

SDN-BASED STATELESS FIREWALL

Student Name: Hemendu Roy

Email: hroy6@asu.edu

Submission Date: 03-05-2022

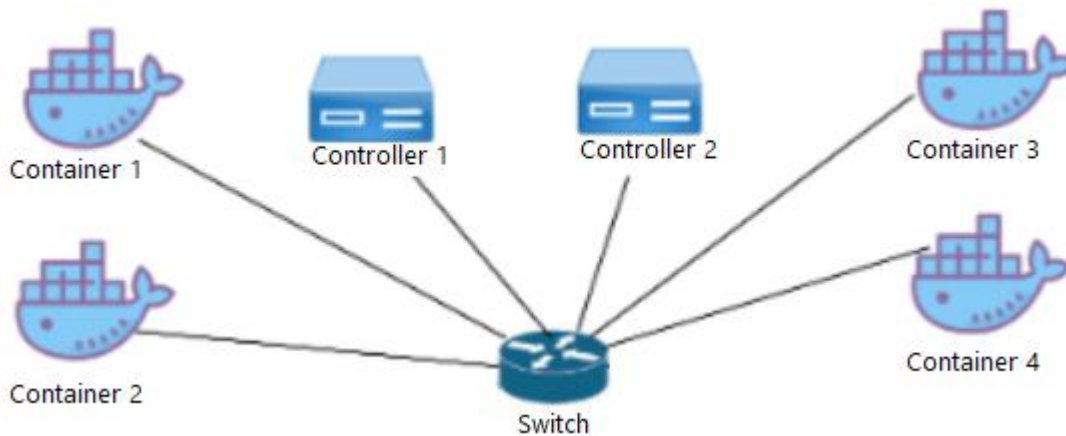
Class Name and Term: CSE 548 Spring 2022

I. PROJECT OVERVIEW

In this project, we're using mininet and containernet to develop a Stateless Firewall based on a Software Defined Network architecture. The firewall in question is based on OpenFlow with flow-based policies to accept, propagate or drop packets wherein the first packet is inspected against policies and the rest of the data stream is dealt with only subsequently.

II. NETWORK SETUP

Network Topology and Configurations



In this setup, we have set up a mininet environment in containernet with 4 containernet hosts, one OVS switch and two remote controllers as shown in the figure above.

Initial Reachability

Initially, the assigned addresses of each host are as follows.

Container Host	Layer 2 address	Layer 3 address
h1	00:00:00:00:00:01	10.0.0.1
h2	00:00:00:00:00:02	10.0.0.2
h3	00:00:00:00:00:03	10.0.0.3
h4	00:00:00:00:00:04	10.0.0.4

III. SOFTWARE

Open vSwitch – [1] Open vSwitch is a production quality, multilayer virtual switch licensed under the open source Apache 2.0 license. It is designed to enable massive network automation through programmatic extension, while still supporting standard management interfaces and protocols.

tcpdump – [2] tcpdump is used to capture network traffic on a network interface that match a set of Boolean expressions

Mininet – [3] Mininet creates a realistic virtual network, running real kernel, switch and application code, on a single machine (VM, cloud or native), in seconds, with a single command.

Containernet – [4] Containernet is a fork of the famous Mininet network emulator and allows to use Docker containers as hosts in emulated network topologies.

IV. PROJECT DESCRIPTION

In this section, detailed descriptions of the project tasks will be illustrated.

Before developing the firewall, we first make sure our working environment is setup correctly.

Checking python installations.

```
root@ubuntu:/home/ubuntu/pox# python --version
Python 2.7.17
root@ubuntu:/home/ubuntu/pox# python3 --version
Python 3.6.9
```

Checking the mininet installation.

```
root@ubuntu:/home/ubuntu/pox# mn --version
2.3.0d5
root@ubuntu:/home/ubuntu/pox# mn --test pingall
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Waiting for switches to connect
s1
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
*** Stopping 1 controllers
c0
*** Stopping 2 links
..
*** Stopping 1 switches
s1
*** Stopping 2 hosts
h1 h2
*** Done
completed in 5.455 seconds
```

Checking the POX installation.

```
root@ubuntu:/home/ubuntu/pox# ./pox.py -verbose forwarding.hub
POX 0.5.0 (eel) / Copyright 2011-2014 James McCauley, et al.
INFO:forwarding.hub:Proactive hub running.
DEBUG:core:POX 0.5.0 (eel) going up...
DEBUG:core:Running on CPython (2.7.17/Feb 27 2021 15:10:58)
DEBUG:core:Platform is Linux-5.3.0-53-generic-x86_64-with-Ubuntu-18.04-bionic
INFO:core:POX 0.5.0 (eel) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
^CINFO:core:Going down...
INFO:core:Down.
```

Lastly, we check if OVS is installed correctly.

```
root@ubuntu:/home/ubuntu/pox# ovs-vsctl --version
ovs-vsctl (Open vSwitch) 2.17.90
DB Schema 8.3.0
```

Now that we have verified that our environment is setup correctly, we can begin modifying the firewall rules.

The rules for Layer 2 can be found in `/home/ubuntu/pox/l2firewall.config`

The rules for Layer 3 can be found in `/home/ubuntu/pox/l3firewall.config`

Assessments

1. 5 points) Create a mininet based topology with 4 container hosts and one controller switches and run it.

We create the network using the following command

```
sudo mn --topo=single,4 --controller=remote,port=6633 --controller=remote,port=6655 --switch=ovsk --mac
```

```
root@ubuntu:/home/ubuntu# mn --topo=single,4 --controller=remote,port=6633 --controller=remote,port=6655 --switch=ovsk --mac
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0 c1
*** Starting 1 switches
s1 ...
*** Starting CLI:
containernet>
```

Once created, a containernet CLI prompt appears which can now use to work with our host machines.

Now, to verify that our containernet hosts are up and running, we can use `xterm`.

```
File Edit View Search Terminal Tabs Help
root@ubuntu:/home/ubuntu/pox x root@ubuntu:/home/ubuntu x root@ubuntu:/home/ubuntu/pox x
containernet> xterm h1
containernet> xterm h1
containernet> xterm h2
containernet> xterm h2
containernet> xterm h4
containernet> xterm h2
containernet> xterm h1
containernet> xterm h3
containernet>
Interrupt
containernet>
*** Stopping 2 controllers
c0 c1
*** Stopping 8 terms
*** Stopping 4 links
....
*** Stopping 1 switches
s1
*** Stopping 4 hosts
h1 h2 h3 h4
*** Done
completed in 4126.005 seconds
root@ubuntu:/home/ubuntu# mn --topo=single,4 --con3
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0 c1
*** Starting 1 switches
s1 ...
*** Starting CLI:
containernet> xterm h1 h2 h3 h4
containernet>
```

Moreover, we can use *ifconfig* to observe the assigned MAC addresses and IPs of each host as shown above.

To verify connectivity between the hosts, we can use `tcpdump`.

```
"Node: h1"
RX packets 30 bytes 3468 (3.4 KB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 3 bytes 310 (310.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
inet 127.0.0.1 netmask 255.0.0.0
inet6 ::1 prefixlen 128 scopeid 0x10<host>
loop txqueuelen 1000 (Local Loopback)
RX packets 0 bytes 0 (0.0 B)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 0 bytes 0 (0.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@ubuntu:/home/ubuntu# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=27.3 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.506 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.122 ms
^C
--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2024ms
rtt min/avg/max/mdev = 0.122/9.340/27.392/12.765 ms
root@ubuntu:/home/ubuntu#

"Node: h2"
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@ubuntu:/home/ubuntu# tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h2-eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
23:28:11.073805 ARP, Request who-has ubuntu tell 10.0.0.1, length 28
23:28:11.073819 ARP, Reply ubuntu is-at 00:00:00:00:00:02 (oui Ethernet), length
28
23:28:11.076586 IP 10.0.0.1 > ubuntu: ICMP echo request, id 17233, seq 1, length
64
23:28:11.076603 IP ubuntu > 10.0.0.1: ICMP echo reply, id 17233, seq 1, length 6
4
23:28:12.066695 IP 10.0.0.1 > ubuntu: ICMP echo request, id 17233, seq 2, length
64
23:28:12.066759 IP ubuntu > 10.0.0.1: ICMP echo reply, id 17233, seq 2, length 6
4
23:28:13.074332 IP 10.0.0.1 > ubuntu: ICMP echo request, id 17233, seq 3, length
64
23:28:13.074391 IP ubuntu > 10.0.0.1: ICMP echo reply, id 17233, seq 3, length 6
4
23:28:16.084313 ARP, Request who-has 10.0.0.1 tell ubuntu, length 28
23:28:16.084440 ARP, Reply 10.0.0.1 is-at 00:00:00:00:00:01 (oui Ethernet), leng
th 28
```

Now, our mininet environment is working correctly.

2. (5 points) Make the interfaces up and assign IP addresses to interfaces of container hosts.

To assign IP addresses of our choice, we can use the following set of commands.

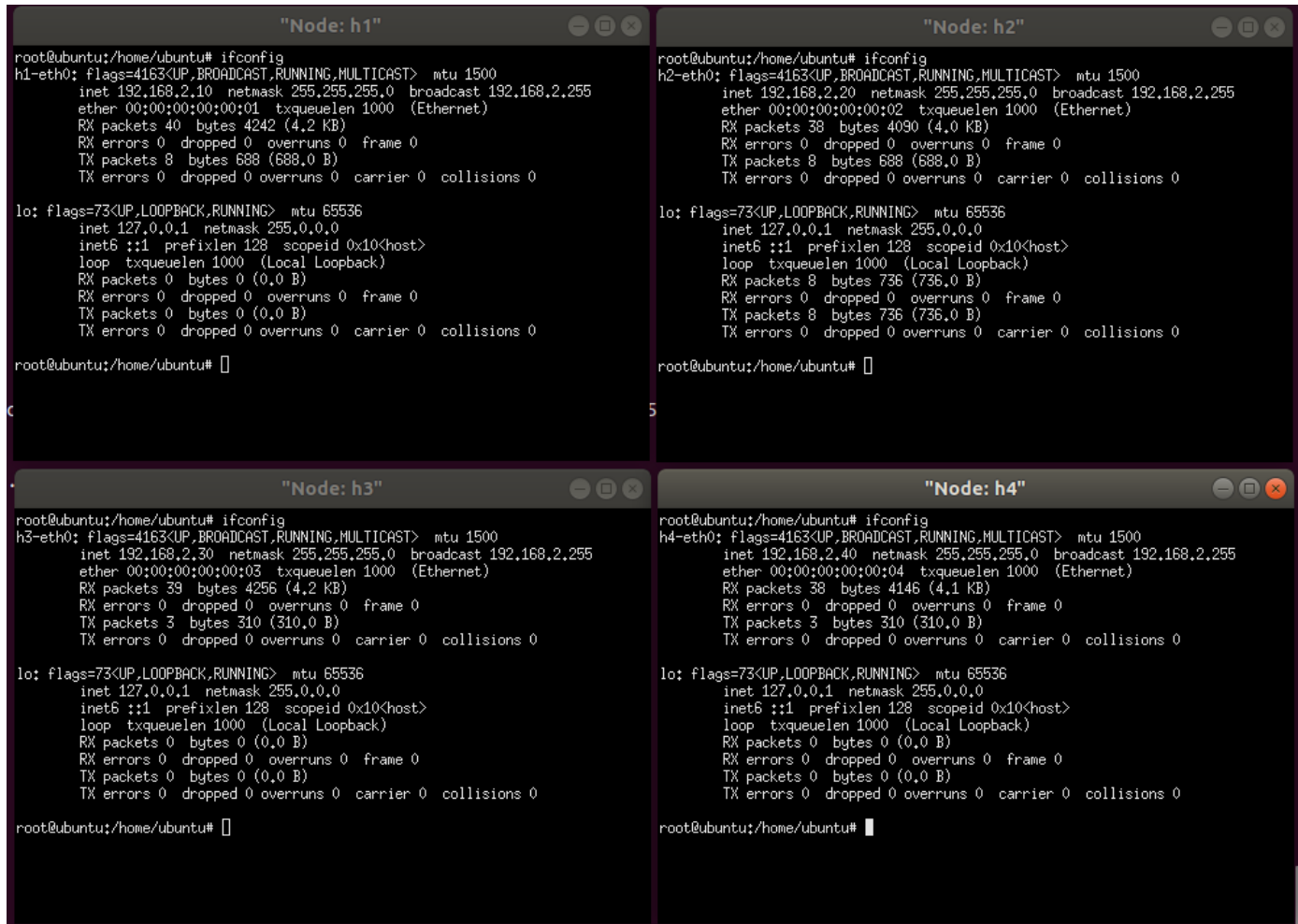
```
h1 ifconfig h1-eth0 192.168.2.10
h2 ifconfig h2-eth0 192.168.2.20
h3 ifconfig h3-eth0 192.168.2.30
h4 ifconfig h4-eth0 192.168.2.40
```

```

containernet> h1 ifconfig h1-eth0 192.168.2.10
containernet> h2 ifconfig h2-eth0 192.168.2.20
containernet> h3 ifconfig h3-eth0 192.168.2.30
containernet> h4 ifconfig h4-eth0 192.168.2.40
containernet>

```

Once again, we use *ifconfig* to verify the new IPs of the hosts.



The image displays four terminal windows, each showing the output of the `ifconfig` command for a specific node. The windows are titled "Node: h1", "Node: h2", "Node: h3", and "Node: h4". Each window shows the configuration for the `h1-eth0` (or `h2-eth0`, `h3-eth0`, `h4-eth0`) and `lo` interfaces. The `h1-eth0` interface is configured with IP 192.168.2.10, netmask 255.255.255.0, and broadcast 192.168.2.255. The `h2-eth0` interface is configured with IP 192.168.2.20, netmask 255.255.255.0, and broadcast 192.168.2.255. The `h3-eth0` interface is configured with IP 192.168.2.30, netmask 255.255.255.0, and broadcast 192.168.2.255. The `h4-eth0` interface is configured with IP 192.168.2.40, netmask 255.255.255.0, and broadcast 192.168.2.255. The `lo` interface is configured with IP 127.0.0.1, netmask 255.0.0.0, and broadcast 127.0.0.1. The output also shows statistics for RX and TX packets, bytes, errors, dropped, overruns, carrier, and collisions.

Assessments 3-8 – Adding Layer 2 and Layer 3 firewall rules

Layer 2 rule i.e., Assessment 6

```

root@ubuntu:/home/ubuntu/pox# cat l2firewall.config
id,mac_0,mac_1
1,00:00:00:00:00:02,00:00:00:00:00:04

```

This instructs the firewall to block traffic originating from the host with MAC address 00:00:00:00:00:02 to that with MAC address 00:00:00:00:00:04.

Layer 3 rules i.e., Assessments 3,4,5,7 and 8

```
root@ubuntu:/home/ubuntu/pox# cat l3firewall.config
priority,src_mac,dst_mac,src_ip,dst_ip,src_port,dst_port,nw_proto
1,any,any,192.168.2.10,192.168.2.30,1,1,icmp
2,any,any,192.168.2.20,192.168.2.40,1,1,icmp
3,any,any,192.168.2.20,any,any,80,tcp
4,any,any,192.168.2.10,192.168.2.20,any,any,tcp
5,any,any,192.168.2.10,192.168.2.20,any,any,udp
```

This instructs the firewall to,

3. block ICMP traffic from source IP 192.168.2.10 and destination IP 192.168.2.30.
4. block ICMP traffic from source IP 192.168.2.20 and destination IP 192.168.2.40.
5. block HTTP traffic from source IP 192.168.2.20.
6. block TCP traffic from 192.168.2.10 to 192.168.2.20.
7. block UDP traffic from 192.168.2.10 to 192.168.2.20.

Now we verify if our rules are working.

```
"Node: h1"
root@ubuntu:/home/ubuntu# ping 192.168.2.30
PING 192.168.2.30 (192.168.2.30) 56(84) bytes of data:
0

"Node: h3"
root@ubuntu:/home/ubuntu# tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h3-eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
```

```
"Node: h1"
root@ubuntu:/home/ubuntu# ping 192.168.2.30
PING 192.168.2.30 (192.168.2.30) 56(84) bytes of data.
^C
--- 192.168.2.30 ping statistics ---
85 packets transmitted, 0 received, 100% packet loss, time 86178ms
root@ubuntu:/home/ubuntu#
```

As we can see, ICMP traffic from host 1 to host 3 is blocked. The pings do not succeed on host 1 and host 3 also shows no captured traffic on tcpdump.

Similarly, ICMP traffic from host 2 to host 4 is also blocked.

The image shows two terminal windows. The top window, titled "Node: h2", shows a successful ping command: `root@ubuntu:/home/ubuntu# ping 192.168.2.40` followed by `PING 192.168.2.40 (192.168.2.40) 56(84) bytes of data.` The bottom window, titled "Node: h4", shows the output of `tcpdump` listening on `h4-eth0`, capturing an ARP request and reply. An "Activate Windows" watermark is visible in the bottom right corner of the terminal area.

```

"Node: h2"
root@ubuntu:/home/ubuntu# ping 192.168.2.40
PING 192.168.2.40 (192.168.2.40) 56(84) bytes of data.

"Node: h4"
root@ubuntu:/home/ubuntu# tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h4-eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
23:54:31.039678 ARP, Request who-has ubuntu tell 192.168.2.20, length 28
23:54:31.039697 ARP, Reply ubuntu is-at 00:00:00:00:00:04 (oui Ethernet), length 28

```

The image shows a terminal window titled "Node: h2" where a ping command to 192.168.2.40 fails. The output shows "37 packets transmitted, 0 received, 100% packet loss, time 36973ms".

```

"Node: h2"
root@ubuntu:/home/ubuntu# ping 192.168.2.40
PING 192.168.2.40 (192.168.2.40) 56(84) bytes of data.
^C
--- 192.168.2.40 ping statistics ---
37 packets transmitted, 0 received, 100% packet loss, time 36973ms
root@ubuntu:/home/ubuntu#

```

Finally, we confirm that host 1 can access host 2 and also that host 3 can access host 4. This tells us that it is indeed the firewall blocking certain traffic and the network is setup correctly.

The image displays four terminal windows from a Kali Linux desktop, each showing network traffic analysis results. The windows are titled "Node: h2", "Node: h3", "Node: h4", and "Node: h2".

- Top-left window (Node: h2):** Shows ping statistics for 192.168.2.20. It reports 85 packets transmitted, 0 received, and 100% packet loss. It also shows a detailed packet capture for an ICMP Echo (ping) request from 192.168.2.20 to 192.168.2.20.
- Top-right window (Node: h2):** Shows ping statistics for 192.168.2.40. It reports 37 packets transmitted, 0 received, and 100% packet loss. It also shows a detailed packet capture for an ICMP Echo (ping) request from 192.168.2.40 to 192.168.2.20.
- Bottom-left window (Node: h3):** Shows ping statistics for 192.168.2.40. It reports 4 packets transmitted, 4 received, and 0% packet loss. It also shows a detailed packet capture for an ICMP Echo (ping) request from 192.168.2.40 to 192.168.2.20.
- Bottom-right window (Node: h4):** Shows ping statistics for 192.168.2.40. It reports 12 packets transmitted, 12 received, and 0% packet loss. It also shows a detailed packet capture for an ICMP Echo (ping) request from 192.168.2.40 to 192.168.2.20.

To verify if HTTP traffic from host 2 is blocked, we can start a rudimentary HTTP server on host 1 and then use *curl* to access it from the other hosts.

The image displays four terminal windows, each representing a different node (h1, h2, h3, h4) in a network. Each window shows the execution of a curl command to download a list of files from a web server. The files are being saved to the /home/ubuntu/ directory.

- Node: h1:** Shows the execution of a curl command to download a list of files from a web server. The files are being saved to the /home/ubuntu/ directory.
- Node: h2:** Shows the execution of a curl command to download a list of files from a web server. The files are being saved to the /home/ubuntu/ directory.
- Node: h3:** Shows the execution of a curl command to download a list of files from a web server. The files are being saved to the /home/ubuntu/ directory.
- Node: h4:** Shows the execution of a curl command to download a list of files from a web server. The files are being saved to the /home/ubuntu/ directory.

As seen in the above figure, hosts 3 and 4 can access the webpage. A 200 HTTP response is also seen on host 1. However, on host 2, nothing is returned, Therefore we have successfully blocked HTTP traffic from host 2 using our firewall.

After waiting for a while, the curl times out on host 2.

hosts 1 and 3 show that port 80 is being served whereas host 2 does not.\

To test if TCP traffic from host 1 to 2 is blocked,

The screenshot displays three terminal windows with the following content:

```

"Node: h1"
root@ubuntu:/home/ubuntu# nc -v 192.168.2.20 7777
this is host 1
nc: connect to 192.168.2.20 port 7777 (tcp) failed: Connection timed out
root@ubuntu:/home/ubuntu# this is host 1

Command 'this' not found, did you mean:
  command 'thin' from deb thin

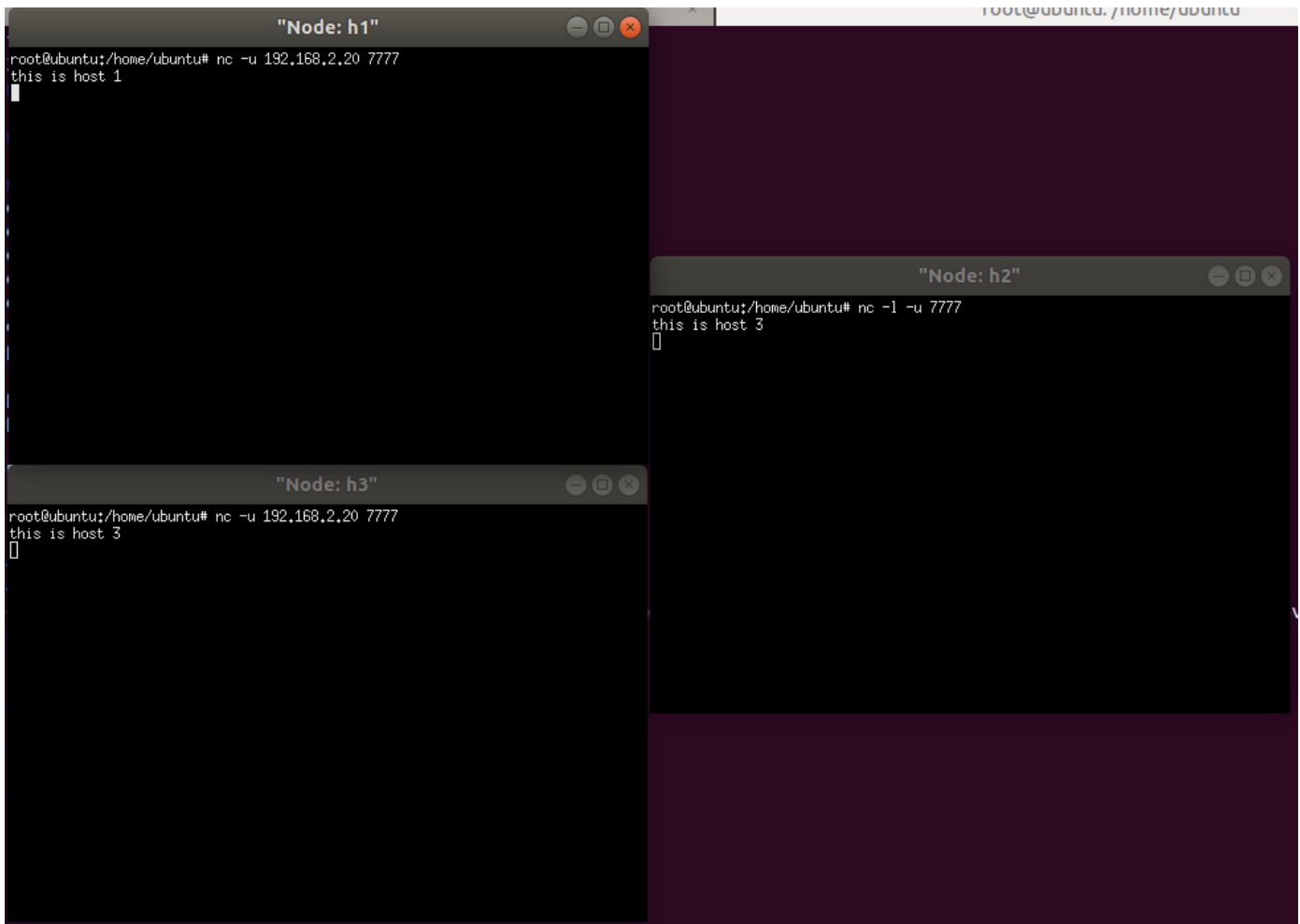
Try: apt install <deb name>
root@ubuntu:/home/ubuntu#

"Node: h2"
root@ubuntu:/home/ubuntu# nc -l -p 7777
this is host 3
[]

"Node: h3"
root@ubuntu:/home/ubuntu# nc 192.168.2.20 7777
this is host 3
[]
  
```

We see than host 3 is able to set TCP packets to host 2 but the same cannot be done from host 1, the connection times out.

To test if UDP traffic from host 1 to 2 is blocked,



We can see that UDP packets sent from host 3 to host 2 are received but those from host 1 to host 2 are dropped.

V. CONCLUSION

I learnt about what the mininet and containernet tools do and how to use them.

More importantly, I understood the theory behind a flow based firewall and how to setup and configure one.

I also learnt how to operate the Open vSwitch tool using its CLI commands.

VI. APPENDIX B: ATTACHED FILES

Demo Video - <https://www.youtube.com/watch?v=GY7Ts9JUBr8>

Layer 2 Firewall rules

```
id,mac_0,mac_1
1,00:00:00:00:00:02,00:00:00:00:00:04
```

Layer 3 Firewall rules

```
priority,src_mac,dst_mac,src_ip,dst_ip,src_port,dst_port,nw_proto
1,any,any,192.168.2.10,192.168.2.30,1,1,icmp
2,any,any,192.168.2.20,192.168.2.40,1,1,icmp
3,any,any,192.168.2.20,any,any,80,tcp
4,any,any,192.168.2.10,192.168.2.20,any,any,tcp
```

5,any,any,192.168.2.10,192.168.2.20, any, any,udp

VII. REFERENCES

- [1] Open vSwitch - <https://www.openvswitch.org/>
- [2] tcpdump Linux man page - <https://linux.die.net/man/8/tcpdump>
- [3] Mininet - <http://mininet.org/>
- [4] Containernet - <https://containernet.github.io/>