# SDN Firewall with POX

## Table of Contents

## SDN Firewall with POX Project

In this project, we will use Software Defined Networking (SDN) principles to create a configurable firewall using an OpenFlow enabled Switch. The Software Defined Networking (OpenFlow) functionality allows we to programmatically control the flow of traffic on the network.

This project has three phrases as follows:

1. Mininet Tutorial – This phase is a brief overview of Mininet.

2. Wireshark Tutorial – This phase is a brief introduction to packet capture using Wireshark/tshark. We will examine the packet format for various traffic to learn of the different header values used in Phase 3.
3. SDN Firewall – This phase involves completing code to build a simple traffic blocking firewall using OpenFlow with the POX Controller based on rules passed to it from a configuration file. In addition, we will create a set of rules to test the firewall implementation.

## Part 0: Project References

We will find the following resources useful in completing this project. It is recommended that we review these resources before starting the project.

- IP Header Format - https://erg.abdn.ac.uk/users/gorry/course/inet-pages/ip-packet.html
- TCP Packet Header Format - https://en.wikipedia.org/wiki/Transmission_Control_Protocol
- UDP Packet Header Format - https://en.wikipedia.org/wiki/User_Datagram_Protocol
- The ICMP Protocol - https://en.wikipedia.org/wiki/Internet_Control_Message_Protocol
- IP Protocols - https://en.wikipedia.org/wiki/List_of_IP_protocol_numbers
- TCP and UDP Service and Port References - https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers
- Wireshark - https://www.wireshark.org/docs/wsug_html/
- CIDR Calculator - https://account.arin.net/public/cidrCalculator
- CIDR - https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing

## Part 1: Files Layout

- cleanup.sh – this file called by using following command line: **./cleanup.sh**
  This file will clean up the Mininet Environment and kill all zombie Python and POX processes.
- sdn-topology.py – this file creates the Mininet topology used in this assignment. This is like what we created in the Simulating Networks project. When evaluating the code against the ruleset specified in this project, do not change it. However, we are encouraged to make the own topologies (and rules) to test the firewall. Look at the start-topology.sh file to see how to start a different topology.
- ws-topology.py – this file is substantially like sdn-topology, but it does not call the POX Controller. We will use this during the Wireshark exercise.
- setup-firewall.py – this file sets up the frameworks used in this project**.** This file will create the appropriate POX framework and then integrates the rules implemented in sdnfirewall.py into the OpenFlow engine. It will also read in the values from the configure.pol file and validate that the entries are valid. If we make changes to this file, the autograder will likely have issues with the final code as the autograder uses the unaltered distribution version of this file.
- start-firewall.sh – this is the shell script that starts the firewall. This file must be started before the topology is started. It will copy files to the appropriate directory and then start the POX OpenFlow controller. This file is called by using following command line: **./start-firewall.sh**

- start-topology.sh – this is the shell script that starts the Mininet topology used in the assignment. All it does is call the sdn-topology.py file with superuser permissions. This file is called by using following command line: **./start-topology.sh**
- test-client.py – this is a python test client program used to test the firewall. This file is called using the following command line: **python test-client.py PROTO SERVERIP PORT SOURCEPORT** where PROTO is either T for TCP, U for UDP, or G for GRE, SERVERIP is the IP address of the server (destination), PORT is the destination port, and optionally SOURCEPORT allows we to configure the source port that we are using. Example: **python test-client.py T 10.0.1.1 80**
- test-server.py – this is a python test server program used to test the firewall. This file is called using the following command line: **python test-server.py PROTO SERVERIP PORT** where PROTO is either T for TCP, U for UDP, G for GRE, SERVERIP is the IP address of the server (the server we are running this script on), and PORT is the service port.
  Example: **python test-server.py T 10.0.1.1 80**
- test-suite – This is a student developed test script that was developed in 2021 that can be used to test the implementation AFTER WE FINISH BOTH THE IMPLEMENTATION FILES. The test cases in the main folder will be used to evaluate the implementations for the first run. An alternate configuration and topology will also be used to evaluate the implementations. This will be similar to, but not identical to what is found in the extra sub-folder. See Appendix A for information on how to use the test suite.

## Part 2: Before We Begin

This project assumes basic knowledge about IP and TCP/UDP Protocols. It is highly encouraged that we review the following items before starting. This will help we in understanding the contents of IP packet headers and what we may need to match.

- What is the IP (Internet Protocol)? What are the different types of Network Layer protocols? ○ Review TCP and UDP? How does TCP or UDP differ from IP?
- Examine the packet header for a generic IP protocol entry. Contrast that with the packet header for a TCP packet, and for a UDP packet. What are the differences? What does each field mean?
- What constitutes a TCP Connection? How does this contrast with a UDP connection.
- A special IP protocol is ICMP. Why is ICMP important? What behavior happens when we do an ICMP Ping? If we block an ICMP response, what would we expect to see?
- If we block a host from ICMP, will we be able to send TCP/UDP traffic to it? ○ Can we explain what happens if we get a ICMP Destination Unreachable response? ○ What is CIDR notation? How do we subnet a network? ○ What IP Protocols use Source or Destination Ports?

## Part 3: Review of Mininet

Mininet is a network simulator that allows we to explore SDN techniques by allowing we to create a network topology including virtual switches, links, hosts/nodes, and controllers. It will also allow we to set the parameters for each of these virtual devices and will allow we to simulate real-world applications on the different hosts/nodes.

The following code sets up a basic Mininet topology similar to what is used for this project:

```
#!/usr/bin/python

from mininet.topo import Topo from
mininet.net  import Mininet
from mininet.node import CPULimitedHost, RemoteController
from mininet.util import custom from mininet.link import
TCLink from mininet.cli  import CLI
 class FirewallTopo(Topo):     def __init__(self, cpu=.1,
bw=10, delay=None, **params):
super(FirewallTopo,self).__init__()

        # Host in link configuration
hconfig = {'cpu': cpu}
        lconfig = {'bw': bw, 'delay': delay}

        # Create the firewall switch
s1 = self.addSwitch('s1')

        hq1 = self.addHost('hq1',ip='10.0.0.1',mac='00:00:00:00:00:1e', **hconfig)
self.addLink(s1,hq1)

        us1 = self.addHost( 'us1', ip='10.0.1.1', mac='00:00:00:01:00:1e', **hconfig)
self.addLink(s1,us1)
```

This code defines the following virtual objects:

- Switch s1 – this is a single virtual switch with the label ‘s1’. In Mininet, we may have as many virtual ports as we need – for Mininet, “ports” are considered to be a virtual ethernet jack, not an application port that we would use in building the firewall.
- Hosts hq1 and us1 – these are individual virtual hosts that we can access via xterm and other means. We can define the IP Address, MAC/Hardware Addresses, and configuration parameters that can define cpu speed and other parameters using the hconfig dictionary.
- Links between s1 and hq1 and s1 and us1 – consider these like an ethernet cable that we would run between a computer and the switch port. We can define individual port numbers on each side (i.e., port on the host and port on the virtual switch), but it is advised to let Mininet automatically wire the network. Like hosts, we can define configuration parameters to set link speed, bandwidth, and latency.

Useful Mininet Commands:

- For this project, we can start Mininet and load the firewall topology by running the **./start-topology.sh** from the project directory. We can quit Mininet by typing in the **exit** command.
- After we are done running Mininet, it is recommended that we cleanup Mininet. There are two ways of doing this. The first is to run the **sudo mn -c** command from the terminal and the second is to use the **./cleanup.sh** script provided in the project directory. Do this after every run to minimize any problems that might hang or crash Mininet.
- We can use the xterm command to start an xterm window for one of the virtual hosts. This command is run from the mininet> prompt. For example, we can type in **us1 xterm &** to open a xterm window for the virtual host us1. The & causes the window to open and run in the background. In this project, we will run the test-*-client.py and test-*-server.py in each host to test connectivity.

- The **pingall** command that is run from the mininet> prompt will cause all hosts to ping all other hosts. Note that this may take a long time. To run a ping between two hosts, we can specify **host1 ping host2** (for example, **us1 ping hq1** which will show the result of host us1 pinging hq1).
- The **help** command will show all Mininet commands and **dump** will show information about all hosts in the topology.

## Part 4: Wireshark

Wireshark is a network packet capture program that will allow we to capture a stream of network packets and examine them. Wireshark is used extensively to troubleshoot computer networks and in the field of information security. We will be using Wireshark to examine the packet headers to learn how to use this information to match traffic that will be affected by the firewall we are constructing.

tshark is a command line version of Wireshark that we will be using to capture the packets between mininet hosts and we will use Wireshark for the GUI to examine these packets. However, we will be allowed to use the Wireshark GUI if we would like in doing the packet capture.

Please watch the video referenced in Part 2 if we would like to follow along in time for a live packet capture.

- Step 1: Open up a Terminal Window and change directory to the SDNFirewall directory that was extracted in Part 1.

- Step 2: The first action is to start up the Mininet topology used for the Wireshark capture exercise. This topology matches the topology that we will be using when creating and testing the firewall. To start this topology, run the following command:

  **sudo python ws-topology.py**

  This will startup a Mininet session with all hosts created.

- Step 3: Start up two xterm windows for hosts us1 and us2. After we start each xterm window, it is recommended that we run the following command in each xterm window as we load them to avoid confusion about which xterm belongs to which host:

  **export PS1="hostname >"** replacing hostname with the actual hostname.

  Type in the following commands at the Mininet prompt.

  **us1 xterm &** (then run **export PS1="us1 >"** in the xterm window that pops up)
  **us2 xterm &** (likewise, **export PS1="us2 >"** in the second xterm window)

- Step 4: In this step, we want to start capturing all the traffic that traverses through the ethernet port on host us1. We do this by running tshark (or alternatively, wireshark) as follows from the mininet prompt:

**us1 sudo tshark -w /tmp/packetcapture.pcap**

This will start tshark and will output a pcap formatted file to /tmp/capture.pcap. Note that this file is created as root, so we will need to change ownership to mininet to use it in future steps – **chown mininet:mininet /tmp/packetcapture.pcap**

If we wish to use the Wireshark GUI instead of tshark, we would call **us1 sudo wireshark &**. We may use this method, but the TA staff will not provide support for any issues that may occur.

- Step 5: Now we need to capture some traffic. Do the following tasks in the appropriate xterm window:

  in us1 xterm: **ping 10.0.1.2** (hit control C after a few ping requests)
  In us2 xterm: **ping 10.0.1.1** (likewise hit control C after a few ping requests)
  In us1 xterm: **python test-server.py T 10.0.1.1 80**
  In us2 xterm: **python test-client.py T 10.0.1.1 80**
  After the connection completes, in the us1 xterm, press Control-C to kill theserver.
  In us1 xterm: **python test-server.py U 10.0.1.1 8000** In us2 xterm: **python test-client.py U 10.0.1.1 8000**
  In us1 xterm: press Control C to kill the server
  In Mininet Terminal: press Control C to stop tshark

- Step 6: At the mininet prompt, type in exit and press enter. Next, do the chown command as described in step 4 above to the packet capture. We may also close the two xterm windows as they are finished. Copy the /tmp/packetcapture.pcap to the project directory. This file is the deliverable for this phase of the project.

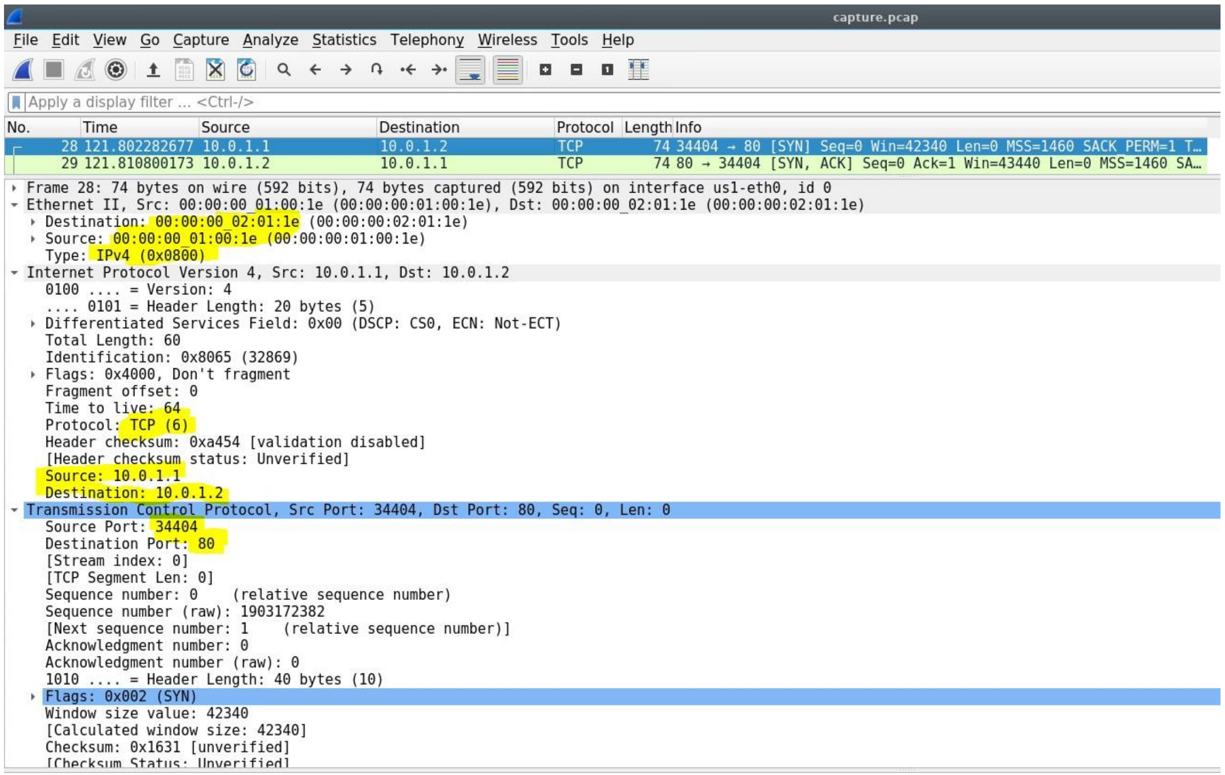
- Step 7: At the bash prompt on the main terminal, run:

  **sudo wireshark**

  Go to the File => Open menu item, browse to the /tmp directory and select the pcap file that we saved using tshark.

We will get a GUI that looks like the example packet capture. We will have a numbered list of all the captured packets with brief information consisting of source/destination, IP protocol, and a description of the packet. We can click on an individual packet and will get full details including the Layer 2 and Layer 3 packet headers, TCP/UDP/ICMP parameters for packets using those IP protocols, and the data contained in the packet.

**Example Packet Capture – Host us1 making web request to Host us2**


capture.pcap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

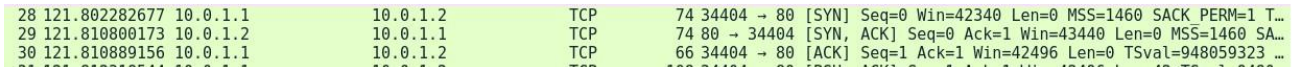
```
No.  Time           Source     Destination  Protocol Length Info
  28 121.802282677  10.0.1.1   10.0.1.2     TCP      74 34404 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=1 T…
  29 121.810800173  10.0.1.2   10.0.1.1     TCP      74 80 → 34404 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 SA…

› Frame 28: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface us1-eth0, id 0
⌄ Ethernet II, Src: 00:00:00_01:00:1e (00:00:00:01:00:1e), Dst: 00:00:00_02:01:1e (00:00:00:02:01:1e)
  › Destination: 00:00:00_02:01:1e (00:00:00:02:01:1e)
  › Source: 00:00:00_01:00:1e (00:00:00:01:00:1e)
    Type: IPv4 (0x0800)
⌄ Internet Protocol Version 4, Src: 10.0.1.1, Dst: 10.0.1.2
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  › Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 60
    Identification: 0x8065 (32869)
  › Flags: 0x4000, Don't fragment
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
    Header checksum: 0xa454 [validation disabled]
    [Header checksum status: Unverified]
    Source: 10.0.1.1
    Destination: 10.0.1.2
⌄ Transmission Control Protocol, Src Port: 34404, Dst Port: 80, Seq: 0, Len: 0
    Source Port: 34404
    Destination Port: 80
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 0    (relative sequence number)
    Sequence number (raw): 1903172382
    [Next sequence number: 1    (relative sequence number)]
    Acknowledgment number: 0
    Acknowledgment number (raw): 0
    1010 .... = Header Length: 40 bytes (10)
  › Flags: 0x002 (SYN)
    Window size value: 42340
    [Calculated window size: 42340]
    Checksum: 0x1631 [unverified]
    [Checksum Status: Unverified]
```

Note the highlighted fields. We will be using the information from these fields to help build the firewall implementation and ruleset. Note the separate header information for TCP. This will also be the case for UDP packets.

Also, examine the three-way handshake that is used for TCP. What do we expect to find for UDP? ICMP?

**Example TCP Three-Way Handshake**



```
28 121.802282677 10.0.1.1      10.0.1.2      TCP      74 34404 → 80 [SYN] Seq=0 Win=42340 Len=0 MSS=1460 SACK_PERM=1 T…
29 121.810800173 10.0.1.2      10.0.1.1      TCP      74 80 → 34404 [SYN, ACK] Seq=0 Ack=1 Win=43440 Len=0 MSS=1460 SA…
30 121.810889156 10.0.1.1      10.0.1.2      TCP      66 34404 → 80 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSval=948059323 …
```

Please examine the other packets that were captured to help we familiarize yourself with Wireshark.

## Part 5: SDN Firewall Implementation Details

Using the information that we learned above in running Wireshark, we will be creating two files – one is a firewall configuration file that will specify different header parameters to match in order to allow or block certain traffic and the second is the implementation code to create OpenFlow Flow Modification objects that will create the firewall using the parameters given in the firewall configuration file.

We may create temporary rulesets to help we complete Part 5b below.

### Part 5a: Specifications of configure.pol

The configure.pol file is used by the firewall implementation code to specify the rules that the firewall will use to govern a connection. We do not need to code this first, but the format of the file is important as the implementation code will need to use these items. The format of the file is a collection of lines that have the proper format:

**Rule Number, Action, Source MAC, Destination MAC, Source IP, Destination IP, Protocol, Source Port, Destination Port, Comment/Note**

- **Rule Number** = this is a rule number to help we track a particular rule - it is not used in the firewall implementation. It can be of any value and is NOT validated in setup-firewall.py. The value need not be unique and can be numeric or text.

- **Action** = **Block** or **Allow** **Block** rules will block traffic that matches the remaining parameters of this rule. **Allow** rules will override Block rules to allow specific traffic to pass through the firewall (see below for an example). The entry is a string in (Block,Allow).

- **Source / Destination MAC** address in form of xx:xx:xx:xx:xx:xx (example: 00:a2:c4:3f:11:09) or a "-" if we are not matching this item. We may use MAC Addresses to match an individual host. In the real world, we would use it to match a particular piece of hardware. The MAC address of a particular host is defined inside the sdn-topology.py file.

- **Source / Destination IP Network** Address in form of xxx.xxx.xxx.xxx/xx in CIDR notation or a "-" if we are not matching this item.. We can use this to match either a single IP Address (using it's IP address and a subnet mask of /32, or a particular Subnet. An entry here would look like: 10.0.0.1/32.
  The IP address of a particular host is defined inside the sdn-topology.py file.

- **Protocol** = integer IP protocol number per IANA (0-254) or a "-" if we are not matching this item.. An example is ICMP is IP Protocol 1, TCP is IP Protocol 6, etc. This must be an integer.

- **Source / Destination Port** = if Protocol is TCP or UDP, this is the Application Port Number per IANA. For example, web traffic is generally TCP Port 80. Do not try to use port numbers to differentiate the different elements of the ICMP protocol. If we are not matching this item or are using an IP Protocol other than TCP or UDP, this field should be a "-".

- **Comment/Note** = this is for the use in tracking rules.

**Special Notes About Firewall Configurations:**

- Any field not being used for a match should have a '-' character as its entry. A '-' means that the item is not being used for matching traffic. **It is valid for any rule element except for Action to have a '-'.** (i.e., a rule like: **1,Block,-,-,-,-,-,-,-,Block the world** is valid, but not a rule that will be tested). ***HINT: Do not use any item that has a "-" in its field as an element that we will match. If we pass a "-" to a field in a match rule, we will cause POX to crash and the firewall will not work.***

  ***A "-" is valid for ALL FIELDS except Action DO NOT PASS A "-" INTO ONE OF THE OPENFLOW MATCH VARIABLES OR THE CODE WILL CRASH..***

- **When a rule states to block the world from accessing a particular host, this means that we are matching against all possible hosts which may include hosts that are not in the topology.** ***HINT: Think about how we would match arbitrary traffic from anywhere on the network. Don't overthink this. Also, due to restrictions placed on the implementation by POX, please do not use 0.0.0.0/0 as an address for "world". In a real-world situation, this address would be valid as addressing any host on the internet.***

- Note that a rule does not necessarily need a MAC or IP Address. Also, it is possible to have a rule that only has network addresses and no ports/protocols. **What won't ever be tested is using a src/dst port WITHOUT an IP Protocol**.

- What is the difference between source and destination? Source makes a request of the destination. For ports, we will most often use destination ports, **but make sure that the firewall implements both source and destination ports**. For IP and MAC addresses, we will use both most of the time.

- When should I use MAC vs IP Addresses? We will want to interchange them in this file to test the robustness of the implementation. It is valid to specify a Source MAC address and a Destination IP Address.

**Example Rules (included in the project files:**

**1,Block,-,-,10.0.0.1/32,10.0.1.0/24,6,-,80,Block 10.0.0.1 host from accessing a web server on the 10.0.1.0/24 network**

**2,Allow,-,-,10.0.0.1/32,10.0.1.125/32,6,-,80,Allow 10.0.0.1 host to access a web server on 10.0.1.125 overriding rule**

**What do these rules do?**

The first rule basically blocks host hq1 (IP Address 10.0.0.1/32) from accessing a web server on any host on the us network (the subnet 10.0.1.0/24 network). The web server is running on the TCP IP Protocol (6) and uses TCP Port 80.

The second rule overrides the initial rule to allow hq1 (IP Address 10.0.0.1/32) to access a web server running on us5 (IP Address 10.0.1.125/32)

By definition – from the sdn-topology.py file:

```
This class defines the Mininet Topology for the network used in this project. This
network consists of the following hosts/networks:

    Headquarters Network (hq1-hq5). Subnet 10.0.0.0/24

    US Network (us1-us5). Subnet 10.0.1.0/24

    India Network (in1-in5). Subnet 10.0.20.0/24

    China Network (cn1-cn5). Subnet 10.0.30.0/24

    UK Network (uk1-uk5). Subnet 10.0.40.0/24
```

In Part 6, we will be given a set of firewall conditions that we will need to create the configure.pol needed for the submission.

We may create temporary rulesets to help we complete Part 5b below.

## Part 5b: Implementing the Firewall in Code

After reviewing the format of the configure.pol file, we will now code a generic implementation of a firewall that will use the values provided from the configuration file (passed to we as dictionary items). As it is provided, the firewall implementation code blocks no traffic. We must implement code that does the following:

- **Create an OpenFlow Flow Modification object ○ Create a POX Packet Matching object that will integrate the elements from a single entry in the firewall configuration rule file (which is passed in the policy dictionary) to match the different IP and TCP/UDP headers if there is anything to match (i.e., no "-" should be passed to the match object, nor should None be passed to a match object if a "-" is provided).**
- **Create a POX Output Action, if needed, to specify what to do with the traffic.**

Please reference code examples in Appendix C, or we may refer to the POX API documentation (WARNING, this is long and the API is confusing).

We will need to rewrite the rule = None to reference the Flow Modification object.

The code will go into a section that will repeat itself for every line in the firewall configuration file that is passed to it. The "rule" item that is added to the "rules" list is an OpenFlow Modification object. The process of injecting this rule into the POX controller is handled automatically for we in the start-firewall.py file.

TIP: If the implementation code segment is more than 25-30 lines, we are making it too difficult. The POX API can provide many features that are not used in this project. The Appendix provides all of the information that we will need to code the project.

**Key Information:**

- policies is a python list that contains one entry for each rule line contained in the configure.pol file. Each individual line of the configure.pol file is represented as a dictionary object named policy. This dictionary has the following keys:
- **policy['mac-src'] = Source MAC Address (00:00:00:00:00:00) or "-" ○ policy['mac-dst'] = Destination MAC Address (00:00:00:00:00:00) ) or "-" ○ policy['ip-src'] = Source IP Address (10.0.1.1/32) in CIDR notation ) or "-" ○ policy['ip-dst'] = Destination IP Address (10.0.1.1/32) ) or "-" ○ policy['ipprotocol'] = IP Protocol (6 for TCP) ) or "-" ○ policy['port-src'] = Source Port for TCP/UDP (12000) ) or "-" ○ policy['port-dst'] = Destination Port for TCP/UDP (80) ) or "-"**
- **policy['rulenum'] = Rule Number (1) ○ policy['comment'] = Comment (Example Rule) ○ policy['action'] = Allow or Block**

  Use these to match traffic. Please note that all fields are strings and may contain a '-' character. We may either use policy['ip-dst'] or the split policy['ip-dst-address']/[policy['ip-dst-subnet'] in the implementation (the split was requested by prior semesters), but realize that if we use the ip-dstaddress and ip-dst-subnet, we will need to carefully check the implementation to ensure that it is blocking the addresses we intend to block.

- We will need to assume that all traffic is IPV4. It is acceptable to hardcode this value. **Do not hardcode other values. The code should be generic enough to handle any possible configuration.**

- Hints: ○ The difference between an Allow or a Block is dependent on an Action and the Priority.
- We don't necessarily need an action. See Appendix C for a discussion of what happens to a packet after it is matched.
- There should be two priorities – one for ALLOW and one for BLOCK. Separate them sufficiently to override any exact matching behavior that the POX controller implements). It is suggested one priority be 0 or 1 and the other one above 10000. The reasoning for this is discussed in Appendix C.

- Outputting extra print debug lines will not adversely impact the autograder.

## Part 6: Configuration Rules

We will need to submit a **configure.pol** file to create policies that implement the following scenarios. We may implement the rules in any manner that we want, but it is recommended using this step as an opportunity to check the firewall code implementation. The purpose of these rules is to test the firewall and to help determine how traffic flows across the network (source vs destination, protocols, etc).

DO NOT block all traffic by default and only allow traffic specified. We will lose many points because the firewall is open by default and only blocks the traffic that is specified.

We work for GT-SDN Corporation that has offices in the US, China, India, and the UK, with a US headquarters that acts as the datacenter for the company. The task is to implement a firewall that accomplishes the following goals:

- **Task 1:** On the headquarters network, we have two active DNS servers (using the newer DNSover-TLS standard operating on TCP and UDP Port 853). hq1 provides DNS service to the public (the world) and hq2 provides a private DNS service that should be accessible only to the 5 corporate networks (i.e., the US, China, India, UK, and Headquarters network). (DNS-over-TLS is restricted to TCP and UDP Protocol 853 for the purpose of satisfying the ruleset for this project)

  **Rule Objective: A connection from any host in the word should be able connect to hq1 on TCP and UDP 853. However, only hosts on the US, China, India, UK, and HQ networks should be able to connect to TCP and UDP 853 on host hq2. All other hosts should NOT be able to connect to the DNS Server on host hq2.**

- **Task 2:** On the headquarters network, the host hq3 acts as a VPN server that connects to each of the other sites (hosts us3, uk3, in3, and cn3) using the OpenVPN server (standard ports – using both TCP and UDP Ports 1194 will satisfy the requirements for this rule). Create a set of firewall rules that will only allow the 4 offsite hosts (us3, uk3, in3, and cn3) access to the hq3 OpenVPN server. No other hosts in the world should be able to access the OpenVPN server on hq3.

  **Rule Objective: Only hosts us3, uk3, in3, and cn3 should connect to TCP and UDP Port 1194 on host HQ3. No other host should be able to connect to TCP and UDP 1194 on hq3.**

- **Task 3:** Allow the hosts on the Headquarters network to be reachable via an ICMP ping from the world (including from the us, uk, in, and cn networks). However, the us, uk, in, and cn networks should not be reachable from the world (due to firewall implementation limitations, the hq network would be able to ping the us, uk, in, and cn network. Why? What changes could be made to the implementation requirements to allow this?)

  **Rule Objective: All hosts can receive a complete ping request/response to any HQ network computers. Any hosts attempting to ping the US, UK, IN, and CN networks should NOT get a complete ping request/response from these hosts EXCEPT for the HQ network. In order to satisfy the first part of this rule, the HQ network must be able to ping the US, UK, IN, and CN network.**

- **Task 4:** One of the main routes for ransomware to enter a corporate network is through a remote desktop connection with either an insecure version of a server protocol or via leaked or weak credentials (using either the Microsoft Remote Desktop protocol or the Virtual Networking Computing (VNC) protocols as the remote desktop server). For this task, write a set of rules that will block the internet from connecting to a remote desktop server on the five corporate networks. Allow the uk, us, in, and cn to connect to a remote desktop server on the headquarters network. (Use TCP Port 3389 for Remote Desktop and TCP Port 5900 for VNC)

  **Rule Objective: Block any hosts outside of the corporate network (HQ, US, UK, IN, and CN) from connecting to any hosts on corporate network on TCP ports 3389 and 5900). Computers on the corporate network CAN connect to TCP ports 3389 and 5900 on the headquarters network. Connections between the other corporate networks (cn1 to us1) are not defined and can be set as desired.**

  **We only need to block in the direction specified.**

- **Task 5:** The servers located on hosts us3 and us4 run a micro webservice on TCP Port 8510 that processes financial information. Access to this service should be blocked from hosts uk2, uk3, uk4, uk5, in4, in5, us5, and hq5. (Hint: Note the IP Addresses for these hosts and we may use the smallest subnet mask that handles the listed hosts using CIDR notation).

**Rule Objective: This rule is designed to test using different CIDR notations to bracket hosts together. Otherwise, the rule is to be interpreted as written. We do not need to use CIDR notation to combine hosts, but it will result in many additional rules.**

**TIPS:**

- Note that we only need to use an ALLOW rule to override a BLOCK rule. So if no block rule is supplied, the ALLOW is not needed. This is important for Rule #1. We will definitely need to use ALLOW rules for Rule #3.
- Note that if a rule specifies that TCP Port 8510 should be blocked, it does not mean that UDP Port 8510 should be blocked. The autograder checks to ensure that connections are not being overblocked.

## Appendix A: How to Test Host Connectivity

### Part A: How to Test Manually

When we are developing the implementation or troubleshooting a firewall rule, we will want to test by hand. Unfortunately this process is a bit difficult.

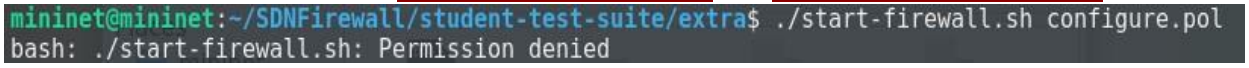
**1,Block,-,-,10.0.0.1/32,10.0.1.0/24,6,-,80,Block 10.0.0.1 from accessing a web server on the 10.0.1.0/32 network Startup Procedure:**

| **SUMMER 2024 Important Note:** |
|---|
| Please run the following code after setting up the project code. This will update POX to the latest version. |
| **cd ~ git clone**<br>**https://github.com/noxrepo/pox.git cd pox**<br>**git checkout halosaur** |

- Step 1: Open two terminal windows or tabs on the VM and change to the SDNFirewall directory.

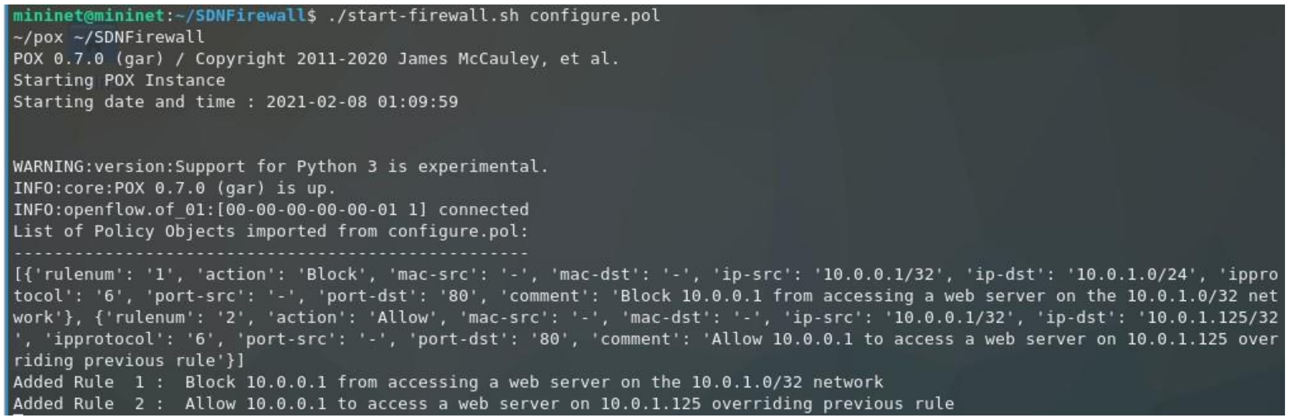
- Step 2: In the first terminal window, type in: **./start-firewall.sh configure.pol**

  **If we get the following error, run chmod +x start-firewall.sh and chmod +x start-topology.sh**



```
mininet@mininet:~/SDNFirewall/student-test-suite/extra$ ./start-firewall.sh configure.pol
bash: ./start-firewall.sh: Permission denied
```

  This should start up POX, read in the rules, and start up an OpenFlow Controller. We will see something like this in the terminal window:

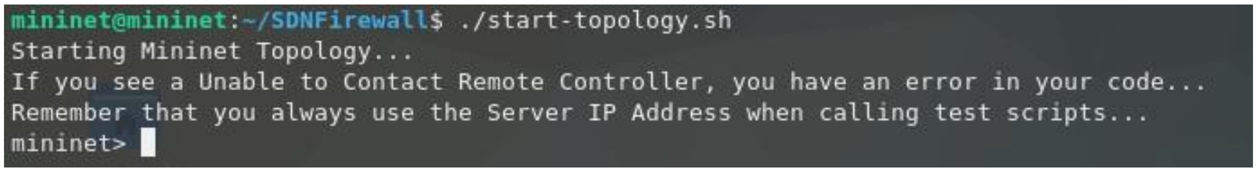


```
mininet@mininet:~/SDNFirewall$ ./start-firewall.sh configure.pol
~/pox ~/SDNFirewall
POX 0.7.0 (gar) / Copyright 2011-2020 James McCauley, et al.
Starting POX Instance
Starting date and time : 2021-02-08 01:09:59


WARNING:version:Support for Python 3 is experimental.
INFO:core:POX 0.7.0 (gar) is up.
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
List of Policy Objects imported from configure.pol:
----------------------------------------------------
[{'rulenum': '1', 'action': 'Block', 'mac-src': '-', 'mac-dst': '-', 'ip-src': '10.0.0.1/32', 'ip-dst': '10.0.1.0/24', 'ippro
tocol': '6', 'port-src': '-', 'port-dst': '80', 'comment': 'Block 10.0.0.1 from accessing a web server on the 10.0.1.0/32 net
work'}, {'rulenum': '2', 'action': 'Allow', 'mac-src': '-', 'mac-dst': '-', 'ip-src': '10.0.0.1/32', 'ip-dst': '10.0.1.125/32
', 'ipprotocol': '6', 'port-src': '-', 'port-dst': '80', 'comment': 'Allow 10.0.0.1 to access a web server on 10.0.1.125 over
riding previous rule'}]
Added Rule  1 :  Block 10.0.0.1 from accessing a web server on the 10.0.1.0/32 network
Added Rule  2 :  Allow 10.0.0.1 to access a web server on 10.0.1.125 overriding previous rule
```

- Step 3: In the second terminal window, type in: **./start-topology.sh**

  This should start up mininet and load the topology. We should see the following:



```
mininet@mininet:~/SDNFirewall$ ./start-topology.sh
Starting Mininet Topology...
If you see a Unable to Contact Remote Controller, you have an error in your code...
Remember that you always use the Server IP Address when calling test scripts...
mininet> 
```

  This will start the firewall and set the topology. We do not need to repeat Steps 1-3 unless we are done testing, need to restart the firewall, or need to restart mininet. When we are done with testing all of the rules we intend to use, type in "quit" in the mininet window, close all of the extraneous xterm windows generated, and run the mininet cleanup script **./cleanup.sh**

**How to test connectivity between two hosts:**

- Step 1: To test the rule shown above, we want to use host us1 as server/destination and host hq1 as the client. The rule we are testing involves the hq1 host attempting to connect to the web server port (TCP Port 80) on host us1. At the mininet prompt, type in the following two commands on two different lines:

  **hq1 xterm &** **us1 xterm &**

  Two windows should have popped up. We can always identify which xterm is which by running the command: **ip address** from the bash shell. This will give we the IP address for the xterm window, which will then let we discover which xterm window belongs to which host.

- Step 2: In the xterm window for us1 (which is the destination host of the rule – remember that the destination is always the server), type in the following command:

  **python test-server.py T 10.0.1.1 80**

  This sets up the test server for us1 that will be listening on TCP port 80. The IP Address specified is always the IP address of the machine we are running it on. **If we attempt to start the test-server on a machine**

**that does not have the IP address that is specified in the command, we will get the following error: OSError:  [Errno 99] Cannot assign requested address.**

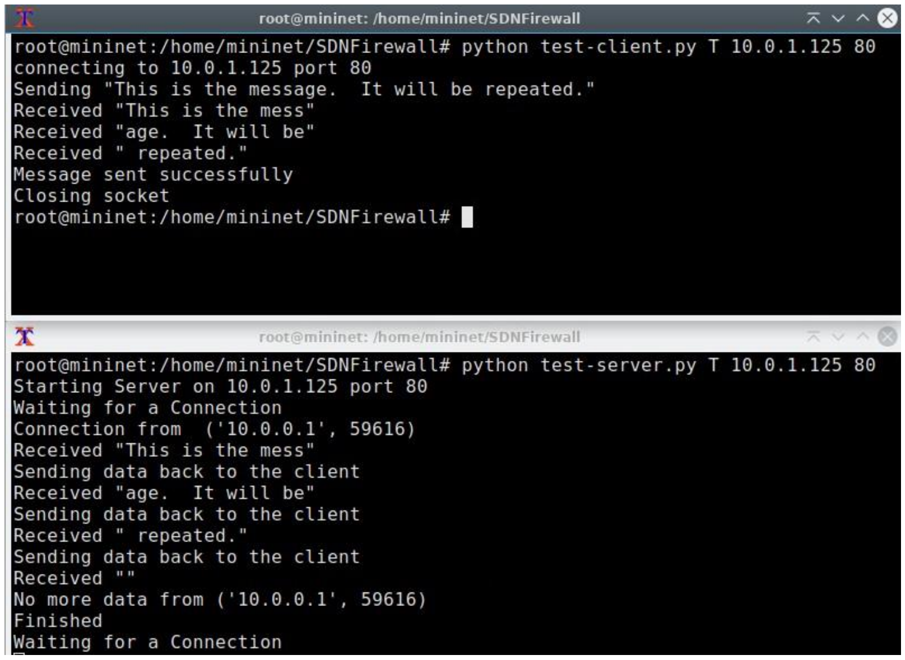
○ Step 6:  In the xterm window for hq1 (which is the source host of the rule – remember that the source is always the client), type in the following command:

**python test-client.py T 10.0.1.1 80**

This will start up a client that will connect to the TCP Port 80 on the server 10.0.1.1 (destination IP address) and will send a message string to the server. However, if the firewall is set to block this connection, we will never see the message pass on either of the client or the server.

Examples of Connection Status:

- The two windows below depict a successful un-blocked connection between the client and the server.

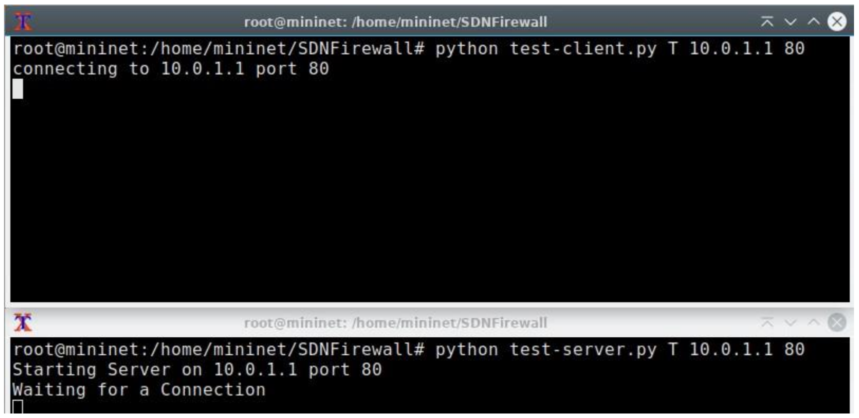

root@mininet: /home/mininet/SDNFirewall

```
root@mininet:/home/mininet/SDNFirewall# python test-client.py T 10.0.1.125 80
connecting to 10.0.1.125 port 80
Sending "This is the message.  It will be repeated."
Received "This is the mess"
Received "age.  It will be"
Received " repeated."
Message sent successfully
Closing socket
root@mininet:/home/mininet/SDNFirewall# 
```

root@mininet: /home/mininet/SDNFirewall

```
root@mininet:/home/mininet/SDNFirewall# python test-server.py T 10.0.1.125 80
Starting Server on 10.0.1.125 port 80
Waiting for a Connection
Connection from  ('10.0.0.1', 59616)
Received "This is the mess"
Sending data back to the client
Received "age.  It will be"
Sending data back to the client
Received " repeated."
Sending data back to the client
Received ""
No more data from ('10.0.0.1', 59616)
Finished
Waiting for a Connection
```

- A blocked connection will look like this (note that the client may take a while to timeout):

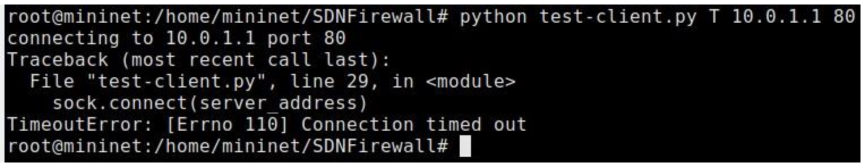

root@mininet: /home/mininet/SDNFirewall
root@mininet: /home/mininet/SDNFirewall

```
root@mininet:/home/mininet/SDNFirewall# python test-client.py T 10.0.1.1 80
connecting to 10.0.1.1 port 80
```

```
root@mininet:/home/mininet/SDNFirewall# python test-server.py T 10.0.1.1 80
Starting Server on 10.0.1.1 port 80
Waiting for a Connection
```

We may hit Control C to kill both the server and the client.

- A timed out connection is shown below. The difference between a timed-out connection on how the connection was blocked or if it was blocked on a different side of the connection.



```
root@mininet:/home/mininet/SDNFirewall# python test-client.py T 10.0.1.1 80
connecting to 10.0.1.1 port 80
Traceback (most recent call last):
  File "test-client.py", line 29, in <module>
    sock.connect(server_address)
TimeoutError: [Errno 110] Connection timed out
root@mininet:/home/mininet/SDNFirewall#
```

- If we get an error that says "No route to destination", we have blocked the routing protocol. Ensure that we do not have a Unspecified Prerequisite Error

Repeat this process for every rule we wish to test. If we feel that after some initial testing that the implementation and ruleset is good, we may then proceed to using the automated test suite.

## Part B: Automated Testing Suite

The automated Test Suit was developed by a student in Summer 2021 (htian66) and has been updated to match the current version of this project. The Autograder works in a similar manner, but is instrumented differently to grade. The Autograder will use the exact same test-cases to grade the configure.pol and sdn-firewall.py files, but the alternate topology is similar, but not the same as the extra test cases.

### How to test normal cases:

1. Change to the test-scripts directory
2. Copy the `sdn-firewall.py` and `configure.pol` into this directory.
3. Run `./start-firewall.sh configure.pol` as usual.
4. Open a new window, run `sudo python test_all.py`.
5. Total passed cases are calculated. Wrong cases will be displayed. For example, `2: us1 -> hq1 with U at 53, should be True, current False` means the connection from client us1 to host hq1 using UDP at hq1 53 port is failed, which should be successful. The first number is the index (0-based) of testcases.

Notes:

- One testcase in the `testcases.txt` file is given here: `us1 hq1 U 53 True` -> us1 client should be able to access hq1 host with TCP protocol at port 80. Please pull request to fill the `testcases.txt` file.
- For goal 3, `P` is used in `testcaeses.txt` to represent `ping`.
- `test-server.py` and `test-client.py` are slighted modified from the original version to support `GRE` protocol testing.

## **How to test extra cases:**

1. Change to the test-scripts/extras directory
2. Copy the `sdn-firewall.py` into this directory (do NOT copy configure.pol as there are different rules and hosts being used).
3. Run `./start-firewall.sh configure.pol` as usual.
4. Open a new window, run `sudo python test_all.py`.
5. Total passed cases are calculated. Wrong cases will be displayed. For example, `2: us1 -> hq1 with U at 53, should be True, current False` means

Note that we can extend the extra test cases to test other scenarios like source ports and additional IP Protocols.  Feel free to post the topology, configure.pol, and testcases.txt file for these additional scenarios.

## Appendix B: POX API Excerpt

This section contains a highly modified excerpt from the POX Manual (modified to remove extraneous features not used in this project and to provide clarifications). We should not need to use any other POX objects for this project. TA Comments are highlighted. Everything on these pages is important to complete the project.

Excerpted and modified from: [https://noxrepo.github.io/pox-doc/html/](https://noxrepo.github.io/pox-doc/html/)

**Object Definitions:**

## Flow Modification Object

The main object used for this project is a "Flow Modification" object. This adds a rule to the OpenFlow controller that will affect a modification to the traffic flow based on priority, packet characteristic matchin, and an action that will be done to the traffic that is matched. IF AN OBJECT is matched, it is pulled from the network stream and will only be forwarded, modified, or redirected if we do an action. If we do not specify an action and the packet is matched, the packet will basically be dropped.

The following class descriptor describes the contents of a flow modification object. We need to define the match, priority, and actions for the object.

```
class ofp_flow_mod (ofp_header):   def
__init__ (self, **kw):
ofp_header.__init__(self)    self.header_type =
OFPT_FLOW_MOD    self.match = ofp_match()
   self.priority = OFP_DEFAULT_PRIORITY
self.actions = []
```

### Match Structure

OpenFlow defines a match structure – ofp_match – which enables we to define a set of headers for packets to match against.

The match structure is defined in pox/OpenFlow/libOpenFlow_01.py in class ofp_match. Its attributes are derived from the members listed in the OpenFlow specification, so refer to that for more information, though they are summarized in the table below.

We should create a match object and attach it to the flow modification object.

| Attribute | Meaning |
|---|---|

| | |
|---|---|
| ~~in_port~~ | ~~The Physical or virtual port on the switch that the packet arrived from. This is not an application port – Think of this as the port in a switch where we plug in an ethernet cable. Do we know where this is in the defined network?~~ |
| dl_src | Ethernet/MAC source address (Type of EthAddr) |
| dl_dst | Ethernet/MAC destination address (Type of EthAddr) |
| dl_type | Ethertype / length (e.g. 0x0800 = IPv4) (Type of Integer) |
| nw_proto | IP protocol (e.g., 6 = TCP) or lower 8 bits of ARP opcode (Type of integer) |
| nw_src | IP source address (Type of String or IPAddr) |
| nw_dst | IP destination address (Type of String or IPAddr) |
| tp_src | TCP/UDP source application port (Type of Integer) |
| tp_dst | TCP/UDP destination application port (Type of Integer) |

Attributes may be specified either on a match object or during its initialization. That is, the following are equivalent:

```
matchobj = of.ofp_match(tp_src = 5, dl_dst = EthAddr("01:02:03:04:05:06"))
#.. or ..
matchobj = of.ofp_match() matchobj.tp_src
= 5
matchobj.dl_dst = EthAddr("01:02:03:04:05:06")
```

## IP Address Handling

IP addresses can be specified in many different ways. When crafting the rule, we can set a network address than can contain a subset of hosts on a particular network.

```
my_match.nw_src = "192.168.42.0/24"
my_match.nw_src = (IPAddr("192.168.42.0"), 24) # RECOMMEND TO NOT USE
```

IMPORTANT NOTE ABOUT IP ADDRESSES

TA Note: What isn't very clear by this documentation is that nw_* is expecting a network address. If we are calling out an IP Address like 10.0.1.1/32, it is an acceptable response to nw_*. However, if we are calling out a subnet like 10.0.1.0/24, the IP address portion of the response MUST BE the Network Address.

From Wikipedia: IP addresses are described as consisting of two groups of bits in the address: the most significant bits are the network prefix, which identifies a whole network or subnet, and the least significant set forms the host identifier, which specifies a particular interface of a host on that network. This division is used as the basis of traffic routing between IP networks and for address allocation policies. (https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing)

Thus for a /24 network, the first 24 bits of the address comprises the network address. Thus, it would be 10.0.1.0. For a /25 network, there would be two networks in the 10.0.1.x space – a 10.0.1.0/25 and a 10.0.1.128/25.

The implementation code does NOT need to convert the given IP Address into a network – we can assume that any given address in a possible configuration file must be valid. However, the configure.pol file MUST be using the proper form if we are using a CIDR notation other than /32. Why would we do this? To reduce the number of rules needed. We may use this for the 5th rule from Part 6.

Note that some fields have *prerequisites*. Basically this means that we can't specify higher-layer fields without specifying the corresponding lower-layer fields also. For example, we can not create a match on a TCP port without also specifying that we wish to match TCP traffic. And in order to match TCP traffic, we must specify that we wish to match IP traffic. Thus, a match with only tp_dst=80, for example, is invalid. We must also specify nw_proto=6 (TCP), and dl_type=0x800 (IPv4). If we violate this, we should get the warning message 'Fields ignored due to unspecified prerequisites'. For more information on this subject, see the FAQ entry "I tried to install a table entry but got a different one. Why?" as shown below:

FAQ Entry:

This question also presents itself as "What does the Fields ignored due to unspecified prerequisites warning mean?"

Basically this means that we specified some higher-layer field without specifying the corresponding lower-layer fields also. For example, we may have tried to create a match in which we specified only tp_dst=80, intending to capture HTTP traffic. We can't do this. To match TCP port 80, we must also specify that we intend to match TCP (nw_proto=6). And in order to match on the TCP protocol, we must also match on IPV4 type (dl_type=0x800).

## OpenFlow Actions

The final aspect needed to fully implement a flow modification object is the action. With this, we specify what we want done to a port. This can include forwarding, dropping, duplicating and redirecting, or modify the header parameters. For the purposes of this project, we are only dealing with forwarding of match traffic to its destination. But please note that for a Software Defined Network system, we can do all sorts of actions including round robin server, DDOS blocking, and many other possible options.

### Output

Forward packets out of a physical or virtual port. Physical ports are referenced to by their integral value, while virtual ports have symbolic names. Physical ports should have port numbers less than 0xFF00.

Structure definition:

```
class ofp_action_output (object):
def __init__ (self, **kw):
   self.port = None # Purposely bad -- require specification
```

**port (int) the output port for this packet. Value could be an actual physical switch port number or one of the ~~following virtual ports expressed as constants:~~**

- ~~of.OFPP_IN_PORT - Send back out the physical switch port the packet was received on. Except possibly OFPP_NORMAL, *this is the only way to send a packet back out its incoming port. (This redirects a packet back to the sender)*~~
- of.OFPP_NORMAL - Process via normal L2/L3 legacy switch configuration (i.e., send traffic to its destination without modification) – See https://study-ccna.com/layer-3-switch/ for information on how normal L2/L3 legacy switches work.
- of.OFPP_FLOOD – output to all OpenFlow ports except the input port and those with flooding disabled via the OFPPC_NO_FLOOD port config bit (generally, this is done for STP) This is an option where the packet will be forwarded to all physical or virtual switch ports except the source. This is not an ideal situation as it causes excessive traffic and could be used to create a denial of service attack because of the packet amplification that it may cause. (See DNS amplification attack for an example of the ramifications of this.
- of.OFPP_ALL - output all OpenFlow ports except the in port. This is similar to OFPP_FLOOD, except it sends to ports that had flooding disable. This may cause even more excessive traffic than OFPP_FLOOD
- of.OFPP_CONTROLLER - Send to the OpenFlow controller. This would send a packet to the switch hardware itself, and not to a particular host system.

## Example: Sending a FlowMod Object

The following example describes how to create a flow modification object including matching a destination IP Address, IP Type, and Destination IP Port, and setting an action that would redirect the matching packet out to physical switch port number 4 (note that we generally DO NO KNOW what physical switch port to use.

```
rule = of.ofp_flow_mod()
```

```
rule.match = of.ofp  match()
rule.match.dl  type = 0x800
rule.match.nw  dst = "192.168.101.101/32"
rule.match.nw  proto = 6
rule.match.tp  dst = 80
rule.priority = 42
rule.actions.append(of.ofp  action  output(port = of.OFPP  CONTROLLER))
```

Flow Modification Objects work as thus:

1. Packet enters the system and is examined by the Flow Modification Objects (1 for each rule in the configuration ruleset)
2. The packet will then be examined to see if the different header items match the items specified for that particular rule.
3. If the packet matches all of the applicable items, it is pulled from the stream for we to program an action for it (forward it, readdress it, change it).  If we don’t do an action for it, the package is essentially dropped.  If the packet does not match all of the applicable header items, it continues to the next Flow Modification rule to test it.  If it isn’t matched by any rules, it is passed on to the specific destination.

For this project, we are making a flow modification object and action while using a matching pattern that can match any or all of the different parameters  of the header.  Make the implementation generic.