IMPLEMENTING PACKET FILTERING ON SOFTWARE DEFINED NETWORKS USING MININET AND POX

Co-authored by Akash Danabhai Chavda

Dale Lindskog

Pavol Zavarsky

Project report

Submitted to the Faculty of Graduate Studies,

Concordia University of Edmonton

in Partial Fulfillment of the

Requirements for the

Final Research Project for the Degree

MASTER OF INFORMATION SYSTEMS SECURITY MANAGEMENT

Concordia University of Edmonton

FACULTY OF GRADUATE STUDIES

Edmonton, Alberta

April 2020

IMPLEMENTING PACKET FILTERING ON SOFTWARE DEFINED NETWORKS USING MININET AND POX

Akash Danabhai Chavda

Implementing Packet Filtering on Software Defined Networks using Mininet and POX

Akash Danabhai Chavda
Information Systems Security Management
Concordia University of Edmonton
Edmonton, AB, Canada
achavda@student.concordia.ab.ca

Dale Lindskog
Information Systems Security Management
Concordia University of Edmonton
Edmonton, AB, Canada
dale.lindskog@concordia.ab.ca

Pavol Zavarsky
Information Systems Security Management
Concordia University of Edmonton
Edmonton, AB, Canada
pavol.zavarsky@concordia.ab.ca

***Abstract* – This research paper investigates security components of the POX controller and describes an additional component, developed by the author, that provides enhanced filtering functionality.**

Keywords – SDN, POX, Mininet, OpenFlow, CLI, Packet Filtering

I. Introduction

Software Defined Network (SDN) is a paradigm which enables the separation of data plane and control plane by means of centralized control over the data plane, and therefore the network topology, by allowing the data plane to be monitored remotely through control plane. It can also be explained as centralized control over the topology through control the plane. The control plane consists of those elements that monitors the switches of SDN network topology, and the data plane consists of forwarding elements such as switches. Software Defined Networking enables network programmability, facilitated by the OpenFlow architectural paradigm. "The OpenFlow protocol is an architectural approach that enables SDN to separate the control plane from data plane, and abstract a forwarding path that enables a controller, connected by a secure channel to network devices, to address network functions" [6].

There are several tools available in the market to create Software Defined Networking environment. Mininet is a network simulator which allows, with use of only one computer, quick prototyping of a large virtual infrastructure network [1]. A controller is a component which can, from a single point of management, modify the topology, view the topology, and deploy security. POX, NOX, OpenDayLight and some other controllers are examples. In this paper, the POX controller is being used to implement security features on the SDN topologies created by Mininet. The POX controller has a number of components which provide various of functions useful in managing topology centrally. This paper has focused on components that provide packet filtering functionality.

POX is currently distributed with components that enable layer 2 filtering and specifically the ability to block the incoming traffic based on MAC addresses. This paper describes the addition of layer 3 filtering, and specifically filtering based on seven out of the twelve attributes specified in the OpenFlow protocol, namely MAC source address (dl_src), MAC destination address (dl_dst), IP source address (nw_src), IP destination address (nw_dst), TCP or UDP source port (tp_src), TCP or UDP destination port (tp_dst) and transport layer protocol (nw_proto)**.** In addition, this implemented component which is developed by author by revising the script of [20], can block or allow the traffic based on the direction. It also implements both blacklisting and whitelisting.

This paper is organized into four sections. Section II presents related work and technologies; Section III describes experimental environment, software development, results. The paper concludes with Section IV: Contribution to Knowledge.

II. Related Work and Technologies

In traditional networks, network configuration is on a per node basis, and this is time consuming and error prone. Network Virtualization (NV), Network Function Virtualization (NFV), and Software Defined Networking (SDN) address this issue. Although these technologies operate differently, they all facilitate programmable networks [2].

A. NV, NFV,SDN

Researchers from [2] have elaborated Network Virtualization as linking element of two network segments in a logical way. This virtual network can be created by combining multiple physical networks or software-based networks. This virtual network not only combines multiple networks, but it can also create separate independent network from a physical network.

NFV virtualizes network layers 4 through 7, such as firewalls or IDPSs or even load balancing. The functions that, traditionally, need specialized hardware to run, are being emulated by NFV to use in virtualization technology [14]. Since the NFV only replaces the proprietary middle box and networks devices with virtual function, the centralized management of topology cannot be achieved only through NFV [14].

The main characteristic of SDN is separation of data plane and control plane which makes the network programmable. The control plane indicates the network what to do while the data plane sends packets to specific destinations. Unlike traditional networks, SDN relies on switches that can be programmed by the controller.

OpenFlow is a standardized protocol to control SDN switches and an implementation of these switches is known as OpenVswitch [2]. An OpenFlow controller in SDN allows us to periodically collect information from network devices concerning their status, and can send commands instructing the network about how to handle traffic [3]. Controllers that use the OpenFlow protocol are POX, NOX, Beacon, OpenDayLight, Ryu and Floodlight. These controllers are developed in different programming languages such as Python, Ruby, Java, etc. The information gathered by OpenFlow protocol, is being passed to controller operating system for monitoring [3]. Network Function Virtualization becomes more feasible with the introduction of SDN, where the virtualized network functions can move around the network dynamically with SDN's network reconfiguration functionality. "NV and NFV can work on existing networks because the reside on servers and interact with traffic sent to them while the SDN requires a new network concept where data and control planes are separated" [2].

The layered architecture of Software Defined Networks allows users to program the controller. It has three architecture layers, namely the Infrastructure layer, Controller layer, and Application layer.

The infrastructure layer consists of network elements. In this layer switches and hosts are being placed that is called as network elements. The controller layer consist element that instructs switches placed in infrastructure layer. Controller monitors the topology and can make necessary changes to topology. The application layer enables users for network programming. In other words, the network administrator sits at this layer and passes the instruction to controller through programs which can be developed in different programming languages. The southbound interface acts as a communication link between infrastructure layer and controller layer. The northbound interface pass instruction to controller layer received from application layer.

Separation of control plane from data plane enables a controller, with a secure channel connecting network devices, to directly manipulate network functions. In addition, OpenFlow switches store cached information in form of flow tables that include detailed information about the traffic streams in the switch which can be analyzed to manage flow [4]. The logically centralized view of SDN topologies is achieved via open interfaces and abstraction of lower-level functionalities, and which transforms the network into a programmable platform to dynamically adapt behavior of SDN topologies [5]. SDN architecture carries several benefits with it which are listed [4][6]: SDN architecture allows faster implementation of new and enhanced services; It enables deep viewing into all network flows through an OpenFlow switch.

This analysis enables enhanced direct management and control over those flows, and therefore, enhanced network service and infrastructure management and control. It also allows for many more options for dynamic provisioning and automated dynamic response to conditions based on flow analytics. It provides substantially improved options for creating customizable network services and infrastructure.

B. *OpenFlow Protocol*

"Programmable networking using SDN is generally based on the OpenFlow protocol, an architectural approach that separates the control plane from the data plane, abstract the forwarding path, and enables a controller, connected by a secure channel to network devices to address network functions" [6]. An OpenFlow switch has a flow table that stores cached information on traffic streams, and this information can be interrogated and analyzed at a highly granular level so that the results can initiate a response to control the behaviors of specific individual flows supported by the switch [7]. The technique for analysis of flow, that is monitor the flow through controller to detect flow and patterns and respond to the generated result, was primarily developed and deployed for L2 services, and then was extended to both L2 and L3 services [6]. The OpenFlow controllers and OpenFlow switches are connected through an interface called OpenFlow channel, through which switches are configured and managed by controller, events are received from the switches and packets are sent out to the switches [8]. The main type of messages sent through this channel are threefold [8]:

- Controller-to-switch messages are sent by the controllers to directly manage and inspect the state of the switch
- Asynchronous messages are sent by the switch to update the controller about network events and changes to the switch state
- Symmetric messages can be initiated by either the switch or the controller and sent without solicitation

C. *SDN and Packet Filtering Scripts on POX Controller*

In Software Defined Networking, network topologies have three main components, namely switches, hosts and the controller. Packet filtering rules can be implemented at switches or at the controller. The reason to select switches as filtering elements is to reduce workload on the controller, and therefore reduce latency.

There are two approaches to place firewall rules at the switch: the reactive approach and the proactive approach. [13] have used the reactive approach at the switch level to implement firewall rules using twelve match fields defined in the OpenFlow standard. These match fields are listed in Table II. With the reactive approach, packets are handled directly as they come into the switch, while for the proactive approach, rules are pre-installed in the switch's flow table.

Software Defined Networking is a vast field and many packet filtering scripts have been implemented using different controllers and topology simulators. This research has focused on packet filtering scripts that have been implemented using POX controller and written in Python programming language. This research evaluated a number of packet filtering scripts found on GitHub that facilitate users to implement packet filtering capabilities on their SDN topology. These packet filtering scripts are evaluated based on the attributes they use to filter traffic. In addition, the detailed comparison of these scripts is represented in TABLE I.

Most of the packet filtering scripts used files that contains comma separated values such as source address, destination address, and so on, which are considered as preliminary attributes to filter a packet. The user needs to modify this file every time the new traffic flow hit the switches, and this is no feasible in the real world. In addition, these rules are for blocking traffic between specified source and destination address, that is, they implement default permit policy. Moreover, most of the POX scripts examined in this research do not change their security policies as new traffic flows appear on switch. Instead, the rule must be manually inserted in the file. To load this CSV file, a Python script needs to be run again, and that is not exactly a programmable network. There are other ways through which rules can be added to the flow table of switches, and that is by creating lists in the Python script itself. These lists contain values of attributes to be passed to the function at the time of execution.

TABLE I. Comparison of different Firewall Python Script on POX

Reference	*Behavior*	*Addition of Rules to Switch at CLI*	*Filtering Functionality*						
			Source MAC	*Destination MAC*	*Source IP*	*Destination IP*	*Source Port*	*Destination Port*	*Protocol*
[15]	Blacklisting	No	Yes	Yes	No	No	No	No	No
[16]	Blacklisting	No	Yes	Yes	No	No	No	No	No
[17]	Blacklisting	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
[18]	Blacklisting	No	No	No	Yes	Yes	No	No	Yes
[19]	Blacklisting	No	Yes	Yes	No	No	No	No	No
[20]	Blacklisting	No	Yes	Yes	Yes	Yes	No	No	No
[21]	Blacklisting	No	No	No	Yes	Yes	Yes	Yes	No
[22]	Blacklisting	No	Yes	Yes	Yes	Yes	No	No	No
[23]	Blacklisting	No	No	No	Yes	Yes	Yes	Yes	Yes
[24]	Blacklisting	No	Yes	Yes	No	No	No	No	No
[25]	Blacklisting	No	Yes	Yes	Yes	Yes	Yes	Yes	No
[26]	Blacklisting	No	No	No	Yes	Yes	No	Yes	No
[27]	Blacklisting	No	Yes	Yes	No	No	No	No	No
[28]	Blacklisting	No	Yes	Yes	No	No	No	No	No

TABLE I represents the fact that no evaluated scripts have the capability of adding rules at the command line interface (CLI). The addition of rules at the CLI is essential part of making a controller capable of filtering traffic as new traffic flow hits the switch. TABLE I also represents that many scripts make use of a small number of attributes supported by OpenFlow protocol. The Python script implemented by [17] considered all seven attributes to filter traffic, but that script implements only blacklisting.

D. *Related Research on OpenFlow Protocol*

Researchers from [10] proposed that OpenFlow switches can be used as packet filters by considering the packet attributes such as source IP, destination IP, source port, destination port and type of packet (TCP/UDP). They allowed ICMP and ARP packets to perform communication tests in a topology they created using the Mininet network emulator [10]. They wrote a Python script that reads from the CSV file, and therefore that user will have to modify that
file manually whenever need the arises to allow a flow from a new incoming packet.

It is true that an OpenFlow controller does not implement security by itself. But a controller can be programmed in a way that it can secure the network topology controlled by it. Researchers in [8] proposed a secure OpenFlow Ryu controller in which basic packet filtering rules are implemented that inspect properties of each packet and use a Bayesian network classifier for detecting and filtering unusual packet flows or DDoS attacks.

E. *Mininet*

Mininet is an open source network simulator that allows user to create network environments for experiments. Typically, it is used for creating SDN environments and testing. The latest version of Mininet comes with pre-installed controller named POX. It allows quick prototyping of large virtual networks with the use of only one computer.

It enables users to create virtual prototypes of scalable networks based on protocol such as OpenFlow, using primitive virtualization operating system [1]. Researchers from [1] have listed features of Mininet as described below:

- It provides a simple and cheap way for testing networks for OpenFlow application development.
- It allows multiple researchers to independently work on the same network topology.
- It allows the testing of a large and complex topology, without even the necessity of a physical network.
- It includes tools to debug and run tests across the network.
- It supports numerous topologies and includes a basic set of topologies.
- It provides simple Python APIs for creating and testing networks.

F. *POX*

The POX controller is a Python-based controller that allows user to control a network topology created by Mininet. POX can also provide some security features in the given topology with the help of components. Components can be understood as functionalities of the POX controller. POX has a number of components for different functionalities, but there are few components which provide or are related to security of network topology. POX allows user to develop their own components according to their need. POX has evolved over the years and has five versions/branches: Angler, Betta, Carp, Dart, and Eel.

The EEL is the latest branch of POX controller and is still under development. POX supports OpenFlow 1.1. This paper has made use of the EEL branch of the POX controller.

TABLE II. SDN Firewall Attributes in OpenFlow [13]

Attribute	Type	Description	Example
in_port	Integer	It denotes the switch port number the packet arrived on.	Int: 30
dl_src	String	Ethernet Source address	String: '00:00:00:00:00:01'
dl_dst	String	Ethernet Destination address	String: '00:00:00:00:00:02'
dl_vlan	Integer	Indicates VLAN ID	Int: 2
dl_vlan_pcp	Integer	Indicates VLAN priority	Int: 0
dl_type	String	Indicates Ether type	IP_TYPE, ARP_TYPE
nw_tos	Integer	Indicates TOS/DS bits	Int: 0
nw_proto	String	Indicates IP protocol	ICMP_PROTOCOL, TCP_PROTOCOL, UDP_PROTOCOL
nw_src	String	Indicates IP Source address	String: '10.0.0.1'
nw_dst	String	Indicates IP Destination address	String: '10.0.0.2'
tp_src	Integer	Indicates TCP/UDP Source port	Int: 80
tp_dst	Integer	Indicates TCP/UDP Destination port	Int: 333

III. EXPERIMENTAL ENVIRONMENT, SOFTWARE DEVELOPMENT AND RESULTS

The experiment performed for this research made use of a virtual Ubuntu Linux instance created and run on VMware Workstation. The Mininet network emulator and POX controller were installed on this Ubuntu machine. After investigating existing security components of POX, a Python script was written and integrated into POX, in order to provide layer 2 and layer 3 filtering functionality, to support the seven different attributes supported by the OpenFlow protocol: source and destination MAC addresses, source and destination IP addresses, source and destination transport layer port addresses, and protocol. In addition, blacklisting and whitelisting capabilities were implemented. The source code for this Python script is given in Appendix A of this paper.

On the virtual Ubuntu machine, two terminals were opened: one for running the SDN topology and a second for running Python script that adds packet filtering capabilities to the switches of SDN topology started in terminal one. The commands used for running topology and Python script are as follow:

Mininet terminal:

```
sudo mn --topo single,7 --mac --switch ovsk --controller remote
```

POX terminal:

```
/home/pox/pox.py log.level --DEBUG pox.py pox.misc.SDN_FIREWALL
```

The command used in Mininet terminal, shown above, will create a topology with seven hosts connected to a single switch. The Python script is placed in the 'pox/misc' directory, and has the name 'SDN_FIREWALL.py'. To open a CLI in the POX terminal, the pre-existing component of POX, 'pox.py', is executed along with the Python script implemented for this research. After the command is executed in both the terminals, rules can be added from the CLI using interactive variables of the implemented Python script depicted (6) in Appendix A. The Python script in Appendix A allows adding and deleting specific rules, clearing all the rules, and listing rules from the CLI using interactive variable, all achieved via various depicted in (2), (3), (4), and (5) of Appendix A.

In Appendix A, (1) depicts the function that is loaded to enable whitelisting. This rule will have priority 65535. This script is implemented in a way that rules with the least priority will be executed first. In other words, rules will be executed in a descending order of priority.

In Appendix A, (2) shows the function that allows us to add blocking/allowing rules to the switches. It has the capability to consider seven different attributes at a time to make packet filtering decisions. In Appendix A, (3) represents the function of deleting rules with a specified priority. In Appendix A, (4) represents the function of clearing all the rules from the switches' flow table. In Appendix A, (5) represents the function of listing the active rules on switches.

In the given Python script, rules can also be added into the script itself before executing it. For the purpose of testing this script, nine different rules were added to the script in a way, chosen in such a way as to test all the filtering functionalities implemented for this research. The tabular representation of these rules is given in Appendix B of this paper.

In Appendix B, the first rule is added to test whitelisting. Rules numbered 2, 3, 6, and 7 allow only ICMP communication among the specified hosts, and the results of testing these rules are given in Appendix C. In the results shown in Appendix C, successful communication among hosts is denoted as host names and unsuccessful communication among hosts is denoted as 'X'. The rules numbered 4 and 5, test filtering based on source and destination transport layer port addresses. The result achieved by adding rule number 4 and 5 is presented in Appendix D of this paper. The rules numbered 8 and 9, allow UDP traffic on the specified port and IP addresses. Appendix E shows the results achieved by adding rules numbered 8 and 9. All the results were as expected. In order to implement blacklisting, the very first rule in Appendix B should be removed, and the value of the 'Permit' argument should be set to 'false'.

IV. Contribution to Knowledge

Experimental results show that the added POX component not only provides layer 2 security, such as filtering based on MAC addresses, but it can also provide layer 3 security, such as filtering through IP addresses, and layer 4 security, such as transport layer port numbers. Prior to the software development described in section III, the available distribution of POX was capable only of inspecting the source and destination IP or MAC address fields. But a real-world firewall must be capable of examining many more fields of the packet, and the extensions to POX described in this paper contribute to that. In addition, as SDN is emerging partly because of its capability for network programming, these experiments were performed both on simple topologies with seven hosts, along with more complex topologies such as you might find in a datacenter. This Python script provides more intelligent filtering functionality such as filtering based on source/destination port addresses, source/destination MAC and IP addresses, and transport layer protocol. This Python script also has capability of filtering the traffic based on the direction of traffic: incoming/outgoing. In addition, the filtering rules can be added to switch from the CLI as described in section III. This CLI makes the Python script more user friendly to use and allows users to handle the traffic as they hit the switch. Finally, this script implements both blacklisting and whitelisting.

Preliminary Bibliography

[1] L.R. Prete, C.M. Schweitzer, A.A. Shinoda, and R.L. Oliveira, “Simulation in an SDN network scenario using the POX Controller,” 2014 IEEE Colombian Conference on Communications and Computing (COLCOM), 2014.

[2] A. Haji, A. Letaifa, and S. Tabbane, “Elastic Architecture based NFV and OpenStack to deploy VA service,” 2018 32nd International conference on Advanced Information Networking and Application Workshops, 2018.

[3] S.R. Basnet, R.S. Chaulagain, S. Pandey and S. Shakya, “Distributed High Performance Computing in OpenStack Cloud over SDN Infrastructure,” 2017 IEEE International Confernece on Smart Cloud, 2017.

[4] J. Mambretti, J. Chen, and F. Yeh, “Next Generation Clouds, The Chameleon Cloud Testbed, and Software Defined Networking (SDN),” 2015 International Conference on Cloud Computing Research and Innovation, 2015.

[5] A.N. Toosi, J. Son, and R. Buyya, “Clouds-Pi: A Low-Cost Raspberry-Pi-Based Micro Datacenter for Software Defined Cloud Computing,” IEEE Cloud Computing, Vol. 5, 2018.

[6] J. Mambretti, J. Chen, and F. Yeh, “Software-Defined Network Exchanges (SDXs) and Infrastructure (SDI): Emerging Innovations In SDN and SDI Interdomain Multi-Layer Services and Capabilities,” 26th International Teletraffic Congress (ITC), 2014.

[7] N. McKeown, et al., OpenFlow: Enabling Innovation in Campus Networks, ACM SIGCOMM Computer Communication Review, 2008, 2 , 69-74.

[8] N. Sophakan, and C. Sathiwiriyawong, “Securing OpenFlow Controller of Software-Defined Networks using Bayesian Network,” 2018 22nd International Computer Science and Engineering Conference (ICSEC), 2018.

[9] J. Son, and R. Buyya, “SDCon: Integrated Control Platform for Software-Defined Clouds,” IEEE Transactions on Parallel and Distributed Systems, Vol. 30, No. 1, 2019.

[10] D. Balagopal, and X.K. Rani, “NetWatch: Empowering Software-Defined network Switches for Packet Filtering,” 2015 International Conference on Applied and Theoretical Computing and Communiation Technology, 2015.

[11] U. Ashraf, “Placing Controllers in Software-Defined Wireless Mesh Networks,” 2018 International Conference on Computing, Mathematics and Engineering Technologies, 2018.

[12] P. Rengaraju, S. Senthil Kumar, and C. Lung, “Investigation of Security and QoS on SDN Firewall Using MAC Filtering,” 2017 International Conference on Computer Communication and Informatics (ICCCI), 2017.

[13] M. Suh, S. H. Park, B. Lee, and S. Yang, “Building Firewall over Software-Defined Network Controller,” 16th International Conference on Advanced Communication Technology, 2014.

[14] J. H. Cox, J. Chung, S. Donovan, J. Ivey, R. J. Clark, G. Riley, and H. L. Owen, “Advancing Software-Defined Networks: A Survey”, IEEE Access, 2017.

[15] GitHub, Repository: A-Firewall-on-POX-SDN-Controller, Available at: https://github.com/vamshireddy/A-Firewall-on-POX-SDN-Controller/blob/master/firewall.py

[16] GitHub, Repository: Hyderimran7SDN-POX-Firewall, Available at: https://github.com/hayderimran7/sdn-pox-firewall/blob/master/sdn-pox-openflow-fw.py

[17] GitHub, Repository: Codes-libertes/sdn-pox-firewall, Available at: “https://github.com/codes-libertes/sdn-pox-firewall/blob/master/stateful_firewall_legimity_debug.py”

[18] GitHub, Repository: Kyberdrb/sdnfirewall, Available at: https://github.com/kyberdrb/sdnfirewall/blob/master/main.py

[19] GitHub, Repository: Bwjsfjz1969/mininet-SDN-POX-Firewall, Available at: https://github.com/bwjsfjz1969/mininet-SDN-POX-Firewall/blob/master/pox_firewall.py

[20] GitHub, Repository: yehiaArafa/SDN-Firewall, Available at: https://github.com/yehiaArafa/SDN-Firewall

[21] GitHub, Repository: agrawalamod/SDN-Project, Availabel at: https://github.com/agrawalamod/SDN-Project/blob/master/firewall.py

[22] GitHub, Repository: jashdesai95/SDN-OpenFlow-Pox-Firewall, Available at: https://github.com/jashdesai95/SDN-OpenFlow-Pox-Firewall/blob/master/firewall.py

[23] GitHub, Repository: Biwenzhuu/SDN-firewall, Available at: https://github.com/Biwenzhuu/SDN-firewall/blob/master/controller.py

[24] GitHub, Repository: Waynezhang1995/Simple-POX-Firewall, Available at: https://github.com/waynezhang1995/Simple-POX-Firewall/blob/master/firewall.py

[25] GitHub, Repository: Uscwy/pox_router, Available at: https://github.com/uscwy/pox_router/blob/master/bonus/firewall.py

[26] GitHub, Repository: Raonadeem/Pyretic-statefull-firewall, Available at: https://github.com/raonadeem/Pyretic-statefull-firewall/blob/master/statefull_firewall.py

[27] GitHub, Repository: FreddMai/SDN, Available at: https://github.com/FreddMai/SDN/blob/master/firewall.py

[28] GitHub, Repository: Dominators-CMPE210/CMPE210_Project, Available at: https://github.com/Dominators-CMPE210/CMPE210_Project

APPENDIX

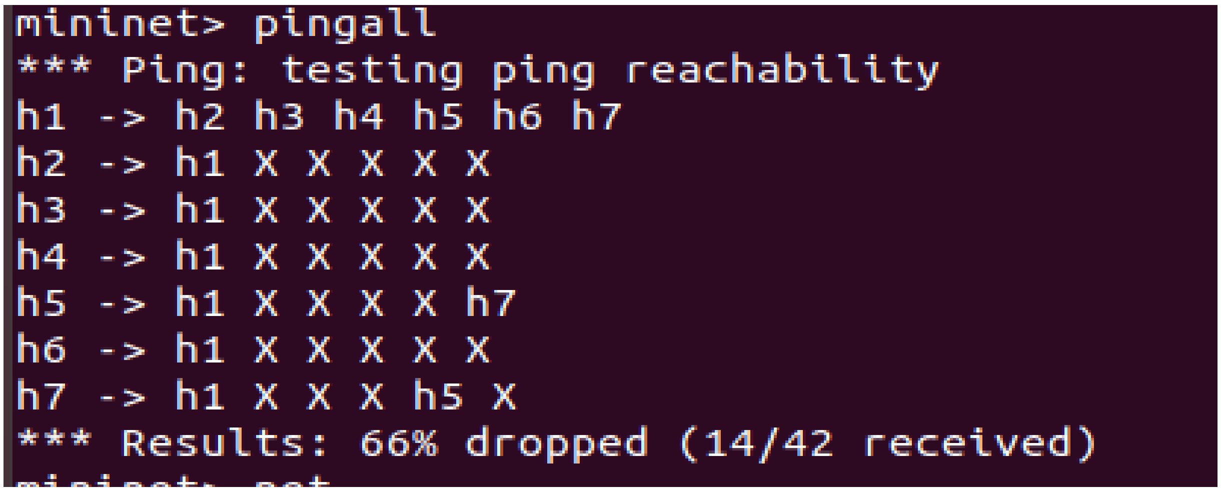
A. The source code of implemented Python script.

```
from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.util import dpid_to_str
from pox.lib.util import str_to_bool
from pox.lib.addresses import EthAddr,IPAddr,parse_cidr
import pox.lib.packet as pkt
import time

log = core.getLogger()

_flood_delay = 0

class LearningSwitch (object):
  def __init__ (self, connection, transparent):
    self.connection = connection
          self.transparent = transparent

          self.macToPort = {}
          self.priority = 65535
          self.rules = {}

          self.defaultRules()
          connection.addListeners(self)
          self.hold_down_expired = flood_delay == 0

  def defaultRules(self): _______________________________________________ (1)
    self.writeRule('0.0.0.0/0', '0.0.0.0/0', priority=0)

#add rules to be allowed according to source and destination MAC addresses, source and destination IP addresses,
source and destination Port addresses and Protocol.

          self.writeRule('0.0.0.0/0','10.0.0.1',permit=True,proto=1)
          self.writeRule('10.0.0.1','0.0.0.0/0',permit=True,proto=1)
          self.writeRule('10.0.0.2','0.0.0.0/0',permit=True,src_port=80,proto=6)
          self.writeRule('10.0.0.3','10.0.0.2',permit=True,dst_port=80,proto=6)
          self.writeRule('00:00:00:00:00:05','10.0.0.7',permit=True,proto=1)
          self.writeRule('10.0.0.7','00:00:00:00:00:05',permit=True,proto=1)
          self.writeRule('10.0.0.4','0.0.0.0',permit=True,src_port=53,proto=17)
          self.writeRule('10.0.0.3','10.0.0.4',permit=True,dst_port=53,proto=17)

#Function to add rules to the switches
def writeRule(self,src,dst,permit=False,duration=1000,src_port=None,dst_port=None,proto=None,priority=None): ___ (2)

      if priority != None:
          try:
              self.deleteRule(priority)
       except:
          log.debug('Rule does not exist')

    log.debug("Adding firewall rule in between %s: %s",src,dst)
    if not isinstance(duration, tuple):
      duration = (duration,duration)
    match = of.ofp_match(dl_type = 0x800)

    try:
       src.index(".")
       match.nw_src = parse_cidr(src)
    except ValueError:
       match.dl_src = EthAddr(src)

    try:
       dst.index(".")
       match.nw_dst = parse_cidr(dst)
    except ValueError:
       match.dl_dst = EthAddr(dst)

    match.tp_src = src_port
    match.tp_dst = dst_port
    match.nw_proto = proto

    msg = of.ofp_flow_mod()
    msg.match = match
    msg.idle_timeout = duration[0]
    msg.hard_timeout = duration[1]
    msg.command = of.OFPFC_ADD
    if priority == None:
       priority = self.priority
       self.priority -= 1
    msg.priority = priority
    if permit == True:
       action = of.ofp_action_output(port=of.OFPP_NORMAL)
       msg.actions.append(action)
    self.rules[msg.priority] = msg
    self.connection.send(msg)

  def deleteRule(self, priority):        #To delete specific rule ______________________________________________ (3)
    msg = of.ofp_flow_mod(command=of.OFPFC_DELETE, priority=priority)
    for connection in core.openflow.connections:
      connection.send(msg)
      log.debug("Clearing all flows from %s",dpid_to_str(connection.dpid))
    try:
              del self.rules[priority]
    except:
       log.debug('Rule does not exist')

  def clearRules(self):           #To clear rules from every switch that is connected ____________________________ (4)
    msg = of.ofp_flow_mod(command=of.OFPFC_DELETE)
    for connection in core.openflow.connections:
      connection.send(msg)
      log.debug("Clearing all flows from %s",dpid_to_str(connection.dpid))

  def listRules(self):            #Function to show the rules that are currently active on switches ________________ (5)
      for i in self.rules:
               log.debug(self.rules[i].match)

  def _handle_PacketIn (self, event):
    packet = event.parsed

    def flood (message = None):
      msg = of.ofp_packet_out()
      if time.time() - self.connection.connect_time >= _flood_delay:
       if self.hold_down_expired is False:
         self.hold_down_expired = True
         log.info("%s: Flood hold-down expired -- flooding",
              dpid_to_str(event.dpid))

       if message is not None: log.debug(message)
       msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
       else:
         pass

       msg.data = event.ofp
       msg.in_port = event.port
       self.connection.send(msg)

  def drop (duration = None):
       if duration is not None:
         if not isinstance(duration, tuple):
           duration = (duration,duration)
         msg = of.ofp_flow_mod()
         msg.match = of.ofp_match.from_packet(packet)
         msg.idle_timeout = duration[0]
         msg.hard_timeout = duration[1]
         msg.buffer_id = event.ofp.buffer_id
         self.connection.send(msg)
       elif event.ofp.buffer_id is not None:
                 msg = of.ofp_packet_out()
         msg.buffer_id = event.ofp.buffer_id
         msg.in_port = event.port
         self.connection.send(msg)

      self.macToPort[packet.src] = event.port

      if not self.transparent:
       if packet.type == packet.LLDP_TYPE or packet.dst.isBridgeFiltered():
         drop()
         return

      if packet.dst.is_multicast:
       flood()
      else:
       if packet.dst not in self.macToPort:
         flood("Port for %s unknown -- flooding" % (packet.dst,)) #4a
       else:
         port = self.macToPort[packet.dst]
         if port == event.port:
           log.warning("Same port for packet from %s -> %s on %s.%s.  Drop."
               % (packet.src, packet.dst, dpid_to_str(event.dpid), port))
           drop(10)
           return

         log.debug("installing flow for %s.%i -> %s.%i" %
                   (packet.src, event.port, packet.dst, port))
                         msg = of.ofp_flow_mod()
         msg.match = of.ofp_match.from_packet(packet, event.port)
         msg.idle_timeout = 10
         msg.hard_timeout = 30
         msg.actions.append(of.ofp_action_output(port = port))
         msg.data = event.ofp
         self.connection.send(msg)

  masterSwitch = list()
  class l2_learning (object):
    def __init__ (self, transparent):
      core.openflow.addListeners(self)
      self.transparent = transparent

    def _handle_ConnectionUp (self, event):
      log.debug("Connection %s" % (event.connection,))
      x = LearningSwitch(event.connection, self.transparent)
      masterSwitch.append(x)
      core.Interactive.variables['fw'] = x    #Interactive Variable to add rules to the switches from CLI  ____________ (6)


  def launch (transparent=False, hold_down=_flood_delay):
    try:
      global _flood_delay
      _flood_delay = int(str(hold_down), 10)
      assert _flood_delay >= 0
    except:
      raise RuntimeError("Expected hold-down to be a number")

    core.registerNew(l2_learning, str_to_bool(transparent))
```

B. Following table represents the ruleset which was implemented to test the functionalities of Python script.

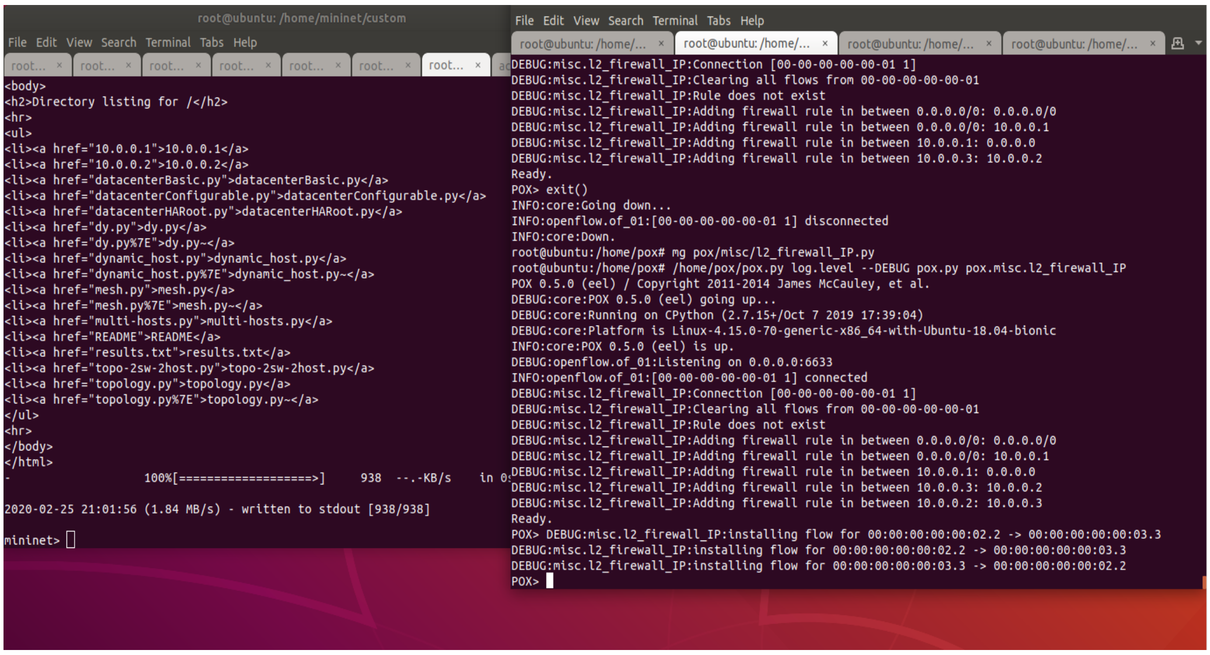
Rule No.	Action	Source	Destination	Source Port	Destination Port	Protocol
1	Deny	0.0.0.0/0	0.0.0.0/0	None	None	None
2	Allow	0.0.0.0/0	10.0.0.1	None	None	1
3	Allow	10.0.0.1	0.0.0.0/0	None	None	1
4	Allow	10.0.0.2	0.0.0.0/0	80	None	6
5	Allow	10.0.0.3	10.0.0.2	None	80	6
6	Allow	00:00:00:00:00:05	10.0.0.7	None	None	1
7	Allow	10.0.0.7	00:00:00:00:00:05	None	None	1
8	Allow	10.0.0.4	0.0.0.0/0	53	None	17
9	Allow	10.0.0.3	10.0.0.4	None	53	17

C. Following image depicts the results achieved by implementing the rule number 1, 2, 3, 6 and 7. These rules allow ICMP communication between specified hosts.



```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7
h2 -> h1 X X X X X
h3 -> h1 X X X X X
h4 -> h1 X X X X X
h5 -> h1 X X X X h7
h6 -> h1 X X X X X
h7 -> h1 X X X h5 X
*** Results: 66% dropped (14/42 received)
```

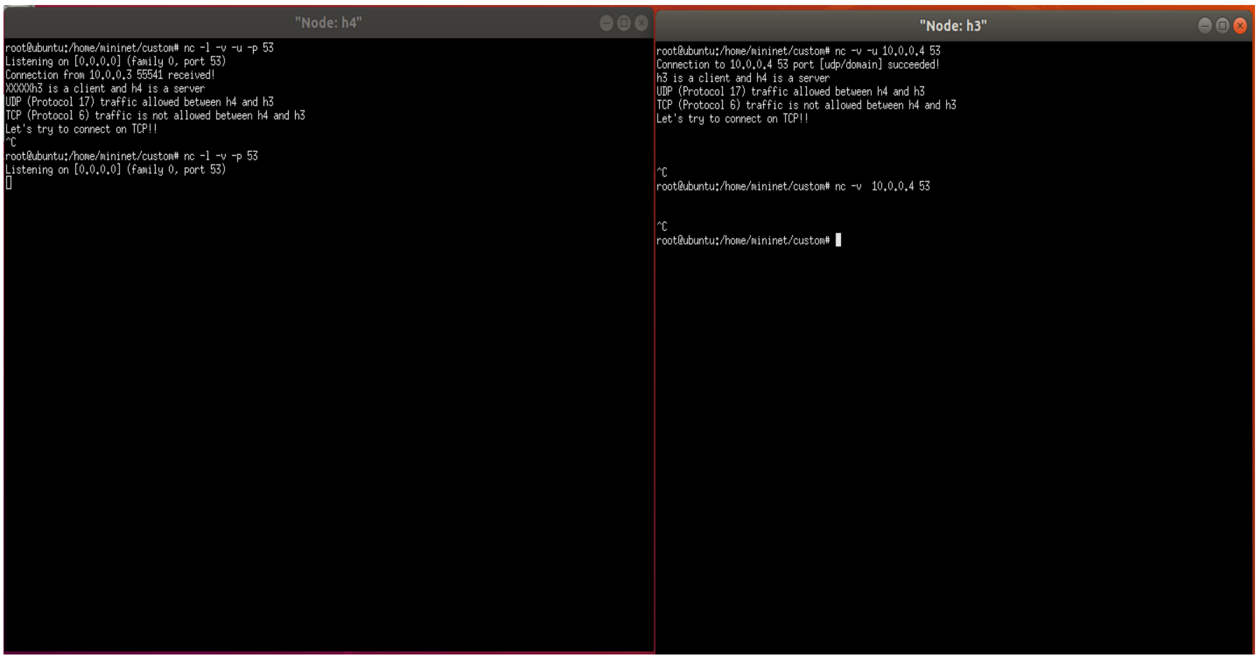
D. Following image depicts the results achieved by implementing the rule number 4 and 5. These rules allow TCP traffic on port 80 between specified hosts.


root@ubuntu: /home/mininet/custom
File Edit View Search Terminal Tabs Help
root@ubuntu: /home/...

```
<body>
<h2>Directory listing for /</h2>
<hr>
<ul>
<li><a href="10.0.0.1">10.0.0.1</a>
<li><a href="10.0.0.2">10.0.0.2</a>
<li><a href="datacenterBasic.py">datacenterBasic.py</a>
<li><a href="datacenterConfigurable.py">datacenterConfigurable.py</a>
<li><a href="datacenterHARoot.py">datacenterHARoot.py</a>
<li><a href="dy.py">dy.py</a>
<li><a href="dy.py%7E">dy.py~</a>
<li><a href="dynamic_host.py">dynamic_host.py</a>
<li><a href="dynamic_host.py%7E">dynamic_host.py~</a>
<li><a href="mesh.py">mesh.py</a>
<li><a href="mesh.py%7E">mesh.py~</a>
<li><a href="multi-hosts.py">multi-hosts.py</a>
<li><a href="README">README</a>
<li><a href="results.txt">results.txt</a>
<li><a href="topo-2sw-2host.py">topo-2sw-2host.py</a>
<li><a href="topology.py">topology.py</a>
<li><a href="topology.py%7E">topology.py~</a>
</ul>
<hr>
</body>
</html>
-                   100%[===================>]     938  --.-KB/s    in 0

2020-02-25 21:01:56 (1.84 MB/s) - written to stdout [938/938]

mininet> 
```

```
DEBUG:misc.l2_firewall_IP:Connection [00-00-00-00-00-01 1]
DEBUG:misc.l2_firewall_IP:Clearing all flows from 00-00-00-00-00-01
DEBUG:misc.l2_firewall_IP:Rule does not exist
DEBUG:misc.l2_firewall_IP:Adding firewall rule in between 0.0.0.0/0: 0.0.0.0/0
DEBUG:misc.l2_firewall_IP:Adding firewall rule in between 0.0.0.0/0: 10.0.0.1
DEBUG:misc.l2_firewall_IP:Adding firewall rule in between 10.0.0.1: 0.0.0.0
DEBUG:misc.l2_firewall_IP:Adding firewall rule in between 10.0.0.3: 10.0.0.2
Ready.
POX> exit()
INFO:core:Going down...
INFO:openflow.of_01:[00-00-00-00-00-01 1] disconnected
INFO:core:Down.
root@ubuntu:/home/pox# mg pox/misc/l2_firewall_IP.py
root@ubuntu:/home/pox# /home/pox/pox.py log.level --DEBUG pox.py pox.misc.l2_firewall_IP
POX 0.5.0 (eel) / Copyright 2011-2014 James McCauley, et al.
DEBUG:core:POX 0.5.0 (eel) going up...
DEBUG:core:Running on CPython (2.7.15+/Oct 7 2019 17:39:04)
DEBUG:core:Platform is Linux-4.15.0-70-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 1] connected
DEBUG:misc.l2_firewall_IP:Connection [00-00-00-00-00-01 1]
DEBUG:misc.l2_firewall_IP:Clearing all flows from 00-00-00-00-00-01
DEBUG:misc.l2_firewall_IP:Rule does not exist
DEBUG:misc.l2_firewall_IP:Adding firewall rule in between 0.0.0.0/0: 0.0.0.0/0
DEBUG:misc.l2_firewall_IP:Adding firewall rule in between 0.0.0.0/0: 10.0.0.1
DEBUG:misc.l2_firewall_IP:Adding firewall rule in between 10.0.0.1: 0.0.0.0
DEBUG:misc.l2_firewall_IP:Adding firewall rule in between 10.0.0.3: 10.0.0.2
DEBUG:misc.l2_firewall_IP:Adding firewall rule in between 10.0.0.2: 10.0.0.3
Ready.
POX> DEBUG:misc.l2_firewall_IP:installing flow for 00:00:00:00:00:02.2 -> 00:00:00:00:00:03.3
DEBUG:misc.l2_firewall_IP:installing flow for 00:00:00:00:00:02.2 -> 00:00:00:00:00:03.3
DEBUG:misc.l2_firewall_IP:installing flow for 00:00:00:00:00:03.3 -> 00:00:00:00:00:02.2
POX> 
```

E. Following image depicts the results achieved by implementing the rule number 8 and 9. These rules allow UDP traffic on port 53 between specified hosts.



```
root@ubuntu:/home/mininet/custom# nc -l -v -u -p 53
Listening on [0.0.0.0] (family 0, port 53)
Connection from 10.0.0.3 55541 received!
XXXXXh3 is a client and h4 is a server
UDP (Protocol 17) traffic allowed between h4 and h3
TCP (Protocol 6) traffic is not allowed between h4 and h3
Let's try to connect on TCP!!
^C
root@ubuntu:/home/mininet/custom# nc -l -v -p 53
Listening on [0.0.0.0] (family 0, port 53)
```

```
root@ubuntu:/home/mininet/custom# nc -v -u 10.0.0.4 53
Connection to 10.0.0.4 53 port [udp/domain] succeeded!
h3 is a client and h4 is a server
UDP (Protocol 17) traffic allowed between h4 and h3
TCP (Protocol 6) traffic is not allowed between h4 and h3
Let's try to connect on TCP!!



^C
root@ubuntu:/home/mininet/custom# nc -v  10.0.0.4 53


^C
root@ubuntu:/home/mininet/custom# 
```