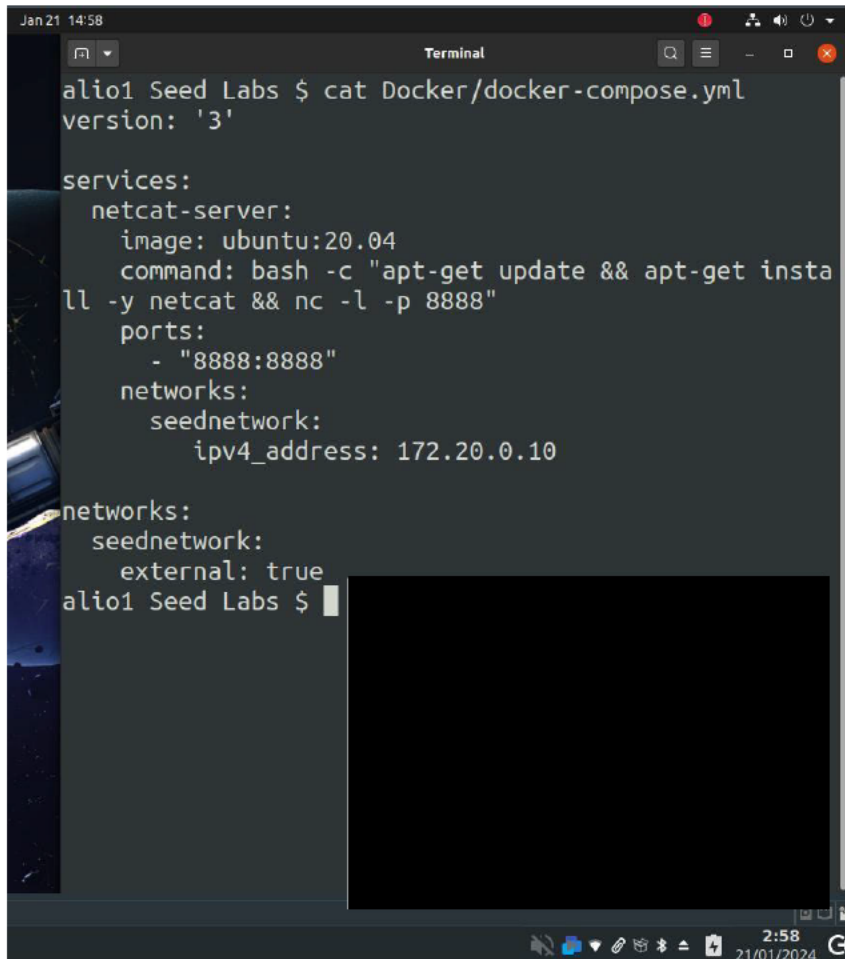


Tasks

Part 1: netcat

- A container was created using the ubuntu:20.04 image shown in the screenshot below. It installs netcat and runs the server using the nc -l command and runs on port 8080 with the addition of the -p 8080.

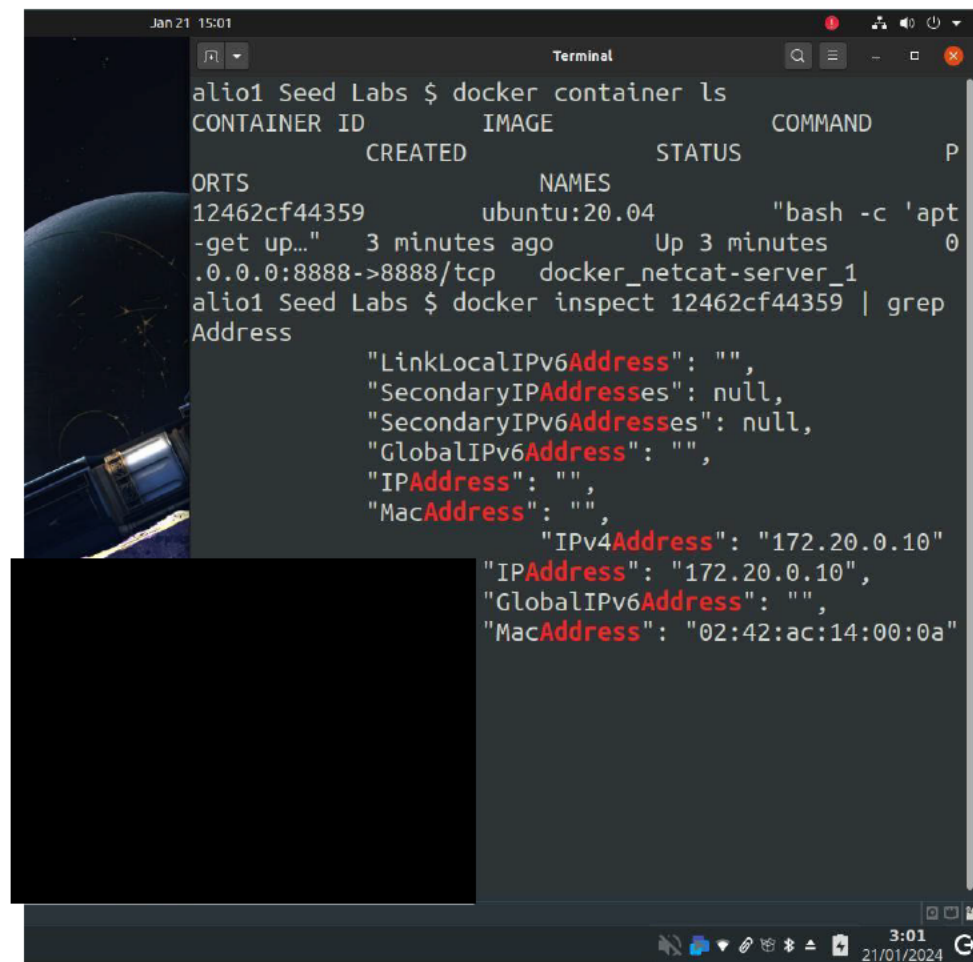
A screenshot of a terminal window titled "Terminal" with a dark background. The terminal shows the command `cat Docker/docker-compose.yml` being executed. The output is a YAML configuration for a Docker service named `netcat-server`. The configuration specifies the `image` as `ubuntu:20.04`, the `command` as `bash -c "apt-get update && apt-get install -y netcat && nc -l -p 8888"`, the `ports` as `- "8888:8888"`, and the `networks` as `seednetwork` with a static `ipv4_address` of `172.20.0.10`. The `seednetwork` is also defined as `external: true`. The terminal prompt is `alio1 Seed Labs $`. The window's title bar shows the date and time as "Jan 21 14:58". The system tray at the bottom right shows the time as "2:58" and the date as "21/01/2024".

```
Jan 21 14:58
Terminal
alio1 Seed Labs $ cat Docker/docker-compose.yml
version: '3'

services:
  netcat-server:
    image: ubuntu:20.04
    command: bash -c "apt-get update && apt-get install
ll -y netcat && nc -l -p 8888"
    ports:
      - "8888:8888"
    networks:
      seednetwork:
        ipv4_address: 172.20.0.10

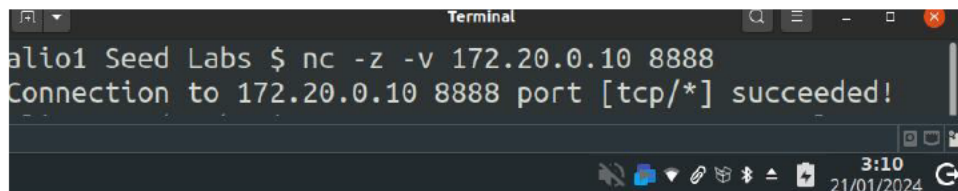
networks:
  seednetwork:
    external: true
alio1 Seed Labs $
```

- Adding this container to the seednetwork was accomplished by utilizing the YAML shown in the screenshot above. Networks: `seednetwork: ipv4_address` gives it a static IP address. The second `networks:seednetwork:external:true` confirms it is an external network.
- To confirm this, I used the command: `docker inspect 12462cf44359 | grep Address`

A terminal window titled "Terminal" with a search icon and window controls. The user is in a shell on a host named "alio1 Seed Labs". They first run "docker container ls" which shows a table of containers. One container, "12462cf44359", is running with image "ubuntu:20.04" and command "bash -c 'apt-get up...'". Then they run "docker inspect 12462cf44359 | grep Address", which outputs network configuration details for the container, including "IPv4Address": "172.20.0.10".

```
Jan 21 15:01
Terminal
alio1 Seed Labs $ docker container ls
CONTAINER ID   IMAGE     COMMAND
CREATED        STATUS    NAMES
12462cf44359   ubuntu:20.04  "bash -c 'apt-get up...'
3 minutes ago  Up 3 minutes  0
.0.0.0:8888->8888/tcp  docker_netcat-server_1
alio1 Seed Labs $ docker inspect 12462cf44359 | grep
Address
"LinkLocalIPv6Address": "",
"SecondaryIPAddresses": null,
"SecondaryIPv6Addresses": null,
"GlobalIPv6Address": "",
"IPAddress": "",
"MacAddress": "",
"IPv4Address": "172.20.0.10"
"IPAddress": "172.20.0.10",
"GlobalIPv6Address": "",
"MacAddress": "02:42:ac:14:00:0a"
```

- To demonstrate the usage of nc commands I first tested reaching the nc server using nc -z -v. The -z tests connection without connecting and -v outputs verbosly.

A terminal window titled "Terminal" showing the execution of the command "nc -z -v 172.20.0.10 8888". The output is "Connection to 172.20.0.10 8888 port [tcp/*] succeeded!".

```
Terminal
alio1 Seed Labs $ nc -z -v 172.20.0.10 8888
Connection to 172.20.0.10 8888 port [tcp/*] succeeded!
```

Part 2: Packet Sniffing and Spoofing

- Task 1.1: Sniffing packets
 - 1.1A: This task was accomplished with the python script provided in the book; the only modification was the interface name. See the screenshot below for ICMP / ping examples on the right terminal

The screenshot shows a Linux desktop with two terminal windows open. The left terminal window displays the output of a ping command to 10.9.0.5, showing packet statistics and round-trip times. The right terminal window shows the output of a netstat command, displaying network statistics for various protocols including ICMP, RAW, and Ethernet. The desktop background is a dark, abstract image with a yellow and blue light streak.

Terminal 1 (Left):

```

all01 Seed Labs $ ping 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=0.178 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=64 time=0.167 ms
^C
--- 10.9.0.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 0.118/0.147/0.167/0.019 ms
all01 Seed Labs $
  
```

Terminal 2 (Right):

```

all01 Seed Labs $ sudo ./sniffer.py
###[ Ethernet ]###
dst      = 01:42:0a:09:00:05
src      = 01:42:c1:84:e2:d5
type     = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 84
id       = 46890
flags    = DF
frag     = 0
ttl      = 64
proto    = icmp
chksum   = 0x0f07
src      = 10.9.0.1
dst      = 10.9.0.5
loptions \
###[ ICMP ]###
type     = echo-request
code     = 0
chksum   = 0xace2
id       = 0x7
seq      = 0x1
###[ RAW ]###
load     = '\x1a\x00\xde\x00\x00\x00\xbd\\\x07\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f \t\n\r /()*+,-./01234567'
###[ Ethernet ]###
dst      = 01:42:c1:84:e2:d5
src      = 01:42:0a:09:00:05
type     = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 16434
id       = 16434
flags    =
frag     = 0
ttl      = 64
proto    = icmp
chksum   = 0x2a60
src      = 10.9.0.5
dst      = 10.9.0.1
loptions \
  
```

The second screenshot shows the output of running the script without sudo. The errors that were raised occur because tools like Wireshark and our packet sniffer utilize sockets which require superuser privileges/permission

```
alio1 Seed Labs $ ./sniffer.py
Traceback (most recent call last):
  File "./sniffer.py", line 7, in <module>
    pkt = sniff(iface='br-935f87722f66', filter='icmp', prn=print_pkt)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 1036, in sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.8/dist-packages/scapy/sendrecv.py", line 906, in _run
    sniff_sockets[L2socket(type=ETH_P_ALL, iface=iface,
  File "/usr/local/lib/python3.8/dist-packages/scapy/arch/linux.py", line 398, in __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(type)) # noqa: E501
  File "/usr/lib/python3.8/socket.py", line 231, in __init__
    _socket._init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
alio1 Seed Labs $
```

- 1.1B: For this task, I modified the code to create three functions, one for each filter. Therefore, the code in the subsequent screenshot iterates through the packets, and sets key variables like `target_ip` and `target_port` for the second filter to meet the criteria and print using the built-in `packet.summary()` feature.

```

alioi Seed Labs $ sudo ./sniffer.py
Ether / IP / ICMP 172.16.187.170 > 1.1.1.1 echo-request 0 / Raw
Ether / IP / ICMP 1.1.1.1 > 172.16.187.170 echo-reply 0 / Raw
Ether / IP / ICMP 172.16.187.170 > 1.1.1.1 echo-request 0 / Raw
Ether / IP / ICMP 1.1.1.1 > 172.16.187.170 echo-reply 0 / Raw
Ether / IP / ICMP 172.16.187.170 > 1.1.1.1 echo-request 0 / Raw
Ether / IP / ICMP 1.1.1.1 > 172.16.187.170 echo-reply 0 / Raw
Ether / IP / ICMP 172.16.187.170 > 1.1.1.1 echo-request 0 / Raw
Ether / IP / ICMP 1.1.1.1 > 172.16.187.170 echo-reply 0 / Raw
Ether / IP / ICMP 172.16.187.170 > 1.1.1.1 echo-request 0 / Raw
Ether / IP / ICMP 1.1.1.1 > 172.16.187.170 echo-reply 0 / Raw

64 bytes from 1.1.1.1: icmp_seq=3 ttl=128 time=15.3 ms
64 bytes from 1.1.1.1: icmp_seq=4 ttl=128 time=17.2 ms
64 bytes from 1.1.1.1: icmp_seq=5 ttl=128 time=19.2 ms
^C
--- 1.1.1.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4000ms
rtt min/avg/max/mdev = 11.487/28.043/29.048/4.970 ms
alioi Seed Labs $ print sniffer.py
Error: no 'print' method found for type 'text/x-python'
alioi Seed Labs $ cat sniffer.py
#!/usr/bin/env python
from scapy.all import *

def print_pkt(pkt):
    # pkt.show()

    pkt = sniff(lf='br-93f8772f0e', filter='icmp', prn=print_pkt)

def capture_icmp_packets(packet):
    if ICMP in packet:
        print(packet.summary())

def capture_tcp_from_ip_and_port(packet, target_ip, target_port):
    if IP in packet and TCP in packet:
        if packet[IP].src == target_ip and packet[TCP].dport == target_port:
            print(packet.summary())

def capture_subnet_packets(packet, subnet):
    if IP in packet:
        src_ip = packet[IP].src
        dst_ip = packet[IP].dst
        if src_ip.startswith(subnet) or dst_ip.startswith(subnet):
            print(packet.summary())

sniff(filter='icmp', prn=capture_icmp_packets)

target_ip = 'your_target_ip'
target_port = 23
sniff(filter='ip and tcp and src {target_ip} and dst port {target_port}', prn=capture_tcp_from_ip_and_port)

subnet = '10.0.0.'
sniff(filter='net {subnet}/24', prn=capture_subnet_packets)

alioi Seed Labs $

```

- Task 1.2: Spoofing ICMP packets
 - For this task, we sent a packet using Scapy. The only modifications needed to the code were adding the IP address. The screenshot below shows the successful sending of the ICMP request. As this is only one request, it is not captured by Wireshark, there would need to be more packets send to register

```

>>> from scapy.all import *
>>> a.dst = '172.16.187.170'
>>> b = ICMP()
>>> p = a/b
>>> send(p)
.
Sent 1 packets.
>>> a
<IP dst=172.16.187.170 |>
>>> b
<ICMP |>
>>> p
<IP frag=0 proto=icmp dst=172.16.187.170 |<ICMP |>>
>>> send(p)
.
Sent 1 packets.
>>>

```

- Task 1.3: Traceroute

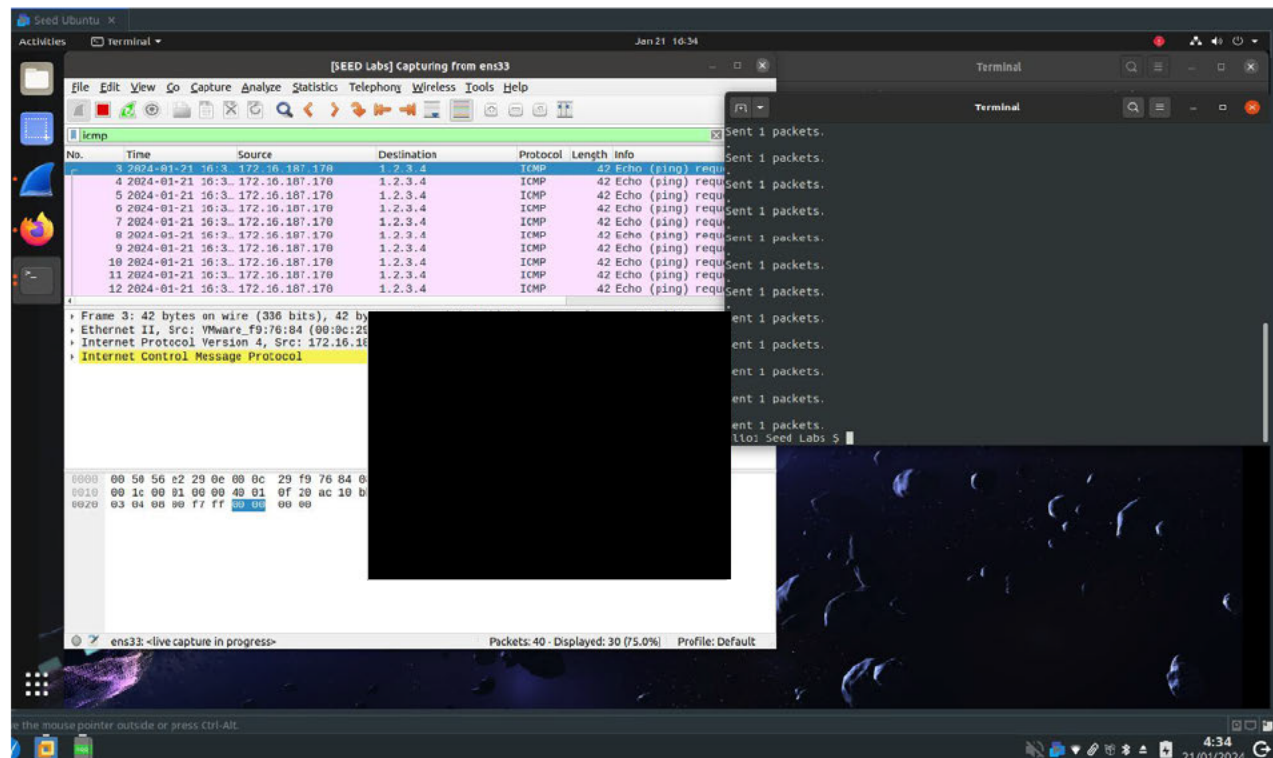
- For this step, I created a program to automate the process so that I did not have to increment TTL. I chose the typical default value for TTL in traceroute (being 30). In the first screenshot, I show the code, the second featured the output of `sudo ./traceroute`

```
Sent 1 packets.
alio1 Seed Labs $ cat traecroute
#!/usr/bin/env python3
from scapy.all import *

a = IP()
a.dst = '1.2.3.4'
p = ICMP()
ttl = 0
while ttl < 30:
    ttl += 1
    send(a/b)

alio1 Seed Labs $
```

- For my code, I followed the code in the book as an outline, however, when it came to setting TTL, I set it to zero and the run a while loop that would iterate through TTL0 – TTL 29 to ensure that the traceroute would be seen by WireShark.



- Task 1.4: Sniffing and-then Spoofing

- To start this section, I had to research and found that it was a bit outside the scope of my abilities. So, I researched the question and found a great resource that goes through the problem. I have included the link [here](#).
- What the code must do is identify ARP packets (which I believe the reason for `pkt[#].type == 8`) to start the Request and Reply. The subsequent section of code sets parameters to facilitate the REQUEST and REPLY section. By this, we set a source and destination, as well as ID, Sequence, and Load to fill the packet with necessary information. We can see the code working through print strings that show which step it is on.
 - We can make modifications to this code to return if it detects packets other than ICMP to expedite the process.