

CS 4990-04 Network and Web Security
Packet Sniffing & Spoofing Lab Report
Due: June 10 2019

Jay Chen
Nick Sacayan

Lab Task Set 1: Using Tools to Sniff and Spoof Packets

Task 1.1: Sniffing Packets

Task 1.1A.

This task involved running provided sample code with and without root privilege. The sample code provided by the lab is as follows:

```
from scapy.all import *

def print_pkt(pkt):
    pkt.show()

pkt = sniff(filter="icmp",prn=print_pkt)|
```

If the code is run with root privilege, the following output is produced:

```
[06/10/19]seed@VM:~$ sudo python sniffer.py
[sudo] password for seed:
#### Ethernet ####
  dst      = 08:00:27:7c:e3:4d
  src      = 08:00:27:f4:06:98
  type     = 0x800
#### IP ####
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 84
  id       = 51896
  flags    = DF
  frag     = 0
  ttl      = 64
  proto    = icmp
  chksum   = 0x57e4
  src      = 10.0.2.6
  dst      = 10.0.2.7
  \options \
#### ICMP ####
  type     = echo-request
  code     = 0
  chksum   = 0xe745
```

This output shows a cut of the first packet captured by the sniffer. The IP and part of the ICMP headers are visible.

If the code is run without root privilege, the following output is produced:

```
[06/06/19]seed@VM:~/Documents$ python sniffer.py
Traceback (most recent call last):
  File "sniffer.py", line 6, in <module>
    pkt = sniff(filter="icmp",prn=print_pkt)
  File "/home/seed/.local/lib/python2.7/site-packages/scapy/sendrecv.py", line 731, in sniff
    *arg, **karg)] = iface
  File "/home/seed/.local/lib/python2.7/site-packages/scapy/arch/linux.py", line 567, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type))
  File "/usr/lib/python2.7/socket.py", line 191, in __init__
    sock = _realsocket(family, type, proto)
socket.error: [Errno 1] Operation not permitted
[06/06/19]seed@VM:~/Documents$
```

The operation is not permitted because raw sockets are not allowed without root privilege.

Task 1.1B.

This task involved setting filters using Scapy while capturing packets.

- **Capture only the ICMP packet**

```
from scapy.all import *  
  
def print_pkt(pkt):  
    pkt.show()  
  
pkt = sniff(filter="icmp",prn=print_pkt)|
```

```
###[ IP ]###  
version    = 4  
ihl        = 5  
tos        = 0x0  
len        = 84  
id         = 51896  
flags      = DF  
frag       = 0  
ttl        = 64  
proto      = icmp  
chksum     = 0x57e4  
src        = 10.0.2.6  
dst        = 10.0.2.7  
\options   \
```

The sample code provided beforehand filters out icmp already so I reused it and its results. As you can see in the proto field of the IP header, the packet received is the desired ICMP protocol.

- **Capture any TCP packet coming from a particular IP and with a destination port number 23**

```
from scapy.all import *  
  
def print_pkt(pkt):  
    pkt.show()  
  
pkt = sniff(filter="tcp and host 10.0.2.6 and dst port 23",prn=print_pkt)
```

By changing the string assigned to filter to the following, I was able to capture only tcp protocol packets from the ip 10.0.2.6 (my other VM) with the destination port of 23. Traffic on port 23 was created using a Telnet connection request from the victim VM to the attacker VM. The following output was produced:

```
^C[06/10/19]seed@VM:~$ sudo python sniffer.py  
###[ Ethernet ]###  
dst      = 08:00:27:7c:e3:4d  
src      = 08:00:27:f4:06:98  
type     = 0x800  
###[ IP ]###  
version  = 4  
ihl      = 5  
tos      = 0x10  
len      = 60  
id       = 39369  
flags    = DF  
frag     = 0  
ttl      = 64  
proto    = tcp  
chksum   = 0x88d6  
src      = 10.0.2.6  
dst      = 10.0.2.7  
\options \
```

The output shows a packet of tcp protocol coming from ip 10.0.2.6 as desired.

- **Capture packets comes from or to go to a particular subnet. You can pick any subnet, such as 128.230.0.0/16; you should not pick the subnet that your VM is attached to.**

```
from scapy.all import *  
  
def print_pkt(pkt):  
    pkt.show()  
  
pkt = sniff(filter="net 128.230.0.0/16",prn=print_pkt)
```

To capture packets from or to a particular subnet (I used the subnet address provided by the lab as an example), I simply entered net 128.260.0.0/16 to filter by the provided subnet. I obtained the following output:

```
[06/10/19]seed@VM:~$ sudo python sniffer.py  
###[ Ethernet ]###  
  dst      = 52:54:00:12:35:00  
  src      = 08:00:27:7b:f6:71  
  type     = 0x800  
###[ IP ]###  
  version  = 4  
  ihl      = 5  
  tos      = 0x0  
  len      = 60  
  id       = 10336  
  flags    = DF  
  frag     = 0  
  ttl      = 64  
  proto    = tcp  
  chksum   = 0x8e27  
  src      = 10.0.2.8  
  dst      = 128.230.247.70
```

As you can see in the dst field of the IP header, the packet captured is being sent to an address within the 128.260.0.0/16 subnet.

Task 1.2: Spoofing ICMP Packets

In this section I was tasked with editing sample code to spoof a packet. The provided code was as follows:

```
>>> from scapy.all import *  
>>> a = IP() ①  
>>> a.dst = '10.0.2.3' ②  
>>> b = ICMP() ③  
>>> p = a/b ④  
>>> send(p) ⑤  
  
.  
Sent 1 packets.
```

My edited code is as follows:

```
from scapy.all import *
a = IP()
a.dst = '10.0.2.8'      #address of my victim vm
a.src = '1.2.3.4'       #spoofed source ip
b = ICMP()
p = a/b
a.show()                #shows packet|
send(p)
```

First, I changed the destination address of the IP packet to the address of my second VM which I want to receive the packet. Second I added `a.src = '1.2.3.4'` to spoof the source address of the packet. I also added `a.show()` so I could see the packet when it sends, this step however was not necessary. My code produced the following output:

```
[06/10/19]seed@VM:~$ sudo python spoofer.py
###[ IP ]###
version    = 4
ihl        = None
tos        = 0x0
len        = None
id         = 1
flags      =
frag       = 0
ttl        = 64
proto      = hopopt
chksum     = None
src        = 1.2.3.4
dst        = 10.0.2.8
\options   \

Sent 1 packets.
```

As you can see, the outgoing packet has a source ip of 1.2.3.4 and a destination of 10.0.2.8

Source	Destination	Protocol	Length	Info
PcsCompu_7c:e3:4d	Broadcast	ARP	60	Who has 10.0.2.8? Tell 10
PcsCompu_7b:f6:71	PcsCompu_7c:e3:4d	ARP	42	10.0.2.8 is at 08:00:27:7
1.2.3.4	10.0.2.8	ICMP	60	Echo (ping) request id=6
10.0.2.8	1.2.3.4	ICMP	42	Echo (ping) reply id=6
PcsCompu_7b:f6:71	RealtekU_12:35:00	ARP	42	Who has 10.0.2.1? Tell 10
RealtekU_12:35:00	PcsCompu_7b:f6:71	ARP	60	10.0.2.1 is at 52:54:00:1

Using wireshark on the victim (receiving) VM, I was able to capture the same packet as it was received. The packet was successfully spoofed as the victim VM thinks that the packet came from 1.2.3.4 and sends a reply to 1.2.3.4.

Task 1.3: Traceroute

This section tasked us with writing our own traceroute program. To do this, we send any packet with a TTL of 1. This will cause the packet to drop at the first router which will return an error message. You then resend the packet with an incrementing TTL until the packet reaches its destination. To do this I wrote the following code:

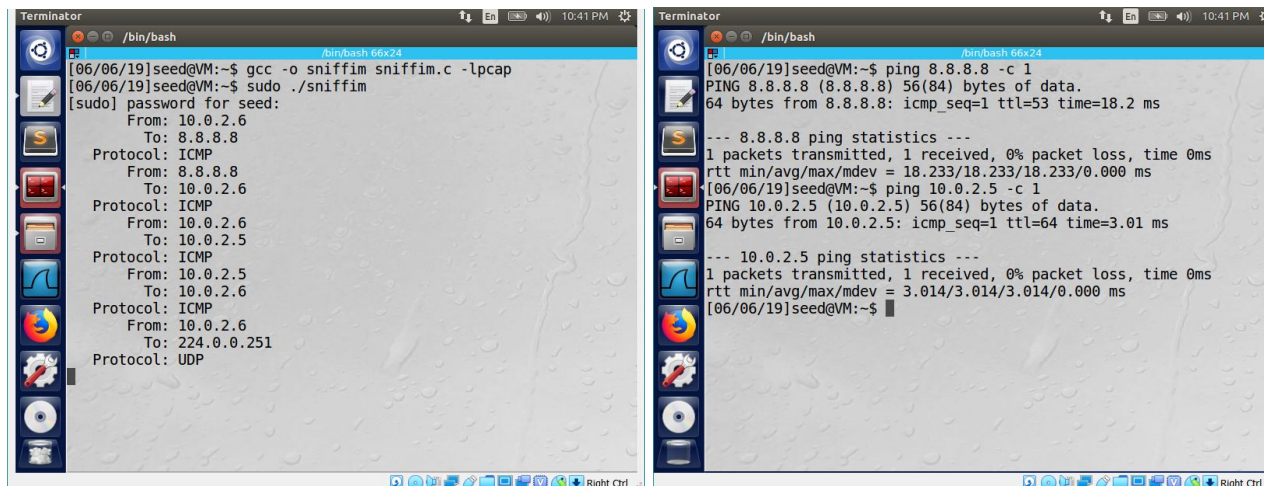
```
from scapy.all import *  
  
ttl = 1  
while ttl < 5:  
    a = IP()  
    a.dst = '1.2.3.4'  
    a.ttl = ttl  
    b = ICMP()  
    send(a/b)  
    ttl+=1
```

My code first creates an IP datagram, then sets the destination to 1.2.3.4, and finally edits the TTL. The TTL value is incremented at the end of every loop. As 1.2.3.4 is an unallocated address, there is no end destination. However the wireshark capture still provides the routers at which the packet dropped.

Source	Destination	Protocol	Length	Info
10.0.2.7	1.2.3.4	ICMP	42	Echo (ping) request
10.0.2.1	10.0.2.7	ICMP	70	Time-to-live exceeded
10.0.2.7	1.2.3.4	ICMP	42	Echo (ping) request
10.0.2.7	1.2.3.4	ICMP	42	Echo (ping) request
10.0.2.7	1.2.3.4	ICMP	42	Echo (ping) request
192.168.1.254	10.0.2.7	ICMP	70	Time-to-live exceeded
76.206.252.1	10.0.2.7	ICMP	70	Time-to-live exceeded

The wireshark capture shows return messages from multiple IP's all giving a Time-to-live exceeded error. Even though packets don't always use the same path, we can still assume a path from 10.0.2.1 to 192.168.1.254 to 76.206.252.1. I decided to only send three packets as I felt it was sufficient to demonstrate the trace route.

Task 2.1A: Sniffing Packets



This task involved using the pcap library in C to create a sniffer program that can capture packets and print the source and destination IP address. To get the correct source and destination IP address, the packet headers were typecasted to C structs so that each field could be accessed by a variable name rather than an integer offset on the packet data itself in memory.

After typecasting the sniffed packet to an Ethernet header and an IP header, the data for IP addresses could be accessed simply by calling the following code, where ip is the name of the ip header struct.

```
inet_ntoa(ip->iph_sourceip) //or use ip->iph_destip
```

The above method converts the sourceip address from network byte order to dotted address notation so that it can be easily recognized by users. Shown in the images above is the IP addresses read and printed from each packet by the attacker VM as different addresses were pinged by the victim VM.

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

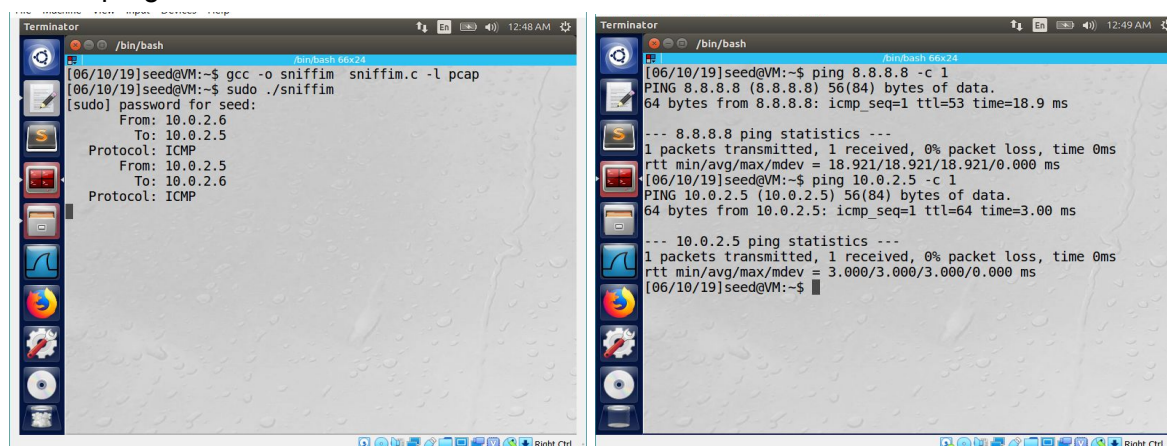
- `pcap_open_live()` - Opens a pcap session, so the sniffer program is now available for incoming packets. The method heading has a parameter to turn on promiscuous mode, which allows detection of all traffic on the network, not just packets addressed to the host.
- `pcap_compile()` - Compiles the filter, which is written as a boolean string, into BPF code that can be read by the computer.
- `pcap_setfilter()` - Sets pcap with a BPF compiled filter.
- `pcap_loop()` - Starts packet capture, processes a specified number of packets or can be looped indefinitely to listen on the network until the user quits.
- `pcap_close()` - Closes the session to release resources.

Question 2. Why do you need the root privilege to run a sniffer program? Where does the program fail if it is executed without the root privilege?

- When the program is run without root privilege in bash it results in a "Segmentation fault" error. Upon research, this error can occur in C when a program is attempting to access memory locations normally not allowed to a user. Therefore root privilege is required to run a sniffer program because enabling promiscuous mode allows the program to operate in restricted memory to receive packets from anywhere in the network.

Question 3. Please turn on and turn off the promiscuous mode in your sniffer program. Can you demonstrate the difference when this mode is on/off? Please describe how you can demonstrate this.

- Promiscuous mode is toggled on or off by setting the int parameter in the `pcap_open_live()` function to true (1) or false (0), as shown below:
- `pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf); //set to true`
- However, when tested the sniffer program still picked up traffic from either host or victim VM, and this was due to promiscuous mode being enabled in the NAT network settings for VM. When promiscuous mode was disabled in network settings and set as false in the pcap method call, then the sniffer program only detected traffic targeted to the host IP address. This is demonstrated in the images below, it is shown that when the victim VM pings IP 8.8.8.8 the packet is not captured in non-promiscuous mode, but the packet designated for the attacker VM IP is still captured. In promiscuous mode, this same test was tried in Task 2.1A, and both packets were sniffed by the program that time.



The image consists of two side-by-side screenshots of a Terminator terminal window. The left screenshot shows the compilation and execution of a sniffer program. The right screenshot shows the results of ping tests from a victim VM.

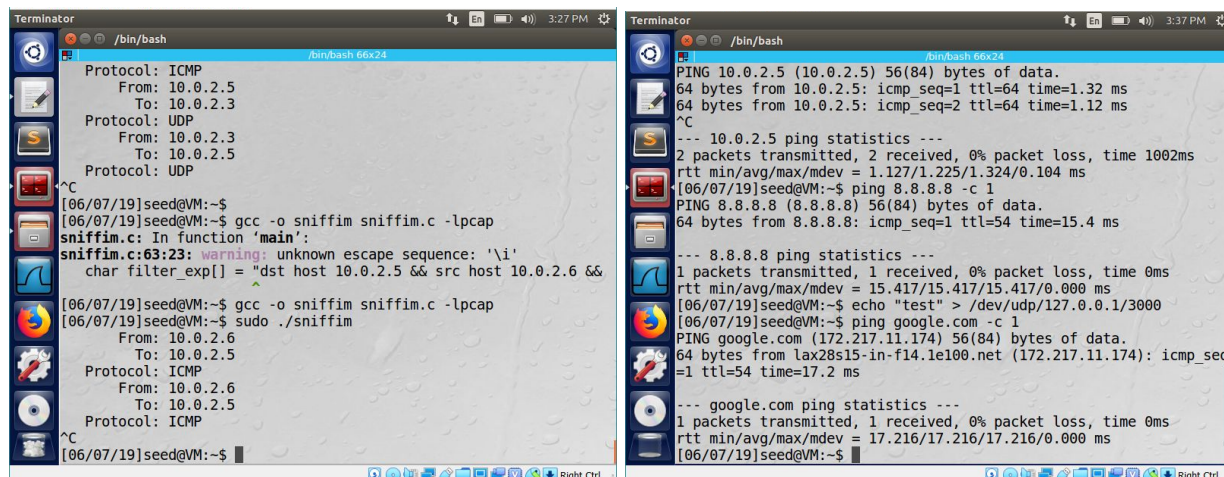
```
Terminator /bin/bash
[06/10/19]seed@VM:~$ gcc -o sniffim sniffim.c -l pcap
[06/10/19]seed@VM:~$ sudo ./sniffim
[sudo] password for seed:
From: 10.0.2.6
To: 10.0.2.5
Protocol: ICMP
From: 10.0.2.5
To: 10.0.2.6
Protocol: ICMP
```

```
Terminator /bin/bash
[06/10/19]seed@VM:~$ ping 8.8.8.8 -c 1
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=53 time=18.9 ms

--- 8.8.8.8 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 18.921/18.921/18.921/0.000 ms
[06/10/19]seed@VM:~$ ping 10.0.2.5 -c 1
PING 10.0.2.5 (10.0.2.5) 56(84) bytes of data:
64 bytes from 10.0.2.5: icmp_seq=1 ttl=64 time=3.00 ms

--- 10.0.2.5 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 3.000/3.000/3.000/0.000 ms
[06/10/19]seed@VM:~$
```

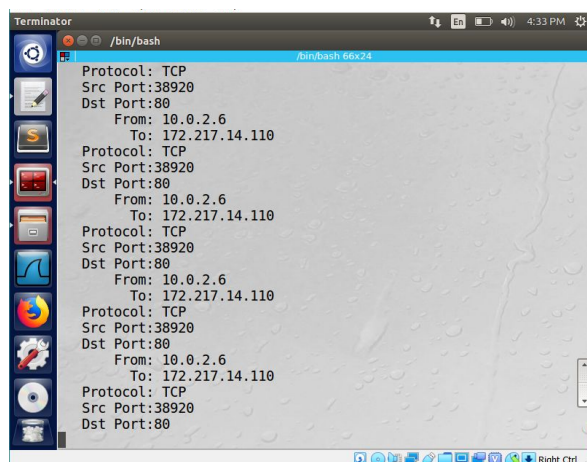

Task 2.1B: Filtering Packets



This task involved filtering packets to only capture ICMP packets with a specific destination and source IP address. Below is the boolean filter used, which specifies the correct IPs and uses "ip proto 0x01" to get ICMP packets, because 1 is the ip message protocol number for ICMP.

```
char filter_exp[] = "dst host 10.0.2.5 && src host 10.0.2.6 && ip proto 0x01";
```

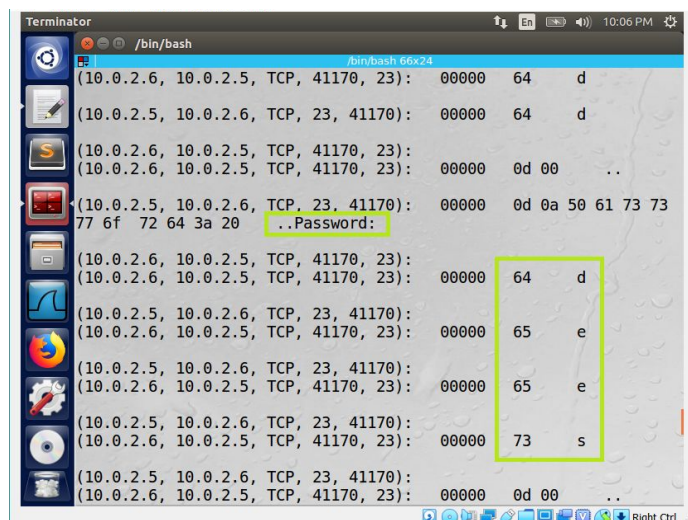
To test that the code works and filters packets as intended, we tested ping for correct & incorrect IP addresses, and tried sending a UDP request from the victim VM, but only the correct ICMP packets between the 2 specified IPs were captured by the sniffer program as intended.



The task also asked to try filtering TCP packets for a destination port between 0-100. As shown in the image, the filter (shown below) successfully captured google.com when visited by the victim VM on HTTP port 80, but not on HTTPS port 443. To specify the port, we used the command: `nc google.com [portnumber]` on the victim VM.

```
char filter_exp[] = "tcp dst portrange 0-100";
```

Task 2.1C: Sniffing Passwords



For this task, the victim VM logged into telnet, which encodes passwords as plain text ASCII characters in the payload field of the TCP packet, and the attacker VM's sniffer program decoded the TCP payload to print the password. As shown in the screenshot above, the "Password" prompt and the 4 characters "d-e-e-s" were detected as the correct telnet password by the attacker VM.

In order to get the correct payload offset, the TCP packet header was typecasted to a struct just like the Ethernet and IP headers were typecasted before.

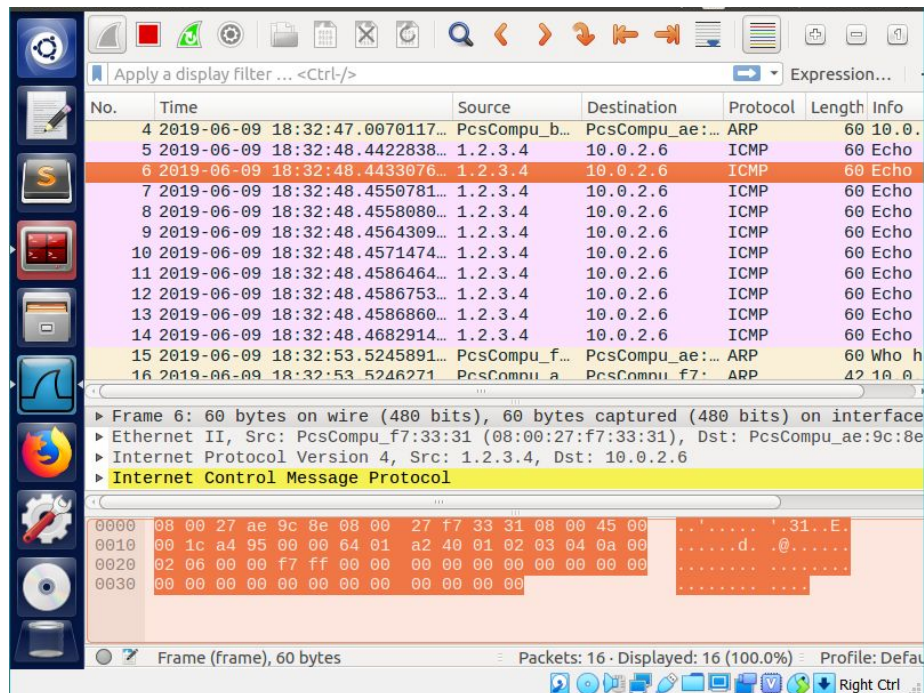
```
struct tcp_header *tcp = (struct tcp_header *) (packet + sizeof(struct ipheader) + sizeof(struct ethheader));  
int size_tcp = TH_OFF(tcp) * 4;  
payload = (u_char *) (packet + sizeof(struct ethheader) + sizeof(struct ipheader) + size_tcp);
```

The tcp_header struct contains the TH_OFF() macro, which calculates the size of the offset between the beginning of the tcp header and the payload. As demonstrated by the image, the payload starts at the correct offset from the tcp header data according to the bitwise math below.

```
u_char th_offx2; /* data offset*/  
#define TH_OFF(th) (((th) -> th_offx2 & 0xf0) >> 4)
```

The above calculation does a bitwise and operation between the offset value encoded in the tcp packet header and the hex value 0xf0, then bitwise shifts to the right by 4. Then, when the total size of the tcp header is needed, this value is multiplied by 4 to get memory addressable data.

Task 2.2A: Spoofing Packets



No.	Time	Source	Destination	Protocol	Length	Info
4	2019-06-09 18:32:47.0070117...	PcsCompu_b...	PcsCompu_ae:...	ARP	60	10.0.2.6
5	2019-06-09 18:32:48.4422838...	1.2.3.4	10.0.2.6	ICMP	60	Echo
6	2019-06-09 18:32:48.4433076...	1.2.3.4	10.0.2.6	ICMP	60	Echo
7	2019-06-09 18:32:48.4550781...	1.2.3.4	10.0.2.6	ICMP	60	Echo
8	2019-06-09 18:32:48.4558080...	1.2.3.4	10.0.2.6	ICMP	60	Echo
9	2019-06-09 18:32:48.4564309...	1.2.3.4	10.0.2.6	ICMP	60	Echo
10	2019-06-09 18:32:48.4571474...	1.2.3.4	10.0.2.6	ICMP	60	Echo
11	2019-06-09 18:32:48.4586464...	1.2.3.4	10.0.2.6	ICMP	60	Echo
12	2019-06-09 18:32:48.4586753...	1.2.3.4	10.0.2.6	ICMP	60	Echo
13	2019-06-09 18:32:48.4586860...	1.2.3.4	10.0.2.6	ICMP	60	Echo
14	2019-06-09 18:32:48.4682914...	1.2.3.4	10.0.2.6	ICMP	60	Echo
15	2019-06-09 18:32:53.5245891...	PcsCompu_f...	PcsCompu_ae:...	ARP	60	Who h
16	2019-06-09 18:32:53.5246271...	PcsCompu_a	PcsCompu_f7:	ARP	42	10.0.2.6

Frame 6: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface
 Ethernet II, Src: PcsCompu_f7:33:31 (08:00:27:f7:33:31), Dst: PcsCompu_ae:9c:8e
 Internet Protocol Version 4, Src: 1.2.3.4, Dst: 10.0.2.6
 Internet Control Message Protocol

0000 08 00 27 ae 9c 8e 08 00 27 f7 33 31 08 00 45 0031..E.
 0010 00 1c a4 95 00 00 64 01 a2 40 01 02 03 04 0a 00d. .@..
 0020 02 06 00 00 f7 ff 00 00 00 00 00 00 00 00 00 00
 0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Frame (frame), 60 bytes Packets: 16 · Displayed: 16 (100.0%) Profile: Defau

For this task, the spoof program sent spoofed packets from the false source IP address "1.2.3.4" to the victim VM. As shown in the Wireshark packet capture, the ICMP packets show a source from the spoofed IP 1.2.3.4, although they were actually sent from the attacker VM with a different IP address. To spoof the IP addresses, a raw socket was used with a typecasted IP header, and the fields for source and destination address were specified as follows:

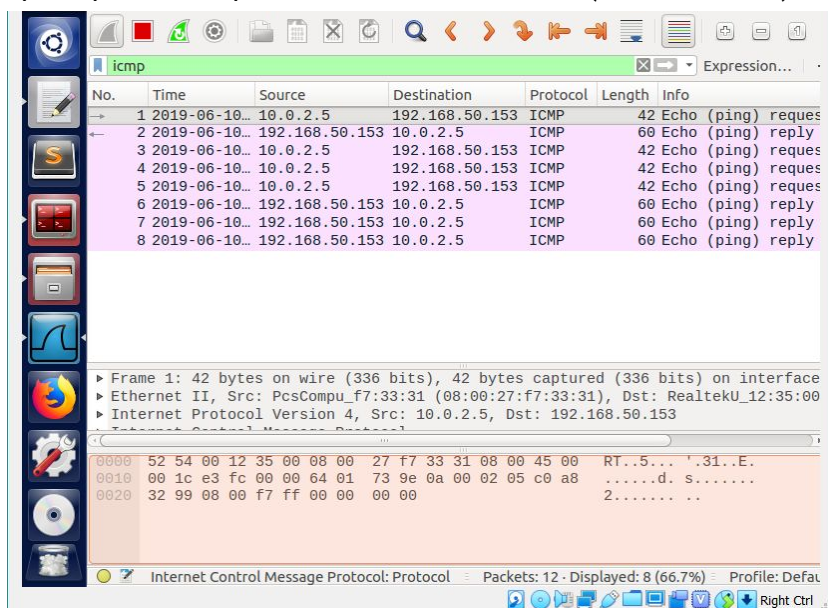
```
ip->iph_sourceip.s_addr = inet_addr("1.2.3.4");
ip->iph_destip.s_addr = inet_addr("10.0.2.6");
```

Task 2.2B: Spoofing Packets from a Remote IP

This task involved sending spoofed packets on behalf of another IP address. The "remote" address of the PC running the 2 VMs was used as the remote source, while the destination was the victim VM's IP address. The attacker VM's spoofing program sent 4 spoofed echo request packets on behalf of the specified IP. Wireshark detected the ICMP echo request as coming from the spoofed IP address, and also successfully detected the reply from the remote VM's address. Shown below is Wireshark running on the attacker VM, with both the request and reply packets.

The code used to send packets from a spoofed address is similar to the previous example, except both the destination and source IP addresses were specified.

```
ip->iph_sourceip.s_addr = inet_addr("192.168.50.153");  
ip->iph_destip.s_addr = inet_addr("10.0.2.5");
```



Question 4. Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?

- No, because there is a maximum length for IP packets of 65535 bytes, and exceeding this amount causes an overflow error and the program does not compile.

Question 5. Using raw socket programming, do you have to calculate the checksum for the IP header?

- No, the computer automatically calculates the checksum for the IP header, so it is just set to 0 by the program as a placeholder.

Question 6. Why do you need the root privilege to run the programs that use raw sockets? Where does the program fail if executed without the root privilege?

- Root privilege is needed because raw sockets can read any packets sent on the network, which is a major security risk. Without running with root privilege, the program compiles and runs through Bash, but Wireshark is unable to detect any sent packets because the program didn't have the authorization necessary to send the spoofed packets.