OpenFlow Firewall and NAT Devices

OpenFlow Firewall and NAT Devices
Step-by-step Instructions

Overview:

This is a very simple tutorial with two topologies demonstrating an OpenFlow Firewall and an OpenFlow NAT.

Prerequisites:

For this tutorial you need a GENI Experimenter Portal account and be a member of at least one project.

- If you don't have an account yet sign up!

Tools:

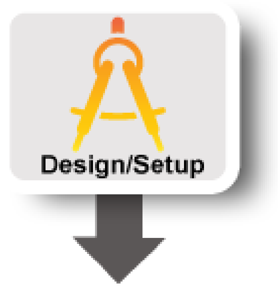
All the tools will already be installed at your nodes. For your reference we are going to use:

- A Ryu controller.

Where to get help:

For any questions or problem with the tutorial please email geni-users@googlegroups.com

Step-by-step Instructions



Step 1: Get Ready:

The first thing we need to do is login to the portal.

a. Go to the GENI Experimenter Portal and click the **Use GENI** button.
b. From the Drop Down menu select your institution. If you got an account through the GENI Identity Provider, please select **GENI Project Office**.
   Tip: Start typing the name of your institution and see the list become smaller.
c. You will be transferred to the Login Page of your institution. **Fill in** your username and password.

Step 2: Launch your experiment:

a. At the portal home page click the **+New slice** button.
   Tip: If you are not a member of any project and you don't know how to procede, email us
b. Name your slice **xxxfw** (where xxx are your initials), since slice names within a project must be unique. If you like, enter a description of the slice (optional).
c. Click **Create slice** .
d. Once the slice page loads, click the **Add Resources** button.
   NOTE: If you get a warning about not having uploaded ssh keys just follow the instructions on providing an ssh key before you proceed.
e. Scroll down to the **Choose RSpec** section, and from the drop down list select the existing RSpec labeled

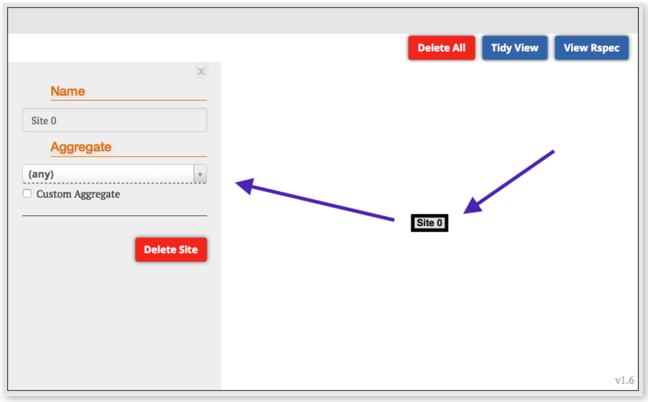
OpenFlow Firewall.

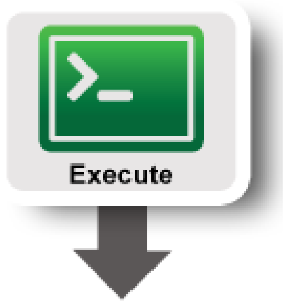
f. Notice that a graphical representation of the proposed topology appears on the canvas above the Choose RSpec section. Click the **Site 1** node, and then select **any *InstaGENI* aggregate** (any with *InstaGENI* in it's name) from the *Site* drop down list which appears on the left.

g. Next, click the **Reserve Resources** near the button of the page. After a few moments a results page should display details for the resources which have been reserved and are now being provisioned.

h. Click Home on the top menu bar, and repeat the above steps to create a second slice named **xxxnat** (where xxx are your initials) which instead makes use of the **OpenFlow NAT** RSpec.

i. Click Home on the top menu bar, and then select one of your slices. In the canvas you should see your network. Once your resources are imaged and booted, each associated icon on the canvas will turn green, indicating it is *ready* to be used. Be patient, this will likely take several minutes to complete.

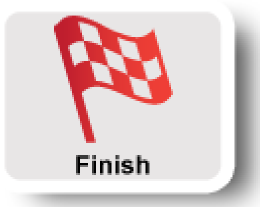




Step 3: OpenFlow Network Devices

You have reserved two topologies on different slices. In each of them you will run a different controller on an OVS switch to turn the switch into either a firewall or a NAT respectively.

1. Follow the detailed steps for the Firewall controller.
2. Follow the detailed steps for the NAT controller.



Step 4: Cleanup experiment:

After you are done with your experiment, you should always release your resources so that other experimenters can use the resources. In order to cleanup your slice :

a. Press the **Delete** button in the bottom of your Jacks canvas.

Wait and after a few moments all the resources will have been released and you will have an empty canvas again. Notice that your slice is still there. There is no way to delete a slice, it will be removed automatically after its expiration date, but remember that a slice is just an empty container so it doesn't take up any resources.

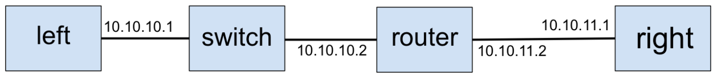
Attachments

- openflow-nat-fw-combo.rspec (5.1 KB) - added by *sedwards@bbn.com* 12 months ago.

OpenFlow Firewall

This exercise is based on as assignment by Sonia Famy, Ethan Blanton and Sriharsha Gangam of Purdue University.

For this experiment we will run an OpenFlow Firewall.



a. **Log into `switch`** and run the following commands to download and run the firewall controller:

   ```
   sudo pip install oslo.config
   ```

b. Run a simple learning switch controller:

   ```
   cd /tmp/ryu
   ./bin/ryu-manager --verbose ryu/app/simple_switch.py
   ```

c. To verify simple connectivity, **log into `right`** in a separate ssh terminal and ping `left`

   ```
   ping left
   ```

   Notice the printouts of the ryu simple switch controller.
d. Back in the **switch** ssh session, stop your controller with Ctrl-c and remove all your flows using the following command:

   ```
   sudo ovs-ofctl del-flows br0
   ```

   (optinal) Notice that you can no longer ping left from right.
e. On the **switch** ssh session, make your switch into a firewall by downloading and running the appropriate Ryu controller:

   ```
   cd
   wget http://www.gpolab.bbn.com/exp/OpenFlowExampleExperiment/ryu/gpo-ryu-firewall.tar.gz
   tar xvfz gpo-ryu-firewall.tar.gz
   cd gpo-ryu-firewall/
   /tmp/ryu/bin/ryu-manager simple_firewall.py
   ```

   WARNING If at some point your controller prints an error, kill it (ctrc-c) and start it again.
f. On the **`right`** ssh session run a `nc` server:

   ```
   nc -l 5001
   ```

g. **Log into `left`** and run a `nc` client:

   ```
   nc 10.10.11.1 5001
   ```

h. Type some text in `left` and it should appear in `right` and vise versa.
i. In the terminal for `switch` you should see messages similar to those below about the flow being passed or not:

   ```
   Extracted rule {'sport': '57430', 'dport': '5001', 'sip': '10.10.10.1', 'dip': '10.10.11.1'}
   Allow Connection rule {'dport': '5001', 'dip': '10.10.11.1', 'sip': '10.10.10.1', 'sport': 'any'}
   ```

j. Type `CTRL-C` (on left or right) to kill `nc`.
k. Run a `nc` server on port 5002, then 5003.
   - Compare the observed behavior to the contents of `~/gpo-ryu-firewall/fw.conf`. *Does the behavior match the configuration file?*
   - Stop the Firewall controller and run a simple switch controller. Is there any traffic being blocked now? Don't forget to delete the flows after you stop the controller
   - Feel free to modify the configuration file to allow more traffic.

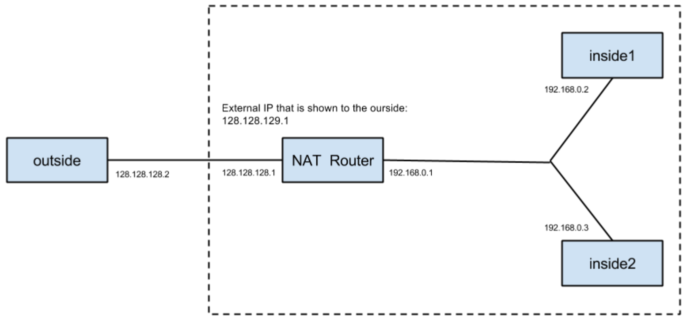
Return to the main page

Attachments

- Firewall.png (19.1 KB) - added by *sedwards@bbn.com* 12 months ago.
- Firewall-2.png (20.8 KB) - added by *sedwards@bbn.com* 12 months ago.

OpenFlow NAT Router

In this section, we are going to build a router for a network with a private address space that needs a one-to-many NAT (IP Masquerade) for some reason (e.g. short on public IP space or security) using OpenFlow. As shown in the figure below, hosts `inside1` and `inside2` are part of the private network, while host `outside` is outside. The LAN has only one public IP — **128.128.129.1**. (The external IPs we use, 128.128.128.0/24 and 128.128.129.0/24, are just an example. If your public IP happens to be in this subnet, change them to others.)



1 Test reachability before starting controller

1.1 Login to your hosts

To start our experiment we need to ssh into all of our hosts. Depending on which tool and OS you are using there is a slightly different process for logging in. If you don't know how to SSH to your reserved hosts take a look in this page. Once you have logged in, follow the rest of the instructions.

1.1a Install some software

On the NAT node run:

```
wget https://bootstrap.pypa.io/ez_setup.py -O - | sudo python
sudo pip install oslo.config
```

1.2 Test reachability

a. First we start a ping from `inside1` to `inside2`, which should work since they are both inside the same LAN.

```
inside1:~$ ping 192.168.0.3 -c 10
```

b. Then we start a ping from `outside` to `inside1`, which should timeout as there is no routing information in its routing table. You can use `route -n` to verify that.

```
outside:~$ ping 192.168.0.2 -c 10
```

c. Similarly, we cannot ping from `insideX` to `outside` (128.128.128.2).

d. You can also use Netcat (`nc`) to test reachability of TCP and UDP, as in the OpenFlow Firewall tutorial. The behavior should be the same (no connection).

2 Start controller to enable NAT

2.1 Access a server from behind the NAT

You can try to write your own controller to implement NAT. However, we've provided you a functional controller, which is a file called `nat.py` under `/tmp/ryu/` .

a. Start the controller on `NAT` host:

```
nat:~$ cd /tmp/ryu/
nat:~$ ./bin/ryu-manager nat.py
```

You should see output similar to following log after the switch is connected to the controller

```
loading app nat.py
loading app ryu.controller.dpset
loading app ryu.controller.ofp_handler
instantiating app ryu.controller.dpset of DPSet
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app nat.py of NAT
switch connected <ryu.controller.controller.Datapath object at 0x2185210>
```

b. On `outside`, we start a nc server:

```
outside:~$ nc -l 6666
```

and we start a nc client on `inside1` to connect it:

```
inside1:~$ nc 128.128.128.2 6666
```

c. You should now be able to type a message in either nc session and see it appear in the other session. Now, try the same thing between `outside` and `inside2`.

d. On `nat`, where you started the controller, you should see a log similar to:

```
Created mapping 192.168.0.3 31596 to 128.128.129.1 59997
```

Note that there should be only one log per connection, because the rest of the communication will re-use the mapping.

3 Handle ARP and ICMP

One common mistake experimenters make, when writing OpenFlow controllers, is forgetting to handle ARP and ICMP message causing their controllers to not work as expected.

Handling ARP is trivial in this example, as NAT does not involve ARP. However, that is not the case for ICMP. If you only process translations for TCP/UDP, you will find you cannot ping between `outside` and `insideX`, though nc is working properly. Handling ICMP is not as straightforward as for TCP/UDP because for ICMP you cannot get port information to bind with. Our provided solution makes use of the ICMP echo identifier. You could come up with different approach using ICMP sequence numbers, etc. To see ICMP working do the following.

a. On `inside1`, start pinging `outside`.

```
inside1:~$ ping 128.128.128.2
```

b. Do the same thing on `inside2`.

```
inside2:~$ ping 128.128.128.2
```

You should see both ping commands are working.

c. On `outside`, use `tcpdump` to check the packets it receives.

```
outside:~$ sudo tcpdump -i eth1 -n icmp
```

Notice that it is receiving two groups of icmp packets, differentiated by id.

Return to the main page