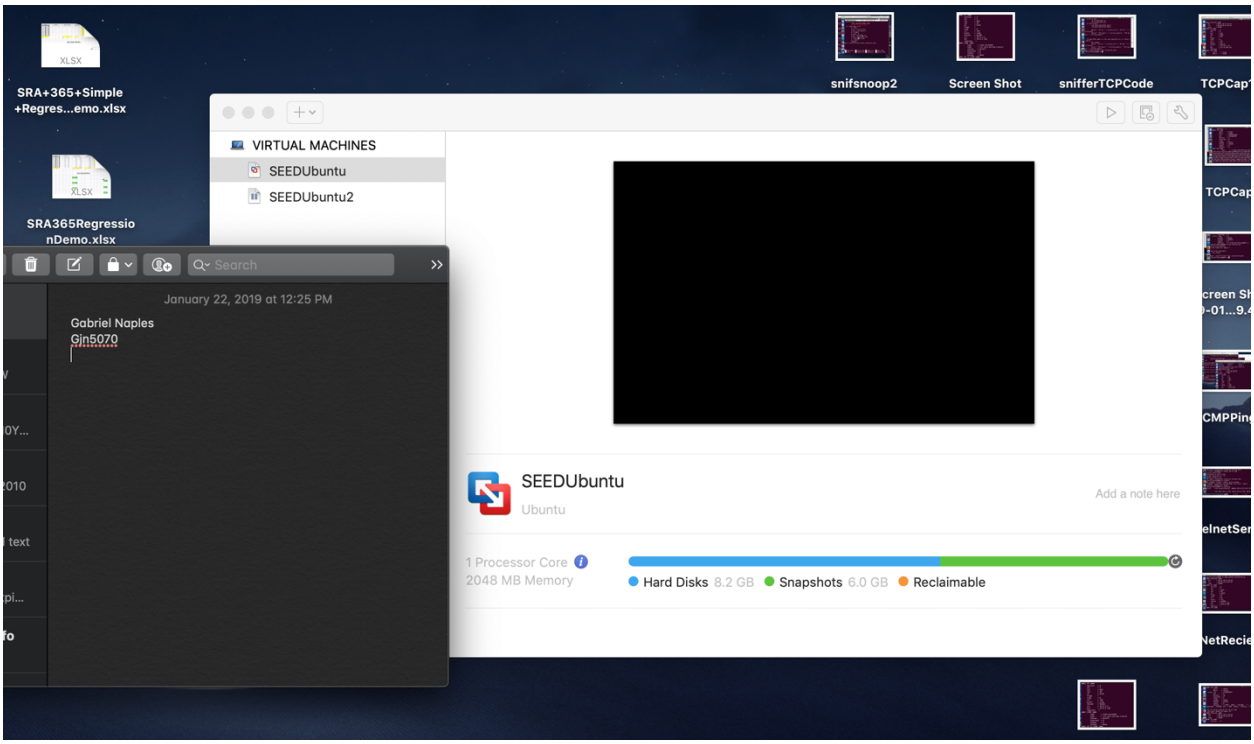
Gabriel Naples
1/23/19
IST 451
Lab 1

Proof of SEED lab download




**Section 2.1**

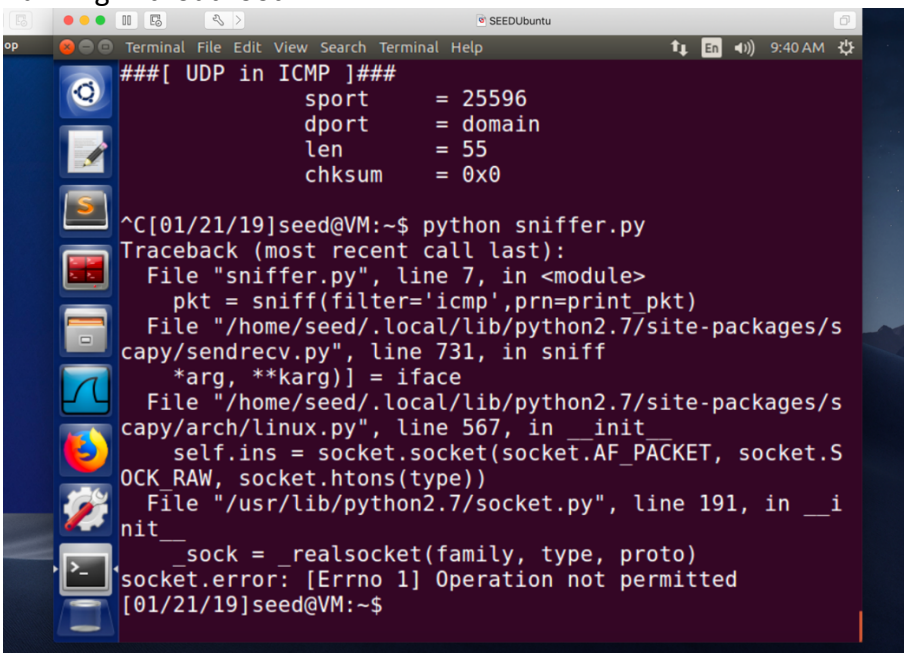
1.1A

For this section I had to run the simple sniffer tool both with and without using the sudo command to gain root privilege. When I ran the sniffer tool without root privilege, I was given an error message that stated that I was not permitted to run that code, and I received no packets. When I used the "sudo python sniffer.py" command I was able to successfully run the sniffer with root access. The 'sudo' command gives me root access and administrative privileges are required to run the sniffer tool.

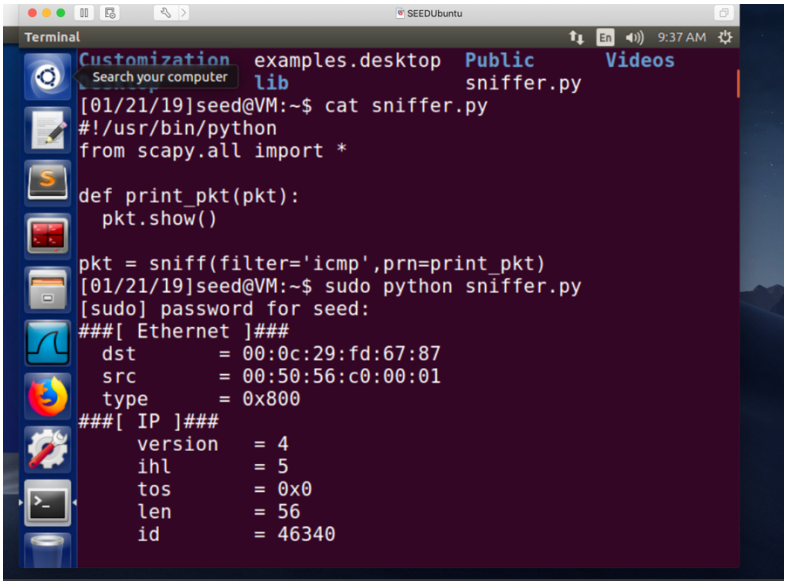
sniffer.py code:

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

def print_pkt(pkt):
  pkt.show()

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

Running without root:




Running with root:




1.1B

For this section I had to apply filters to the packet sniffer program. For the initial capture of just the ICMP packet I used the same simple program that was used above. That program already applies a filter that only takes the ICMP packets and doesn't take other packets. After that I made a more complex program that was able to siphon out TCP packets that came from

either specific IP addresses and went to specific ports. This more complex program is able to be easily modified to allow for filtering in more specific ways (I have 2 pictures of code below for this section as I didn't know the simpler way to do this section until after I made more roundabout way). After that I created a simple filter that scanned all packets that interacted with a specific subnet whether the packets were sourced from that net or being sent to it.

ICMP filter code:

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

def print_pkt(pkt):
  pkt.show()

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

TCP filter (Searches for any TCP packet that comes from the IP 10.0.2.129 and goes to port 23):

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

def print_pkt(pkt):
  if IP in pkt:
        ip_src=pkt[IP].src
        ip_dst=pkt[IP].dst
  if TCP in pkt:
        tcp_sport=pkt[TCP].sport
        tcp_dport=pkt[TCP].dport

        #print " IP src " + str(ip_src)+ " IP dst " + str(ip_dst)
        #print " TCP sport " + str(tcp_sport)+ " TCP dport " + str
(tcp_dport)


  if((pkt[TCP].dport == 23) and (pkt[IP].src == "10.0.2.129")):
        pkt.show()
        print " IP src " + str(ip_src) + " IP dst " + str(ip_dst)
        print " TCP sport " + str(tcp_sport)+ " TCP dport " + str(
tcp_dport)
pkt = sniff(filter='tcp',prn=print_pkt)
```

What this code does is analyzes the packet it receives and only prints the packet if it contains the specific values that I'm telling it to search for. If there is an IP section in the packet then is sets the source ip to a variable (ip_src=pkt[IP].src) and it does the same thing for the destination ip that the packet is being sent. Then it does a similar function in the TCP section of the packet where it stores the source port and destination port to separate variables. It prints the IP source, IP destination, TCP source port, and TCP destination port for every tcp packet it reads (these print statements can be quickly commented out for a more precise display of information, but they give a simple overview of what the other TCP packets are doing). In the following 'if' statement it scans to see if the destination port of the packet is equal to 23 and

then if the source ip address of the packet is 10.0.2.129. If both of these variables are equal, then it prints out the entire packet. This is different than the previous print statements because they only showed the quick information whereas this displays all of the information being held within the packet. The very last line of this program is where you can see that it is filtering for TCP packets, so it is only reading TCP packets.

After I made the above program, I learned that there was a much simpler and straight forward way to do the exact same thing. That code is directly below, however; understanding the above code made it a lot easier for me to write the sniffer and spoofer program for the last section of this lab.

Simpler TCP scanner that takes a specific IP and port:

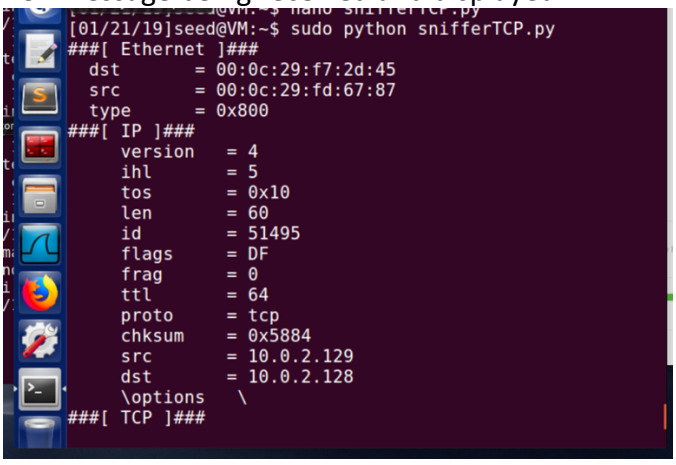
```
from scapy.all import *

def print_pkt(pkt):
  pkt.show()

pkt = sniff(filter = 'ip 10.0.2.128 and tcp dst port 23',prn=print
_pkt)
```

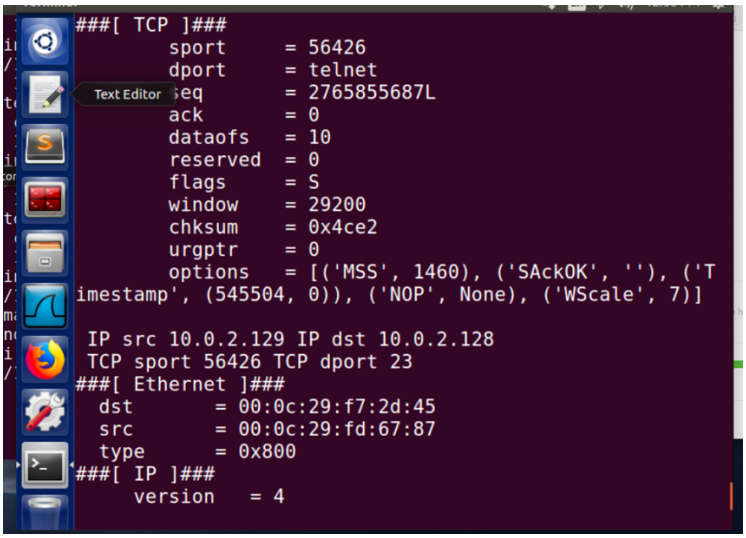
TCP message to port 23 from IP 10.0.2.129 being sent

```
VM login: Connection closed by foreign host.
Terminator
]seed@VM:~$ telnet 10.0.2.128
Trying 10.0.2.128...
Connected to 10.0.2.128.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: Connection closed by foreign host.
```

This is me sending a telnet message from IP 10.0.2.129 to 10.0.2.128. Port 23 is the port designated to telnet messages to this way it guarantees a packet is being sent to port 23 which is a variable that the TCP sniffer was scanning for.

- TCP message being received and displayed:






This picture shows that the dport is telnet which is the destination port the packet is being sent. The text right above the “Ethernet” section of the packet shows the IP source, which is 10.0.2.129 and was being scanned for, as well as the TCP destination port, 23.

Scanning a specific subnet:

```
from scapy.all import *

def print_pkt(pkt):
  pkt.show()

pkt = sniff(filter='net 10.0.2.*', prn=print_pkt)
```

This program scans for all packets interacting with the subnet ’10.0.2.’ which includes all IP addresses within that subnet ex: 10.0.2.128 or 10.0.2.1.

Message saying its scanning:

```
[01/23/19]seed@VM:~$ sudo python snifferNet.py
tcpdump: unknown network '10.0.2.'
###[ Ethernet ]###
  dst       = 00:00:00:00:00:00
  src       = 00:00:00:00:00:00
  type      = 0x86dd
###[ IPv6 ]###
     version   = 6
     tc        = 0
     fl        = 217476
     plen      = 8
     nh        = UDP
     hlim      = 64
     src       = ::1
     dst       = ::1
###[ UDP ]###
        sport     = 59808
        dport     = 45599
        len       = 8
```

**Section 2.2**

1.2 Spoofing

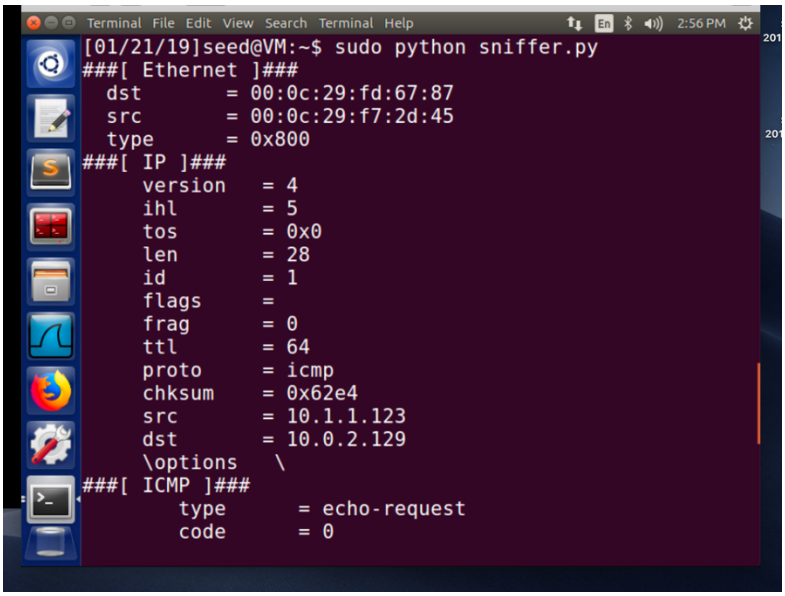
For this section I created a program that compiled a packet and send an ICMP echo request packet to a destination IP, however the packet is spoofed to look like it didn't come from my device, but whatever IP I tell it to appear from.

Spoof program code:

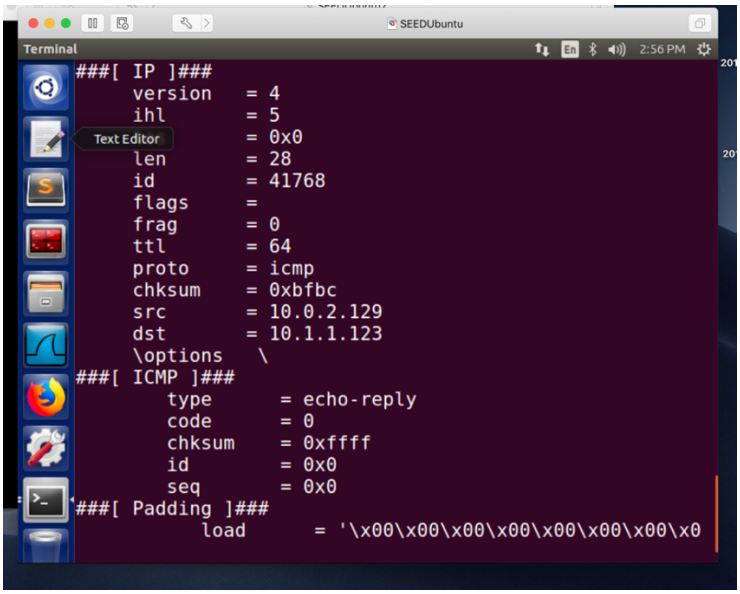
```
from scapy.all import *

a = IP()
a.src = '10.1.1.123'
a.dst = '10.0.2.129'
b = ICMP()
p = a/b
send(p)
```

This program shows that the source of the packet (a.src) comes from the IP address 10.1.1.123 and it gets sent to the destination (a.dst) of 10.0.2.129. This was a successful transfer and it received an echo reply from the destination of 10.0.2.129 to the spoof address of 10.1.1.123.

Spoof packets being received and sent:




You can see in the above screenshot that the source address (src) IP 10.1.1.123 which is the spoofed address.



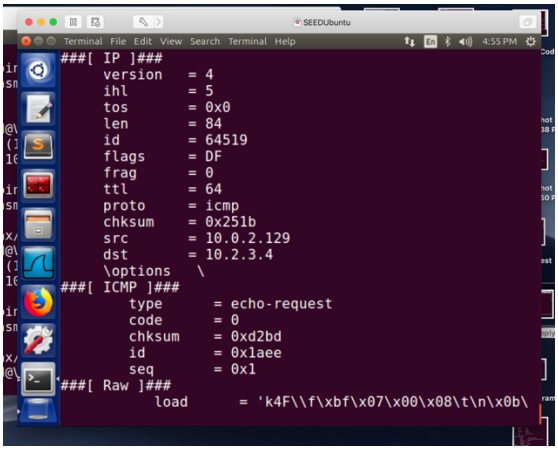

Above shows the source (10.0.2.129) sending a reply to the spoofed IP address (10.1.1.123)

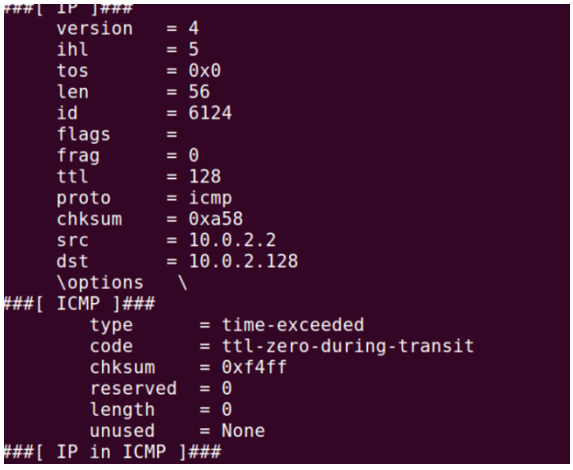
**Section 2.3**

1.3

The point of this section was to essentially manually do the traceroute command through a program that you adjust so you can see the IP of each router a packet jumps to before it reaches its destination. In this section I wrote a small program that sent an ICMP echo – request to a specific IP address but the time to live was set very low, this way the packet would fail at its first jump and the router it failed at would send an error message back to my device. This error packet contains the IP address from the router that sent it back so that gives me the IP of one of the jumps the packet had to make. I then adjusted the time to live to let it reach the next destination and this process was repeated until the packed reached its final destination.

Packets being sent and fail packet being received from router with IP 10.0.2.2:



**Section 2.4**

1.4

For this task I had to create a program that monitored network traffic until it saw an echo request being sent. Then the program read that packet and sent an echo reply to the IP address that sent the request while being masked as the IP the first device was trying to reach. Essentially, device 1 tries to ping device 2, but device 3 intercepts the message and sends a reply pretending to be device 2 back to device 1.

Code to sniff and then spoof:

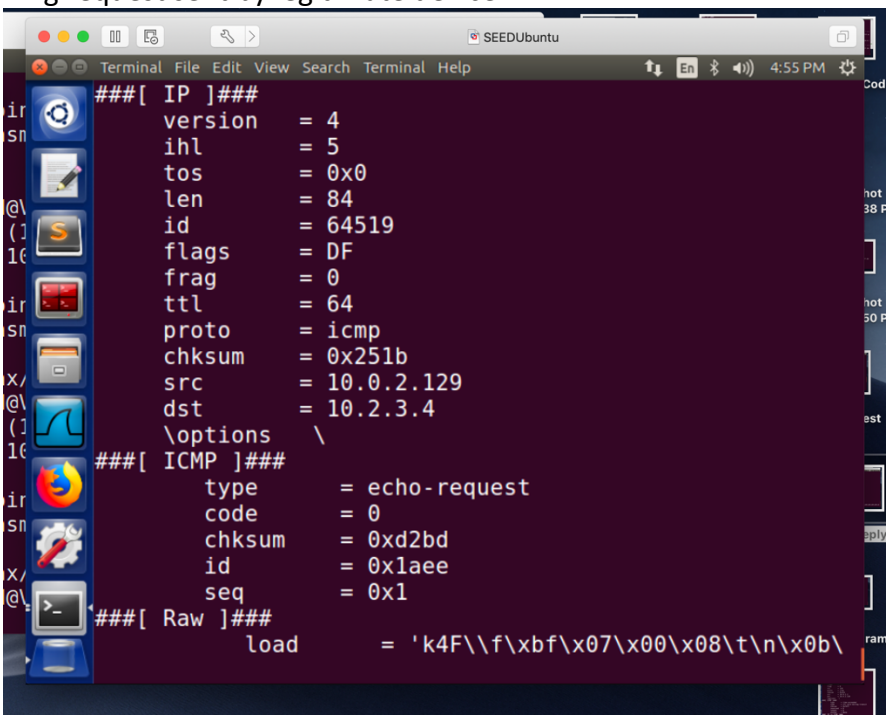
```
#!/bin/bin/python

from scapy.all import *

def print_pkt(pkt):
  if IP in pkt:
        ip_src=pkt[IP].src
        ip_dst=pkt[IP].dst
  if ICMP in pkt:
        icmp_type=pkt[ICMP].type
        #print str(icmp_type)
  if icmp_type == 8:
        a = IP()
        a.src = str(ip_dst)
        a.dst = str(ip_src)
        b = ICMP()
        b.type = 0
        b.id = pkt[ICMP].id
        b.seq = pkt[ICMP].seq
        c = pkt[Raw].load
        p = a/b/c
        send(p)
pkt = sniff(filter="icmp",prn=print_pkt)
```

This code scans the network with a filter on that only looks at ICMP packets (as denoted by the last line of code). First it takes both the IP source address and destination address and sets them both to variables. Then it scans the ICMP section of the packet and extrapolates the type from the ICMP packet and sets it to the variable 'icmp_type'. Then there is an 'if' statement that runs if the 'icmp_type' variable is equivalent to '8'. It compares it to the variable '8' because that is the code associated with the Echo-Request type. The point of this lab is to send a reply to this request message, so that the device will get a response to its ping even though it's a spoofed response. If the program sees that it is an echo request, then it runs the code that spoofs an echo reply packet.

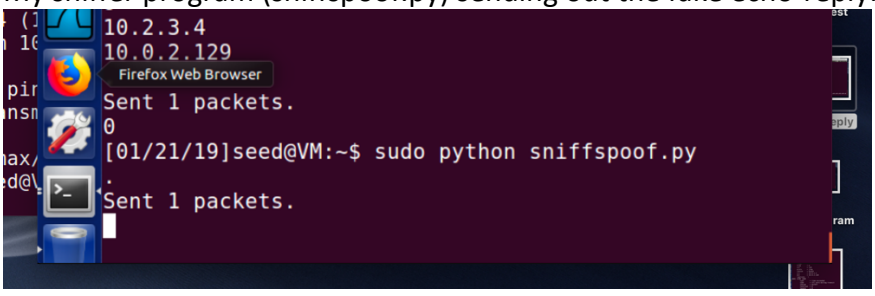
In the spoof echo-reply packet the source IP address has to be set as the destination IP the intercepted packet was trying to reach. It then sets the destination IP for the spoofed echo-reply as the source IP that was intercepted. Next it sets the ICMP type of the packet to '0'. That is the code that denotes the packet specifically as an 'echo-reply'. The ICMP ID and sequence are set to the same ID and Sequence as the intercepted echo-request packet (I noticed that when a normal ping is sent and a response is received both of these are always the same).

Lastly, the Raw data, extrapolated from the end of the echo-request packet, is attached after the ICMP section of the packet on the echo-reply. All of the ICMP and RAW data has to be there and sent back, otherwise the device that initiated the ping will not receive the message as a valid response and will get a message that they experienced 100% packet loss. If all of these aspects are attached, then the device initiating the ping will receive a valid response and register a successful ping.

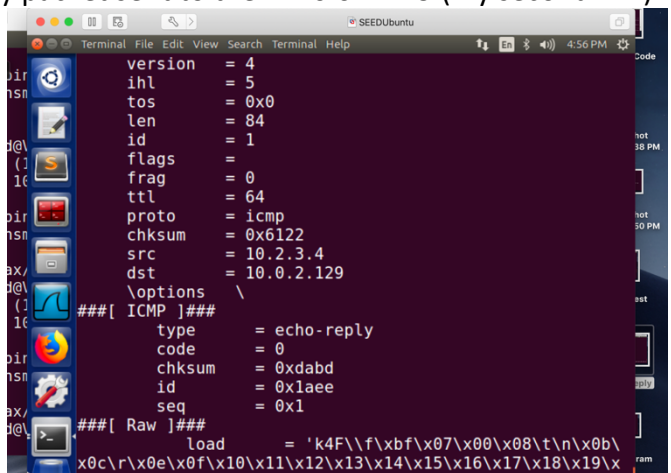
Ping request sent by legitimate device:




My sniffer program (sniffspoof.py) sending out the fake echo-reply:




The echo-reply packet sent to the IP 10.0.2.129 (my second VM):



My second VM (10.0.2.129) receiving a valid ping from 10.2.3.4: