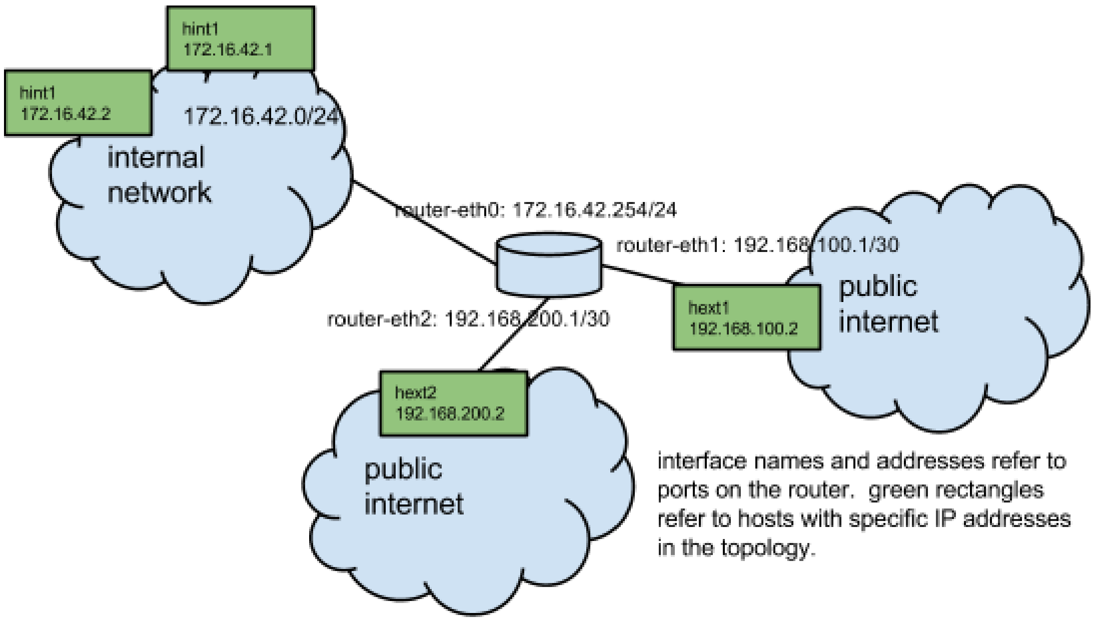
COSC 465: Computer Networking
Spring 2014
Project 6: IP router 4: firewall features
Due: 3 April 2014, 11:59:59pm
Note: You're welcome to work with someone else on this project

Overview

The goal for this project is to add basic firewalling capabilities to the IP router you've been building over the past 3 projects. The firewall rules we will use will be simple, *static* rules (i.e., no "stateful" or dynamic rules), with one twist: we'll add the option to *rate-limit* certain packet flows using the token bucket algorithm.

For this project, there are no automated tests. You'll need to construct your own unit tests to test the firewall functionality, and/r run the router in Mininet and direct specific types of traffic to it. Some suggestions for how to go about testing are listed at the end of this project description.

Below is a picture of the example network topology that is created by the Mininet script, and which is assumed in examples below.


hint1
172.16.42.1
hint1
172.16.42.2
172.16.42.0/24
internal network
router-eth0: 172.16.42.254/24
router-eth1: 192.168.100.1/30
router-eth2: 192.168.200.1/30
hext1
192.168.100.2
public internet
hext2
192.168.200.2
public internet
interface names and addresses refer to ports on the router. green rectangles refer to hosts with specific IP addresses in the topology.

Tasks to accomplish

There are two things to accomplish for this lab:

(1) Implement firewall rules. We will use a fairly basic syntax, with no "stateful" rules. All rules will be loaded from a text file named `firewall_rules.txt`, so now your router will expect *two* text files to exist on startup: `forwarding_table.txt` and `firewall_rules.txt`. The syntax and semantics of the firewall rules is described in detail, below. As a preview and example, however, here is a rule that *permits* IP traffic (with any transport protocol) with source address 10.0.0.1 and any destination address:

```
# allow IP packets with src 10.0.0.1 destined to anywhere
permit ip src 10.0.0.1 dst any
```

(Note that the line beginning with # is just a comment.)

The firewall rules should be applied to packets *just after they have been received at the router*. You probably already have code to test what type of packet has arrived, in order to properly handle it. The firewall should apply just to IPv4 packets (not to ARP, IPv6, or any other type of packet). As described in detail below, rules can either *permit* or *deny* packets. For packets that are *permitted*, they should be processed as normal by the router. For any *denied* packets, they should be dropped/ignored and no further processing should be done on them.

You should start with the router code that you have so far (i.e., your router from project 5). The key features of your router that you'll need are IP forwarding and ARP request/response handling. (That is, no ICMP error generation features will be tested when I evaluate your code.)

(2) Implement the token bucket algorithm for any rules that specify rate limits. Rules that *permit* traffic may optionally have rate limits specified, in bytes per second. Here is the same example rule from above, but with a rate limit specified as 1000 bytes per second:

```
# allow, but rate-limit packets with src 10.0.0.1
# destined to anywhere, with limit as 1000 bytes/sec
permit ip src 10.0.0.1 dst any ratelimit 1000
```

When implementing the token bucket algorithm, you should do the following:

- Tokens should be semantically equated to some number of bytes allowed. Thus, if there are 100 tokens in the bucket, you should allow 100 bytes to be forwarded.
- Tokens should be added to the bucket every 0.5 seconds. If the rate limit configured is *r* bytes per second, exactly *r/2* tokens should be added to the bucket every 0.5 seconds. For the example above, you would add 500 tokens to the bucket every 0.5 seconds.
    - Note: from your experience in operating systems, you should know (or at least suspect) that OS timers are fickle. Just beware and devise your code so that it doesn't assume a perfect timer, because a perfect timer doesn't exist. Remember that in the net.recv_packet() call, there's a timeout value you can give in order not to block on the call. In order to ensure you update the token bucket every 0.5 seconds, just manipulate the timeout value.
- The maximum number of tokens in the bucket is *2*r*, where *r* is the configured rate limit in bytes/second.
- The minimum number of tokens in the bucket should be 0 (zero).

- When a packet arrives that matches a rule with a rate limit configured, you should (a) determine the size of the packet in bytes, including everything from the IP header onward (i.e., include everything *except* the Ethernet header), (b) check whether the packet size is less than or equal to the number of tokens. If it is, subtract that number of tokens and allow the packet. Otherwise, the packet should be dropped.

Note that with this algorithm, if the rate limit is set too low, you may never allow *any* packets at all (e.g., if you set the limit to 15 bytes/sec, you'd *never* allow any TCP traffic since the smallest TCP packet is 40 bytes, which is greater than 2*15). Also note that the rate limit you achieve is not expected to be *exactly* the specified rate limit. Refer to the textbook and class notes if you have no idea what the token bucket algorithm is.

Getting started

To get started, clone the git repo at https://github.com/jsommers/cosc465_project6. Run ./setup.sh in the repo to clone the POX and SRPY code, then copy in your existing router code. Note that the setup.sh file does *not* create the runtests.sh or runreal.sh scripts. To run your router in Mininet, you can simply open an xterm on the router node, and use the following command (assuming you named the router file as `myrouter4.py`):

```
# python srpy/srpy.py myrouter4.py
```

You can supply any command-line arguments to SRPY if you want to turn on debugging (-d) or verbose output (-v), etc. For example, to turn on debugging output and verbose output, you can type:

```
# python srpy/srpy.py -v -d myrouter4.py
```

Firewall rule specification and syntax

The firewall rules to be loaded by the router must be included in a text file named `firewall_rules.txt`. The allowable syntax for rules is as follows:

```
[permit|deny] ip src [srcnet|any] dst [dstnet|any]
[permit|deny| icmp src [srcnet|any] dst [dstnet|any]
[permit|deny] [udp|tcp] src [srcnet|any] srcport [portno|any] dst [dstnet|any] dstport [portno|any]
```

Note that items in square braces indicate items to be made concrete in a specific rule. For example, a valid rule is:

```
permit ip src 10.0.0.1 dst any
```

which would allow any IP packets (any protocol) with source address 10.0.0.1 and any destination address.

Note that the srcnet or dstnet values may either be an exact IP address, or an IP prefix indicating a subnet of addresses. Also, portno should be specified as a single integer between 0 and 65535. "any", somewhat obviously, should match anything.

Here is another example:

```
	deny tcp src 1.2.3.0/24 srcport any dst any dstport 80
```

This rule blocks any TCP traffic with source address in the range 1.2.3.0-255, with any source TCP port, any destination IP address, and a destination port of 80.

It is straightforward to access TCP and UDP port numbers using the POX packet library. See the POX documentation for details and examples.

You may also find the `parse_cidr` function in the POX address library useful. It accepts a string IP address or network prefix as a parameter, and returns a tuple containing an IPAddr object with just the network address, and the prefix length as an int. For example, parse_cidr("149.43.0.0/16") returns (IPAddr("149.43.0.0"),16). If you just pass a single IP address (no cidr prefix), the prefix length returned will be 32 (as expected). (This function is in the pox.lib.addresses module.)

Blank lines are allowed in the firewall_rules.txt file, and any line on which the first non-whitespace character is # should be ignored. Thus, you should allow Python-like comments, but you do not need to handle the situation where a comment and a rule appear on the *same* line --- comments will always appear separately.

Rule order matters! Packets should be checked against the rules in the same order as the rules are listed in `firewall_rules.txt`. When a matching rule is found, the action (permit/deny) according to the rule should apply, and no more rules should be checked. If no rules match, then the default action should be to *permit* the packet. Note that in the example rule set below, the last rule explicitly drops all packets but when I test your code I may (read: I will) use a different rule set.

Rate limits can be applied to any "permit" rule. To specify a rate limit, the syntax is "ratelimit [bytessec]", included at the *end* of a rule. The rate limit accounting should apply to the *entire* packet *except* the Ethernet header (i.e., the packet size used for rate limit accounting should just include the IP header and beyond).

Here is an example set of firewall rules (the same ruleset included in `firewall_rules.txt` in the git repo):

```
	# drop everything from a network that we don't trust
	deny ip src 192.168.42.0/24 dst any

	# allow traffic to/from an internal web server that should
	# be accessible to external hosts
```

```
permit tcp src 172.16.42.42 srcport 80 dst any dstport any
permit tcp src any srcport any dst 172.16.42.42 dstport 80

# allow DNS (udp port 53) traffic in/out of network
permit udp src 172.16.42.0/24 srcport any dst any dstport 53
permit udp src any srcport 53 dst 172.16.42.0/24 dstport any

# allow internal hosts access to web (tcp ports 80 and 443)
# rate limit http traffic to 100 kB/s (12500 bytes/sec)
permit tcp src 172.16.42.0/24 srcport any dst any dstport 80 ratelimit 12500
permit tcp src any srcport any dst 172.16.42.0/24 dstport 80 ratelimit 12500
permit tcp src 172.16.42.0/24 srcport any dst any dstport 443
permit tcp src any srcport any dst 172.16.42.0/24 dstport 443

# permit, but rate limit icmp to 100 bytes/sec
permit icmp src any dst any ratelimit 100

# block everything else
deny ip src any dst any
```

Grading and Testing

Grading guidelines

- The firewall rule implementation (not including rate limits) is worth 70% of the grade for this project. A simplistic implementation of the firewall rules will likely result in a mess of nested if statements (or the equivalent). **For the full 70% you will need to think of a better implementation**. By *better implementation*, I mean one that is very efficient in the sense that it does as little work as possible when testing whether a packet should be permitted or denied.
  - A hint: think of how you can "compile" the rules ahead of time (i.e., when you load the rules and initialize the firewall), where the compilation process might be fairly complex, but it facilitates fast checking of packets. For example, you might think of how you can employ bitwise operations to make things fast.
  - When testing whether a packet should be permitted or denied, my own implementation has a total of 2 if statements: one to handle the special case that we don't check port numbers for generic IP packets or ICMP packets and one to check the token bucket filter. If you can do better than 2 if statements (i.e., one or zero if statements in rule checking), you'll get 5 points bonus. Note that I haven't used any obscure Python features or special libraries to get to 2 if statements.
- The rate limitation capabilities are worth 30% of the overall grade.
- Code "cleanliness" is worth up to 10%. Keep your methods SOFA (they should be short, do one thing, take few args, and implement a single level of abstraction).

Suggestions for testing

For testing, I'd recommend creating a separate file (Python "module") like `firewall.py` for implementing firewall features and making a Firewall class (or something similar) that exposes a method for checking whether a packet "passes" the firewall, and a method that should get periodically called for updating token bucket state(s). From within your router code, you can simply say "`import firewall`" or "`from firewall import Firewall`" to use the firewall code in your router. If you implement and integrate the firewall this way, there should be no more than a small handful of lines changed in the main router code.

More importantly, if you put the firewall code in a separate file, you can very easily create unit tests for verifying whether firewall methods work as expected. You can do it either just in a test function that only gets invoked when firewall.py is run *directly* (i.e., not when it is imported), or in a separate testing file that imports firewall and uses the very nice capabilities in the Python unittest module (http://docs.python.org/2/library/unittest.html).

An example firewall.py file is shown below for you to get started with. It has a "tests" function that only gets invoked if the file is run directly (i.e., python firewall.py). There's also an example of constructing a UDP packet (that should be permitted, according to the rule set above) for testing. For additional examples of how to create TCP and UDP headers (as well as any other header) for testing, see the POX documentation.

```
import sys
import os
import os.path
sys.path.append(os.path.join(os.environ['HOME'],'pox'))
sys.path.append(os.path.join(os.getcwd(),'pox'))
import pox.lib.packet as pktlib
from pox.lib.packet import ethernet,ETHER_BROADCAST,IP_ANY
from pox.lib.packet import arp,ipv4,icmp,unreach,udp,tcp
from pox.lib.addresses import
EthAddr,IPAddr,netmask_to_cidr,cidr_to_netmask,parse_cidr
import time

class Firewall(object):
    def __init__(self):
        # load the firewall_rules.txt file, initialize some data
        # structure(s) that hold the rule representations
        pass


def tests():
    f = Firewall()

    ip = ipv4()
    ip.srcip = IPAddr("172.16.42.1")
    ip.dstip = IPAddr("10.0.0.2")
    ip.protocol = 17
    xudp = udp()
    xudp.srcport = 53
    xudp.dstport = 53
    xudp.payload = "Hello, world"
    xudp.len = 8 + len(xudp.payload)
    ip.payload = xudp

    print len(ip) # print the length of the packet, just for fun

    # you can name this method what ever you like, but you'll
    # need some method that gets periodically invoked for updating
    # token bucket state for any rules with rate limits
    f.update_token_buckets()

    # again, you can name your "checker" as you want, but the
    # idea here is that we call some method on the firewall to
    # test whether a given packet should be permitted or denied.
    assert(f.allow(ip) == True)

    # if you want to simulate a time delay and updating token buckets,
    # you can just call time.sleep and then update the buckets.
    time.sleep(0.5)
    f.update_token_buckets()

if __name__ == '__main__':
    # only call tests() if this file gets invoked directly,
    # not if it is imported.
    tests()
```