

Ex6 – Sniffing & Spoofing

First Part:

In this assignment, we will understand truly all the sniffing and spoofing, which are two major concepts in network security.

This assignment is divided to two part; part one will be in python programming and will contain basic understanding in many fields such as: ICMP spoofing, TCP sniffer etc.

The second part will be written in c language and will contain basic understanding in many fields such as: spoofing and sniffing of ICMP and TSP etc.

So, let's start!

Python Part:

In this section we use mainly the scapy library which contain a lot opportunities with everything which related to our subject.

At first, we created class that shows us all the IP information and display it fully.

In the command " a= IP()" we are not given any information to this function so it take the default IP (127.0.0.1).

This is the given information:

```
[01/05/22]seed@GUYSEEDLAB:~/Desktop$ subl init.py
[01/05/22]seed@GUYSEEDLAB:~/Desktop$ sudo python3 init.py
###[ IP ]###
version    = 4
ihl        = None
tos        = 0x0
len        = None
id         = 1
flags      =
frag       = 0
ttl        = 64
proto      = hopopt
chksum     = None
src        = 127.0.0.1
dst        = 127.0.0.1
\options   \
```

Task 1.1A:

```
1 from scapy.all import *
2
3 def print_pkt(pkt):
4     pkt.show()
5
6
7 interfaces = ['enp0s3', 'lo']
8 pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
```

In the all project we used the scapy library which contain all the needed library for this part.

Code: In the code we create function which will print all the information which out 'pkt' variable sniffed.

The sniff function get interfaces which related to our network inside our machine, than we filter it with the 'icmp' filter because we are sending a ping to a random IP, and last the 'prn' variable get the 'print_pkt' function.

So, what we got when we start this program and ping to a website while it running?

Pay Attention! >> we use the 'sudo' command, because we need a root privilege.

```
[01/05/22]seed@GUYSEEDLAB:~/Desktop$ sudo python3 sniffer.py
###[ Ethernet ]###
  dst      = 52:54:00:12:35:00
  src      = 08:00:27:a3:48:96
  type     = IPv4
###[ IP ]###
  version  = 4
  ihl      = 5
```

```
[01/05/22]seed@GUYSEEDLAB:~/Desktop$ ping www.google.com
PING www.google.com (142.250.185.164) 56(84) bytes of data.
64 bytes from fra16s51-in-f4.1e100.net (142.250.185.164): icmp_seq=1 ttl=109 time=86.4 ms
64 bytes from fra16s51-in-f4.1e100.net (142.250.185.164): icmp_seq=2 ttl=109 time=83.2 ms
64 bytes from fra16s51-in-f4.1e100.net (142.250.185.164): icmp_seq=3 ttl=109 time=82.9 ms
64 bytes from fra16s51-in-f4.1e100.net (142.250.185.164): icmp_seq=4 ttl=109 time=88.2 ms
```

without root privilege>>

```
[01/05/22]seed@GUYSEEDLAB:~/Desktop$ python3 sniffer.py
Traceback (most recent call last):
  File "sniffer.py", line 8, in <module>
    pkt = sniff(iface=interfaces, filter='icmp', prn=print_pkt)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1036, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 894, in _run
    sniff_sockets.update(
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 895, in <genexpr>
    (L2socket(type=ETH_P_ALL, iface=ifname, *arg, **karg),
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 398, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type)) # noqa: E501
  File "/usr/lib/python3.8/socket.py", line 231, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
[01/05/22]seed@GUYSEEDLAB:~/Desktop$ █
```

1.1B >> Capture ICMP packets

In this part we wanted to capture ICMP packets only, with the previous commend (show())

We got a lot of information, and right now we want ICMP packets only, so we needed to create a sniffer which sniff only ICMP packets.

```
sniff_icmp.py
~/Desktop

1 from scapy.all import *
2
3 def print_pkt(pkt):
4     if pkt[ICMP] is not None:
5         if pkt[ICMP].type == 0 or pkt[ICMP].type==8:
6             print("-----ICMP PACKET-----")
7             print(f"\tSOURCE:{pkt[IP].src}")
8             print(f"\tDEST:{pkt[IP].dst}")
9
10            if pkt[ICMP].type == 0:
11                print(f"\tICMP type : ECHO-REPLAY")
12
13            if pkt[ICMP].type == 8:
14                print(f"\tICMP type: ECHO-REQUEST")
15
16 interfaces = ['enp0s3','lo']
17 pkt= sniff(iface=interfaces, filter='icmp', prn= print_pkt)
```

As we can see in the code, first of all , we check if there are ICMP packets after we sniff, as we know, there are two types of ICMP which we interested in, “ ICMP ECHO (PING) REPLAY” And “ICMP ECHO (PING)”.

And then we printed all the information we interesting in.

```
[01/05/22]seed@GUYSEEDLAB:~/Desktop$ sudo python3 sniff_icmp.py
-----ICMP PACKET-----
      SOURCE:10.0.2.5
      DEST:8.8.8.8
      ICMP type: ECHO-REQUEST
-----ICMP PACKET-----
      SOURCE:8.8.8.8
      DEST:10.0.2.5
      ICMP type : ECHO-REPLAY
```

1.1B >> Capture TCP packets

In this part we wanted to capture TCP packets only, right now we want TCP packets only, so we needed to create a sniffer which sniff only TCP packets on port 23 and host 10.0.2.5

A screenshot of a code editor window titled 'sniff_tcp.py' with a file icon on the left and a 'Save' button on the right. The editor contains Python code for sniffing TCP packets. The code is as follows:

```
1 from scapy.all import *
2
3 def print_pkt(pkt):
4     if pkt[TCP] is not None:
5         print("-----TCP PACKET-----")
6         print(f"\tSOURCE: {pkt[IP].src}")
7         print(f"\tDEST: {pkt[IP].dst}")
8         print(f"\t TCP SRC PORT:{pkt[TCP].sport}")
9         print(f"\t TCP DEST PORT:{pkt[TCP].dport}")
10
11 interfaces=['enp0s3','lo']
12 pkt= sniff(iface=interfaces, filter='tcp port 23 and src host 10.0.2.5', prn=print_pkt)
```

As we can see in the code above, if the pkt variable (which got the sniff information) in the [TCP] place is not None, we print all the needed information.

In aim to got this information we use the telnet command, that because this protocol is working with TCP port 23 (as we asked too).

```
[01/05/22] seed@GUYSEEDLAB:~/.../volumes$ sudo python3 sniff_tcp.py
-----TCP PACKET-----
SOURCE: 10.0.2.5
DEST: 10.0.2.5
TCP SRC PORT:54858
TCP DEST PORT:23
-----TCP PACKET-----
SOURCE: 10.0.2.5
DEST: 10.0.2.5
TCP SRC PORT:54858
TCP DEST PORT:23
```

1.1B >> Capture specific Subnet packets

In this part we filtered the wanted information with a given subnet (128.230.0.0/16) and uses the show command, but right now it will show us the information we filtered.

```
Open  ▾  [+]
```

sniffer_subnet.py
ShareFiles ~/ShareFiles/volumes

```
1|from scapy.all import *
2|
3|def print_pkt(pkt):
4|    pkt.show()
5|
6|interfaces = ['enp0s3','lo']
7|pkt = sniff(iface=interfaces, filter= 'dst net 128.230.0.0/16', prn=print_pkt)
```

```
[01/05/22]seed@GUYSEEDLAB:~/../volumes$ sudo python3 sniffer_subnet.py
###[ Ethernet ]###
  dst      = 52:54:00:12:35:00
  src      = 08:00:27:a3:48:96
  type     = IPv4
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 20
  id       = 1
  flags    =
  frag     = 0
  ttl      = 64
  proto    = hopopt
  chksum   = 0xedfe
  src      = 10.0.2.5
  dst      = 128.230.0.0 ←
  \options \
```

as we can see in the image above we got the subnet order we wanted.

1.2B >> Spoofing ICMP Packets

What is packet spoofing? >>

Packet Spoofing is a creation of internet protocol packets with purpose of concealing the identity of the sender or impersonating another computer system.

The attacker creates an IP packet and sends it to the server, which is known as an SYN (synchronize) request.


```
Open  ▾  [+]
```

```
icmp_spoofing.py
ShareFiles ~/ShareFiles/volumes

1 from scapy.all import *
2
3 a=IP()
4 a.src = '15.04.9.8'      #my birthday
5 a.dst = '10.0.2.5'
6 send(a/ICMP())
7 ls(a)
```

Here again we use the scapy package, then I wrote what is my source IP and what is the destination IP, the source IP is a random one (in this case I choose my birthday).

I wanted to see if it's actually work so I use wire shark and the code itself, to see if the information I have sent (in the ls(a)) command doesn't rise any problem, and this is what I have got:

```
[01/05/22]seed@GUYSEEDLAB:~/.../volumes$ sudo python3 icmp_spoofing.py
.
Sent 1 packets.
version      : BitField  (4 bits)          = 4              (4)
ihl          : BitField  (4 bits)          = None           (None)
tos          : XByteField                    = 0              (0)
len          : ShortField                    = None           (None)
id           : ShortField                    = 1              (1)
flags        : FlagsField  (3 bits)         = <Flag 0 ()>    (<Flag 0 ()>)
frag        : BitField  (13 bits)          = 0              (0)
ttl          : ByteField                     = 64             (64)
proto        : ByteEnumField                = 0              (0)
chksum       : XShortField                  = None           (None)
src          : SourceIPField                = '15.04.9.8'    (None)
dst          : DestIPField                  = '10.0.2.5'     (None)
options      : PacketListField              = []             ([])
[01/05/22]seed@GUYSEEDLAB:~/.../volumes$
```

As we can see the response in my lab, the packet was sent successfully and all the information inside.

In addition I have got an Echo (ping) replay , and Echo (ping) request in my wireshark:

5	2022-01-05 11:0...	15.4.9.8	10.0.2.5	ICMP	44 Echo (ping) request	id=
6	2022-01-05 11:0...	10.0.2.5	15.4.9.8	ICMP	44 Echo (ping) reply	id=

▶ Frame 5: 44 bytes on wire (352 bits), 44 bytes captured (352 bits) on interface any, id 0

▶ Linux cooked capture

▶ Internet Protocol Version 4, Src: 15.4.9.8, Dst: 10.0.2.5

▶ Internet Control Message Protocol

0000	00 02 03 04 00 06 00 00 00 00 00 00 00 00 08 00
0010	45 00 00 1c 00 01 00 00 40 01 56 d0 0f 04 09 08	E.....@.V.....
0020	0a 00 02 05 08 00 f7 ff 00 00 00 00

1.3 Traceroute>>

What is Traceroute>>

Traceroute is a method to find the path of the data between one website to another via TCP/IP protocols.

```
tracert.py
ShareFiles -/ShareFiles/volumes

1|from scapy.all import *
2
3|bool = True
4|i = 1
5|while bool:
6|    a = IP(dst = '8.8.8.8', ttl = i)#creating variable which changing ttl
7|    b = ICMP()
8|    get = sr1(a/b , timeout = 6, verbose=0)
9
10|    if get is None:
11|        print(f"{i} TIME OUT!")
12|    elif get.type == 0:
13|        print(f"{i} {get.src}")
14|        bool= False
15|    else:
16|        print (f"{i} {get.src}")
17
18|    i= i+1
```

Again, in our code we use the scapy library, most of our code is inside a while loop, the reason for that is we want the loop keep iterating as long as it is routing. Here I choose to the traceroute to google is IP, And the TTL is growing in every iteration. Than we used the sr1() function from scapy's library, The function sr1() is a variant that only returns one packet that answered the packet (or the packet set) sent. With that function we can know how much routers the data is going through.

```
[01/06/22]seed@GUYSEEDLAB:~/.../volumes$ sudo python3 traceroute.py
1 10.0.2.1
2 192.168.14.1
3 212.179.37.1
4 10.250.83.6
5 212.25.77.18
6 10.250.99.13
7 212.25.70.69
8 172.253.69.97
9 66.249.95.53
10 8.8.8.8
[01/06/22]seed@GUYSEEDLAB:~/.../volumes$ █
```

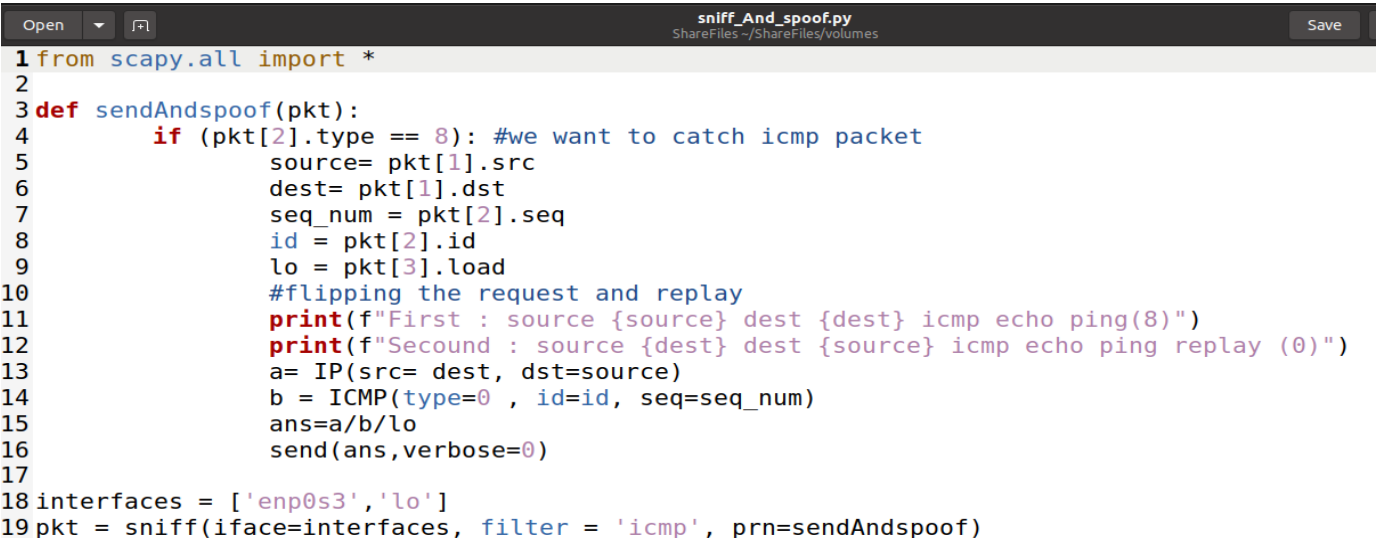
When I ran the code as you can see, I went through 10 routers until I arrived to the wanted destination, and we can see the routers IP I went through.

Than I ran again the traceroute but I used the preserved function and this is the information I got:

```
[01/06/22] seed@GUYSEEDLAB:~$ traceroute 8.8.8.8
traceroute to 8.8.8.8 (8.8.8.8), 30 hops max, 60 byte packets
 1  _gateway (10.0.2.1)  0.425 ms  0.330 ms  0.301 ms
 2  * * *
 3  * * *
 4  * * *
 5  * * *
 6  * * *
 7  * * *
 8  * * *
 9  * * *
```

1.4 Sniffing and Spoofing>>

In this part of the task we use the ARP protocol, The Address Resolution Protocol (ARP) is a communication protocol used for discovering the link layer address, such as a MAC address, associated with a given internet layer address, typically an IPv4 address.



```
sniff_And_spoof.py
ShareFiles ~/ShareFiles/volumes

1 from scapy.all import *
2
3 def sendAndspoof(pkt):
4     if (pkt[2].type == 8): #we want to catch icmp packet
5         source= pkt[1].src
6         dest= pkt[1].dst
7         seq_num = pkt[2].seq
8         id = pkt[2].id
9         lo = pkt[3].load
10        #flipping the request and replay
11        print(f"First : source {source} dest {dest} icmp echo ping(8)")
12        print(f"Secound : source {dest} dest {source} icmp echo ping replay (0)")
13        a= IP(src= dest, dst=source)
14        b = ICMP(type=0 , id=id, seq=seq_num)
15        ans=a/b/lo
16        send(ans,verbose=0)
17
18 interfaces = ['enp0s3','lo']
19 pkt = sniff(iface=interfaces, filter = 'icmp', prn=sendAndspoof)
```

In our code we used again in the scapy library, in the if condition we check if the packet is type is a ICMP echo ping type, than we took all the necessary information to spoof the packet, the original source and destination are the same, bet when we return an answer, we flip between the source and destination and send the answer after we spoof the packet. In addition, we used the same sniff tool, using the icmp filter, and the same interfaces.

In this part we asked to run three different scenarios, that in every scenario we ping to a different IP, and capture the results.

First, we ping to fiction IP: '1.2.3.4'.

If the code we wrote isn't running, we will see a 100% packet lost.

```
^C[01/06/22]seed@GUYSEEDLAB:~/.../volumes$ sudo python3 sniff_And_spoof.py
First : source 10.0.2.5 dest 1.2.3.4 icmp echo ping(8)
Secound : source 1.2.3.4 dest 10.0.2.5 icmp echo ping replay (0)
First : source 10.0.2.5 dest 1.2.3.4 icmp echo ping(8)
Secound : source 1.2.3.4 dest 10.0.2.5 icmp echo ping replay (0)
First : source 10.0.2.5 dest 1.2.3.4 icmp echo ping(8)
Secound : source 1.2.3.4 dest 10.0.2.5 icmp echo ping replay (0)
```

as we can see our sniffer succeeded to capture the ping to the fictional IP and response instead of it!

```
[01/06/22]seed@GUYSEEDLAB:~$ ping -c 3 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=19.2 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=18.4 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=25.3 ms

--- 1.2.3.4 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2006ms
rtt min/avg/max/mdev = 18.382/20.952/25.302/3.092 ms
```

At the image above, if our codes wasn't running during the ping command none of the packet would be received, but as we can see, thanks to the sniff & spoof tool, the ping message received successfully.

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help						
Apply a display filter ... <Ctrl-/>						
No.	Time	Source	Destination	Protocol	Length	Info
37	2022-01-06 12:3...	10.0.2.5	10.0.2.5	TELNET	85	Telnet Data ...
38	2022-01-06 12:3...	10.0.2.5	10.0.2.5	TCP	68	54862 → 23 [ACK] Seq=1359736028 Ack=371792299
39	2022-01-06 12:3...	10.0.2.5	10.0.2.5	TELNET	85	Telnet Data ...
40	2022-01-06 12:3...	10.0.2.5	10.0.2.5	TCP	68	54858 → 23 [ACK] Seq=816781432 Ack=3119078592
41	2022-01-06 12:3...	PcsCompu_a3:48:96		ARP	44	Who has 10.0.2.1? Tell 10.0.2.5
42	2022-01-06 12:3...	RealtekU_12:35:00		ARP	62	10.0.2.1 is at 52:54:00:12:35:00
43	2022-01-06 12:3...	10.0.2.5	10.0.2.5	TELNET	70	Telnet Data ...

Frame 1: 77 bytes on wire (616 bits), 77 bytes captured (616 bits) on interface any, id 0
 Linux cooked capture
 Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.53
 User Datagram Protocol, Src Port: 46332, Dst Port: 53
 Domain Name System (query)

As we can in the wireshark sniffing tool, there is a “conversation” in the network, the ARP asking who has a specific IP, and our program (“the attacker”) response that she knows, and in that way we could “lie” about the packet capture.

In the second scenario, we asked to ping to '10.9.0.99', which this IP is not existing IP on our LAN.

When I tried to run our program and tried to spoof the packet like in the first scenario, I didn't got any feedback, and all the packet got lost.

```
[01/06/22]seed@GUYSEEDLAB:~$ ping 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
From 10.9.0.1 icmp_seq=1 Destination Host Unreachable
From 10.9.0.1 icmp_seq=2 Destination Host Unreachable
From 10.9.0.1 icmp_seq=3 Destination Host Unreachable
From 10.9.0.1 icmp_seq=4 Destination Host Unreachable
From 10.9.0.1 icmp_seq=5 Destination Host Unreachable
From 10.9.0.1 icmp_seq=6 Destination Host Unreachable
From 10.9.0.1 icmp_seq=7 Destination Host Unreachable
From 10.9.0.1 icmp_seq=8 Destination Host Unreachable
From 10.9.0.1 icmp_seq=9 Destination Host Unreachable
From 10.9.0.1 icmp_seq=10 Destination Host Unreachable
From 10.9.0.1 icmp_seq=11 Destination Host Unreachable
From 10.9.0.1 icmp_seq=12 Destination Host Unreachable
^C
--- 10.9.0.99 ping statistics ---
14 packets transmitted, 0 received, +12 errors, 100% packet loss, time 13321ms
pipe 3
```

The first scenario was the most interesting for me.

Here we tried to spoof packet which sent to known IP, google is IP (8.8.8.8) and the results

was very interesting:


```
^C[01/06/22]seed@GUYSEEDLAB:~/.../volumes$ sudo python3 sniff_And_spoof.py
First : source 10.0.2.5 dest 8.8.8.8 icmp echo ping(8)
Secound : source 8.8.8.8 dest 10.0.2.5 icmp echo ping replay (0)
First : source 10.0.2.5 dest 8.8.8.8 icmp echo ping(8)
Secound : source 8.8.8.8 dest 10.0.2.5 icmp echo ping replay (0)
First : source 10.0.2.5 dest 8.8.8.8 icmp echo ping(8)
Secound : source 8.8.8.8 dest 10.0.2.5 icmp echo ping replay (0)
First : source 10.0.2.5 dest 8.8.8.8 icmp echo ping(8)
Secound : source 8.8.8.8 dest 10.0.2.5 icmp echo ping replay (0)
```

In the code above it seems that our program succeeded to capture this packet and spoof them.

```
64 bytes from 8.8.8.8: icmp_seq=13 ttl=116 time=69.4 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=14 ttl=64 time=15.6 ms
64 bytes from 8.8.8.8: icmp_seq=14 ttl=116 time=63.9 ms (DUP!)
^C
--- 8.8.8.8 ping statistics ---
14 packets transmitted, 14 received, +14 duplicates, 0% packet loss, time 13040ms
rtt min/avg/max/mdev = 12.621/37.760/75.373/20.840 ms
```

However, as we can see in the image above, for every ping that we have sent, we got a little message next to it : DUP!, which means duplicate.

When we are trying to spoof packet from an existing IP, the terminal can tell us that the packet was duplicate, how many duplicate etc.



No.	Time	Source	Destination	Protocol	Length	Info
19	2022-01-06 12:5...	8.8.8.8	10.0.2.5	ICMP	100	Echo (ping) reply id=0x0018, seq=6/1536, t
20	2022-01-06 12:5...	8.8.8.8	10.0.2.5	ICMP	100	Echo (ping) reply id=0x0018, seq=6/1536, t
21	2022-01-06 12:5...	PcsCompu_a3:48:96		ARP	44	Who has 10.0.2.3? Tell 10.0.2.5
22	2022-01-06 12:5...	PcsCompu_4c:8f:3c		ARP	62	10.0.2.3 is at 08:00:27:4c:8f:3c
23	2022-01-06 12:5...	10.0.2.5	8.8.8.8	ICMP	100	Echo (ping) request id=0x0018, seq=7/1792, t
24	2022-01-06 12:5...	8.8.8.8	10.0.2.5	ICMP	100	Echo (ping) reply id=0x0018, seq=7/1792, t
25	2022-01-06 12:5...	8.8.8.8	10.0.2.5	ICMP	100	Echo (ping) reply id=0x0018, seq=7/1792, t

As we can see in the image above, again we succeeded to capture this packets and “lie” that we are google, but in the previous image we saw how the terminal can tell us that this packet was duplicate by someone.

C part:

In this part of the assignment we work mainly with pcap, which is an API than can capture traffic in the internet.

```
sniffer.c
1  #include <pcap.h>
2  #include <stdio.h>
3  #include <arpa/inet.h>
4  #include <stdlib.h>
5
6  // ethernet struct.
7  struct ethheader
8  {
9      u_char ether_dhost [6];
10     u_char ether_shost [6];
11     u_short ether_type;
12 };
13
14 // IPV4 struct.
15 struct ipheader
16 {
17     unsigned char ip_hl: 4;
18     unsigned char ip_v: 4;
19     unsigned short int ip_flag: 3;
20     unsigned short int ip_off: 13;
21     struct in_addr source_ip;
22     struct in_addr destination_ip;
23     // unsigned char ip_tos; // this is more fields that ip have
24     // unsigned short int ip_len; // but in this function we are not using them
25     // unsigned short int ip_id;
26     // unsigned char ip_ttl;
27     // unsigned char ip_protocol;
28     // unsigned short int ip_checksum;
29 };
30
31
32 void got_packet(u_char *args, const struct pcap_pkthdr *header,
33 const u_char *packet)
34 {
35     printf("A Packet has been Captured\n");
36     struct ethheader *e = (struct ethheader *)packet;
37     if( htons(e->ether_type)== 0x0800)
38     {
39         struct ipheader *ip = (struct ipheader*) (packet + sizeof(struct ethheader));
40         printf(" Source: %s\n", inet_ntoa(ip->source_ip));
41         printf(" Destination: %s\n",inet_ntoa(ip->destination_ip));
42     }
43 }
44
45 int main()
46 {
47     pcap_t *handle;
48     char_errbuf[PCAP_ERRBUF_SIZE];
49     struct bpf_program fp;
50     char filter_exp[] = "icmp";
51     bpf_u_int32 net;
52     handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
53     // Step 2: Compile filter exp into BPF psuedo-code
54     pcap_compile(handle, &fp, filter_exp, 0, net);
55     if (pcap_setfilter(handle, &fp) !=0) {
56         pcap_peror(handle, "Error:");
57         exit(EXIT_FAILURE);
58     }
59     // Step 3: Capture packets
60     pcap_loop(handle, -1, got_packet, NULL);
61     pcap_close(handle); //Close the handle
62     return 0;
63 }
```

In our code, at first we filled and create the IP struct and the ‘ethheader’ struct.

Then, in the main function, we are using the pcap_open_live in aim that our network could be open for sniffing.

Than we are using the pcap_compile and pcap_setfilter to determine filters for the sniffing.

After that, we have the pcap_loop function that which this function run in an endless loop until we (the users) stop the activity), and in it we calling every time to the function “got_packet” which display in every iteration the source and the destination IP.

| this picture is with promiscuous mode 1|

```
[01/08/22]seed@GUYSEEDLAB:~/.../volumes$ sudo ./sniffer
A Packet has been Captured
Source: 137.205.64.0
Destination: 64.1.148.199
A Packet has been Captured
Source: 5.124.0.0
```

Here we need also to use the sudo command in aim to run the program, that because we are looking for raw packets directly from our Network card, and it could be a very dangerous process to the user.

When we tried to run the program with no root privilege, we got a ‘segmentation fault’ error.

When the promiscuous mode is on we can sniff packet that are not necessarily sent to our network card but this packet went through it. The change of the promiscuous mode is in pcap_open_live function.

|this picture is with promiscuous mode 0|

```
[01/08/22]seed@GUYSEEDLAB:~/.../volumes$ sudo ./sniffer
A Packet has been Captured
Source: 173.113.64.0
Destination: 64.1.113.35
A Packet has been Captured
Source: 5.133.0.0
Destination: 114.1.38.240
A Packet has been Captured
Source: 173.162.64.0
```

Capture Icmp messages between two hosts>>

```
[01/08/22]seed@GUYSEEDLAB:~/.../volumes$ sudo ./sniffer_icmp
A Packet has been Captured
Source: 120.225.64.0
Destination: 64.1.173.176
A Packet has been Captured
Source: 64.1.173.176
Destination: 120.225.64.0
A Packet has been Captured
Source: 120.225.64.0
Destination: 64.1.173.176
```

Capture TCP with port range 10-100>>

```
[01/08/22]seed@GUYSEEDLAB:~/.../volumes$ sudo ./sniff_range
A Packet has been Captured
Source: 11.175.64.0
Destination: 64.6.15.243
A Packet has been Captured
Source: 11.176.64.0
Destination: 64.6.15.242
A Packet has been Captured
Source: 11.177.64.0
Destination: 64.6.15.241
A Packet has been Captured
```

Capture the password using telnet>>

Telnet sending the password char after char in different packets, using that, we succeeded to capture my password (dees):

```
[01/08/22]seed@GUYSEEDLAB:~/.../volumes$ sudo ./sniff_pass
```

```
TCP PACKET HAD BEEN CAPTURED:
```

```
B....S
```

```
EF.7@#-
```

```
.....??
```

```
W... Q..S
```

```
TCP PACKET HAD BEEN CAPTURED:
```

```
B
```

```
V..T4.
```

```
@EW.
```

```
....>
```

```
R.HF..
```

```
TCP PACKET HAD BEEN CAPTURED:
```

```
B....S
```

```
EF.@@@.]
```

```
....??
```

```
W... 0..E
```

```
TCP PACKET HAD BEEN CAPTURED:
```

```
B....S
```

```
EF.A@@-Z
```

```
... ..??
```

```
W... Q..E
```

```
TCP PACKET HAD BEEN CAPTURED:
```

```
B....S
```

```
EF.B@@-U
```

```
... .. ?
```

```
J..IE..._D
```

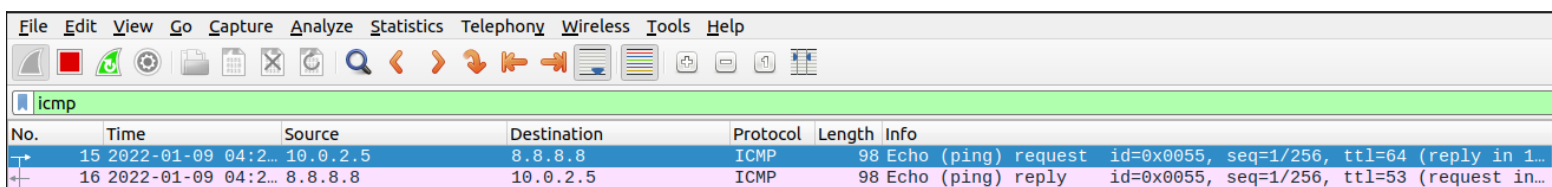
2.2A – Spoofing program>>

In this part we asked for to write a packet spoofing program in c and provide an evidence (we used in wireshark) to show that is actually work.

So we created a raw socket like in the previous tasks with an IP header, ICMP header and ethernet header.

Than we used the BSD checksum to calculate our checksum.

While we sent the ping to '8.8.8.8' we turn on the wireshark and see if capture it:

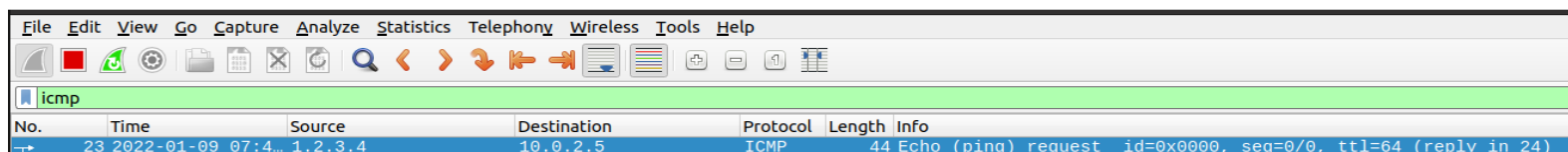


The image shows a Wireshark network packet capture. The filter is set to 'icmp'. Two packets are visible: packet 15 is an ICMP Echo (ping) request from 10.0.2.5 to 8.8.8.8, and packet 16 is the corresponding Echo (ping) reply from 8.8.8.8 to 10.0.2.5. Both packets have a length of 98 bytes.

No.	Time	Source	Destination	Protocol	Length	Info
15	2022-01-09 04:2...	10.0.2.5	8.8.8.8	ICMP	98	Echo (ping) request id=0x0055, seq=1/256, ttl=64 (reply in 1...
16	2022-01-09 04:2...	8.8.8.8	10.0.2.5	ICMP	98	Echo (ping) reply id=0x0055, seq=1/256, ttl=53 (request in...

2.2A>> ICMP Spoof packet>>

In this part we asked to wee if out sniffing program will capture the spoofing to unsexist IP.



The image shows a Wireshark network packet capture. The filter is set to 'icmp'. A single packet is visible: packet 23 is an ICMP Echo (ping) request from 1.2.3.4 to 10.0.2.5. The packet length is 44 bytes. The info field indicates it is a request with id=0x0000, seq=0/0, and ttl=64.

No.	Time	Source	Destination	Protocol	Length	Info
23	2022-01-09 07:4...	1.2.3.4	10.0.2.5	ICMP	44	Echo (ping) request id=0x0000, seq=0/0, ttl=64 (reply in 24)

As we can see in the image above, we succeeded to spoof our packet and send An ECHO ping request from a fictional IP, and the wireshark manage to

capture it.

Questions:

1. The answer to that question is NO.

if we will change to IP packet length to a random value and not to his real length we will capture error when we will try to activate the program, the error will be in the “sendto” function.

2. There is no need to calculate the checksum of the IP header, our operating system knows how to calculate it by itself.

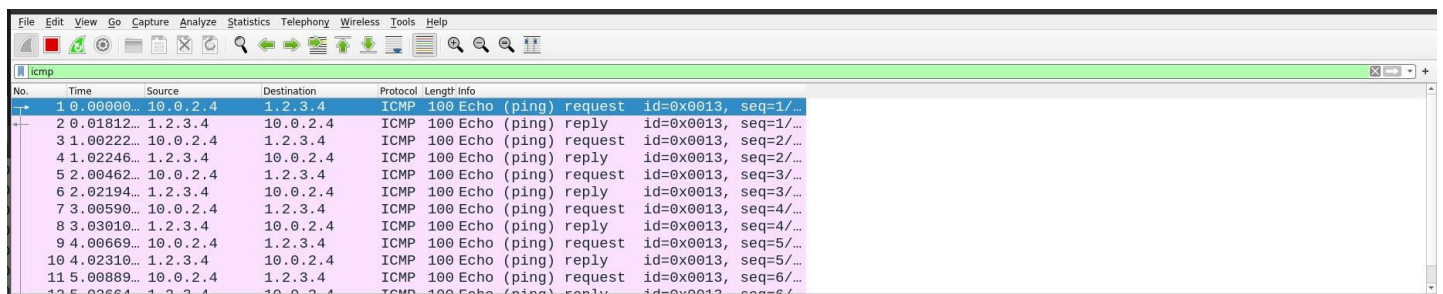
In other cases, in ICMP header for example, we do need to calculate the checksum, that because we need to verify the validity of the ICMP header, implement this function insure us to check if data went lost or not, but in IP header there is no need to calculate the check sum.

3. Like we see in previous excises, we have to use the root privilege in aim to run our program, the reason to that is that in this task we are dealing a lot with raw sockets, and when we ran raw sockets we have to do it while we in a root privilege, the reason to that is the sniffing process could be very dangerous to the user, we need to computer permission to sniff everything that happening in our Network card.

Sniff and spoof>>

In this part of the task we asked to spoof an ICMP request when we are doing ping to unsexist IP (1.2.3.4), our attacker capture this request and replay on it, and we also can see it in wireshark:

```
[01/09/22] seed@VM: ~/.../volumes$ ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=18.1 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=20.3 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=17.3 ms
64 bytes from 1.2.3.4: icmp_seq=4 ttl=64 time=24.2 ms
64 bytes from 1.2.3.4: icmp_seq=5 ttl=64 time=16.4 ms
64 bytes from 1.2.3.4: icmp_seq=6 ttl=64 time=17.8 ms
^C
--- 1.2.3.4 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5009ms
rtt min/avg/max/mdev = 16.440/19.034/24.237/2.601 ms
```



The image shows a Wireshark packet capture window with the filter 'icmp'. The packet list shows 11 packets. The first 6 packets are requests from 10.0.2.4 to 1.2.3.4, and the next 5 are replies from 1.2.3.4 to 10.0.2.4. The packet details pane shows the selected packet (No. 1) as an ICMP Echo (ping) request with ID 0x0013 and sequence 1.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request id=0x0013, seq=1/...
2	0.018120	1.2.3.4	10.0.2.4	ICMP	100	Echo (ping) reply id=0x0013, seq=1/...
3	1.002220	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request id=0x0013, seq=2/...
4	1.022460	1.2.3.4	10.0.2.4	ICMP	100	Echo (ping) reply id=0x0013, seq=2/...
5	2.004620	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request id=0x0013, seq=3/...
6	2.021940	1.2.3.4	10.0.2.4	ICMP	100	Echo (ping) reply id=0x0013, seq=3/...
7	3.005900	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request id=0x0013, seq=4/...
8	3.030180	1.2.3.4	10.0.2.4	ICMP	100	Echo (ping) reply id=0x0013, seq=4/...
9	4.006690	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request id=0x0013, seq=5/...
10	4.023100	1.2.3.4	10.0.2.4	ICMP	100	Echo (ping) reply id=0x0013, seq=5/...
11	5.008890	10.0.2.4	1.2.3.4	ICMP	100	Echo (ping) request id=0x0013, seq=6/...

In conclusion:

We learned a lot in this assignment, we deeply understood what is the meaning of spoof packets, ARP protocol telnet and more, this task was learnable and fun!