This document lives here:
http://inst.eecs.berkeley.edu/~ee122/fa12/project3/project-spec.pdf

Project 3 - Routers, Protocols and Firewalls

UC Berkeley, EE 122, Fall 2011

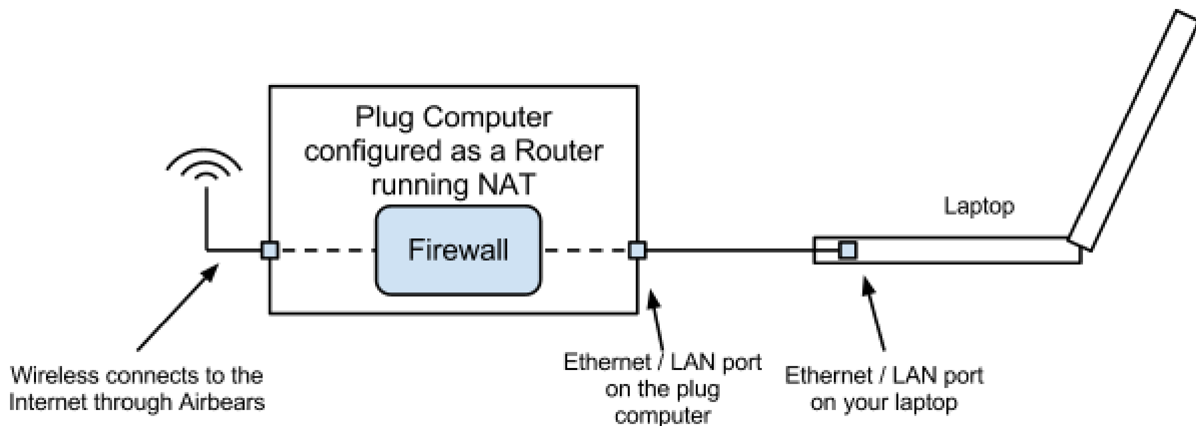
Version 1

Part 1 due on November 16, 2012, 11:59:59 PM

The goal of this project is for you to learn about routers, protocols and firewalls; in short, how real networks work. You will be given a plug computer configured to run as a router which can also function as a firewall. Your task is to write the rules for the firewall to allow/disallow a specified set of TCP connections.

This month-long project is divided into 2 parts, each part has it own submission deadline. In the first part, you will have to programmatically configure a firewall to manage connections based on simple set of rules. The goal of the first part is to become familiar with the environment of the router and the firewall's API. In the second part, you will be extending the set of rules to manage more complex patterns of connections. Here, the goal is to understand the structure of protocols like FTP and HTTP, and build rules to manage them effectively.

This document first explains the router environment - how each plug computer is configured as a NAT router. Then it will elaborate on the firewall functionality present in the plug and how it can be used to allow / deny / monitor connections passing through the plug computer. Later it will specify the set of rules that you are supposed to write for each part of the project.



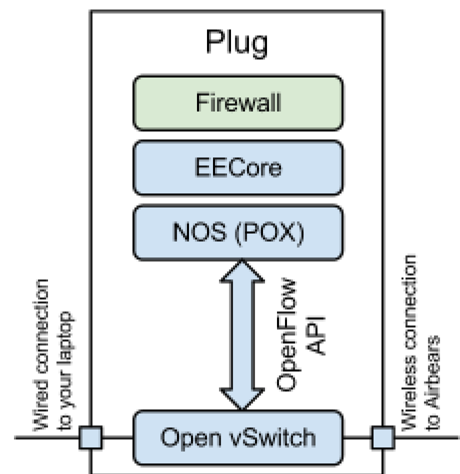
Configuration

Each plug computer has two network interfaces - a wired Ethernet interface, and a wireless interface. As shown in the figure, the wireless interface is configured to automatically connect to *AirBears*, and the wired Ethernet interface is to be connected to the Ethernet port of your laptop, using the provided Ethernet cable. The plug computers are configured to act as NAT routers - the wireless interface is the external port which connects to the Internet through *AirBears*, and your laptop is connected to the internal port, effectively becoming a client behind the NAT. More details on preparing and connecting to the plug computer are provided in the **Guide to your Plug Computer** (http://inst.eecs.berkeley.edu/~ee122/fa12/project3/guide-to-plug.pdf). Please read this document in detail before attempting to use the plug.

This system is designed to let you do all your Internet activities through it (maybe a little slowly), except those connections that are blocked by the plug's internal firewall. The goal of this project is for you to implement rules for the firewall based on the provided specifications. *In the rest of this document, we may refer to the plug computer as a router*.

Software Architecture

To give you some context, the software architecture in the plug is as follows (see adjoining figure). The wired and wireless ports are internally connected together by a virtual switch (Open vSwitch), which has the capability to forward or drop packets as instructed. The Network Operating System (NOS) gives these instructions (rules) to the virtual switch using the OpenFlow protocol. The EECore library uses the firewall code that you will write to instruct NOS to allow/block/monitor certain connections.



From the project's point of view, you only need to understand the firewall component. For the more daring minds who want to know more about this, take a look at this brief overview about *POX (http://www.noxrepo.org/pox/about-pox/)*.

Firewall - Model and API

All TCP connections through the router have to pass through a firewall. The firewall has the following functionality

1. It can allow a TCP connection through the router. In other words, the allowed connection can continue uninterrupted.
2. It can deny a TCP connection, that is, it will not forward any packets that belong to the TCP connection (uniquely identified by the 4-tuple <*source ip, source port, destination ip, destination port*>).
3. It can monitor all the data in a TCP connection, and if required, allow or deny other connections based on that data.

These functionalities are implemented using an event driven system - as packets arrive at the router, events are raised according to various conditions. The event-handler function of each event decides the fate of the connection whose packet raised the event. This will explained shortly.

There are 3 types of events in our firewall system.

1. `ConnectionIn`: This event is raised when an attempt to set up a new TCP connection is detected by the router, that is, when the first SYN packet of the TCP connection arrives at the router. Based on the characteristics of the connection (destination IP, destination port, etc), the system can take any of the 4 actions:
    a. Deny the connection attempt.
    b. Allow the connection to continue uninterrupted.
    c. Defer the allow/deny decision for later. This temporarily allows the connection attempt to continue, but raises the `DeferredConnectionIn` (see next) event when data transfer starts.
    d. Mark the connection for monitoring of all data transferred through it (that is, only if the connection is allowed to continue).
2. `DeferredConnectionIn`: If a TCP connection has been marked for deferred decision, then this event is raised when the first packet carrying application-level data arrives at the router. Note that for any connection, this event can occur only after the `ConnectionIn` event.
3. `MonitorData`: If a TCP connection has been marked for monitoring, then every packet that belongs to the connection will raise this event when it arrives at the router. The data in the packet can be inspected, and accordingly the connection may be allowed to continue or denied. Again, for a connection, this event can occur only after the `ConnectionIn` event.

Implementing rules for the firewall requires appropriately handling these events. The `Firewall` class defined in **/root/pox/ext/firewall.py** specifies the 3 event handlers. The functionality of each function is explained as follows.

```
class Firewall (object):
      This class defines the event handler functions corresponding to
      the three possible events in the system.

      def _handle_ConnectionIn (self, event, flow, packet):
```

This function is called by the system on a *ConnectionIn* event, that is, when the first packet of a new TCP connection is received by the router.

packet: This is an object representing the packet whose arrival raised this *ConnectionIn* event, that is the first SYN packet of a new TCP connection. Details of this object are explained later.

flow: This is an object representing the TCP connection. It has the following fields.

flow.src : Source IP address of the TCP connection
flow.srcport : Source port of the TCP connection
flow.dst : Destination IP addr. of the TCP connection
flow.dstport : Destination port of the TCP connection

event: This is an object representing the new connection event. Details of this object are unnecessary. You only need to remember the set of actions that can performed as a response to the event. If no action is specified, then the connection is by default not be allowed. These actions are as follows

event.action.forward = True

Allows the connection to proceed.

event.action.forward = False

Drops the current packet and disallows the connection by dropping every packet that belong to this connection.

event.action.deny = True

Drops the current packet and disallows the connection by sending a TCP RST to the source of the connection. This is the preferred way of denying a connection over event.action.forward = False (can you think why this is preferred?)

event.action.defer = True

Allows the connection to start, and sets the system to raise the *DeferredConnectionIn* event when the first data packet in this connection arrives at the router.

event.action.monitor_forward = True

Allows monitoring of outgoing (from your system to Internet) data on the connection. In other words, every packet of data sent out over the TCP connection raises the *MonitorData* event.

event.action.monitor_backward = True

Similar to monitor_forward, but allows monitoring of incoming data (from Internet to your system) on the connection.

def _handle_DeferredConnectionIn (self, event, flow, packet):

This function is called by the system on a *DeferredConnectionIn* event. As explained before, this only happens on the connections that have been marked for deferring (using event.action.defer = True) in the corresponding *ConnectionIn* event.

packet: Similar to that of *ConnectionIn*, this represents the first data packet in a TCP connection that raised this *DeferredConnectionIn* event. Details later.

flow: Same as that of *ConnectionIn*.

event: Same as that of *ConenctionIn*, except that **event.action.defer** is unavailable. In other words, you cannot defer the connection any more, and you have to either allow it, or disallow it, or monitor it.

def _handle_MonitorData (self, event, packet, reverse):

This function is called by the system on MonitorData event, that is, when a packet is received that belongs to the connections that have been marked for monitoring. If a connection has been marked for only outgoing events, then only outgoing packets will raise this event. And vice versa.

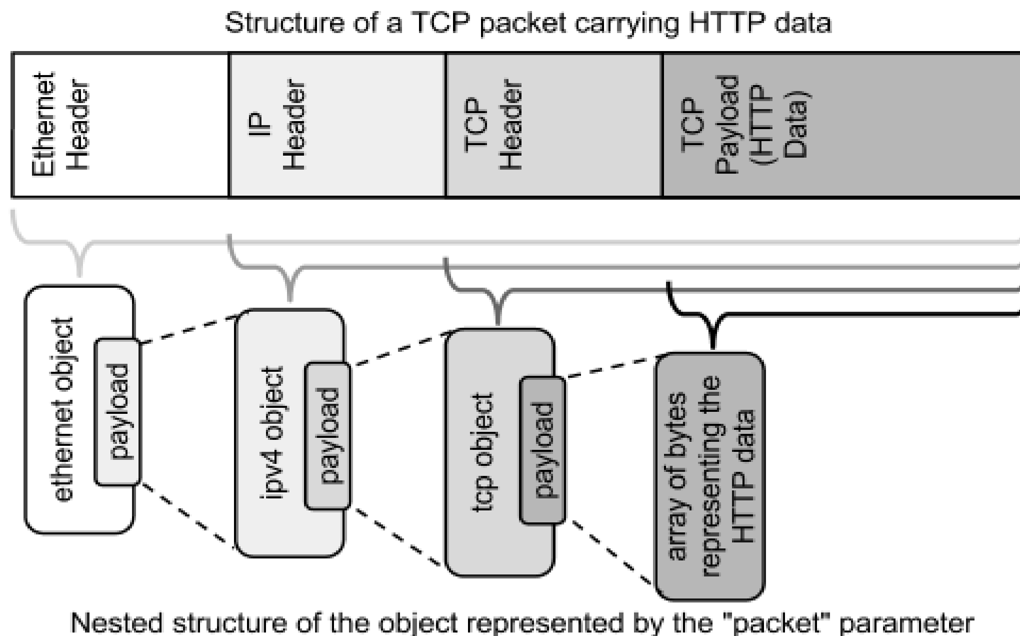
reverse: *False* or *True*, based on whether the event was raised on outgoing (forward) packet or incoming (backward) packet, respectively.

packet: Similar to that of *ConnectionIn*, this represents the packet in a TCP connection that raised this *MonitorData* event. Details later.

event: This is just a placeholder. This has no actions. Ignore this for now.

The packet parameter in the above functions is an object of type `ethernet`. As the name suggests, this represents an Ethernet packet. The `ethernet` class has a property called `payload` that returns are an object of type `ipv4`. This representing the IP data (including the the header) within the Ethernet packet. Similarly, the `payload` property of `ipv4` class returns an object of type `tcp` (if packet belongs to a TCP connection), which represents the TCP data (including the header). And `payload` of `tcp` gives the raw payload data.

To illustrate the concept, take a look at the figure below. This illustrates the object structure of a TCP packet carrying HTTP data. The object representing each layer encapsulates the data belonging to the higher layer. So for a HTTP packet, `packet.payload.payload.payload` gives the raw HTTP data.



All these classes (`ethernet`, `ipv4`, `tcp`) are subclasses of parent class `packet_base`. To know more details about properties and functionalities of these classes, take a look at the directory **/root/pox/lib/packet/**.

The default *firewall.py* provided to you allows all incoming TCP connection (note the `event.action.Forward = True` in the function `_handle_ConnectionIn`). You are supposed to use the API explained earlier to appropriately allow / deny / defer / monitor connections based on the characteristics of the connection (destination port, etc) and the transferred data.

Note that the system is hard-coded to use the `Firewall` class from the file *firewall.py*. So implement your rules in that file. Creating a different file and/or making a subclass of `Firewall` will not work. You have to modify the Firewall class in the same file to implement your own rules.

Firewall Rules - what you have to implement

Part 1 - Due on November 16, 2012, 11:59:59 PM

For the first part, implement a firewall that does the following.

1. Each line in the file **/root/pox/ext/banned-ports.txt** has a port number. Deny all TCP connections which has one of these as the destination (external) ports. As you work through this, it might help to realize that in this case the first packet from a connection is the SYN from your computer to the server on the internet.

2. Each line in the file **/root/pox/ext/banned-domains.txt** has a domain name (like *google.com*). Deny all HTTP connections to these domains as well as their subdomains[1]. For example, for *google.com,* you will have to deny connections *www.google.com* as well as subdomains like *images.google.com*. However, do not deny connections to the domain *mygoogle.com*. [Hint: There are multiple ways to do it, but the right way to do it is to identify the connection by analyzing its HTTP header information.]

3. Each line in the file **/root/pox/ext/monitored-strings.txt** is an *IP address* and a *search string* (separated by a colon). That is,

   <IP address>:<search string>

   For each connection made to any of the IP addresses present in the file (that is, external IP address of a connection matches any in the file), count the number of times the corresponding search string appears in the incoming or outgoing data on the connection. After the connection is over, write (append) the count to the file **/root/pox/ext/counts.txt** in the following CSV format

   <external IP address>,<external port>,<search string>,<count>

   You may treat a connection to be over if it has not transferred any data for more than **30 seconds.** The order of writing counts to the file does not matter. **You must however flush after each write, i.e. make sure that your writes actually end up on disk, not in a buffer somewhere.**

   The search strings can be of arbitrary length (can be larger a single packet size) and the matched substring in the connection data may cross the boundary of a packet. You have to keep data in memory appropriately to identify such matches. However, your implementation should be efficient in using memory. For example, you should not keep the complete connection data in memory. Also, you are not allowed to use files to store your data (you may use them for debugging purpose but not for the final submission).

   Furthermore, there can be multiple search strings associated with the same IP address. For every connection to the IP, you are supposed to count and output all the search strings associated with that IP.

   Here is an short example to explain all of this. Lets say, there are the following two search strings.

   1.2.3.4:abc
   1.2.3.4:abcabc

   Let say, a connection from your laptop to 1.2.3.4:5678 transfers the following data.

   Outgoing data: ***abc****def****abcabc***

[1] http://en.wikipedia.org/wiki/Subdomain

Incoming data: *def**abcabc**qwe**abc***

Your firewall should append the following lines to the file /root/pox/ext/counts.txt .

1.2.3.4,5678,abc,6

1.2.3.4,5678,abcabc,2

Both, incoming and outgoing data have 3 instances of “abc” and 1 instance of “abcabc” each.

Your rules must be such that only the connections that fit the above criteria should be blocked and monitored, and everything else must be allowed.

If you want to use timer, you should use the POX timer, rather than Python’s Threading.Timer class. Refer to this link for more information on how POX timers work.

In the plug, you will find 3 sample files, banned-ports.txt, banned-domains.txt, and monitored-strings.txt, in the directory /root/pox/ext/ . These files contain sample data that have been provided to you to understand the input format. We are going to test your firewall with our own set of 3 files, and your firewall should work against arbitrary ports / domains / search strings that may be present in those files. You can assume that these 3 input files will always be located in the said locations, that is, in /root/pox/ext/ .

Tips and Tricks

Part 1

Some advise about how to start off the project.

- Start by reading the guide to the plug computer in detail. Prepare the USB stick with the given image, and verify whether your plug works. Familiarize yourself with the plug’s software environment. This will require you to work in a console environment a lot, so better get used to it if you are not.
- Understand the given APIs and test each event one by one. For example, try configuring the `ConnectionIn` handler to block connections to port 80, and see whether you are able to browse the Internet through the router or not.
- Read up about the HTTP protocol, and understand what information they carry. Use that information to filter out which connection to allow / deny / defer / monitor.

Submission Details

You are required to turn in the following files -

1. *firewall.py*
2. *README.txt*

The *README.txt* must contain the following information.

1. Your and your partner’s (if any) names

2. What problems or challenges did you face in implementing your firewall?

Cheating and Other Rules

You should not touch any other code other than firewall.py. We are aware that Python is self-modifying and therefore you could write code that rewrites the other files in the system. *You will receive zero credit for turning in a solution that modifies any other file in the filesystem.*

The project is designed to be solved independently, but we STRONGLY encourage to work in pairs. Grading will remain the same whether you choose to work alone or with a partner; both partners will receive the same grade *regardless of the distribution of work between the two partners* (so choose a partner wisely!).

You may not share code with any classmates other than your partner. You **may** discuss the assignment requirements or your solutions -- *away from a computer and without sharing code* -- but you should not discuss the detailed nature of your solution. Assignments suspected of cheating or forgery will be handled according to the Student Code of Conduct. Apparently 23% of academic misconduct cases at a certain junior university are in Computer Science, but we expect *you all* to uphold high academic integrity and pride in doing *your own work.*