Assignment 3
Please submit it by June 19

In this assignment, you will learn how to write network applications on POX. Assignment 2 introduced you to POX and l2_learning switch. The exercises are repeated here for your practice. After the exercise, you will be asked to create and submit a network application that implements Layer 2 Firewall that disables inbound and outbound traffic between two systems based on their MAC address. So, make sure that you follow each step carefully.

(Note: you can skip this section and start directly with the assignment at the end, if you feel confident and are already familiar with POX and its basic functions)

Overview

The network you'll use in this exercise includes 3 hosts and a switch with an OpenFlow controller (POX):

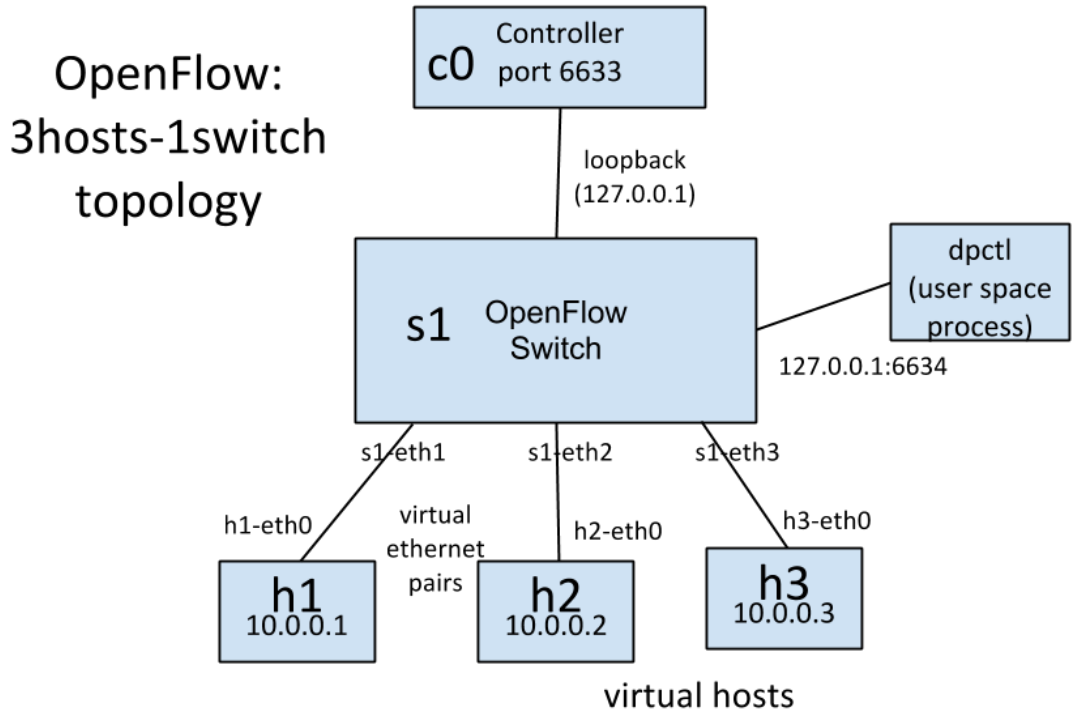

Figure 1: Topology for the Network under Test

POX is a Python based SDN controller platform geared towards research and education. For more details on POX, see https://openflow.stanford.edu/display/ONL/POX+Wiki

We're not going to be using the reference controller anymore, which is the default controller that Mininet uses during it simulation. Make sure that it's not running in the background:

```
$ ps -A | grep controller
```

If so, you should kill it either press Ctrl-C in the window running the controller program, or from the other SSH window:

```
$ sudo killall controller
```

You should also run `sudo mn -c` and restart Mininet to make sure that everything is clean and using the faster kernel switch: From you Mininet console:

```
mininet> exit
$ sudo mn -c
$ sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

The POX controller comes pre-installed with the provided VM image.

Now, run the basic hub example:

```
$ pox.py log.level --DEBUG forwarding.hub
```

This tells POX to enable verbose logging and to start the hub component.

The switches may take a little bit of time to connect. When an OpenFlow switch loses its connection to a controller, it will generally increase the period between which it attempts to contact the controller, up to a maximum of 15 seconds. Since the OpenFlow switch has not connected yet, this delay may be anything between 0 and 15 seconds. If this is too long to wait, the switch can be configured to wait no more than N seconds using the --max-backoff parameter. Alternately, you exit Mininet to remove the switch(es), start the controller, and then start Mininet to immediately connect.

Wait until the application indicates that the OpenFlow switch has connected. When the switch connects, POX will print something like this:

```
INFO:openflow.of_01:[Con 1/1] Connected to 00-00-00-00-00-01
DEBUG:samples.of_tutorial:Controlling [Con 1/1]
```

Verify Hub behavior with tcpdump

Now verify that hosts can ping each other, and that all hosts see the exact same traffic - the behavior of a hub. To do this, we'll create xterms for each host and view the traffic in each. In the Mininet console, start up three xterms:

```
mininet> xterm h1 h2 h3
```

Arrange each xterm so that they're all on the screen at once. This may require reducing the height to fit a cramped laptop screen.

In the xterms for h2 and h3, run `tcpdump`, a utility to print the packets seen by a host:

```
# tcpdump -XX -n -i h2-eth0
```

and respectively:

```
# tcpdump -XX -n -i h3-eth0
```

In the xterm for h1, send a ping:

```
# ping -c1 10.0.0.2
```

The ping packets are now going up to the controller, which then floods them out all interfaces except the sending one. You should see identical ARP and ICMP packets corresponding to the ping in both xterms running tcpdump. This is how a hub works; it sends all packets to every port on the network.

Now, see what happens when a non-existent host doesn't reply. From h1 xterm:

```
# ping -c1 10.0.0.5
```

You should see three unanswered ARP requests in the tcpdump xterms. If your code is off later, three unanswered ARP requests is a signal that you might be accidentally dropping packets.

You can close the xterms now.

Now, lets look at the hub code:

```
from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.util import dpidToStr

log = core.getLogger()


def _handle_ConnectionUp (event):
  msg = of.ofp_flow_mod()
  msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
  event.connection.send(msg)
  log.info("Hubifying %s", dpidToStr(event.dpid))

def launch ():
  core.openflow.addListenerByName("ConnectionUp", _handle_ConnectionUp)

  log.info("Hub running.")
```

Table 1. Hub Controller

Useful POX API's

- `connection.send( ... )` function sends an OpenFlow message to a switch.

When a connection to a switch starts, a ConnectionUp event is fired. The above code invokes a `_handle_ConnectionUp ()` function that implements the hub logic.

- `ofp_action_output` class

This is an action for use with `ofp_packet_out` and `ofp_flow_mod`. It specifies a switch port that you wish to send the packet out of. It can also take various "special" port numbers. An example of this, as shown in Table 1, would be `OFPP_FLOOD` which sends the packet out all ports except the one the packet originally arrived on.

Example. Create an output action that would send packets to all ports:

```
out_action = of.ofp_action_output(port = of.OFPP_FLOOD)
```

- `ofp_match` class (not used in the code above but might be useful in the assignment)

Objects of this class describe packet header fields and an input port to match on. All fields are optional -- items that are not specified are "wildcards" and will match on anything.

Some notable fields of ofp_match objects are:

- `dl_src` - The data link layer (MAC) source address
- `dl_dst` - The data link layer (MAC) destination address
- `in_port` - The packet input switch port

Example. Create a match that matches packets arriving on port 3:

```
match = of.ofp_match()
match.in_port = 3
```

- `ofp_packet_out` OpenFlow message (not used in the code above but might be useful in the assignment)

The `ofp_packet_out` message instructs a switch to send a packet. The packet might be one constructed at the controller, or it might be one that the switch received, buffered, and forwarded to the controller (and is now referenced by a `buffer_id`).

Notable fields are:

- `buffer_id` - The buffer_id of a buffer you wish to send. Do not set if you are sending a constructed packet.
- `data` - Raw bytes you wish the switch to send. Do not set if you are sending a buffered packet.
- `actions` - A list of actions to apply (for this tutorial, this is just a single `ofp_action_output` action).
- `in_port` - The port number this packet initially arrived on if you are sending by `buffer_id`, otherwise `OFPP_NONE`.

Example. send_packet() method:

```
def send_packet (self, buffer_id, raw_data, out_port, in_port):
  """
   Sends a packet out of the specified switch port.
   If buffer_id is a valid buffer on the switch, use that.  Otherwise,
   send the raw data in raw_data.
   The "in_port" is the port number that packet arrived on.  Use
   OFPP_NONE if you're generating this packet.
   """
  msg = of.ofp_packet_out()
  msg.in_port = in_port
  if buffer_id != -1 and buffer_id is not None:
    # We got a buffer ID from the switch; use that
    msg.buffer_id = buffer_id
  else:
    # No buffer ID from switch -- we got the raw data
    if raw_data is None:
      # No raw_data specified -- nothing to send!
      return
    msg.data = raw_data

  action = of.ofp_action_output(port = out_port)
  msg.actions.append(action)

  # Send message to switch

  self.connection.send(msg)
```

Table 2: Send Packet

- `ofp_flow_mod` OpenFlow message

This instructs a switch to install a flow table entry. Flow table entries match some fields of incoming packets, and executes some list of actions on matching packets. The actions are the same as for `ofp_packet_out`, mentioned above (and, again, for the tutorial all you need is the simple `ofp_action_output` action). The match is described by an `ofp_match` object.

Notable fields are:

- `idle_timeout` - Number of idle seconds before the flow entry is removed. Defaults to no idle timeout.
- `hard_timeout` - Number of seconds before the flow entry is removed. Defaults to no timeout.
- `actions` - A list of actions to perform on matching packets (e.g., `ofp_action_output`)
- `priority` - When using non-exact (wildcarded) matches, this specifies the priority for overlapping matches. Higher values are higher priority. Not important for exact or non-overlapping entries.
- `buffer_id` - The `buffer_id` of a buffer to apply the actions to

immediately. Leave unspecified for none.

- `in_port` - If using a buffer_id, this is the associated input port.
- `match` - An `ofp_match` object. By default, this matches everything, so you should probably set some of its fields!

Example. Create a flow_mod that sends packets from port 3 out of port 4.

```
fm = of.ofp_flow_mod()
fm.match.in_port = 3
fm.actions.append(of.ofp_action_output(port = 4))
```

Verify Switch behavior with tcpdump

This time, let's verify that hosts can ping each other when the controller is behaving like a Layer 2 learning switch. Kill the POX controller by pressing Ctrl-C in the window running the controller program and run the l2_learning example:

```
$ pox.py log.level --DEBUG forwarding.l2_learning
```

Like before, we'll create xterms for each host and view the traffic in each. In the Mininet console, start up three xterms:

```
mininet> xterm h1 h2 h3
```

Arrange each xterm so that they're all on the screen at once. This may require reducing the height of to fit a cramped laptop screen.

In the xterms for h2 and h3, run `tcpdump`, a utility to print the packets seen by a host:

```
# tcpdump -XX -n -i h2-eth0
```

and respectively:

```
# tcpdump -XX -n -i h3-eth0
```

In the xterm for h1, send a ping:

```
# ping -c1 10.0.0.2
```

Here, the switch examines each packet and learn the source-port mapping. Thereafter, the source MAC address will be associated with the port. If the destination of the packet is already associated with some port, the packet will be sent to the given port, else it will be flooded on all ports of the switch.

You can close the xterms now.

The code for l2_learning application is provided under ~/pox/pox/forwarding
Please see POX Wiki (https://openflow.stanford.edu/display/ONL/POX+Wiki) for more

information.

Assignment

Background

A Firewall is a network security system that is used to control the flow of ingress and egress traffic usually between a more secure local-area network (LAN) and a less secure wide-area network (WAN). The system analyses data packets for parameters like L2/L3 headers (i.e., MAC and IP address) or performs deep packet inspection (DPI) for higher layer parameters (like application type and services etc) to filter network traffic. A firewall acts as a barricade between a trusted, secure internal network and another network (e.g. the Internet) which is supposed to be not very secure or trusted.

In this assignment, your task is to implement a layer 2 firewall that runs alongside the MAC learning module on the POX OpenFlow controller. The firewall application is provided with a list of MAC address pairs i.e., access control list (ACLs). When a connection establishes between the controller and the switch, the application installs flow rule entries in the OpenFlow table to disable all communication between each MAC pair.

Network Topology

Your firewall should be agnostic of the underlying topology. It should take MAC pair list as input and install it on the switches in the network. To make things simple, we will implement a less intelligent approach and will install rules on **all** the switches in the network.

Handling Conflicts

POX allows running multiple applications concurrently i.e., MAC learning can be done in conjunction with firewall, but it doesn't automatically handles rule conflicts. You have to make sure, yourself, that conflicting rules are not being installed by the two applications e.g., both applications trying to install a rule with same src/dst MAC but with different actions. For this assignment, setting the priority of event listeners for each application would do the trick. More information on how to do this can be found in the POX Wiki under the mac_blocker section.

Understanding the Code

To start this exercise, download the following files from conneX->Resources (Assignment 3 folder)

- `firewall.py`: a skeleton class which you will update with the logic for installing firewall rules.
- `firewall-policies.csv`: **an example list of MAC pairs (i.e., policies) read as input by the firewall application. Modify as necessary**.

The `firewall.py` is populated with a skeleton code. It consists of a `firewall` class that has

a `_handle_ConnectionUp` function. It also has a global variable, `policyFile`, that holds the path of the `firewall-policies.csv` file. Whenever a connection is established between the POX controller and the OpenFlow switch the `_handle_ConnectionUp` functions gets executed.

Your task is to read the policy file and update the `_handle_ConnectionUp` function. The function should install rules in the OpenFlow switch that drop packets whenever a matching src/dst MAC address (for any of the listed MAC pairs) enters the switch. (Note: make sure that you handle the conflicts carefully. Follow the technique described in the section above)

Testing your Code

Once you have your code, copy the `firewall.py` to the `~/pox/pox/misc` directory on your VM. Also in the same directory, create the following file:

```
$ cd ~/pox/pox/misc
$ touch firewall-policies.csv
```

and copy the following lines in it:

```
id,mac_0,mac_11,00:00:00:00:00:01,00:00:00:00:00:02
```

This will cause the firewall application to install a flow rule entry to disable all communication between host (h1) and host (h2).

Run POX controller:

```
$ cd ~
$ pox.py forwarding.l2_learning misc.firewall &
```

This will run the controller **with both MAC learning and firewall application**.

Now run mininet:

```
$ sudo mn --topo single,3 --controller remote --mac
```

In mininet try to ping host (h2) from host (h1):

```
mininet> h1 ping -c1 h2
```

What do you see? If everything has be done and setup correctly then host (h1) should not be able to ping host (h2).

Now try pinging host (h3) from host (h1):

```
mininet> h1 ping -c1 h3
```

What do you see? Host (h1) is able to ping host (h3) as there is no flow rule entry installed in the network to disable the communication between them.

Submitting your Code

CSC 485A Students: Try your firewall on the Datacenter topology of Assignment 1. Set rules to block traffic from h1 to h5 and h3 to h8. Please submit your firewall.py and firewall-policies.py on conneX as well as submit a report on your results.

CSC 586A Students: Extend your code so that firewall will block traffic based on IPv4 destination address. Try your firewall on the Datacenter topology of Assignment 1. Set rules to block traffic from h1 to h5 and h3 to h8. Please submit your firewall.py and firewall-policies.py on conneX as well as submit a report on your results.

Note, if during the execution `submit.py` script crashes for some reason or you terminate it using CTRL+C, make sure to clean mininet environment using:

```
$ sudo mn -c
```

Also, if it still complains about the controller running. Execute the following command to kill it:

```
$ sudo fuser -k 6633/tcp
```

---

* These instructions are adapted from from SDN course offered by Dr. Nick Feamster, mininet.org and POX Wiki