Matthew Whittaker (u5011539), COMP3100, mini-project, Team Fireman

Abstract
My mini-project is a report explaining iptables. I discuss how it interacts with the linux system from both underneath and above, how it works and how it is used. This information is relevant to our project because our project does not just require we use iptables, but involves some understanding of the deeper concepts.

I did not go in depth into the usage (command line syntax), as this is just as easily derived from the linux man pages as it would be from this report. Also, at the time of writing the rules we will be using are not known.

Task description	Importance	Time Spent	Difficulty	Link (cs.anu.edu.au/redmine/issues/$n)
Learning python	High	4 hours	High	553
Mini-project research	Low	2 hours	High	731
Mini-project writeup	Low	2 hours	Medium	731
Normal Meetings	Medium	4 hours	Low	496, 1043, 1077, 1078
Big Client Meeting	High	2 hours	Medium	1079
Install Fedora	Medium	1 hour	Low	1080
Initial Client Meeting	Medium	2 hours	Medium	524

General Discussion

Our project will be written nearly (or all) in Python, so the team is each learning Python. We each need to be running Fedora, and Tom is setting up some Fedora testing environments because our software will be running primarily on Fedora.

The meetings are going well: the team seems to communicate well and are happy with one another. The second client meeting was very useful and gave us a good idea of where we should be heading next.

APPENDIX - The actual mini-project

Overview of iptables/ip_tables

iptables is a program used for configuring the **ip_tables** kernel module. It allows users to configure rule based packet processing done by **ip_tables**. These rules can filter traffic, change packet source/destination network/application(port) addresses and perform other functions not needed by our project. **iptables** configures these rules from user space (root permissions are required).

Fairly sophisticated rules can be added to influence processing at many points in the kernel protocol stack, which means **iptables** provides solutions to a wide variety of problems. The syntax for adding rules is quite simple. The program is widely used enough that for most problems, ad hoc solutions can easily be found by searching the internet (without properly learning how to use the program).

For our project, we will be using **iptables** to manage firewall rules. We will call it each time we need to modify/query the current rules. There is no alternative to **iptables** available to us, as acquiring the necessary kernel resources (netfilter hooks) would require us to write a kernel module which is presumably beyond the scope of our project and would cause interoperability issues. This is not ideal. Although **iptables** is well suited to a command line environment it is not entirely appropriate for our project as it provides no API. For dynamic behaviour, this is inefficient: executing a process consumes much more resources than an API call. This forces us to use system calls to execute **iptables** to do the work, this is a common anti-pattern that will make our code more messy.

*This document will now enumerate the key components which form the system context in which **iptables** operates. This should to some extent clarify how it works and what it does. These components are: the protocol stack, netfilter, ip_tables, and iptables.*

*Finally the usage of **iptables** will be discussed, along with some examples.*

System components/context

Protocol stack:

- linux has protocol stack: received packets are passed through to higher layers until it reaches the application
- at each stage some processing occurs and headers are stripped
- when a packet is sent from an application, what might happen is the packet gets passed

to TCP module, which adds a header and passes to the IP module, which adds a header and passes to data link module, which adds a header and passes it on to the network interface device driver
- protocol stack diagram: http://docs.oracle.com/cd/E19120-01/open.solaris/819-3000/images/ipov.fig88.gif

Netfilter:
- throughout the protocol stack, both on upwards and downwards traversal, hooks are provided to give give kernel modules the ability to control the stack traversal process
- using a hook involves a kernel module registering a callback function with the kernel to be called whenever a packet arrives
- the hooks allow the kernel module to drop/ignore/modify/etc packets
- hook locations: pre-routing, input, forwarded, output, post-routing
- obfuscated diagram: http://upload.wikimedia.org/wikipedia/commons/3/37/Netfilter-packet-flow.svg

ip_tables:
- ip_tables is the name of a kernel module which is typically the primary user of these hooks
- its goal is to implement a set of rules for packet direction and manipulation
- it not only can filter and manipulate packets, but do this in a stateful way (for example it stores information about current TCP sessions)
- things it can be used for includes: can act as a firewall (a filter), can provide transparent proxying (by transparently redirecting traffic from one application to a different local or remote service), and can implement NAT (network address translation)
- it is rule based - users add rules to change how packets are treated
- provides tables which each attempt to implement some functionality: filtering, mangling, nat
- each table makes uses of netfilter hooks
- each table contains a chain of rules that packets are compared against, as well as extra chains that can be jumped to/from
- it does not provide any API, so to use it we will have to execute the iptables program via python/java/whatever

iptables:
- iptables is a user space program which is often executed by BASH and requires root permissions
- it provides an interface to the ip_tables kernel module
- it is the only interface to ip_tables that we have

Basic usage

iptables uses several tables, each implementing some functionality. The most common is the *filter* table (this is the default, and we for our project we will probably not implement anything beyond filtering functions). Each table has chains corresponding to a point in the protocol stack, and implements a unique set of targets (actions).

Each table has a set of *chains*. These are lists of rules. A rule consists of match criteria and a target (action). When a packet comes in, it is checked against all of the rules in order. When it matches a rule it may continue through the chain, or halt. For example a packet might be logged, in which case it will continue traversing the chain, or it may be ACCEPTed, in which case it is passed to the application layer and no more rules are checked.

The filter table has 3 built in *chains*. These are INPUT, OUTPUT, and FORWARD. The INPUT chain matches packets just before they are to be sent to the application, the OUTPUT chain matches packets just after they are sent from the application, and the FORWARD chain matches packets that the kernel is forwarding (if it is behaving as a router).

Now that we know the idea of the filter table, lets add a rule to it:

>iptables -t filter -A INPUT -j DROP

-t specifies the table to be used. Because filter is the most common table, we can omit this and filter will be chosen by default.

-A appends the rule to a chain. We chose INPUT. This means the rule will be matched against incoming packets, before they are passed to the application.

-j specifies the target to jump to if the rule matches a packet. We have specified no matching criteria, so ALL packets are matched. They are all passed to the DROP target which drops the packet and discontinues traversal of the chain.

Now, let's enable incoming web traffic:

>iptables -I INPUT 1 -j ACCEPT --sport 80

This time we didn't include -t filter, as this is the default.

Because the rules are checked sequentially, the rule can't be appended to the end because the previous rule dropped all packets. We must put the rule in front of the DROP rule, so we use -I INPUT 1 to insert the rule as the first rule.

Here we jump to the ACCEPT target on a match, which means the packet is allowed.

--sport specifies a source port to match. Port 80 is the standard http port, so this matches and

accepts all incoming traffic from port 80.

Let's check our rules:
>iptables -L

This lists all the rules for the filter table (remember it is default).

We can backup our rules like this:
>iptables-save > iptables.backup

And restore them later like this:
>iptables-restore < iptables.backup

According to wikipedia, iptables-restore changes rules faster than iptables: http://en.wikipedia.org/wiki/Iptables "Front-ends in textual or graphical fashion allow users to click-generate simple rulesets; scripts usually refer to shell scripts (but other scripting languages are possible too) **that call iptables or (the faster) iptables-restore** with a set of predefined rules, or rules expanded from a template with the help of a simple configuration file."

The second most important table is the nat table. It implements targets and chains for network address translation (anything that changes the network address, not just simple nat). This can be used for gateway routers, proxies etc. I will not discuss it any more here.

For more sophisticated operations we can perform with **iptables**, read the manual. We probably don't need to know much more, especially until we decide what functionality our software will provide.