Hands-on Lab Description

CS-CNS-00101 – OpenFlow Based Stateless firewall

Category:

```
CS-CNS: Computer Network Security
```

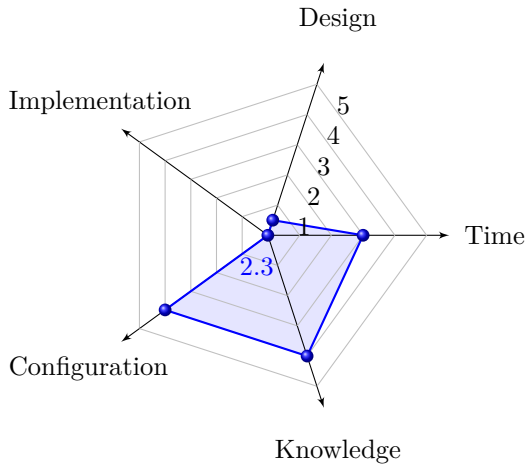
Objectives:

```
Understand the data plane and control plane framework of SDN
Understand the communication between SDN controller and switches
Implement OpenFlow based stateless firewall.
Implement simple firewall rules based on Openflow rules
```

Estimated Lab Duration:

```
Expert: 60 minutes
Novice: 180 minutes
```

Difficulty Diagram:



Difficulty Table.

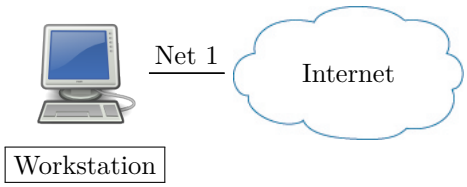
Measurements	Values (0-5)
Time	3
Design	0.5
Implementation	0
Configuration	4
Knowledge	4
Score (Average)	2.3

Required OS:

```
Linux: Ubuntu 18.04 LTS (Bionic Beaver)
```

Lab Running Environment:

```
VirtualBox https://www.virtualbox.org/ (Reference Labs: CS-SYS-00101)
```




```
1 Workstation: Linux (Ubuntu 18.04 LTS)
2 Network Setup:
  Client-side Net 1: 10.0.2.0/24
```

Lab Preparations:

```
1 Basic knowledge about open virtual switch (CS-NET-00006),
2 Mininet (CS-NET-00007),
3 POX controller(CS-NET-00008), and
4 containernet (CS-NET-00009)
5 Required packages installed: mininet, POX, OVS
```

Lab Overview

In this lab, you will emulate Denial of Service (DoS) attacks, which can target various components in the SDN infrastructure, in an SDN networking environment. You will need to set up an SDN-based firewall networking environment based on mininet, containernet, POX controller, and Over Virtual Switch (OVS). To mitigate DoS attacks, you will need to develop a mitigation strategy by using SDN-based firewall to counter the implemented DoS attacks.

To complete the lab, you will submit a sequence of screenshots and corresponding illustrations to demonstrate how they fulfilled the firewall's packet filtering requirements. Your work will be evaluated based on the completeness of implementing firewall filtering rules to demonstrate how to successfully counter the implemented DoS attack.

In summary, students will do:

- Setup the OpenFlow-based SDN environment using miniment or containernet.
- Setup POX OpenFlow controllers.
- Implement flow-based filtering rules based on flow control requirements. SDN controller based L2 and L3.
- Implement flow-based firewall SDN testing Firewall rules for regulating network traffic.
- Test OpenFlow-based SDN Firewall Implementation.

In the following tasks, task 1 and task 2 will guide you to set up the system environment and following a step-by-step procedure to practice a simple firewall, respectively. By finishing the task 2, you will need to setup an SDN-based firewall based on the requirements provided in task 3.

Task 1.0 Preparation of setting up lab environment

Suggestions for Task 1:

1. Review and exercise the following labs CS-SYS-00001 (Linux tutorial), CS-NET-00001 (Network setup) and CS-NET-00002 (Gateway setup) before you do Task 1.1.
2. Review and exercise the the following labs: Open vSwitch and Basic Setup (CS-NET-00006), *mininet* (CS-NET-00007), POX (CS-NET-00008), and *containernet* (CS-NET-00009) before you do Task 2.1.

Note that the services (Mininet, OVS and POX) may have already been setup on your server. You need to verify if these servers are setup properly to conduct your required tasks.

Before starting the lab, you need to check several software components and see if they have been set up properly. They are:

- Check if python is installed

```
$       python --version
```

- Check if python3.x is installed

```
$       python3 --version
```

- check if *mininet* is installed (or check CS-NET-00007 for *mininet* installation and setup)

```
$       mn --version
$       sudo mn --test pingall
```

This will create a temporary hosts and switch and pings to all hosts that it created. If *mininet* is installed properly, it will execute and clean all the created hosts later and exit successfully.

- Check installation of pox. Go to directory where POX is installed, e.g., a common source code of POX is installed in directory */home/ubuntu/pox*. Here, we use $POX_DIR represents the POX directory in your system.

```
$       cd /home/ubuntu/pox % depends on where the pox folder is created.
$       ./pox.py -verbose forwarding.hub
```

This will start a pox session and prints DEBUG messages as presented below. This indicates that POX is running fine. To exit press *Ctrl-C*.

```
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/Nov 7 2019 10:07:09)
DEBUG:core:Platform is
    Linux-5.3.0-40-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
```

- Check OVS installation

```
$       ovs-vsctl --version
```

It should display OVS version and details when it was installed, something like below:

```
ovs-vsctl (Open vSwitch) 2.9.5
DB Schema 7.15.1
```

Task 2.0 Test Openflow-based SDN Firewall

A firewall is used as a barrier to protect networked computers by blocking malicious network traffic generated by attacker. Internal traffic is not seen and cannot be filtered by a traditional firewall. An SDN firewall works both as a network traffic flow filter and a policy checker. An example of OpenFlow-based SDN setup is shown in Figure CS-CNS-00101.2. The first packet of a flow goes through the controller and is is checked by by the SDN controller based on it allowed or disabled network traffic policy, in which the subsequent packets of the flow directly match the flow policy defined in the controller. The firewall policy is centrally defined and enforced at the controller.

Security policies for a network are defined centrally at the controller. The firewall converts these policies into flow rules which are installed in the network by flow programming application. Firewall will have a centralized view of a network and it can inspect traffic to detect and drop unwanted network traffic. Unlike traditional firewalls that scan and filter every packet coming and going through it, an SDN-based firewall analyzes the first packet of each flow and installs rules for the rest of that flow in relevant switches.

OpenFlow provides mechanism to communicate to the controller from switches and vice-versa. A firewall application based on OpenFlow 1.0 and POX controller is used in this lab to demonstrate lab work in the SDN environment.

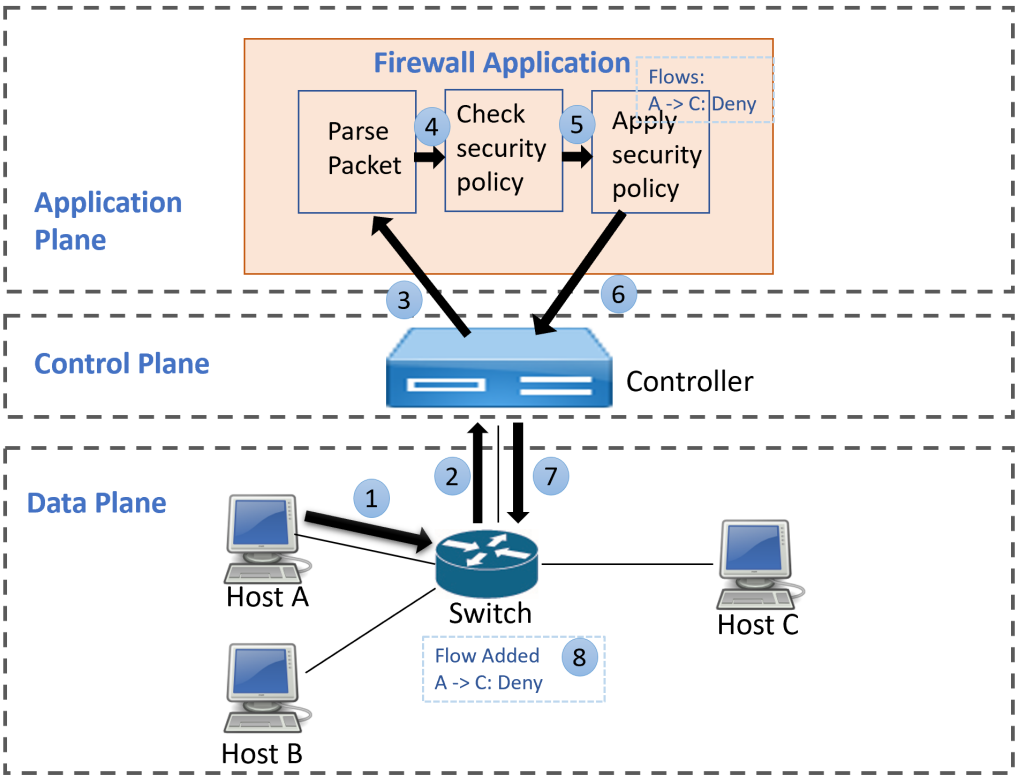


Figure CS-CNS-00101.2
Network topology for SDN.

OpenFlow

Control plane in SDN makes the forwarding decisions and applies policies to entire network. This information sharing between SDN controller and switches takes place using APIs provisioned by OpenFlow protocol. OpenFlow enabled switches maintain the flow routing policies in flow tables. A typical rule presents in a flow-table has 3 different field sets for packet handling:

- Match,
- Action, and
- Statistics.

The match set allows packet filtering based on header fields. After a successful match, the packet undergoes respective actions. As per the OpenFlow standard, the first packet of a network flow is typically sent to the controller, which then inspects the packet headers and decides actions to be taken for rest of the flow. This action can be either to drop or forward. The central visibility in SDN is of great advantage as the controller can extrapolate the future impact of traffic on any other node in the network.

SDN Controller

In an SDN, a controller acts as a heart of network operating system. Hardware-based services of traditional networks like firewall and load balancer run as software applications within a controller.

Task 2.1 Getting source code

1. Now, you can download lab resource files by using following command. (Pre-requisite: Check if wget is installed using command 'wget –version'. If wget is not installed, install it using 'sudo apt install wget')

```
$       wget https://gitlab.thothlab.org/thoth-group/ThoThLabResource/raw/master/
          lab-cs-cns-00101.zip
$       unzip lab-cs-cns-00101.zip
```

```
     Source code files are located in the folder 'lab-cs-cns-00101'.
```

2. We need to use this firewall application with POX controller. To do so, copy *L3Firewall.py* file into the pox folder such as *./pox/pox/forwarding*. Here we assume POX_DIR is directory where POX source code is present, e.g., If you have installed POX in */home/ubuntu/*, then your $POX_DIR is */home/ubuntu/pox*.

```
$       sudo cp lab-cs-cns-00101/l2firewall.config $POX_DIR/. % copy the layer 2
          config file.
$       sudo cp lab-cs-cns-00101/l3firewall.config $POX_DIR/. % copy the layer 3
          config file.
$       sudo cp lab-cs-cns-00101/L3Firewall.py $POX_DIR/pox/forwarding/. % copy the
          forwarding file.
```

First, let's dive into the example *L3Firewall* application, and please refer to Figure CS-CNS-00101.2 as the reference example for the communication flow between hosts, controller and switch.

POX application starts off with launching the class defined with additional parameters, if any. In this case, the codes register the class "learning" with input configuration file in "l2config" and "l3config". User can provide these files using options *–l2config* and *–l3config*, if not provided, program takes the default files provided in the application.

```
def launch (l2config="l2firewall.config",l3config="l3firewall.config"):
   parser = argparse.ArgumentParser()
   parser.add_argument('--l2config', action='store', dest='l2config',
              help='Layer 2 config file', default='l2firewall.config')
   parser.add_argument('--l3config', action='store', dest='l3config',
              help='Layer 3 config file', default='l3firewall.config')
   core.registerNew(Firewall,l2config,l3config)
```

Learning class consists of basic input parameters to start the application. Here, *l2config* and *l3config* file consists of user specified firewall rules. That will get added in the *fwconfig* set which will contain individual rules.

Now, this initialization calls *_handle_connectionUp* as part of POX initialization architecture. POX connection is set here and basic layer 2 rules are installed in firewall.

```
def _handle_ConnectionUp (self, event):
   self.connection = event.connection
   for (source, destination) in self.disbaled_MAC_pair:
      message = of.ofp_flow_mod() # OpenFlow massage. Instructs a switch to install a
          flow
      match = of.ofp_match() # Create a match
      match.dl_src = source # Source address
      match.dl_dst = destination # Destination address
      message.priority = 65535 # Set priority (between 0 and 65535)
      message.match = match
      event.connection.send(message) # Send instruction to the switch
   log.debug("Firewall rules installed on %s", dpidToStr(event.dpid))
```

All incoming packets are handled by *_handle_packetIn* function. This will define what to do with incoming packets. From the incoming packet flow, match fields are retrieved and packets are handled based on their protocol type, e.g., APR packets are handled differently than IP packets.

```
def _handle_PacketIn(self, event):
      packet = event.parsed
      match = of.ofp_match.from_packet(packet)

      if(match.dl_type == packet.ARP_TYPE and match.nw_proto == arp.REQUEST):
        self.replyToARP(packet, match, event)

      if(match.dl_type == packet.IP_TYPE):
        ip_packet = packet.payload
        print "Ip_packet.protocol = ", ip_packet.protocol
        if ip_packet.protocol == ip_packet.TCP_PROTOCOL:
         log.debug("TCP it is !")
        self.replyToIP(packet, match, event, self.rules)
```

Once the packets are identified and separated according to protocols, they are sent to *InstallFlow*. This is main part of code where OpenFlow rule is formed based on rules in *config* file provided.

There are two types of timer values used for the openflow flows. The idle timeout value determines how long a flow rule can be present in the flow table if there is no network traffic associated with that flow. In this case, for each flow, if there is no packets hitting a rule, that rule is removed from the device. Hard timeouts are the absolute values for time after which the rule is removed from the device. The switch maintains an entry time for each flow rule. The default timeout values for a flow is zero unless specified. In that case, an entry remains in the table permanently unless explicitly deleted.

```
def installFlow(self, event, offset, srcmac, dstmac, srcip, dstip, sport, dport,
    nwproto):
      msg = of.ofp_flow_mod()
      match = of.ofp_match()
      if(srcip != None):
         match.nw_src = IPAddr(srcip)
      if(dstip != None):
         match.nw_dst = IPAddr(dstip)
      match.nw_proto = int(nwproto)
      match.dl_src = srcmac
      match.dl_dst = dstmac
      match.tp_src = sport
      match.tp_dst = dport
      match.dl_type = pkt.ethernet.IP_TYPE
      msg.match = match
      msg.hard_timeout = 0
      msg.idle_timeout = 7200
      msg.priority = priority + offset
      event.connection.send(msg)
```

The input configuration file *l2firewall.config* file contains following rules for each line in the configuration file:

```
Priority, Source_MAC, Destination_MAC
```

The configuration file is specified as:

```
id,mac_0,mac_1
1,00:00:00:00:00:01,00:00:00:00:00:02
```

The input configuration file *l3firewall.config* file contains following bi-directional rule fields for each line in the configuration file:

```
Priority,  Source_MAC, Destination_MAC, Source_IP,  Destination_IP,
    Source_PORT, Destination_PORT,   Network_PROTOCOL
```

The configuration file is specified as:

```
priority,src_mac,dst_mac,src_ip,dst_ip,src_port,dst_port,nw_proto
1,any,any,10.0.0.1,10.0.0.2,1,1,icmp
2,any,any,10.0.0.2,10.0.0.1,1,1,icmp
3,any,any,10.0.0.1,10.0.0.2,any,any,tcp
4,any,any,10.0.0.2,10.0.0.1,any,any,tcp
5,any,any,10.0.0.1,10.0.0.2,any,any,udp
6,00:00:00:00:00:03,any,10.0.0.3,any,any,any,tcp
```

Note that POX takes the defualt IP ranges from "10" network if no specified IP address is given for each host, in which it assigns 10.0.0.1 to *h1* ... 10.0.0.9 to *h9*. This firewall configuration file consists of firewall rules for *BLOCKING* traffic. In *l3firewall.config* file:

```
The first rule BLOCKS ICMP packets from source IP address 10.0.0.1 to
    destination IP address 10.0.0.2 and vice versa
The second rule BLOCKS ICMP packets from source IP address 10.0.0.2 to
    destination IP address 10.0.0.3 and vice versa
The third rule BLOCKS TCP packets from source IP address 10.0.0.1 to
    destination IP address 10.0.0.2 and vice versa
The fourth rule BLOCKS TCP packets from source IP address 10.0.0.2 to
    destination IP address 10.0.0.1 and vice versa
The fifth rule BLOCKS UDP packets from source IP address 10.0.0.1 to
    destination IP address 10.0.0.2 and vice versa
The sixth rule BLOCKS TCP packets from source MAC address 00:00:00:00:00:03
    and IP address 10.0.0.3 and vice versa
```

3. Run POX controller, where here we assume *$POX_DIR* is the POX directory where pox source codes present:

```
$         cd $POX_DIR
$         sudo ./pox.py openflow.of_01 --port=6655 pox.forwarding.l2_learning
            pox.forwarding.L3Firewall --l2config="l2firewall.config"
            --l3config="l3firewall.config"
```

Here POX controller is invoked by *./pox.py* command and we run *l2_learning.py* application and *L3Firewall.py* file from POX controller. All the forwarding applications has to be stored in */pox/pox/forwarding* directory. To give relative path from directory where POX binary is present, we follow convention *pox.forwarding.L3Firewall*

4. Run *mininet* using *containernet*. Now, you can open a new terminal window, run the following command:

```
$         sudo mn --topo=single,9 --controller=remote,port=6633
            --controller=remote,port=6655 --switch=ovsk --mac
```

It will create a *mininet* environment in *containernet* with 9 *containernet* hosts, one OVS switch and two remote controllers. Option –mac will assign small, unique and fixed set of mac address based on host id. It will remain constant after every run.

We have started POX controller earlier, and thus, POX will connect to this *mininet* environment as a remote controller. The *mininet* topology should look like Figure CS-CNS-00101.3.

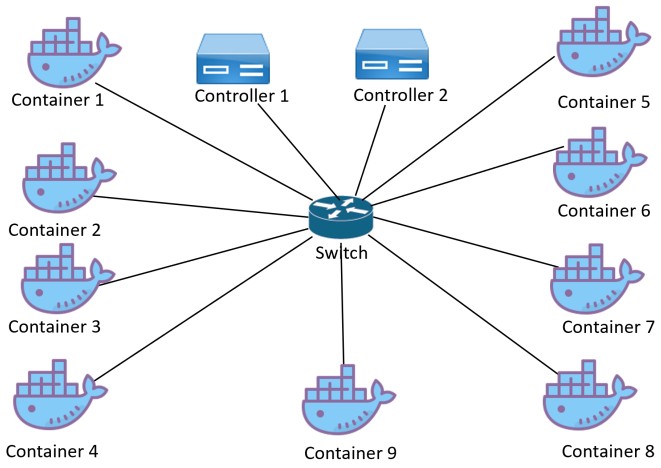


Figure CS-CNS-00101.3
Network topology for SDN.

Following table CS-CNS-00101.2 depicts the mapping between MAC addresses and respective IP addresses in specified *mininet* topology. Since we have enabled *–mac* option in *mininet* topology, MAC address fill be fixed for each *containernet* host in every run.

Container Host	Layer 2 Address (MAC address)	Layer 3 Address (IP address)
Container host h1	00:00:00:00:00:01	10.0.0.1
Container host h2	00:00:00:00:00:02	10.0.0.2
Container host h3	00:00:00:00:00:03	10.0.0.3
Container host h4	00:00:00:00:00:04	10.0.0.4
Container host h5	00:00:00:00:00:05	10.0.0.5
Container host h6	00:00:00:00:00:06	10.0.0.6
Container host h7	00:00:00:00:00:07	10.0.0.7
Container host h8	00:00:00:00:00:08	10.0.0.8
Container host h9	00:00:00:00:00:09	10.0.0.9

Table CS-CNS-00101.2
Mapping of MAC addresses with IP addresses in respective *containernet* hosts

Note that in the following subsections, you can use two ways to test your virtual networks from the command-line of a host:

1. You can specify from which host to initiate a command. For example:

```
containernet> h1 ping h3  % ping from h1 to h3
containernet> h1 hping3 -c 5 h2 -V --tcp-timestamp % use hping3 to sent tcp
    packets to h2
```

2. you can start a host terminal and use network configuration to perform the test. For example, the following command is to start an x-terminal of host h1:

```
containernet> xterm h1
```

Once the x-terminal is started you can run the above commands just like in a real host command-line, such as:

```
$    ping 10.0.0.3   % ping from h1 to h3 (10.0.0.3)
$    hping3 -c 5 10.0.0.2 -V --tcp-timestamp % use hping3 to sent tcp packets to h2
       (10.0.0.2)
```

Task 2.2 Verifying working of the firewall

In this lab, Open vSwitch is used. The new rules added based on the match and controller signals, can be viewed from OVS switches. To do so, you have to dump flow entries at OVS switches and use OpenvSwitch commands to verify and check the rules getting added in the Switches.

When you run *L3Firewall* application from POX and create a *mininet* environment, an SDN is formed with *mininet* containers as hosts, OVS as a data plane switch to handle switching functionalities and POX to handle control plane functionality.

Open another terminal to run OVS commands that will verify the functionality of POX.

1. Check *openvswitch* database configuration

```
$          sudo ovs-vsctl show
```

2. list ports for specific switch

```
$          sudo ovs-vsctl list-ports s1
```

3. Check OpenFlow switch details

```
$          sudo ovs-ofctl show s1
```

4. Check OpenFlow flow details

```
$          sudo ovs-ofctl dump-flows s1
```

We have provisioned a virtual switch called OVS using mininet. ovs-ofctl provides a commandline tool to manange and control openflow switches. ovs-ofctl prints the output of all the flows in the same order the switch sends them.

Consider following example output of an openflow entry:

```
        cookie=0x0, duration=125.376s, table=0, n_packets=4, n_bytes=392,
            priority=65535,dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:04
            actions=drop
```

The output consists of a few important parameters which are essential in our case.

- **cookie**: This field can be added while adding or deleting a flow entry to the openflow rules. We do not require this field to be updated, hence it uses default value as 0x0.
- **Duration**: This includes the time in seconds in which the flow entry resides in the table.
- **table**: We can specify the table number here as any number between 0 to 255.
- **n_packets**: This specifies the number of packets that matched with this flow entry. In this particular case, total 4 packets matched with this entry.
- **n_bytes**: This specifies the number of bytes that matched with this flow entry. In this particular case, total 392 bytes matched with this entry.

- **priority**: Priority is very imporant entry in case of flows. A higher value will match before a lower value. Priority values van be any value between 0 to 65535. If we want a rule to get precendence in case if there could be multiple match for a packet, then the packet will follow the rule which has higher priority value. If this value is not specified, the default value would be 32768.
- **dl_src**: These are the matching criteria provided in a rule file of firewall (either l2firewall.config or l3firewall.config). This specifies source mac address.
- bf dl_dst:These are the matching criteria provided in a rule file of firewall (either l2firewall.config or l3firewall.config). This specifies destination mac address.
- **action**: This decides what to do with a packet that matches this specified rule.

After sending the packets, if the incoming packet matches with any of the specified flow, the packet count in that specific flow entry will increase. This indicates that there is a packet matching with the criteria specified in the openflow rule. This is how we know if the rules we want to install, are working in OpenFlow switch.

Note that these flows have a timeout mechanism, hence a particular flow will not be seen all the time.

In the *containernet* terminal, send ICMP packets from hosts to check the firewall functionality.

(in *containernet* terminal:)

```
containernet> h1 ping h2
```

This will send ICMP requests from *containernet* host *h1* to *containernet* host *h2*. Since we have a rule in *l3firewall.config* file for blocking traffic from IP address 10.0.0.1 (this IP address belongs to *containernet* host *h1*) to destination IP address 10.0.0.2 (this IP address belongs to *containernet* host h2), it will add the rule from *config* file to OpenFlow rule in switch.

This addition of rule can be verified from OVS commands which will dump flows at switch *s1*.

As you start ping from host *h1* to h2, in OVS terminal you can check the rule gets added with source address and destination address as provided in *l3firewall.config* file. You can also see number of packets sent from *h1* to *h2*

All the packets sent and received from IP addresses given in the *l3firewall.config* file are *BLOCKED* by the firewall application. Any other network traffic is *ALLOWED* by the firewall.

To demonstrate this, try to ping from host *h1* to host *h9*. There is no rule added to block network traffic from host *h1* (IP: 10.0.0.1) to host *h9* (10.0.0.9).

(in *containernet* terminal:)

```
containernet> h1 ping h9
```

Packets will be sent from *container* host *h1* to h9. To check this, run OVS command in ovs-terminal:

```
$ sudo ovs-ofctl dump-flows s1
```

To verify firewall rule for blocking TCP packets specified in rule number 3, send tcp packet using hping3 tool.

(In *containernet* terminal window:)

```
h1 hping3 -c 5 h2 -V --tcp-timestamp
```

5. Print port statistics

```
$ sudo ovs-ofctl dump-ports-desc s1
```

6. Check status of the flow tables

```
$          sudo ovs-ofctl dump-tables s1
```

7. Check all hidden flows for particular switch

```
$          sudo ovs-appctl bridge/dump-flows s1
```

Lab Assessment (100 points)

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

1. (5 points) 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.
2. (5 points) 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.
3. (15 points) 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. (15 points) 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. (15 points) Add new rule to l3config file for blocking HTTP traffic from source IP 192.168.2.20.
6. (15 points) 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. (15 points) Add new rule to l3config file for blocking tcp traffic from 192.168.2.10 to 192.168.2.20.
8. (15 points) Add new rule to l3config file for blocking udp traffic from 192.168.2.10 to 192.168.2.20.