Project Report 2 – SDN-Based Stateless Firewall

Student Name: Marco Ermini

Email: mermini@asu.edu

Submission Date: 14th Jun, 2021

Class Name and Term: CSE548 Summer 2021

I. Project Overview

In this lab we are exploring how to set up a software defined environment based on mininet and containernet. We also get to practice how to set up an OpenFlow based flow-level firewall on SDN. Finally, we need to set up and practice flow-based firewall filtering policies such as enabling the ability to accept, drop, or reject the incoming flows thus ensuring the safety of the system from malicious attacking network traffic.

All the files and configurations used for this lab have been uploaded on GitHub; references are provided throughout the text and in the Appendix A at the bottom of the File.

II. Network Setup

Since I have experienced some issues with connecting the VM to my lab through my host PC running Windows 10, I have chosen to set up the VM in VirtualBox in a bridged network configuration.

In this way, I could avoid configuring a static IP address to the VM and simply opted to fetch an IP for the VM via DHCP; this IP is then assigned directly to my router, which can be useful to troubleshoot eventual connectivity issues. On the Ubuntu/Linux side this has brought no issues whatsoever, once the configuration commands are adjusted (e.g. use "dhclient br0" rather than assigning an IP address with "ifconfig br0").

Because of the use of DHCP, depending on the lab run and when I restarted the VM, it may have assumed a different address in the 192.168.1/24 network. This does not affect the outcome of the lab exercises, but it may look inconsistent in the screenshots. Apologies about that.

Please find below the initial set-up of the virtual infrastructure as I have configured it in VirtualBox.

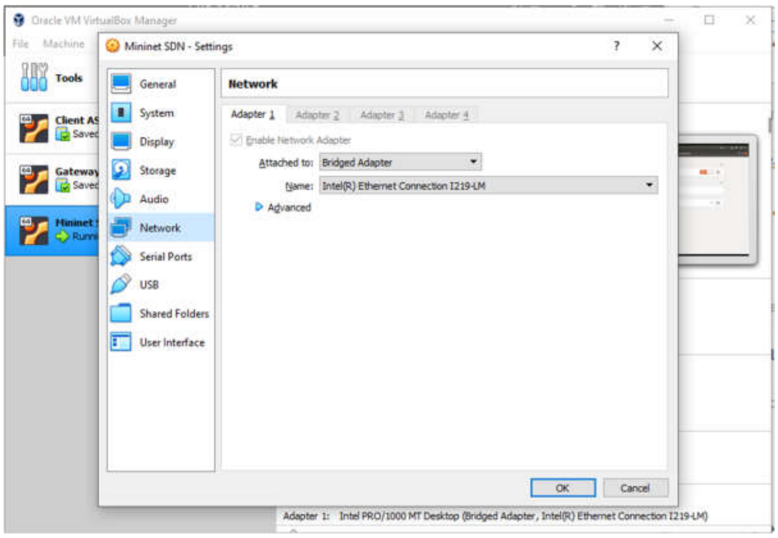


Figure 1 - Bridged network setup in VirtualBox

III. SOFTWARE

For this first lab, the following software has been used:

- Various network tools (specifically, tcpdump, ping, traceroute, hping3, and nc – netcat)
- POX (GitHub link: https://noxrepo.github.io/pox-doc/html/
- Open vSwitch: http://www.openvswitch.org/
- Open vSwitch Cheat Sheet: https://therandomsecurityguy.com/openvswitch-cheatsheet/
- Containernet: https://containernet.github.io/
- Containernet tutorial: https://github.com/containernet/containernet/wiki/Tutorial:-Getting-Started
-

IV. PROJECT DESCRIPTION

In this assignment, I have executed the various labs steps and obtained the proofs that the assignments have been completed.

A. Lab "lab-cs-net-00006 OpenVirtual Switch"

As mentioned in the "Network Setup" section, the interesting part of this lab happens because a bridge is required, and I am using DHCP. The following screenshots illustrate how DHCP is used on the bridge and the MAC address of the VM external interface is assigned to the bridge.

```
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.143  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::853e:cc54:13a3:86a8  prefixlen 64  scopeid 0x20<link>
        inet6 fe80::cd42:eba6:2e84:8cf  prefixlen 64  scopeid 0x20<link>
        inet6 fd7c:4c9f:36ca:4:57c:1cb8:7746:5562  prefixlen 64  scopeid 0x0<global>
        inet6 2a01:c23:b82c:d704:84f4:9343:3fd7:ae42  prefixlen 64  scopeid 0x0<global>
        inet6 2a01:c23:b82c:d704:c10c:4065:f2f0:ee8d  prefixlen 64  scopeid 0x0<global>
        inet6 fd59:3177:8a0a:1:c0f3:3620:f246:7ed8  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::e0a5:a4c4:9f32:74e7  prefixlen 64  scopeid 0x20<link>
        inet6 2a01:c23:b82c:d701:eb5a:73c9:d0c2:f78b  prefixlen 64  scopeid 0x0<global>
        ether 08:00:27:38:8a:7b  txqueuelen 1000  (Ethernet)
        RX packets 2400  bytes 2018074 (2.0 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1489  bytes 154252 (154.2 KB)
        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 323  bytes 29513 (29.5 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 323  bytes 29513 (29.5 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@ubuntu:~# ifconfig enp0s3 0
root@ubuntu:~# dhclient br0
```

Figure 2 - On the created bridge br0, dhclient is run

```
root@ubuntu:~# ifconfig br0
br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.143  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 2a01:c23:b82c:d701:a00:27ff:fe38:8a7b  prefixlen 64  scopeid 0x0<global>
        inet6 fd59:3177:8a0a:1:60c5:4ad8:f4a8:201f  prefixlen 64  scopeid 0x0<global>
        inet6 2a01:c23:b82c:d701:60c5:4ad8:f4a8:201f  prefixlen 64  scopeid 0x0<global>
        inet6 fd59:3177:8a0a:1:a00:27ff:fe38:8a7b  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::c0f0:ddff:fe41:1941  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:38:8a:7b  txqueuelen 1000  (Ethernet)
        RX packets 434  bytes 76406 (76.4 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 146  bytes 22239 (22.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 3 - The bridge inherits the same IP of the enp0s3 interface

The bridge obtains the same IP from the router because it uses the same MAC address of the interface, and therefore the route is just reusing a DHCP lease already assigned.

```
root@ubuntu:~# ping 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=2 ttl=118 time=164 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=117 time=1991 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=117 time=833 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=118 time=15.5 ms
^C
--- 8.8.8.8 ping statistics ---
5 packets transmitted, 4 received, 20% packet loss, time 4033ms
rtt min/avg/max/mdev = 15.558/751.109/1991.123/779.396 ms, pipe 2
root@ubuntu:~#
root@ubuntu:~# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         _gateway        0.0.0.0         UG    0      0        0 br0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 br0
root@ubuntu:~# ping google.com
PING google.com(fra07s30-in-x200e.1e100.net (2a00:1450:4001:803::200e)) 56 data bytes
64 bytes from fra24s02-in-x0e.1e100.net (2a00:1450:4001:803::200e): icmp_seq=1 ttl=118 time=1603 ms
64 bytes from fra24s02-in-x0e.1e100.net (2a00:1450:4001:803::200e): icmp_seq=2 ttl=118 time=1679 ms
64 bytes from fra24s02-in-x0e.1e100.net (2a00:1450:4001:803::200e): icmp_seq=3 ttl=118 time=1629 ms
64 bytes from fra24s02-in-x0e.1e100.net (2a00:1450:4001:803::200e): icmp_seq=4 ttl=118 time=1439 ms
^C
--- google.com ping statistics ---
5 packets transmitted, 4 received, 20% packet loss, time 4030ms
rtt min/avg/max/mdev = 1439.251/1588.127/1679.683/90.173 ms, pipe 2
root@ubuntu:~#
```

Figure 4 - Pinging the outside world

As we can see, pinging on some Internet hosts works – even in IPv6! However, there is a performance penalty to be paid by the double software bridge (on the Linux VM and on the VirtualBox host), which causes packet losses. Generally speaking, it caused no issues – I could run *apt* and update and install packages as needed.

1) Lab Assessment 1: The OVS…

- …can correctly show the created bridges via the open vswitch command:

```
root@ubuntu:~# ovs-vsctl show
b0159eb6-dfd1-4693-b7a2-f90817a738ac
    Bridge "br0"
        Port "enp0s3"
            Interface "enp0s3"
        Port "br0"
            Interface "br0"
                type: internal
    ovs_version: "2.9.8"
root@ubuntu:~#
```

Figure 5 - lab-cs-net-00006 - Lab 1-1

- … can show the correct bound between the br0 and ens33 (in my case, enp0s3):

```
root@ubuntu:~# ovs-vsctl show
b0159eb6-dfd1-4693-b7a2-f90817a738ac
    Bridge "br0"
        Port "enp0s3"
            Interface "enp0s3"
        Port "br0"
            Interface "br0"
                type: internal
    ovs_version: "2.9.8"
root@ubuntu:~#
```

Figure 6 - lab-cs-net-00006 - Lab 1-2

- …can show the correct IP address assigned to the br0 (in my case, it is obtained via DHCP):

```
root@ubuntu:~# ifconfig br0
br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.143  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 2a01:c23:b82c:d704:a00:27ff:fe38:8a7b  prefixlen 64  scopeid 0x0<global>
        inet6 2a01:c23:b82c:d701:a00:27ff:fe38:8a7b  prefixlen 64  scopeid 0x0<global>
        inet6 fd7c:4c9f:36ca:4:60c5:4ad8:f4a8:201f  prefixlen 64  scopeid 0x0<global>
        inet6 fd59:3177:8a0a:1:60c5:4ad8:f4a8:201f  prefixlen 64  scopeid 0x0<global>
        inet6 fd7c:4c9f:36ca:4:a00:27ff:fe38:8a7b  prefixlen 64  scopeid 0x0<global>
        inet6 2a01:c23:b82c:d701:60c5:4ad8:f4a8:201f  prefixlen 64  scopeid 0x0<global>
        inet6 fd59:3177:8a0a:1:a00:27ff:fe38:8a7b  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::c0f0:ddff:fe41:1941  prefixlen 64  scopeid 0x20<link>
        inet6 2a01:c23:b82c:d704:60c5:4ad8:f4a8:201f  prefixlen 64  scopeid 0x0<global>
        ether 08:00:27:38:8a:7b  txqueuelen 1000  (Ethernet)
        RX packets 5232  bytes 552068 (552.0 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 329  bytes 47437 (47.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@ubuntu:~#
```

Figure 7- lab-cs-net-00006 - Lab 1-3/1

As a counterproof, I am showing that the ep0s3 interface has no (IPv4) address. To note that IPv6 is unaffected by "ifconfig 0" (I wonder if this has to do with the packet loss; note that both interfaces' statistics do not report any transmission error):

```
root@ubuntu:~# ifconfig enp0s3
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::853e:cc54:13a3:86a8  prefixlen 64  scopeid 0x20<link>
        inet6 fe80::cd42:eba6:2e84:8cf  prefixlen 64  scopeid 0x20<link>
        inet6 fd7c:4c9f:30ca:4:37c:1cb8:7746:5302  prefixlen 64  scopeid 0x0<global>
        inet6 2a01:c23:b82c:d704:84f4:9343:3fd7:ae42  prefixlen 64  scopeid 0x0<global>
        inet6 2a01:c23:b82c:d704:c10c:4065:f2f0:ee8d  prefixlen 64  scopeid 0x0<global>
        inet6 fd59:3177:8a0a:1:c0f3:3620:f246:7ed8  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::e0a5:a4c4:9f32:74e7  prefixlen 64  scopeid 0x20<link>
        inet6 2a01:c23:b82c:d701:eb5a:73c9:d0c2:f78b  prefixlen 64  scopeid 0x0<global>
        ether 08:00:27:38:8a:7b  txqueuelen 1000  (Ethernet)
        RX packets 7531  bytes 2634788 (2.6 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1858  bytes 201547 (201.5 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@ubuntu:~#
```

Figure 8 - lab-cs-net-00006 - Lab 1-3/2

2) *Lab Assessment 2: After finishing the configuration of br0 in OVS, the machine…*

- …can show the correct routing table for especially for the br0 and ens3 (in my case, enp0s3):

```
root@ubuntu:~# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         _gateway        0.0.0.0         UG    0      0        0 br0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 br0
root@ubuntu:~# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    0      0        0 br0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 br0
root@ubuntu:~#
```

Figure 9 - lab-cs-net-00006 - Lab 2-1

- …can ping outside such as ping 8.8.8.8 correctly:

```
root@ubuntu:~# ping 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=2 ttl=118 time=164 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=117 time=1991 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=117 time=833 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=118 time=15.5 ms
^C
--- 8.8.8.8 ping statistics ---
5 packets transmitted, 4 received, 20% packet loss, time 4033ms
rtt min/avg/max/mdev = 15.558/751.109/1991.123/779.396 ms, pipe 2
root@ubuntu:~#
root@ubuntu:~# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
default         _gateway        0.0.0.0         UG    0      0        0 br0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 br0
root@ubuntu:~# ping google.com
PING google.com(fra07s30-in-x200e.1e100.net (2a00:1450:4001:803::200e)) 56 data bytes
64 bytes from fra24s02-in-x0e.1e100.net (2a00:1450:4001:803::200e): icmp_seq=1 ttl=118 time=1603 ms
64 bytes from fra24s02-in-x0e.1e100.net (2a00:1450:4001:803::200e): icmp_seq=2 ttl=118 time=1679 ms
64 bytes from fra24s02-in-x0e.1e100.net (2a00:1450:4001:803::200e): icmp_seq=3 ttl=118 time=1629 ms
64 bytes from fra24s02-in-x0e.1e100.net (2a00:1450:4001:803::200e): icmp_seq=4 ttl=118 time=1439 ms
^C
--- google.com ping statistics ---
5 packets transmitted, 4 received, 20% packet loss, time 4030ms
rtt min/avg/max/mdev = 1439.251/1588.127/1679.683/90.173 ms, pipe 2
root@ubuntu:~#
```

Figure 10 - Ping outside

B. Lab "lab-cs-net-00007 - Mininet"

1) Setting up mininet and Running mininet topology…

- *…can correctly create the topology with a single switch and four hosts*

```
root@ubuntu:~# sudo mn --topo=linear,4
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (h4, s4) (s2, s1) (s3, s2) (s4, s3)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Starting CLI:
containernet> nodes
```

Figure 11 - lab-cs-net-0007 - Lab 1-1

- …can correctly create the linear topology with five switch and one host for every switch:

```
root@ubuntu:~# mn --topo=linear,5
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5
*** Adding switches:
s1 s2 s3 s4 s5
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (h4, s4) (h5, s5) (s2, s1) (s3, s2) (s4, s3) (s5, s4)
*** Configuring hosts
h1 h2 h3 h4 h5
*** Starting controller
c0
*** Starting 5 switches
s1 s2 s3 s4 s5 ...
*** Starting CLI:
containernet>
```

Figure 12- lab-cs-net-0007 - Lab 1-2

2) For each topology above, they should be able to:

- show the correct number of nodes within the current topology

  Single switch and four hosts:

```
root@ubuntu:~# sudo mn --topo=linear,4
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (h4, s4) (s2, s1) (s3, s2) (s4, s3)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Starting CLI:
containernet> nodes
available nodes are:
c0 h1 h2 h3 h4 s1 s2 s3 s4
containernet> iperf
```

Figure 13 - lab-cs-net-0007 - Lab 2-1 – topology 1

Linear topology of five switch and one host for each switch:

```
root@ubuntu:~# mn --topo=linear,5
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5
*** Adding switches:
s1 s2 s3 s4 s5
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (h4, s4) (h5, s5) (s2, s1) (s3, s2) (s4, s3) (s5, s4)
*** Configuring hosts
h1 h2 h3 h4 h5
*** Starting controller
c0
*** Starting 5 switches
s1 s2 s3 s4 s5 ...
*** Starting CLI:
containernet> nodes
available nodes are:
c0 h1 h2 h3 h4 h5 s1 s2 s3 s4 s5
containernet>
```

Figure 14 - lab-cs-net-0007 - Lab 2-1 – topology 2

- …perform bandwidth measurement.

Single switch and four hosts:

```
ubuntu@ubuntu:~$ sudo mn --topo single,4
*** Creating network
*** Adding controller
*** 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
*** Starting 1 switches
s1 ...
*** Starting CLI:
containernet> iperf
*** Iperf: testing TCP bandwidth between h1 and h4
*** Results: ['20.2 Gbits/sec', '20.2 Gbits/sec']
containernet>
```

Figure 15- lab-cs-net-0007 - Lab 2-2 - Topology 1

Linear topology of five switch and one host for each switch:

```
ubuntu@ubuntu:~$ sudo mn --topo linear,5
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5
*** Adding switches:
s1 s2 s3 s4 s5
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (h4, s4) (h5, s5) (s2, s1) (s3, s2) (s4, s3) (s5, s4)
*** Configuring hosts
h1 h2 h3 h4 h5
*** Starting controller
c0
*** Starting 5 switches
s1 s2 s3 s4 s5 ...
*** Starting CLI:
containernet> iperf
*** Iperf: testing TCP bandwidth between h1 and h5
*** Results: ['15.1 Gbits/sec', '15.1 Gbits/sec']
containernet>
```

Figure 16- lab-cs-net-0007 - Lab 2-2 - Topology 2

- …display the correct link information among hosts and switches:

Single switch and four hosts:

```
ubuntu@ubuntu:~$ sudo mn --topo single,4
*** Creating network
*** Adding controller
*** 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
*** Starting 1 switches
s1 ...
*** Starting CLI:
containernet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
h3 h3-eth0:s1-eth3
h4 h4-eth0:s1-eth4
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0 s1-eth3:h3-eth0 s1-eth4:h4-eth0
c0
containernet>
```

Figure 17- lab-cs-net-0007 - Lab 2-3 - Topology 1

Linear topology of five switch and one host for each switch:

```
ubuntu@ubuntu:~$ sudo mn --topo linear,5
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5
*** Adding switches:
s1 s2 s3 s4 s5
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (h4, s4) (h5, s5) (s2, s1) (s3, s2) (s4, s3) (s5, s4)
*** Configuring hosts
h1 h2 h3 h4 h5
*** Starting controller
c0
*** Starting 5 switches
s1 s2 s3 s4 s5 ...
*** Starting CLI:
containernet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
h3 h3-eth0:s3-eth1
h4 h4-eth0:s4-eth1
h5 h5-eth0:s5-eth1
s1 lo:  s1-eth1:h1-eth0 s1-eth2:s2-eth2
s2 lo:  s2-eth1:h2-eth0 s2-eth2:s1-eth2 s2-eth3:s3-eth2
s3 lo:  s3-eth1:h3-eth0 s3-eth2:s2-eth3 s3-eth3:s4-eth2
s4 lo:  s4-eth1:h4-eth0 s4-eth2:s3-eth3 s4-eth3:s5-eth2
s5 lo:  s5-eth1:h5-eth0 s5-eth2:s4-eth3
c0
containernet>
```

Figure 18- lab-cs-net-0007 - Lab 2-3 - Topology 2

3) *Create another tree topology of depth 2 and fanout 8.*

- Startup with "*sudo mn --topo tree,depth=2 fanout=8*":

```
ubuntu@ubuntu:~$ sudo mn --topo tree,depth=2,fanout=8
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25 h26 h27 h28 h29 h30 h31 h32 h33 h
34 h35 h36 h37 h38 h39 h40 h41 h42 h43 h44 h45 h46 h47 h48 h49 h50 h51 h52 h53 h54 h55 h56 h57 h58 h59 h60 h61 h62 h63 h64
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7 s8 s9
*** Adding links:
(s1, s2) (s1, s3) (s1, s4) (s1, s5) (s1, s6) (s1, s7) (s1, s8) (s1, s9) (s2, h1) (s2, h2) (s2, h3) (s2, h4) (s2, h5) (s2, h6
) (s2, h7) (s2, h8) (s3, h9) (s3, h10) (s3, h11) (s3, h12) (s3, h13) (s3, h14) (s3, h15) (s3, h16) (s4, h17) (s4, h18) (s4,
h19) (s4, h20) (s4, h21) (s4, h22) (s4, h23) (s4, h24) (s5, h25) (s5, h26) (s5, h27) (s5, h28) (s5, h29) (s5, h30) (s5, h31)
 (s5, h32) (s6, h33) (s6, h34) (s6, h35) (s6, h36) (s6, h37) (s6, h38) (s6, h39) (s6, h40) (s7, h41) (s7, h42) (s7, h43) (s7
, h44) (s7, h45) (s7, h46) (s7, h47) (s7, h48) (s8, h49) (s8, h50) (s8, h51) (s8, h52) (s8, h53) (s8, h54) (s8, h55) (s8, h5
6) (s9, h57) (s9, h58) (s9, h59) (s9, h60) (s9, h61) (s9, h62) (s9, h63) (s9, h64)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16 h17 h18 h19 h20 h21 h22 h23 h24 h25 h26 h27 h28 h29 h30 h31 h32 h33 h
34 h35 h36 h37 h38 h39 h40 h41 h42 h43 h44 h45 h46 h47 h48 h49 h50 h51 h52 h53 h54 h55 h56 h57 h58 h59 h60 h61 h62 h63 h64
*** Starting controller
c0
*** Starting 9 switches
s1 s2 s3 s4 s5 s6 s7 s8 s9 ...
*** Starting CLI:
containernet>
```

Figure 19- lab-cs-net-0007 - Lab 3-1

- the host1 can correctly ping the host64:

```
c0
*** Starting 9 switches
s1 s2 s3 s4 s5 s6 s7 s8 s9 ...
*** Starting CLI:
containernet> h1 ping -c 2 h64
PING 10.0.0.64 (10.0.0.64) 56(84) bytes of data.
64 bytes from 10.0.0.64: icmp_seq=1 ttl=64 time=107 ms
64 bytes from 10.0.0.64: icmp_seq=2 ttl=64 time=0.515 ms

--- 10.0.0.64 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.515/54.245/107.975/53.730 ms
containernet>
```

Figure 20- lab-cs-net-0007 - Lab 3-2

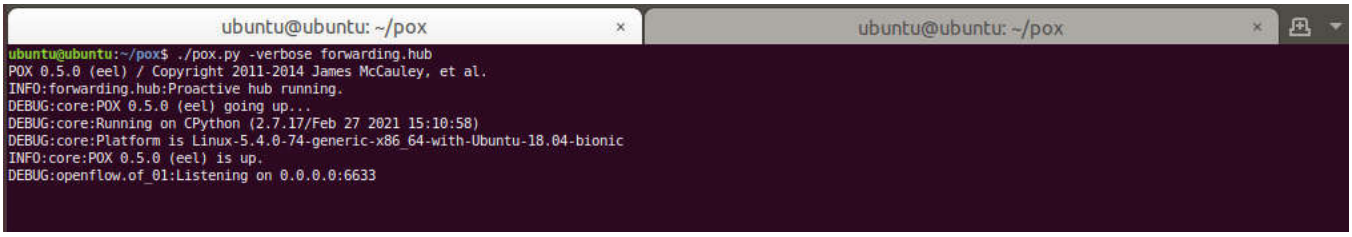
(End of "lab-cs-net-00007 - Mininet")

C. Lab "CS-NET-00008 – SDN Controller (POX)"

1) The POX controller can:

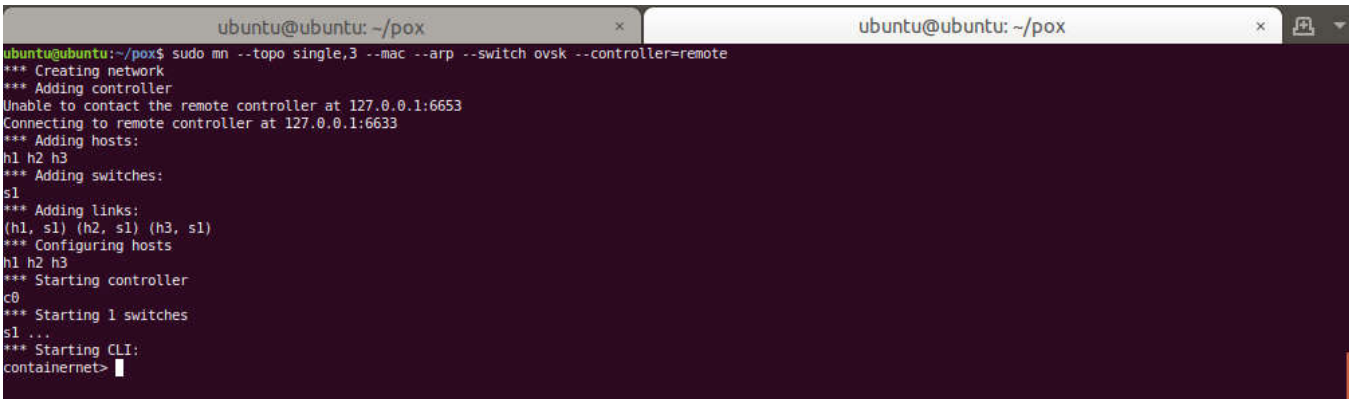
- connect to a mininet topology:

  The behavior of pox and mininet can be observed by switching between the two tabs where the programs are running, showing that mininet connected to openflow:



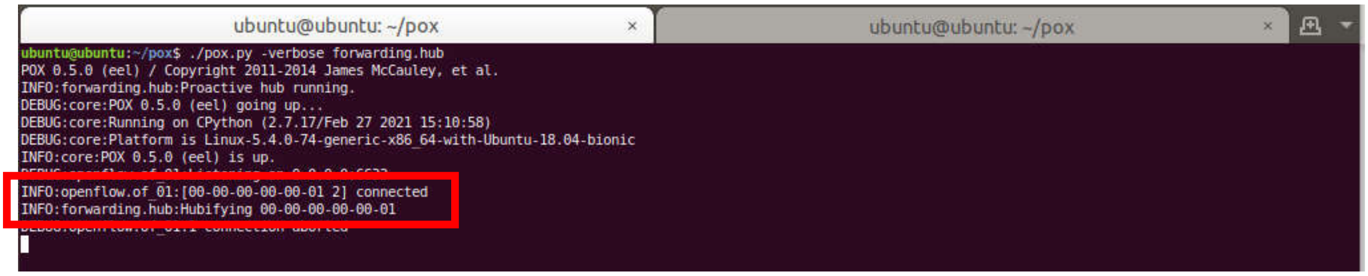
```
ubuntu@ubuntu:~/pox$ ./pox.py -verbose forwarding.hub
POX 0.5.0 (eel) / Copyright 2011-2014 James McCauley, et al.
INFO:forwarding.hub:Proactive hub running.
DEBUG:core:POX 0.5.0 (eel) going up...
DEBUG:core:Running on CPython (2.7.17/Feb 27 2021 15:10:58)
DEBUG:core:Platform is Linux-5.4.0-74-generic-x86_64-with-Ubuntu-18.04-bionic
INFO:core:POX 0.5.0 (eel) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
```

Figure 21 - lab-cs-net-0087 - Lab 1-1/1



```
ubuntu@ubuntu:~/pox$ sudo mn --topo single,3 --mac --arp --switch ovsk --controller=remote
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Connecting to remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
containernet>
```

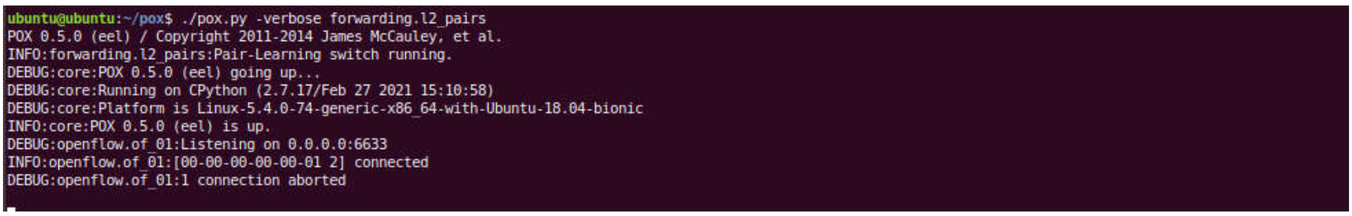
Figure 22 - lab-cs-net-0087 - Lab 1-1/2



```
ubuntu@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-5.4.0-74-generic-x86_64-with-Ubuntu-18.04-bionic
INFO:core:POX 0.5.0 (eel) is up.
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
INFO:forwarding.hub:Hubifying 00-00-00-00-00-01
```

Figure 23 - lab-cs-net-0087 - Lab 1-1/3

- start the forwarding.l3 learning component correctly



```
ubuntu@ubuntu:~/pox$ ./pox.py -verbose forwarding.l2_pairs
POX 0.5.0 (eel) / Copyright 2011-2014 James McCauley, et al.
INFO:forwarding.l2_pairs:Pair-Learning switch 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-5.4.0-74-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
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
DEBUG:openflow.of_01:1 connection aborted
```

Figure 24 - lab-cs-net-0087 - Lab 1-2/1

Figure 25 - lab-cs-net-0087 - Lab 1-2/2

2) *While connect to the controller, the mininet topology can (Please take snapshots for each task)*
   - open the xterm terminal correctly
   - enable the ping command between two different host in two different xterm terminal (two different hosts)

On this first image, I have opened the three xterm with the command "xterm h1 h2 h3" highlighted on the bottom. I have also run a couple of ping commands.

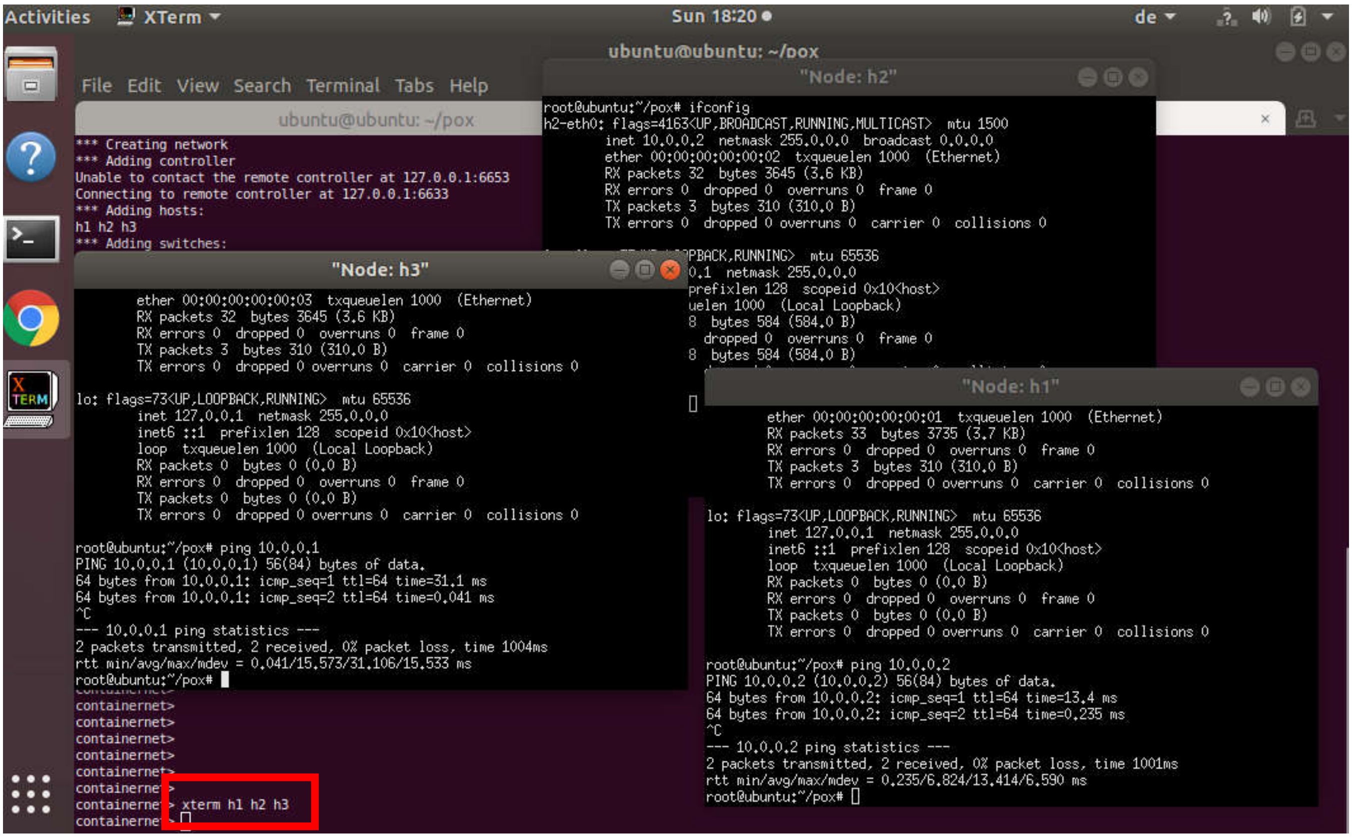


Figure 26 - lab-cs-net-0087 - Lab 2-1

In this screenshot, I am running tcpdump on one xterm while executing a ping from another. In this way it is possible to notice the incoming packets.

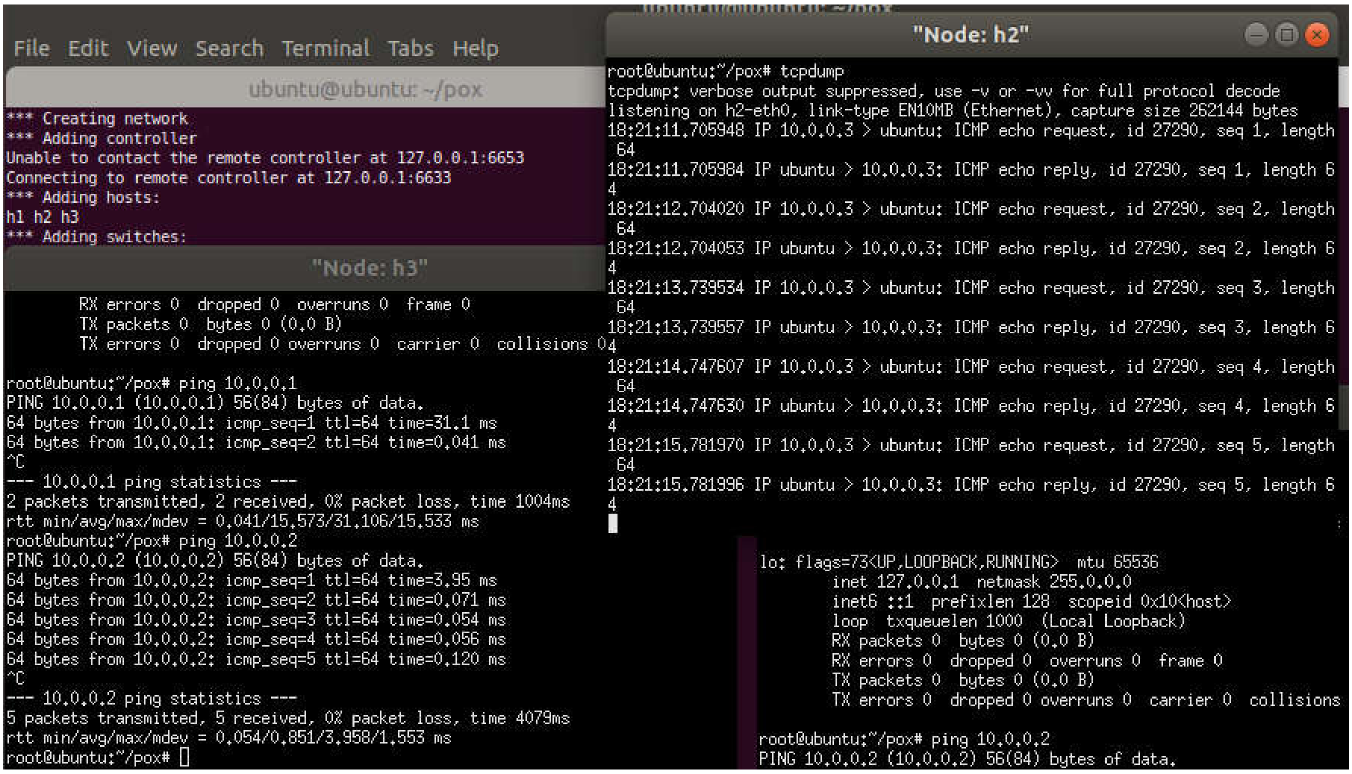


Figure 27 - lab-cs-net-0087 - Lab 2-2

3) Extra Activity

I have started the pox webserver connected to mininet to test its functionality.

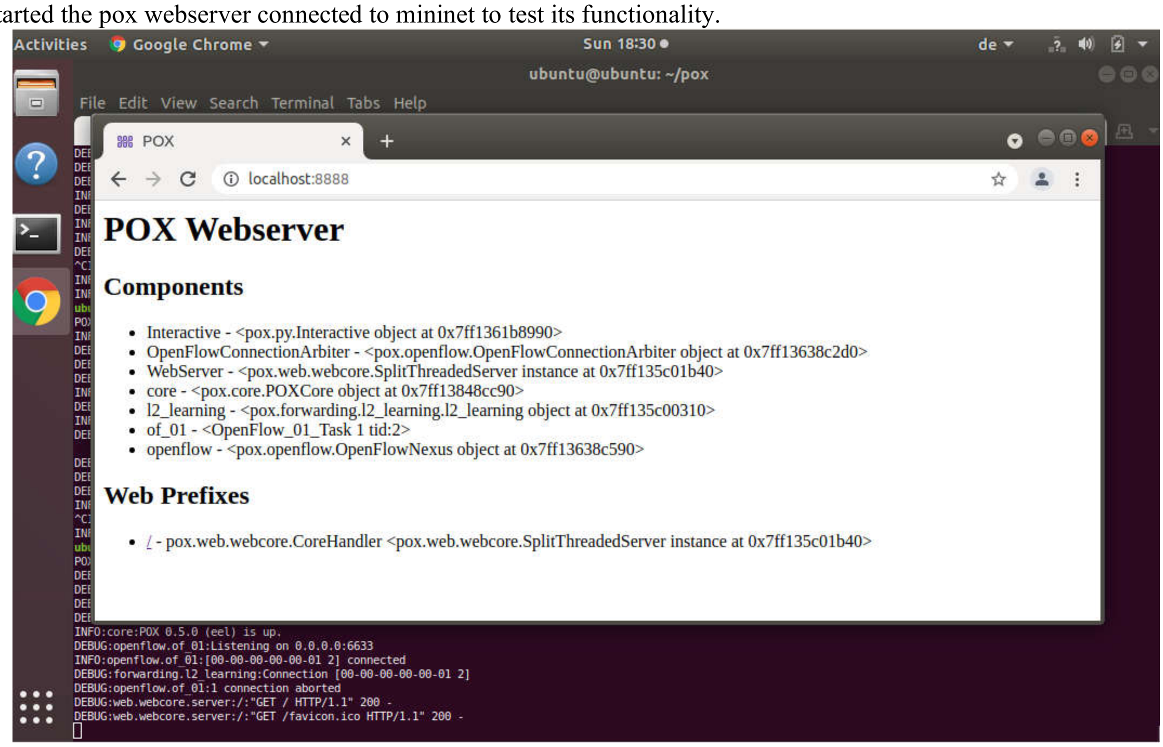


Figure 28 - - lab-cs-net-0087 - Extra Activity

D. Lab "CS-NET-00009 – Containernet"

1) The containernet topology can:

- successfully implement the test ping command in a temporary topology

```
ubuntu@ubuntu:~/containernet$ sudo 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 5.824 seconds
ubuntu@ubuntu:~/containernet$
```

Figure 29 - lab-cs-net-0009 – Lab 1-1

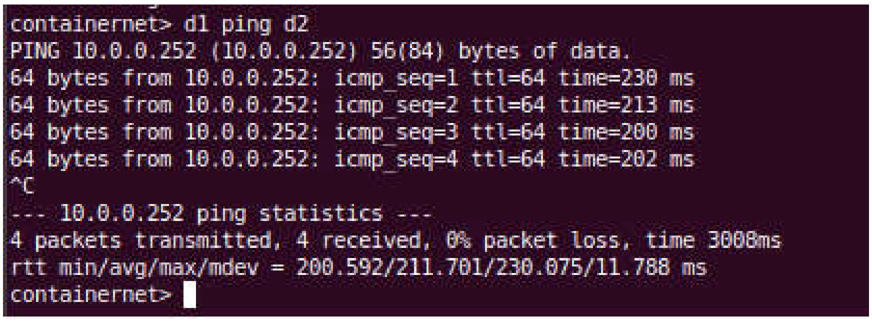
After started the scripts the container example.py can

- create the indicated topology

```
ubuntu@ubuntu:~/containernet$ sudo python3 examples/containernet_example.py
*** Adding controller
*** Adding docker containers
1:
d1: kwargs {'ip': '10.0.0.251'}
d1: update resources {'cpu_quota': -1}
1:
d2: kwargs {'ip': '10.0.0.252'}
d2: update resources {'cpu_quota': -1}
*** Adding switches
*** Creating links
(1.00Mbit 100ms delay) (1.00Mbit 100ms delay) (1.00Mbit 100ms delay) (1.00Mbit 100ms delay) *** Starting network
*** Configuring hosts
d1 d2
*** Starting controller
c0
*** Starting 2 switches
s1 (1.00Mbit 100ms delay) s2 (1.00Mbit 100ms delay) ...(1.00Mbit 100ms delay) (1.00Mbit 100ms delay)
*** Testing connectivity
d1 -> d2
d2 -> d1
*** Results: 0% dropped (2/2 received)
*** Running CLI
*** Starting CLI:
containernet>
```

Figure 30 - lab-cs-net-0009 – Lab 1-2

- enable the communication between the d1 and d2 (link up the two switches):



```
containernet> d1 ping d2
PING 10.0.0.252 (10.0.0.252) 56(84) bytes of data.
64 bytes from 10.0.0.252: icmp_seq=1 ttl=64 time=230 ms
64 bytes from 10.0.0.252: icmp_seq=2 ttl=64 time=213 ms
64 bytes from 10.0.0.252: icmp_seq=3 ttl=64 time=200 ms
64 bytes from 10.0.0.252: icmp_seq=4 ttl=64 time=202 ms
^C
--- 10.0.0.252 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3008ms
rtt min/avg/max/mdev = 200.592/211.701/230.075/11.788 ms
containernet>
```

Figure 31 - lab-cs-net-0009 – Lab 1-3

(End of “lab-cs-net-00009 - Containernet”)

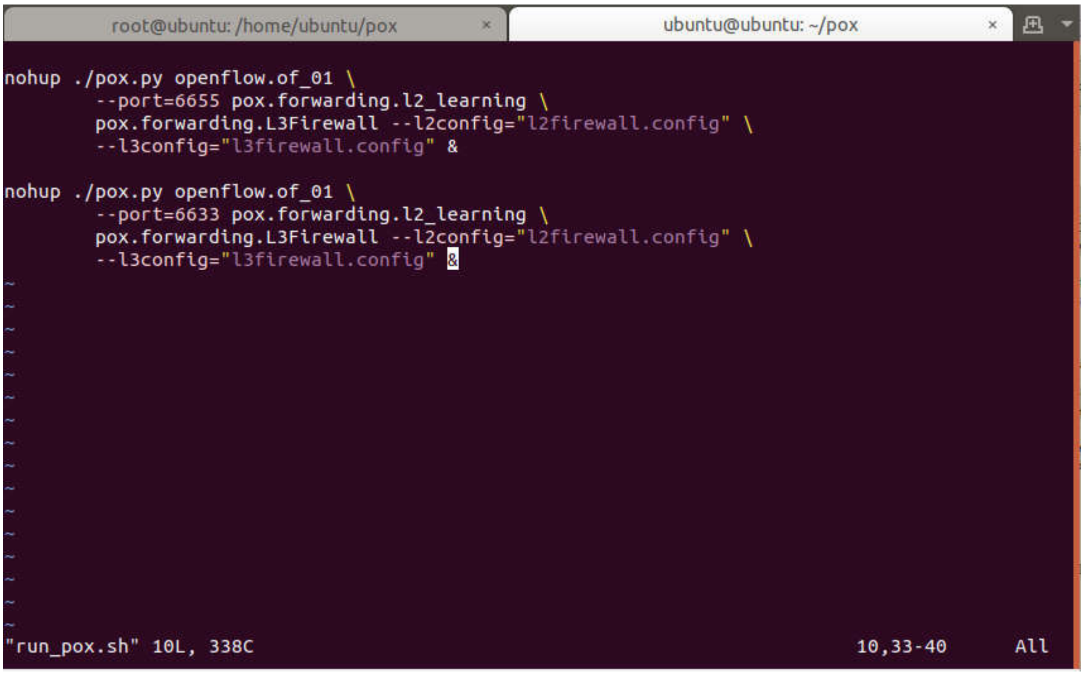
E. Lab "CS-CNS-00101" – OpenFlow Based Stateless firewall

In this lab, the students are required to verify working of stateless firewall and try adding different rules using config files.

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

- *Add link from controller1 to switch 1.*
- *Add link from controller2 to switch 1.*
- *Add link from switch 1 to container 1.*
- *Add link from switch 1 to container 2.*
- *Add link from switch 1 to container 3.*
- *Add link from switch 1 to container 4.*

This is accomplished by two steps: first, creating two controllers with pox (only one of the two will be really used). I have created a script, called "run_pox.sh" (present on GitHub and the Appendices), which spawns two controllers, as show below.

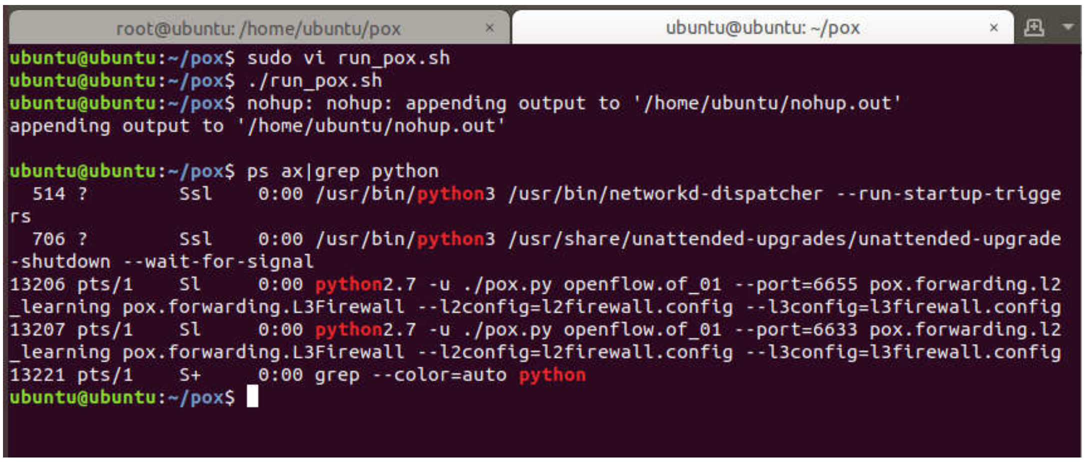

```
root@ubuntu: /home/ubuntu/pox          ubuntu@ubuntu: ~/pox

nohup ./pox.py openflow.of_01 \
        --port=6655 pox.forwarding.l2_learning \
        pox.forwarding.L3Firewall --l2config="l2firewall.config" \
        --l3config="l3firewall.config" &

nohup ./pox.py openflow.of_01 \
        --port=6633 pox.forwarding.l2_learning \
        pox.forwarding.L3Firewall --l2config="l2firewall.config" \
        --l3config="l3firewall.config" &
"run_pox.sh" 10L, 338C                                  10,33-40      All
```

Figure 32 - CS-CNS-00101 – run_pox.sh

The controllers are identifiable as python programs from the Ubuntu process status:


```
ubuntu@ubuntu:~/pox$ sudo vi run_pox.sh
ubuntu@ubuntu:~/pox$ ./run_pox.sh
ubuntu@ubuntu:~/pox$ nohup: nohup: appending output to '/home/ubuntu/nohup.out'
appending output to '/home/ubuntu/nohup.out'

ubuntu@ubuntu:~/pox$ ps ax|grep python
  514 ?        Ssl    0:00 /usr/bin/python3 /usr/bin/networkd-dispatcher --run-startup-trigge
rs
  706 ?        Ssl    0:00 /usr/bin/python3 /usr/share/unattended-upgrades/unattended-upgrade
-shutdown --wait-for-signal
13206 pts/1    Sl     0:00 python2.7 -u ./pox.py openflow.of_01 --port=6655 pox.forwarding.l2
_learning pox.forwarding.L3Firewall --l2config=l2firewall.config --l3config=l3firewall.config
13207 pts/1    Sl     0:00 python2.7 -u ./pox.py openflow.of_01 --port=6633 pox.forwarding.l2
_learning pox.forwarding.L3Firewall --l2config=l2firewall.config --l3config=l3firewall.config
13221 pts/1    S+     0:00 grep --color=auto python
ubuntu@ubuntu:~/pox$
```

Figure 33 - CS-CNS-00101 – run_pox.sh running

The second step is to run another script from another Terminal tab or windows, launching mininet. This file, called "run_lab.sh", is also presented in the Appendices and GitHub.

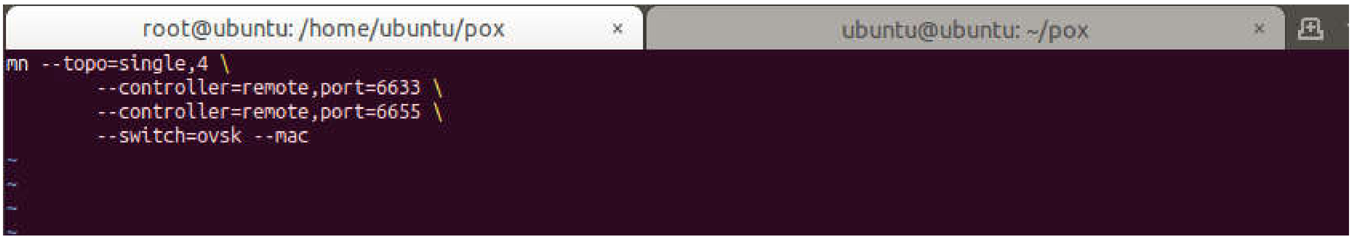


Figure 34 - CS-CNS-00101 – run_lab.sh

After mininet starts, it will bind to the two pox controllers without errors (if everything goes smoothly):

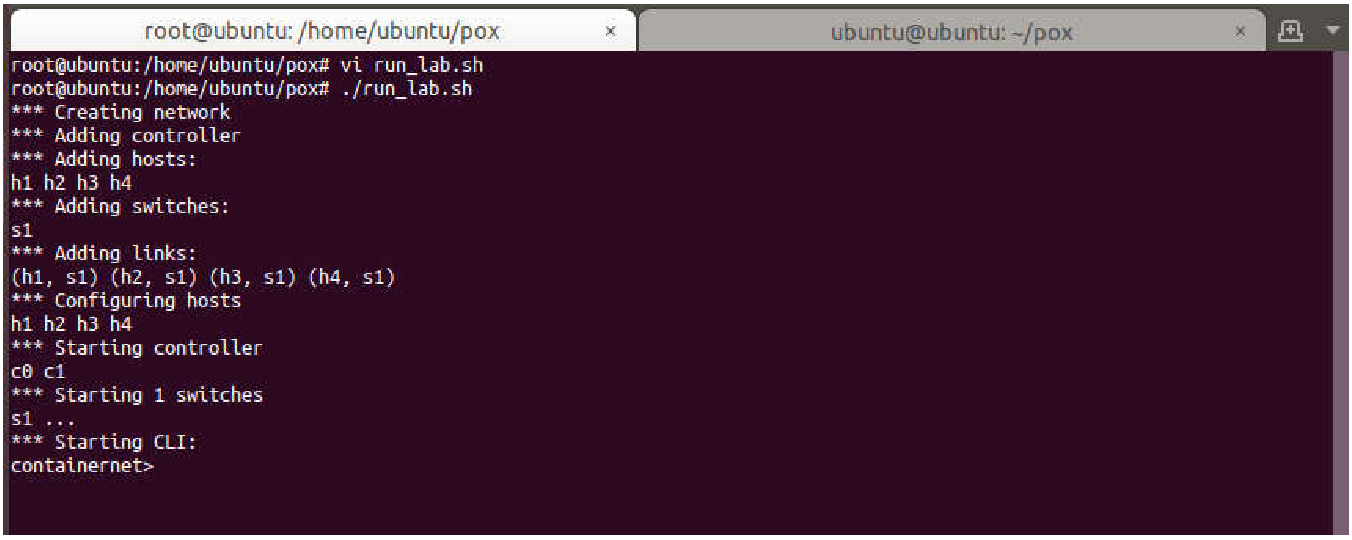


Figure 35 - CS-CNS-00101 – run_lab.sh running

At this point we can run the xterm and observe that the containers work.

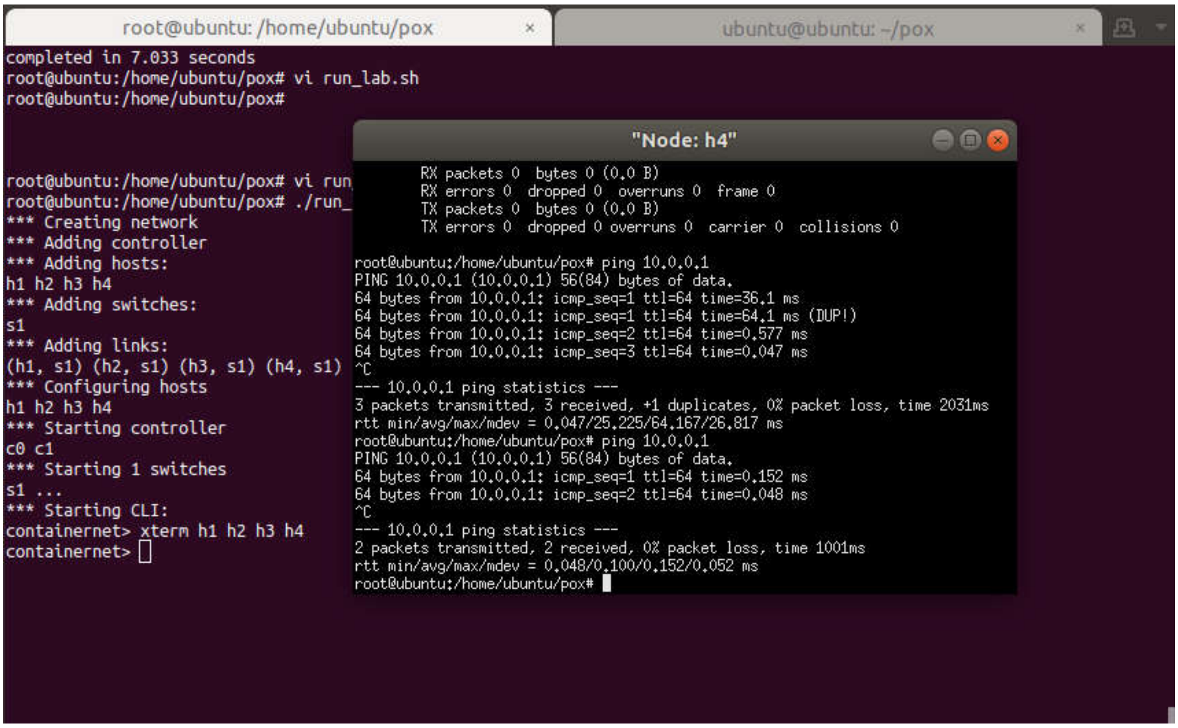


Figure 36 - CS-CNS-00101 – Lab 1

2) *Make the interfaces up and assign IP addresses to interfaces of container hosts.*
   - *Assign IP address 192.168.2.10 to container host #1.*
   - *Assign IP address 192.168.2.20 to container host #2.*
   - *Assign IP address 192.168.2.30 to container host #3.*
   - *Assign IP address 192.168.2.40 to container host #4*

The following commands are ran from the mininet CLI to assign them the addresses requested by the lab:

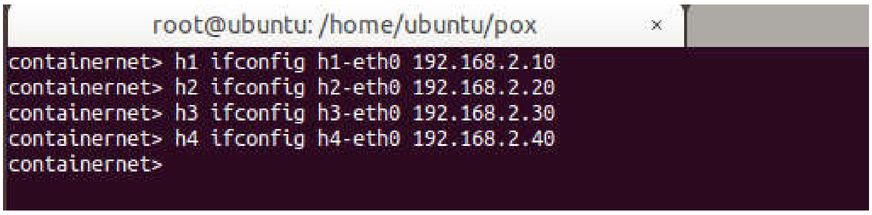


Figure 37 - CS-CNS-00101 – Change IP addresses

The next screenshot to document that the containers have all assumed the correct IP addresses.

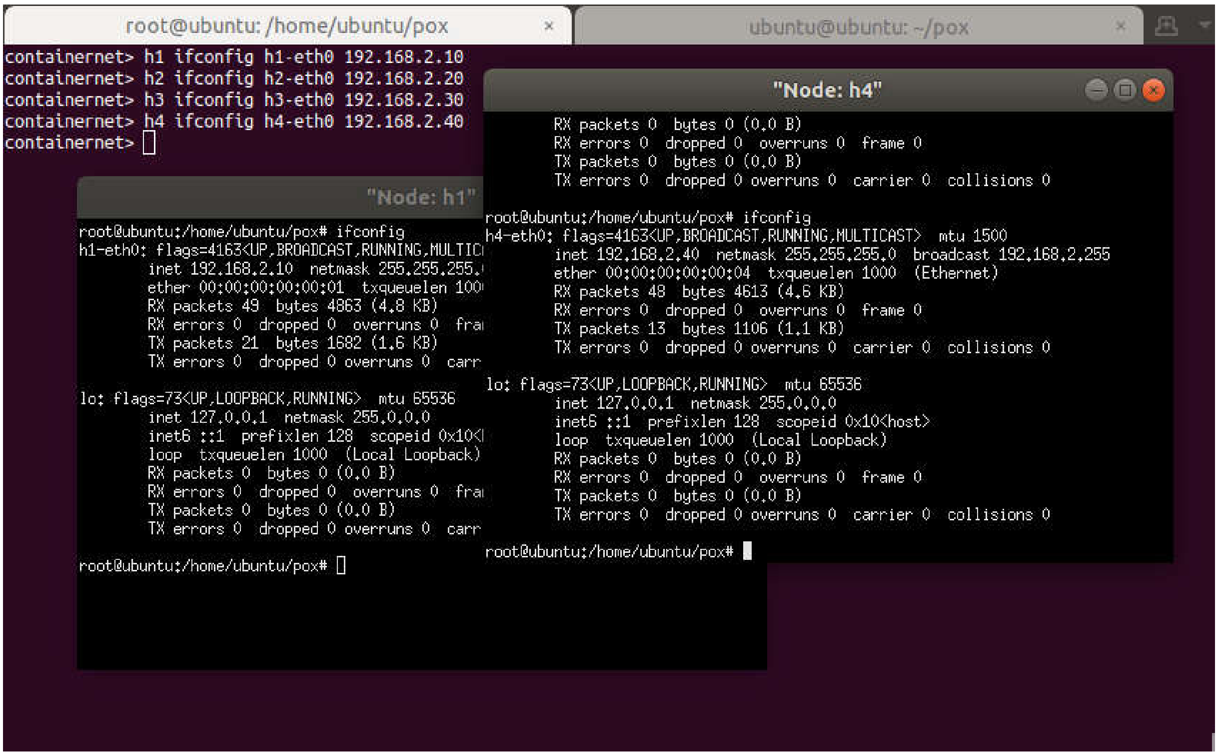


Figure 38 - CS-CNS-00101 – Lab 2

3) *Add new rule to l3config file for blocking ICMP traffic from source IP 192.168.2.10 and destination IP 192.168.2.30.*
4) *Add new rule to l3config file for blocking ICMP traffic from source IP 192.168.2.20 and destination IP 192.168.2.40.*
5) *Add new rule to l3config file for blocking HTTP traffic from source IP 192.168.2.20.*
6) *Add new rule to l2config file for blocking traffic from MAC address 00:00:00:00:00:02 to destination MAC address 00:00:00:00:00:04.*
7) *Add new rule to l3config file for blocking tcp traffic from 192.168.2.10 to 192.168.2.20.*
8) *Add new rule to l3config file for blocking udp traffic from 192.168.2.10 to 192.168.2.20.*

The following screenshot documents the 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,1,80,tcp
4,any,any,192.168.2.10,192.168.2.20,1,1,tcp
5,any,any,192.168.2.10,192.168.2.20,1,1,udp
~
~
~
```

Figure 39 - CS-CNS-00101 – l3firewall.config

The following screenshot documents the l2firewall.config:

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

Figure 40 - CS-CNS-00101 – l2firewall.config

At this point, we restart both pox and mininet.
From looking at the "nohup.out" generated by the run_pox.sh, it is possible to see the Layer3 rules enabled (but not the Layer2):

```
ubuntu@ubuntu:~/pox$ tail -f ../nohup.out
src_ip, dst_ip, src_port, dst_port 192.168.2.10 192.168.2.20 1 1
src_ip, dst_ip, src_port, dst_port 192.168.2.10 192.168.2.30 1 1
src_ip, dst_ip, src_port, dst_port 192.168.2.20 192.168.2.40 1 1
src_ip, dst_ip, src_port, dst_port 192.168.2.20 any 1 80
src_ip, dst_ip, src_port, dst_port 192.168.2.10 192.168.2.20 1 1
src_ip, dst_ip, src_port, dst_port 192.168.2.10 192.168.2.20 1 1
WARNING:core:Redefined signal handler for SIGHUP
INFO:core:POX 0.5.0 (eel) is up.
WARNING:core:Redefined signal handler for SIGHUP
INFO:core:POX 0.5.0 (eel) is up.
```

Figure 41 - CS-CNS-00101 – checking nohup.out

After we restart mininet, we must re-assign the correct IP addresses (fortunately, the CLI history has stored the command, so we must just retrieve them with the arrows and feed them to the Open vSwitch pressing Enter):

```
root@ubuntu:/home/ubuntu/pox#
root@ubuntu:/home/ubuntu/pox# ./run_lab.sh
*** Creating network
*** Adding controller
*** 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> 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>
```

Figure 42 - CS-CNS-00101 – Changing IP addresses again

Let’s now test out our rules to see if they work.

Let’s start from pinging h3 from h1. The ping should fail, but in fact, the first packet gets true, but all the subsequent are blocked. Seems like one rule must be triggered first before the vSwitch “loads” them in, but at this point they are all enabled.

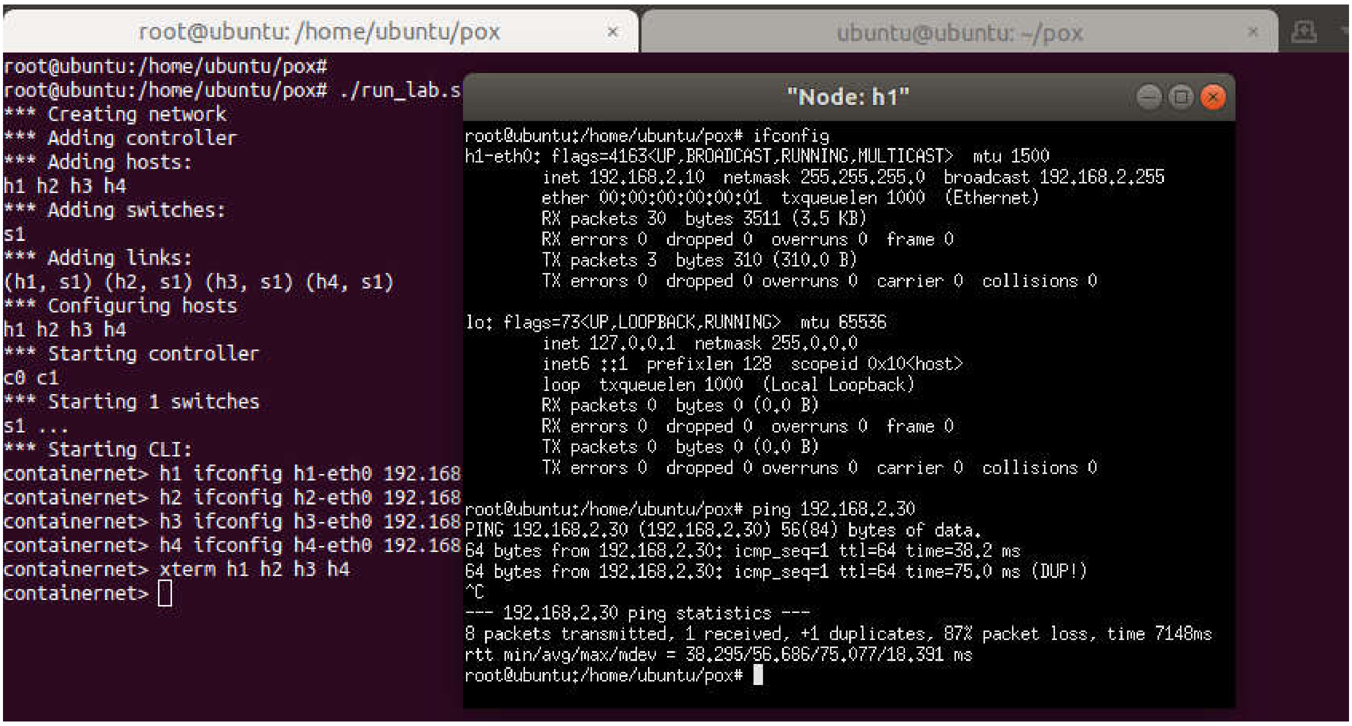


Figure 43 - CS-CNS-00101 – first ping from h1 to h3 gets through, then the rest are blocked

In fact, pinging h4 from h2 fails (not even the first packet gets through):

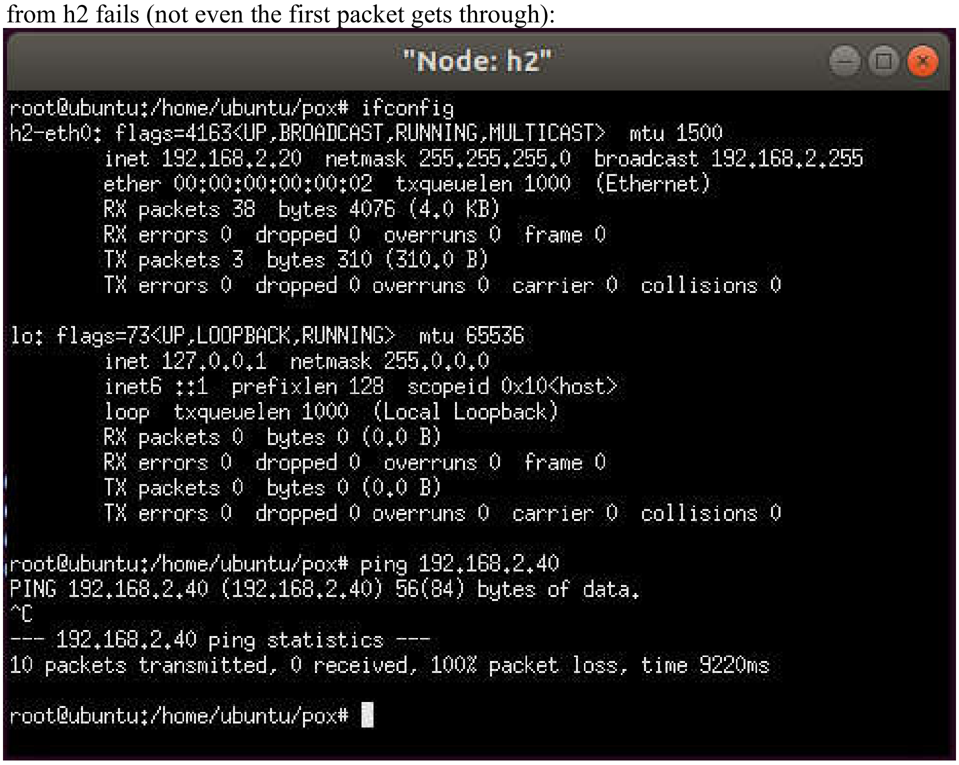


Figure 44 - CS-CNS-00101 – pings blocked from h2 to h4

I start the Python SimpleHTTPServer on port 80 of node h3 and test fetching a page from the node h4, which works (as expected):

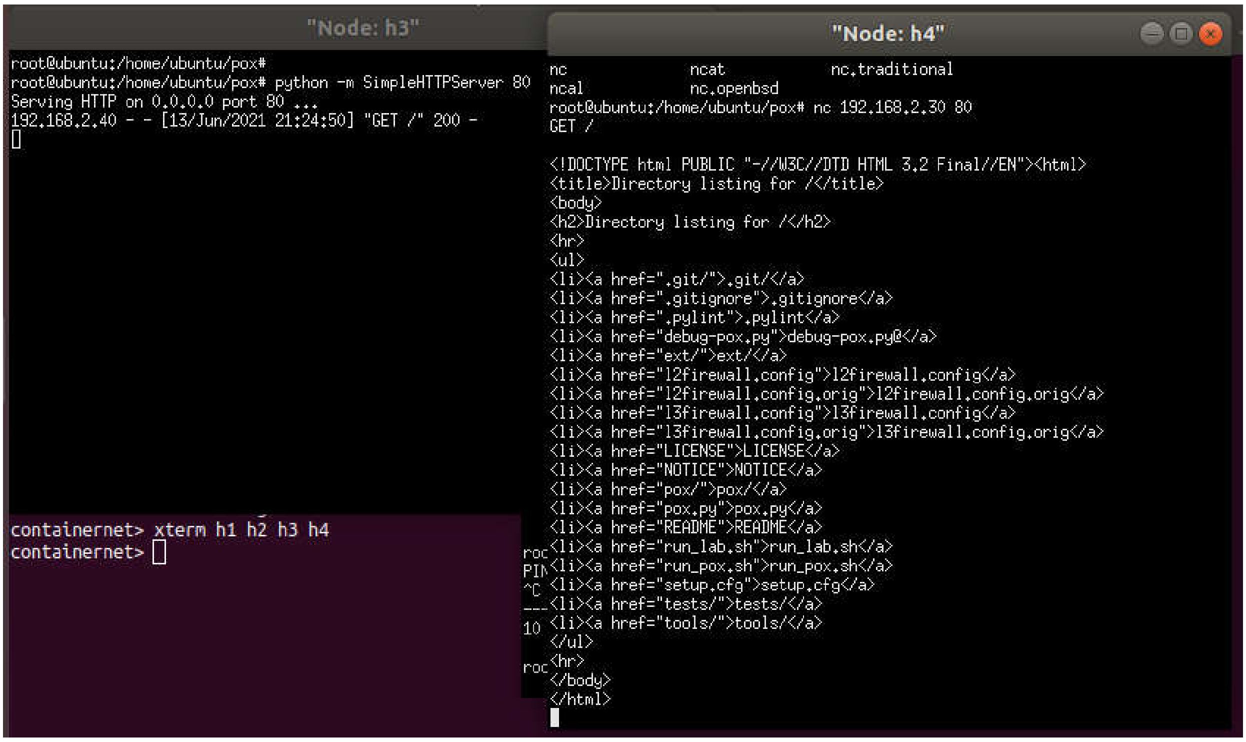


Figure 45 - CS-CNS-00101 – h4 can reach web server on h3

However, trying to "browse" the web page at port 80 from node h2 – no matter if with netcat or curl – fails as the connection is dropped and, as we can see, it never reaches the node h3 – confirming the Layer3 rule works as intended:

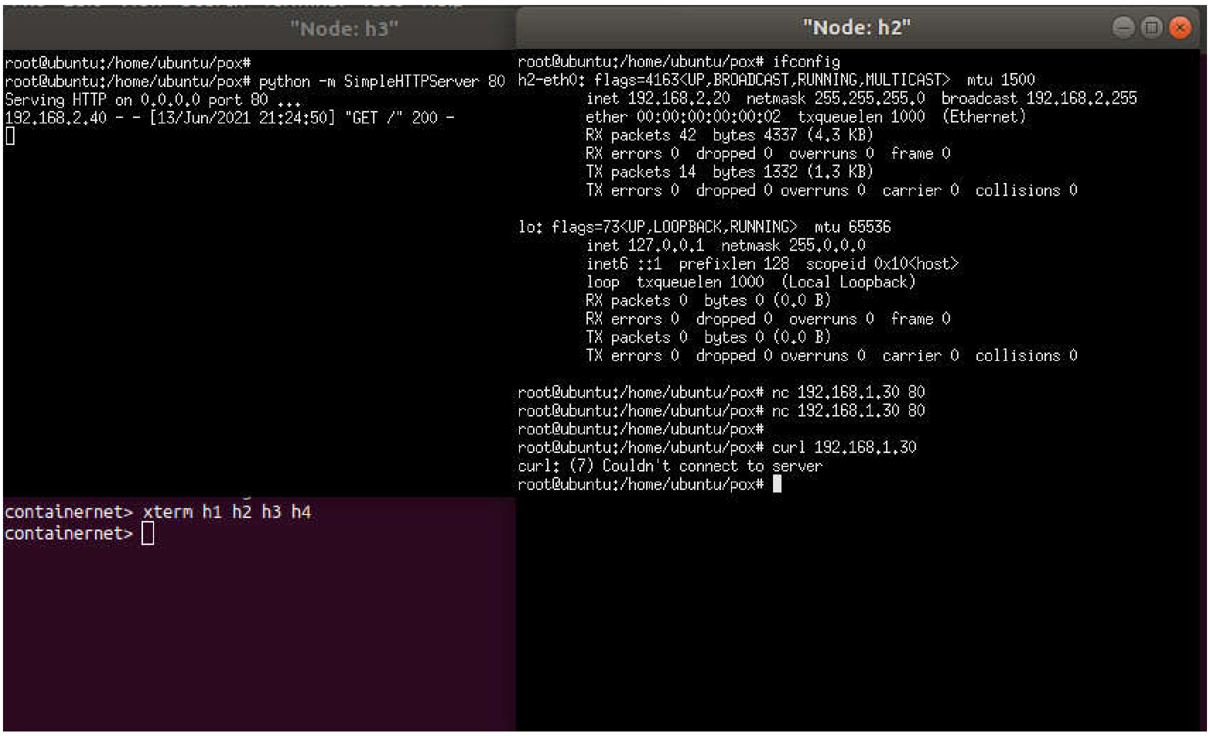


Figure 46 - CS-CNS-00101 – h2 cannot reach web server on h3

Another test against the web server on h3, this time from h1, shows that effectively only h2 is blocked from reaching web pages.

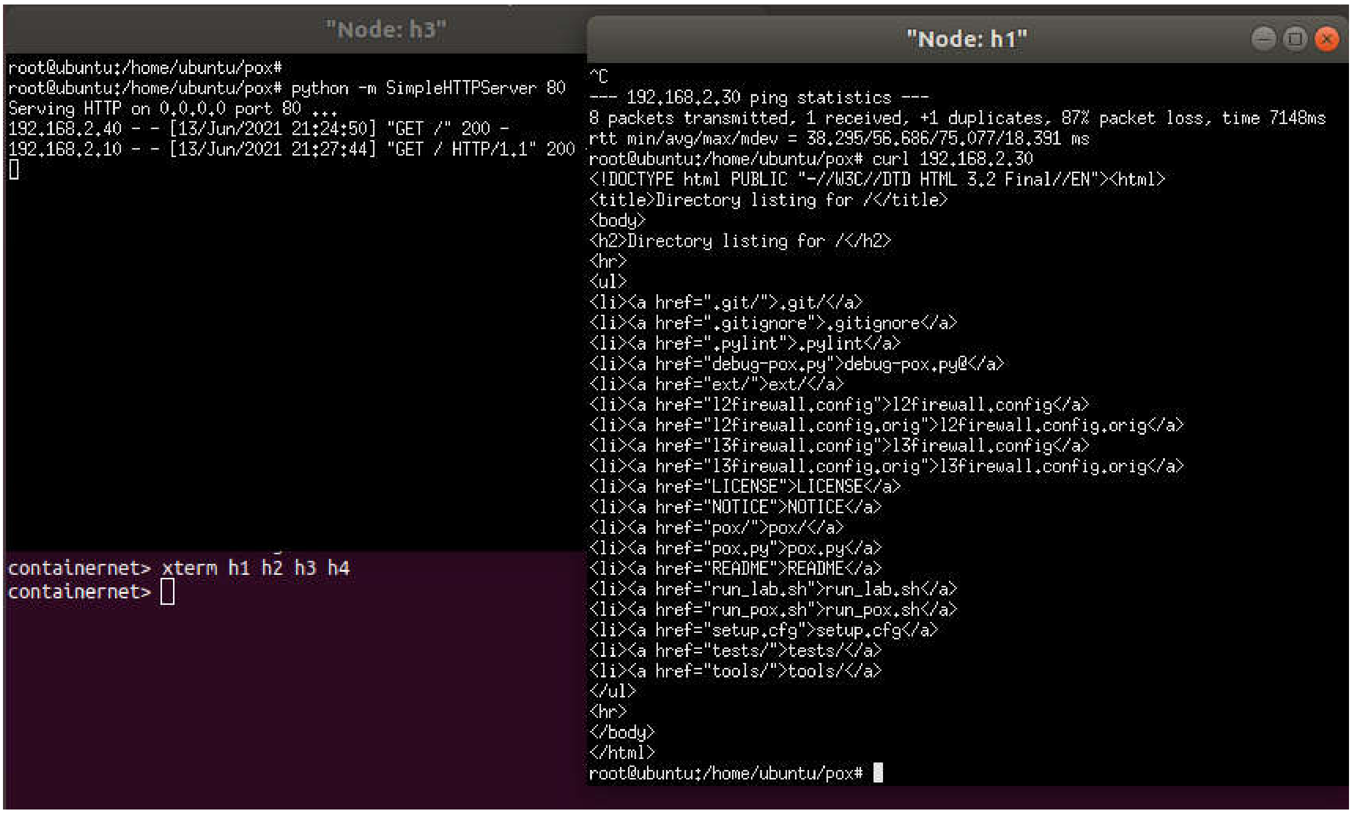


Figure 47 - CS-CNS-00101 – h1 can reach web server on h3

Finally, between h2 and h4, the block should be at the Layer2 level. To test this, I have tried first using ping, then running tcpdump on h4, while doing scans from node h2.

The first is a UDP scan:

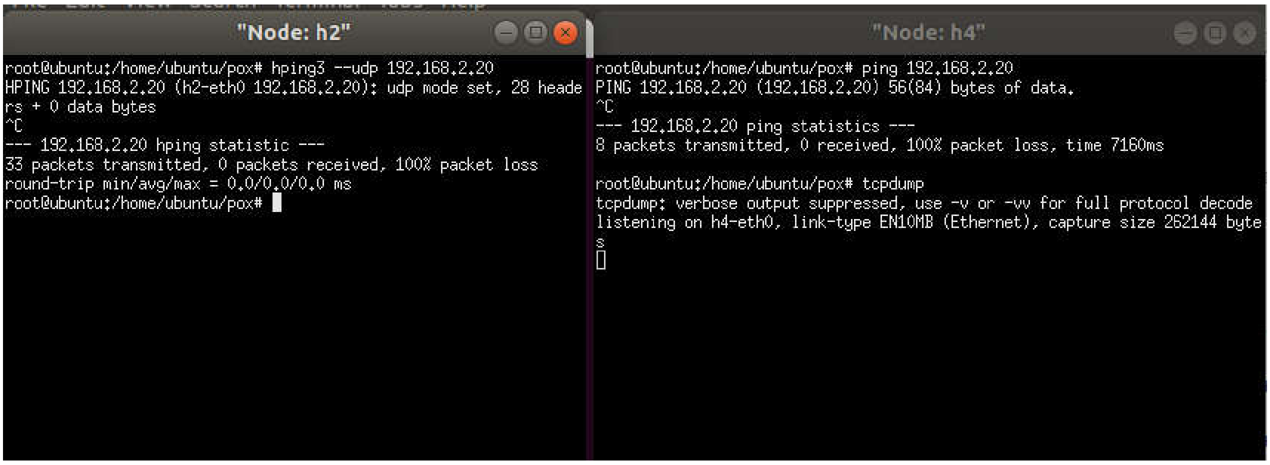


Figure 48 - CS-CNS-00101 – h2 ping and UDP scan against h4 fail

The second is a TCP scan on some 8000 circa ports. As we can see, no packet ever reaches node h4 – be it ICPM, TCP or UDP.

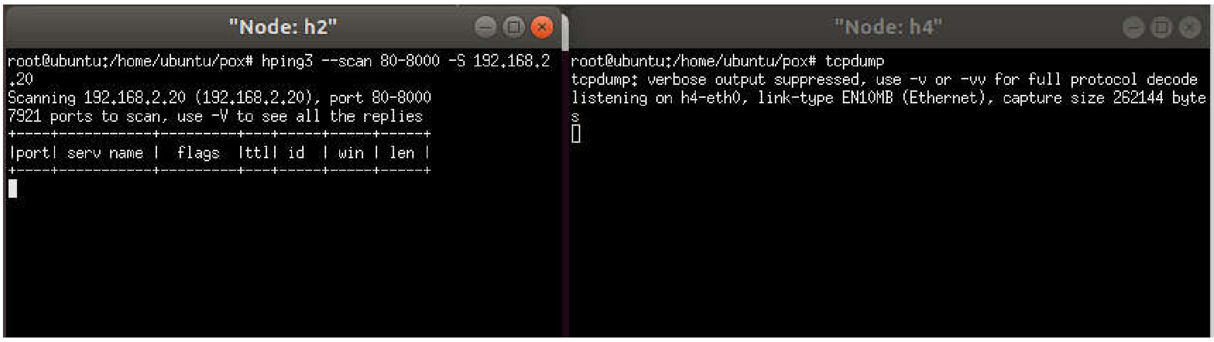


Figure 49 - CS-CNS-00101 – h2 TCP scan against h4 fails

This is how hping3 spits out its failure against h4; the result from h4 shows that no packet has ever reached the interface.

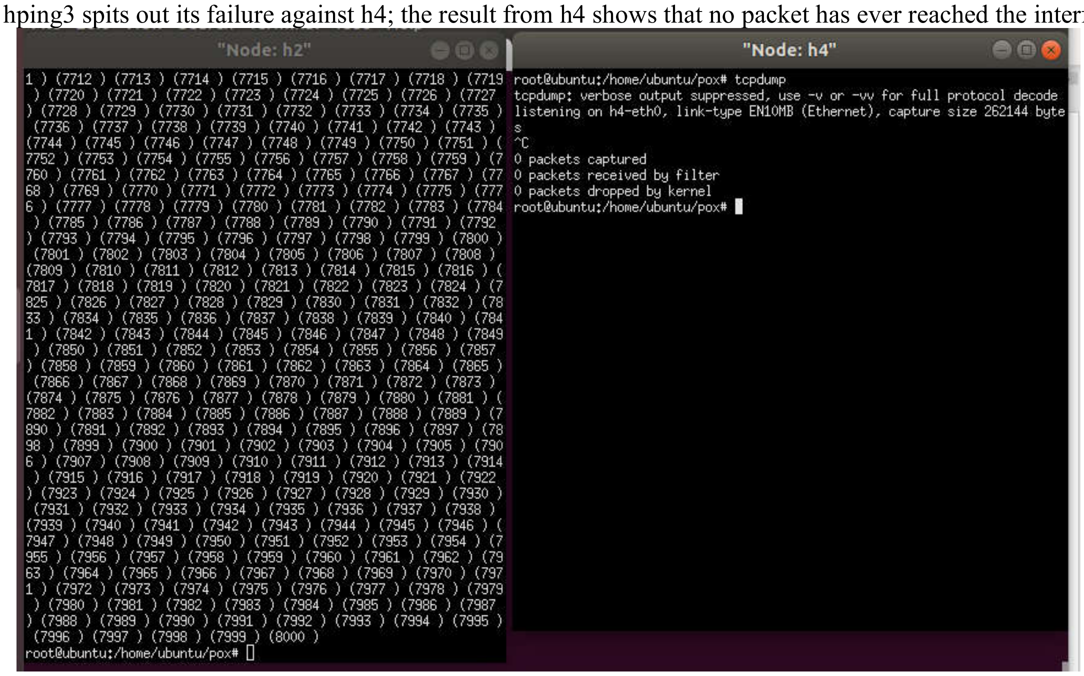


Figure 50 - CS-CNS-00101 – h2 TCP scan against h4 fails

Finally, we can dump the flows directly from the vSwitch, to verify that rules are in place (packet counts only show a few rules because the screenshot has been taken throughout restarts):

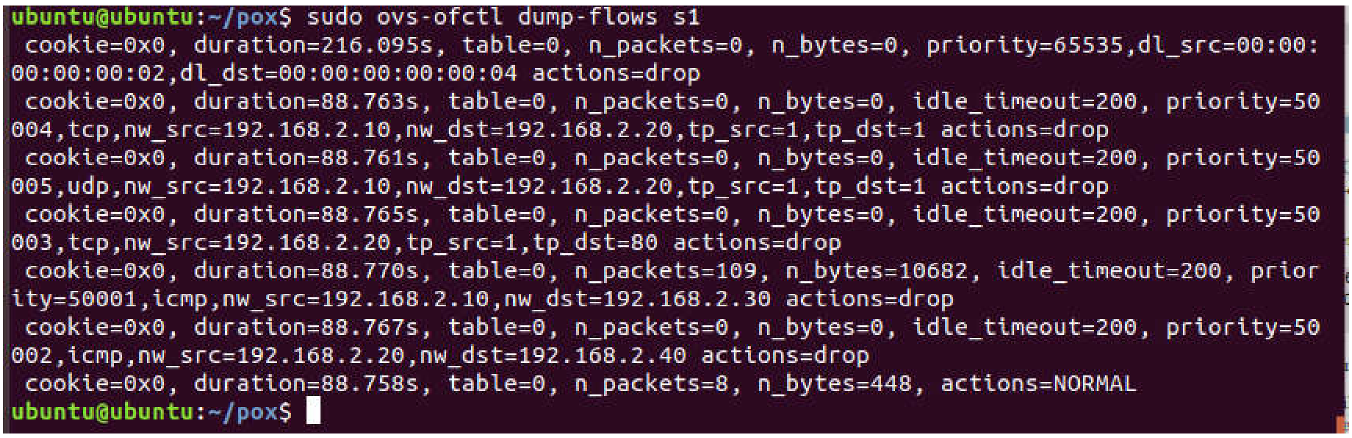


Figure 51 - CS-CNS-00101 – ovs-ofctl dump-flows s1

V. Appendix A: Files for the Lab

Please find the list of files created for this lab and mentioned throughout this document, plus their GitHub link for download.

The overall GitHub directory for the project is: https://github.com/markoer73/CSE-548/tree/main/Project%202%20-%20SDN-Based%20Stateless%20Firewall

Project-Report-2 SDN-Based Stateless Firewall.docx	https://github.com/markoer73/CSE-548/blob/main/Project%202%20-%20SDN-Based%20Stateless%20Firewall/Project-Report-2%20SDN-Based%20Stateless%20Firewall.docx
l2firewall.config	https://github.com/markoer73/CSE-548/blob/main/Project%202%20-%20SDN-Based%20Stateless%20Firewall/l2firewall.config
l3firewall.config	https://github.com/markoer73/CSE-548/blob/main/Project%202%20-%20SDN-Based%20Stateless%20Firewall/l3firewall.config
run_lab.sh	https://github.com/markoer73/CSE-548/blob/main/Project%202%20-%20SDN-Based%20Stateless%20Firewall/run_lab.sh
run_pox.sh	https://github.com/markoer73/CSE-548/blob/main/Project%202%20-%20SDN-Based%20Stateless%20Firewall/run_pox.sh

A. File Content

1) 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,1,80,tcp
4,any,any,192.168.2.10,192.168.2.20,1,1,tcp
5,any,any,192.168.2.10,192.168.2.20,1,1,udp
```

2) l2firewall.config

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

3) run_pox.sh

```
nohup ./pox.py openflow.of_01 \
  --port=6655 pox.forwarding.l2_learning \
  pox.forwarding.L3Firewall --l2config="l2firewall.config" \
  --l3config="l3firewall.config" &

nohup ./pox.py openflow.of_01 \
  --port=6633 pox.forwarding.l2_learning \
  pox.forwarding.L3Firewall --l2config="l2firewall.config" \
  --l3config="l3firewall.config" &
```

4) run_lab.sh

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

VI. References

- Linux NAT Tutorial: https://www.karlrupp.net/en/computer/nat_tutorial
- Ubuntu "Basic Iptables HOWTO": https://help.ubuntu.com/community/IptablesHowTo
- "Iptables Tutorial: Ultimate Guide to Linux Firewall": https://phoenixnap.com/kb/iptables-tutorial-linux-firewall

VIII. Table of Figures

Figure 1 - Bridged network setup in VirtualBox 1
Figure 2 - On the created bridge br0, dhclient is run 3
Figure 3 - The bridge inherits the same IP of the enp0s3 interface 3
Figure 4 - Pinging the outside world 3
Figure 5 - lab-cs-net-00006 - Lab 1-1 4
Figure 6 - lab-cs-net-00006 - Lab 1-2 4
Figure 7- lab-cs-net-00006 - Lab 1-3/1 4
Figure 8 - lab-cs-net-00006 - Lab 1-3/2 5
Figure 9 - lab-cs-net-00006 - Lab 2-1 5
Figure 10 - Ping outside 5
Figure 11 - lab-cs-net-0007 - Lab 1-1 6
Figure 12- lab-cs-net-0007 - Lab 1-2 6
Figure 13 - lab-cs-net-0007 - Lab 2-1 – topology 1 6
Figure 14 - lab-cs-net-0007 - Lab 2-1 – topology 2 7
Figure 15- lab-cs-net-0007 - Lab 2-2 - Topology 1 7
Figure 16- lab-cs-net-0007 - Lab 2-2 - Topology 2 7
Figure 17- lab-cs-net-0007 - Lab 2-3 - Topology 1 8
Figure 18- lab-cs-net-0007 - Lab 2-3 - Topology 2 8
Figure 19- lab-cs-net-0007 - Lab 3-1 9
Figure 20- lab-cs-net-0007 - Lab 3-2 9
Figure 21 - lab-cs-net-0087 - Lab 1-1/1 10
Figure 22 - lab-cs-net-0087 - Lab 1-1/2 10
Figure 23 - lab-cs-net-0087 - Lab 1-1/3 10
Figure 24 - lab-cs-net-0087 - Lab 1-2/1 10
Figure 25 - lab-cs-net-0087 - Lab 1-2/2 11
Figure 26 - lab-cs-net-0087 - Lab 2-1 11
Figure 27 - lab-cs-net-0087 - Lab 2-2 12
Figure 28 - - lab-cs-net-0087 - Extra Activity 12
Figure 29 - lab-cs-net-0009 – Lab 1-1 13
Figure 30 - lab-cs-net-0009 – Lab 1-2 13
Figure 31 - lab-cs-net-0009 – Lab 1-3 14
Figure 32 - CS-CNS-00101 – run_pox.sh 15
Figure 33 - CS-CNS-00101 – run_pox.sh running 15
Figure 34 - CS-CNS-00101 – run_lab.sh 16
Figure 35 - CS-CNS-00101 – run_lab.sh running 16
Figure 36 - CS-CNS-00101 – Lab 1 16
Figure 37 - CS-CNS-00101 – Change IP addresses 17
Figure 38 - CS-CNS-00101 – Lab 2 17
Figure 39 - CS-CNS-00101 – l3firewall.config 18
Figure 40 - CS-CNS-00101 – l2firewall.config 18
Figure 41 - CS-CNS-00101 – checking nohup.out 18
Figure 42 - CS-CNS-00101 – Changing IP addresses again 18
Figure 43 - CS-CNS-00101 – first ping from h1 to h3 gets through, then the rest are blocked 19
Figure 44 - CS-CNS-00101 – pings blocked from h2 to h4 19
Figure 45 - CS-CNS-00101 – h4 can reach web server on h3 20
Figure 46 - CS-CNS-00101 – h2 cannot reach web server on h3 20
Figure 47 - CS-CNS-00101 – h1 can reach web server on h3 21
Figure 48 - CS-CNS-00101 – h2 ping and UDP scan against h4 fail 21
Figure 49 - CS-CNS-00101 – h2 TCP scan against h4 fails 22
Figure 50 - CS-CNS-00101 – h2 TCP scan against h4 fails 22
Figure 51 - CS-CNS-00101 – ovs-ofctl dump-flows s1 22

Contents

I. Project Overview 1
II. Network Setup 1
III. Software 2
IV. Project Description 2
A. Lab "lab-cs-net-00006 OpenVirtual Switch" 3
1) Lab Assessment 1: The OVS… 4
2) Lab Assessment 2: After finishing the configuration of br0 in OVS, the machine… 5
B. Lab "lab-cs-net-00007 - Mininet" 6
1) Setting up mininet and Running mininet topology… 6
2) For each topology above, they should be able to: 6
3) Create another tree topology of depth 2 and fanout 8. 8
C. Lab "CS-NET-00008 – SDN Controller (POX)" 10
1) The POX controller can: 10
2) While connect to the controller, the mininet topology can (Please take snapshots for each task) 11
3) Extra Activity 12
D. Lab "CS-NET-00009 – Containernet" 13
1) The containernet topology can: 13
E. Lab "CS-CNS-00101" – OpenFlow Based Stateless firewall 15
1) Create a mininet based topology with 4 container hosts and one controller switches and run it. 15
2) Make the interfaces up and assign IP addresses to interfaces of container hosts. 17
3) Add new rule to l3config file for blocking ICMP traffic from source IP 192.168.2.10 and destination IP 192.168.2.30. 17
4) Add new rule to l3config file for blocking ICMP traffic from source IP 192.168.2.20 and destination IP 192.168.2.40. 17
5) Add new rule to l3config file for blocking HTTP traffic from source IP 192.168.2.20. 17
6) Add new rule to l2config file for blocking traffic from MAC address 00:00:00:00:00:02 to destination MAC address 00:00:00:00:00:04 17
7) Add new rule to l3config file for blocking tcp traffic from 192.168.2.10 to 192.168.2.20. 17
8) Add new rule to l3config file for blocking udp traffic from 192.168.2.10 to 192.168.2.20. 17
V. Appendix A: Files for the Lab 23
A. File Content 23
1) l3firewall.config 23
2) l2firewall.config 23
3) run_pox.sh 23
4) run_lab.sh 23
VI. References 24
VIII. Table of Figures 24