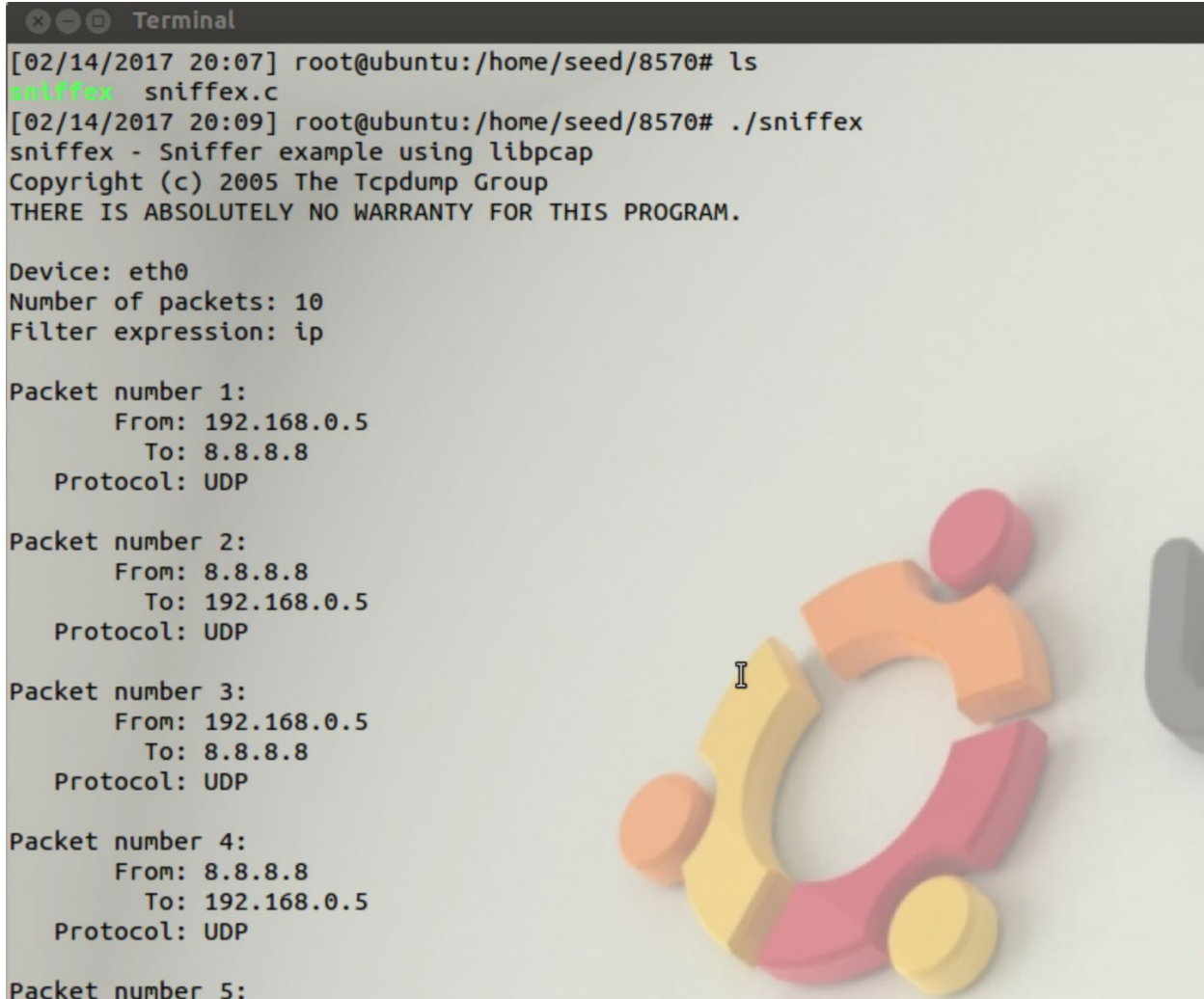


# Project 1

Yang Cao  
cao9@g.clemson.edu

## Task 1: Writing Packet Sniffing Program

### Task 1.a: Understanding sniffex.



```
Terminal
[02/14/2017 20:07] root@ubuntu:/home/seed/8570# ls
sniffex  sniffex.c
[02/14/2017 20:09] root@ubuntu:/home/seed/8570# ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: eth0
Number of packets: 10
Filter expression: ip

Packet number 1:
    From: 192.168.0.5
    To: 8.8.8.8
    Protocol: UDP

Packet number 2:
    From: 8.8.8.8
    To: 192.168.0.5
    Protocol: UDP

Packet number 3:
    From: 192.168.0.5
    To: 8.8.8.8
    Protocol: UDP

Packet number 4:
    From: 8.8.8.8
    To: 192.168.0.5
    Protocol: UDP

Packet number 5:
```

The sniffex.c program has been downloaded and compiled. It ran successfully and produced correct results which were expected.

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

1) Setting the device which we want to sniff on

Method1: The user defines the device and uses the string "dev" to represent the name of an interface which we are going to sniff on.

Method2: Pcap sets the device by itself. If this command fails, it will populate the string with an explanation of the error and save the error message in errbuf.

```
int main(int argc, char *argv[])
{
    char *dev, errbuf[PCAP_ERRBUF_SIZE];

    dev = pcap_lookupdev(errbuf);
    if (dev == NULL) {
        fprintf(stderr, "Couldn't find default device: %s\n", errbuf);
        return(2);
    }
    printf("Device: %s\n", dev);
    return(0);
}
```

## 2) Opening the device for sniffing

We use a function called `pcap_open_live()` to return us session handler.

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

The `char*` device means the device which we setted in last step. Snaplen is an integer which means the maximum number of bytes to be captured by pcap; As for promisc, “true” refers to promiscuous mode and “false” refers to non-promiscuous mode. To\_ms is the read timeout in milliseconds, we should use a non-zero timeout in this process. We can use ebuf to save error messages when it fails to open the device for sniffing.

## 3) Filtering traffic

When we finished previous steps, we use `pcap_compile()` to compile the program before applying our filter.

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

The `pcap_t *p` means the session handle. The `struct bpf_program *fp` is a reference to the position we will save the compiled filter. The string format of `char* str` is regular. The `int optimize` is an integer that determines whether the expression should be optimized or not. `Bpf_u_int32 netmask` determines the network mask of the network the filter applies to. This function returns -1 when it fails, in other cases it means success.

We can apply it after the expression has been compiled.

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

The `pcap_t *p` is our session handler, and the `struct bpf_program *fp` is a reference to the compiled version of the expression.

## 4) The actual sniffing

We can capture a single packet at a time by using `pcap_next()`.

```
u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
```

The `pcap_t *p` means the session handler, and the `struct pcap_pkthdr *h` is a pointer to a structure which contains general information of the packet. This function returns a `u_char`

pointer to the packet. We can also enter a loop which waits for n number of packets to be sniffed before being done through using pcap\_loop().

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

5) Closing the sniffer session

```
/* And close the session */  
pcap_close(handle);
```

We use this function to close the sniffer session after finishing our assignments.

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

We can compile the sniffex.c program, but it can not run successfully without the root privilege. An error message displays that no suitable device is found.

```
Terminal  
[02/15/2017 14:18] seed@ubuntu:~$ ls  
8570      elggData      openssl_1.0.1-4ubuntu5.11.debian.tar.gz  Public  
Desktop   examples.desktop  openssl_1.0.1-4ubuntu5.11.dsc            Templates  
Documents Music          openssl_1.0.1.orig.tar.gz                Videos  
Downloads openssl-1.0.1    Pictures  
[02/15/2017 14:19] seed@ubuntu:~$ cd 8570  
[02/15/2017 14:19] seed@ubuntu:~/8570$ ls  
sniffex.c  
[02/15/2017 14:19] seed@ubuntu:~/8570$ gcc -Wall -o sniffex sniffex.c -lpcap  
sniffex.c: In function 'got_packet':  
sniffex.c:486:10: warning: pointer targets in assignment differ in signedness [-Wpointer-sign]  
sniffex.c:497:3: warning: pointer targets in passing argument 1 of 'print_payload' differ in signedness [-Wpointer-sign]  
sniffex.c:373:1: note: expected 'const u_char *' but argument is of type 'const char *'  
sniffex.c:424:31: warning: variable 'ethernet' set but not used [-Wunused-but-set-variable]  
[02/15/2017 14:20] seed@ubuntu:~/8570$ ./sniffex  
sniffex - Sniffer example using libpcap  
Copyright (c) 2005 The Tcpdump Group  
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.  
  
Couldn't find default device: no suitable device found  
[02/15/2017 14:21] seed@ubuntu:~/8570$
```

It fails in the following code because pcap\_lookupdev() returns null without the root privilege.

```
else {  
    /* find a capture device if not specified on command-line */  
    dev = pcap_lookupdev(errbuf);  
    if (dev == NULL) {  
        fprintf(stderr, "Couldn't find default device: %s\n",  
                errbuf);  
        exit(EXIT_FAILURE);  
    }  
}
```



It can run successfully if we gain the root privilege.

```
Terminal
[02/14/2017 20:56] seed@ubuntu:~/8570$ su
Password:
[02/14/2017 20:58] root@ubuntu:/home/seed/8570# ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: eth0
Number of packets: 10
Filter expression: ip

Packet number 1:
    From: 192.168.0.5
    To: 8.8.8.8
    Protocol: UDP

Packet number 2:
    From: 192.168.0.5
    To: 8.8.8.8
    Protocol: UDP

Packet number 3:
    From: 8.8.8.8
    To: 192.168.0.5
    Protocol: UDP

Packet number 4:
    From: 192.168.0.5
    To: 128.230.208.76
    Protocol: TCP
    Src port: 53995
    Dst port: 80
```

### Task 1.b: Writing Filters.

- Capturing the ICMP packets between two specific hosts by setting the filter\_exp[ ] into "icmp".

```
int main(int argc, char **argv)
{
    char *dev = NULL; /* capture device name */
    char errbuf[PCAP_ERRBUF_SIZE]; /* error buffer */
    pcap_t *handle; /* packet capture handle */

    char filter_exp[] = "icmp"; /* filter expression [3] */
    struct bpf_program fp; /* compiled filter program (expression) */
    bpf_u_int32 mask; /* subnet mask */
    bpf_u_int32 net; /* ip */
    int num_packets = 10; /* number of packets to capture */

    print_app_banner();

    /* check for capture device name on command-line */
    if (argc == 2) {
        dev = argv[1];
    }
    else if (argc > 2) {
        fprintf(stderr, "error: unrecognized command-line options\n\n");
    }
}
```

Then this program is compiled, run successfully and displayed expected results.

```
Terminal
[02/14/2017 21:10] root@ubuntu:/home/seed/8570# ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: eth0
Number of packets: 10
Filter expression: icmp

Packet number 1:
  From: 192.168.0.5
  To: 172.217.4.132
  Protocol: ICMP

Packet number 2:
  From: 172.217.4.132
  To: 192.168.0.5
  Protocol: ICMP

Packet number 3:
  From: 192.168.0.5
  To: 172.217.4.132
  Protocol: ICMP

Packet number 4:
  From: 172.217.4.132
  To: 192.168.0.5
  Protocol: ICMP

Packet number 5:
  From: 192.168.0.5
  To: 172.217.4.132
```

- Capturing the TCP packets which have a destination port range from to port 10 - 100 through setting the filter\_exp[] as "tcp dst portrange 10-100".

```
int main(int argc, char **argv)
{
    char *dev = NULL; /* capture device name */
    char errbuf[PCAP_ERRBUF_SIZE]; /* error buffer */
    pcap_t *handle; /* packet capture handle */

    char filter_exp[] = "tcp dst portrange 10-100"; /* filter expression [3] */
    struct bpf_program fp; /* compiled filter program (expression) */
    bpf_u_int32 mask; /* subnet mask */
    bpf_u_int32 net; /* ip */
    int num_packets = 10; /* number of packets to capture */
}
```

Then the program is compiled, run successfully and displayed expected results.

```
Terminal
[02/14/2017 21:18] root@ubuntu:/home/seed/8570# ./sniffex
sniffex - Sniffer example using libpcap
Copyright (c) 2005 The Tcpdump Group
THERE IS ABSOLUTELY NO WARRANTY FOR THIS PROGRAM.

Device: eth0
Number of packets: 10
Filter expression: tcp dst portrange 10-100

Packet number 1:
  From: 192.168.0.5
  To: 14.215.177.37
  Protocol: TCP
  Src port: 37780
  Dst port: 80

Packet number 2:
  From: 192.168.0.5
  To: 14.215.177.37
  Protocol: TCP
  Src port: 37780
  Dst port: 80

Packet number 3:
  From: 192.168.0.5
  To: 14.215.177.37
  Protocol: TCP
  Src port: 37780
  Dst port: 80
  Payload (471 bytes):
00000  47 45 54 20 2f 20 48 54 54 50 2f 31 2e 31 0d 0a  GET / HTTP/1.1..
00016  48 6f 73 74 3a 20 77 77 77 2e 62 61 69 64 75 2e  Host: www.baidu.
```

### Task 1.c: Sniffing Passwords.

In this assignment, I changed the filter expression into “tcp and port 23” and set num\_packets to a bigger number, such as 100.

```
int main(int argc, char **argv)
{
    char *dev = NULL; /* capture device name */
    char errbuf[PCAP_ERRBUF_SIZE]; /* error buffer */
    pcap_t *handle; /* packet capture handle */

    char filter_exp[] = "tcp and port 23"; /* filter expression [3] */

    struct bpf_program fp; /* compiled filter program (expression) */
    bpf_u_int32 mask; /* subnet mask */
    bpf_u_int32 net; /* ip */
    int num_packets = 100; /* number of packets to capture */
}
```

I use vm1 to execute this program. Meanwhile, I use vm2 to login.



```
Terminal
[02/14/2017 21:47] seed@ubuntu:~$ telnet 192.168.0.5
Trying 192.168.0.5...
Connected to 192.168.0.5.
Escape character is '^]'.
Ubuntu 12.04.2 LTS
ubuntu login: seed
Password:
Last login: Tue Feb 14 21:41:31 PST 2017 from ubuntu.local on pts/4
Welcome to Ubuntu 12.04.2 LTS (GNU/Linux 3.5.0-37-generic i686)

* Documentation:  https://help.ubuntu.com/

New release '14.04.1 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

[02/14/2017 21:48] seed@ubuntu:~$ █
```

Running the sniffing program in VM1 can get the password successfully.

```
Terminal
      To: 192.168.0.5
      Protocol: TCP
      Src port: 47282
      Dst port: 23
      Payload (2 bytes):
00000  0d 00                                     ..

Packet number 31:
      From: 192.168.0.5
      To: 192.168.0.4
      Protocol: TCP
      Src port: 23
      Dst port: 47282
      Payload (12 bytes):
00000  0d 0a 50 61 73 73 77 6f 72 64 3a 20      ..Password:

Packet number 32:
      From: 192.168.0.4
      To: 192.168.0.5
      Protocol: TCP
      Src port: 47282
      Dst port: 23
      Payload (1 bytes):
00000  64
```

## Task 2: Spoofing

### Task 2.a: Write a spoofing program.

I spoofed an IP packet and sent it to 8.8.8.8, my IP address is 192.168.0.4, but I changed the source IP address into 192.168.0.5. Then the wireshark captured this packet.

```
Terminal
[02/15/2017 18:07] seed@ubuntu:~/Downloads$ ifconfig
eth0      Link encap:Ethernet  HWaddr fa:16:3e:f6:c7:cd
          inet addr:192.168.0.4  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: fe80::f816:3eff:fe6:c7cd/64 Scope:Link
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1450  Metric:1
          RX packets:4757 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3135 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4156725 (4.1 MB)  TX bytes:412760 (412.7 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:16436  Metric:1
          RX packets:4821 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4821 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:339650 (339.6 KB)  TX bytes:339650 (339.6 KB)

[02/15/2017 18:07] seed@ubuntu:~/Downloads$
[02/15/2017 18:07] seed@ubuntu:~/Downloads$ sudo ./icmp
[sudo] password for seed:
Index for interface eth0 is 2
[02/15/2017 18:08] seed@ubuntu:~/Downloads$
```

```
eth0 [Wireshark 1.6.7]
Filter:
Expression... Clear Apply

No. Time Source Destination Protocol Length Info
1 2017-02-15 18:08:00.16 192.168.0.5 8.8.8.8 ICMP 46 Echo (ping) request id=0x03e8
2 2017-02-15 18:08:05.16 fa:16:3e:f6:c7:cd fa:16:3e:6c:ca:8e ARP 42 Who has 192.168.0.1? Tell 192
3 2017-02-15 18:08:05.16 fa:16:3e:6c:ca:8e fa:16:3e:f6:c7:cd ARP 42 192.168.0.1 is at fa:16:3e:6c:

Frame 1: 46 bytes on wire (368 bits), 46 bytes captured (368 bits)
Ethernet II, Src: fa:16:3e:f6:c7:cd (fa:16:3e:f6:c7:cd), Dst: fa:16:3e:6c:ca:8e (fa:16:3e:6c:ca:8e)
Internet Protocol Version 4, Src: 192.168.0.5 (192.168.0.5), Dst: 8.8.8.8 (8.8.8.8)
  Version: 4
  Header length: 20 bytes
  Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
  Total Length: 32
  Identification: 0x217a (8570)
  Flags: 0x00

0000 fa 16 3e 6c ca 8e fa 16 3e f6 c7 cd 08 00 45 00 ..>l....>....E.
0010 00 20 21 7a 00 00 ff 01 c9 a5 c0 a8 00 05 08 08 .!Z.....
0020 08 08 08 00 2c 3e 03 e8 00 00 54 65 73 74 ...>...Test
```



## Task 2.b: Spoof an ICMP Echo Request.

In this assignment, my IP address is 192.168.0.5. I spoofed an ICMP Echo Request with the source IP address which is 192.168.0.4 to baidu.com whose IP is 111.13.101.208, and the wireshark captured the packet reply.

```
[02/15/2017 17:47] seed@ubuntu:~/Downloads$ ifconfig
eth0      Link encap:Ethernet  HWaddr fa:16:3e:2a:fe:26
          inet addr:192.168.0.5  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: fe80::f816:3eff:fe2a:fe26/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:1610 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1523 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1328203 (1.3 MB)  TX bytes:293621 (293.6 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:16436  Metric:1
          RX packets:22 errors:0 dropped:0 overruns:0 frame:0
          TX packets:22 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1868 (1.8 KB)  TX bytes:1868 (1.8 KB)

[02/15/2017 17:47] seed@ubuntu:~/Downloads$ sudo ./icmp_spoof
Index for interface eth0 is 2
[02/15/2017 17:47] seed@ubuntu:~/Downloads$
```

Time	Source	Destination	Protocol	Length	Info
2017-02-15 17:47:41.67	111.13.101.208	192.168.0.4	ICMP	46	Echo (ping) reply  id=0x03e8, seq=0/

▼ Frame 1: 46 bytes on wire (368 bits), 46 bytes captured (368 bits)

Arrival Time: Feb 15, 2017 17:47:41.675564000 PST  
Epoch Time: 1487209661.675564000 seconds  
[Time delta from previous captured frame: 0.000000000 seconds]  
[Time delta from previous displayed frame: 0.000000000 seconds]  
[Time since reference or first frame: 0.000000000 seconds]  
Frame Number: 1  
Frame Length: 46 bytes (368 bits)  
Capture Length: 46 bytes (368 bits)  
[Frame is marked: False]  
[Frame is ignored: False]

0000	fa 16 3e 99 62 ec fa 16 3e 9b b8 f5 08 00 45 28	..>.b... >.....E(
0010	00 20 83 c8 00 00 2c 01 75 63 6f 0d 65 d0 c0 a8	. .... uco.e...
0020	00 04 00 00 34 3e 03 e8 00 00 54 65 73 74	....4>... ..Test

● eth0: <live capture in progress> Fil... Packets: 1 Displayed: 1 Marked: 0 Profile: Default

## Task 2.c: Spoof an Ethernet Frame.

I setted 01:02:03:04:05:06 as the source MAC address in order to spoof an ethernet frame. Then wireshark captured this packet.

The image shows a terminal window and a Wireshark packet capture. The terminal window, titled "Terminal", shows the following commands and output:

```
Index for interface eth0 is 2
[02/15/2017 18:08] seed@ubuntu:~/Downloads$
[02/15/2017 18:32] seed@ubuntu:~/Downloads$ ls
icmp icmp4.c tcp4_ll.c
[02/15/2017 18:34] seed@ubuntu:~/Downloads$ vim tcp4_ll.c
[02/15/2017 18:36] seed@ubuntu:~/Downloads$
[02/15/2017 18:36] seed@ubuntu:~/Downloads$ gcc tcp4_ll.c -o mac_spoof
[02/15/2017 18:37] seed@ubuntu:~/Downloads$ sudo ./mac_spoof
[sudo] password for seed:
MAC address for interface eth0 is 01:02:03:04:05:06
Index for interface eth0 is 2
[02/15/2017 18:37] seed@ubuntu:~/Downloads$
[02/15/2017 18:37] seed@ubuntu:~/Downloads$ vim tcp4_ll.c
[02/15/2017 18:37] seed@ubuntu:~/Downloads$
[02/15/2017 18:37] seed@ubuntu:~/Downloads$ gcc tcp4_ll.c -o mac_spoof
[02/15/2017 18:37] seed@ubuntu:~/Downloads$ sudo ./mac_spoof
MAC address for interface eth0 is 01:02:03:04:05:06
Index for interface eth0 is 2
[02/15/2017 18:38] seed@ubuntu:~/Downloads$
```

The Wireshark window, titled "Capturing from eth0 [Wireshark 1.6.7]", shows a packet capture. The packet list shows a single packet (No. 1) at time 2017-02-15 18:38:42.04, from source 192.168.1.132 to destination 192.168.0.1, protocol TCP, length 54 bytes. The packet details pane shows the following information:

- Frame 1: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
- Ethernet II, Src: Woonsang\_04:05:06 (01:02:03:04:05:06), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
- Internet Protocol Version 4, Src: 192.168.1.132 (192.168.1.132), Dst: 192.168.0.1 (192.168.0.1)
- Version: 4
- Header length: 20 bytes
- Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
- Total Length: 40
- Identification: 0x0000 (0)
- Flags: 0x00

The packet bytes pane shows the raw data of the packet:

```
0000 ff ff ff ff ff 01 02 03 04 05 06 08 00 45 00 .....E.
0010 00 28 00 00 00 00 ff 06 38 fa c0 a8 01 84 c0 a8 ..(.....8.....
0020 00 01 00 3c 00 50 00 00 00 00 00 00 00 50 02 ...<.P. ....P.
0030 ff ff 2c 81 00 00 .....
```

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

No, we can not do that. If we want to compile it successfully, the IP packet length field should be the same as the exact length of a whole IP packet.

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

No, we don't have to calculate the checksum in case that we use the raw socket programming. A major reason is the system can be able to fill it.

**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?

We need the root privilege to run the programs that use raw sockets, since the socket() function needs the root privilege to get access to devices. Otherwise, we will get the error "socket() error: Operation not permitted".

It fails in the following code:

```
// Submit request for a socket descriptor to look up interface.
if ((sd = socket (PF_PACKET, SOCK_RAW, htons (ETH_P_ALL))) < 0) {
    perror ("socket() failed to get socket descriptor for using ioctl() ");
    exit (EXIT_FAILURE);
}
```