

Packet Sniffing and Spoofing Lab

Jin Jung, Joshua Barrs, John Park

Copyright © 2006 - 2016 Wenliang Du, Syracuse University.

The development of this document was partially funded by the National Science Foundation under Award No. 1303306 and 1318814. This work is licensed under a Creative Commons Attribution-NonCommercialShareAlike 4.0 International License. A human-readable summary of (and not a substitute for) the license is the following: You are free to copy and redistribute the material in any medium or format. You must give appropriate credit. If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original. You may not use the material for commercial purposes.

1 Overview

Packet sniffing and spoofing are two important concepts in network security; they are two major threats in network communication. Being able to understand these two threats is essential for understanding security measures in networking. There are many packet sniffing and spoofing tools, such as Wireshark, Tcpdump, Netwox, Scapy, etc. Some of these tools are widely used by security experts, as well as by attackers. Being able to use these tools is important for students, but what is more important for students in a network security course is to understand how these tools work, i.e., how packet sniffing and spoofing are implemented in software.

The objective of this lab is two-fold: learning to use the tools and understanding the technologies underlying these tools. For the second object, students will write simple sniffer and spoofing programs, and gain an in-depth understanding of the technical aspects of these programs. This lab covers the following topics:

- Scapy
- Sniffing using the pcap library
- Raw socket

Lab environment. This lab has been tested on our pre-built Ubuntu 16.04 VM.

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

Many tools can be used to do sniffing and spoofing, but most of them only provide fixed functionalities.

Scapy is different: it can be used not only as a tool, but also as a building block to construct other sniffing and spoofing tools, i.e., we can integrate the Scapy functionalities into our own program. In this set of tasks, we will use Scapy for each task.

To use Scapy, we can write a Python program, and then execute this program using Python. See the following example. We should run Python using the root privilege because the privilege is required for spoofing packets. At the beginning of the program (Line À), we should import all Scapy's modules.

```
$ view mycode.py
#!/bin/bin/python

from scapy.all import *

a = IP() a.show()

$ sudo python mycode.py
####[ IP ]####
version
= 4
ihl          = None
...
```

We can also get into the interactive mode of Python and then run our program one line at a time at the Python prompt. This is more convenient if we need to change our code frequently in an experiment.

```
$ sudo python
>>> from scapy.all import *
>>> a = IP()
>>> a.show()
####[ IP ]####
version
= 4
ihl          = None
...
```

2.1 Task 1.1: Sniffing Packets

Wireshark is the most popular sniffing tool, and it is easy to use. We will use it throughout the entire lab. However, it is difficult to use Wireshark as a building block to construct other tools. We will use Scapy for that purpose. The objective of this task is to learn how to use Scapy to do packet sniffing in Python programs. A sample code is provided in the following:

```
#!/usr/bin/python
from
scapy.all import *

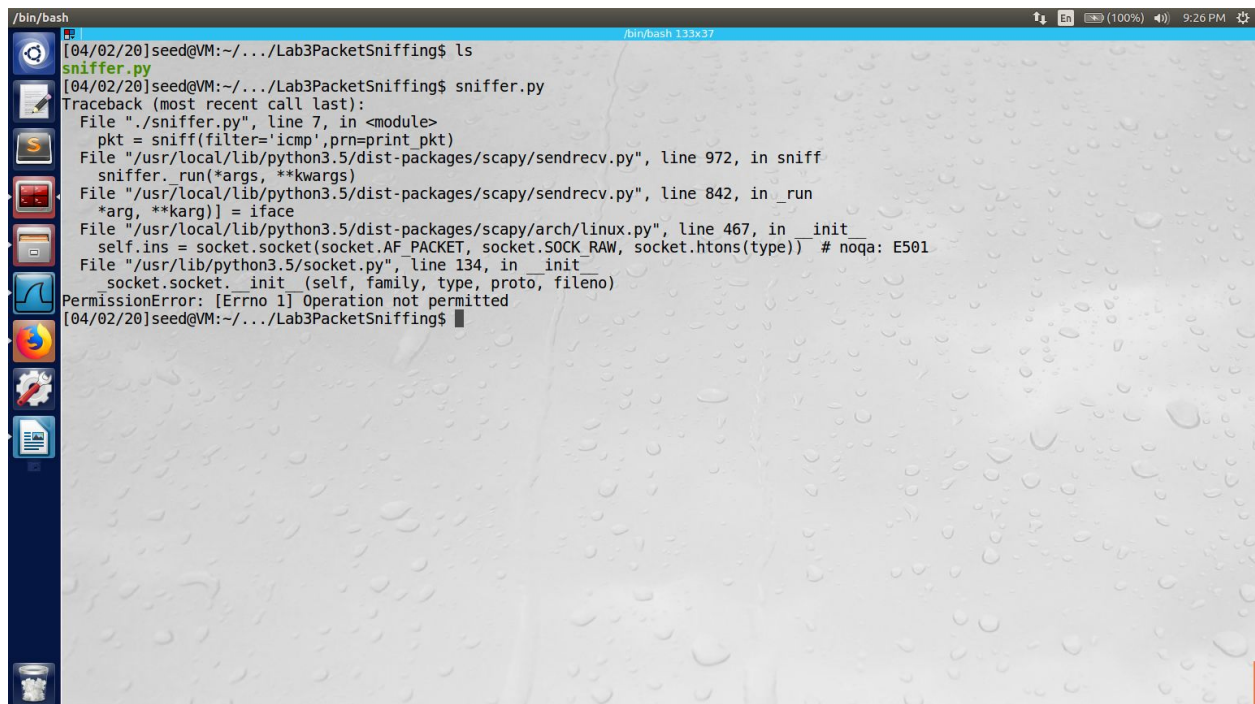
def print_pkt(pkt): pkt.show()

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

Task 1.1A. The above program sniffs packets. For each captured packet, the callback function `printpkt()` 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.

```
// Run the program with the root privilege
$ sudo python sniffer.py
```

```
// Run the program without the root privilege
$ python sniffer.py
```

A screenshot of a terminal window titled '/bin/bash' with a window size of '133x37'. The terminal shows the user 'seed@VM' in the directory '~/.../Lab3PacketSniffing'. The user runs 'ls' and then 'sniffer.py'. A traceback error is displayed, starting with 'Traceback (most recent call last):'. The error points to line 7 in 'sniffer.py' where 'pkt = sniff(filter='icmp', prn=print pkt)' is called. It then traces back through 'scapy/sendrecv.py' (line 972) and 'scapy/arch/linux.py' (line 467) to the 'socket.socket' constructor in 'socket.py' (line 134). The final error is 'PermissionError: [Errno 1] Operation not permitted'. The terminal prompt returns to '\$'.

This is what I currently have. I am honestly not sure why the `sniffer.py` python program is not executing correctly. There is another approach where I typed "`sudo python sniffer.py`." That executed but seemed to enter an infinite loop that did not produce any output.

```

^C[04/03/20]seed@VM:~/.../packetsniffinglab$ sudo python sniffer.py
###[ Ethernet ]###
  dst      = 52:54:00:12:35:02
  src      = 08:00:27:cb:98:07
  type     = 0x800
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 84
  id       = 25498
  flags    = DF
  frag     = 0
  ttl      = 64
  proto    = icmp
  chksum   = 0xc8fe
  src      = 10.0.2.15
  dst      = 1.1.1.1
  \options \
###[ ICMP ]###
  type     = echo-request
  code     = 0
  chksum   = 0x2b8e
  id       = 0x6645
  seq      = 0x1
###[ Raw ]###
  load     = '!~\x87^\xc8K\n\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12
\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#%&'()*+,-./01234567'
###[ Ethernet ]###
  dst      = 08:00:27:cb:98:07
  src      = 52:54:00:12:35:02
  type     = 0x800
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 84
  id       = 7380
  flags    =
  frag     = 0
  ttl      = 53
  proto    = icmp
  chksum   = 0x5ac5
  src      = 1.1.1.1
  dst      = 10.0.2.15
  \options \

```

By running sniffer.py as a superuser and opening up another terminal and pinging 1.1.1.1, we can see the packets in the network sending and receiving the ICMP echo request and reply.

Task 1.1B. Usually, when we sniff packets, we are only interested certain types of packets. We can do that by setting filters in sniffing. Scapy's 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):

- Capture only the ICMP packet

Since the code was already filtering for icmp packets, I modified it to capture only icmp packets that are not of type icmp-echo and not of type icmp-echoreply:

```
#!/usr/bin/python
from scapy.all import*
def print_pkt(pkt): pkt.show()
pkt = sniff(filter='icmp[icmptype] != icmp-echo and icmp[icmptype] != icmp-echoreply', prn = print_pkt)
```

Using traceroute command on a separate terminal allows us to capture an icmp packet that is of type time-exceeded with code time to live zero during transit:

```
[04/03/20]seed@VM:~/.../packetsniffinglab$ sudo python sniffer.py
###[ Ethernet ]###
  dst      = 08:00:27:cb:98:07
  src      = 52:54:00:12:35:02
  type     = 0x800
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0xc0
  len      = 56
  id       = 37342
  flags    =
  frag     = 0
  ttl      = 255
  proto    = icmp
  checksum = 0x1116
  src      = 10.0.2.2
  dst      = 10.0.2.15
  \options \
###[ ICMP ]###
  type     = time-exceeded
  code     = ttl-zero-during-transit
  checksum = 0xf84e
  reserved = 0
  length   = 0
  unused   = None
###[ IP in ICMP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 60
  id       = 2394
  flags    =
  frag     = 0
  ttl      = 1
  proto    = udp
  checksum = 0xa247
  src      = 10.0.2.15
  dst      = 1.1.1.1
  \options \
###[ UDP in ICMP ]###
  sport    = 60745
```


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

Part of Berkeley Packet Filter Documentation:

src port *port*
True if the packet has a source port value of *port*.

port *port*
True if either the source or destination port of the packet is *port*.

dst portrange *port1-port2*
True if the packet is ip/tcp, ip/udp, ip6/tcp or ip6/udp and has a destination port value between *port1* and *port2*. *port1* and *port2* are interpreted in the same fashion as the *port* parameter for **port**.

src portrange *port1-port2*
True if the packet has a source port value between *port1* and *port2*.

portrange *port1-port2*
True if either the source or destination port of the packet is between *port1* and *port2*.
Any of the above port or port range expressions can be prepended with the keywords, **tcp** or **udp**, as in:

```
tcp src port port
```

Modified sniff.py to filter for tcp packets with dst port number 23

```
#!/usr/bin/python
from scapy.all import*
def print_pkt(pkt): pkt.show()
pkt = sniff(filter='tcp dst port 23', prn = print_pkt)
```

Now, when we try to telnet to www1.chapman.edu (192.77.116.69), we can capture the tcp packets:

```
[04/03/20]seed@VM:~/.../packetsniffinglab$ sudo python sniffer.py
###[ Ethernet ]###
dst      = 52:54:00:12:35:02
src      = 08:00:27:cb:98:07
type     = 0x800
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x10
len      = 60
id       = 13100
flags    = DF
frag     = 0
ttl      = 64
proto    = tcp
chksum   = 0xc6de
src      = 10.0.2.15
dst      = 192.77.116.69
\options \
###[ TCP ]###
sport    = 34472
dport    = telnet
seq      = 4268926727L
ack      = 0
dataofs  = 10
reserved = 0
flags    = S
window   = 29200
chksum   = 0x40d0
urgptr   = 0
options  = [('MSS', 1460), ('SackOK', ''), ('Timestamp', (234539, 0)), ('NOP', None), ('WScale', 7)]

###[ Ethernet ]###
dst      = 52:54:00:12:35:02
src      = 08:00:27:cb:98:07
type     = 0x800
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x10
len      = 60
id       = 13101
flags    = DF
frag     = 0
```

- 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.

Changing the filter to: 'net 192.77.116.0 mask 255.255.255.0' search for all src or dst packets matching ip addresses in the range of 192.77.116.0 - 192.77.116.255. Then we can make an http request on the browser for www1.chapman.edu and sniff packets.

```
#!/usr/bin/python
from scapy.all import*
def print_pkt(pkt): pkt.show()
pkt = sniff(filter='net 192.77.116.0 mask 255.255.255.0', prn = print_pkt)
```

```

[04/03/20]seed@VM:~/../packetstniffinglab$ sudo python sniffer.py
###[ Ethernet ]###
dst      = 52:54:00:12:35:02
src      = 08:00:27:cb:98:07
type     = 0x800
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 60
id       = 29742
flags    = DF
frag     = 0
ttl      = 64
proto    = tcp
chksum   = 0x85ec
src      = 10.0.2.15
dst      = 192.77.116.69
\options \
###[ TCP ]###
sport    = 53842
dport    = http
seq      = 2522158520L
ack      = 0
dataofs  = 10
reserved = 0
flags    = S
window   = 29200
chksum   = 0x40d0
urgptr   = 0
options  = [('MSS', 1460), ('SAckOK', ''), ('Timestamp', (549270, 0)), ('NOP', None), ('WScale', 7)]

###[ Ethernet ]###
dst      = 08:00:27:cb:98:07
src      = 52:54:00:12:35:02
type     = 0x800
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 44
id       = 37423
flags    =
frag     = 0
ttl      = 64
proto    = tcp

```

2.2 Task 1.2: 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. The following code shows an example of how to spoof an ICMP packets.

```

>>> from scapy.all import *
>>> a = IP()                                À
>>> a.dst = '10.0.2.3'                      Á
>>> b = ICMP()                              Â
>>> p = a/b                                 Ã
>>> send(p)                                 Ä
. Sent 1 packets.

```

In the code above, Line À creates an IP object from the IP class; a class attribute is defined for each IP header field. We can use `ls(a)` or `ls(IP)` to see all the attribute names/values. We can also use `a.show()` and `IP.show()` to do the same. Line Á shows how to set the destination IP address field. If a field is not set, a default value will be used.


```

>>> ls(a)
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            = []             ([])

```

Line A creates an ICMP object. The default type is echo request. In Line B, we stack a and b together to form a new object. The / operator is overloaded by the IP class, so it no longer represents division; instead, it means adding b as the payload field of a and modifying the fields of a accordingly. As a result, we get a new object that represent an ICMP packet. We can now send out this packet using send() in Line C. Please make any necessary change to the sample code, and then demonstrate that you can spoof an ICMP echo request packet with an arbitrary source IP address.

Added P.src = '192.77.116.69' to masquerade as www1.chapman.edu. Adding print statements confirms the modified packet p is of type icmp echo-request and src ip address has successfully been spoofed:

```

#!/usr/bin/python
from scapy.all import*
a = IP()
a.dst = '10.0.2.3'
b = ICMP()
p = a/b
p.src = '192.77.116.69'
print("a: ")
a.show()
print("p: ")
p.show()

send(p)

```

```

[04/03/20]seed@VM:~/.../packetsniffinglab$ sudo python spoofer.py
a:
###[ IP ]###
  version = 4
  ihl     = None
  tos     = 0x0
  len     = None
  id      = 1
  flags   =
  frag    = 0
  ttl     = 64
  proto   = hopopt
  chksum  = None
  src     = 10.0.2.15
  dst     = 10.0.2.3
  \options \

p:
###[ IP ]###
  version = 4
  ihl     = None
  tos     = 0x0
  len     = None
  id      = 1
  flags   =
  frag    = 0
  ttl     = 64
  proto   = icmp
  chksum  = None
  src     = 192.77.116.69
  dst     = 10.0.2.3
  \options \
###[ ICMP ]###
  type    = echo-request
  code    = 0
  chksum  = None
  id      = 0x0
  seq     = 0x0
.
Sent 1 packets.
[04/03/20]seed@VM:~/.../packetsniffinglab$

```

2.3 Task 1.3: Traceroute

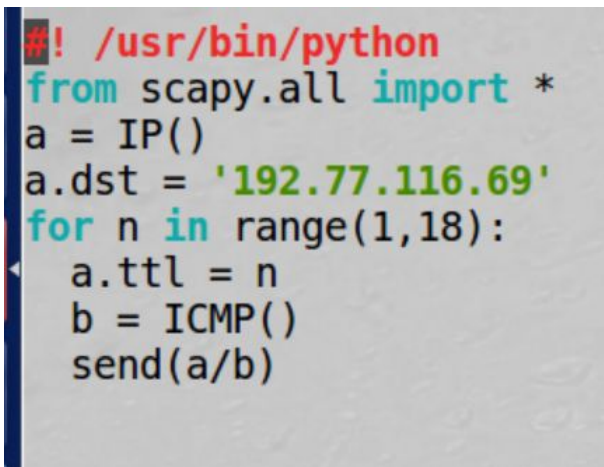
The objective of this task is to use Scapy to estimate the distance, in terms of number of routers, between your VM and a selected destination. This is basically what is implemented by the traceroute tool. In this task, we will write our own tool. The idea is quite straightforward: just send a packet (any type) to the destination, with its Time-To-Live (TTL) field set to 1 first. This packet will be dropped by the first router, which will send us an ICMP error message, telling us that the time-to-live has exceeded. That is how we get the IP address of the first router. We then increase our TTL field to 2, send out another packet, and get the IP address of the second router. We will repeat this procedure until our packet finally reach the destination. It should be noted that this

experiment only gets an estimated result, because in theory, not all these packets take the same route (but in practice, they may within a short period of time). The code in the following shows one round in the procedure.

```
a = IP()
a.dst = '1.2.3.4'
a.ttl = 3 b =
ICMP()
send(a/b)
```

If you are an experienced Python programmer, you can write your tool to perform the entire procedure automatically. If you are new to Python programming, you can do it by manually changing the TTL field in each round, and record the IP address based on your observation from Wireshark. Either way is acceptable, as long as you get the result.

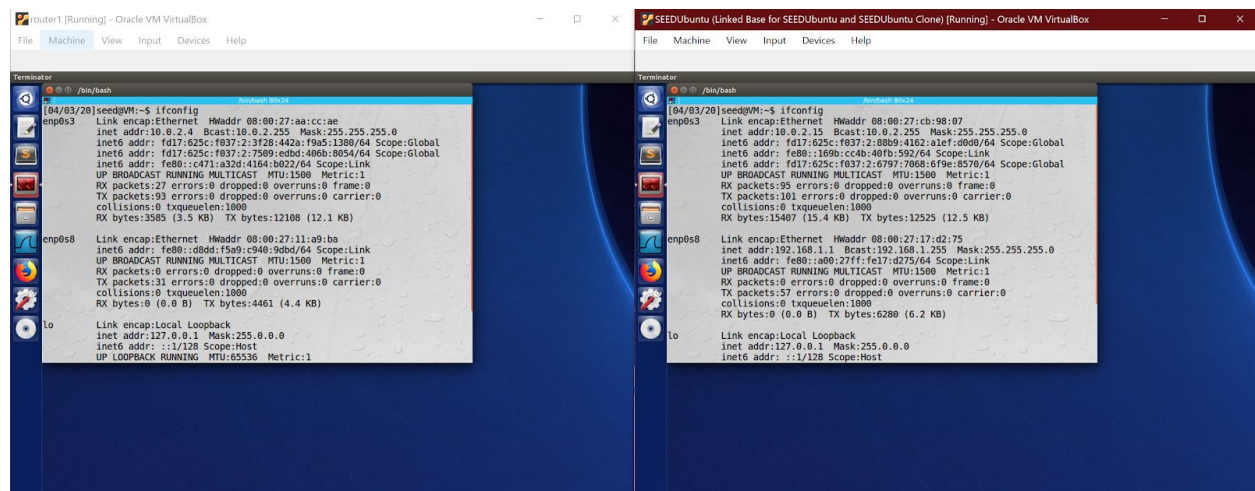
Using the program below, and wireshark to capture icmp packets, it seems that the first icmp echo-reply packet is captured when the ttl is set to 17. So it is estimated that it takes 17 hops to reach www1.chapman.edu (192.77.116.69):



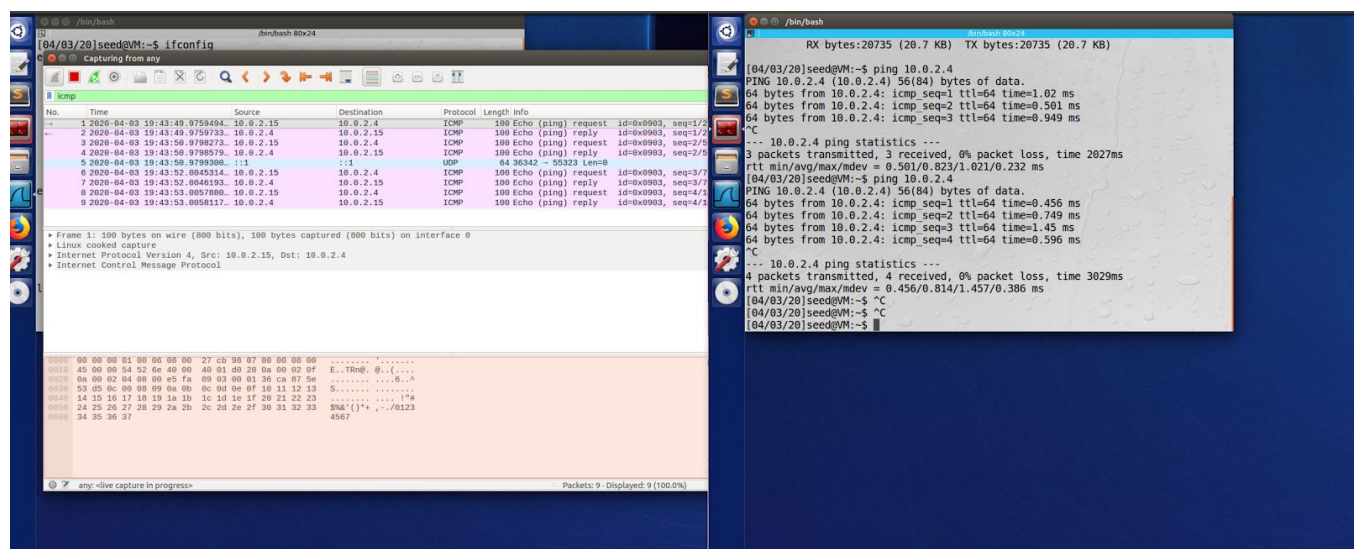
```
#!/usr/bin/python
from scapy.all import *
a = IP()
a.dst = '192.77.116.69'
for n in range(1,18):
    a.ttl = n
    b = ICMP()
    send(a/b)
```


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, regardless 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.

Using the preset NAT network, two VMs are running. One has been assigned the ip 10.0.2.4 and the other 10.0.2.15:



Using wireshark on one vm and ping on the other, we can see that the vms can communicate with one another:



I couldn't get the packet sniffer to capture the icmp packets from the other vm, so I had to ssh into the other vm in order to "promiscuously" run a packet sniffer after gaining root privilege to spy on the other vm and deploy the spoof program.


```
[04/04/20]seed@VM:~$ ssh -l jinjung 10.0.2.15
jinjung@10.0.2.15's password:
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

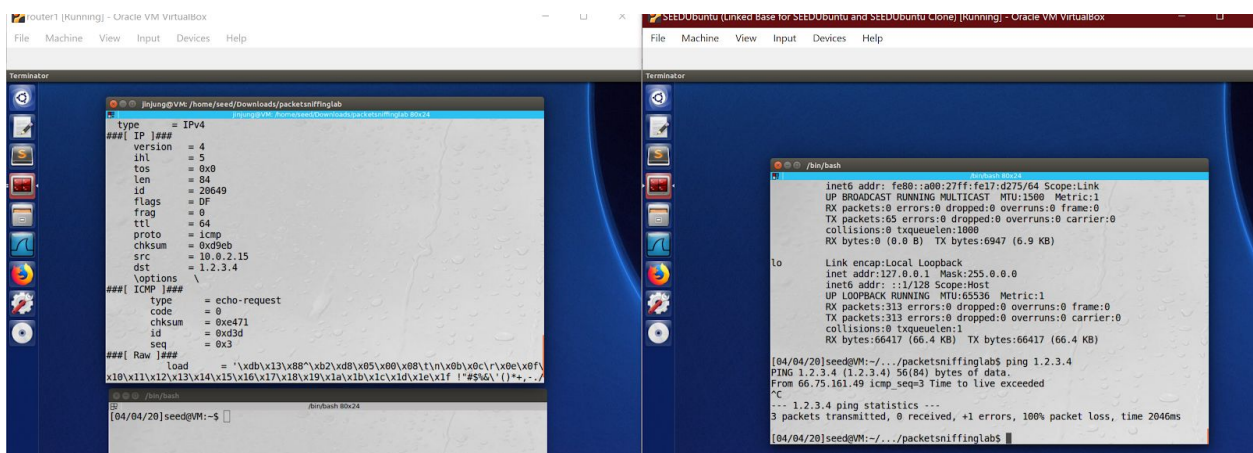
1 package can be updated.
0 updates are security updates.

Last login: Sat Apr  4 00:49:39 2020 from 10.0.2.4
jinjung@VM:~$ ls
examples.desktop
jinjung@VM:~$ cd /
jinjung@VM:/$ cd home/
jinjung@VM:/home$ dir
jinjung seed
jinjung@VM:/home$ cd seed/
jinjung@VM:/home/seed$ cd Downloads/packetsniffinglab/
jinjung@VM:/home/seed/Downloads/packetsniffinglab$
```

Modified the sniffer.py to filter for icmp echo-request packets with dst host 1.2.3.4:

```
#!/usr/bin/python
import sys
from scapy.all import *

sys.path.append('/usr/local/bin/scapy')
def print_pkt(pkt): pkt.show()
pkt = sniff(filter='dst host 1.2.3.4', prn = print_pkt)
```



Modified code to spoof an icmp echo-reply for each captured echo-request packet. Attempted to modify all the IP and ICMP header information. The spoofed reply packet seems very close, but not quite there:

```
#!/usr/bin/python3
import sys
from scapy.all import *

#sys.path.append('/usr/local/bin/scapy')
def print_pkt(pkt):
    pkt.show()
    a = IP()
    a.dst = pkt[IP].src
    a.src = pkt[IP].dst
    a.seq = pkt[IP].seq
    a.id = pkt[IP].id
    a.chksum = pkt[IP].chksum
    a.version = pkt[IP].version
    a.ihl = pkt[IP].ihl
    a.len = pkt[IP].len
    a.flags = pkt[IP].flags
    b = ICMP(type = 'echo-reply')
    b.chksum = pkt[ICMP].chksum
    b.id = pkt[ICMP].id
    b.seq = pkt[ICMP].seq
    b.code = pkt[ICMP].code
    p = a/b
    p.chksum = pkt[ICMP].chksum

    p.show()
    send(p)
pkt = sniff(filter='dst host 1.2.3.4', prn = print_pkt)
```

Captured ICMP echo-request:

```
jinjung@VM:/home/seed/Downloads/packetsniffinglab$ sudo ./sniffer.py
###[ Ethernet ]###
  dst      = 52:54:00:12:35:00
  src      = 08:00:27:cb:98:07
  type     = IPv4
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 84
  id       = 45644
  flags    = DF
  frag     = 0
  ttl      = 64
  proto    = icmp
  chksum   = 0x7848
  src      = 10.0.2.15
  dst      = 1.2.3.4
  \options \
###[ ICMP ]###
  type     = echo-request
  code     = 0
  chksum   = 0x820a
  id       = 0x10e4
  seq      = 0x1
###[ Raw ]###
  load     = '\xac\x88^\x8b\x08\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#%&\'()*+,-./01234567'
```

This is the spoofed icmp echo-reply:

```
###[ IP ]###
  version  = 4
  ihl      = 5
  tos      = 0x0
  len      = 84
  id       = 45647
  flags    = DF
  frag     = 0
  ttl      = 64
  proto    = icmp
  chksum   = 0x3e04
  src      = 1.2.3.4
  dst      = 10.0.2.15
  \options \
###[ ICMP ]###
  type     = echo-reply
  code     = 0
  chksum   = 0x3e04
  id       = 0x10e4
  seq      = 0x2
.
Sent 1 packets.
###[ Ethernet ]###
```

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

3.1 Task 2.1: Writing Packet Sniffing Program

Sniffer programs can be easily written using the pcap library. With pcap, the task of sniffers becomes invoking a simple sequence of procedures in the pcap library. At the end of the sequence, packets will be put in buffer for further processing as soon as they are captured. All the details of packet capturing are handled by the pcap library.

```
#include <pcap.h>
#include <stdio.h>

/* This function will be invoked by pcap for each captured packet.
   We can process each packet inside the function.
*/ void got_packet(u_char *args, const struct pcap_pkthdr *header, const
u_char *packet)
{ printf("Got a packet\n");
}

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 pcap session on NIC with name eth3
  //                               Students needs to change "eth3" to the name
  //                               found on their own machines (using ifconfig).
  handle = pcap_open_live("eth3", BUFSIZ, 1, 1000, errbuf);

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

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

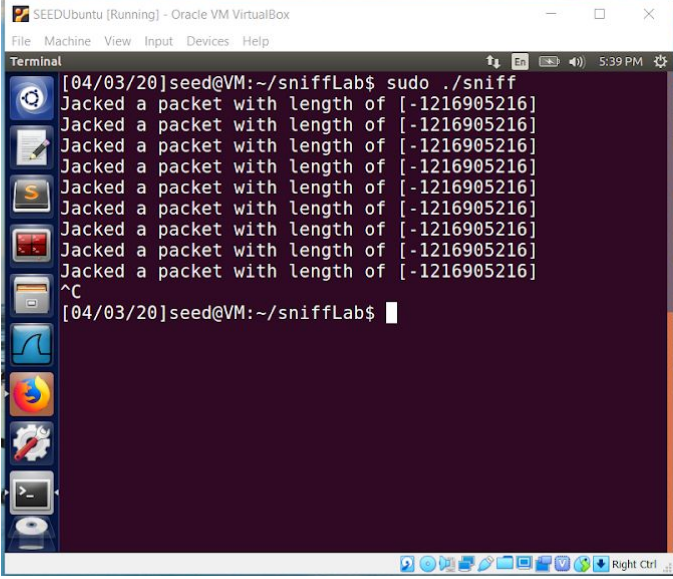
  pcap_close(handle);          //Close the handle
  return 0; }

// Note: don't forget to add "-lpcap" to the compilation command.
// For example: gcc -o sniff sniff.c -lpcap
```

Tim Carstens has also written a tutorial on how to use pcap library to write a sniffer program. The tutorial is available at <http://www.tcpdump.org/pcap.htm>.

Task 2.1A: Understanding How a Sniffer Works In this task, students need to write a sniffer program to print out the source and destination IP addresses of each captured packet. Students should provide screenshots as evidences to show that their sniffer

program can run successfully and produces expected results. In addition, please answer the following questions:



The screenshot shows a terminal window titled "SEEDUbuntu [Running] - Oracle VM VirtualBox". The terminal output is as follows:

```
[04/03/20]seed@VM:~/sniffLab$ sudo ./sniff
Jacked a packet with length of [-1216905216]
Jacked a packet with length of [-1216905216]
Jacked a packet with length of [-1216905216]
Jacked a packet with length of [-1216905216]
Jacked a packet with length of [-1216905216]
Jacked a packet with length of [-1216905216]
Jacked a packet with length of [-1216905216]
Jacked a packet with length of [-1216905216]
^C
[04/03/20]seed@VM:~/sniffLab$
```

I created a while(1) loop with its body being

"packet = pcap_next(handle, &header);

printf("Jacked a packet with length of [%d]\n", header.len);"

This was the output with promiscuous mode on.

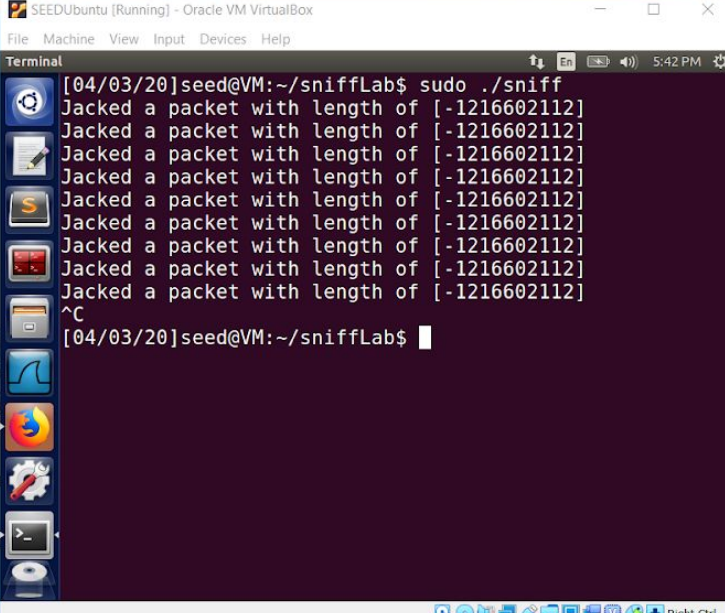
Question 1. Please use your own words to describe the sequence of the library calls that are essential for sniffer programs. This is meant to be a summary, not detailed explanation like the one in the tutorial or book.

First, we must open up a live interface that the system will use. This is done by pcap_open_live(). Then, we must set up filtering rules so that the user sniffs what they are looking for. Then we execute the sniff by using capturing packets. Finally, we close the session.

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

You need root privilege because the sniffer program needs to access a network device. A user without root privilege cannot do this. pcap_lookupdev() fails because the function cannot look up a device without root privileges.

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



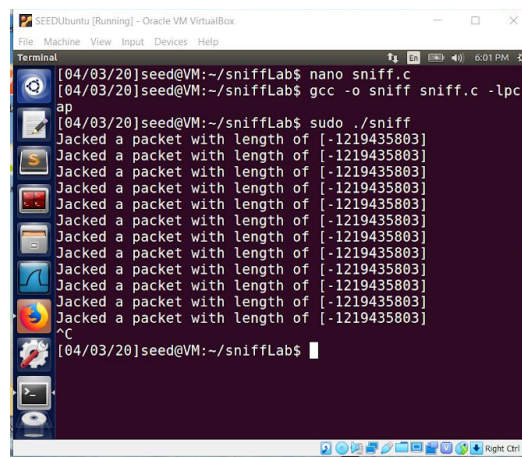
```
SEEDUbuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
[04/03/20]seed@VM:~/sniffLab$ sudo ./sniff
Jacked a packet with length of [-1216602112]
Jacked a packet with length of [-1216602112]
Jacked a packet with length of [-1216602112]
Jacked a packet with length of [-1216602112]
Jacked a packet with length of [-1216602112]
Jacked a packet with length of [-1216602112]
Jacked a packet with length of [-1216602112]
Jacked a packet with length of [-1216602112]
Jacked a packet with length of [-1216602112]
^C
[04/03/20]seed@VM:~/sniffLab$
```

When promiscuous mode is off in the sniffer program, then all LAN devices listen to traffic. This will cause the sniffer program to capture more packets than intended to.

Task 2.1B: Writing Filters. Please write filter expressions for your sniffer program to capture each of the followings. You can find online manuals for pcap filters. In your lab reports, you need to include screenshots to show the results after applying each of these filters.

- Capture the ICMP packets between two specific hosts.

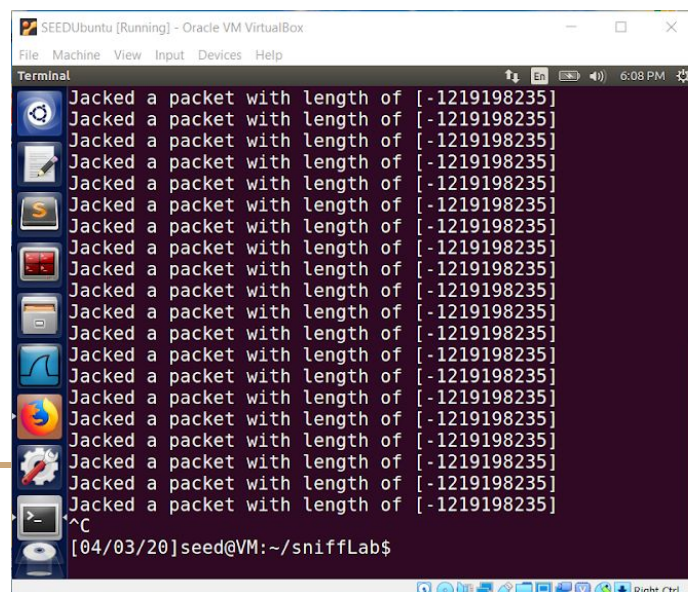
Capture ICMP packets by inserting "icmp and (src host 10.0.2.15 and dst host 8.8.8.8) or (src host 8.8.8.8 and dst host 10.0.2.15)" in "filter_exp[]"



```
SEEDUbuntu [Running] - Oracle VM VirtualBox
Terminal
[04/03/20]seed@VM:~/sniffLab$ nano sniff.c
[04/03/20]seed@VM:~/sniffLab$ gcc -o sniff sniff.c -lpc
ap
[04/03/20]seed@VM:~/sniffLab$ sudo ./sniff
Jacked a packet with length of [-1219435803]
Jacked a packet with length of [-1219435803]
Jacked a packet with length of [-1219435803]
Jacked a packet with length of [-1219435803]
Jacked a packet with length of [-1219435803]
Jacked a packet with length of [-1219435803]
Jacked a packet with length of [-1219435803]
Jacked a packet with length of [-1219435803]
Jacked a packet with length of [-1219435803]
Jacked a packet with length of [-1219435803]
Jacked a packet with length of [-1219435803]
^C
[04/03/20]seed@VM:~/sniffLab$
```

- Capture the TCP packets with a destination port number in the range from 10 to 100.

Capture TCP packets by inserting "tcp dst portrange 10-100" in "filter_exp[]"



```
SEEDUbuntu [Running] - Oracle VM VirtualBox
Terminal
Jacked a packet with length of [-1219198235]
Jacked a packet with length of [-1219198235]
Jacked a packet with length of [-1219198235]
Jacked a packet with length of [-1219198235]
Jacked a packet with length of [-1219198235]
Jacked a packet with length of [-1219198235]
Jacked a packet with length of [-1219198235]
Jacked a packet with length of [-1219198235]
Jacked a packet with length of [-1219198235]
Jacked a packet with length of [-1219198235]
Jacked a packet with length of [-1219198235]
Jacked a packet with length of [-1219198235]
Jacked a packet with length of [-1219198235]
Jacked a packet with length of [-1219198235]
Jacked a packet with length of [-1219198235]
Jacked a packet with length of [-1219198235]
Jacked a packet with length of [-1219198235]
Jacked a packet with length of [-1219198235]
Jacked a packet with length of [-1219198235]
^C
[04/03/20]seed@VM:~/sniffLab$
```

Task 2.1C: Sniffing Passwords. Please show how you can use your sniffer program to capture the password when somebody is using telnet on the network that you are monitoring. You may need to modify your sniffer code to print out the data part of a captured TCP packet (telnet uses TCP). It is acceptable if you print out the entire data part, and then manually mark where the password (or part of it) is.

To sniff for passwords, we must sniff on port 23(telnet). So in “filter_exp[]” we add the filter “tcp port 23”.

3.2 Task 2.2: Spoofing

When a normal user sends out a packet, operating systems usually do not allow the user to set all the fields in the protocol headers (such as TCP, UDP, and IP headers). Oses will set most of the fields, while only allowing users to set a few fields, such as the destination IP address, the destination port number, etc. However, if users have the root privilege, they can set any arbitrary field in the packet headers. This is called packet spoofing, and it can be done through *raw sockets*.

Raw sockets give programmers the absolute control over the packet construction, allowing programmers to construct any arbitrary packet, including setting the header fields and the payload. Using raw sockets is quite straightforward; it involves four steps: (1) create a raw socket, (2) set socket option, (3) construct the packet, and (4) send out the packet through the raw socket. There are many online tutorials that can teach you how to use raw sockets in C programming. We have linked some tutorials to the lab's web page. Please read them, and learn how to write a packet spoofing program. We show a simple skeleton of such a program.

```
int sd; struct sockaddr_in sin; char buffer[1024]; // You can
change the buffer size
```

```
/* Create a raw socket with IP protocol. The IPPROTO_RAW parameter
```

```

*    tells the system 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);
}

/* This data structure is needed when sending the packets
*    using sockets. Normally, we need to fill out several
*    fields, but for raw sockets, we only need to fill out
*    this one field */ sin.sin_family = AF_INET;

// Here you can construct the IP packet using buffer[] //      -
construct the IP header ...
//                      - construct the TCP/UDP/ICMP header ...
//                      - fill in the data part if needed ...
// Note: you should pay attention to the network/host byte order.

/* Send out the IP packet.
*    ip_len is the actual size of the packet. */ if(sendto(sd, buffer, ip_len, 0, (struct
sockaddr *)&sin, sizeof(sin)) < 0) {
    perror("sendto() error"); exit(-1);
}

```

Task 2.2A: Write a spoofing program. Please write your own packet spoofing program in C. You need to provide evidences (e.g., Wireshark packet trace) to show that your program successfully sends out spoofed IP packets.

Task 2.2B: Spoof an ICMP Echo Request. Spoof an ICMP echo request packet on behalf of another machine (i.e., using another machine's IP address as its source IP address). This packet should be sent to a remote machine on the Internet (the machine must be alive). You should turn on your Wireshark, so if your spoofing is successful, you can see the echo reply coming back from the remote machine.

Questions. Please answer the following questions.

- Question 4. Can you set the IP packet length field to an arbitrary value, regardless of how big the actual packet is?
- Question 5. Using the raw socket programming, do you have to calculate the checksum for the IP header?
- 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?

3.3 Task 2.3: Sniff and then Spoof (optional meant to do with lab partner)

In this task, you will combine the sniffing and spoofing techniques to implement the following sniff-andthen-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, regardless of whether machine X is alive or not, the ping program will always receive a reply, indicating that X is alive. You need to write such a program in C, and include screenshots in your report to show that your program works. Please also attach the code (with adequate amount of comments) in your report.

4 Guidelines

4.1 Filling in Data in Raw Packets

When you send out a packet using raw sockets, you basically construct the packet inside a buffer, so when you need to send it out, you simply give the operating system the buffer and the size of the packet. Working directly on the buffer is not easy, so a common way is to typecast the buffer (or part of the buffer) into structures, such as IP header structure, so you can refer to the elements of the buffer using the fields of those structures. You can define the IP, ICMP, TCP, UDP and other header structures in your program. The following example show how you can construct an UDP packet:


```

struct ipheader {
type field; ..... }

    struct udpheader {
        type field; .....
    }

// This buffer will be used to construct raw packet. char buffer[1024];

// Typecasting the buffer to the IP header structure struct ipheader *ip
= (struct ipheader *) buffer;

// Typecasting the buffer to the UDP header structure struct
udpheader *udp = (struct udpheader *) (buffer + sizeof(struct
ipheader));

// Assign value to the IP and UDP header fields.
ip->field = ...; udp->field = ...;

```

4.2 Network/Host Byte Order and the Conversions

You need to pay attention to the network and host byte orders. If you use x86 CPU, your host byte order uses *Little Endian*, while the network byte order uses *Big Endian*.

Whatever the data you put into the packet buffer has to use the network byte order; if you do not do that, your packet will not be correct. You actually do not need to worry about what kind of Endian your machine is using, and you actually should not worry about if you want your program to be portable.

What you need to do is to always remember to convert your data to the network byte order when you place the data into the buffer, and convert them to the host byte order when you copy the data from the buffer to a data structure on your computer. If the data is a single byte, you do not need to worry about the order, but if the data is a short, int, long, or a data type that consists of more than one byte, you need to call one of the following functions to convert the data:

```

htonl(): convert unsigned int from host to network byte order. ntohl(): reverse of
htonl().
htons(): convert unsigned short int from host to network byte order. ntohs(): reverse
of htons().

```

You may also need to use `inetaddr()`, `inetnetwork()`, `inetntoa()`, `inetaton()` to convert IP addresses from the dotted decimal form (a string) to a 32-bit integer of network/host byte order. You can get their manuals from the Internet.

5 Submission

Students need to submit a detailed lab report to describe what they have done, what they have observed, and how they interpret the results. Reports should include evidences to support the observations. Evidences include packet traces, screenshots, etc. Reports should also list the important code snippets with explanations. Simply attaching code without any explanation will not receive credits.