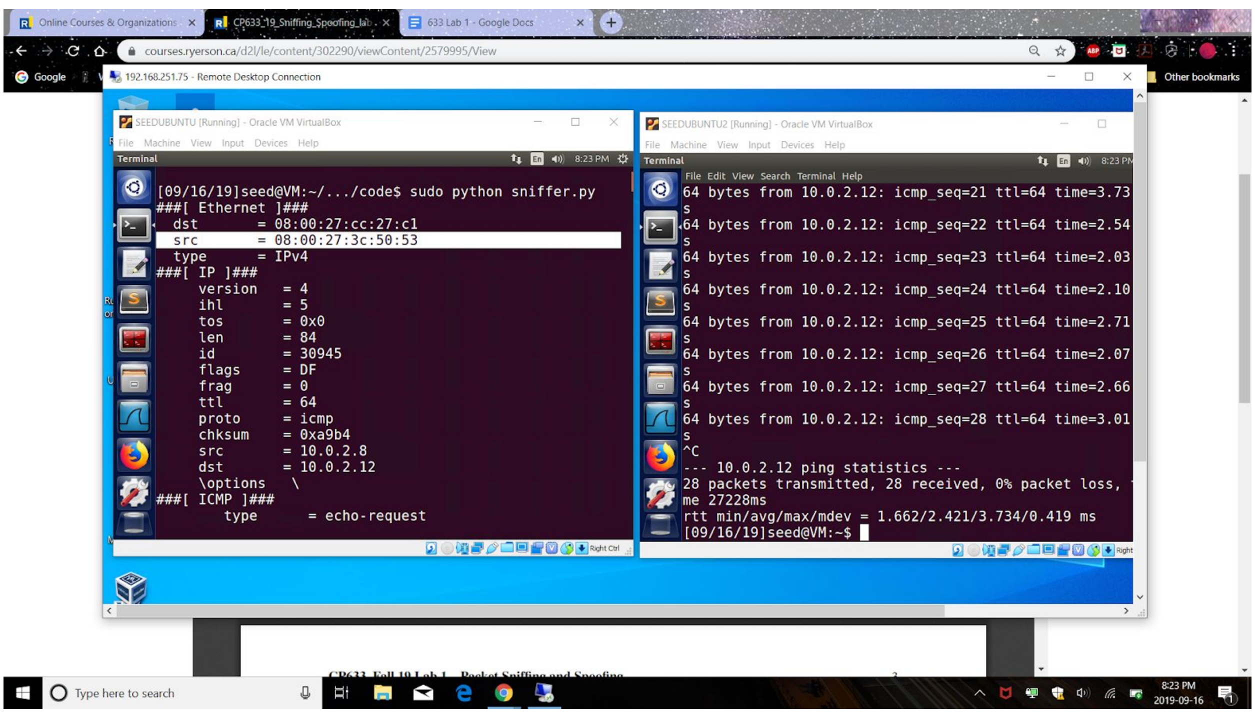
Ryerson University
CPS-633 Lab 1 Report
Packet Sniffing and Spoofing Lab
Group 2

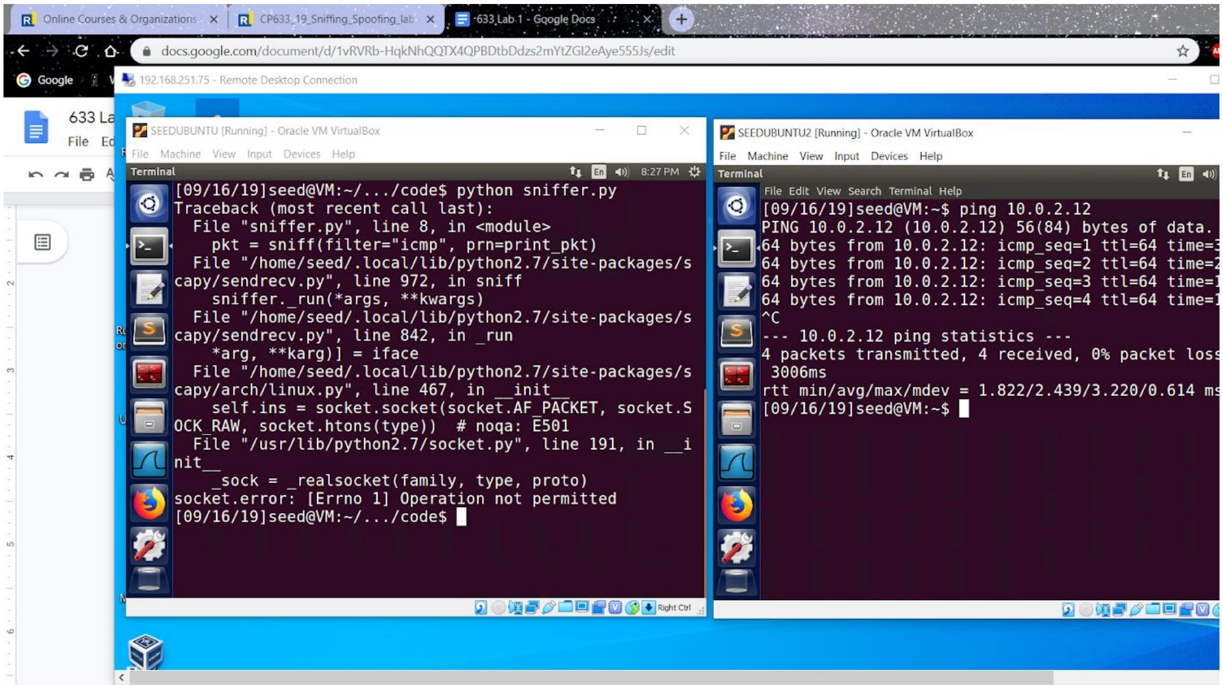
Nick: 647-781-8275
Matt: 500805168
Thomas: 500865018
Christopher: 500840595

September 22, 2019

Task 2.1a
The sniffer.py program sniffs packets. For each captured packet, the callback function print pkt() will be invoked; this function will print out some of the information about the packet. Run the program with the root privilege and demonstrate that you can indeed capture packets. After that, run the program again, but without using the root privilege; describe and explain your observations.



- After pinging the first VM (left) from the second VM (right) , then running sudo python sniffer.py on the second, the following output was produced. It consisted of information about the packet such as it’s source and destination, along with several other pieces of data. This data was separated into four parts: ethernet, IP, ICMP, and raw.



- After pinging the first VM (left) from the second VM (right) running python sniffer.py without the elevated privileges granted from sudo, the operation was not permitted, resulting in an error message. This is because in order to run sniffer.py to monitor traffic, it requires root privileges of sudo (superuserdo).

Task 2.1b

Usually, when we sniff packets, we are only interested in certain types of packets. We can do that by setting filters in sniffing. Scapys filter use the BPF (Berkeley Packet Filter) syntax; you can find the BPF manual from the Internet. Please set the following filters and demonstrate your sniffer program again (each filter should be set separately):

<u>ICMP</u>

We wrote the following program to detect and display icmp packets. We left it running, and then pinged various websites from another shell. Using this filter, only the icmp packets were logged.

sniffer.py:

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

Terminal:

```
$ sudo python3 sniffer.py
###[ Ethernet ]###
  dst        = 58:6d:8f:f9:5e:be
  src        = ac:bc:32:82:0f:33
  type       = IPv4
###[ IP ]###
     version    = 4
```

```
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 39339
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x84e2
     src       = 10.0.0.63
     dst       = 172.217.165.3
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xcfbe
        id        = 0x992a
        seq       = 0x0
###[ Raw ]###
           load      =
']\x85;\xea\x00\x06\n\x9e\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1
a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

TCP

We modified the program to only pick up packets with the host 10.0.0.63, our local IP used for testing. Note that the filter will allow packets with this host as either the source or destination–and because this is the local host, this filter simply filters out noise on the local network that it need not be concerned with. This has the potential to be used as a sort of firewall program, blocking all requests to an application except those from specified hosts. Using this filter, only TCP packets from the IP 10.0.0.63 on port 23 were logged

sniffer.py:

```
from scapy.all import *
def print_pkt(pkt):
  pkt.show()
pkt = sniff(filter="tcp and host 10.0.0.63 and port 23",prn=print_pkt)
```

Terminal:

```
$ sudo python3 sniffer.py
###[ Ethernet ]###
  dst       = 58:6d:8f:f9:5e:be
  src       = ac:bc:32:82:0f:33
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 64
     id        = 0
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
```

```
     chksum    = 0x69c6
     src       = 10.0.0.63
     dst       = 141.117.57.46
     \options   \
###[ TCP ]###
        sport     = 61886
        dport     = telnet
        seq       = 3931366200
        ack       = 0
        dataofs   = 11
        reserved  = 0
        flags     = S
        window    = 65535
        chksum    = 0x68c0
        urgptr    = 0
        options   = [('MSS', 1460), ('NOP', None), ('WScale', 6), ('NOP', None), ('NOP', None), ('Timestamp', (1151072603, 0)), ('SAckOK', b''), ('EOL', None)]
```

<u>Subnet</u>

Yet again, we modified sniffer.py to allow packets from a specific subnet. We chose the CIDR block 32, meaning that one and only one address would be sniffed. If we reduced this number, we would be decreasing the number of significant bits, thus sniffing a wider range of addresses. So, using this filter, only packets from the subnet /32 were captured.

sniffer.py:

```
from scapy.all import *
def print_pkt(pkt):
  pkt.show()
pkt = sniff(filter="tcp and net 10.0.0.63/32",prn=print_pkt)
```

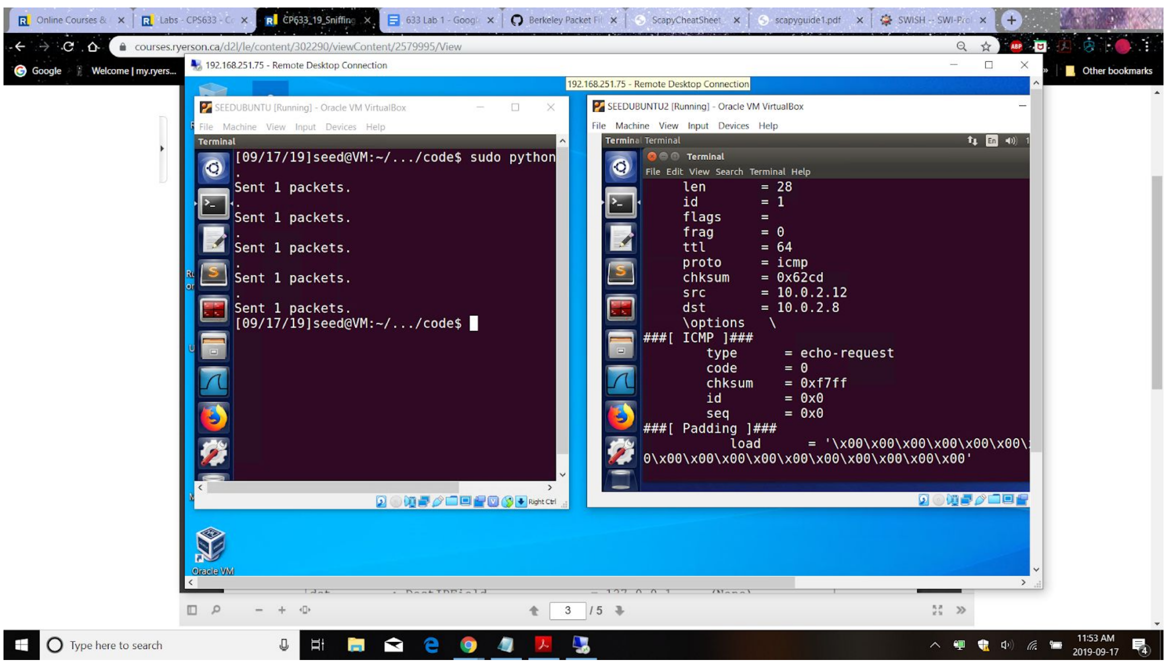
Terminal:

```
$ sudo python3 sniffer.py
###[ Ethernet ]###
  dst       = 58:6d:8f:f9:5e:be
  src       = ac:bc:32:82:0f:33
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 40
     id        = 20826
     flags     =
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0xabb9
     src       = 10.0.0.63
     dst       = 192.241.178.140
     \options   \
###[ TCP ]###
        sport     = 60685
```

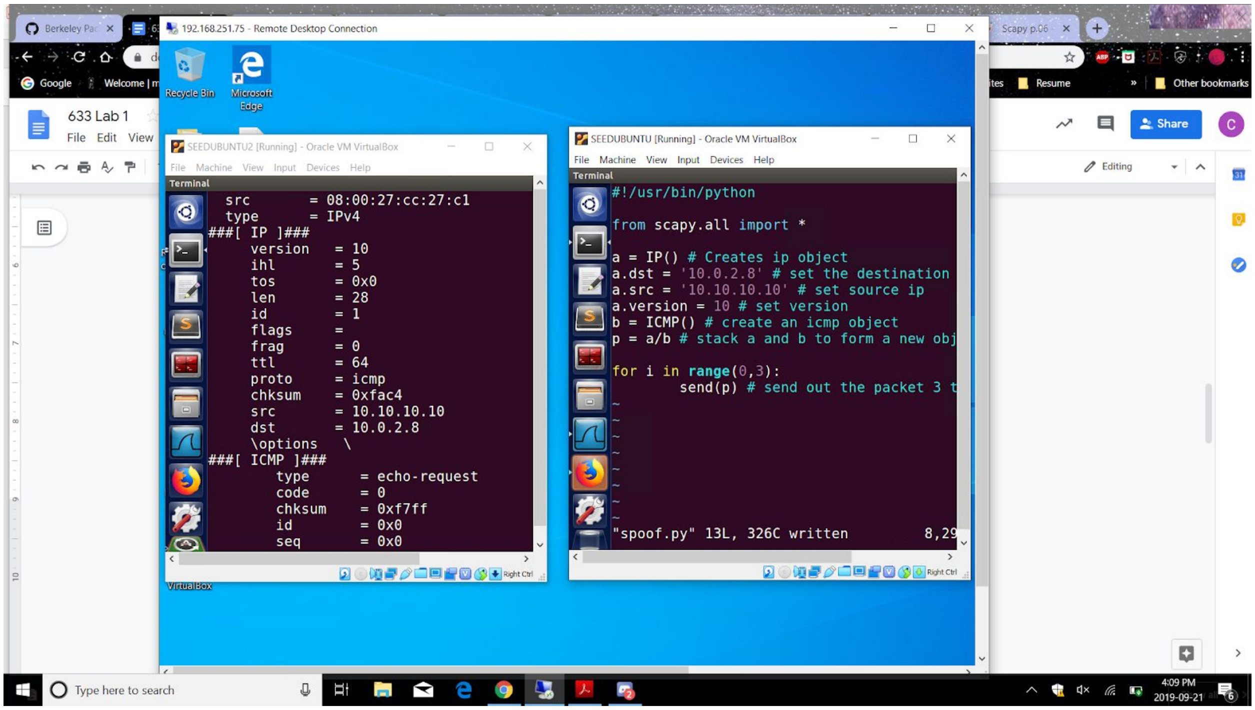
```
        dport     = https
        seq       = 2426524735
        ack       = 1702172156
        dataofs   = 5
        reserved  = 0
        flags     = A
        window    = 2048
        chksum    = 0x5afc
        urgptr    = 0
        options   = []
```

Task 2.2 Spoofing ICMP packets

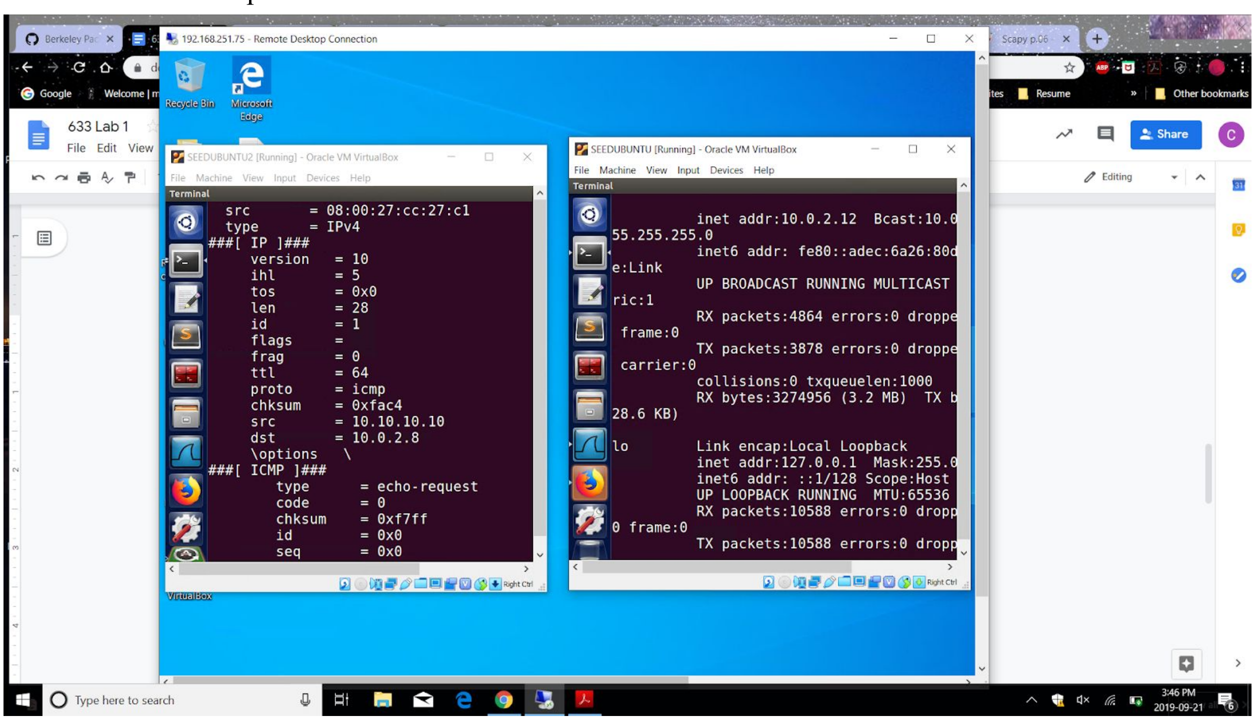
As a packet spoofing tool, Scapy allows us to set the fields of IP packets to arbitrary values. The objective of this task is to spoof IP packets with an arbitrary source IP address. We will spoof ICMP echo request packets, and send them to another VM on the same network. We will use Wireshark to observe whether our request will be accepted by the receiver. If it is accepted, an echo reply packet will be sent to the spoofed IP address.



- Example of running sniffer2.py on VM2 (right), then running spoof.py on the VM1 (left) without any changes made. As seen above, VM2 receives the packet and the source is from.



- The above code has spoof.py running on VM1 (right) with the src ip and version parameters changed to 10 and 10.10.10.10 respectively. The second vm (left) is running sniffer.py and is shown receiving the packet sent from vm1 and displaying the spoofed information. To further emphasis the point, the image below shows VM2 receiving a packet from the source 10.10.10.10, however the packet was sent from VM1 which has the address 10.0.2.12.



Task 3.0 Traceroute

The objective of this task is to use Scapy to estimate the distance, in terms of number of routers, between your VM and a selected destination. This is basically what is implemented by the traceroute tool. In this task, we will write our own tool.

Using this python program, we will programmatically increase the ttl of the packet from 1 to 100 and record the results. We stopped it at 100 because if the route really needs more than 100 packet switches, then something might be very seriously wrong with the network. The destination is google.ca (172.217.164.227). We expect the first few to fail, and to only get a successful response after the first few failures. Essentially, we just did a traceroute

traceroute.py:

```
from scapy.all import *
a = IP()
a.dst = "google.ca"
for i in range(1, 100):
  a.ttl = i
  b = ICMP()
  send(a/b)
```

Results from Wireshark:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	100.65.100.75	172.217.164.227	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=1 (no response found!)
2	0.003302	10.7.200.19	100.65.100.75	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
3	0.008183	100.65.100.75	172.217.164.227	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=2 (no response found!)
4	0.011442	100.65.96.1	100.65.100.75	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
5	0.016425	100.65.100.75	172.217.164.227	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=3 (no response found!)
6	0.019410	10.96.4.21	100.65.100.75	ICMP	186	Time-to-live exceeded (Time to live exceeded in transit)
7	0.024533	100.65.100.75	172.217.164.227	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=4 (no response found!)
8	0.027319	10.96.4.2	100.65.100.75	ICMP	186	Time-to-live exceeded (Time to live exceeded in transit)
9	0.032815	100.65.100.75	172.217.164.227	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=5 (no response found!)
10	0.036062	128.100.200.241	100.65.100.75	ICMP	182	Time-to-live exceeded (Time to live exceeded in transit)
11	0.041850	100.65.100.75	172.217.164.227	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=6 (no response found!)
12	0.043503	128.100.200.243	100.65.100.75	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
13	0.050348	100.65.100.75	172.217.164.227	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=7 (no response found!)
14	0.055334	128.100.200.201	100.65.100.75	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
15	0.060682	100.65.100.75	172.217.164.227	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=8 (no response found!)
16	0.063310	10.4.128.33	100.65.100.75	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
17	0.068634	100.65.100.75	172.217.164.227	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=9 (no response found!)
18	0.071303	10.96.7.5	100.65.100.75	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
19	0.077333	100.65.100.75	172.217.164.227	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=10 (no response found!)
20	0.079417	10.96.7.30	100.65.100.75	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
21	0.088290	100.65.100.75	172.217.164.227	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=11 (no response found!)
22	0.091370	10.16.128.2	100.65.100.75	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
23	0.097058	100.65.100.75	172.217.164.227	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=12 (no response found!)
24	0.103499	205.211.94.241	100.65.100.75	ICMP	110	Time-to-live exceeded (Time to live exceeded in transit)
25	0.104791	100.65.100.75	172.217.164.227	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=13 (no response found!)
26	0.108215	206.108.34.6	100.65.100.75	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
27	0.113971	100.65.100.75	172.217.164.227	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=14 (no response found!)
28	0.120171	74.125.244.145	100.65.100.75	ICMP	110	Time-to-live exceeded (Time to live exceeded in transit)
29	0.124418	100.65.100.75	172.217.164.227	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=15 (no response found!)
30	0.127661	216.239.41.247	100.65.100.75	ICMP	70	Time-to-live exceeded (Time to live exceeded in transit)
31	0.132612	100.65.100.75	172.217.164.227	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=16 (no response found!)
32	0.135465	172.217.164.227	100.65.100.75	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=48

So, in this example, there were 15 packet switches from end to end, with various routers handled the packet.

4.0 Sniff-and-Spoof

In this task, you will combine the sniffing and spoofing techniques to implement the following sniff-and-then-spoof program. You need two VMs on the same LAN. From VM A, you ping an IP X. This will generate an ICMP echo request packet. If X is alive, the ping program will receive an echo reply, and print out the response. Your sniff-and-then-spoof program runs on VM B, which monitors the LAN through packet sniffing. Whenever it sees an ICMP echo request, regardless of what the target IP address is, your program should immediately send out an echo reply using the packet spoofing technique.

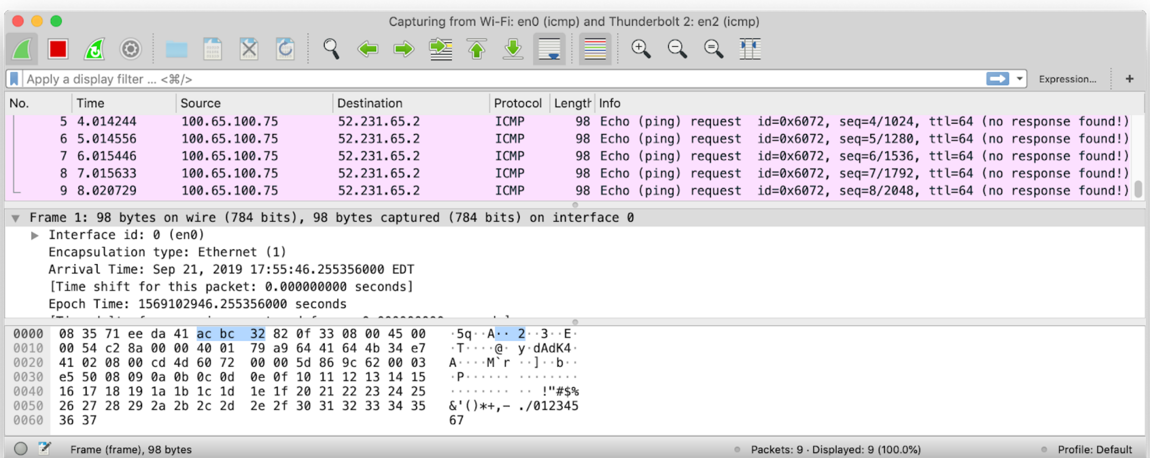
Therefore, regard- less of whether machine X is alive or not, the ping program will always receive a reply, indicating that X is alive. You need to use Scapy to do this task. In your report, you need to provide evidence to demonstrate that your technique works.

From the first vm we will ping the arbitrary host koreatimes.co.kr. This host does not respond to pings. We can verify this by pinging it:

Terminal:

```
$ ping koreatimes.co.kr
PING koreatimes.co.kr (52.231.65.2): 56 data bytes
Request timeout for icmp_seq 0
Request timeout for icmp_seq 1
Request timeout for icmp_seq 2
^C
--- koreatimes.co.kr ping statistics ---
4 packets transmitted, 0 packets received, 100.0% packet loss
```

Wireshark also verifies this:

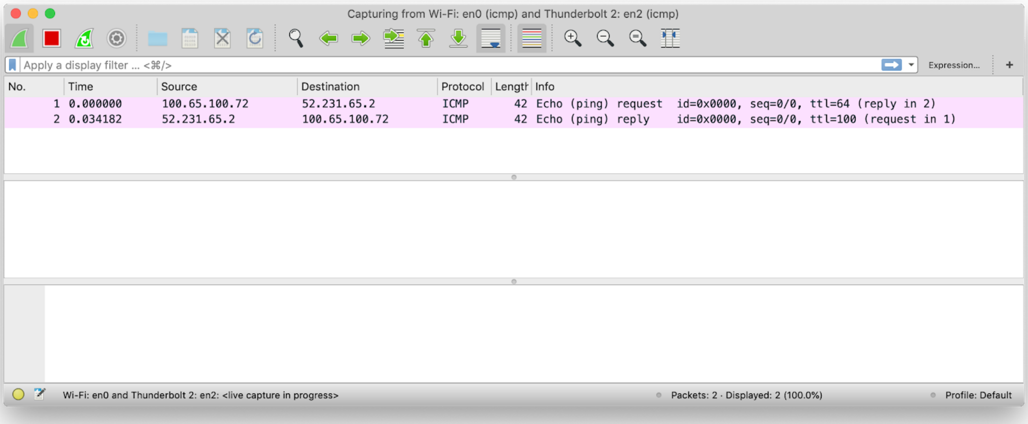


So we've confirmed that this host doesn't reply to pings. We wrote this program that will sniff the ICMP requests, craft a fake response packet, and send it back to that host.

sniff-and-spoof.py:

```
from scapy.all import *
def spoof_response(pkt):
  if (pkt[1].type == 8):
    a = IP()
    a.dst = pkt.getlayer(IP).src
    a.src = pkt.getlayer(IP).dst
    a.ttl = 100
    b = ICMP(type=0)
    p = a/b
    send(p,iface="en0")
pkt = sniff(filter="icmp", prn=spoof_response)
```

Our program generates a spoofed response with the TTL to 100. And when we ping the host again, we can see the difference. Results from wireshark:


Capturing from Wi-Fi: en0 (icmp) and Thunderbolt 2: en2 (icmp)
Apply a display filter ... <⌘/>
Expression...
No. Time Source Destination Protocol Length Info
1 0.000000 100.65.100.72 52.231.65.2 ICMP 42 Echo (ping) request id=0x0000, seq=0/0, ttl=64 (reply in 2)
2 0.034182 52.231.65.2 100.65.100.72 ICMP 42 Echo (ping) reply id=0x0000, seq=0/0, ttl=100 (request in 1)
Wi-Fi: en0 and Thunderbolt 2: en2: <live capture in progress>
Packets: 2 · Displayed: 2 (100.0%)
Profile: Default