Task 1: Sniffing packets using Scapy

(a) With root privileges: No problem, was able to sniff all packets.

```
test1@medBlueS:~/Documents$ sudo python3 sniffer.py
###[ Ethernet ]###
  dst       = 52:54:00:12:35:02
  src       = 08:00:27:cb:ce:76
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 44083
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x7257
     src       = 10.0.2.15
     dst       = 8.8.8.8
     \options   \
###[ ICMP ]###
```

Without root privileges: Got “Permission Error: Operation not permitted”. The error seems to be connected to socket creation since the error messages include socket.py as a reference to a file that creates an error. The socket.py file caught my attention first, but it seems that scapy needs to be able to access other files/libraries in order to execute correctly.

```
test1@medBlueS:~/Documents$ python3 sniffer.py
Traceback (most recent call last):
  File "sniffer.py", line 7, in <module>
    pkt = sniff(prn=print_pkt)
  File "/usr/local/lib/python3.7/dist-packages/scapy/sendrecv.py", line 1036, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.7/dist-packages/scapy/sendrecv.py", line 907, in _run
    *arg, **karg)] = iface
  File "/usr/local/lib/python3.7/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/python3.7/socket.py", line 151, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
```

(b) I. Capturing ICMP Packets.

For this part I pinged Google’s 8.8.8.8 server using the command line.

```
root@kali:~/Documents# 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=115 time=29.9 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=115 time=29.5 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=115 time=33.5 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=115 time=29.2 ms
```

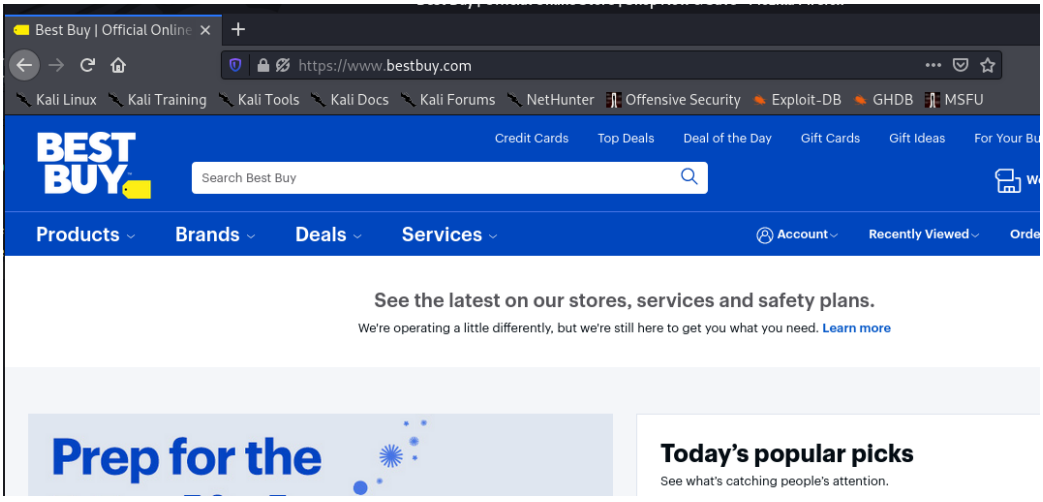
This resulted in the following output:

```
###[ Ethernet ]###
  dst       = 52:54:00:12:35:02
  src       = 08:00:27:7c:8e:8e
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 28390
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xafa4
     src       = 10.0.2.15
     dst       = 8.8.8.8
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x3dce
        id        = 0xb16e
        seq       = 0x17
###[ Raw ]###
           load      = '\x92:\x8e_\x00\x00\x00\x00(?\x01\x00\x00\x0
)*+,-./01234567'
```

In the picture above you can see the destination IP (dst) of Google's server. Along with the type of ICMP packet, echo-request. There were more packets outputted, but this is a picture of just one echo request sent to the server.

II. Capture TCP Packets

To capture TCP Packets, I used the browser to visit a website, I opted to use *bestbuy.com*.



I used Wireshark to identify the IP address of the website, to confirm that the website was indeed responding through port 443 and to identify through which port my VM was receiving the packets. In this case, I selected port 53180 from among other ports.

1277	24.437625911	173.222.252.188	10.0.2.15	TCP	60 443 → 53180 [FIN, ACK] Seq=
1278	24.440939889	10.0.2.15	173.222.252.188	TLSv1.2	93 Application Data
1279	24.441193045	10.0.2.15	173.222.252.188	TLSv1.2	78 Application Data
1280	24.441208777	10.0.2.15	173.222.252.188	TCP	54 53180 → 443 [FIN, ACK] Seq=
1281	24.441444924	173.222.252.188	10.0.2.15	TCP	60 443 → 53180 [ACK] Seq=9367
1282	24.441477021	173.222.252.188	10.0.2.15	TCP	60 443 → 53180 [ACK] Seq=9367
1283	24.441483499	173.222.252.188	10.0.2.15	TCP	60 443 → 53180 [ACK] Seq=9367

The following is a screenshot of the *sniffer.py* output.

```
###[ Ethernet ]###
  dst       = 08:00:27:7c:8e:8e
  src       = 52:54:00:12:35:02
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 40
     id        = 9294
     flags     =
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x9fd8
     src       = 173.222.252.188
     dst       = 10.0.2.15
     \options   \
###[ TCP ]###
        sport     = https
        dport     = 53180
        seq       = 27050645
        ack       = 2544827340
        dataofs   = 5
        reserved  = 0
        flags     = A
        window    = 65535
        chksum    = 0xd005
        urgptr    = 0
        options   = []
###[ Padding ]###
           load      = '\x00\x00\x00\x00\x00\x00'
```

Above you can see the src IP of the website, and that it is responding through the default *https* port 443 to port 53180 in my VM.

Task 2: Writing a C-Program to Sniff Packets

(a)I. In your own words, describe the sequence of the library calls that are essential for sniffer programs. (Just summary only)

First there needs to be a pcap session created and activated using the function:

pcap_open_live(device, max_bytes,mode,time_out);

In this function we include the device name, which can be found using ifconfig and can vary for each device. I used kali Linux and was able to find that my device name was *eth0.* The second parameter *max_bytes* represents the maximum bytes that can be captured. The third parameter *mode* can be defined using 0 for non-promiscuous mode, and 1 for promiscuous mode. Non-promiscuous is set when destinations are to be filtered out, which is our case. The last parameter *time_out* is used to define the time out time in milliseconds. This function creates a pcap_t object which is used as a parameter for the next function.

pcap_compile(handle,reference_filter,filter,optimized);

This function compiles the handle and filter. The second parameter is a reference/pointer to the compiled version of the filer, the filter parameter represents the actual filter in string form. The last parameter defines whether the function would be optimized or kept as a standard.

pcap_setfilter(handle,reference_filter);

This function applies the actual filter into our handle. It uses the variables used previously.

pcap_loop(handle,packets_sniff,callback_fxn,callback_args);

This function calls our function, *got_packet*, every time a packet is received.

pcap_freecode(reference_filter);

Frees allocated memory of the compiled filter.

pcap_close(handle);

Closes the created pcap handle.

It is important to note that instead of *pcap_loop*, there are other functions such as *pcap_nex* or *pcap_dispatch*.

II. 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?

For this C file, the errors that surfaced without root privileges are a little bit different than those from the scapy python file.

```
test1@medBlueS:~/Documents$ ./mySniffer
Welcome to my Sniffer
Press 1 to Capture all packets
Press 2 to capture ICMP packets
Press 3 to capture TCP Packets from ports x to y
Press Ctrl+C to exit anytime
1
Capturing all packets
Device not found test1@medBlip r
```

The device was not found in all my attempts to execute without root privileges. This is due because root permissions are needed to access the network devices (eth0, etc). After executing with sudo privileges again there was no problem capturing packets.

(b) Filter expressions are presented to the user in a menu like form.

I. Menu option 2. Captures ICMP packets coming from the address of my localhost (10.0.2.15). I decided not to define the receiver host because I pinged Google's 8.8.8.8 when I tested my script. The filter expression is defined below:

```
if(selection == 2){
 //I used my localhost IP, change to match localhost
 printf("Capturing ICMP from localhost\n");
 strcpy(filter_exp, "icmp and src host 10.0.2.15");
```

To define a receiver host, add this the the filer_exp: "*and dst host [receiver_ip]*"

II. Menu option 3. Captures TCP packets coming from a user defined port range. The code checks for a valid range from low to a high numbered port.

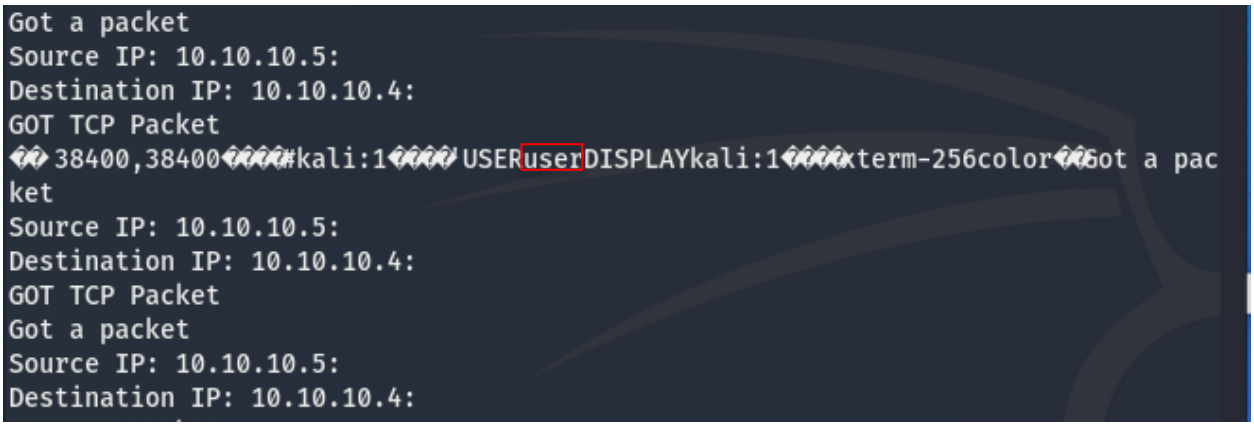
```
}else if(selection == 3){
 //variables to save lower and upper limit for port range
 int low_limit;
 int high_limit;

 printf("Input Port Lower Limit:\n");
 scanf("%d",&low_limit);
 printf("Input Port Upper Limit:\n");
 scanf("%d",&high_limit);
 if(low_limit < high_limit || low_limit == high_limit){
     printf("Valid port range\n");
```

Menu option 1. Captures all packets coming to the localhost.

(c) To sniff telnet password, a user needs to select Menu option 3 to limit the scope of the sniffer to TCP packets. The *got_packet* function prints TCP packet contents. A successful telnet connection password can be captured in plaintext as shown below. First, I selected to only capture packets from port 23, the commonly used port for telnet connections.

```
root@kali:~/Documents# ./mySniffer
Welcome to my Sniffer
Press 1 to Capture all packets
Press 2 to capture ICMP packets
Press 3 to capture TCP Packets from ports x to y
Press Ctrl+C to exit anytime
3
Input Port Lower Limit:
23
Input Port Upper Limit:
23
Valid port range
```

Then, after the client and host exchanged a correct set of credentials, I was able to see the password in plaintext as part of a gathered TCP packet as shown below:



*The password is highlighted with a red box.

Resources:

(1) https://0xbharath.github.io/art-of-packet-crafting-with-scapy/scapy/sniffing/index.html
(2) http://www.tcpdump.org/pcap.htm.