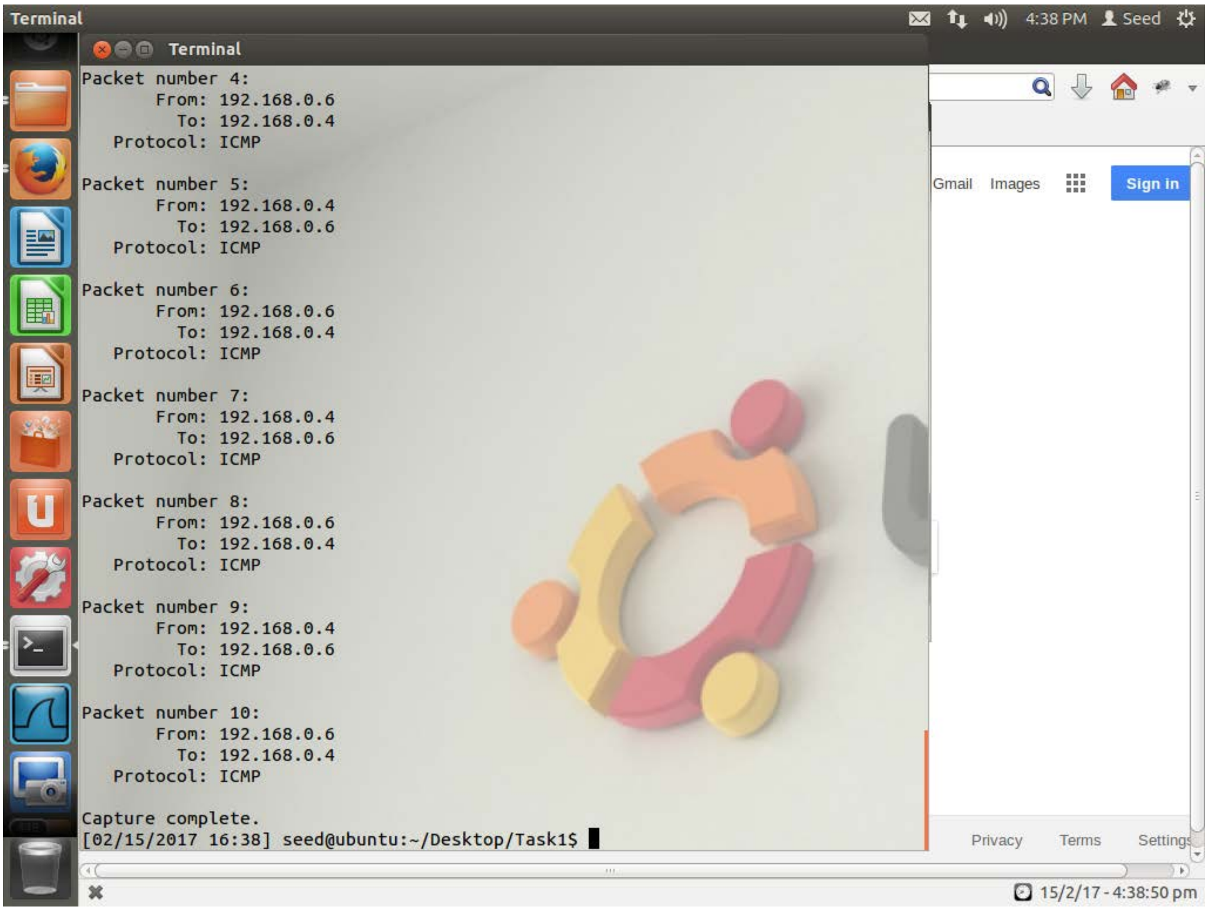
```
else {
                /* find a capture device if not specified on
command-line */
                dev = pcap_lookupdev(errbuf);
                if (dev == NULL) {
                        fprintf(stderr, "Couldn't find default
device: %s\n",
                            errbuf);
                        exit(EXIT_FAILURE);
                }
```

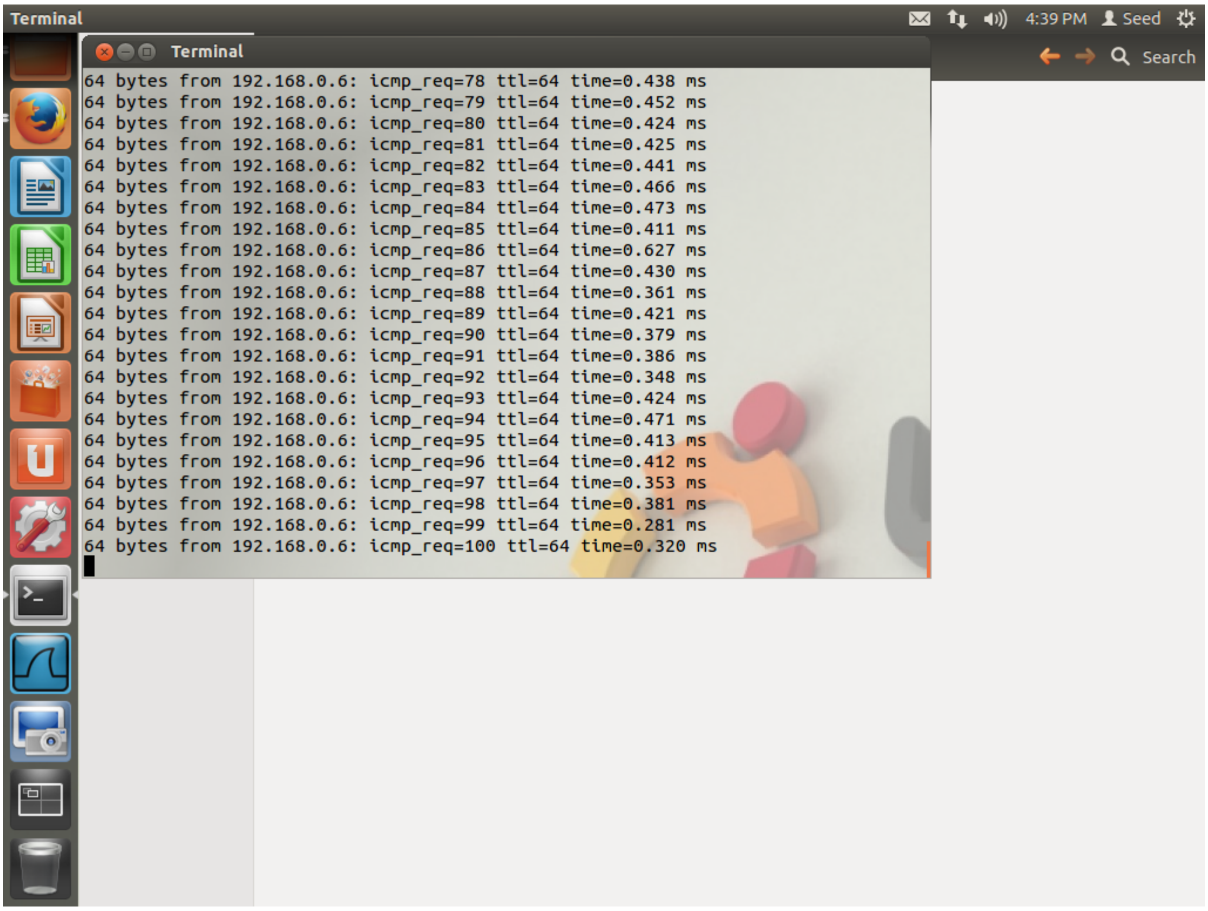
Task 1.b. Writing Filters

Please write filter expressions to capture each of the followings. In your lab reports, you need to include screen dumps to show the results of applying each of these filters.

- Capture the ICMP packets between two specific hosts.

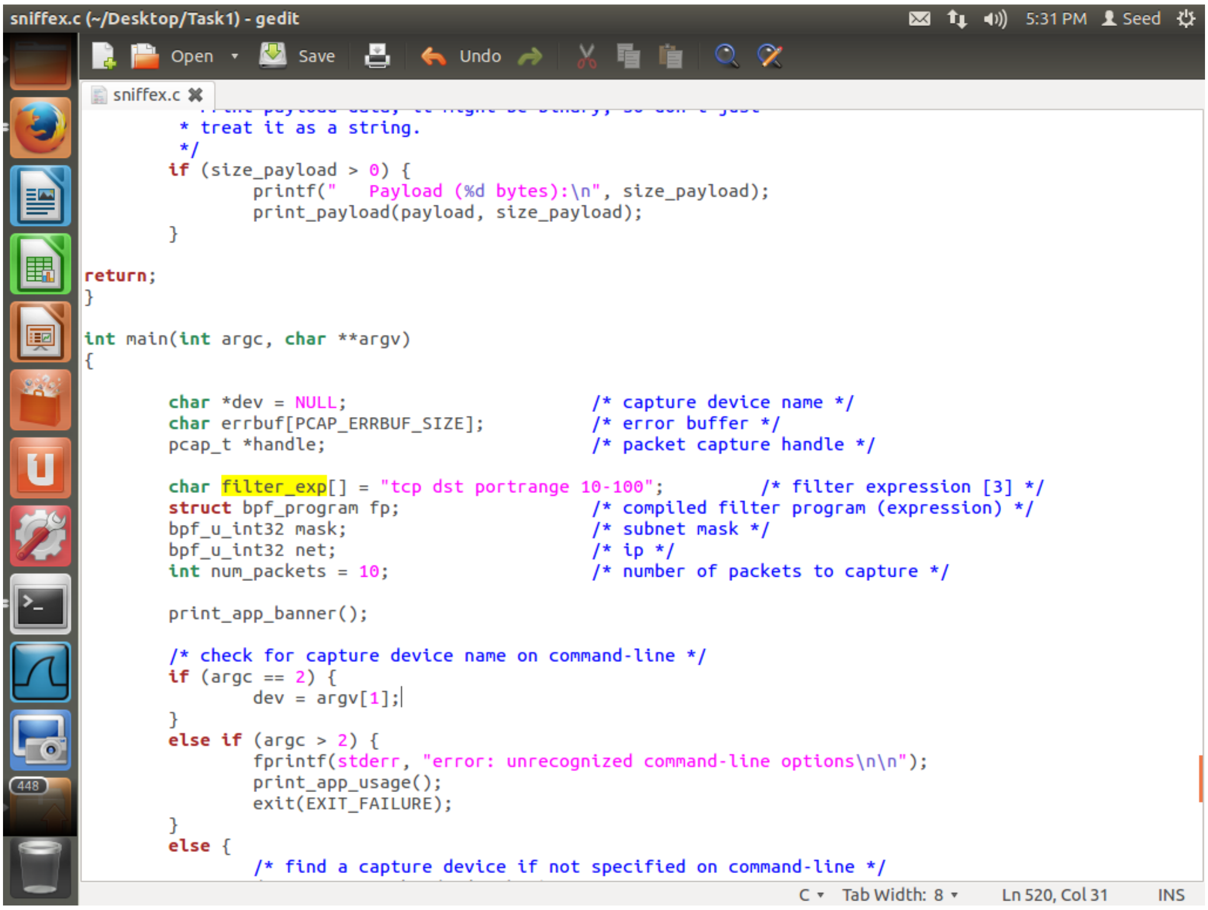
For this experiment, I ran sniffex program on VM1 (192.168.0.6) and pinged VM1 from VM2 (192.168.0.4). The sniffex program running on VM1 captures both the ping request from VM1, and ping echo to VM2.





- Capture the TCP packets that have a destination port range from to port 10 - 100.

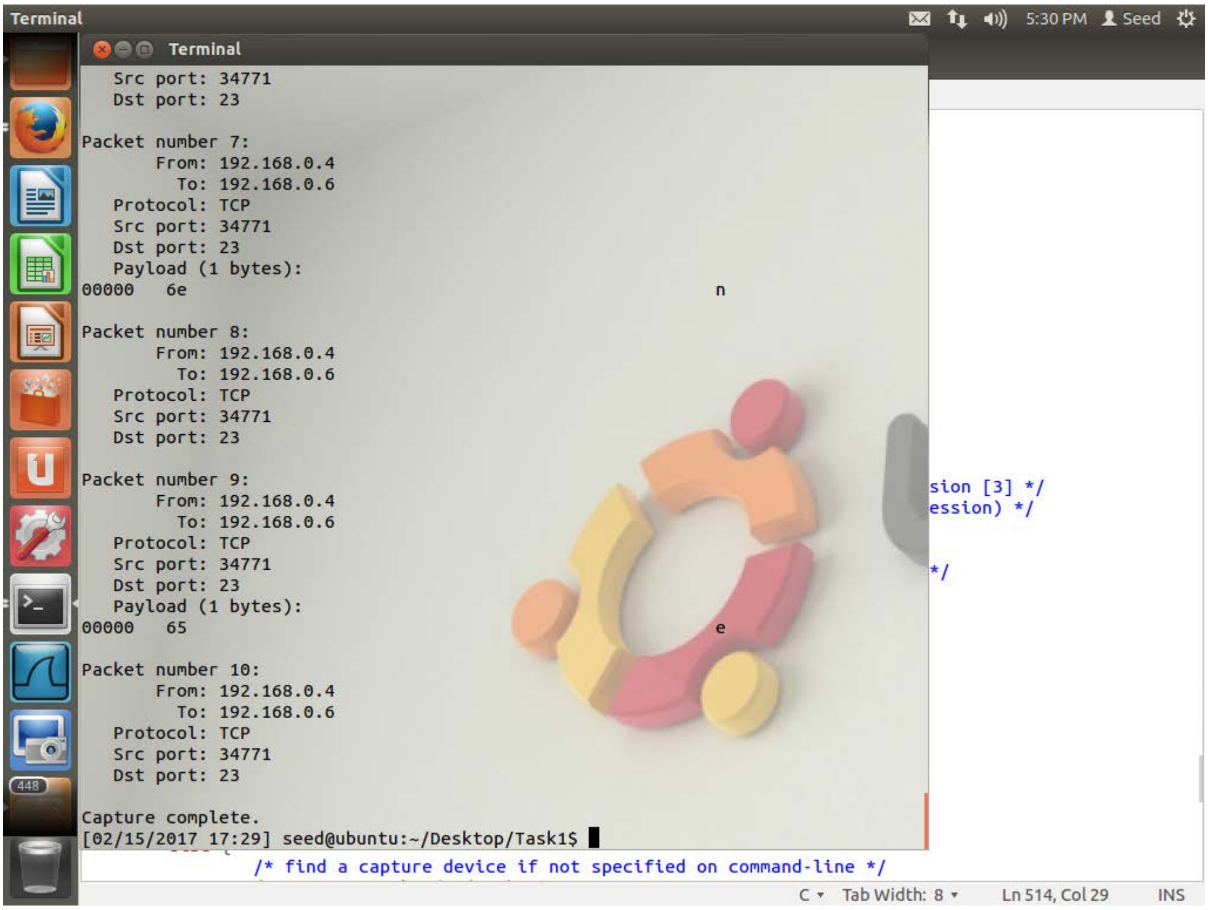
  For this experiment, I changed the filter_exp to 'tcp dst portrange 10-100', saved, compiled and ran the program on VM1 (192.168.0.6). I ran telnet to VM1 from VM2 (192.168.0.4). The sniffex program running on VM1 captured all the TCP packets reaching to and originating from VM1.



sniffex.c (~/Desktop/Task1) - gedit

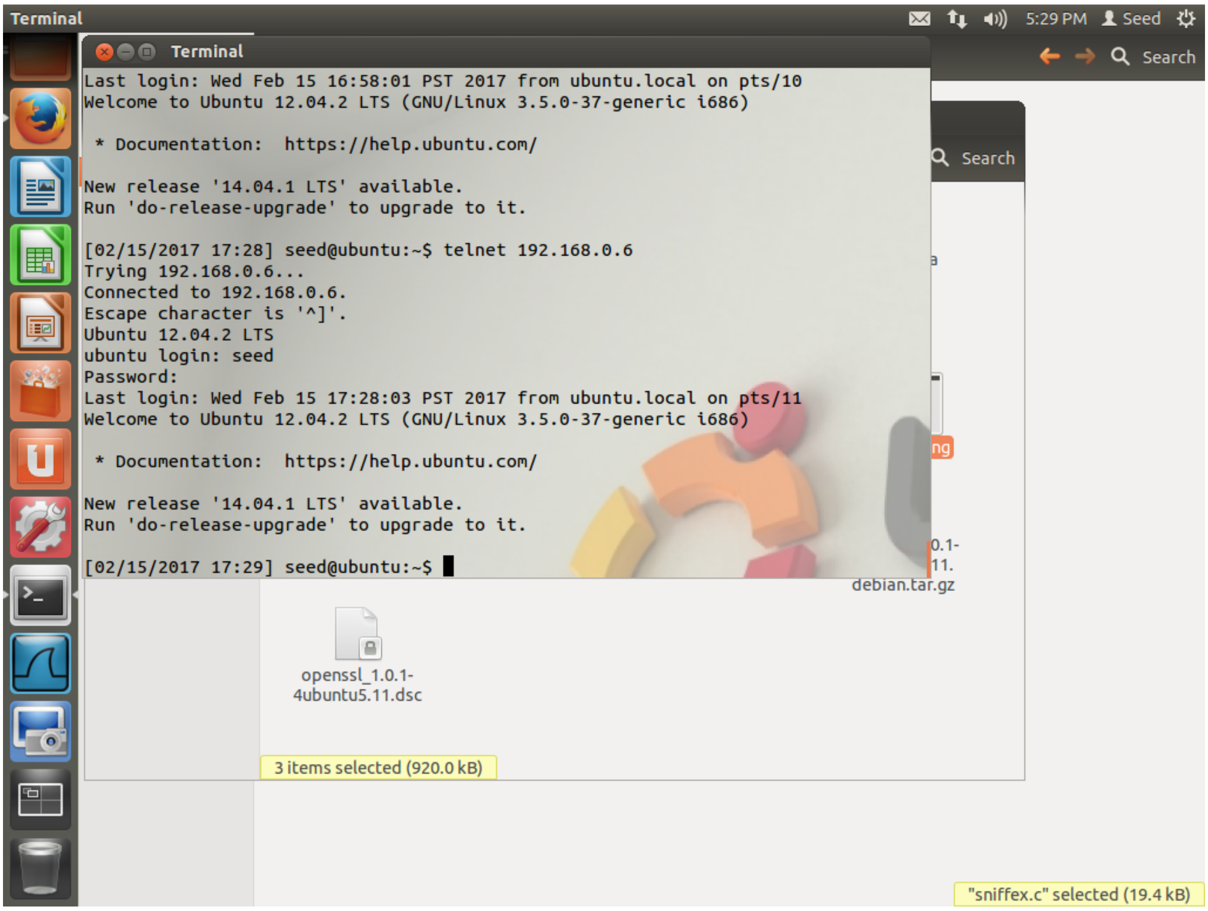
```
         * treat it as a string.
         */
        if (size_payload > 0) {
                printf("   Payload (%d bytes):\n", size_payload);
                print_payload(payload, size_payload);
        }

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[] = "tcp dst portrange 10-100";         /* 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 */
```

C ▾ Tab Width: 8 ▾ Ln 520, Col 31 INS



Terminal

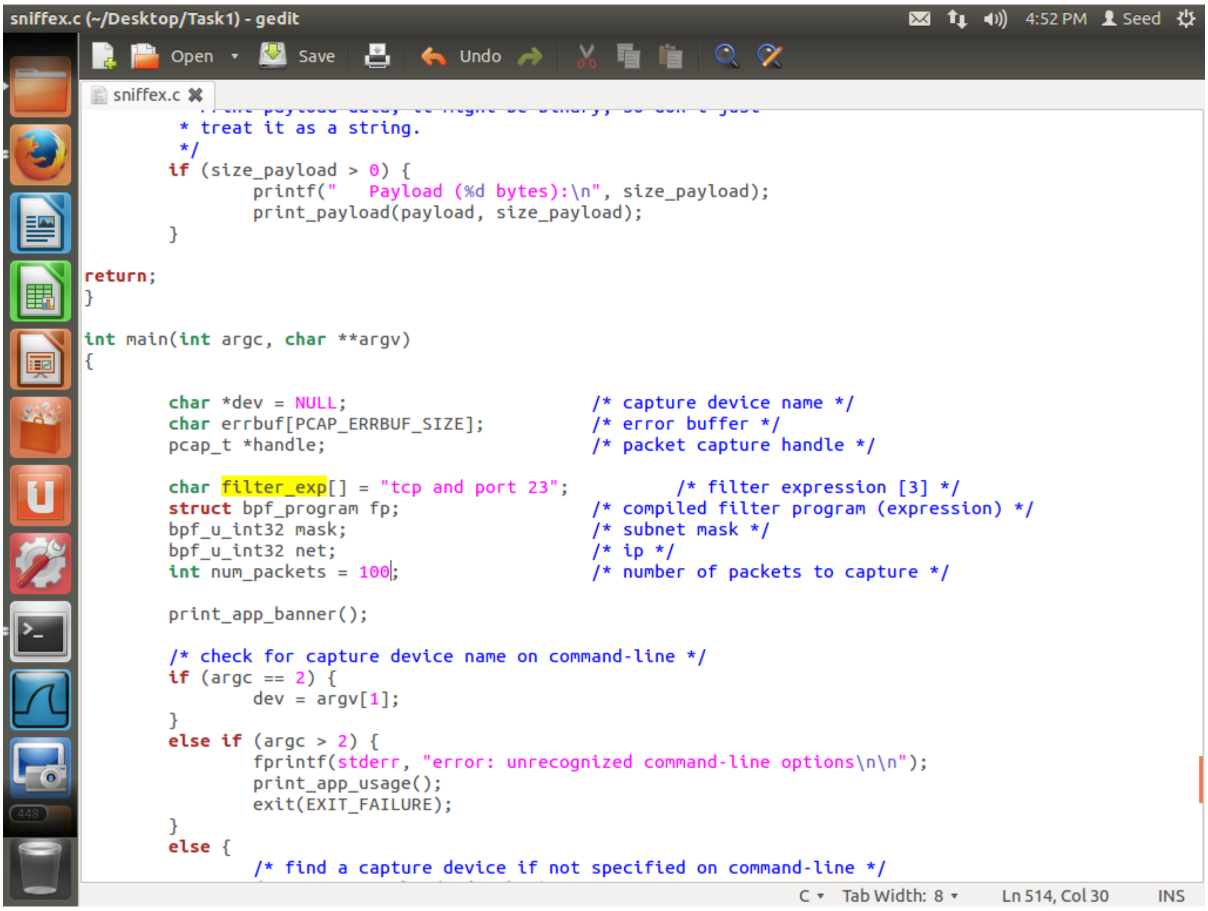
```
    Src port: 34771
    Dst port: 23

Packet number 7:
        From: 192.168.0.4
          To: 192.168.0.6
    Protocol: TCP
    Src port: 34771
    Dst port: 23
    Payload (1 bytes):
00000   6e                                                      n

Packet number 8:
        From: 192.168.0.4
          To: 192.168.0.6
    Protocol: TCP
    Src port: 34771
    Dst port: 23

Packet number 9:
        From: 192.168.0.4
          To: 192.168.0.6
    Protocol: TCP
    Src port: 34771
    Dst port: 23
    Payload (1 bytes):
00000   65                                                      e

Packet number 10:
        From: 192.168.0.4
          To: 192.168.0.6
    Protocol: TCP
    Src port: 34771
    Dst port: 23

Capture complete.
[02/15/2017 17:29] seed@ubuntu:~/Desktop/Task1$
```



Task 1.c. Sniffing Passwords

Please show how you can use sniffex to capture the password when somebody is using telnet on the network that you are monitoring.

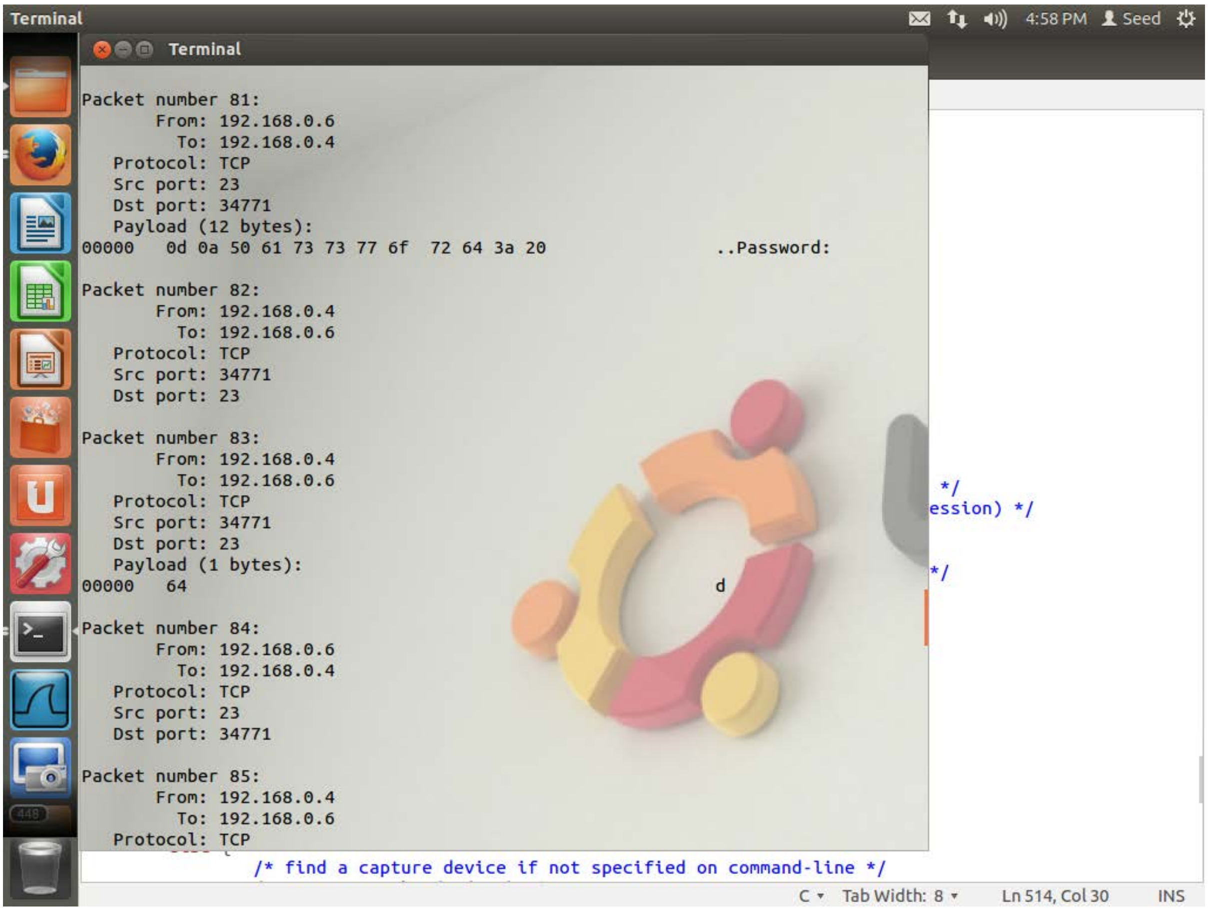
For this experiment, the filter_exp[] is changed to “tcp and port 23” in the sniffex.c program, compiled and run in VM1 (192.168.0.6). Telnet is run on form VM2 (192.168.0.4) to VM1. All the communication between VM1 and VM2 is captured by our sniffer program. Our password ‘dees’ is captured in the packets 81-89 as shown in the screen dump



sniffex.c (~/Desktop/Task1) - gedit

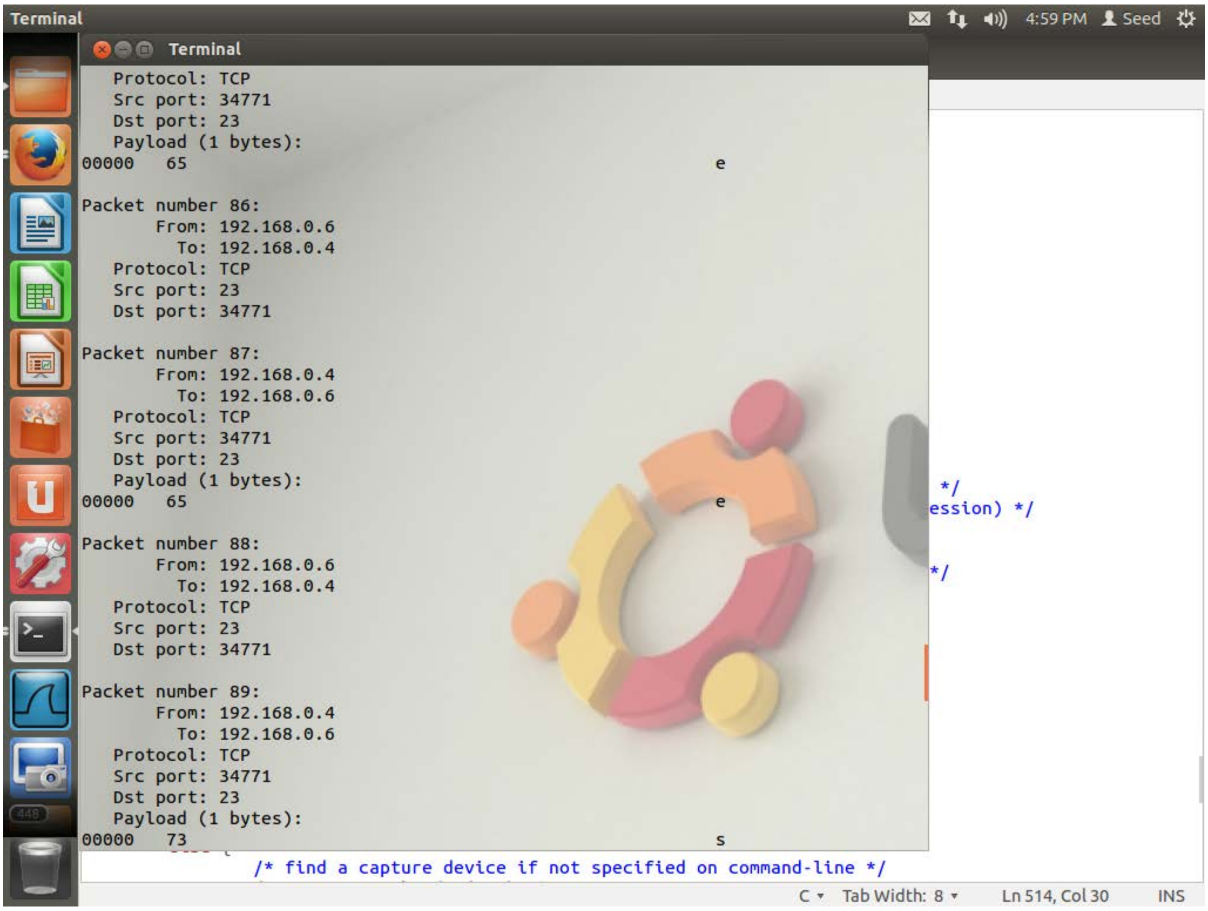
```c
	 * treat it as a string.
	 */
	if (size_payload > 0) {
		printf("   Payload (%d bytes):\n", size_payload);
		print_payload(payload, size_payload);
	}

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[] = "tcp and port 23";		/* 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 = 100;			/* 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 */
```

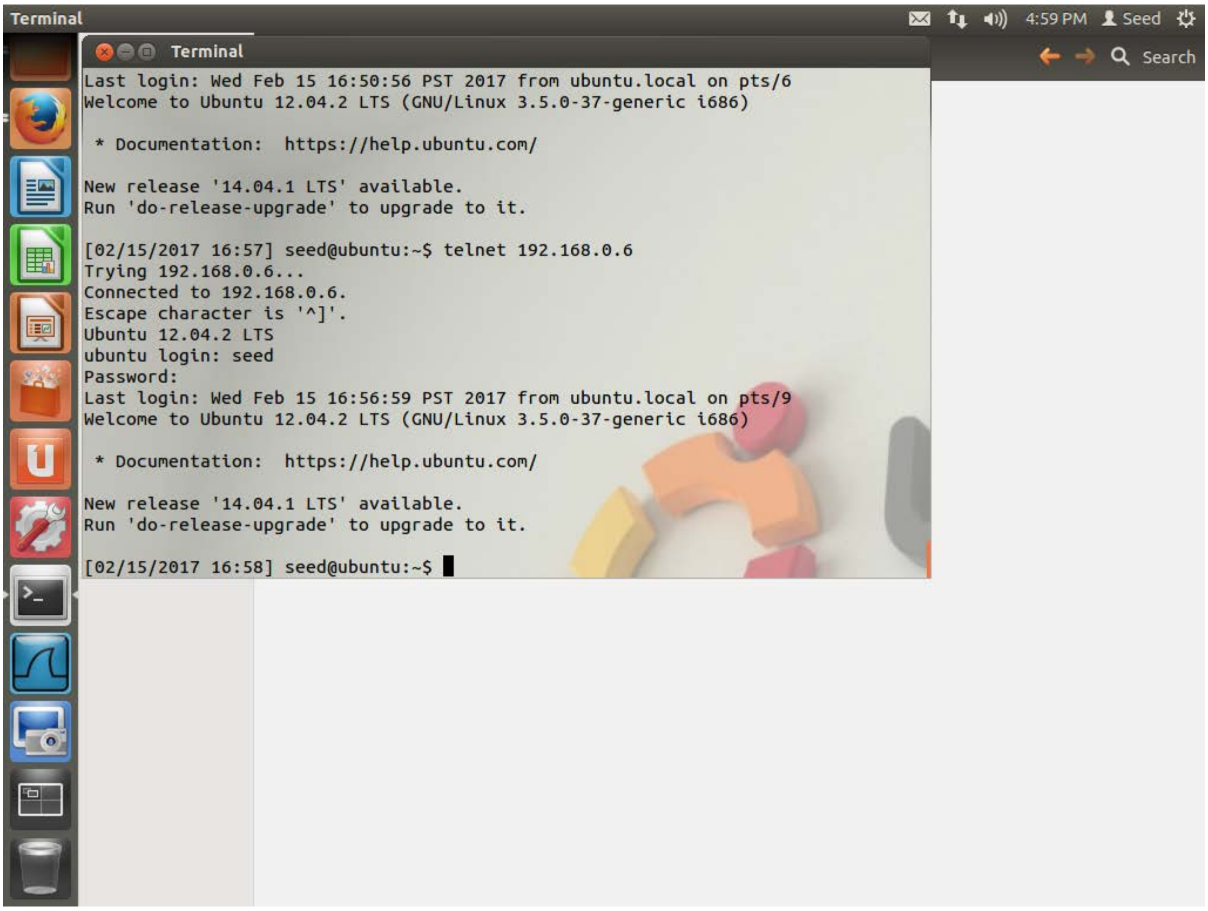
C Tab Width: 8 Ln 514, Col 30 INS

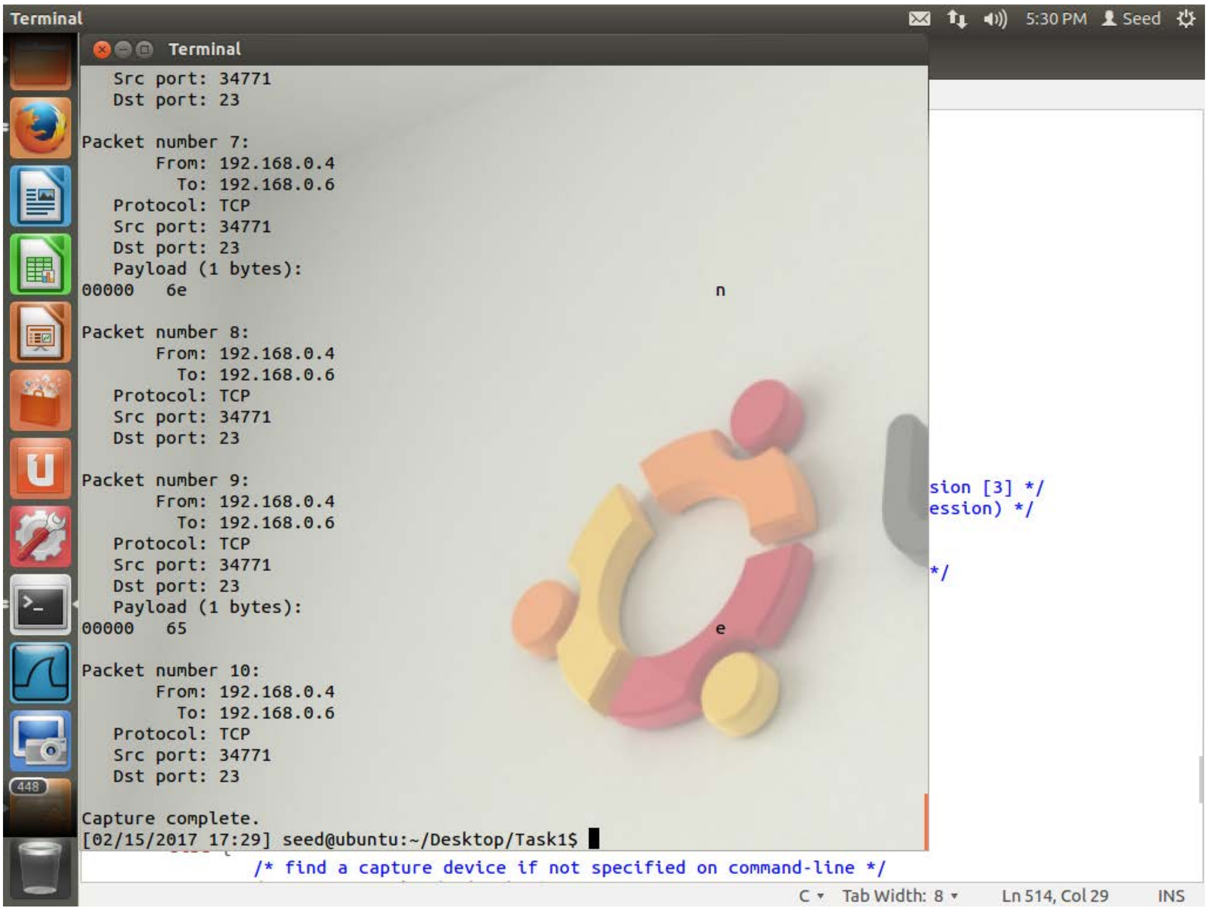


Terminal

```
Packet number 81:
       From: 192.168.0.6
         To: 192.168.0.4
   Protocol: TCP
   Src port: 23
   Dst port: 34771
   Payload (12 bytes):
00000   0d 0a 50 61 73 73 77 6f  72 64 3a 20              ..Password:

Packet number 82:
       From: 192.168.0.4
         To: 192.168.0.6
   Protocol: TCP
   Src port: 34771
   Dst port: 23

Packet number 83:
       From: 192.168.0.4
         To: 192.168.0.6
   Protocol: TCP
   Src port: 34771
   Dst port: 23
   Payload (1 bytes):
00000   64                                                d

Packet number 84:
       From: 192.168.0.6
         To: 192.168.0.4
   Protocol: TCP
   Src port: 23
   Dst port: 34771

Packet number 85:
       From: 192.168.0.4
         To: 192.168.0.6
   Protocol: TCP
```


Terminal
4:59 PM Seed
Terminal
Protocol: TCP
Src port: 34771
Dst port: 23
Payload (1 bytes):
00000 65 e
Packet number 86:
From: 192.168.0.6
To: 192.168.0.4
Protocol: TCP
Src port: 23
Dst port: 34771
Packet number 87:
From: 192.168.0.4
To: 192.168.0.6
Protocol: TCP
Src port: 34771
Dst port: 23
Payload (1 bytes):
00000 65 e
Packet number 88:
From: 192.168.0.6
To: 192.168.0.4
Protocol: TCP
Src port: 23
Dst port: 34771
Packet number 89:
From: 192.168.0.4
To: 192.168.0.6
Protocol: TCP
Src port: 34771
Dst port: 23
Payload (1 bytes):
00000 73 s
/* find a capture device if not specified on command-line */
C Tab Width: 8 Ln 514, Col 30 INS


Terminal
4:59 PM Seed
Terminal
Search
Last login: Wed Feb 15 16:50:56 PST 2017 from ubuntu.local on pts/6
Welcome to Ubuntu 12.04.2 LTS (GNU/Linux 3.5.0-37-generic i686)
* Documentation: https://help.ubuntu.com/
New release '14.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.
[02/15/2017 16:57] seed@ubuntu:~$ telnet 192.168.0.6
Trying 192.168.0.6...
Connected to 192.168.0.6.
Escape character is '^]'.
Ubuntu 12.04.2 LTS
ubuntu login: seed
Password:
Last login: Wed Feb 15 16:56:59 PST 2017 from ubuntu.local on pts/9
Welcome to Ubuntu 12.04.2 LTS (GNU/Linux 3.5.0-37-generic i686)
* Documentation: https://help.ubuntu.com/
New release '14.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.
[02/15/2017 16:58] seed@ubuntu:~$



TASK 2: SPOOFING

Task 2.a. Write a spoofing program

You can write your own packet program or download one. You need to provide evidences (e.g., Wireshark packet trace) to show us that your program successfully sends out spoofed IP packets

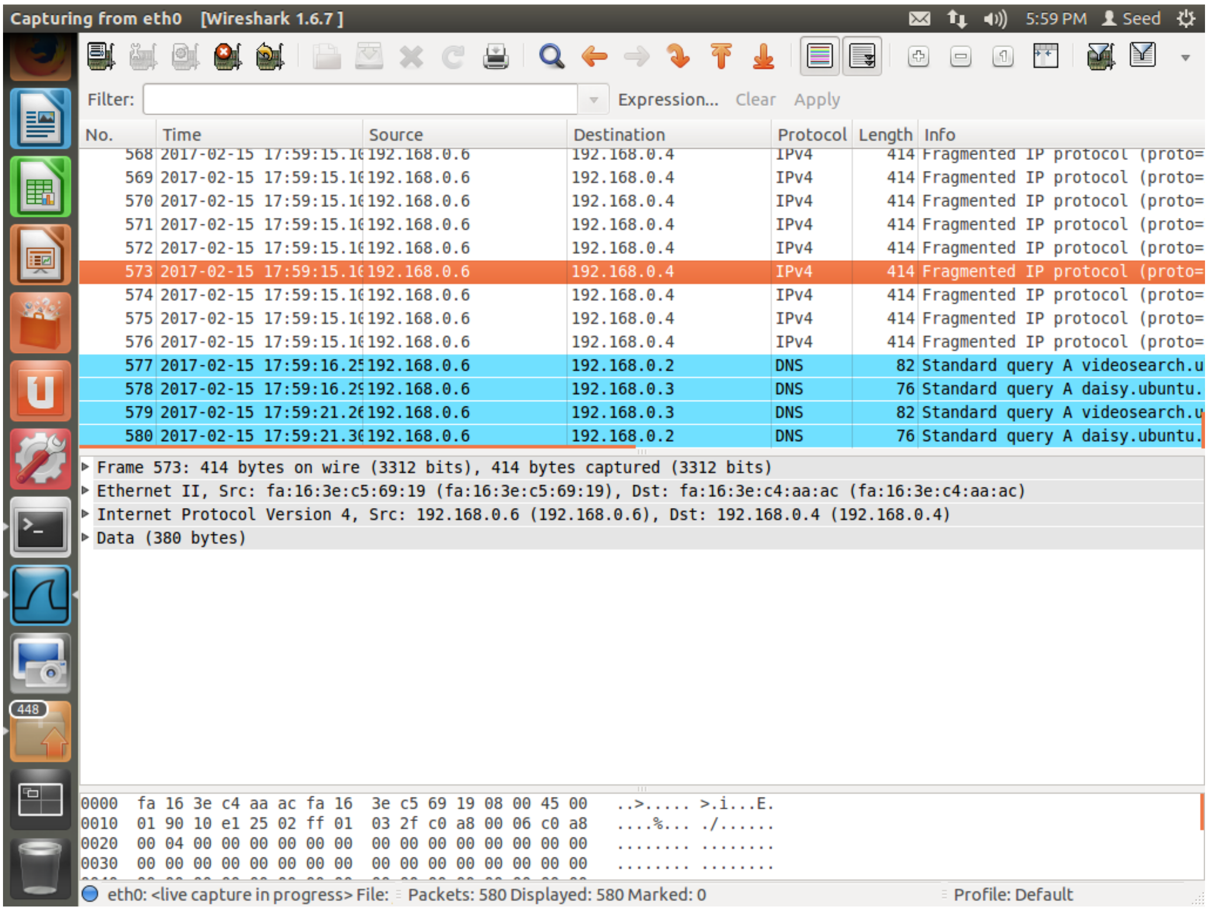
The spoofing program was downloaded from http://www.tenouk.com/Module43a.html. This program is compiled on VM1 using the command line code

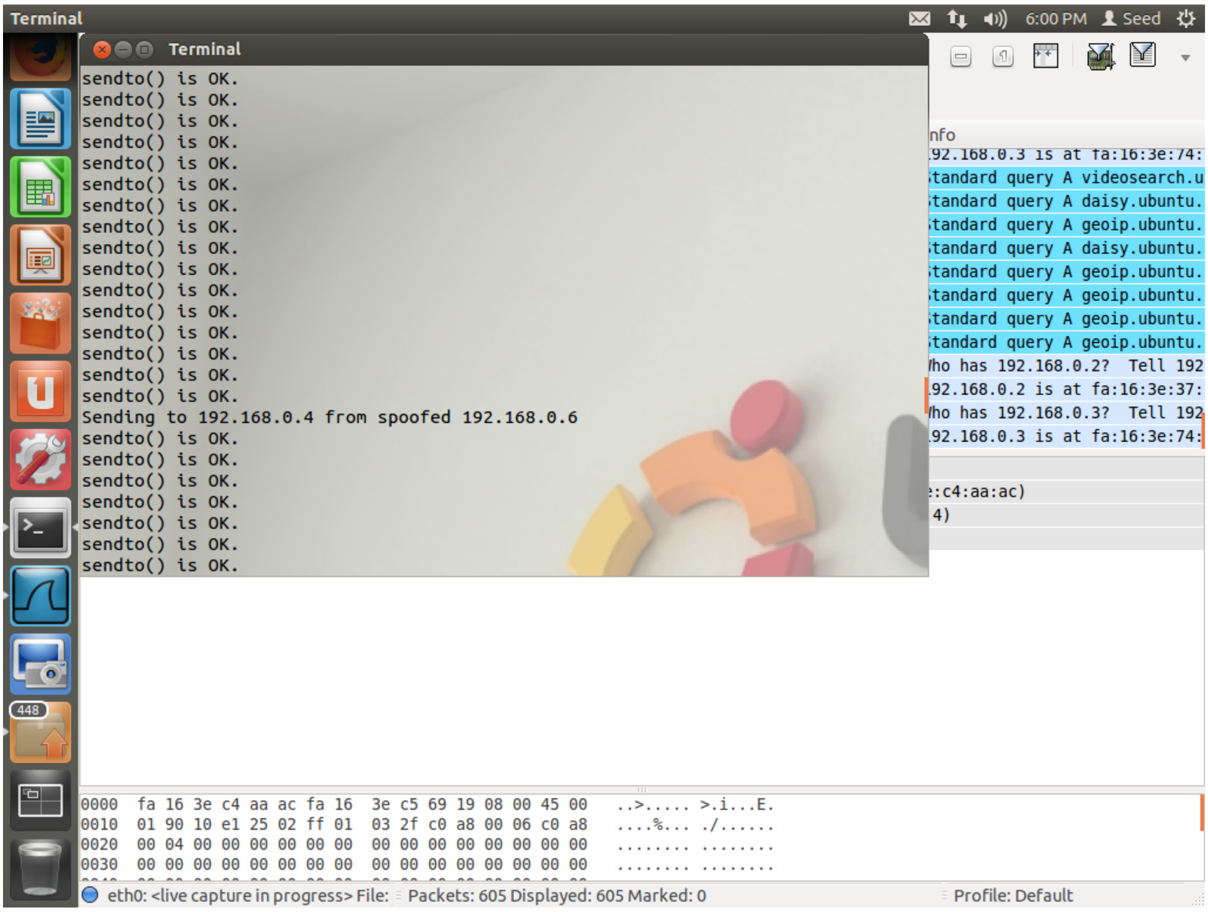
gcc spoofimcp.c -o spoof

It is run from the terminal using the command line code

sudo ./spoof <Source IP> <Destination IP> <Number of Packets>

The screen dump of terminal and Wireshark is provided below.


Capturing from eth0 [Wireshark 1.6.7]
5:59 PM Seed
Filter:
Expression... Clear Apply
No. Time Source Destination Protocol Length Info
568 2017-02-15 17:59:15.1 192.168.0.6 192.168.0.4 IPv4 414 Fragmented IP protocol (proto=
569 2017-02-15 17:59:15.1 192.168.0.6 192.168.0.4 IPv4 414 Fragmented IP protocol (proto=
570 2017-02-15 17:59:15.1 192.168.0.6 192.168.0.4 IPv4 414 Fragmented IP protocol (proto=
571 2017-02-15 17:59:15.1 192.168.0.6 192.168.0.4 IPv4 414 Fragmented IP protocol (proto=
572 2017-02-15 17:59:15.1 192.168.0.6 192.168.0.4 IPv4 414 Fragmented IP protocol (proto=
573 2017-02-15 17:59:15.1 192.168.0.6 192.168.0.4 IPv4 414 Fragmented IP protocol (proto=
574 2017-02-15 17:59:15.1 192.168.0.6 192.168.0.4 IPv4 414 Fragmented IP protocol (proto=
575 2017-02-15 17:59:15.1 192.168.0.6 192.168.0.4 IPv4 414 Fragmented IP protocol (proto=
576 2017-02-15 17:59:15.1 192.168.0.6 192.168.0.4 IPv4 414 Fragmented IP protocol (proto=
577 2017-02-15 17:59:16.2 192.168.0.6 192.168.0.2 DNS 82 Standard query A videosearch.u
578 2017-02-15 17:59:16.2 192.168.0.6 192.168.0.3 DNS 76 Standard query A daisy.ubuntu.
579 2017-02-15 17:59:21.2 192.168.0.6 192.168.0.3 DNS 82 Standard query A videosearch.u
580 2017-02-15 17:59:21.3 192.168.0.6 192.168.0.2 DNS 76 Standard query A daisy.ubuntu.
Frame 573: 414 bytes on wire (3312 bits), 414 bytes captured (3312 bits)
Ethernet II, Src: fa:16:3e:c5:69:19 (fa:16:3e:c5:69:19), Dst: fa:16:3e:c4:aa:ac (fa:16:3e:c4:aa:ac)
Internet Protocol Version 4, Src: 192.168.0.6 (192.168.0.6), Dst: 192.168.0.4 (192.168.0.4)
Data (380 bytes)
0000 fa 16 3e c4 aa ac fa 16 3e c5 69 19 08 00 45 00 ..>..... >.i...E.
0010 01 90 10 e1 25 02 ff 01 03 2f c0 a8 00 06 c0 a8 ....%... ./......
0020 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
eth0: <live capture in progress> File: Packets: 580 Displayed: 580 Marked: 0
Profile: Default


Terminal
6:00 PM Seed
Terminal
sendto() is OK.
sendto() is OK.
sendto() is OK.
sendto() is OK.
sendto() is OK.
sendto() is OK.
sendto() is OK.
sendto() is OK.
sendto() is OK.
sendto() is OK.
sendto() is OK.
sendto() is OK.
sendto() is OK.
sendto() is OK.
sendto() is OK.
sendto() is OK.
Sending to 192.168.0.4 from spoofed 192.168.0.6
sendto() is OK.
sendto() is OK.
sendto() is OK.
sendto() is OK.
sendto() is OK.
sendto() is OK.
sendto() is OK.
0000 fa 16 3e c4 aa ac fa 16 3e c5 69 19 08 00 45 00 ..>..... >.i...E.
0010 01 90 10 e1 25 02 ff 01 03 2f c0 a8 00 06 c0 a8 ....%... ./......
0020 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
eth0: <live capture in progress> File: Packets: 605 Displayed: 605 Marked: 0
Profile: Default

Task 2.b: Spoof

Spoof an ICMP echo request packet on behalf of another machine (i.e., using another machine's IP address as its source IP address).

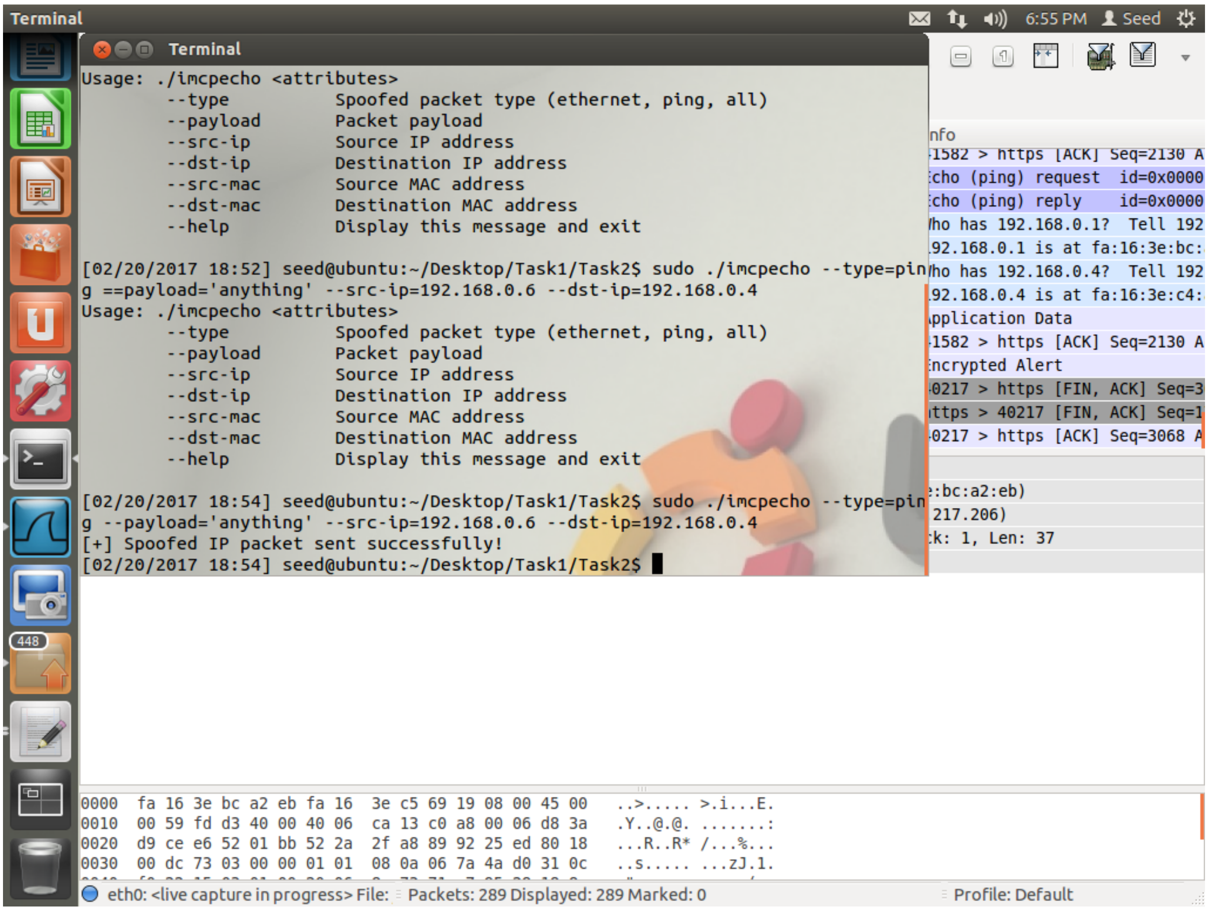
The spoofing program was downloaded from http://www.tenouk.com/Module43a.html. This program is compiled on VM1 using the command line code

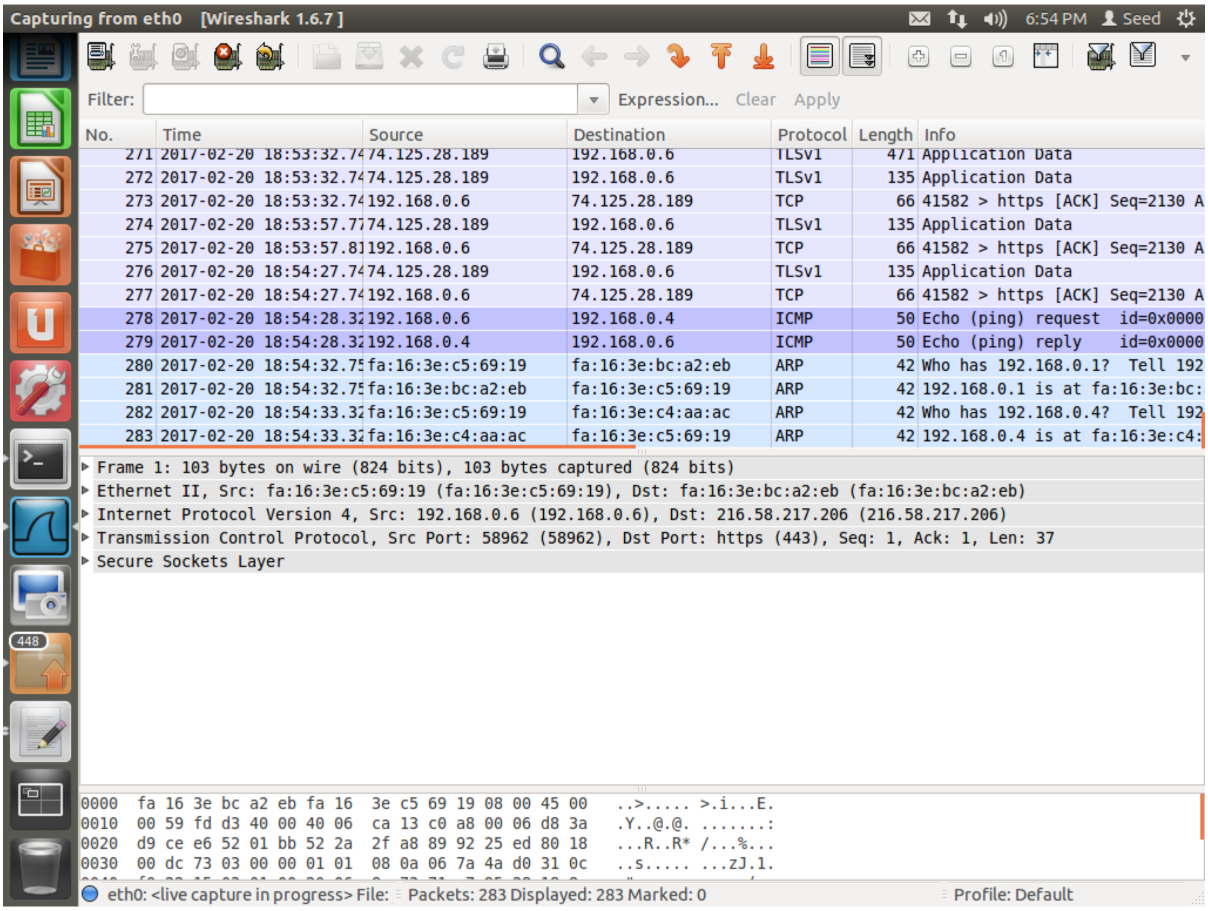
gcc spoof2.c -o icmpecho

It is run from the terminal using the command line code

sudo ./icmpecho –type=ping –payload='anything' –src-ip=<Source IP> --dst-ip=<Destination IP>

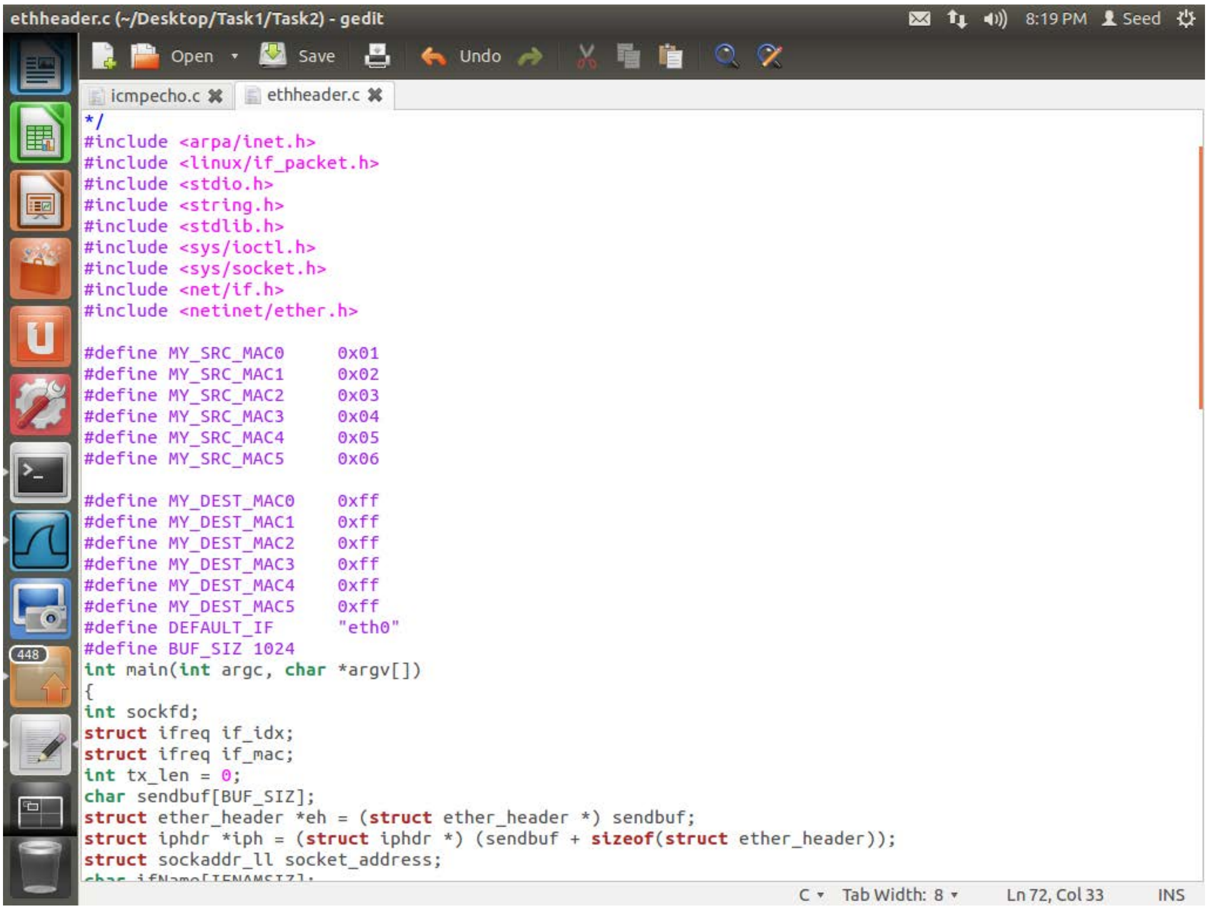
The screen dump of terminal and Wireshark is provided below.



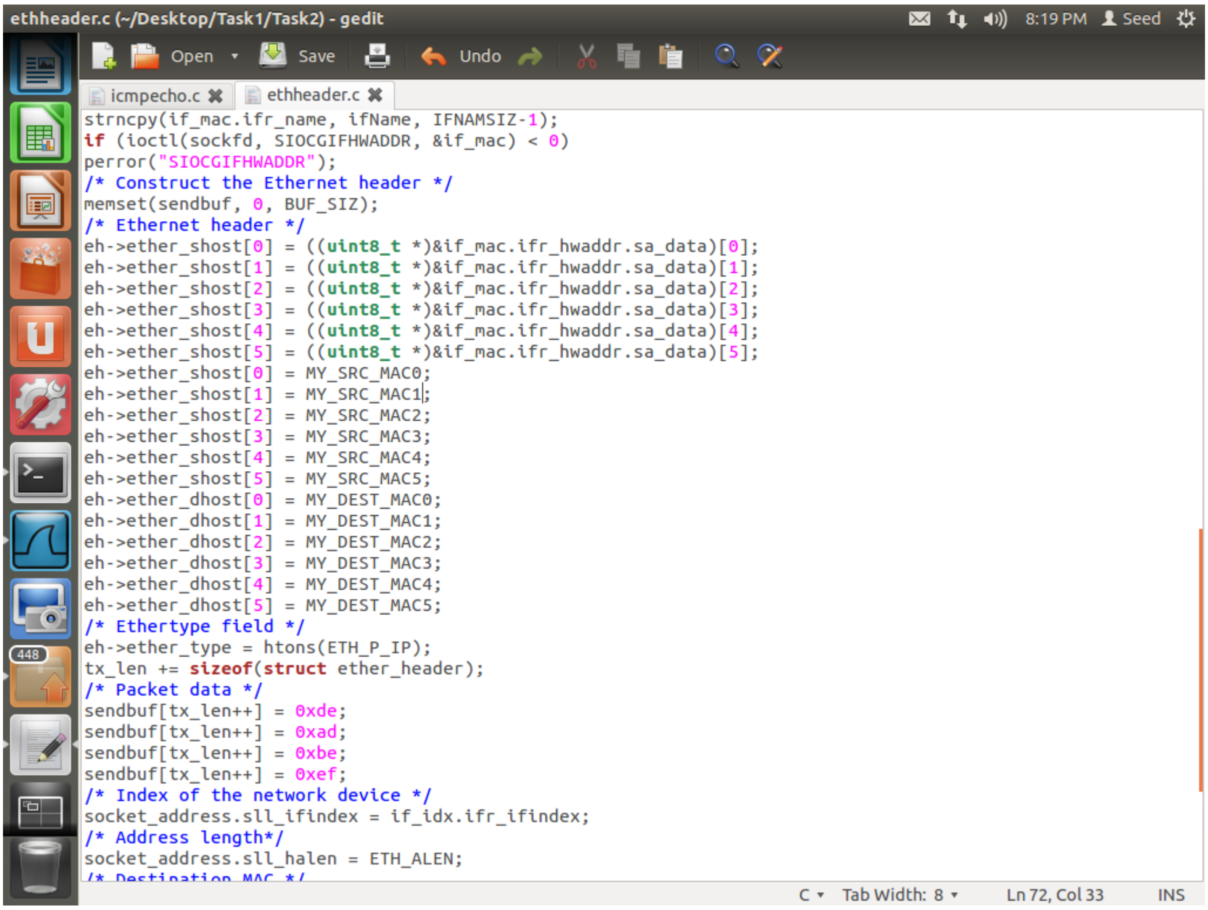


Task 2.c: Spoof an Ethernet Frame.

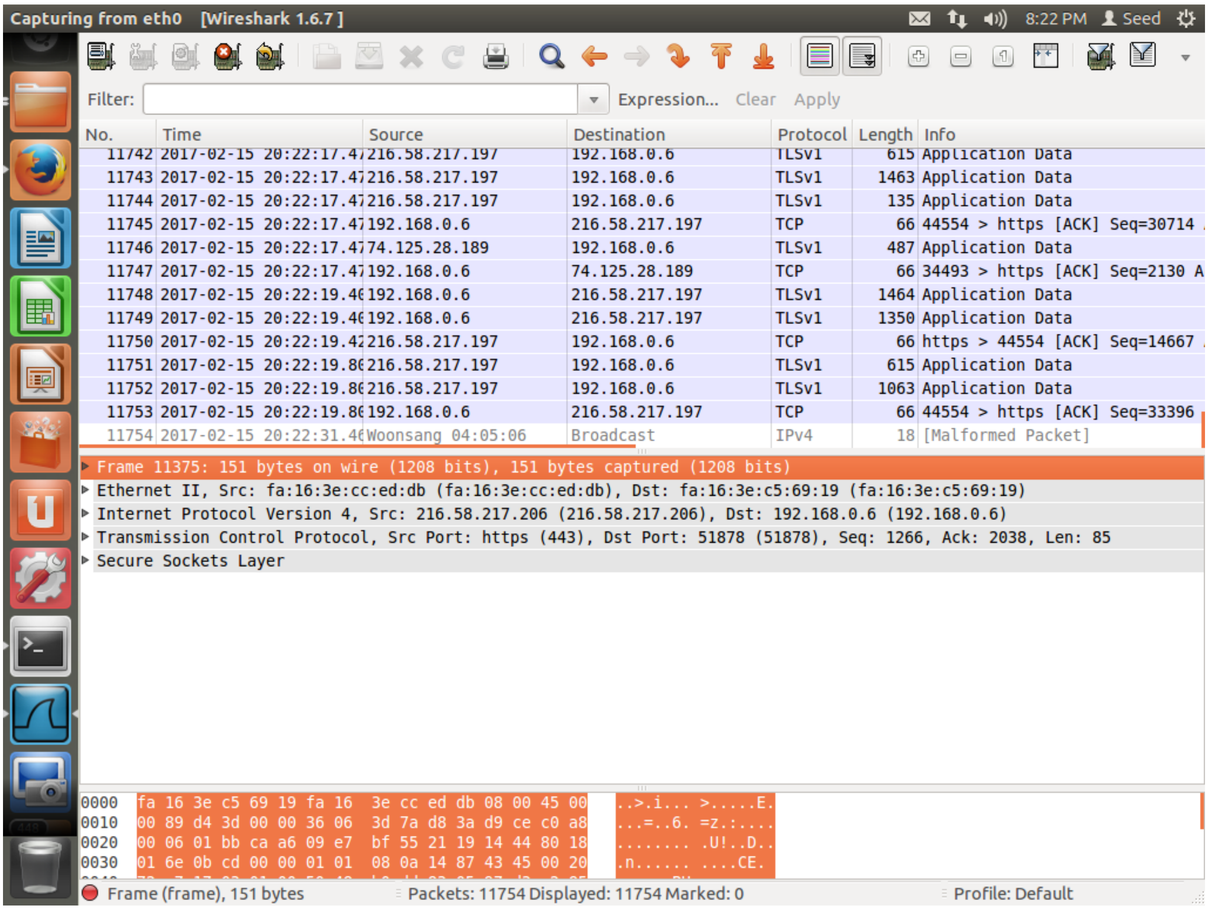
The MY_SRC_MAC0, MY_SRC_MAC1, MY_SRC_MAC2, MY_SRC_MAC3, MY_SRC_MAC4, MY_SRC_MAC5 was defined as 0x01, 0x02, 0x03, 0x04, 0x05, 0x06 respectively to get the source mac address as 01:02:03:04:05:06. The MY_ DEST _MAC0, MY_ DEST _MAC1, MY_ DEST _MAC2, MY_ DEST _MAC3, MY_ DEST _MAC4, MY_ DEST _MAC5 was defined as 0xff, 0xff, 0xff, 0xff, 0xff, 0xff respectively to get the source mac address as ff:ff:ff:ff:ff:ff to broadcast it to the entire network. This can be seen in the screen dump. The screen dump of Wireshark shows Woonsang 04:05:06 as broadcast, which is nothing but a packet from the source 01:02:03:04:05:06 being broadcasted.




```c
*/
#include <arpa/inet.h>
#include <linux/if_packet.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/ether.h>

#define MY_SRC_MAC0     0x01
#define MY_SRC_MAC1     0x02
#define MY_SRC_MAC2     0x03
#define MY_SRC_MAC3     0x04
#define MY_SRC_MAC4     0x05
#define MY_SRC_MAC5     0x06

#define MY_DEST_MAC0    0xff
#define MY_DEST_MAC1    0xff
#define MY_DEST_MAC2    0xff
#define MY_DEST_MAC3    0xff
#define MY_DEST_MAC4    0xff
#define MY_DEST_MAC5    0xff
#define DEFAULT_IF      "eth0"
#define BUF_SIZ 1024
int main(int argc, char *argv[])
{
int sockfd;
struct ifreq if_idx;
struct ifreq if_mac;
int tx_len = 0;
char sendbuf[BUF_SIZ];
struct ether_header *eh = (struct ether_header *) sendbuf;
struct iphdr *iph = (struct iphdr *) (sendbuf + sizeof(struct ether_header));
struct sockaddr_ll socket_address;
char ifName[IFNAMSIZ];
```




```c
strncpy(if_mac.ifr_name, ifName, IFNAMSIZ-1);
if (ioctl(sockfd, SIOCGIFHWADDR, &if_mac) < 0)
perror("SIOCGIFHWADDR");
/* Construct the Ethernet header */
memset(sendbuf, 0, BUF_SIZ);
/* Ethernet header */
eh->ether_shost[0] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)[0];
eh->ether_shost[1] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)[1];
eh->ether_shost[2] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)[2];
eh->ether_shost[3] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)[3];
eh->ether_shost[4] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)[4];
eh->ether_shost[5] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)[5];
eh->ether_shost[0] = MY_SRC_MAC0;
eh->ether_shost[1] = MY_SRC_MAC1;
eh->ether_shost[2] = MY_SRC_MAC2;
eh->ether_shost[3] = MY_SRC_MAC3;
eh->ether_shost[4] = MY_SRC_MAC4;
eh->ether_shost[5] = MY_SRC_MAC5;
eh->ether_dhost[0] = MY_DEST_MAC0;
eh->ether_dhost[1] = MY_DEST_MAC1;
eh->ether_dhost[2] = MY_DEST_MAC2;
eh->ether_dhost[3] = MY_DEST_MAC3;
eh->ether_dhost[4] = MY_DEST_MAC4;
eh->ether_dhost[5] = MY_DEST_MAC5;
/* Ethertype field */
eh->ether_type = htons(ETH_P_IP);
tx_len += sizeof(struct ether_header);
/* Packet data */
sendbuf[tx_len++] = 0xde;
sendbuf[tx_len++] = 0xad;
sendbuf[tx_len++] = 0xbe;
sendbuf[tx_len++] = 0xef;
/* Index of the network device */
socket_address.sll_ifindex = if_idx.ifr_ifindex;
/* Address length*/
socket_address.sll_halen = ETH_ALEN;
/* Destination MAC */
```



Question 4: Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?

Yes, if the user has runs the code with administrative privileges, IP packet length field can be set to any arbitrary value.

Question 5: Using the raw socket programming, do you have to calculate the checksum for the IP header?

No. When we use raw socket programming, we can ask the kernel to fill the checksum field.

Question 6: Why do you need the root privilege to run the programs that use raw sockets? Where does the program fail if executed without the root privilege?

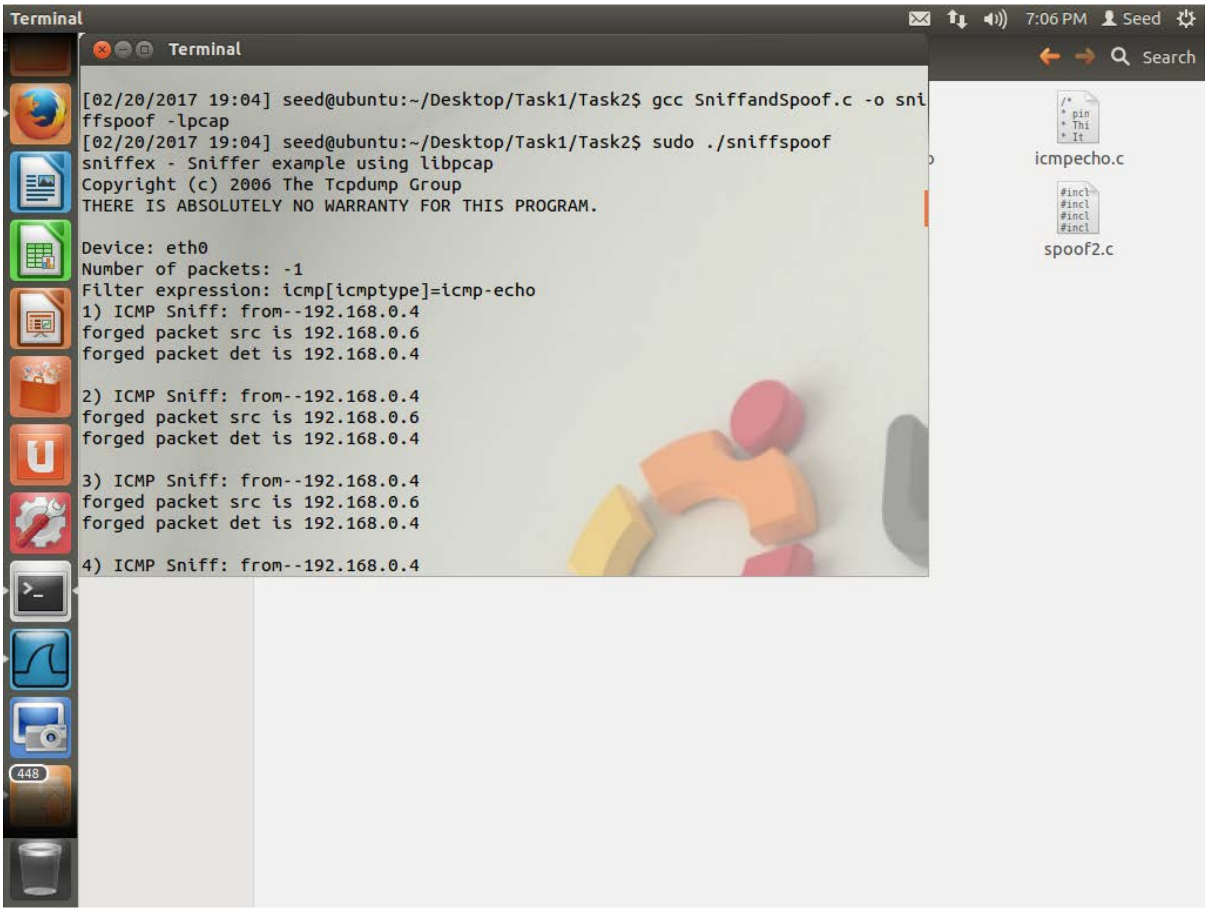
Not everyone is allowed to run raw socket programs. This is done to keep the rules of the networking in check. Example: You cannot bind a port less than 1024 for user applications. But with root privileges you can overcome these rules or restrictions imposed. And you simulate a server on any port with raw socket programming with root privileges.

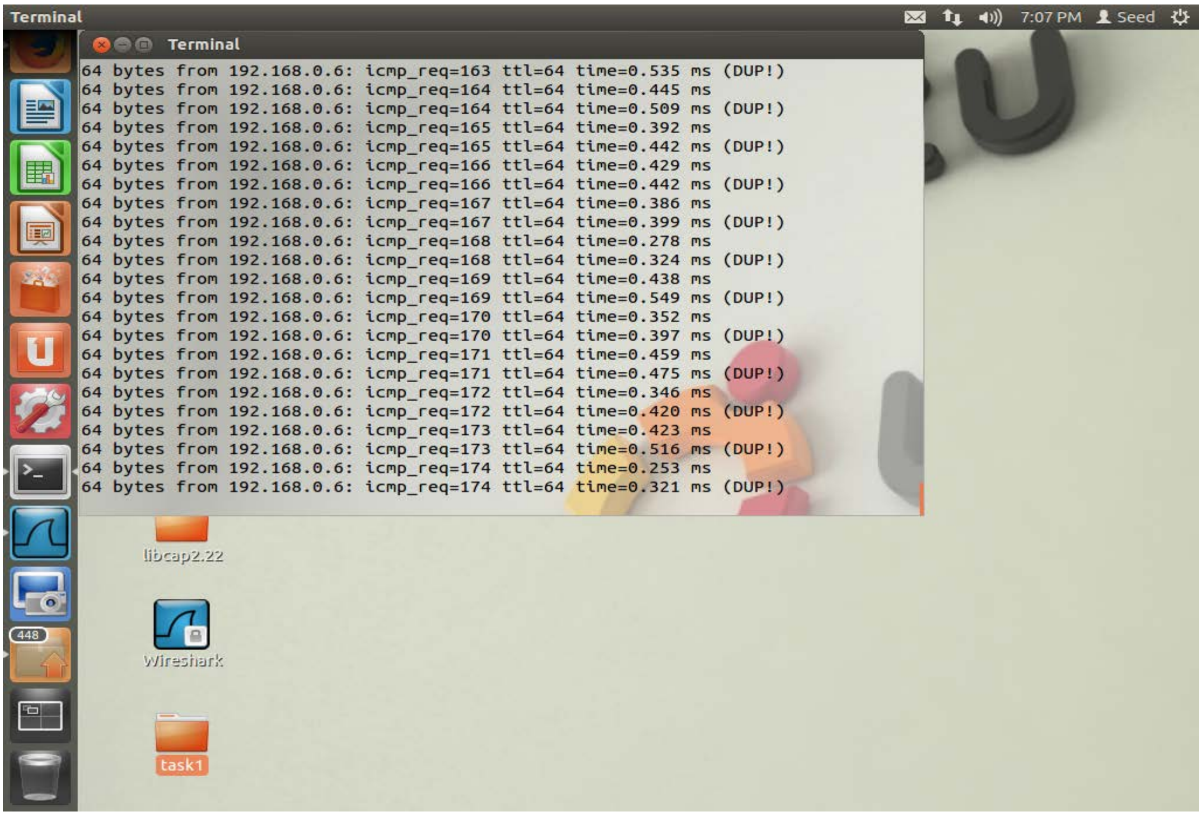
TASK 3: SNIFFING AND SPOOFING

Sniff and then Spoof in this task, you will combine the sniffing and spoofing techniques to implement the following sniff-and then-spoof program. You need two VMs on the same LAN. From VM A, you ping an IP X. This will generate an ICMP echo request packet. If X is alive, the ping program will receive an echo reply, and print out the response. Your sniff-and-then-spoof program runs on VM B, which monitors the LAN through packet sniffing. Whenever it sees an ICMP echo request, regardless of what the target IP address is, your program should immediately send out an echo reply using the packet spoofing technique. Therefore, regardless of whether machine X is alive or not, the ping program will always receive a reply, indicating that X is alive. You need to write such a program, and include screen dumps in your report to show that your program works. Please also attach the code (with adequate amount of comments) in your report.

The program was downloaded from https://blogofyu.wordpress.com/2014/02/10/internet-security-sniffing-and-spoofing-lab-report/

For this experiment, the program is compiled and run in VM1 (192.168.0.6). I then pinged VM1 from VM2 (192.168.0.4). The program on VM1 sniffs the ICMP echo request packets and is captured. The program then forges a new packet and this spoofed ICMP echo reply packet is sent. The screen dumps and the code is provided below





Code:

```
/* Sniff and spoof icmp packet   */

#define APP_NAME    "sniffex"

#define APP_DESC             "Sniffer example using libpcap"

#define APP_COPYRIGHT        "Copyright (c) 2006 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 <net/ethernet.h>
```

```
#include <netinet/in.h>

#include <netinet/ip.h>

#include <netinet/ip_icmp.h>

#include <arpa/inet.h>


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

#define SNAP_LEN 1518


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

#define SIZE_ETHERNET sizeof(struct ethhdr)


/* Spoofed packet containing only IP and ICMP headers */

struct spoof_packet

{

   struct ip iph;

   struct icmp icmph;

};

void

got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet);


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;
}

/*
```

```
 * Generates ip/icmp header checksums using 16 bit words. nwords is number of 16 bit words
 */
unsigned short in_cksum(unsigned short *addr, int len)
{
	int nleft = len;
	int sum = 0;
	unsigned short *w = addr;
	unsigned short answer = 0;

	while (nleft > 1) {
		sum += *w++;
		nleft -= 2;
	}

	if (nleft == 1) {
		*(unsigned char *) (&answer) = *(unsigned char *) w;
		sum += answer;
	}

	sum = (sum >> 16) + (sum & 0xFFFF);
	sum += (sum >> 16);
	answer = ~sum;
	return (answer);
}

/*
 * dissect/print packet
 */
void
```

```
got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{

        static int count = 1;               /* packet counter */

        int s;    // socket
        const int on = 1;

        /* declare pointers to packet headers */
        const struct ether_header *ethernet = (struct ether_header*)(packet);
        const struct ip *iph;            /* The IP header */
        const struct icmp *icmph;          /* The ICMP header */
        struct sockaddr_in dst;

        int size_ip;

        /* define/compute ip header offset */
        iph = (struct ip*)(packet + SIZE_ETHERNET);
        size_ip = iph->ip_hl*4;           // size of ip header

        if (iph->ip_p != IPPROTO_ICMP || size_ip < 20) {  // disregard other packets
                return;
        }

        /* define/compute icmp header offset */
        icmph = (struct icmp*)(packet + SIZE_ETHERNET + size_ip);

        /* print source and destination IP addresses */
        printf("%d) ICMP Sniff: from--%s\n", count, inet_ntoa(iph->ip_src) );
```

```
/* Construct the spoof packet and allocate memory with the length of the datagram
*/
char buf[htons(iph->ip_len)];
struct spoof_packet *spoof = (struct spoof_packet *) buf;

/* Initialize the structure spoof by copying everything in request packet to spoof
packet*/
memcpy(buf, iph, htons(iph->ip_len));
/* Modify ip header */

//swap the destination ip address and source ip address
(spoof->iph).ip_src = iph->ip_dst;
(spoof->iph).ip_dst = iph->ip_src;

//recompute the checksum, you can leave it to 0 here since RAW socket will
compute it for you.
(spoof->iph).ip_sum = 0;

/* Modify icmp header */

// set the spoofed packet as echo-reply
(spoof->icmph).icmp_type = ICMP_ECHOREPLY;
// always set code to 0
(spoof->icmph).icmp_code = 0;

(spoof->icmph).icmp_cksum = 0;      // should be set as 0 first to recalculate.
(spoof->icmph).icmp_cksum = in_cksum((unsigned short *) &(spoof->icmph),
sizeof(spoof->icmph));
//print the forged packet information
printf("forged packet src is %s\n",inet_ntoa((spoof->iph).ip_src));
```

```
        printf("forged packet det is %s\n\n",inet_ntoa((spoof->iph).ip_dst));

        memset(&dst, 0, sizeof(dst));
        dst.sin_family = AF_INET;
    dst.sin_addr.s_addr = (spoof->iph).ip_dst.s_addr;

        /* create RAW socket */
        if((s = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0) {
    printf("socket() error");
                return;
        }

        /* socket options, tell the kernel we provide the IP structure */
        if(setsockopt(s, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on)) < 0) {
                printf("setsockopt() for IP_HDRINCL error");
                return;
        }

        if(sendto(s, buf, sizeof(buf), 0, (struct sockaddr *) &dst, sizeof(dst)) < 0) {
                printf("sendto() error");
        }

        close(s);       // free resource

        //free(buf);
        count++;
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[] = "icmp[icmptype]=icmp-echo";		/* 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 = -1;			/* number of packets to capture, set -1 to
capture all */

	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 call back function */
	pcap_loop(handle, num_packets, got_packet, NULL);

	/* clean-up */
	pcap_freecode(&fp);
	pcap_close(handle);

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

return 0;
}
```