README.md

Packet Sniffing and Spoofing

Intro

My name is Keith Sabine and I will be working through this SEED security lab as part of my Undergrad CS studies.

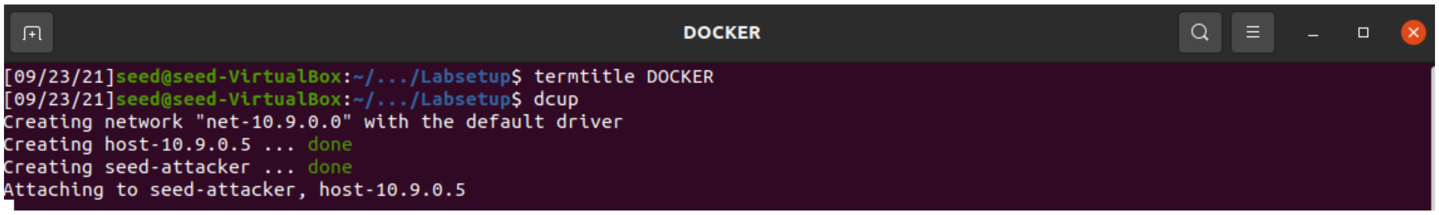
Some notes

All of the machine configurations are available on the SEED website.(https://seedsecuritylabs.org/Labs_20.04/Networking/Sniffing_Spoofing/). I will only be posting my progress and observations as I work through the lab and answer the questions it requires. I set up the SEED VM using the Ubuntu 20.04 VM config. I will be completing both Task 1 and Task 2.

Environment Setup using Container

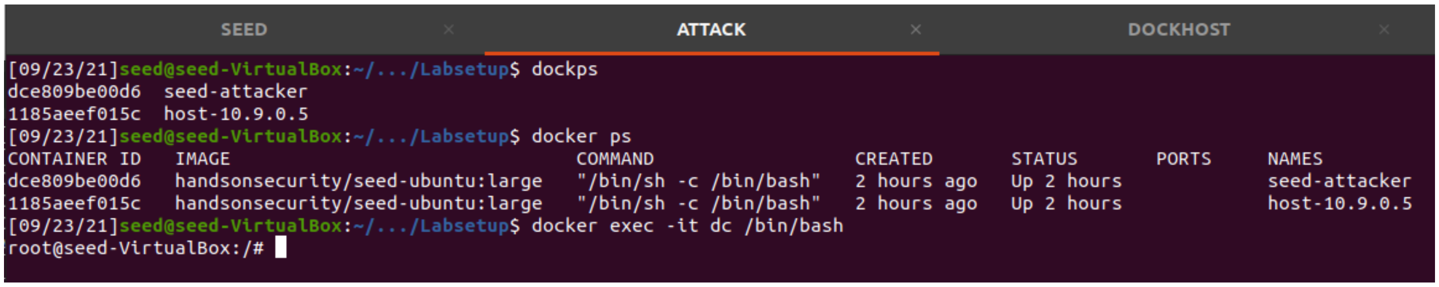
You can download the Labsetup.zip from the Lab website which contains scripts to set up Docker Images. I have never worked with Docker before so this step has been new for me. The lab gives us some commands to build and the SEED VM setup gave us a nice bash config to make these commands easy.

Once you compose the docker images you can use the alias command `dcup` to boot up our Docker network



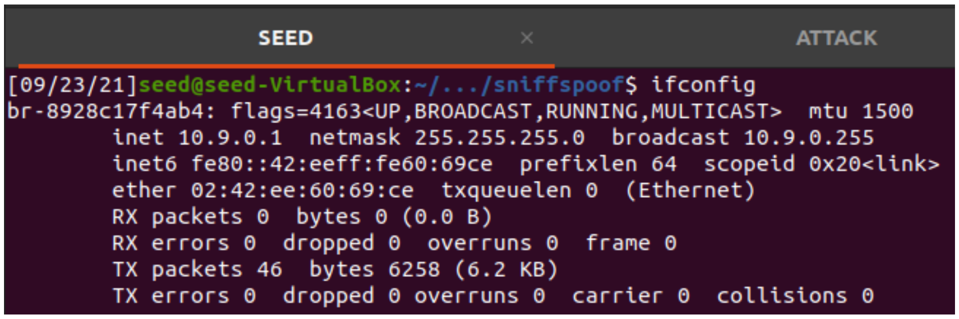
```
[09/23/21]seed@seed-VirtualBox:~/.../Labsetup$ termtitle DOCKER
[09/23/21]seed@seed-VirtualBox:~/.../Labsetup$ dcup
Creating network "net-10.9.0.0" with the default driver
Creating host-10.9.0.5 ... done
Creating seed-attacker ... done
Attaching to seed-attacker, host-10.9.0.5
```

Next I want to make sure I am able to have a shell open in both the attacker and the host machine for quick testing and reference so I use more of the commands in docker in seperate terminal windows for easy access between the VM and the two docker machines. Use `dockps` to find the ID and the Name of each docker image and then activate a shell in each I named my terminal windows ATTACK and DOCKHOST for the shells I am using in each `docker exec -it dc /bin/bash/` opens the shell @ root level.



```
[09/23/21]seed@seed-VirtualBox:~/.../Labsetup$ dockps
dce809be00d6  seed-attacker
1185aeef015c  host-10.9.0.5
[09/23/21]seed@seed-VirtualBox:~/.../Labsetup$ docker ps
CONTAINER ID   IMAGE                               COMMAND                  CREATED         STATUS         PORTS     NAMES
dce809be00d6   handsonsecurity/seed-ubuntu:large   "/bin/sh -c /bin/bash"   2 hours ago     Up 2 hours               seed-attacker
1185aeef015c   handsonsecurity/seed-ubuntu:large   "/bin/sh -c /bin/bash"   2 hours ago     Up 2 hours               host-10.9.0.5
[09/23/21]seed@seed-VirtualBox:~/.../Labsetup$ docker exec -it dc /bin/bash
root@seed-VirtualBox:/# 
```

The final part of setting up requires getting the network interface that our Docker images will be communicating over. The docker-compose.yml script shows that we have assigned the IP 10.9.0.0/24 to the network so we will need to identify the network device on our VM that matches the script setup. So lets check `ifconfig`



```
[09/23/21]seed@seed-VirtualBox:~/.../sniffspoof$ ifconfig
br-8928c17f4ab4: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.1  netmask 255.255.255.0  broadcast 10.9.0.255
        inet6 fe80::42:eeff:fe60:69ce  prefixlen 64  scopeid 0x20<link>
        ether 02:42:ee:60:69:ce  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 46  bytes 6258 (6.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

We see the device name on my machine is br-8928c17f4ab4 this is important so I copy it into a textfile for use later.

The Docker is set up and I have the terminals open! Things are looking good so far, but there is a lot more to learn about docker and the environment we have set up here beyond the basic instructions provided by the lab. Seed has posted a comprehensive tutorial for using Docker with their labs and I will keep the link for reference Docker Tutorial

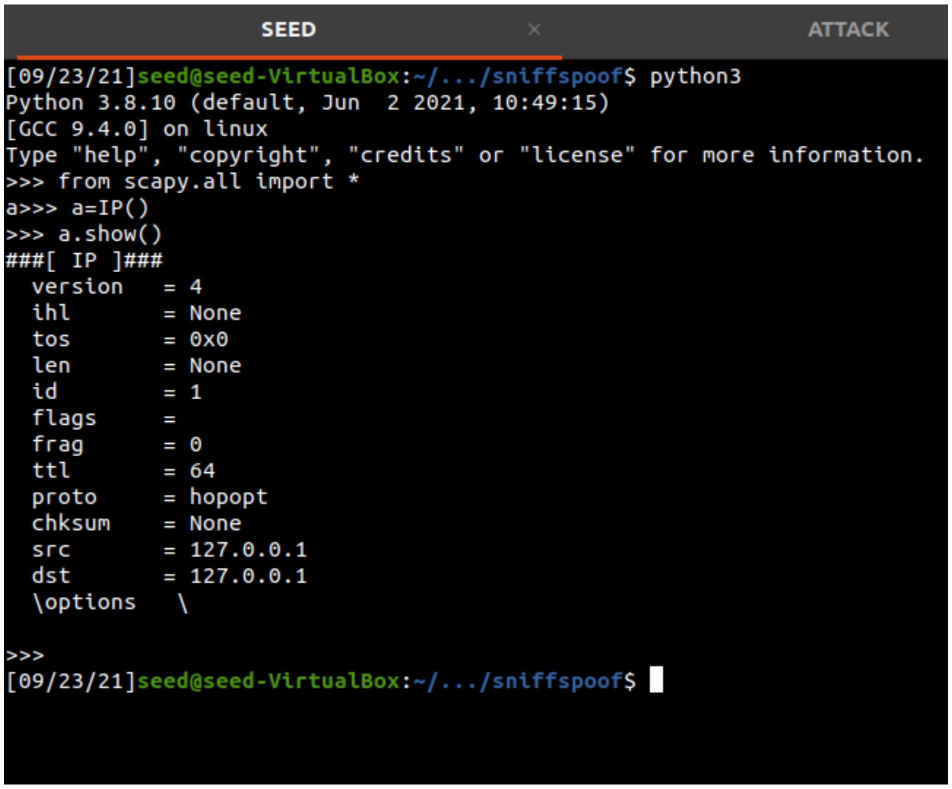
Using Scapy to Sniff and Spoof Packets

Scapy is a Python program enabling user to send, sniff, dissect, and forge network packets. Time to learn a bit about it for the upcoming tasks!

running the example script:

```
from scapy.all import * #We import all modules from Scapy to use
a = IP()
a.show()
```

creates an IP packet for us



```
[09/23/21]seed@seed-VirtualBox:~/.../sniffspoof$ python3
Python 3.8.10 (default, Jun  2 2021, 10:49:15)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from scapy.all import *
a>>> a=IP()
>>> a.show()
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = hopopt
  chksum    = None
  src       = 127.0.0.1
  dst       = 127.0.0.1
  \options   \

>>>
[09/23/21]seed@seed-VirtualBox:~/.../sniffspoof$ 
```

Lets use Scapy to do some packet sniffing for us.

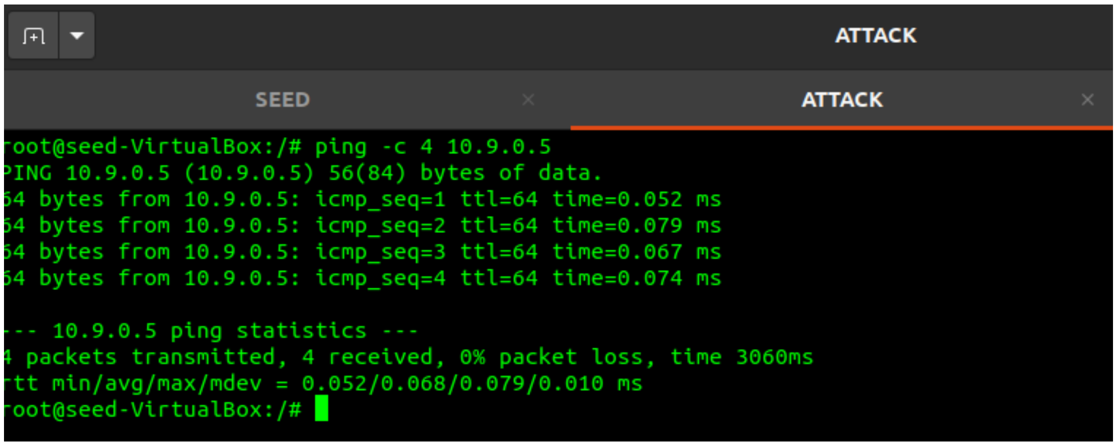
Task 1.1 Sniffing Packets

We are going to take the sample code and run it on our vm, you can find the script in the repo or simply copy/paste it, but be careful as your interface name may be different. notice how sniff requires an interface name, iface, to be specified. iface can be one interface or multiple (use a list).

```
#!/usr/bin/env python3

def print_pkt(pkt):
        pkt.show()

pkt = sniff(iface='br-8928c17f4ab4' , filter='icmp', prn=print_pkt)
```

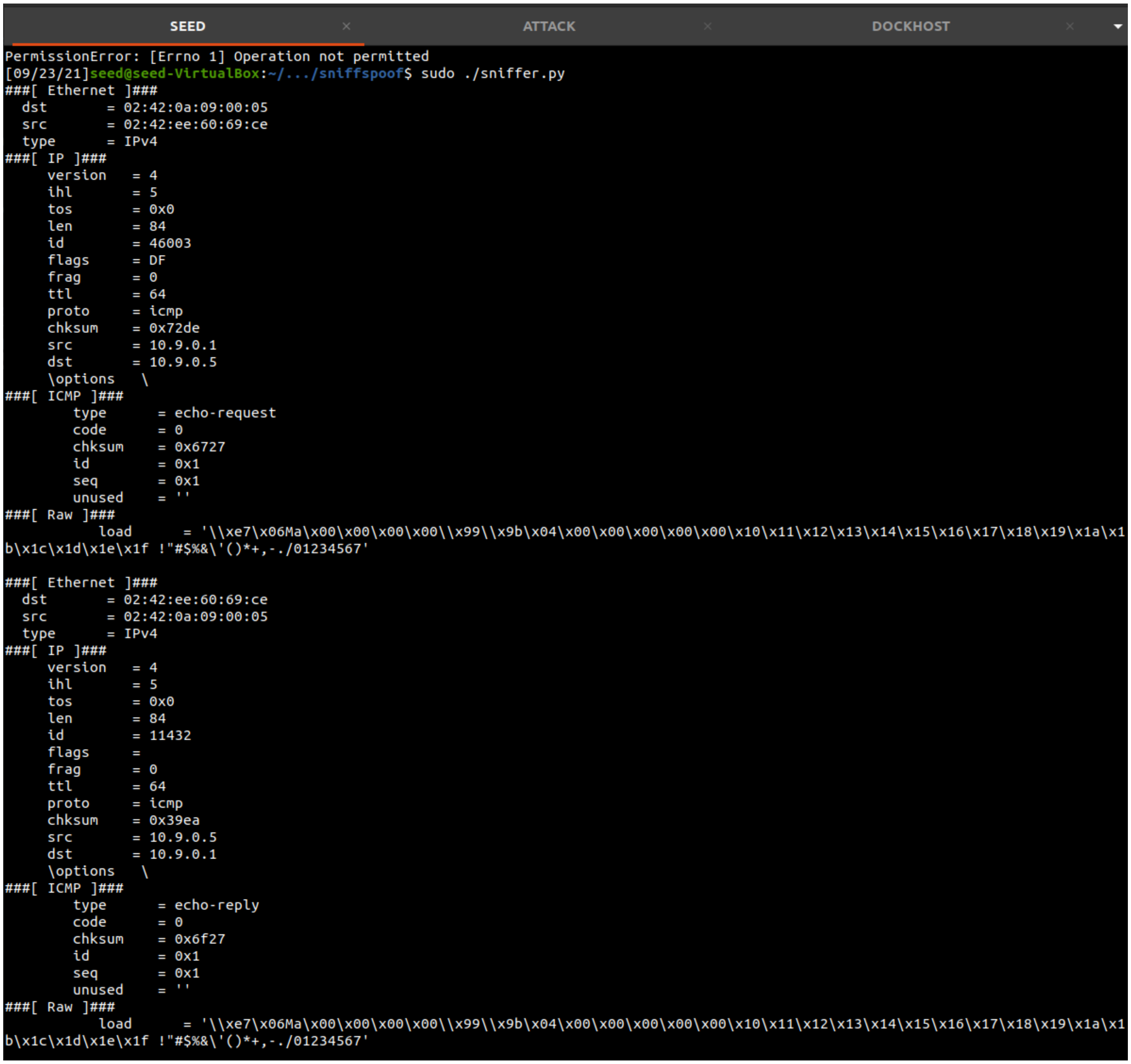
Task 1.1A

We will sniff from the VM to start. The filter is set for ICMP packets so lets send a ping to the container network interface. Don't forget that root privilege will be needed to view packets. Sending a ping to the br-8928c17f4ab4 interface from the ATTACK container we can sniff the ICMP activity from our VM.



```
root@seed-VirtualBox:/# ping -c 4 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=64 time=0.052 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.079 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=64 time=0.067 ms
64 bytes from 10.9.0.5: icmp_seq=4 ttl=64 time=0.074 ms

--- 10.9.0.5 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3060ms
rtt min/avg/max/mdev = 0.052/0.068/0.079/0.010 ms
root@seed-VirtualBox:/# 
```

Ping sent!



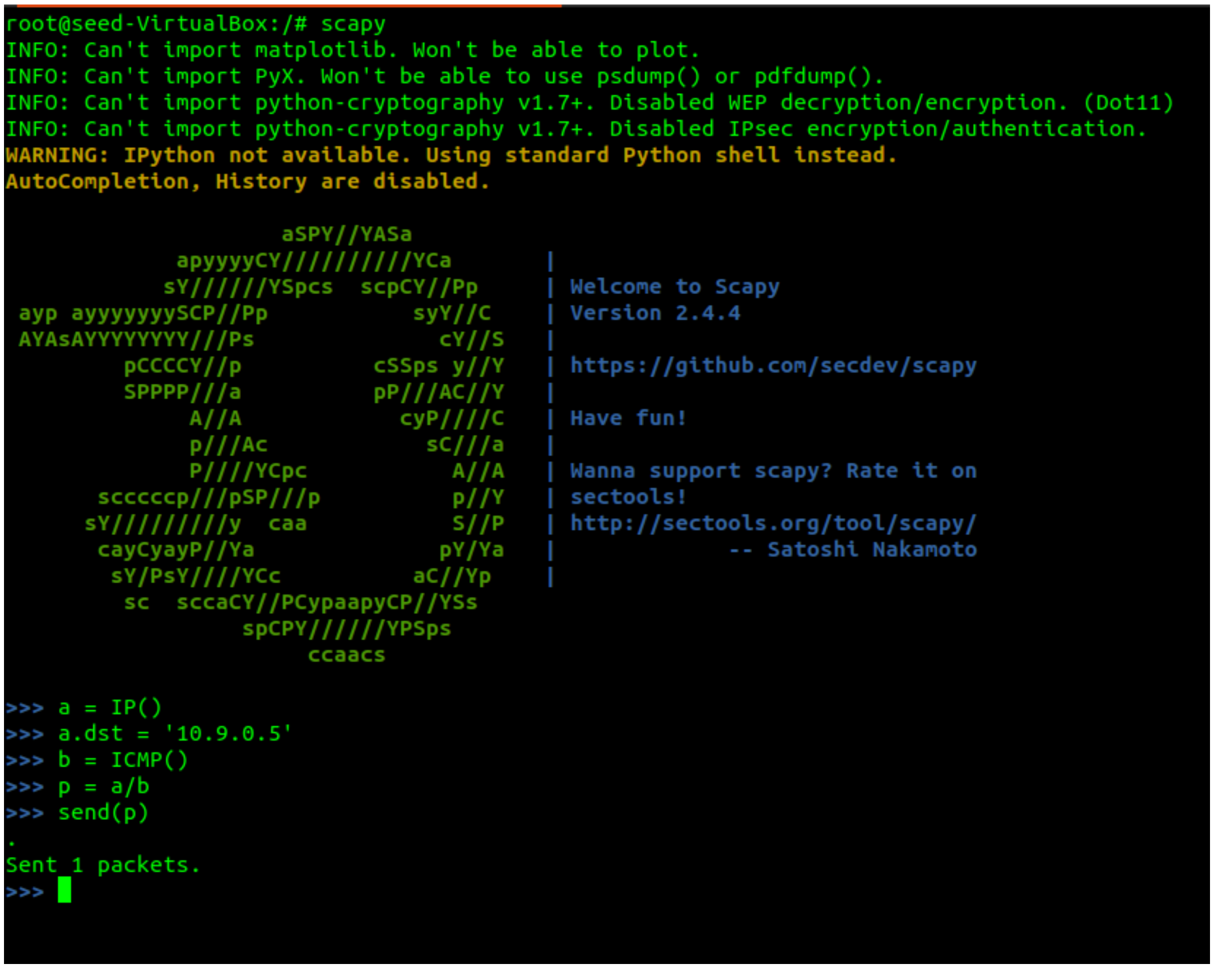
```
PermissionError: [Errno 1] Operation not permitted
[09/23/21]seed@seed-VirtualBox:~/.../sniffspoof$ sudo ./sniffer.py
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:ee:60:69:ce
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 46003
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x72de
     src       = 10.9.0.1
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x6727
        id        = 0x1
        seq       = 0x1
        unused    = ''
###[ Raw ]###
           load      = '\\xe7\x06Ma\x00\x00\x00\x00\\x99\\x9b\x04\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1
b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'

###[ Ethernet ]###
  dst       = 02:42:ee:60:69:ce
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 11432
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x39ea
     src       = 10.9.0.5
     dst       = 10.9.0.1
     \options   \
###[ ICMP ]###
        type      = echo-reply
        code      = 0
        chksum    = 0x6f27
        id        = 0x1
        seq       = 0x1
        unused    = ''
###[ Raw ]###
           load      = '\\xe7\x06Ma\x00\x00\x00\x00\\x99\\x9b\x04\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1
b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'
```

Running the same program without root privilege results in an Operation not permitted value. Why? Direct access to network devices is restricted to root users. If you didnt need root privilege then every user on a system could potentially control network adapters which would be a bad idea.

Task 1.1B

After messing around with Scapy for awhile trying to get its built in sprintf function to work I caved and created a print function to neatly display the ICMP traffic like [SOURCE] -> [DEST]. Now we can test out the filter functionality in scapy.

First we create a packet on the attacker container using scapy

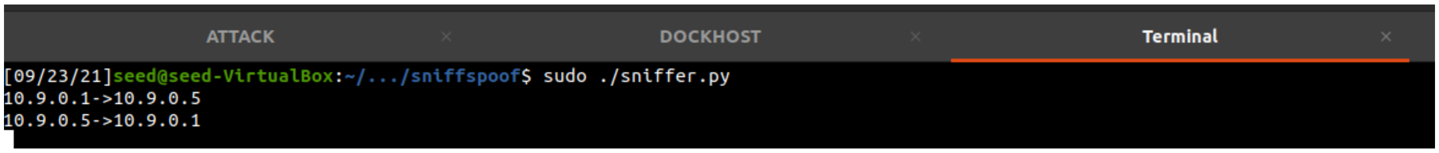


```
root@seed-VirtualBox:/# scapy
INFO: Can't import matplotlib. Won't be able to plot.
INFO: Can't import PyX. Won't be able to use psdump() or pdfdump().
INFO: Can't import python-cryptography v1.7+. Disabled WEP decryption/encryption. (Dot11)
INFO: Can't import python-cryptography v1.7+. Disabled IPsec encryption/authentication.
WARNING: IPython not available. Using standard Python shell instead.
AutoCompletion, History are disabled.

                     aSPY//YASa
             apyyyyCY//////////YCa       |
            sY//////YSpcs  scpCY//Pp     | Welcome to Scapy
 ayp ayyyyyyySCP//Pp           syY//C    | Version 2.4.4
 AYAsAYYYYYYYY///Ps              cY//S   |
         pCCCCY//p          cSSps y//Y   | https://github.com/secdev/scapy
         SPPPP///a          pP///AC//Y   |
              A//A            cyP////C   | Have fun!
              p///Ac            sC///a   |
              P////YCpc           A//A   | Wanna support scapy? Rate it on
       scccccp///pSP///p          p//Y   | sectools!
      sY/////////y  caa           S//P   | http://sectools.org/tool/scapy/
       cayCyayP//Ya              pY/Ya   |              -- Satoshi Nakamoto
        sY/PsY////YCc          aC//Yp    |
         sc  sccaCY//PCypaapyCP//YSs
                  spCPY//////YPSps
                       ccaacs

>>> a = IP()
>>> a.dst = '10.9.0.5'
>>> b = ICMP()
>>> p = a/b
>>> send(p)
.
Sent 1 packets.
>>> 
```

Then with our sniffer.py running a new function:

```
#Task 1.1B --- Capturing only the ICMP packets
def icmp_only():
    pkt = sniff(iface='br-8928c17f4ab4' , filter='icmp', prn=pprint_pkt)
```

We see the ICMP packets neatly on the VM



```
[09/23/21]seed@seed-VirtualBox:~/.../sniffspoof$ sudo ./sniffer.py
10.9.0.1->10.9.0.5
10.9.0.5->10.9.0.1
```

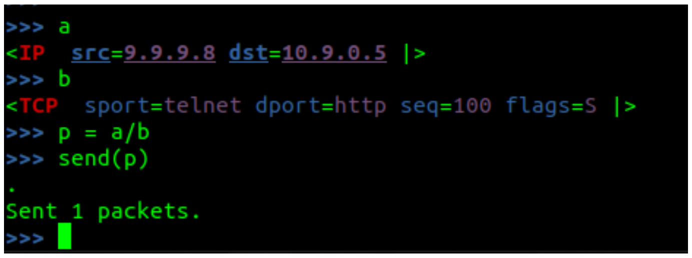
And say we want to send a TCP packet on the same interface, we get no packets showing on our VM terminal. Thanks BPF!

The next task requires us to capture ANY TCP packets that come from a particular IP with a destination port 23. Using the filter option in the sniff() function from scapy allows us to accomplish this simply.

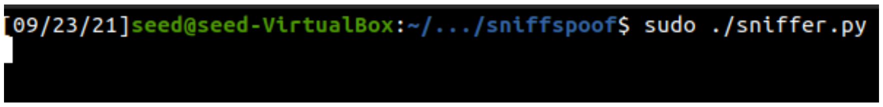
We use the filters option in sniff() scapy function like this:

```
#Task 1.1B --- Capturing only TCP packets going to port 23
def tcp_port23():
    pkt = sniff(iface='br-8928c17f4ab4', filter='tcp and host 9.9.9.9 and port 23', prn=pprint_pkt)
```

Crafting and sending a packet on my attacking machine gets us no response from the sniffer

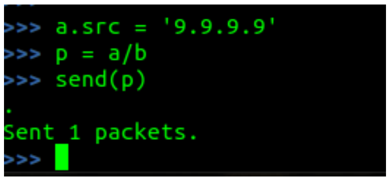


```
>>> a
<IP  src=9.9.9.8 dst=10.9.0.5 |>
>>> b
<TCP  sport=telnet dport=http seq=100 flags=S |>
>>> p = a/b
>>> send(p)
.
Sent 1 packets.
>>> 
```

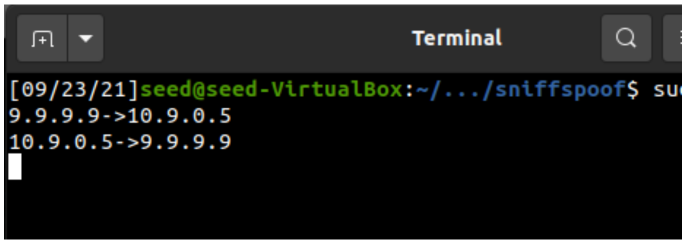


```
[09/23/21]seed@seed-VirtualBox:~/.../sniffspoof$ sudo ./sniffer.py
```

When the packet is set to what we filter for however we receive results.



```
>>> a.src = '9.9.9.9'
>>> p = a/b
>>> send(p)
.
Sent 1 packets.
>>> 
```

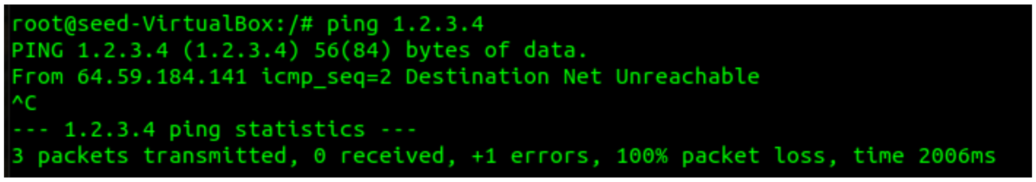


```
[09/23/21]seed@seed-VirtualBox:~/.../sniffspoof$ su
9.9.9.9->10.9.0.5
10.9.0.5->9.9.9.9
```

The next task is to set the filter to capture packets coming from or going to a particular subnet. So pick any subnet that the VM is not attached to and try it out. Since we don't want a subnet the VM is attached to we need to change the interface we are observing. So I used the filter function to search for traffic on the internet facing interface. I then sent a ping to 1.2.3.4 and captured it leaving the attacker container to its destination. I used a class A subnet

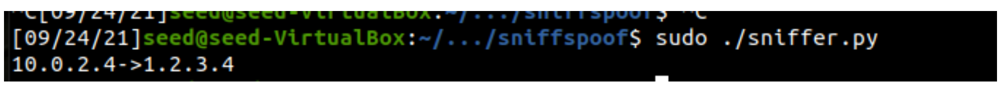
```
#Task 1.1B --- Capturing packets FROM a subnet
def tofrom_subnet():
    pkt = sniff(iface='enp0s3', filter='net 1.0.0.0 mask 255.0.0.0', count=1,prn=pprint_pkt)
```

With this in the python script I sent a packet from my Attacker container to 1.2.3.4



```
root@seed-VirtualBox:/# ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
From 64.59.184.141 icmp_seq=2 Destination Net Unreachable
^C
--- 1.2.3.4 ping statistics ---
3 packets transmitted, 0 received, +1 errors, 100% packet loss, time 2006ms
```

The filter worked!



```
[09/24/21]seed@seed-VirtualBox:~/.../sniffspoof$ sudo ./sniffer.py
10.0.2.4->1.2.3.4
```

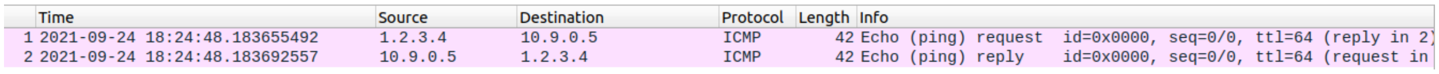
TASK 1.2 Spoofing ICMP Packets

We have already used spoofed packets to easily test our filter settings, but lets go over the details here. Scapy makes it really intuitive to create spoofed packets.

```
# We will construct a ICMP packet layer by layer and send it
```

```
def spoof_icmp():

    print("SENDING SPOOFED ICMP------------\n")
    a = IP()
    a.dst = '10.9.0.5'
    a.src = '1.2.3.4'
    b = ICMP()
    pkt = a/b
    send(pkt)

spoof_icmp()
```

This script comes directly from the SEED lab instructions except we have inserted a new source IP address and when we execute the script and monitor the network device of our attacker machine we can see that the ICMP packet is sent out with the spoofed source IP address. I used wireshark, but our packet sniffer.py would work to see this as well.



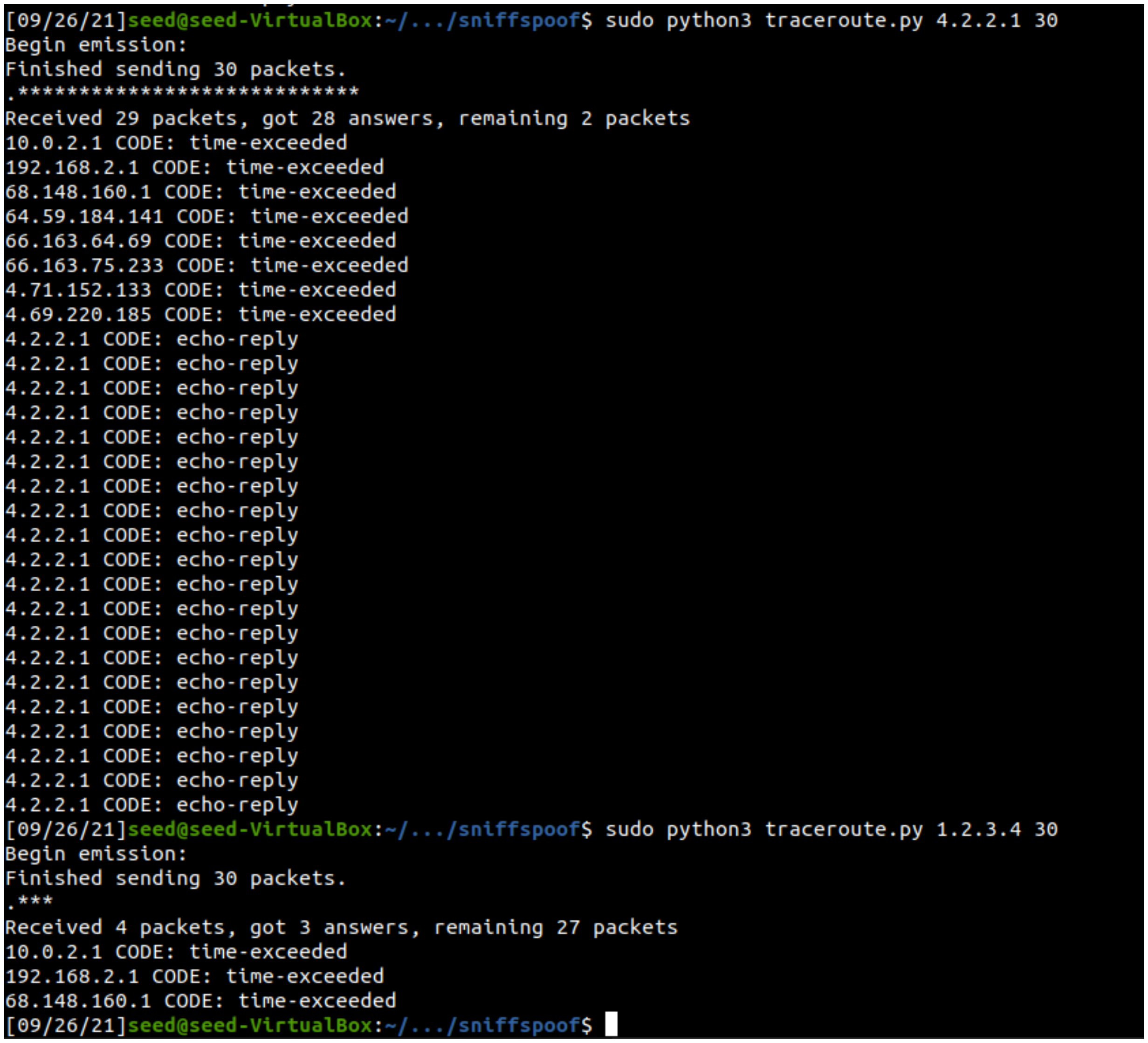
	Time	Source	Destination	Protocol	Length	Info
1	2021-09-24 18:24:48.183655492	1.2.3.4	10.9.0.5	ICMP	42	Echo (ping) request id=0x0000, seq=0/0, ttl=64 (reply in 2)
2	2021-09-24 18:24:48.183692557	10.9.0.5	1.2.3.4	ICMP	42	Echo (ping) reply id=0x0000, seq=0/0, ttl=64 (request in

Task 1.3 Traceroute

The SEED labs want us to create a simple traceroute tool with scapy. So we will do like traceroute does and send an echo-request with a time to live of 1 and listen for an ICMP "Time Exceeded" response then increment the time-to-live until we get an ICMP "port unreachable". The last packet we send will either be the max ttl we assigned or the host. For reference I referred to a workshop on scapy from user 0xbharath on github

```
#--Program is interactable, change address and ttl as you like with
#  command line arguments
# REF: https://0xbharath.github.io/art-of-packet-crafting-with-scapy/network_recon/traceroute/index.html

def scapy_traceroute(addr, ttl):

    ans,unans=sr(IP(dst=addr,ttl=(1,ttl))/ICMP(), timeout=5)
    ans.summary( lambda s,r : r.sprintf("%IP.src% CODE: %ICMP.type%"))


if __name__ == "__main__":

    if len(sys.argv) < 2 or len(sys.argv) > 3:
        print("Simple Scapy traceroute program:\n",
              "USAGE: traceroute.py [ADDRESS] , [TTL]")
        sys.exit(0)

    else:
        addr = sys.argv[1]
        ttl = int(sys.argv[2])
        scapy_traceroute(addr, ttl)
```

This simple tool will output the IP of the source from the packets received as well as the code received at the end. The ICMP codes are important to a tool like traceroute as they allow us to know at what point we reach the destination, or at what point are our probes stopped. A difference in this program is that the scapy sr() function has been given a range of packets to send. It will send ALL packets at once and not one by one like a traceroute -I [address] will give us.



```
[09/26/21]seed@seed-VirtualBox:~/.../sniffspoof$ sudo python3 traceroute.py 4.2.2.1 30
Begin emission:
Finished sending 30 packets.
.****************************
Received 29 packets, got 28 answers, remaining 2 packets
10.0.2.1 CODE: time-exceeded
192.168.2.1 CODE: time-exceeded
68.148.160.1 CODE: time-exceeded
64.59.184.141 CODE: time-exceeded
66.163.64.69 CODE: time-exceeded
66.163.75.233 CODE: time-exceeded
4.71.152.133 CODE: time-exceeded
4.69.220.185 CODE: time-exceeded
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
4.2.2.1 CODE: echo-reply
[09/26/21]seed@seed-VirtualBox:~/.../sniffspoof$ sudo python3 traceroute.py 1.2.3.4 30
Begin emission:
Finished sending 30 packets.
.***
Received 4 packets, got 3 answers, remaining 27 packets
10.0.2.1 CODE: time-exceeded
192.168.2.1 CODE: time-exceeded
68.148.160.1 CODE: time-exceeded
[09/26/21]seed@seed-VirtualBox:~/.../sniffspoof$
```

In my example you see the tool probing two addresses. I didn't refine it further since there is still much more to go through in the lab!

Task 1.4 Sniffing and-then Spoofing

Now we will use scapy to create a Sniff and-then Spoof program. It will run on the VM and interact with the User container. The Computer and Internet Security book by Wenliang Du gives us a nice example to use for the program. I've adapted it for the exercise. REMEMBER!!! The network interface on your setup will be different than mine so be sure to specify the correct iface. I ran these commands from the host interface while sniff_spoof_icmp.py was running from my vm:

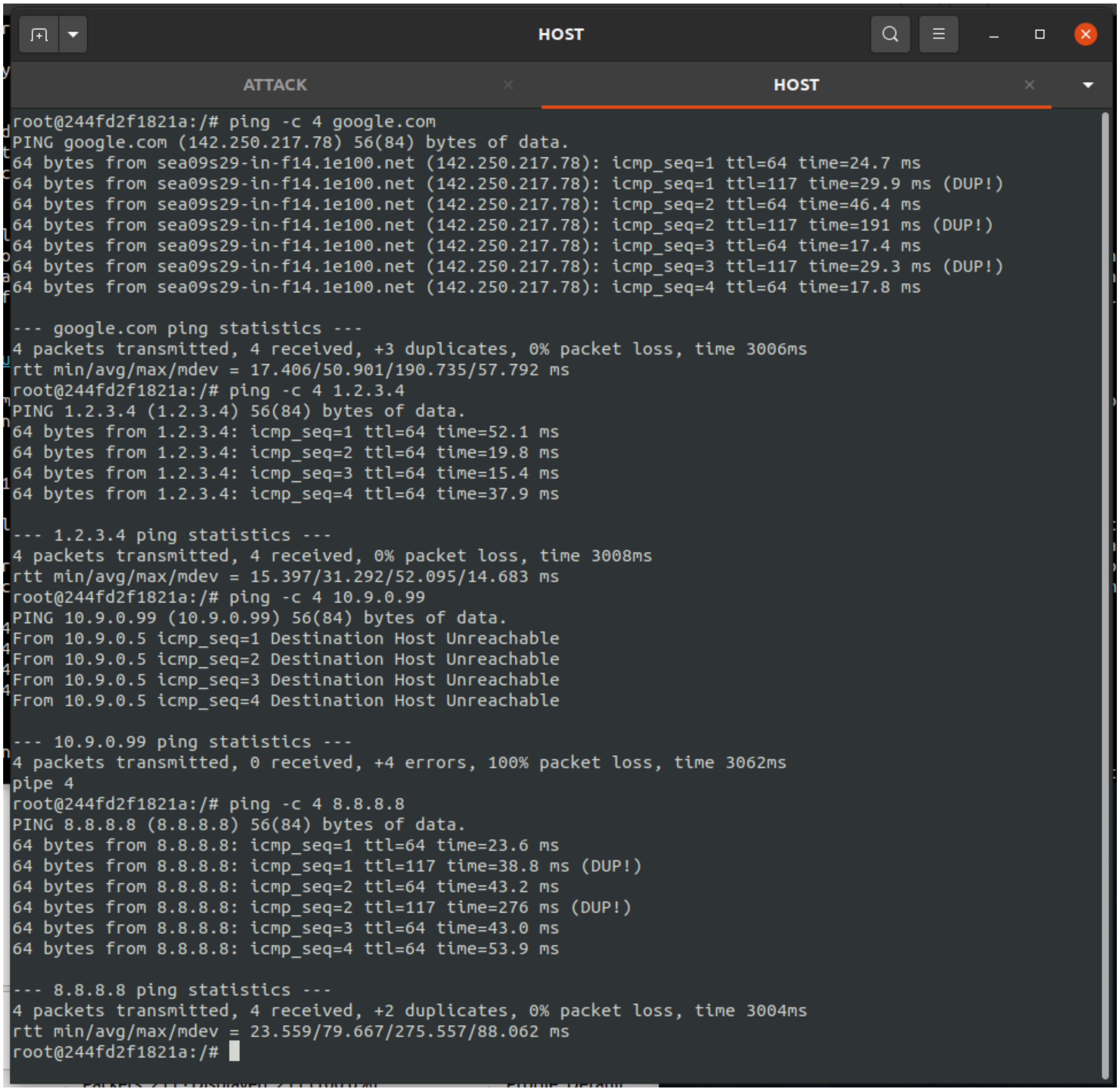
`ping -c 4 google.com` An existing host on the internet

`ping -c 4 1.2.3.4` A non-existing host on the internet

`ping -c 4 10.9.0.99` A non-existing host on the LAN

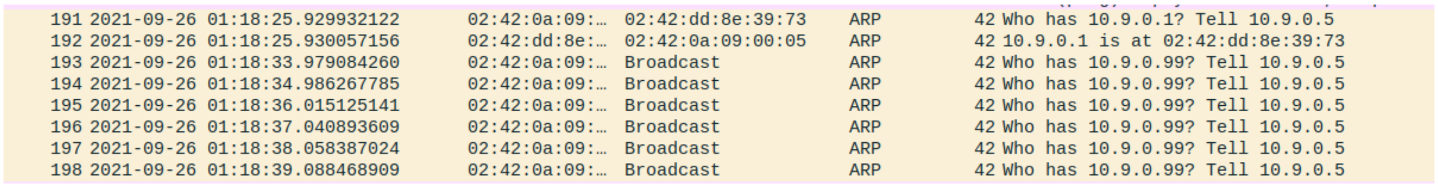
`ping -c 4 8.8.8.8` Another existing host on the internet.

The python program appeared to work! At least for the existing hosts outside of the local network.



```
root@244fd2f1821a:/# ping -c 4 google.com
PING google.com (142.250.217.78) 56(84) bytes of data.
64 bytes from sea09s29-in-f14.1e100.net (142.250.217.78): icmp_seq=1 ttl=64 time=24.7 ms
64 bytes from sea09s29-in-f14.1e100.net (142.250.217.78): icmp_seq=1 ttl=117 time=29.9 ms (DUP!)
64 bytes from sea09s29-in-f14.1e100.net (142.250.217.78): icmp_seq=2 ttl=64 time=46.4 ms
64 bytes from sea09s29-in-f14.1e100.net (142.250.217.78): icmp_seq=2 ttl=117 time=191 ms (DUP!)
64 bytes from sea09s29-in-f14.1e100.net (142.250.217.78): icmp_seq=3 ttl=64 time=17.4 ms
64 bytes from sea09s29-in-f14.1e100.net (142.250.217.78): icmp_seq=3 ttl=117 time=29.3 ms (DUP!)
64 bytes from sea09s29-in-f14.1e100.net (142.250.217.78): icmp_seq=4 ttl=64 time=17.8 ms

--- google.com ping statistics ---
4 packets transmitted, 4 received, +3 duplicates, 0% packet loss, time 3006ms
rtt min/avg/max/mdev = 17.406/50.901/190.735/57.792 ms
root@244fd2f1821a:/# ping -c 4 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
64 bytes from 1.2.3.4: icmp_seq=1 ttl=64 time=52.1 ms
64 bytes from 1.2.3.4: icmp_seq=2 ttl=64 time=19.8 ms
64 bytes from 1.2.3.4: icmp_seq=3 ttl=64 time=15.4 ms
64 bytes from 1.2.3.4: icmp_seq=4 ttl=64 time=37.9 ms

--- 1.2.3.4 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3008ms
rtt min/avg/max/mdev = 15.397/31.292/52.095/14.683 ms
root@244fd2f1821a:/# ping -c 4 10.9.0.99
PING 10.9.0.99 (10.9.0.99) 56(84) bytes of data.
From 10.9.0.5 icmp_seq=1 Destination Host Unreachable
From 10.9.0.5 icmp_seq=2 Destination Host Unreachable
From 10.9.0.5 icmp_seq=3 Destination Host Unreachable
From 10.9.0.5 icmp_seq=4 Destination Host Unreachable

--- 10.9.0.99 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3062ms
pipe 4
root@244fd2f1821a:/# ping -c 4 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=64 time=23.6 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=117 time=38.8 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=64 time=43.2 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=117 time=276 ms (DUP!)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=64 time=43.0 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=64 time=53.9 ms

--- 8.8.8.8 ping statistics ---
4 packets transmitted, 4 received, +2 duplicates, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 23.559/79.667/275.557/88.062 ms
root@244fd2f1821a:/#
```

So why did the local LAN request not go through? We need to understand how the host machine is determining the location of 10.9.0.99, which is the ARP protocol. This is what happened over the network interface when the ping to 10.9.0.99 was sent:



```
191 2021-09-26 01:18:25.929932122    02:42:0a:09:…  02:42:dd:8e:39:73   ARP    42 Who has 10.9.0.1? Tell 10.9.0.5
192 2021-09-26 01:18:25.930057156    02:42:dd:8e:…  02:42:0a:09:00:05   ARP    42 10.9.0.1 is at 02:42:dd:8e:39:73
193 2021-09-26 01:18:33.979084260    02:42:0a:09:…  Broadcast           ARP    42 Who has 10.9.0.99? Tell 10.9.0.5
194 2021-09-26 01:18:34.986267785    02:42:0a:09:…  Broadcast           ARP    42 Who has 10.9.0.99? Tell 10.9.0.5
195 2021-09-26 01:18:36.015125141    02:42:0a:09:…  Broadcast           ARP    42 Who has 10.9.0.99? Tell 10.9.0.5
196 2021-09-26 01:18:37.040893609    02:42:0a:09:…  Broadcast           ARP    42 Who has 10.9.0.99? Tell 10.9.0.5
197 2021-09-26 01:18:38.058387024    02:42:0a:09:…  Broadcast           ARP    42 Who has 10.9.0.99? Tell 10.9.0.5
198 2021-09-26 01:18:39.088468909    02:42:0a:09:…  Broadcast           ARP    42 Who has 10.9.0.99? Tell 10.9.0.5
```

Because 10.9.0.5 and 10.9.0.99 are on the same subnet (remember we set up the machines using the docker-compose.yml script? Yea its in there! The script assigns the network 10.9.0.0 to a subnet:

```
networks:
    net-10.9.0.0:
        name: net-10.9.0.0
        ipam:
            config:
                - subnet: 10.9.0.0/24
```

Visually our network looks like this: (The VM is outside of the subnet, but still has access to the network device)

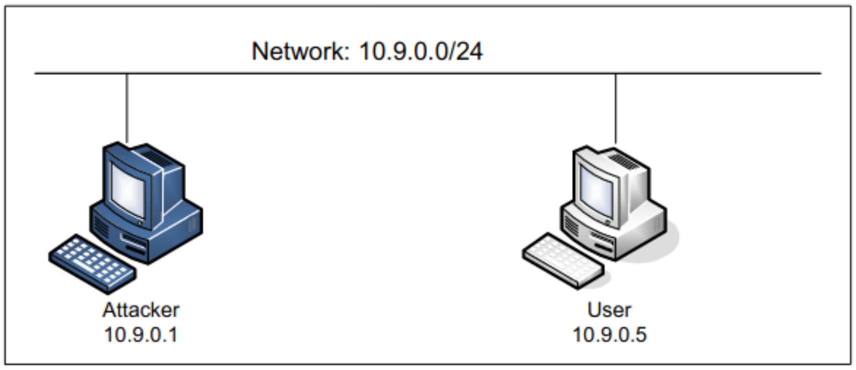


Figure 1: Lab environment setup

When the Host machine tries to create a connection with another machine on its own subnet it first needs to know 10.9.0.99 's MAC address. Since there are no replies to the ARP request and the MAC is not stored in the ARP cache on the host machine a connection cannot be made. Ultimately this is why ARP exists, to route traffic on a LAN through the correct interfaces. Unless we modify our python code to also sniff and then spoof an ARP reply our host machine will not know how to communicate with 10.9.0.99 and the ping to it will never be able to be sent.

Writing Programs to Sniff and Spoof Packets

For this part of the lab the C code gets compiled in the VM and then run out of the volumes directory (shared by attacker and VM) and execute the code inside the attacker container.

Task 2.1 Writing a Packet Sniffing program

Writing a sniffer program in C requires a bit more attention to detail than scapy, but the concept is similar. Firstly we need to open capture session on the specified NIC:

```
handle = pcap_open_live("br-0229d0abdb25", BUFSIZ, 1, 1000, errbuf);
```

pcap_open_live is the library function we use to initialize the session. It fires up a socket for our sniffer to use

```
pcap_compile(handle, &fp, filter_exp, 0, net);
if (pcap_setfilter(handle, &fp) != 0) {
        pcap_perror(handle, "Error:");
        exit(EXIT_FAILURE);
}
```

pcap_compile and pcap_setfilter compiles and sets the BPF filter on our opened socket

```
pcap_loop(handle, -1, [CALLBACK FUNCTION], NULL);
```

pcap_loop invokes a loop running over the socket to grab any packets. The CALLBACK FUNCTION is invoked to perform more operations on the packet.

Before the callback function is written we need to define the different layers of our packet in structs in order to type cast our packet into readable chunks.

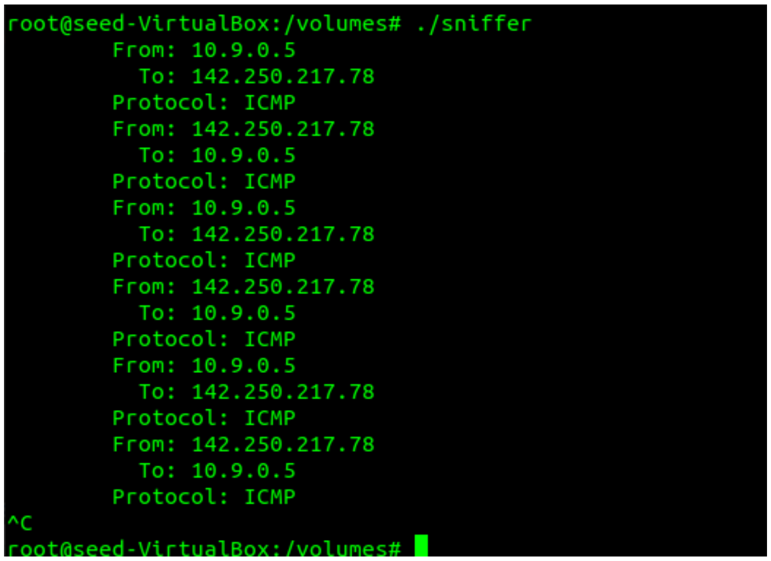
Before typecasting: PACKET:

[...................]

After:

[[ETHER][IP][.....]

We can then increment the sizeof our structure on our captured packet to move the pointer around and analyze each partof it. I used the textbook here for reference and I won't post all of my code in the report, but here's the result of my program capturing packets on my attacker container coming from the host container over the docker network.



```
root@seed-VirtualBox:/volumes# ./sniffer
        From: 10.9.0.5
          To: 142.250.217.78
        Protocol: ICMP
        From: 142.250.217.78
          To: 10.9.0.5
        Protocol: ICMP
        From: 10.9.0.5
          To: 142.250.217.78
        Protocol: ICMP
        From: 142.250.217.78
          To: 10.9.0.5
        Protocol: ICMP
        From: 10.9.0.5
          To: 142.250.217.78
        Protocol: ICMP
        From: 142.250.217.78
          To: 10.9.0.5
        Protocol: ICMP
^C
root@seed-VirtualBox:/volumes#
```

TASK 2.1A Understanding how a Sniffer works

The sniffer works because of the packet capture library. So I will answer the questions in the lab here.

1.

The library calls that are essential for a sniffer program are the calls that create a socket on which to access the NIC through our OS from our program and pcap_open_live() establishes this. We also need a way to understand what we are receiving, by compiling the BPF filter on our socket we are ready to use established filtering expressions on our packet and conversely understanding incoming packets as well we do this with pcap_compile and pcap_setfilter. We then need a call that will loop over the socket in order to capture and analyze the packets, which we have in pcap_loop. We also need a way to close the socket and pcap_close is the library call that we use.

2.

Root privilege is required to run a sniffer program in because we need special access in order to perform actions directly on the NIC, specifically creating a raw socket and activating promiscuous mode on the NIC. The program would fail specifically at the operating system, as the library call is run that opens a raw socket to the NIC. The operating system will attempt the system call required and the kernel will determine the user's privileges and deny the program access to use the system calls invoked by pcap if their privileges are insufficient.

3.

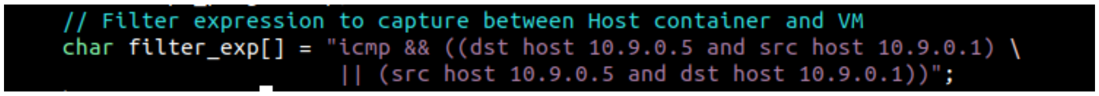
Promiscuous mode allows the NIC to listen to ALL the traffic on its LAN and without promiscuous mode enabled the NIC only see traffic passing to that particular interface. HOWEVER Since the attacker container has been set into host mode it allows the attacker to see ALL traffics on the LAN we set up, it sees all the network interfaces of the host so setting the promiscuous mode to 0 or 1 on the attacker container gives the same result.

TASK 2.1B Writing Filters

We now need to write BPF filter expressions for the sniffer program to capture specific packets.

Capturing ICMP packets between two specific hosts:

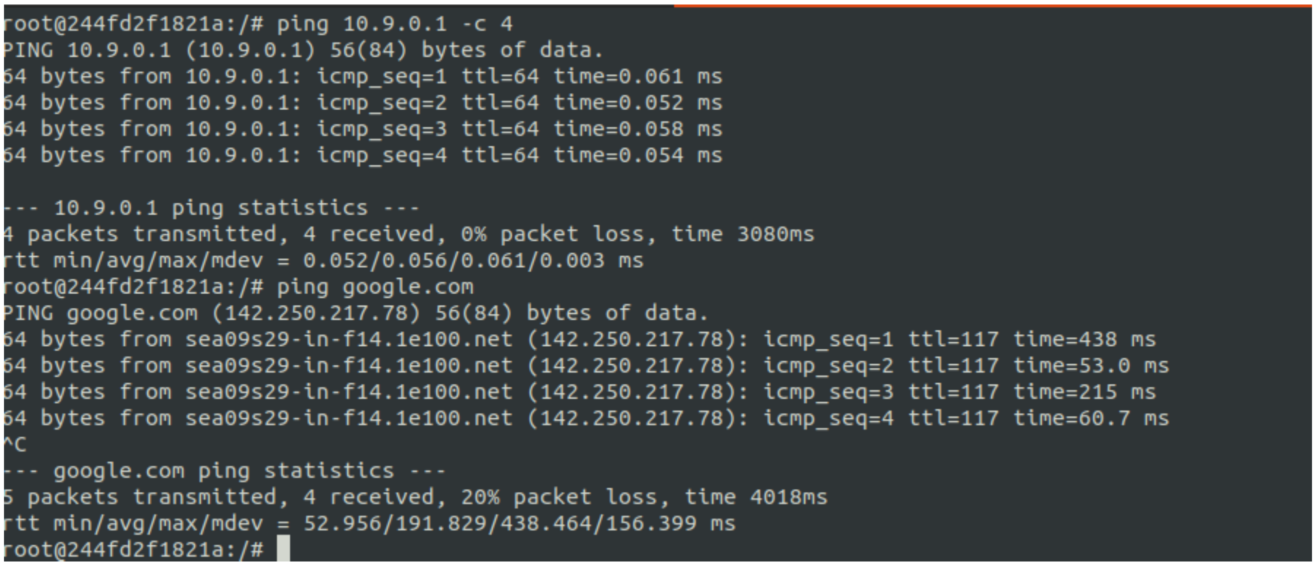
This task just requires us to use a BPF expression in the filter_exp string to ensure we capture the correct packets with sniff_icmp program.



```
    // Filter expression to capture between Host container and VM
    char filter_exp[] = "icmp && ((dst host 10.9.0.5 and src host 10.9.0.1) \
                         || (src host 10.9.0.5 and dst host 10.9.0.1))";
```

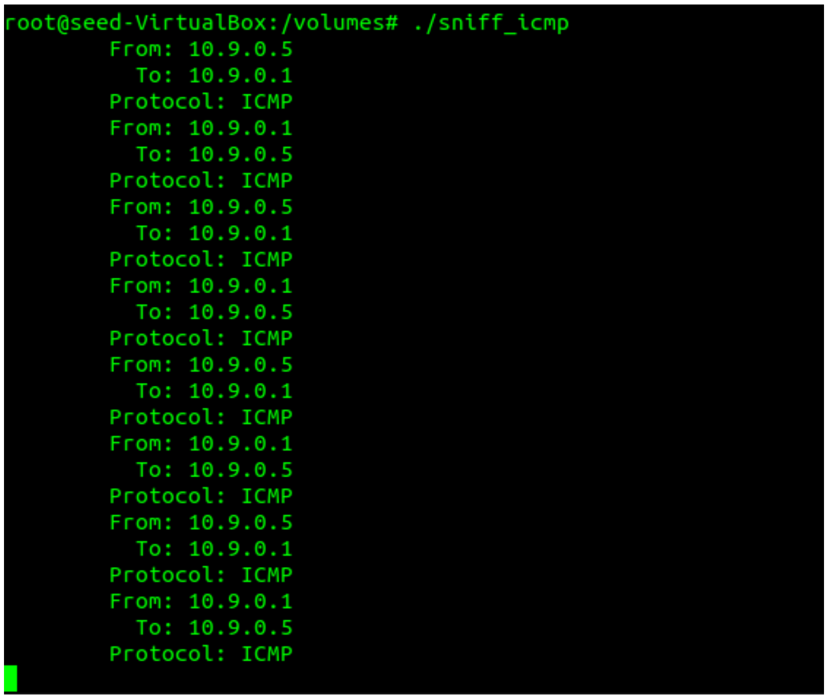
With this expression in place we are capturing packets only between the Host container and the VM

When I send out pings from the host:



```
root@244fd2f1821a:/# ping 10.9.0.1 -c 4
PING 10.9.0.1 (10.9.0.1) 56(84) bytes of data.
64 bytes from 10.9.0.1: icmp_seq=1 ttl=64 time=0.061 ms
64 bytes from 10.9.0.1: icmp_seq=2 ttl=64 time=0.052 ms
64 bytes from 10.9.0.1: icmp_seq=3 ttl=64 time=0.058 ms
64 bytes from 10.9.0.1: icmp_seq=4 ttl=64 time=0.054 ms

--- 10.9.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3080ms
rtt min/avg/max/mdev = 0.052/0.056/0.061/0.003 ms
root@244fd2f1821a:/# ping google.com
PING google.com (142.250.217.78) 56(84) bytes of data.
64 bytes from sea09s29-in-f14.1e100.net (142.250.217.78): icmp_seq=1 ttl=117 time=438 ms
64 bytes from sea09s29-in-f14.1e100.net (142.250.217.78): icmp_seq=2 ttl=117 time=53.0 ms
64 bytes from sea09s29-in-f14.1e100.net (142.250.217.78): icmp_seq=3 ttl=117 time=215 ms
64 bytes from sea09s29-in-f14.1e100.net (142.250.217.78): icmp_seq=4 ttl=117 time=60.7 ms
^C
--- google.com ping statistics ---
5 packets transmitted, 4 received, 20% packet loss, time 4018ms
rtt min/avg/max/mdev = 52.956/191.829/438.464/156.399 ms
root@244fd2f1821a:/#
```

And the output our sniff_icmp program outputs only the icmp packets that were sent between the two hosts. The google.com pings were not intercepted by the sniff_icmp program



```
root@seed-VirtualBox:/volumes# ./sniff_icmp
        From: 10.9.0.5
          To: 10.9.0.1
        Protocol: ICMP
        From: 10.9.0.1
          To: 10.9.0.5
        Protocol: ICMP
        From: 10.9.0.5
          To: 10.9.0.1
        Protocol: ICMP
        From: 10.9.0.1
          To: 10.9.0.5
        Protocol: ICMP
        From: 10.9.0.5
          To: 10.9.0.1
        Protocol: ICMP
        From: 10.9.0.1
          To: 10.9.0.5
        Protocol: ICMP
        From: 10.9.0.5
          To: 10.9.0.1
        Protocol: ICMP
        From: 10.9.0.1
          To: 10.9.0.5
        Protocol: ICMP
```

The next task is Capturing TCP packets with a destination port number in the range of 10 to 100: tcp10to100

Again we change our filter expression to reflect the packets we want to intercept on the interface.

```
char filter_exp[] = "tcp portrange 10-100";
```

This filter will ensure we are getting the packets we want for the task. In order to test this I went ahead and defined a tcpheader structure inside the code to use the same technique for finding the beginning of the IP header, this time we need to multiply the IP header length field by 4 (since the header length is the number of words the header containrs) and then use the packet pointer to move ahead by the size of the ethernet header and the IP header to arrive at the TCP header. I used the pcap tutorial to find a very useful approach to creating the new header the pcap tutorial by Tim Carstens, here is the link for reference: pcap programming tutorial

```
ip_size = ip->iph_ihl * 4;
struct tcpheader * tcp = (struct tcpheader *)(packet + sizeof(struct ethheader) + ip_size);
```

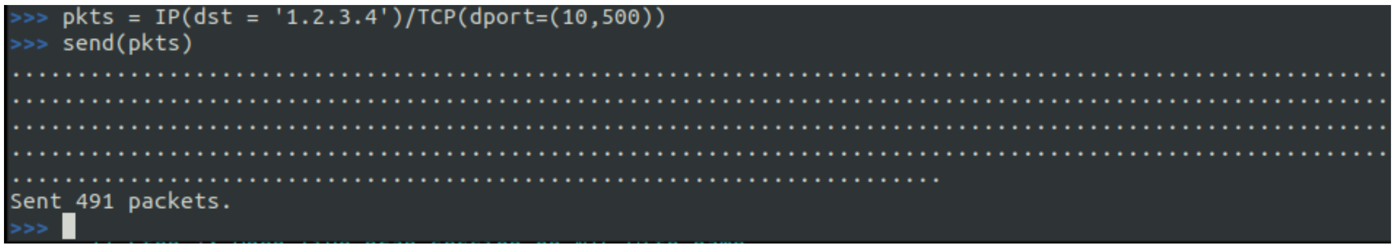
Then to verify that we are only getting the correct ports I've added in a print statement to pull the destination port (what we are filtering for) and print it out for us. We need to use the arpa/inet.h library for this in order to convert the network bytes into host bytes. If we don't convert the byte order then incorrect values will be printed! This function is provided by arpa/inet.h:

```
// This function from arpa/inet.h will convert NETWORK order bytes to HOST order bytes!
uint16_t ntohs(uint16_t netshort);
```

To demonstrate the filter actually works I created a quick Scapy script it sends out tcp packets to 1.2.3.4 from port 10-500. Our program only picks up the first 91 packets (10-100 inclusive) so we know the filter expression works.

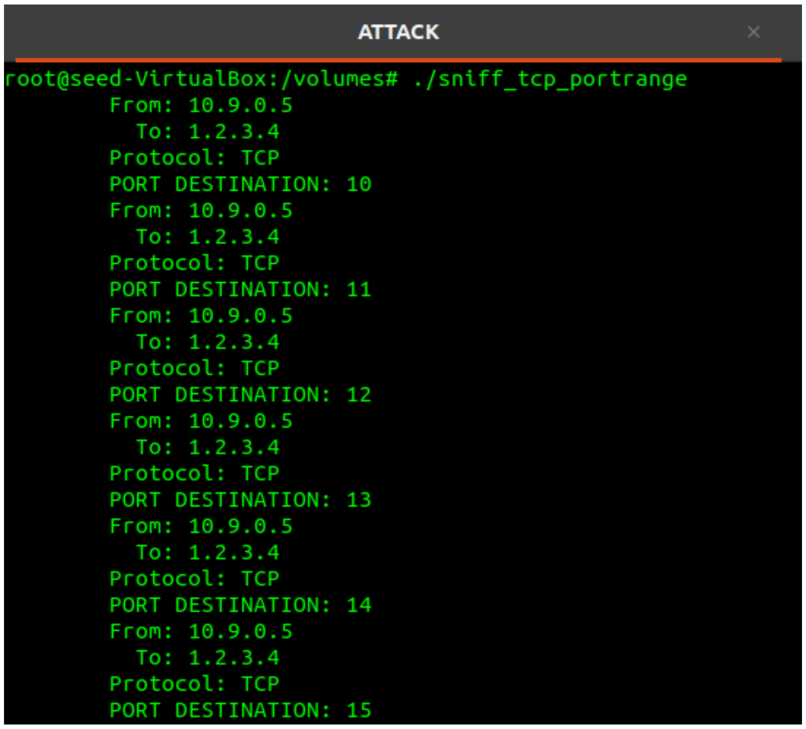
Some screenshots:

The scapy Script



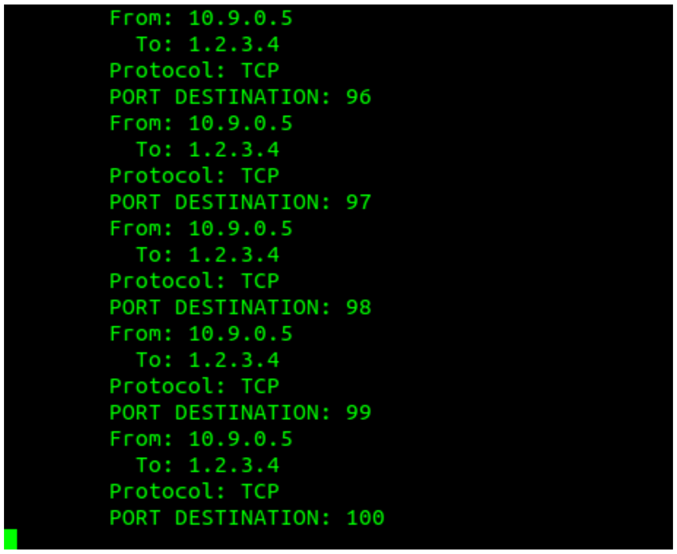
```
>>> pkts = IP(dst = '1.2.3.4')/TCP(dport=(10,500))
>>> send(pkts)
.............................................................................................
.............................................................................................
.............................................................................................
.............................................................................................
..............................................................
Sent 491 packets.
>>> 
```

The program receive packets



```
root@seed-VirtualBox:/volumes# ./sniff_tcp_portrange
        From: 10.9.0.5
          To: 1.2.3.4
        Protocol: TCP
        PORT DESTINATION: 10
        From: 10.9.0.5
          To: 1.2.3.4
        Protocol: TCP
        PORT DESTINATION: 11
        From: 10.9.0.5
          To: 1.2.3.4
        Protocol: TCP
        PORT DESTINATION: 12
        From: 10.9.0.5
          To: 1.2.3.4
        Protocol: TCP
        PORT DESTINATION: 13
        From: 10.9.0.5
          To: 1.2.3.4
        Protocol: TCP
        PORT DESTINATION: 14
        From: 10.9.0.5
          To: 1.2.3.4
        Protocol: TCP
        PORT DESTINATION: 15
```

And skipped a bit to show where the packet capture did not continue after the scapy script executed on host

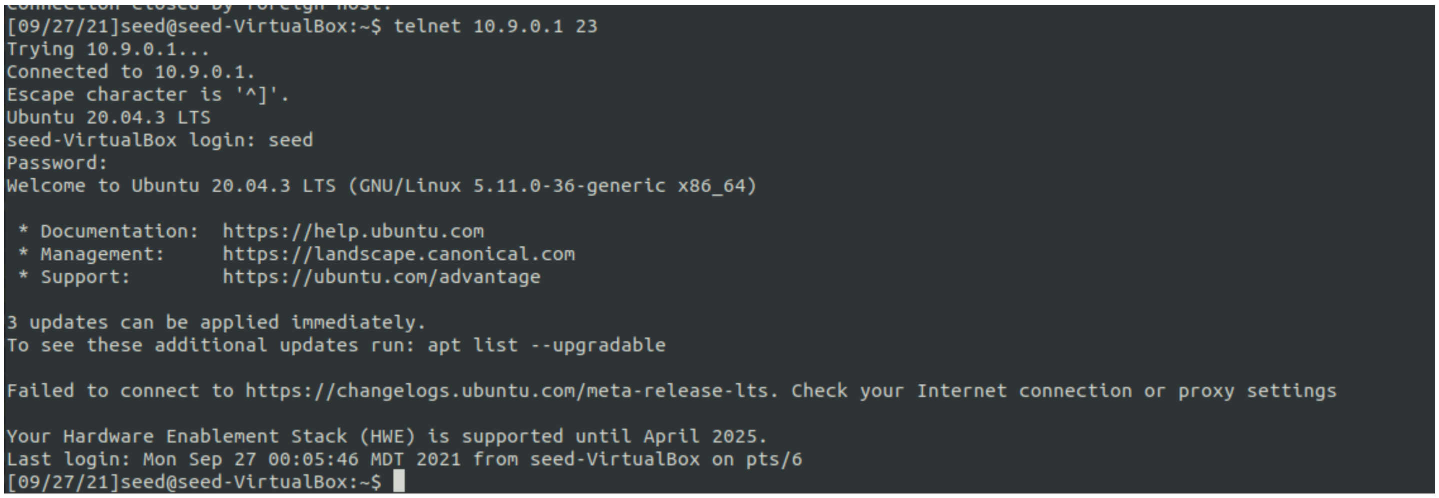


```
        From: 10.9.0.5
          To: 1.2.3.4
        Protocol: TCP
        PORT DESTINATION: 96
        From: 10.9.0.5
          To: 1.2.3.4
        Protocol: TCP
        PORT DESTINATION: 97
        From: 10.9.0.5
          To: 1.2.3.4
        Protocol: TCP
        PORT DESTINATION: 98
        From: 10.9.0.5
          To: 1.2.3.4
        Protocol: TCP
        PORT DESTINATION: 99
        From: 10.9.0.5
          To: 1.2.3.4
        Protocol: TCP
        PORT DESTINATION: 100
```

TASK 2.1C Sniffing Passwords

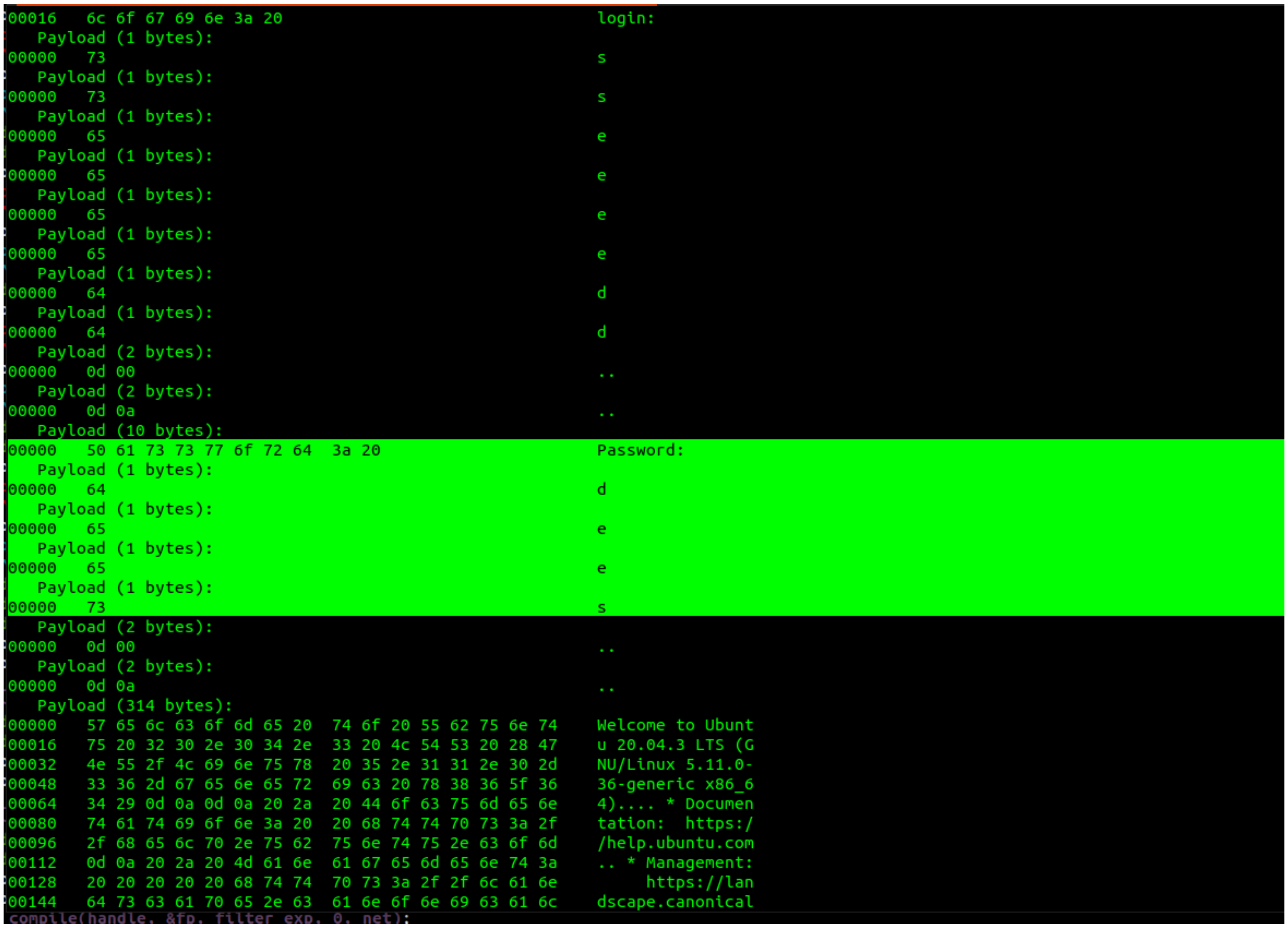
Using our sniffer program to capture telnet passwords on the network we are monitoring this requires us to modify our code again to be able to receive and read the payload data received/sent by each tcp packet. I changed my filter_expression slightly, to grab all traffic from and to port 23 since I will be capturing port 23(telnet) and need to see the password prompt to know when the unsuspecting user is entering their password (and clearly demonstrate it). Again I highly recommend checking out Tim Carsten's tutorial on using pcaplib In my code sniff_telnet_pass.c I have used several of his functions to print out the payload part of the packet in a readable form. Lets take a look at how its working.

The unsuspecting host logs into telnet, blissfully unaware that some malicious actor is listening to the traffic:



```
[09/27/21]seed@seed-VirtualBox:~$ telnet 10.9.0.1 23
Trying 10.9.0.1...
Connected to 10.9.0.1.
Escape character is '^]'.
Ubuntu 20.04.3 LTS
seed-VirtualBox login: seed
Password:
Welcome to Ubuntu 20.04.3 LTS (GNU/Linux 5.11.0-36-generic x86_64)

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

3 updates can be applied immediately.
To see these additional updates run: apt list --upgradable

Failed to connect to https://changelogs.ubuntu.com/meta-release-lts. Check your Internet connection or proxy settings

Your Hardware Enablement Stack (HWE) is supported until April 2025.
Last login: Mon Sep 27 00:05:46 MDT 2021 from seed-VirtualBox on pts/6
[09/27/21]seed@seed-VirtualBox:~$ 
```

sniff_telnet_pass springs into action on the attacker container. Showing us exactly what is going on. And there it is, the password. Thanks!



```
00016   6c 6f 67 69 6e 3a 20                                login: 
   Payload (1 bytes):
00000   73                                                  s
   Payload (1 bytes):
00000   73                                                  s
   Payload (1 bytes):
00000   65                                                  e
   Payload (1 bytes):
00000   65                                                  e
   Payload (1 bytes):
00000   65                                                  e
   Payload (1 bytes):
00000   65                                                  e
   Payload (1 bytes):
00000   64                                                  d
   Payload (1 bytes):
00000   64                                                  d
   Payload (2 bytes):
00000   0d 00                                               ..
   Payload (2 bytes):
00000   0d 0a                                               ..
   Payload (10 bytes):
00000   50 61 73 73 77 6f 72 64  3a 20                      Password: 
   Payload (1 bytes):
00000   64                                                  d
   Payload (1 bytes):
00000   65                                                  e
   Payload (1 bytes):
00000   65                                                  e
   Payload (1 bytes):
00000   73                                                  s
   Payload (2 bytes):
00000   0d 00                                               ..
   Payload (2 bytes):
00000   0d 0a                                               ..
   Payload (314 bytes):
00000   57 65 6c 63 6f 6d 65 20  74 6f 20 55 62 75 6e 74    Welcome to Ubunt
00016   75 20 32 30 2e 30 34 2e  33 20 4c 54 53 20 28 47    u 20.04.3 LTS (G
00032   4e 55 2f 4c 69 6e 75 78  20 35 2e 31 31 2e 30 2d    NU/Linux 5.11.0-
00048   33 36 2d 67 65 6e 65 72  69 63 20 78 38 36 5f 36    36-generic x86_6
00064   34 29 0d 0a 0d 0a 20 2a  20 44 6f 63 75 6d 65 6e    4).... * Documen
00080   74 61 74 69 6f 6e 3a 20  20 68 74 74 70 73 3a 2f    tation:  https:/
00096   2f 68 65 6c 70 2e 75 62  75 6e 74 75 2e 63 6f 6d    /help.ubuntu.com
00112   0d 0a 20 2a 20 4d 61 6e  61 67 65 6d 65 6e 74 3a    .. * Management:
00128   20 20 20 20 20 68 74 74  70 73 3a 2f 2f 6c 61 6e         https://lan
00144   64 73 63 61 70 65 2e 63  61 6e 6f 6e 69 63 61 6c    dscape.canonical
compile(handle, &fp, filter_exp, 0, net);
```

TASK 2.2 Spoofing

Spoofing packets in C requires the use of Raw Sockets in order to be able to spoof the source IP address, when using normal sockets the OS will fill in that information for you. So we need to get our hands dirty and construct packets in the same fashion we were deconstructing packets with the sniffing program. We will need to also construct additional headers for the tasks ahead, like the ICMP header. In short we will need some familiarity with socket programming in C to utilize our program to send packets.

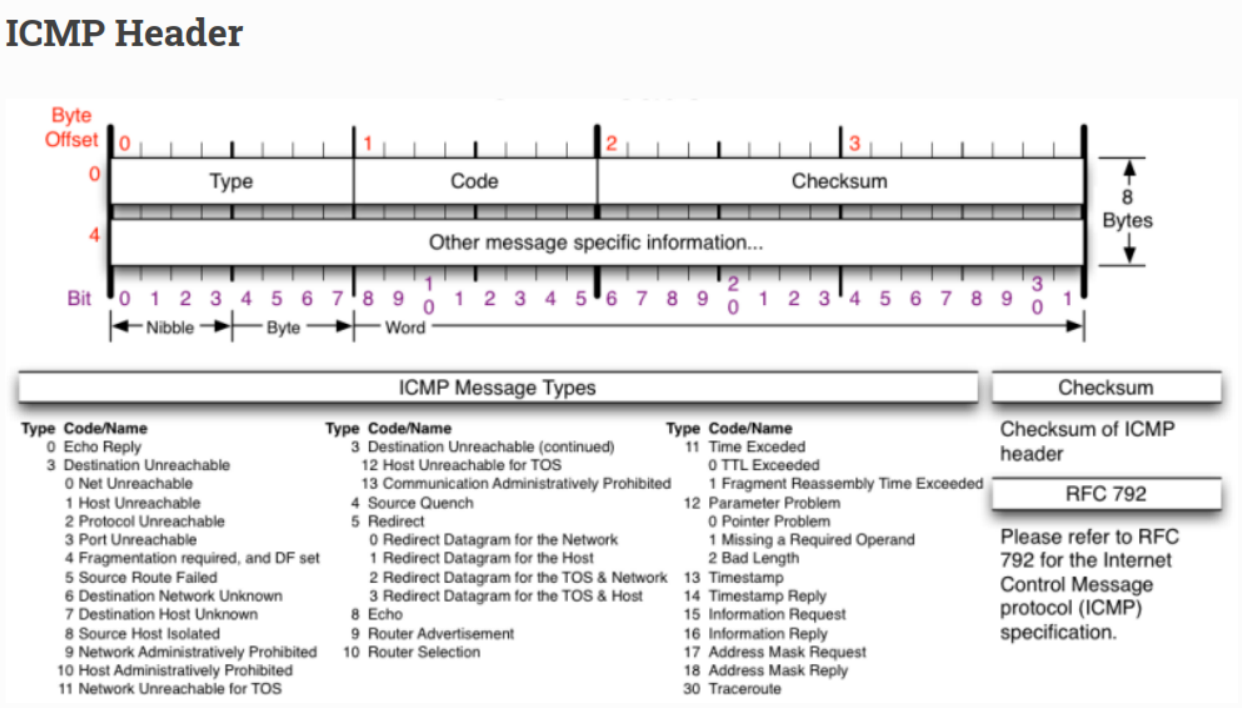
Task 2.2A Write a spoofing program

For my program I closely followed Wenliang Du's Computer and Network Security book chapter 15. I just simply named it spoof.

Firstly I construct the headers I will need for the program, I simply copied the ones I already had from the sniffing program and created a new one for the ICMP header based off the ICMP specification.

```
/* 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;   // Packet Length
        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
};

/* TCP Header */
struct tcpheader {

        unsigned short int      tcph_sport;     // Source Port
        unsigned short int      tcph_dport;     // Destination Port
        unsigned int            tcph_seq;       // Sequence number
        unsigned int            tcph_ack;       // Acknowledgement Number
        unsigned char           tcph_offs;      // data offset
#define TH_OFF(th)      (((th)->tcph_offs & 0xf0) >> 4)
        unsigned char           tcph_flags;     // TCP flags
#define TH_FIN 0x01
#define TH_SYN 0x02
#define TH_RST 0x04
#define TH_PUSH 0x08
#define TH_ACK 0x10
#define TH_URG 0x20
#define TH_ECE 0x40
#define TH_CWR 0x80
#define TH_FLAGS (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
        unsigned short int      tcph_win;       // Window
        unsigned short int      tcph_sum;       // checksum
        unsigned short int      tcph_urp;       // urgent pointer

};


/* ICMP header */

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;     //Identifier
        unsigned short int icmp_seq;    // Sequence number
};
```

I used the diagram here for reference.



ICMP Message Types

Type	Code/Name	Type	Code/Name	Type	Code/Name
0	Echo Reply	3	Destination Unreachable (continued)	11	Time Exceded
3	Destination Unreachable		12 Host Unreachable for TOS		0 TTL Exceeded
	0 Net Unreachable		13 Communication Administratively Prohibited		1 Fragment Reassembly Time Exceeded
	1 Host Unreachable	4	Source Quench	12	Parameter Problem
	2 Protocol Unreachable	5	Redirect		0 Pointer Problem
	3 Port Unreachable		0 Redirect Datagram for the Network		1 Missing a Required Operand
	4 Fragmentation required, and DF set		1 Redirect Datagram for the Host		2 Bad Length
	5 Source Route Failed		2 Redirect Datagram for the TOS & Network	13	Timestamp
	6 Destination Network Unknown		3 Redirect Datagram for the TOS & Host	14	Timestamp Reply
	7 Destination Host Unknown	8	Echo	15	Information Request
	8 Source Host Isolated	9	Router Advertisement	16	Information Reply
	9 Network Administratively Prohibited	10	Router Selection	17	Address Mask Request
	10 Host Administratively Prohibited			18	Address Mask Reply
	11 Network Unreachable for TOS			30	Traceroute

Checksum

Checksum of ICMP header

RFC 792

Please refer to RFC 792 for the Internet Control Message protocol (ICMP) specification.

The second step is to assemble the packet in our code

```
/******************************
1. Fill in the ICMP header
******************************/

// HERE WE DO POINTER MAGIC WITH THE BUFFER WE CREATE IN MAIN()
struct icmpheader *icmp = (struct icmpheader *)(buffer + sizeof(struct ipheader));
icmp->icmp_type = 8; // Type 8 is echo request, 0 is reply.
icmp->icmp_seq = 69;
icmp->icmp_id = 420;
//Calculate checksum for integrity
icmp->icmp_chksum = 0;
icmp->icmp_chksum = in_cksum((unsigned short *) icmp, sizeof(struct icmpheader));

/******************************
2. Fill in the IP Header
******************************/

// See my code in the project for filling the IP header.
```

Filling in ALL of the fields of the ICMP header proved to be important because otherwise we ended up with a malformed packet being transmitted over the wire. This means we would never receive a reply to our spoofed packet. Though there are attacks that are based around malformed packets (See ping of death) the next task asks us to specifically spoof a packet in order to receive a reply so I did not want my spoofed packets to be rejected.

The next step is to ensure our checksum is valid for integrity and this was also code provided by the book which matches the RFC 1071 specifications.

```
/*************************************************************************
 * Calculate a checksum for a given buffer
 *************************************************************************/
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 the 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);


}
```

The checksum for the IP header is calculated by the OS, but when we are using other protocols we must calculate it ourselves.

Finally we get to the raw socket programming where we send out assembled packet. Again I used the code Wenliang Du provides in Computer and Internet Security. With a few changes to suit our ICMP spoofing program

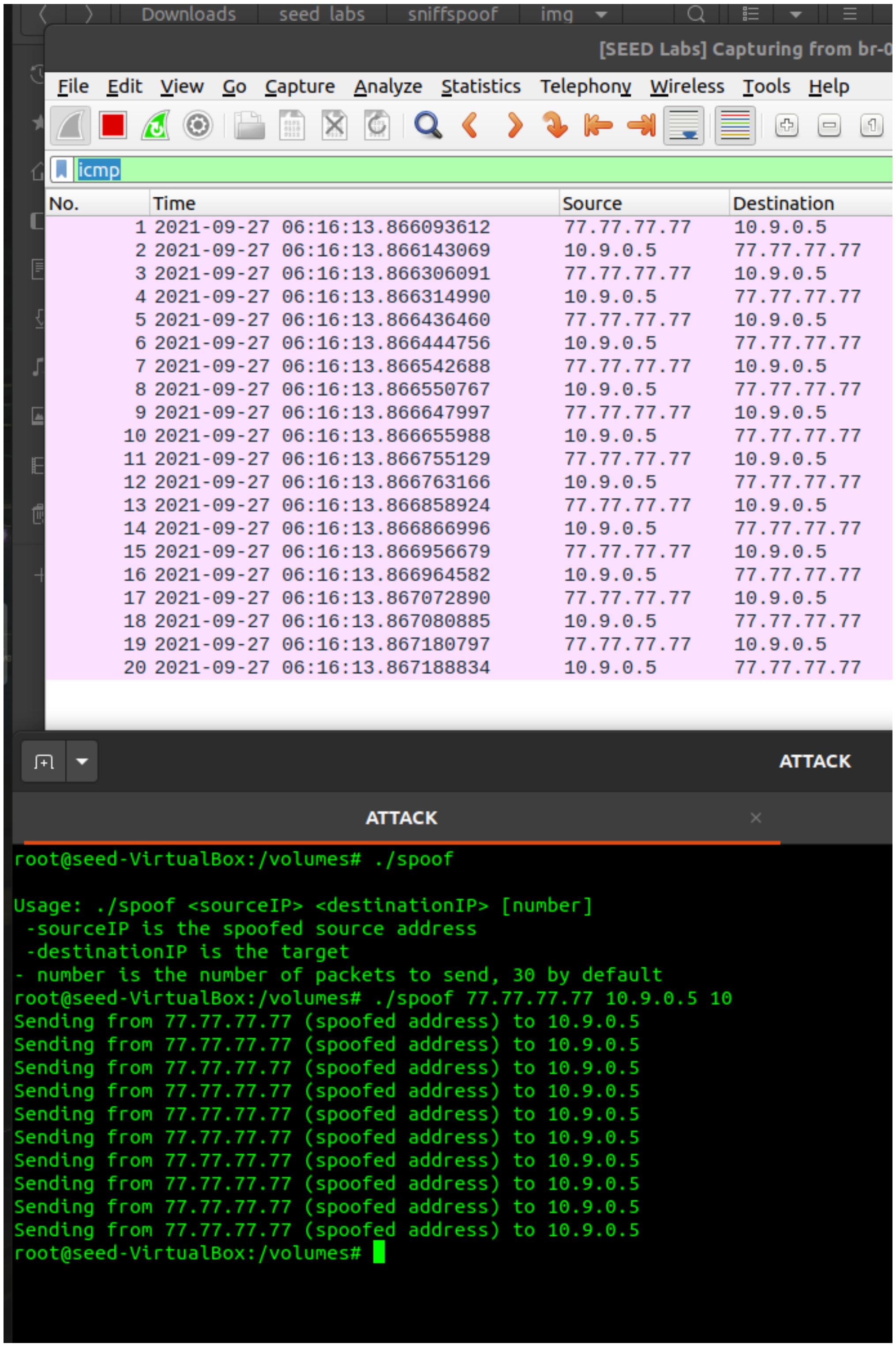
```
/*************************************************************
 * Given a IP packet, send it out using a raw socket.
 *************************************************************/
void send_raw_ip_packet (struct ipheader* ip)
{
        struct sockaddr_in dest_info;
        int enable = 1;

        // Create raw network socket.
        int sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

        // Step 2 Set Socket option.
        setsockopt(sock, IPPROTO_ICMP, IP_HDRINCL, &enable, sizeof(enable));

        // Step 3 provide needed information about destination.
        dest_info.sin_family = AF_INET;
        dest_info.sin_addr = ip->iph_destip;

        //Step 4 send the packet out
        sendto(sock, ip, ntohs(ip->iph_len), 0,
                        (struct sockaddr *)&dest_info, sizeof(dest_info));
        close(sock);
}
```

I have changed the Ip protocol to ICMP here, that is the only change I have made from the book's code.

With the packet constructed, the checksum calculated, and then sent out through the socket we can finally start looking at some results.

I have made my program an interface to facilitate showing for both tasks in 2.2 sending spoofed packets (ICMP uses IP) and will post the wireshark capture so you can see that it does in fact work.

Here is the program spoofing an arbitrary IP address:

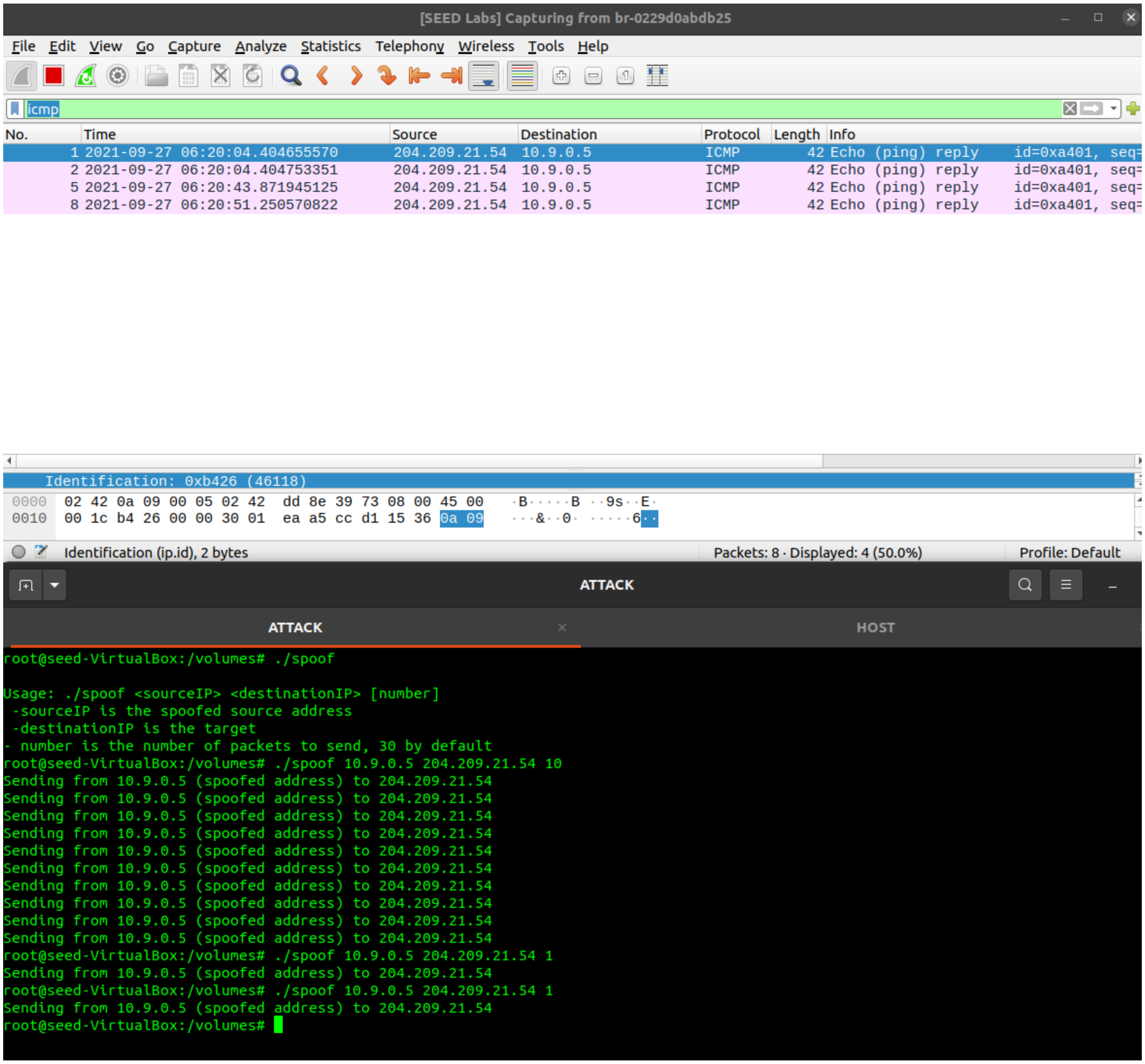

Downloads
seed labs
sniffspoof
img
[SEED Labs] Capturing from br-
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
icmp
ATTACK
ATTACK

```
No.  Time                            Source        Destination
   1 2021-09-27 06:16:13.866093612  77.77.77.77   10.9.0.5
   2 2021-09-27 06:16:13.866143069  10.9.0.5      77.77.77.77
   3 2021-09-27 06:16:13.866306091  77.77.77.77   10.9.0.5
   4 2021-09-27 06:16:13.866314990  10.9.0.5      77.77.77.77
   5 2021-09-27 06:16:13.866436460  77.77.77.77   10.9.0.5
   6 2021-09-27 06:16:13.866444756  10.9.0.5      77.77.77.77
   7 2021-09-27 06:16:13.866542688  77.77.77.77   10.9.0.5
   8 2021-09-27 06:16:13.866550767  10.9.0.5      77.77.77.77
   9 2021-09-27 06:16:13.866647997  77.77.77.77   10.9.0.5
  10 2021-09-27 06:16:13.866655988  10.9.0.5      77.77.77.77
  11 2021-09-27 06:16:13.866755129  77.77.77.77   10.9.0.5
  12 2021-09-27 06:16:13.866763166  10.9.0.5      77.77.77.77
  13 2021-09-27 06:16:13.866858924  77.77.77.77   10.9.0.5
  14 2021-09-27 06:16:13.866866996  10.9.0.5      77.77.77.77
  15 2021-09-27 06:16:13.866956679  77.77.77.77   10.9.0.5
  16 2021-09-27 06:16:13.866964582  10.9.0.5      77.77.77.77
  17 2021-09-27 06:16:13.867072890  77.77.77.77   10.9.0.5
  18 2021-09-27 06:16:13.867080885  10.9.0.5      77.77.77.77
  19 2021-09-27 06:16:13.867180797  77.77.77.77   10.9.0.5
  20 2021-09-27 06:16:13.867188834  10.9.0.5      77.77.77.77
```

```
root@seed-VirtualBox:/volumes# ./spoof

Usage: ./spoof <sourceIP> <destinationIP> [number]
 -sourceIP is the spoofed source address
 -destinationIP is the target
- number is the number of packets to send, 30 by default
root@seed-VirtualBox:/volumes# ./spoof 77.77.77.77 10.9.0.5 10
Sending from 77.77.77.77 (spoofed address) to 10.9.0.5
Sending from 77.77.77.77 (spoofed address) to 10.9.0.5
Sending from 77.77.77.77 (spoofed address) to 10.9.0.5
Sending from 77.77.77.77 (spoofed address) to 10.9.0.5
Sending from 77.77.77.77 (spoofed address) to 10.9.0.5
Sending from 77.77.77.77 (spoofed address) to 10.9.0.5
Sending from 77.77.77.77 (spoofed address) to 10.9.0.5
Sending from 77.77.77.77 (spoofed address) to 10.9.0.5
Sending from 77.77.77.77 (spoofed address) to 10.9.0.5
Sending from 77.77.77.77 (spoofed address) to 10.9.0.5
root@seed-VirtualBox:/volumes# 
```

Task 2.2B Spoof an ICMP echo request.

For this task we use the spoof program to send an ICMP echo request on behalf of the Host container from the attacker container, we capture the replies on our Container network interface. I chose to ping my school's student server (MacEwan University in Edmonton).


[SEED Labs] Capturing from br-0229d0abdb25
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
icmp
ATTACK
ATTACK
HOST

```
No.   Time                            Source          Destination  Protocol  Length Info
    1 2021-09-27 06:20:04.404655570   204.209.21.54   10.9.0.5     ICMP         42 Echo (ping) reply    id=0xa401, seq=
    2 2021-09-27 06:20:04.404753351   204.209.21.54   10.9.0.5     ICMP         42 Echo (ping) reply    id=0xa401, seq=
    5 2021-09-27 06:20:43.871945125   204.209.21.54   10.9.0.5     ICMP         42 Echo (ping) reply    id=0xa401, seq=
    8 2021-09-27 06:20:51.250570822   204.209.21.54   10.9.0.5     ICMP         42 Echo (ping) reply    id=0xa401, seq=

   Identification: 0xb426 (46118)
0000  02 42 0a 09 00 05 02 42  dd 8e 39 73 08 00 45 00   ·B·····B ··9s··E·
0010  00 1c b4 26 00 00 30 01  ea a5 cc d1 15 36 0a 09   ···&··0· ·····6··

Identification (ip.id), 2 bytes          Packets: 8 · Displayed: 4 (50.0%)          Profile: Default
```

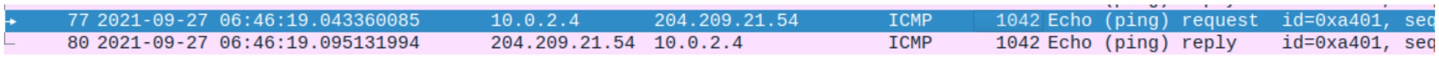
```
root@seed-VirtualBox:/volumes# ./spoof

Usage: ./spoof <sourceIP> <destinationIP> [number]
 -sourceIP is the spoofed source address
 -destinationIP is the target
- number is the number of packets to send, 30 by default
root@seed-VirtualBox:/volumes# ./spoof 10.9.0.5 204.209.21.54 10
Sending from 10.9.0.5 (spoofed address) to 204.209.21.54
Sending from 10.9.0.5 (spoofed address) to 204.209.21.54
Sending from 10.9.0.5 (spoofed address) to 204.209.21.54
Sending from 10.9.0.5 (spoofed address) to 204.209.21.54
Sending from 10.9.0.5 (spoofed address) to 204.209.21.54
Sending from 10.9.0.5 (spoofed address) to 204.209.21.54
Sending from 10.9.0.5 (spoofed address) to 204.209.21.54
Sending from 10.9.0.5 (spoofed address) to 204.209.21.54
Sending from 10.9.0.5 (spoofed address) to 204.209.21.54
Sending from 10.9.0.5 (spoofed address) to 204.209.21.54
root@seed-VirtualBox:/volumes# ./spoof 10.9.0.5 204.209.21.54 1
Sending from 10.9.0.5 (spoofed address) to 204.209.21.54
root@seed-VirtualBox:/volumes# ./spoof 10.9.0.5 204.209.21.54 1
Sending from 10.9.0.5 (spoofed address) to 204.209.21.54
root@seed-VirtualBox:/volumes# 
```

Interestingly when I send out 10 packets I only receive two responses. This is either because my packets are being sent too quickly, or the server will only respond to a set number or ICMP requests/second. But when I send one packet at a time from the attacker container masquerading as the host machine we can see that the MacEwan student server responds with a reply to my spoofed packet.

Questions

4.

The IP packet length cannot be set to an arbitrary value. For several reasons. If you set the ip length higher than the packet actually is, then the packet will become padded with zeros, so yes you can increase the size of your packet, but there is a limit As defined in RFC 791, the maximum packet length of an IPv4 packet including the IP header is 65,535 (216 – 1) bytes, and before we even think of making a packet that large we would need to fragment the packet because of the maximum transmission unit(MTU) across an ethernet wire. Trying to jam a very large packet onto the network will get error'd out by the socket before it is sent. playing with my code a bit, I can increse the size of my echo requests, but not to an arbitrary value. The largest I was able to make the packet length as was 1042 bytes.



```
   77 2021-09-27 06:46:19.043360085   10.0.2.4        204.209.21.54  ICMP     1042 Echo (ping) request  id=0xa401, seq
   80 2021-09-27 06:46:19.095131994   204.209.21.54   10.0.2.4       ICMP     1042 Echo (ping) reply    id=0xa401, seq
```

According to the man page if the message size is too large, then sendto() , the socket function we use to send our packet, will also report an errormessage EMSGSIZE. This means in order to increase the size of the message further we would need to handle breaking it down atomically in the code.

5.

Using raw socket programming we do not need to calculate the checksum for the IP header because the operating system does this for us. When we are working with higher level protocols, such as tcp, udp, icmp we need to calculate their checksums.

6.

Similar to question 2. We need root privilege to run programs with raw sockets because it is required by the kernel as a security measure to protect the system. When the OS uses the socket system calls it has a UID attached to the process and the process of creating/using a raw socket will not be allowed unless your UID is 0 (root) or you have certain privileges assigned to your program (CAP_NET_RAW).

TASK 2.3 Sniff and Then Spoof

Bonus

Packet Sniffing/Spoofing Lab