SDN – Pox Controller Firewall Implementation

Introduction

This assignment required the implementation of an OpenFlow controller with additional firewall functionality, which would filter on both physical and logical addresses of the hosts. The topology consisted of a three layer hierarchical model, with seven switches in total, one at the core, two at the aggregation layer and four at the access layer. Each access switch had two connected hosts.

The OpenFlow controller was implemented in Python using POX, a platform that provides an OpenFlow API wrapper. For this assignment, I implemented a single logically centralized controller that accepts requests from all seven switches. In a production environment attempting to configure SDN, this would most likely not be sufficient.

Setup

To script and launch the topology itself, Mininet, a network emulator for spinning up virtual hosts was used. Each host was assigned an IP address statically when launched (*Figure 1*), and the *--mac* command used when launching the topology in order to map a simple human readable friendly Mac address to each host's logical address.

```
    # Add hosts and switches
    for i in range(1, 9):
        ip = '10.0.0.' + str(i)
        host.append(self.addHost('h' + str(i), ip=ip))
```

Figure 1 - Designating logical address to each host

Controller

When the POX controller is invoked from the command line, it calls the launch method within the script. Within this function, I create an object that functions as a controller, and register it as a POX component.

In the same Python script, I have created another class that acts like a firewall. This class reads from a csv file, and has an attribute that is a nested dictionary of firewall rules. One dictionary for Mac address rules and the other for IP address rules. Finally, I create a Firewall object within the controller, essentially allowing the controller access to these rules.

Communicating with a switch

Within the POX platform, event handling is done in a publish-subscribe type fashion. We can subscribe to specific event of interest and will be alerted when these events are fired. POX is acting as a controller for our seven OpenFlow switches, so obviously it needs to be able to communicate with them. Messages coming from each switch show up as an event and the controller speaks to and manages each switch using the OpenFlow wire protocol or in the POX implementation, through the *pox.openflow.libopenflow_01* module.

Adding Firewall Rules

An OpenFlow switch consists of flow tables and a group table for packet lookups and forwarding. To add firewall rules to each switch, we need to use a FlowMod message to modify the state of a switch by adding to its flow table. Each FlowMod message begins with an OpenFlow header and includes a match field.

Each flow table entry consists of a match field and a set of instructions as to what to do with a packet that matches. When a match is found, the instruction is executed. When a switch is loaded and makes an initial connection to the OpenFlow controller a "ConnectionUp" event is fired. We listen for this event and it is at this point that we add the firewall rules to the switch.

Figure 2 - Adding firewall rules

```
    def _handle_ConnectionUp(self, event):
        firewall_table = self.firewall.firewall
        # Insert flows into switch with FlowMod and match on criteria
        for key, value in firewall_table.iteritems():
            if isinstance(value, dict):
                for src, dst in value.iteritems():
                    if key is 'mac':
                        # Add Flows both way to block this on switch conn
                        self.add_ethernet_rule(event.connection, src, dst)
                        self.add_ethernet_rule(event.connection, dst, src)

                    if key is 'ip':
                        # Match on IP type field in frame
                        msg = of.ofp_flow_mod()
                        msg.match.dl_type = 0x800
                        msg.match.nw_src = IPAddr(src)
                        msg.match.nw_dst = IPAddr(dst)
                        event.connection.send(msg)

    """ Build FlowMod rule for switch flow and send
        it out to the switch the connection came in on"""

    def add_ethernet_rule(self, conn, src, dst):
        msg = of.ofp_flow_mod()
        msg.match.dl_src = EthAddr(src)
        msg.match.dl_dst = EthAddr(dst)
        conn.send(msg)
```

A code snippet of adding these firewall rules to each switch is shown above.

1. The event gets fired.
2. Iterate through the nested dictionary of firewall rules.
3. If the dictionary holds physical addresses:
    I. Create a FlowMod object
    II. Match on the Ethernet source address
    III. Match on the Ethernet destination address
    IV. Repeat, reversing the source and destination addresses to block both ways.
    V. Send out using the connection to the switch that the event came in on. Updating this switches flow table with message value.

4. If the dictionary holds logical addresses:
    I. Match on Ethernet frame types which contain IPv4
    II. Match on source IP
    III. Match on destination IP
    IV. Send out using the connection to the switch that the event came in on.

The screenshot below in *Figure 3* shows the flows from a switch being dumped to the console through Mininet, both before and after a connection was made to the controller.

Figure 3 – Dumping flow entries for switch c1

```
mininet> c1 dpctl dump-flows tcp:127.0.0.1:6634
stats_reply (xid=0x19d1bde4): flags=none type=1(flow)
mininet> c1 dpctl dump-flows tcp:127.0.0.1:6634
stats_reply (xid=0xbf687242): flags=none type=1(flow)
  cookie=0, duration_sec=11s, duration_nsec=532000000s, table_id=0, priority=32768, n_packets=0, n_bytes=0, idle_timeout=0,hard_timeout=0,dl_src=00:00:00:00:00:07,dl_dst=00:00:00:00:00:03,actions=
  cookie=0, duration_sec=11s, duration_nsec=532000000s, table_id=0, priority=32768, n_packets=0, n_bytes=0, idle_timeout=0,hard_timeout=0,dl_src=00:00:00:00:00:01,dl_dst=00:00:00:00:00:08,actions=
  cookie=0, duration_sec=11s, duration_nsec=532000000s, table_id=0, priority=32768, n_packets=0, n_bytes=0, idle_timeout=0,hard_timeout=0,dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:07,actions=
  cookie=0, duration_sec=11s, duration_nsec=532000000s, table_id=0, priority=32768, n_packets=0, n_bytes=0, idle_timeout=0,hard_timeout=0,dl_src=00:00:00:00:00:08,dl_dst=00:00:00:00:00:01,actions=
  cookie=0, duration_sec=11s, duration_nsec=568000000s, table_id=0, priority=65000, n_packets=10, n_bytes=410, idle_timeout=0,hard_timeout=0,dl_dst=01:23:20:00:00:01,dl_type=0x88cc,actions=CONTROLLER:65535
  cookie=0, duration_sec=11s, duration_nsec=532000000s, table_id=0, priority=32768, n_packets=0, n_bytes=0, idle_timeout=0,hard_timeout=0,ip,nw_src=10.0.0.4,nw_dst=10.0.0.6,actions=
```

We can see these match the Firewall rules that were added to the firewall.csv file, shown in *Figure 4.*

Figure 4 - CSV file with firewall rules

```
id,src,dst
mac,00:00:00:00:00:03,00:00:00:00:00:07
mac,00:00:00:00:00:01,00:00:00:00:00:08
ip,10.0.0.4,10.0.0.6
```

In hindsight they probably should have included an action to just drop the packets but it blocked them regardless.

Handling Packets

When there is no flow entry specified in the flow table for each switch it hands the packet over to the controller because it doesn't know what to do with it. The existing code, which was included in the l2_pairs controller worked so there was no sense modifying it, as it is relatively straightforward anyway.

The controller keeps a map of key value pairs mapping the switch and Mac addresses to the port on that switch that we last saw a packet coming from. When a packet comes in and fires the "PacketIn" event, the controller checks to see if there is an entry for the destination Mac address, mapping to a port stored in the table.

If no entry exists in the controllers' table, we create an OpenFlow packet out message instructing the switch to send out packets. In this case, the message will contain within the data field the event ofp, the OpenFlow message that caused this event. We then tell the switch to broadcast this message, the original packet, out all ports except the one it came in on originally.

If an entry does exist, we know the source and destination makes addresses, so we add flows to match these rules so the switch can send directly next time and update the flow tables for both of these switches with this entry.

Conclusion

The screenshot in *Figure 5* below shows that this implementation was a success. A *pingall* command issued from the Mininet CLI delivers and fails were expected as in relation to the rules in the firewall.csv file shown in *Figure 4*.

Figure 5 - Results from pingall command