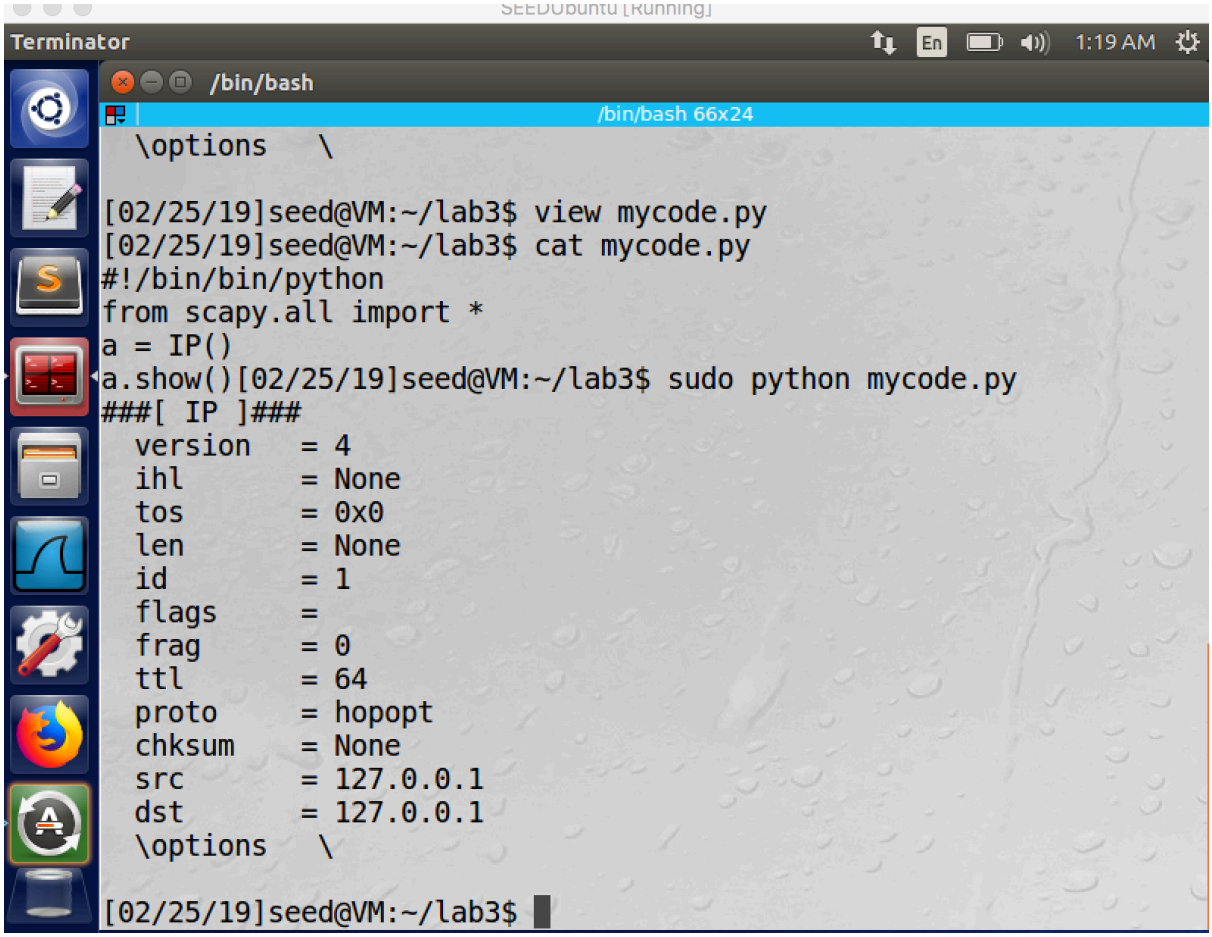
201P Lab 3: Packet Sniffing and Spoofing Lab

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



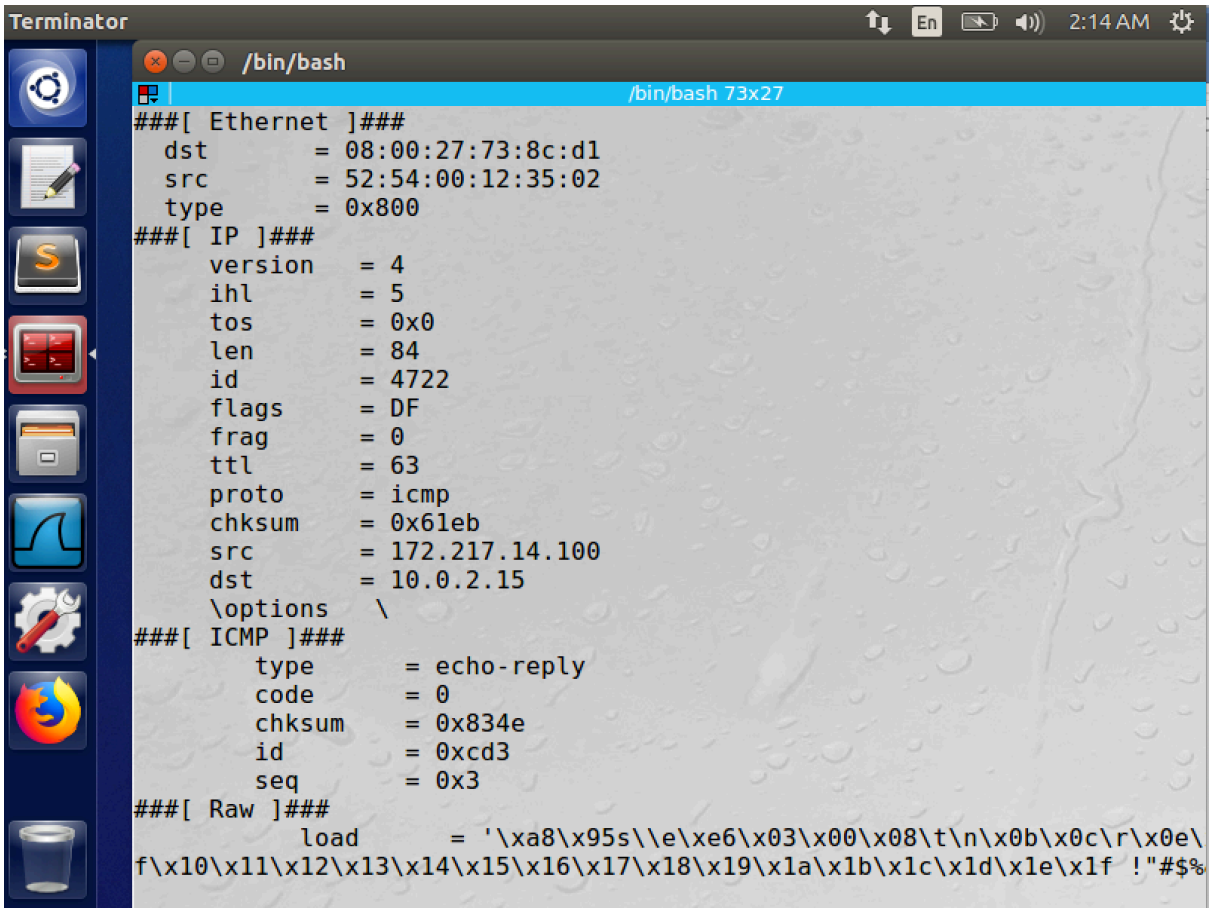
- 2.1 Task 1.1: Sniffing Packets
- Task 1.1A

```
[02/25/19]seed@VM:~/lab3$ cat sniffer.py
#!/usr/bin/python
from scapy.all import *
def print_pkt(pkt):
    pkt.show()
pkt = sniff(filter='icmp',prn=print_pkt)[02/25/19]seed@VM:~/lab3$
```

The above 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.

```
[02/25/19]seed@VM:~/lab3$ sudo python sniffer.py
```



Observations: When sudo run the program , it first waits, since there is no packets transferred by icmp. The Internet Control Message Protocol (ICMP) is a network-layer protocol that is used by hosts to perform a number of basic testing and error notification tasks, like in Ping. So, when do "ping www.google.com", it captures packets.

```
[02/25/19]seed@VM:~/lab3$ sudo python sniffer_MY1.py
0000 Ether / IP / UDP / DNS Qry "detectportal.firefox.com."
0001 Ether / IP / UDP / DNS Qry "detectportal.firefox.com."
0002 Ether / IP / UDP / DNS Qry "detectportal.firefox.com."
0003 Ether / IP / UDP / DNS Qry "detectportal.firefox.com."
0004 Ether / IP / UDP / DNS Ans "detectportal.prod.mozaws.net."
0005 Ether / IP / UDP / DNS Ans "detectportal.prod.mozaws.net."
0006 Ether / IP / UDP / DNS Ans "detectportal.prod.mozaws.net."
0007 Ether / IP / UDP / DNS Ans "detectportal.prod.mozaws.net."
0008 Ether / IP / ICMP / IPerror / UDPerror / DNS Ans "detectportal.pro
mozaws.net."
0009 Ether / IP / UDP / DNS Qry "detectportal.firefox.com."
[02/25/19]seed@VM:~/lab3$ cat sniffer_MY1.py
#!/usr/bin/python
from scapy.all import *
a=sniff(count=10)
a.nsummary()[02/25/19]seed@VM:~/lab3$
```

Observations: I also tried another packet capture, it sniffs for 10 packets when I open the firefox browser, and as soon as 10 packets have sniffed, it will print a summary of the 10 packets that were discovered.

After that, run the program again, but without using the root privilege.

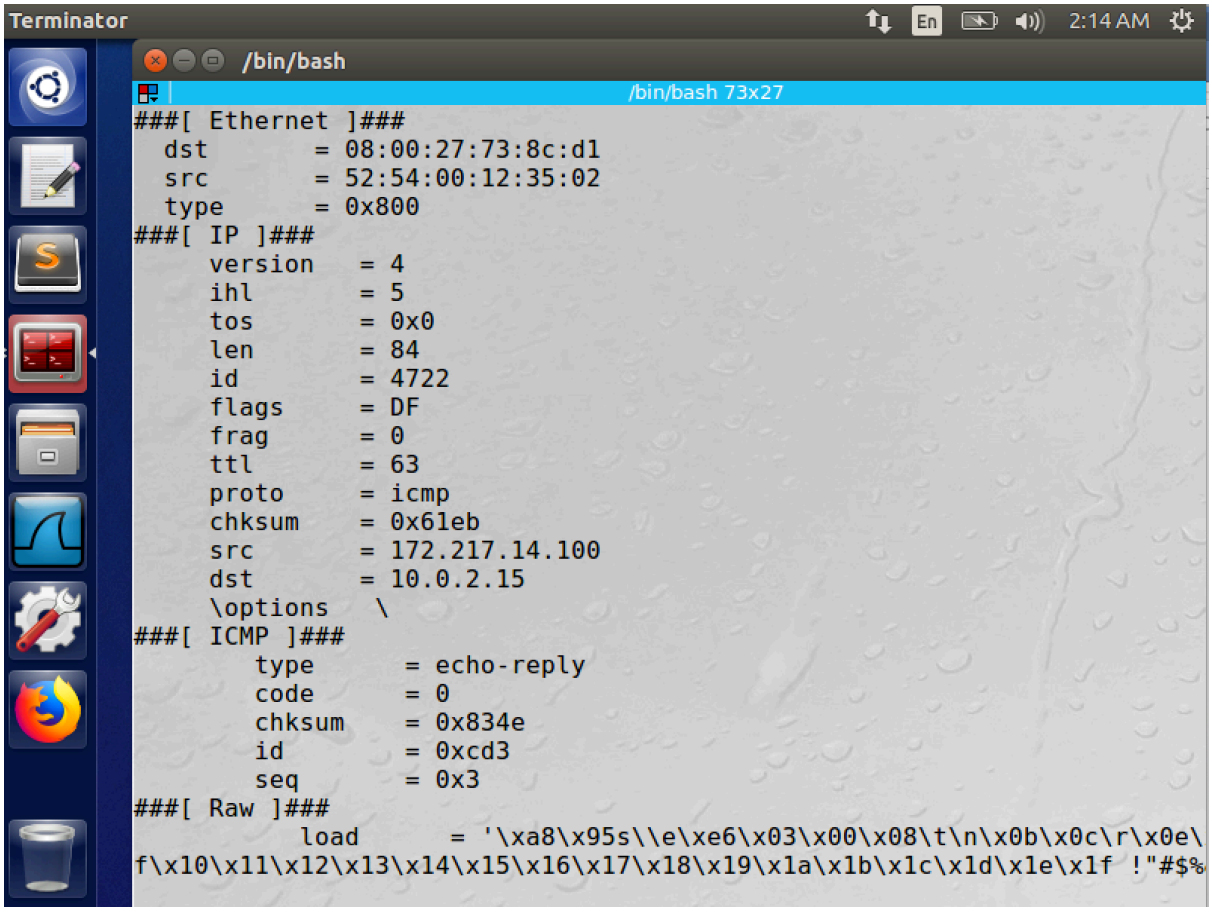
```
^C[02/25/19]seed@VM:~/lab3$ python sniffer.py
Traceback (most recent call last):
  File "sniffer.py", line 5, in <module>
    pkt = sniff(filter='icmp',prn=print_pkt)
  File "/home/seed/.local/lib/python2.7/site-packages/scapy/sendrecv.py
 line 731, in sniff
    *arg, **karg)] = iface
  File "/home/seed/.local/lib/python2.7/site-packages/scapy/arch/linux.
", line 567, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.
ons(type))
  File "/usr/lib/python2.7/socket.py", line 191, in __init__
    _sock = _realsocket(family, type, proto)
socket.error: [Errno 1] Operation not permitted
[02/25/19]seed@VM:~/lab3$
```

Observations: Running the program without root privilege, the operation is not permitted. Because spoofing packets require for root privilege. Sniff has to access the network interface card, which is the physical device that accepts the packets into the system and only root can access it.

- Task 1.1B

1) Capture only the ICMP packet

```
[02/25/19]seed@VM:~/lab3$ cat sniffer.py
#!/usr/bin/python
from scapy.all import *
def print_pkt(pkt):
    pkt.show()
pkt = sniff(filter='icmp',prn=print_pkt)[02/25/19]seed@VM:~/lab3$
```



Observations: As above, use filter='icmp' , we capture only icmp. The observation and explanations are illustrated above in detail.

2) Capture any TCP packet that comes from a particular IP and with a destination port number 23.

```
[02/25/19]seed@VM:~/lab3$ cat sniffer2.py
#!/usr/bin/python
from scapy.all import *
def print_pkt(pkt):
    pkt.show()
pkt = sniff(filter='tcp dst port 23 and src host 10.0.2.15',prn=print_pkt
)[02/25/19]seed@VM:~/lab3$
```

```
[02/25/19]seed@VM:~$ telnet google.com
Trying 216.58.193.206...
^C
```

```
###[ Ethernet ]###
  dst       = 52:54:00:12:35:02
  src       = 08:00:27:73:8c:d1
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 60
     id        = 64511
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0x7ff6
     src       = 10.0.2.15
     dst       = 172.217.5.206
     \options   \
###[ TCP ]###
        sport     = 60452
        dport     = telnet
        seq       = 3153214067L
        ack       = 0
        dataofs   = 10
        reserved  = 0
        flags     = S
        window    = 29200
```

Observations: According to BPF, Use filter=' tcp dst port 23 and src host 10.0.2.15', where src, dst means direction, port and host means port and IP. When running the python, it waits for tcp. So we command “telnet google.com”, and get the TCP packets as above.

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

```
[02/25/19]seed@VM:~$ ping google.com
PING google.com (216.58.193.206) 56(84) bytes of data.
64 bytes from lax02s23-in-f206.1e100.net (216.58.193.206): icmp_se
q=1 ttl=63 time=6.08 ms
64 bytes from lax02s23-in-f206.1e100.net (216.58.193.206): icmp_se
q=2 ttl=63 time=5.59 ms
```

```
[02/25/19]seed@VM:~/lab3$ cat sniffer3.py
#!/usr/bin/python
from scapy.all import *
def print_pkt(pkt):
    pkt.show()
pkt = sniff(filter='host 216.58.293.0/8',prn=print_pkt)[02/25/19]seed@VM:
~/lab3$
```

```
###[ Ethernet ]###
  dst       = 08:00:27:73:8c:d1
  src       = 52:54:00:12:35:02
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 5024
     flags     = DF
     frag      = 0
     ttl       = 63
     proto     = icmp
     chksum    = 0x81f1
     src       = 216.58.193.206
     dst       = 10.0.2.15
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0x604b
        id        = 0x10a1
        seq       = 0xd
###[ Raw ]###
           load      = '\xcc\xb3s\\\\\xf3\x07\x00\x08\t\n\x0b\x0c\r\x0e\x
0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&
```

Observations: In above, we can see that google.com has an ip of 216.58.293.206 (or double-check by ping), so we use this network. And use subnet of 216.58.293.0/8. Running the python, and ping google, we see the sniffer capture packets as above.

- 2.2 Task 1.2: Spoofing ICMP Packets

Since the default Internet method of VMware is NAT, so we change it to bridge to make the host and VM communicate more easily (and better in the net system out of UCI). The ip address of the host and VM are:

```
Rain-SunnydeMacBook-Pro :: 201P_CompSecu/share/3lab » ifconfig | grep "inet " | grep -v 127.0.0.1
        inet 172.20.10.13 netmask 0xfffffff0 broadcast 172.20.10.15
```

```
[02/25/19]seed@VM:~$ ifconfig | grep "inet " | grep -v 127.0.0.1
          inet addr:172.20.10.4  Bcast:172.20.10.15  Mask:255.255.
255.240
```

In the task, Scapy allows us to set the fields of IP packets (in host) to arbitrary values (in VM) on the same network.

Then we write the python code of sending a ICMP packet, running in root privilege. And it successfully sent a packet.

spoof_checkVM.py | spoof_host.py

```
1  #!/usr/bin/python
2  from scapy.all import *
3  a = IP()
4  ls(a)
5  a.dst = '172.20.10.4'
6  b = ICMP()
7  p = a/b
8  send(p)
```

```
Rain-SunnydeMacBook-Pro :: 201P_CompSecu/share/3lab » sudo python spoof_host.py
version    : BitField (4 bits)                = 4               (4)
ihl        : BitField (4 bits)                = None            (None)
tos        : XByteField                       = 0               (0)
len        : ShortField                       = None            (None)
id         : ShortField                       = 1               (1)
flags      : FlagsField (3 bits)              = <Flag 0 ()>     (<Flag 0 ()>)
frag       : BitField (13 bits)               = 0               (0)
ttl        : ByteField                        = 64              (64)
proto      : ByteEnumField                    = 0               (0)
chksum     : XShortField                      = None            (None)
src        : SourceIPField                    = '127.0.0.1'     (None)
dst        : DestIPField                      = '127.0.0.1'     (None)
options    : PacketListField                  = []              ([])
.
Sent 1 packets.
```

Before this, in VM, we have run the sniff codes similar to the former lab, detailed as below, (instead of using Wireshark to observe whether our request will be accepted by the receiver) to show that the arbitrary IP (VM) is actually the destination. We can see that VM first receive a packet from src host, and then send back. So we did spoof an ICMP echo request packet with an arbitrary source IP address.

spoof_checkVM.py

spoof_checkVM.py | spoof_host.py | sniffer.py

```
1  #!/usr/bin/python
2  from scapy.all import *
3  def print_pkt(pkt):
4      pkt.show()
5  pkt = sniff(filter='icmp',prn=print_pkt)
```

```
[02/25/19]seed@VM:~/3lab$ sudo python spoof_checkVM.py > a.log
```

```
###[ Ethernet ]###
  dst       = 08:00:27:73:8c:d1
  src       = 78:31:c1:cd:0d:ea
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 28
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xea7
     src       = 172.20.10.13
     dst       = 172.20.10.4
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xf7ff
        id        = 0x0
        seq       = 0x0
###[ Padding ]###
           load      = '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

```
           load      = '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00'

###[ Ethernet ]###
  dst       = 78:31:c1:cd:0d:ea
  src       = 08:00:27:73:8c:d1
  type      = 0x800
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 28
     id        = 65347
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xf64
     src       = 172.20.10.4
     dst       = 172.20.10.13
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0xffff
        id        = 0x0
        seq       = 0x0
```

- 2.3 Task 1.3: Traceroute

traceroute_ICMP.py

```
#!/usr/bin/python
from scapy.all import *
#hostname = '172.20.10.4'
hostname = 'www.google.com'
#hostname = '172.20.10.13'
for i in range(1, 255):
    pkt = IP(dst=hostname, ttl=i) / ICMP()
    # Send the packet and get a reply
    reply = sr1(pkt, verbose=0, timeout = 10)
    #print(reply.type)
    if reply is None:
        # No reply =(
        break
    elif reply.type == 0:
        # We've reached our destination
        print "Done!", reply.src
        break
    else:
        # We're in the middle somewhere
        print "%d hops away: " % i , reply.src
```

```
Rain-SunnydeMacBook-Pro :: 201P_CompSecu/share/3lab » sudo python traceroute_ICMP.py
Password:
1 hops away:  10.0.1.1
2 hops away:  162.1.1.249
3 hops away:  10.80.44.1
4 hops away:  68.4.14.78
5 hops away:  100.120.104.6
6 hops away:  68.1.1.171
7 hops away:  64.125.13.97
8 hops away:  64.125.28.230
9 hops away:  64.125.28.144
10 hops away:  64.125.27.189
11 hops away:  64.125.35.190
12 hops away:  203.208.173.137
13 hops away:  203.208.182.85
14 hops away:  203.208.149.137
15 hops away:  203.208.191.198
16 hops away:  202.166.123.69
17 hops away:  202.166.120.110
18 hops away:  202.166.124.46
Done! 220.255.2.153
Rain-SunnydeMacBook-Pro :: 201P_CompSecu/share/3lab »
```

Observations: The traceroute utility uses ICMP messages to determine the path a packet takes to reach another host, either on a local network or on the Internet. It accomplishes this task with a clever use of the time-

to-live (TTL) field in the IP header. First, it attempts to send a packet to the target with a TTL of 1. On receiving a packet with a TTL of 1, an intermediate router discards the packet and replies to the sender with an ICMP time exceeded message, revealing the first machine along the path to the target. Next, traceroute sends a packet with a TTL of 2. On reaching the first router in the path, the TTL is decremented by one and forwarded to the next router, which in turn sends an ICMP packet to the original sender. By incrementing the TTL field in this way, traceroute can determine each host along the path to the target.

So We sent a ICMP packet for checking the path to "www.google.com", with the time-to-live increasing, as above. Instead of using WireShark, we print the received packet from the first answer router, which consisting the state of path.

- 2.4 Task 1.4: Sniffing and-then Spoofing

```
Rain-SunnydeMacBook-Pro :: 201P_CompSecu/share/3lab » ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4): 56 data bytes
Request timeout for icmp_seq 0
Request timeout for icmp_seq 1
Request timeout for icmp_seq 2
Request timeout for icmp_seq 3
Request timeout for icmp_seq 4
```

sniff_spoof.py

```python
#!/usr/bin/python
from scapy.all import *
def spoof_pkt(pkt):
    if (pkt[2].type == 8):
    #check if the ICMP is a request
        dst = pkt[1].dst
        src = pkt[1].src
        seq = pkt[2].seq
        id = pkt[2].id

        load=pkt[3].load
        reply = IP(src=dst, dst=src)/ICMP(type=0, id=id, seq=seq)/load
        send(reply)

filter = "icmp"
pkt = sniff(filter = filter, prn=spoof_pkt)
#start sniff
```

```
[02/26/19]seed@VM:~/3lab$ sudo python sniff_spoof.py
[sudo] password for seed:
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
^C.
Sent 1 packets.
```

```
Rain-SunnydeMacBook-Pro :: 201P_CompSecu/share/3lab » ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4): 56 data bytes
Request timeout for icmp_seq 0
Request timeout for icmp_seq 1
64 bytes from 1.2.3.4: icmp_seq=0 ttl=64 time=2065.298 ms
Request timeout for icmp_seq 3
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=3107.310 ms
Request timeout for icmp_seq 5
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=4147.079 ms
Request timeout for icmp_seq 7
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=5183.535 ms
Request timeout for icmp_seq 9
64 bytes from 1.2.3.4: icmp_seq=4 ttl=64 time=6219.737 ms
Request timeout for icmp_seq 11
64 bytes from 1.2.3.4: icmp_seq=5 ttl=64 time=7261.953 ms
Request timeout for icmp_seq 13
^C
--- 1.2.3.4 ping statistics ---
15 packets transmitted, 6 packets received, 60.0% packet loss
round-trip min/avg/max/stddev = 2065.298/4664.152/7261.953/1774.041 ms
```

Observations: At first we can see that without spoof, from A the arbitrary IP X = 1.2.3.4 is not reachable. When we run spoof in VM B, and then ping X in A it becomes reachable and echo reply back to A. Also, in B, it sent packets successfully. It is because that for A and B in the LAN, the can hear the broadcast from each other. Thus the malicious B can receive the echo request and use the information in it to pretend it is the target X, since there is not much integrity and authentication check when packet tranfering.