Assignment 7: SDN and Firewalls

Goal:

The purpose of this assignment is to give you practical experience using Software Defined Networking (SDN) controllers in Python. As the videos have discussed, software defined networking separates the data and control planes so that control of the router can be abstracted up to a controller.

In this assignment, you will implement a firewall by writing code for an SDN controller in Python. More specifically, you will write code to add firewall rules in both POX and Pyretic, two SDN languages for Python. By developing a controller in each language, you will see how different abstractions offer different strengths and weaknesses.

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 Pyretic and POX runtimes. 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 static flow rule entries in the OpenFlow table to disable all communication between each MAC pair.

Overview

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

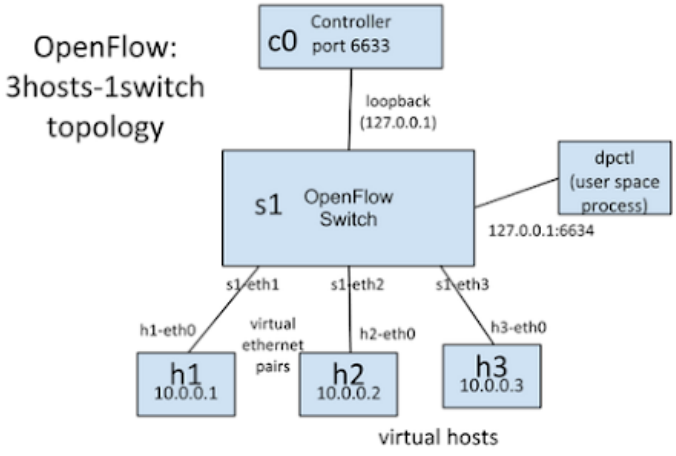

Figure 1: Topology for the Network under Test

POX is a Python based SDN controller platform geared towards research and education. For more details on POX, see About POX or POX Documentation on http://www.NOXRepo.org.

For more details on Pyretic, see http://www.frenetic-lang.org/pyretic/.

As shown above, the network topology for this assignment will consist of an OpenFlow switch, an OpenFlow controller, and 3 hosts. You will create a firewall which restricts communication between the 3 hosts.

To complete this assignment, you will need to add rules to the controller which drop packets with certain source and destination MAC addresses. These MAC address pairs will be provided in the form of a CSV file and the code to read the MAC address pairs into Python has already been provided. For both the POX and Pyretic code, a for loop has already been written to process each MAC address pair. Your job is insert a rule into the controller which will drop offending packets with the given MAC address pair. 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. This should consist of 5-10 lines of code for each controller and the relevant sections have been marked with TODO comments.

Steps

1. Update your repo by changing into your gt-cs6250 directory and running the following command: `git commit -a -m "saving work"`, then `$ git pull --rebase origin master`. If you encounter problems with the pyretic_switch.py code, please run this command again to get the latest version of the script.
2. Run the hub example

    1. In this example, we will explore the difference between hubs and switches by capturing the traffic between nodes when either a hub or a switch is used.
    2. Ensure that no controllers are running by executing `$ sudo killall controller`, `$ sudo fuser -k 6633/tcp` and `$ sudo mn -c` to clean up mininet
    3. ssh into the VM using the -X flag, so `ssh -X mininet@<mininet-addr>` where <mininet-addr> is the address of your mininet VM.
    4. In the ssh terminal, run the command `sudo mn --topo single,3 --mac --switch ovsk --controller remote` to start mininet with the topology from Figure 1.
    5. In the virtualbox terminal, run the command `$ 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]

    6. Open up terminals on each of the hosts by executing the following command in the ssh terminal. This should open 3 new terminals `mininet> xterm h1 h2 h3`

    7. Now, run tcpdump on hosts h2 and h3 to view the visible traffic at those hosts. In terminals 2 run `tcpdump -XX -n -i h2-eth0` and in terminal 3 run `tcpdump -XX -n -i h3-eth0`.
    8. Next, send a ping from h1 and notice how the hub forwards traffic out all ports except for the port which it heard the message from. Run `ping -c1 10.0.0.2` and you should see that the same ARP and ICMP packets should be visible on hosts 2 and 3.
    9. Finally, try sending a ping to a nonexistent host. Run `ping -c1 10.0.0.5` and notice that hosts h2 and h3 both see 3 unanswered ARP requests.
    10. Close each of the xterminal windows
    11. For more information, please see the POX code in Appendix A
3. Run the switch example.

    In the last example, you saw that a hub forwards traffic out of all of its ports. Though you did not interact with the code, the previous example used the POX controller to create the hub. In this example, you will see how a switch only forwards traffic out specific ports. To accomplish this task, you will use the Pyretic controller to create a learning switch.

    1. First, ensure that the POX controller is not running in the background by running `$ sudo fuser -k 6633/tcp` `$sudo mn -c`.
    2. Next, copy the given Pyretic switch controller to the pyretic directory by running `cp pyretic_switch.py ~/pyretic/pyretic/examples/` from your code directory, then moving into that directory by executing the following in the ssh window: `$ cd ~/pyretic/pyretic/examples`

3. Now start the mininet topology in the ssh window by running
   `$ sudo mn --topo single,3 --mac --switch ovsk --controller remote`
4. Now, run the switch example: `$ pyretic.py -v high pyretic.examples.pyretic_switch`
5. As in hub example, view the traffic at each node by running `mininet> xterm h1 h2 h3`, then `tcpdump -XX -n -i h2-eth0` in terminal 2, and `tcpdump -XX -n -i h3-eth0` in terminal 3.
6. Finally, run `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. To get a better idea of the code for this switch, see Appendix B
7. Close each of the xterminal windows

4. Create a firewall with the POX controller

   Now that you have seen an example POX controller, it's time to write your own controller to implement a firewall. Please note that your firewall should block traffic in **BOTH** directions. This means that you should block the flow from source to destination and the flow from destination to source.

   1. Fill in the sections of pox_firewall.py marked with a TODO comment
   2. Ensure that no controllers are running by executing `$ sudo fuser -k 6633/tcp` and `$ sudo mn -c` to clean up mininet
   3. Copy the test firewall policies and your code so that POX can use them
      `$cp firewall-policies.csv pox_firewall.py ~/pox/pox/misc`
   4. Run the controller in a terminal with both MAC learning and firewall applications
      `$pox.py forwarding.l2_learning misc.pox_firewall`
   5. Now run mininet: `$sudo mn --topo single,3 --controller remote --mac`
   6. Next, try pinging host 2 from host 1. If h1 cannot ping h2, then the firewall is working correctly: `mininet> h1 ping -c1 h2`
   7. Finally, verify that h1 and h3 can still talk to each other. If h1 cannot ping h3, then something is wrong:
      `mininet> h1 ping -c1 h3`

5. Create a firewall with the Pyretic controller.
   1. Fill in the section of pyretic_firewall.py marked with a TODO comment. Please note that your firewall should block traffic in **BOTH** directions. This means that you should block the flow from source to destination and the flow from destination to source.
   2. Ensure that no controllers are running by executing `$ sudo fuser -k 6633/tcp` and `$ sudo mn -c` to clean up mininet
   3. Copy the test firewall policies and your code so that Pyretic can use them
      `$cp firewall-policies.csv pyretic_firewall.py ~/pyretic/pyretic/examples`
   4. Run the controller in a terminal with both MAC learning and firewall applications
      `$pyretic.py pyretic.examples.pyretic_firewall`
   5. Now run mininet: `$sudo mn --topo single,3 --controller remote --mac`
   6. Next, try pinging host 2 from host 1. If h1 cannot ping h2, then the firewall is working correctly: `mininet> h1 ping -c1 h2`
   7. Finally, verify that h1 and h3 can still talk to each other. If h1 cannot ping h3, then something is wrong:
      `mininet> h1 ping -c1 h3`

6. Submit your assignment

Appendix A: POX

Now that we have seen POX in action, lets explore the POX code.

```
from pox.core import core
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.priority = 20
  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))
```

Appendix B: Pyretic

Let's have a look at the pyretic_switch code: (it's pretty self explanatory)

```
from pyretic.lib.corelib import *
from pyretic.lib.std import *

# The @dynamic decorator means that the function below will create a new dynamic
# policy class with the name "act_like_switch"
@dynamic
def act_like_switch(self):
    """
    Implement switch-like behavior.
    """

   # Set up the initial forwarding behavior for your mac learning switch to flood
   # all packets
   self.forward = flood()

   # Set up a query that will receive new incoming packets
   self.query = packets(limit=1,group_by=['srcmac','switch'])
```

```
   # Set the initial internal policy value (each dynamic policy has a member 'policy'
   # when this member is assigned, the dynamic policy updates itself)
   self.policy = self.forward + self.query
   # hint: '+' operator on policies is shorthand for parallel composition

   # Function to take each new packet pkt and update the forwarding policy
   # so subsequent incoming packets on this switch whose dstmac matches pkt's srcmac
   # (accessed like in a dictionary pkt['srcmac']) will be forwarded out  pkt's inport
   # (pyretic packets are located, so this value is accessed just like srcmac -
   # i.e., p['inport'])
   def learn_from_a_packet(pkt):
       # Set the forwarding policy
       self.forward = if_(match(dstmac=pkt['srcmac'],
                           switch=pkt['switch']), fwd(pkt['inport']),
                      self.policy)  # hint use 'match', '&', 'if_', and 'fwd'
       # Update the policy
       self.policy = self.forward + self.query # hint you've already written this
       print self.policy

  # learn_from_a_packet is called back every time our query sees a new packet
  self.query.register_callback(learn_from_a_packet)

def main():
   return act_like_switch()
```

Table 2. switch application

Useful Pyretic policies

- match(f=v): filters only those packets whose header field f's value matches v
- ~A: negates a match
- A & B: logical intersection of matches A and B
- A | B: logical union of matches A and B
- fwd(a): forward packet out port a
- flood(): send all packets to all ports on a network minimum spanning tree, except for the input port
- A >> B: A's output becomes B's input
- A + B: A's output and B's output are combined
- if_(M,A,B): if packet filtered by M, then use A, otherwise use B

This revision was added on 2014/03/05 23:06:20.

POPULAR NANODEGREE PROGRAMS +

STUDENT RESOURCES +

UDACITY +

INQUIRIES +

Nanodegree is a trademark of Udacity
© 2011–2018 Udacity, Inc.