Configuring firewall in Linux using Iptables

Ravi Kumar.M.S

January 31, 2004

Contents

1 Introduction **3**

2 Netfilter or iptables? **3**

3 Configure, compile, install, reboot **3**

4 IPTables **4**

5 Filtering **5**

6 NAT **7**

7 Mangle **9**

8 Application **9**

9 Conclusion **10**

1 Introduction

When Linux 2.4 was released, most people focused on what it would do to help the average Linux user and talked about the USB support, firewire, PCMCIA and DRI. While these are great additions to the kernel for the majority of people, often one of the major improvements over 2.2 was overlooked, even though it applies almost as much to any User as it does to a hardened network engineer. This is, of course, the inclusion of the **netfilter** system into the kernel, which provides *packet filtering* and other *more advanced IP features*. Along with netfilter comes **iptables**, which is the 2.4 equivalent of *ipchains*, and provides a user-space interface to the filtering,**N**etwork **A**ddress **T**ranslation (NAT) and **mangling modules**. Were going to look at building 2.4 with support for netfilter and iptables, then building a production level router out of it. For those of you who just have one machine, and use it to connect to the Internet, then many of the same rules apply. The Internet is one giant, generally unrestricted, network which any reasonable person would have reservations about putting any sort of machine on, never mind their own Linux system.

2 Netfilter or iptables?

Often when referring to the firewalling code in 2.4, it will blindly be referred to as *netfilter* or *iptables*, without any justification for using the specific name for it and, given that they are both very different, its worth understanding exactly what each of them do and how we should view the organisation of the firewalling code in the kernel. Netfilter is the system compiled into the kernel which provides hooks into the IP stack which loadable modules (iptables is one) can use to perform operations on packets. As netfilter uses modules for the filtering, you can use an ipchains module to provide exactly the same capabilities as the kernel level ipchains code in 2.2, or even the module for ipfwadm from 2.0. Netfilter is there all of the time, as long as it is compiled in, whether or not you are using any firewalling modules at all. IPTables is split into two parts ...

1. The user-space tools and
2. The kernel-space modules.

The kernel-space modules are distributed with the main kernel, and you compile them as you would any other module, be it sound drivers, a filesystem or USB support. There is the main ip_tables module, as well as modules specifically for NAT, logging, connection tracking and so on. These modules perform the appropriate function on the packets which they get sent by netfilter, depending on the rules which they have in their rule-list, or chain. The user-space iptables code comes in the form of a binary called iptables, which is distributed separately from the main kernel tree, and is used to add, remove or edit rules for the modules. This is comparable to the ipchains binary in 2.2. Often, when referring to iptables, it is assumed to mean the iptables binary, and we will continue to use such a standard here.

3 Configure, compile, install, reboot

Ideally, you need to have a machine which is already running a 2.4 kernel, or have the knowledge to install 2.4 on a machine currently running 2.2., as the required updates to make sure 2.4 runs without problems are outside the scope of this article. This machine is going to be for

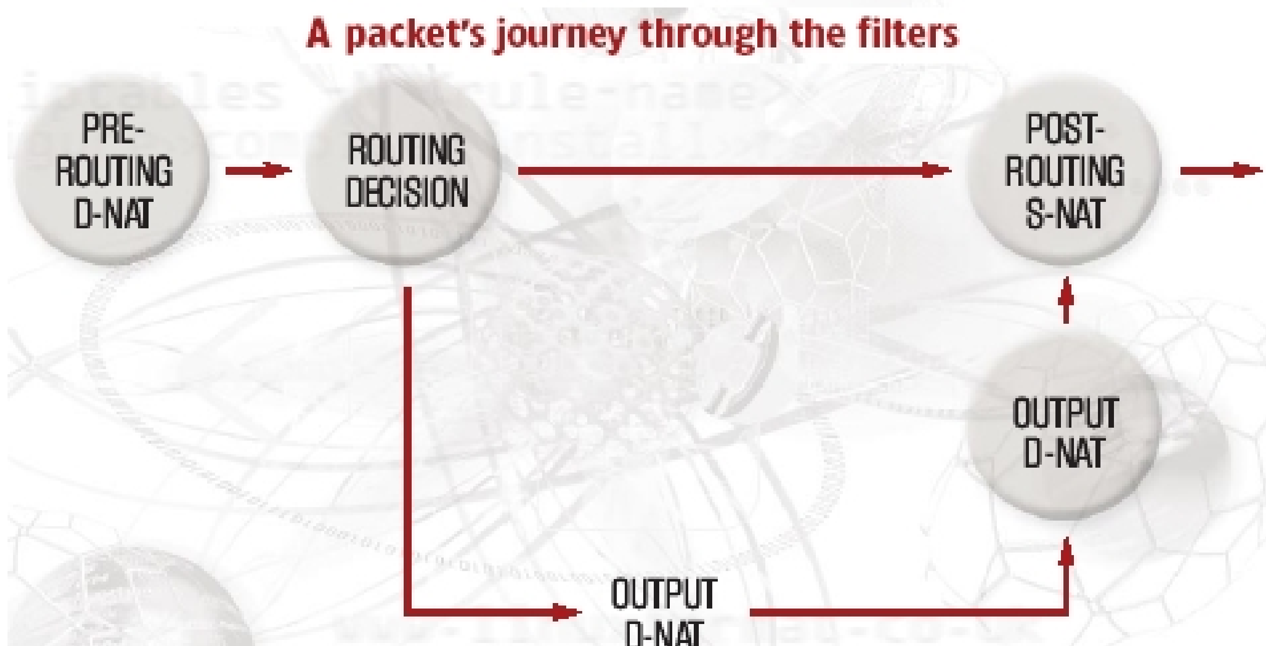
mission critical routing, so the use of the latest bleeding edge kernel is not really necessary; all we need is something which is stable, secure and is not going to corrupt our filesystems.

Aside from all of the other options which you may or may not need, there are numerous settings under Networking Options which dont directly pertain to iptables, but are applicable to many features of it. Firstly, we need to select Network Packet Filtering, which basically enables the use of netfilter, although unless youre intending to become a netfilter developer, you wont need the debugging option. You will probably also want to enable IP: advances router and IP: use netfilter MARK value as routing key. We next need to compile some modules which netfilter can use, with the IP: Netfilter Configuration sub-menu. Everything there needs to be selected as m, apart from the 2.2 and 2.0 support, unless you specifically need to use ipchains or ipfwadm on the machine while you learn to use iptables.

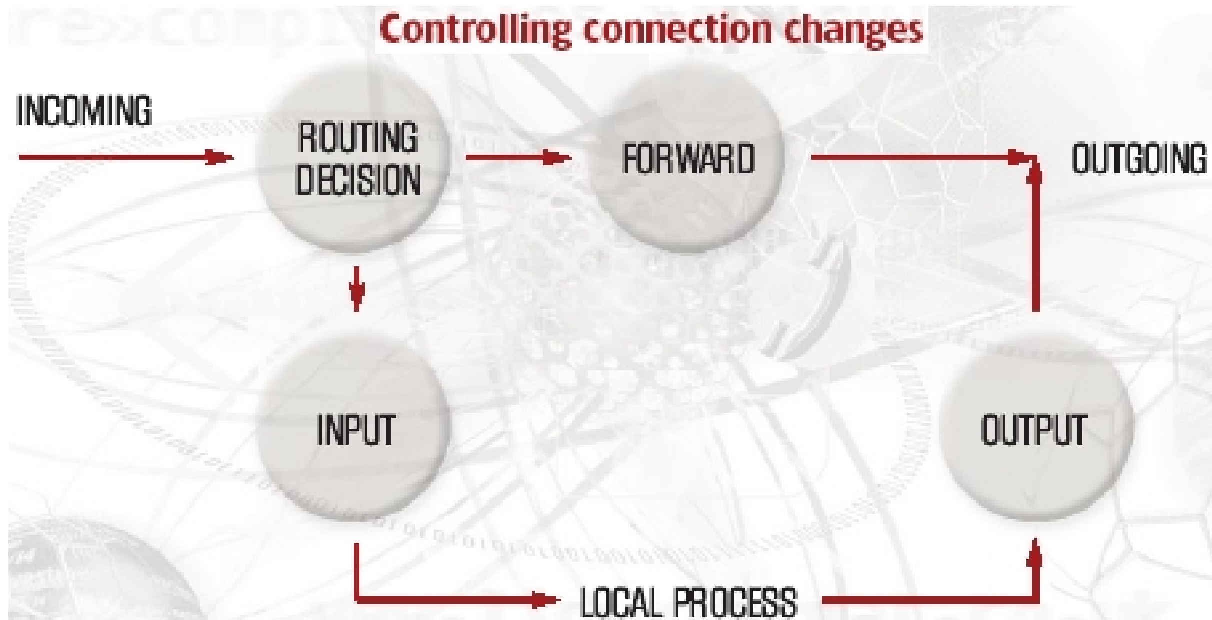
As with any kernel rebuild,
 # make dep && make clean && make bzlilo && make modules && make modules_install
then **reboot**, assuming you are using lilo.

4 IPTables

Once the new kernel is up and running happily, we can go ahead and compile the userspace iptables tool. You can download this, the latest release being 1.2, from **netfilter.kernelnotes.org**. It is basically a matter of doing **make; make install**, as *root*, and everything is sorted out. You will need a configured 2.4 kernel available for iptables to compile against, so if youve not yet built 2.4, or have deleted the source code, you might want to take a few steps back and have another go.



Next we need to load the *ip_tables module* into the kernel using
```
# modprobe ip_tables
```
A lot of the other modules are loaded automatically as we use the various features, but both `ip_nat_ftp` and `ip_conntrack_ftp` need to be loaded manually, and we will look at their usage later.



5 Filtering

As with ipchains, iptables has three lists of rules or chains for filtering. For those of you who are confused about moving from ipchains, they have exactly the same names, but have to be in upper case, so there is **INPUT**, **OUTPUT** and **FORWARD**.
INPUT applies to all packets destined for the local machine, OUTPUT for packets which originated locally and FORWARD for packets which are sent to our machine, but are not actually for it. We can, if we choose, create our own chains to organise our rules into different groups based on other rules.

We create a rule with
`# iptables -N rule-name`
and delete it with
`# iptables -X rule-name`
After this, they behave just like the three default chains, and we can flush them with
`# iptables -F rule-name`
or list their rules with
`# iptables -nL rule-name.`

Using iptables we can perform three actions on the chains which alter their rules. We can either *add*, *insert* or *delete* rules, using **-A**, **-I** and **-D**, respectively, followed by the chain name. So, if we wanted to add another rule to the end of the INPUT chain we would use `iptables -A INPUT`.

Not much use so far, as we need to specify which packets we want the rule to apply to. Matching source and destination IPs and ports is the most straight forward things to do.

If, for example, we want to block all connections to *port 23*, over *tcp*, to a local machine we would do:
`# iptables -A INPUT -p tcp --dport 23 -j DROP`
`-p` sets the IP protocol used, be it *TCP, ICMP, UDP* or one of the other more unused protocols, and `--dport` specifies the *destination port* of the packet. We can, of course, use `--sport` to specify a *source port*, but that is rarely used as connections usually use a random source port, unless they are from a specific service, such as NTP or BIND which has packets coming from

a specific port.

Those who migrated from ipchains will be familiar with the difference between DENY and REJECT. However, the people who wrote iptables thought that DENY and REJECT sounded like the same thing, so there is now DROP and DENY. DROP literally drops the packet without making any effort to clean up afterwards, whereas DENY drops the packet, then returns an ICMP packet to the source of the packet to tell it that the connection was denied.

Describing source and destination IP addresses is often used to distinguish between trusted and unknown networks, and there are a number of different ways to refer to IPs and network addresses with iptables.

Within the rule, we can use `-s` *`ipaddr`* and `-d` *`ip-addr`* to set the source and destination IPs which must match for the rule to be used. We can either use a normal IP, such as `10.1.2.4`, a hostname, such as `mail.domain.com` or a network address `10.1.2.0/24` as an example. In the latter case, it will use either the common slash notation or a proper network address/netmask identifier, and an IP is of course really a /32. If we neglect to use either `-s` or `-d` it will use `0.0.0.0/0.0.0.0` which will match any packet.

Often, we want to drop all internal traffic coming in from a remote network, such as the Internet, and this can be done with a combination of the `-d` flag, and `-i` which refers to the input network device, such as `ppp0` or `eth0`:

```
# iptables -A INPUT -d 10.0.0.0/8 -i ppp0 -j DROP
```

`INPUT` will only understand `-i`, and `OUTPUT -o`, as neither will have a device of the opposite type, but a rule in FORWARD can use both -i and -o, as it is not unlikely that a packet will come in one interface and go out of another, depending on the routing.

Previously, we would check for the SYN flag, which is usually indicative of a packet which is going to start a new connection, in order to prevent incoming connections to a machine.

Unfortunately, this is not a particularly secure way to do it, as it is fairly straightforward to create software which starts connections with malformed packets, and even if you cant do that, there are plenty of things you can download off the Internet which will do all the hard work for you. IPTables has a far better option, in the form of connection tracking modules. Every time a new connection is created, either locally, or by routing through our machine from somewhere else, ip_conntrack catches it and stores the details, so it can use the information to see which connection specific packets belong to. Now, rather than just checking for the SYN flag, we can check to see if it matches a currently established connection, which is much neater.

There are four types of connection which can exist. **NEW** corresponds to packets which are being used to create new connections. This is, of course, done by checking the connection tracking list, rather than checking any packet flags, so will apply to new connections being routed through our machine. **ESTABLISHED** relates to packets from a known connection, and **RELATED** applies to packets related to a active connection, such as an ICMP reply, or via the use of ip_conntrack_ftp, active FTP sessions. Last, but by no means last, is **INVALID** which should be dropped and are malformed or unrecognised packets.

All this matching is done with the `-m` switch, and for connection tracking, or stateful matching, we use `-m` state followed by a `--state` option, then list the packet types we do, or dont, want to match.

If we, for example, wanted to drop all `NEW` or `INVALID` packets coming in ppp0 we would use:

```
# iptables -A INPUT -m state --state NEW, INVALID -j DROP
```

Sickeningly easy, isnt it? We can have it match packets which do not match a specific state:

```
# iptables -A INPUT -m state --state !  INVALID -j INCOMING
```

Which would match all non-INVALID packets, then start to match them against rules in the INCOMING chain.

Matching specific flags of TCP packets is still important, so we can still check them using the `--tcp-flags`. This is slightly different, as it takes two options. Firstly, it needs a list of all flags it should check, then a list of the flags which should be set. If we wanted to perform a check for a `SYN` packet, that is, a packet with the SYN flag set, we would do:

```
# iptables -A INPUT -p tcp --tcp-flags SYN, ACK, FIN SYN -j DROP
```

This translated into English says: *drop all packets which have the SYN flag set, and the ACK and FIN flags not set. All other TCP flags are not checked.* The above is provided as a single option –syn, so:

```
# iptables -A INPUT -p tcp --syn -j DROP
```

is exactly the same. The `-m` flag can match numerous other things, such as source MAC address, which is useful on a network where you only want trusted physical machines accessing services, but the most important match is the rate limiting, which is very useful for log messages, or limiting the connections on a machine. `-m` limit is followed by `--limit`, which sets the rate limiting. If we use `--limit 1/s` it will allow one packet every second to match the rule. You will, however, notice that this doesnt work straight away. As default, it will allow the first five packets straight through, which is not always what we want. The `--limit-burst` option specifies the number of packets which can match the rule before it starts to limit the packets :

```
# iptables -A INPUT -m limit --limit 5/m --limit-burst 10 --syn -j ACCEPT
```

This will allow the first ten packets to pass without any interruption, then limit to one packet every twelve seconds, or five per minute. Every time we hit a limit time, but a packet has not passed, one is added to the current burst, so if we had ten packets, then none for a whole minute, it would allow five packets to pass through before starting to limit them again. After two minutes of no packets, the burst will be back to the beginning and will allow the first ten packets through again. This is especially useful for logging, as we dont want our logs being filled up with loads of repeated information. Unlike ipchains, logging is done with a LOG target, much like ACCEPT or DROP. However, unlike the others, even if a packet matches a defined LOG rule, it will continue to transcend the chain, so you would normally have:

```
# iptables -A INPUT -i ppp0 -m state --state NEW -m limit --limit 1/m --limit-burst
0 -j LOG
# iptables -A INPUT -i ppp0 -m state --state NEW -j DROP
```

in order to log, then drop, all incoming connections on ppp0. *LOG* will take two optional arguments, `--log-level` allowing you to specify a syslogd level, such as debug, info, etc, and `--logprefix`, which lets you set a textual prefix to the log entry, up to twenty-nine characters, so you can easily distinguish which rule threw up the log entry.

6 NAT

So far, weve looked at rules, chains, targets and matches, but as it is called iptables, there must be a table of some sort in there. Well, unknown to us, weve been using a table all along, although it is actually the default, so we didnt notice. **filter** is the table used to filter packets, and contains the three chains, plus any ones created using the iptables utility. There are two other tables, **nat** and **mangle**, which also exist, and were going to look at the nat table first, as it contains some of the most important changes to the 2.4 kernel over 2.2.

Firstly, we can list the chains in the nat table with:

```
# iptables -t nat -nL
```

which will throw up **PREROUTING**, **POSTROUTING** and **OUTPUT**. These three chains, much like the chains in filter, only apply to certain packets, although they are a little

broader. Packets pass through the PREROUTING chain when they enter the machine, whether they are destined for the local machine or for somewhere else, before any routing decisions are taken by the kernel, so it doesnt know where they are going. OUTPUT corresponds to any packet originating from the local machine, and POSTROUTING to any packets leaving our machine, after the routing decision is taken, but did not originate locally.

We can, if we really wanted to, DROP, ACCEPT or LOG packets using these chains, but they are mainly used for Network Address Translation, or NAT, features. You might, at first, think that it does not apply to you, but IP masquerading is a type of NAT, so if you intend to share a network connection, it may be worth paying attention.

There are two varieties of NAT, *source NAT* or *destination NAT*. It doesnt take an experienced network administrator to work out that source NAT changes the source address, or port, of packets and destination NAT changes the destination information. Because of the way NAT works, source NAT, or SNAT, only works in the POSTROUTING chain, and DNAT in the PREROUTING or OUTPUT chains.

The most obvious reason to use NAT is to traffic packets from a public network, such as the Internet, onto an internal LAN, then back out again. We might want to have all SMTP connections to our router from the outside world forwarded onto our mail server on our LAN, and to do this we would use DNAT:

```
# iptables -A PREROUTING -p tcp --dport 25 -i ppp0 -j DNAT --to 10.1.1.2:25
```

And, that is it. It will track packets coming in, and going out, so the outside world does not notice anything odd is going on, even if there are not masquerading or SNAT rules for the internal machines. The only caveat is that all packets for the connection must pass through the router, so you cant traffic packets from the mail server via a different machine to the outside world, as it will not know that it should reverse the DNAT rule when something leaves the network. SNAT is used to hide internal IPs behind public IPs, which is not quite like masquerading. We might want to have all packets coming from 10.1.1.4 to correspond to a specific external IP:

```
# iptables -A POSTROUTING -s 10.1.1.4 -o ppp0 -j SNAT --to 192.168.1.2
```

Of course, 192.168.1.2 is not a public address range, but it is just an example. What you can do with SNAT depends on the IP allocation, if any, from your ISP, so you may only be able to SNAT onto a single IP, but you can have many SNAT rules for a single external IP. **IP Masquerading** is a form of SNAT, except that it is more interested in the interface than the IP address: `# iptables -A POSTROUTING -o ppp0 -j MASQUERADE`
will masquerade all packets going out of ppp0, just like in 2.2, but it is worth knowing what the difference is. Masqueraded connections are handled just like SNAT, until the interface goes down, at which point all connections are dropped, and you have to start again. Obviously, if you have a dynamic IP, you wont want old connections with the wrong IP address hanging around, as they wont do anything useful, but if you have a static IP then you suddenly loose the ability to resume TCP connections when you redial, as the router wont remember how it SNATed them the last time.

Problems are encountered when we SNAT onto an IP belonging to an interface which the packet will not be going out of, as SNAT only changes the packet, not the routing, in the **POSTROUTING** chain. This can be combated by changing the routing for specific packets, which we will cover next.

7 Mangle

The mangle table is a little strange, as it is used to change packet properties, which wont have a direct effect on them. As with nat, mangle has the same three chains which we can use to set packet properties, specifically marking them for later rules. Marking packets is especially useful if you want to use something outside of iptables, such as iproute2, to perform an operation on a specific packet, but it cannot match all of the options we need.If we wanted to mark all packets heading for a SMTP server with the number 1 we would do:

```
# iptables -t mangle -p tcp --dport 25 -j MARK --mark 1
```

You might wonder exactly what the point of that is, but it is the first step in performing per-packet routing, as iproute2 can be used to setup routing tables based on fwmark, rather than the traditional destination IP as route does. This is quite handy if you have a quick, but unreliable DSL connection, and a slow and stable ISDN line, but want all of your mail heading across the ISDN line.

8 Application

So, we now know most of the theory, but what about using it in practice? If we just have a single machine connected to the Internet and dont want to allow any incoming connections, but want to masquerade our LAN behind it we just do:

```
# iptables -A INPUT -m state --state NEW,INVALID -i ppp0 -j DROP
# iptables -t nat -A POSTROUTING -o ppp0 -j MASQUERADE
```

Not forgetting to do:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

Notice that we dont need any ugly rules for active FTP, as we use ip_conntrack_ftp, which stops the incoming FTP connections being tracked as NEW. We might want to allow some ports, such as 113 and 25, for identd and SMTP mail delivery. We just do:

```
# iptables -I INPUT 1 -p tcp -m multiport --dport 113,25 -j ACCEPT
```

multiport is a matching option which can be used to specify multiple ports within the same rule, either –dport or –sport.

To have this happen every time we reboot, we can use a combination of iptables-save and iptables-restore to save and restore the rules, or we can just write a bash script, or pop it on the end of `/etc/rc.d/rc.local`, depending on the distribution. Usually, if youre just starting out, it is best to use a script, as you can change it, rerun it, and you can be 100% sure that if you reboot the machine, it will end up how you have it now.

However, this isnt much use if youve got a couple of hundred machines behind a firewall and want to run proper web and mail servers, and allow internal machines to access the outside world transparently.

The simplest way to do this is to setup a selection of 10.1.x.0/24 networks, and put different classes of machines on them, such as front end servers, back end servers and workstations, as we will want to apply rules to each class in order to secure the network. The actual internal structure of the network depends on the services used, but it is not best to plug backend servers onto a hub along with a load of web servers. The public IPs will all be allocated to the public interface on the router, so the internal machines need not care which IP they are using, let alone how anyone gets to them.

Firstly, we need to take control of the IPs we are going to use, which is nothing more than making sure the public side of the LAN, which will probably consist of little more than a router for the line, knows where to send packets destined for the IPs we have. The quickest way is

to setup IP Aliases for the network interface which faces the outside world, eth0 in our case, so we might have eth0:mail as the interface for the mail servers IP. We could, instead, use arp to publish the NICs MAC addresses relationship to the IP with **arp -Ds 192.168.1.2 eth0 pub**, where `192.168.1.2` is the public IP. Which ever method is chosen, it will have to be performed whenever the machine is rebooted, so it should be inserted within the iptables setup script.

We will want all packets going to `192.168.1.2` to head for our mail server 10.1.1.2, and anything coming from `10.1.1.2`, that is new connections, to look as if they are from `192.168.1.2` :

```
# iptables -t nat -A PREROUTING -p tcp -i eth0 -d 192.168.1.2 --dport 25 -j DNAT
--to 10.1.1.2:25
# iptables -t nat -A POSTROUTING -s 10.1.1.2 -o eth0 -j SNAT --to 192.168.1.2
```

All packets coming from our workstation network should only come out of one IP, which is easily done with another SNAT rule:

```
# iptables -t nat -A POSTROUTING -s 10.1.3.0/24 -o eth0 -j SNAT --to 192.168.1.1
```

We can also drop packets from the PREROUTING chain, which makes it easy to drop all incoming connections to any machine, which does not have a specific DNAT rule, and as were performing operations based on interface, rather than IP, we dont need to explicitly allow 10/8 traffic from being routed through our machine.

We hit a problem with this sort of setup, as if 10.1.3.2 hits our public IP for the mail server 192.168.1.2, the router translates it to 10.1.1.2, so the mail server gets a packet to 10.1.1.2 from 10.1.3.2, which wont travel back through our router in order for the DNAT to be reversed. This is quickly and easily combated with a SNAT rule, which will make all internal connections to any of our public IPs look as if they are coming from the router, and the DNAT will be reverse correctly: `# iptables -t nat -A POSTROUTING -i eth1 -d 192.168.1.0/24 -j SNAT --to 10.1.1.1`

Assuming our router has the internal IP of 10.1.1.1 on eth1. We can extend this further, to force all of the workstations to use a *squid web cache* which lives on `10.1.1.3:3128`.

A simple DNAT rule:

```
# iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 80 -s 10.1.3.0/24 -j DNAT
--to 10.1.1.3:3128
```

Squid needs a couple of options to perform correctly as a transparent cache, but those are well documented at **www.squid-cache.org**.

9 Conclusion

By now, we should have some variety of network running with iptables, and once youve spent time working out all its eccentricities, youll wonder how you ever got along with ipchains. Even for a single machine connected to the Internet with a 56k modem, as a large proportion of people are, the sheer simplicity of its use makes it very difficult for even the most inexperienced user not to make the effort and having a go, assuming they can get 2.4 up and running in the first place. Simply, iptables offers many, many features which ipchains is technically incapable of, and when my own network is sitting on the Internet all hours of the day and night, Im not going to pick second best.