COSC 465: Computer Networking
Spring 2014
Project 7: SDN-based security middlebox
Due: 20 April 2014, 11:59:59pm
Note: You're welcome to work with someone else on this project

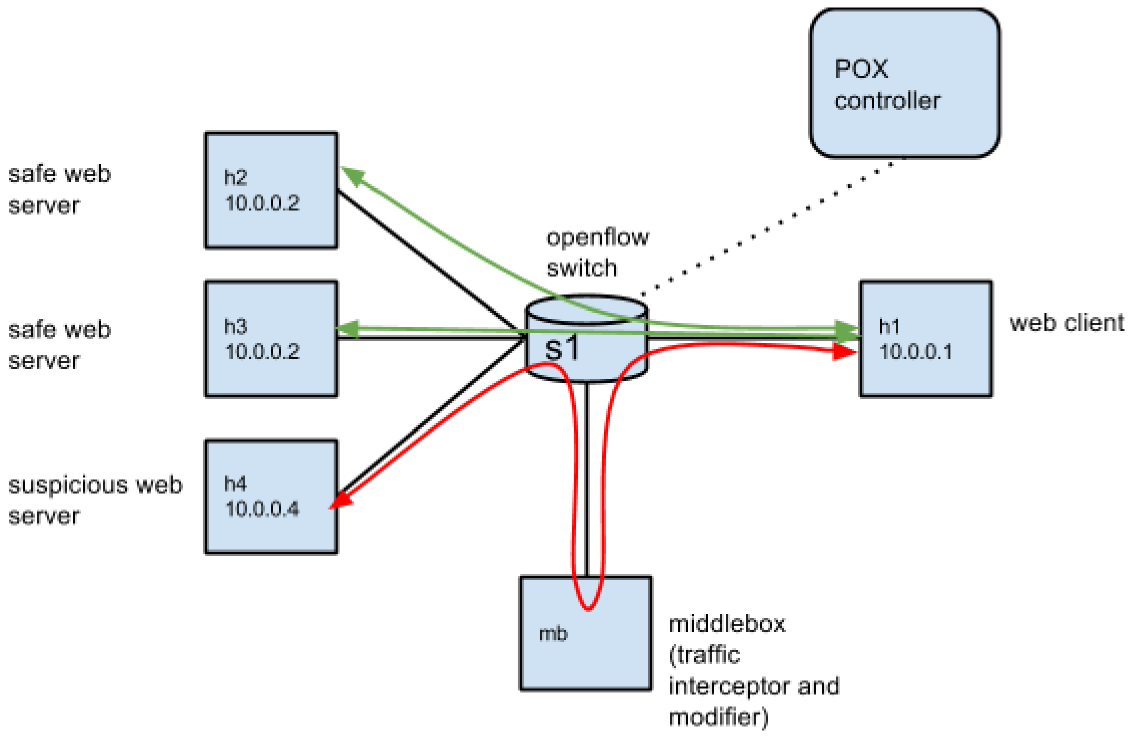
Overview

For this project, you're going to play the role of an unidentified government agency that wishes to monitor and intercept internet traffic to and from a "suspicious" web server. We will use Openflow capabilities to program a switch to transparently divert any of the suspicious traffic to a separate device, and let all other traffic flow normally.

The figure below shows the network setup. There is just one Openflow-enabled switch (s1), to which 4 hosts are connected (h1-h4). h1 will act as a web browser (client) and make requests to one of three servers (h2-h4). Any web traffic to h2 or h3 will be treated as "safe" and will be allowed to proceed normally (those "safe" flows are represented by the green curves in the figure below). Any traffic destined to or coming from the suspicious web server (h4) will be diverted to a special "middlebox" (labeled mb in the figure) before being sent back to the switch and on to its final destination. The traffic flow path between h1 and h4 is represented by the red curve in the figure below.

At the middlebox, we'll do some simple traffic processing based on doing "deep packet inspection" of payloads (i.e., web page content).

All the logic for routing/diverting traffic in the network will be handled with an Openflow controller module in POX.



Getting Started

To get started, clone the repo at https://github.com/jsommers/cosc465_project7. Once you've done that, run the script `setup.sh` inside the repo. (The setup script just clones the pox and srpy repositories.)

There are two pieces of code to write for this project: an Openflow controller module to handle traffic forwarding logic, and a SRPY-based program running on the mb host for manipulating traffic content. Each of these is described below, with a suggested order for writing the code and for testing. There are no automated tests for the Openflow controller module, but there are a few automated (SRPY) tests for the middlebox (mb) code. There are also some pretty simple Mininet experiments you can do to verify expected behavior.

Openflow controller module

First off, an Openflow controller module template is in the project repository in the file `p7.py`. A convenience script to get it running in POX is also in the repo: `startpox.sh`. Note: My `p7.py` solution code is 65 lines long and the template `p7.py` code is 40 lines long, so there isn't a lot of code you'll need to write for the controller module.

The only part of the POX module (`p7.py`) code you'll need to modify is within the `_handle_PacketIn` method. This method gets automatically invoked by POX whenever an Openflow `PacketIn` message arrives from a switch. There are basically three tasks you'll have to accomplish in this controller module:

1. For any non-TCP traffic that arrives, you should send it back to the switch with instructions to flood the packet out all ports except the one on which it arrived. You should just send the packet back to the switch and *not* install a new rule in the flow table. To send the packet back to the switch with a "flood" instruction, you should construct an Openflow PacketOut message (`ofp_packet_out`) back to the switch (POX documentation for ofp_packet_out) with a FLOOD action (POX documentation for output actions).

   See the POX documentation for `ofp_packet_out` and `ofp_action_output` for examples on how to construct those messages and send them back to the switch. Note that `ofp_packet_out` and `ofp_action_output` are both classes in the `pox.openflow.libopenflow_01` module, which is already imported in p7 as the symbol `of` (signifying OpenFlow).

   Once you've constructed the `ofp_packet_out` object, you can send it back to the switch using the following code:

   ```
       pktout = of.ofp_packet_out( … arguments to construct object )
       # your other code to attach flood output action to pktout

       # send pktout message back to switch
       event.connection.send(pktout.pack())
   ```

   Once you write this logic in the controller, you will be able to "ping" any host in the network (e.g., from h1 to h2, h3, or h4). To test this capability, you can do the following:
   - Open two separate terminals.
   - In one terminal, start up Mininet (`$ sudo python start_mininet.py`)
   - In the other terminal, start up your POX controller module (`$ ./startpox.sh`)
   - From the mininet prompt, you can then do something like:
     - `mininet> h1 ping -c3 10.0.0.2 # ping h2 from h1`
     - `mininet> h1 ping -c3 10.0.0.3 # ping h3 from h1`
     - `mininet> h1 ping -c3 10.0.0.4 # ping h4 from h1`

2. The next task to accomplish in the POX controller module is to install Openflow rules in the switch for all TCP flows *except* those with either source or destination IP addresses of 10.0.0.4 (i.e., the IP address of the suspicious server).

   To install a rule to let packets for these flows be forwarded directly to the intended destination, you can create a "flow modification" message and send it back to the switch with instructions to install a rule that matches packets like the one the controller just received, and to forward those packets out the correct output port (i.e., the output port connected to the destination host).

To construct the flow modification message to send back to the switch, you can use the following example:

```
actions = [ of.ofp_action_output(port = outport) ]
flowmod = of.ofp_flow_mod(command=of.OFPFC_ADD,
                idle_timeout=10,
                hard_timeout=10,
                buffer_id=event.ofp.buffer_id,
                match=of.ofp_match.from_packet(packet, inport),
                actions=actions)
event.connection.send(msg.pack())
```

The the ofp_flow_mod object we create indicates that we are *adding* a new rule to the Openflow table on the switch, with timeout values for the rule of 10 seconds. It also indicates that we want to forward a packet stored on the switch with a particular `buffer_id` (i.e., we want to forward the packet that originally generated the PacketIn event at the controller), the rule we install should match all the "standard" packet header fields in the same packet that triggered the PacketIn, and we want the rule action to be to forward the packet out port number `outport`.

You will need to assign something to the variable outport so that the correct port is used to reach a given destination host. Conveniently, the port number is the same number as the last digit of the IP destination address. (The way that Mininet is configured makes this easy.) So if the destination address is 10.0.0.3, the output port number should be 3.

Refer to the POX documentation for ofp_flow_mod and documentation for rule actions for examples and additional information on constructing these objects.

3. The last task to accomplish is to handle traffic that has either a source or destination IP address of 10.0.0.4. There are two cases to handle:
    - If the packet arrived on switch port number 5, the packet just came from the middlebox (port 5 on the switch is connected to the link leading to the mb host). In this case, you should forward the packet directly to the destination host. You should simply use the same logic as described in task #2, above (i.e., send an `ofp_flow_mod` message to install an Openflow rule with an action indicating the correct port number for forwarding the packet to its final destination).
    - If the packet arrived on any other switch port (i.e., not 5), you should install a rule (again, using an `ofp_flow_mod` message) to forward the packet to the mb host. The code for installing this rule is very similar to the other cases *except* that you will also need to rewrite the destination Ethernet address to be the Ethernet address of mb (which is 00:00:00:00:00:05). There is a special Openflow rule action that can be used to modify the Ethernet address of a packet; you will simply need to have *two* actions specified as part of the `ofp_flow_mod` message that you send back to the switch. The first action should be to set the new destination Ethernet address, and the second action should be to forward the

packet out a particular port. In particular:

i. The [POX documentation has an example](#) for how to specify an action for modifying the data link layer destination address (i.e., the Ethernet address). (The action is named `ofp_action_dl_addr`, as you'll see in the documentation.)
ii. The output port should just be 5.

Middlebox traffic manipulator

There are three tasks to accomplish at the middlebox device. The only packet traffic this device is concerned with is TCP traffic to or from 10.0.0.4. There's already a conditional statement in the middlebox starter code (in `middlebox.py`) to that effect (it just tests whether the traffic is TCP, which is fine). Note: my code for the middlebox is about 110 lines and the template file already has 47 lines.

With the TCP traffic that arrives at the middlebox, you should:

1. Collect **all** application content (tcp payload data) and log it to a file named `contentlog.txt`. You can clear this file whenever the middlebox starts up (i.e., use 'w' as the second parameter to the open function call rather than 'a'). You should only save the application data to a file when either a TCP flow ends (i.e., the FIN or RST bits are set in the TCP packet header), or when the middlebox device shuts down. Note that you'll basically be *accumulating* TCP payload content for all packets of a given connection, which may span multiple packets.

   I'd recommend just keeping a Python dictionary to assemble the application content, where the key could just be a string of the form "srcip:srcport:dstip:dstport" (you can easily glean these values from packet headers) and the value is just a string representing the currently seen content.

   When you write content to the file, write it out in the following format:
   ```
   timestamp  srcip:srcport:dstip:dstport  applicationcontent
   ```

   - The timestamp can just be the output of the Python `time.asctime()` call. (Don't forget to import time.)
   - The second item can be easily grabbed from packet headers, and could also simply be the dictionary key you use for finding and adding on to application content.
   - The application content is what you've been accumulating as you observe packets fly by. Note that the app content is likely to span multiple lines. Even though that might make the file contents a little difficult to read, that's ok.

2. Perform some traffic content manipulation:

    - Replace any occurrences of the string "NSA" in the application content with the string "Fluffy Bunnies". Your search for the string NSA can be case sensitive (i.e., it's ok to match NSA exactly in upper case). The content you log to the `contentlog.txt` file should be the *original* content (i.e., with NSA) rather than the modified content.
    - If the string "panda" (case sensitive check is ok) occurs in the application content, you should artificially inject a TCP RST packet to immediately stop the connection in progress. ("panda" is a well-known code word among nefarious types.) To accomplish this, you can do the following:
        i. Set the RST TCP flag in the header of the current packet you're analyzing (`tcphdr.RST = 1`).
        ii. And remove any of the application payload content (`tcphdr.set_payload('')`). That's it.
    - Note that when you change the length of the TCP payload, the tcplen attribute in the TCP header must also be changed (it should be set to 20 + the length of the TCP payload). You don't need to update the iplen attribute in the IP header.

3. To forward traffic back to the switch, you'll need to rewrite the destination Ethernet address in the packet header to be the Ethernet address of the destination host. (It was rewritten by s1 to be the Ethernet address of mb, but now it needs to be changed back to the Ethernet address of the dest host.) You can pretty easily compute the destination Ethernet address by looking at the last digit of the destination IP address:
    - If the destination IP address is of the form 10.0.0.x, the destination Ethernet address should be 00:00:00:00:00:0x.

   You should just send any packets back out the same interface on which they arrived. There are no real forwarding or routing decisions you need to make.

To run the automated tests on the middlebox code, you can run the following script:

```
$ ./testmb.sh
```

There are automated tests to check whether the content manipulation is done correctly (i.e., replacing NSA with Fluffy Bunnies), whether TCP RST packets are correctly injected, and whether Ethernet destination addresses are correctly rewritten. There are no automated tests for the logging capability.

To start the middlebox code in Mininet, you can start up Mininet, open an xterm on mb, and run the startmb.sh script included in the repo (or just run the middlebox.py code on the command line). For example:

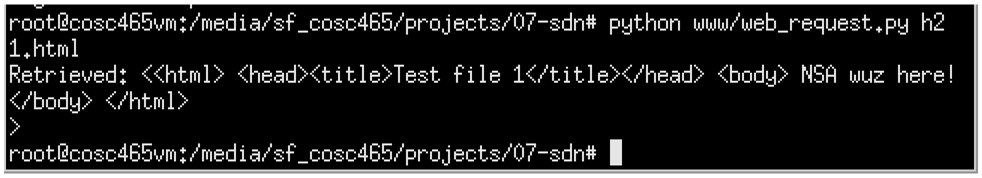
```
$ sudo python start_mininet.py
mininet> xterm mb
mb# ./startmb.sh
-- or --
mb# python ./srpy/srpy.py middlebox
```

Running and testing your code

There are some scripts you can use for making web requests from the host h1 to any other host in the network. To use these scripts, you can start up Mininet (`sudo python start_mininet.py`) then do the following:

- In another terminal (not an xterm within Mininet), start POX:
  - `$ ./startpox.sh`
- At the mininet prompt, open a terminal on mb and start the middlebox code:
  - `mininet> xterm mb`
  - `mb# ./startmb.sh`
- At the mininet prompt, open a terminal on h1
  - `mininet> xterm h1`
- In the xterm, you can run a script to make a web request to one of the servers. The script is in the subdirectory www and is named web_request.py. It takes two arguments: the host name (h2, h3, or h4) and the name of a file to request. There are only two valid files in the www folder that I created, though you can easily create more for testing if you want. The two files are 1.html and 2.html. The 1.html file has the string NSA in it, and 2.html has both the strings NSA and panda in it.
- If you make a request from h1 to either h2 or h3, the web requests should proceed normally and nothing should go through the middlebox. Your controller should install the correct rules to forward packets correctly, and in the xterm where you make the web request, you should see the file contents returned from the web server without NSA modified (it should still be NSA and not Fluffy Bunnies), and the request with panda should also proceed normally. Here's how to use the web requester script:
  - `Example: request 1.html from server h2`
    - `h1# python www/web_request.py h2 1.html`

    The output from this script should be similar to the following (it should just show the contents of the 1.html file).

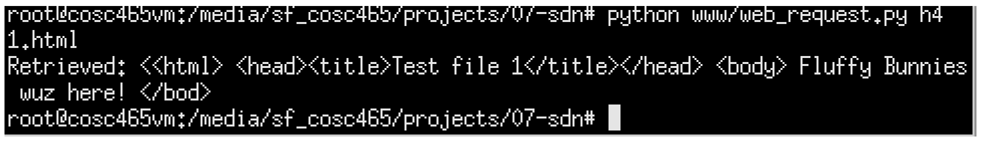


```
root@cosc465vm:/media/sf_cosc465/projects/07-sdn# python www/web_request.py h2
1.html
Retrieved: <<html> <head><title>Test file 1</title></head> <body> NSA wuz here!
</body> </html>
>
root@cosc465vm:/media/sf_cosc465/projects/07-sdn#
```

  - `Example: request 2.html from server h2`
    - `h1# python www/web_request.py h2 2.html`

    The output from this script should be similar to the above example (again, it should just be the contents of 2.html).

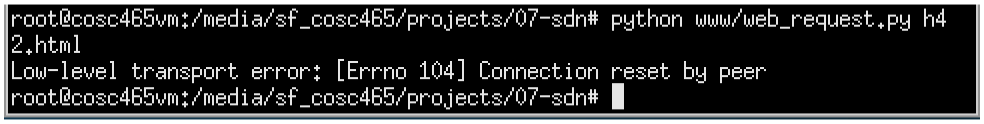
- If you make a request from h1 to h4 for 1.html, you should observe two things:
  - The file contents shown from the web_request.py script should not show the string NSA. Instead the one occurrence of NSA should be replaced with Fluffy Bunnies, as shown below.



```
root@cosc465vm:/media/sf_cosc465/projects/07-sdn# python www/web_request.py h4
1.html
Retrieved: <<html> <head><title>Test file 1</title></head> <body> Fluffy Bunnies
 wuz here! </bod>
root@cosc465vm:/media/sf_cosc465/projects/07-sdn#
```

    - Your contentlog.txt file created by the middlebox.py program should include a new log line with the text of the *original* file (i.e., with NSA, not Fluffy Bunnies). Note that if the middlebox.py program is running and you don't see the output in the file, you may need to call the `flush` method on the file handle in order to force any buffered file data out to disk.

- If you make a request from h1 to h4 for 2.html, you should not see any content at all at h1. Instead, you should see something similar to the following:



```
root@cosc465vm:/media/sf_cosc465/projects/07-sdn# python www/web_request.py h4
2.html
Low-level transport error: [Errno 104] Connection reset by peer
root@cosc465vm:/media/sf_cosc465/projects/07-sdn# 
```

  Note that "connection reset by peer" indicates that the connection received a TCP RST message that caused the connection to be aborted. That bit of nastiness was, of course, inflicted by our own middlebox.

  When you send the RST, you will also likely observe some retransmissions sent by the server (h4), since your RST will cause the client (h1) never to send an ACK back to it. If you'd like, you can send an *additional* TCP RST packet to the server to tell it to abort the connection, but you don't have to.

Once you're done, you can submit your p7.py and middlebox.py programs to Moodle.