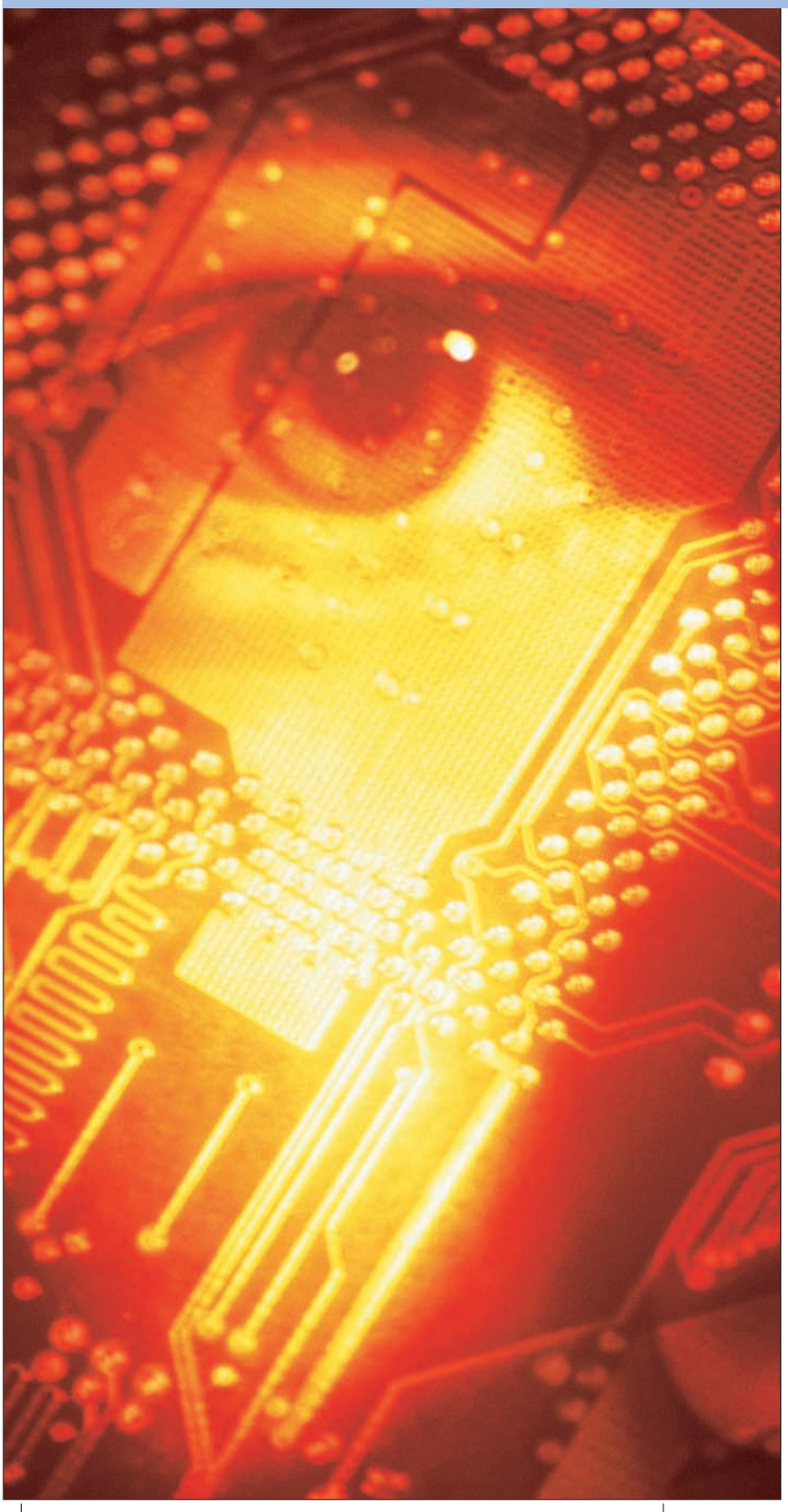
Network security
iptables



PART 1 **In the first of a two part series on iptables, DAVID COULSON looks at installation and configuration.**

Network security has always been a major concern for businesses with online services. With DSL, cable and other 'always-on' technologies becoming the norm, home and small businesses are becoming more vulnerable to attacks and the possibility of having a system exploited. In a perfect world, we could leave unprotected systems online all of the time without feeling threatened by outsiders, but unfortunately the Internet is far from perfect, and there are individuals out there who are happily scanning ISP net blocks for exploitable systems in search of their latest target.

While it's important to keep software up-to-date and limit the amount of potentially exploitable software from our systems, it is also equally important to limit what is available to the outside world. Of course, even the best administrators occasionally make mistakes and unfortunately, there are occasions when a system may be more open than is desired due to a configuration fault.

A firewall can be used to filter out packets heading from the outside world to our network, which would normally allow someone to connect to a network service running on our system. It's worth remembering that, generally, a firewall is a layer 3 and 4 device, so it operates at the IP and TCP/UDP level. This means that a firewall is unable to filter based upon contents of packets. For example, a firewall does not filter email for viruses or check for specific unwanted content within web documents. Usually something more specific to a firewall is required for these capabilities, although the actual software used may be run on the same system as the firewall. Notwithstanding this, a firewall is the first line of defense against an outside attack, so it's well worth taking some time to set it up correct and ensure that our system is as secure as it can be.

Firewalling with Linux

Linux has had firewalling capabilities for a very long time. Back in the 2.0 kernels, we had the *ipfwadm* package, which did basic firewalling. With 2.2, *ipchains* was implemented and offered a significant number of improvements over 2.0. Now, with the 2.4 kernel, all of the old firewalling code was ripped out and a new system implemented. Rather than simply provide firewalling

capabilities, they decided to simply write a system which can take plug-ins, allowing any firewalling system to be used with Linux. This subsystem is known as *netfilter*, and allows *ipfwadm* and *ipchains* to be used with the 2.4 kernel series using the ports of each of these to *netfilter*. However, a far more powerful firewalling system was implemented, known as *iptables*, which not only has far more capabilities than earlier efforts, but it can also handle having these extended through the use of kernel modules, making for a very flexible installation.

We're going to start off by assuming that the Linux system which will be our firewall already has a 2.4 kernel running happily and that building a kernel is not too much of an issue. All distributions which use 2.4 should have *iptables* available, so unless we're using our own kernel, or want to add some extra capabilities to *iptables*, then we won't need to rebuild the kernel.

All of the *netfilter* options live within 'Network options', under IP: Netfilter Configuration. Generally, we'd pick all of the options and compile them into the kernel, except for *ipchains* and *ipfwadm*. Having *iptables* capabilities available as loadable modules can make life a little more complex, plus it is rare to actually have the system up and running without the firewall enabled, so nothing is really being achieved by having them set up as modules. However, some people do like having modules, and *iptables* will automatically load the module as needed when a rule is installed which uses that particular module. The only other reason to compile everything for modular use is if we already have a 2.2 or 2.0 firewall setup, and want to continue with *ipchains* or *ipfwadm* under 2.4 before upgrading to *iptables*. It's not possible to compile both *iptables* and *ipchains* into the kernel, for obvious reasons, but we can build them both as modules and 'modprobe ipchains' if we want to use *ipchains*.

Once we've picked what we need to compile into the kernel, we simply need to run the standard **make bzlilo** and wait for the kernel to install. If needed, **make modules && make modules_install** should be performed in order to install the new *iptables* modules which were compiled. After a reboot, a quick look at the boot logs with **dmesg** should show:

```
ip_tables: (C) 2000-2002 Netfilter core team
```

Of course, if it's available as a module, then one needs to **modprobe ip_tables** before checking for that boot message. If for any reason that message doesn't appear, then it's worth checking the kernel configuration and that

Netfilter and iptables have lots of configuration options, but most people will just want to select them all.

the recompiled kernel is actually running before going any further and wondering why nothing is working.

Getting started with iptables

As well as the kernel capabilities, we also need to obtain the user space program used to setup our firewall, *iptables*. This can be downloaded from **www.netfilter.org** or from our distribution's FTP site. Any distribution which comes with a 2.4 kernel should also include the *iptables* tools. A quick check to ensure that everything is happy is to run:

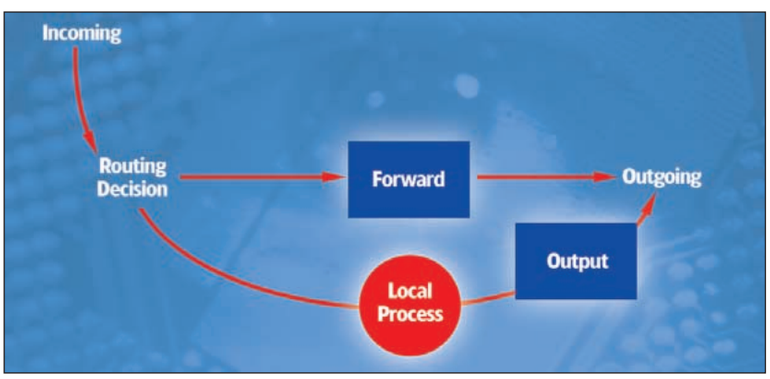
```
iptables -nL
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

If something different appears, ensure that the current release of iptables is the one being used, as older distributions will have older releases of the *iptables* tools, which may cause problems.

iptables splits the packet handling into three different tables, each of which contain a number of chains. The firewalling rules, which we create, are included within a particular chain. The three tables are filter, nat and mangle, all of which we will be looking at in this two-part *Linux Pro* series. The first is quite obvious, and is used for packet filtering. nat is used to provide packet modification capabilities, such as NAT/PAT and, of course, IP masquerading. The final table is the most obscure and is used for setting packet options, such as the Type Of Service (TOS) field and marking packets for further filtering or routeing.

We're starting off by looking at the filter table, which contains three chains. These three chains allows us to filter specific traffic on our system. The INPUT chain is used for traffic which is entering our system and belongs to an IP address which is on our local machine. OUTPUT is used for traffic which originated on the local system, otherwise known as the firewall. Both of these are not used for traffic which is being routed between two network

Each chain in the filter table checks packets depending upon how they are routed through, to or from our firewall.



interfaces on our firewall, which is left for the FORWARD chain. Using these three chains, we can easily block traffic we don't want, plus we can clearly define what type of traffic to allow onto our internal LAN.

The basic syntax of an *iptables* command is:

```
iptables -A INPUT -s 10.0.0.0/8 -j ACCEPT
```

This would add a rule into the INPUT chain, which matches any packet with a source address in the 10.0.0.0/8 netblock. If a packet matches this criteria, then it would use the ACCEPT target, which simply allows the packet on through. *iptables* supports CIDR and netmask syntax for defining networks within rules, so using 10.0.0.0/8 and 10.0.0.0/255.0.0.0 are equally acceptable. When adding a rule, it is added onto the end of the chain of rules, so will be the last one checked. As within the filter rule, whichever rule matches first will be the target for the packet and no other rules will be checked. We can also insert a rule into a specific location of the chain, ensuring that it is checked before other rules. Instead of using **-A** we use **-I** instead:

```
iptables -I INPUT 3 -s 10.0.0.0/8 -j ACCEPT
```

This would insert the rule into the third 'slot' of the chain. Of course, if there is currently only one rule in the chain, it would just be tacked on the end. For speed, if we neglect any slot value with **-I**, the rule will be inserted into the top of the chain, making it the first rule which is checked. Removing rules from a chain is equally as easy. We can either remove a rule in a specific slot location, or remote it based upon the options used when inserting or adding the chain.

To delete the first rule in the chain, we would do:

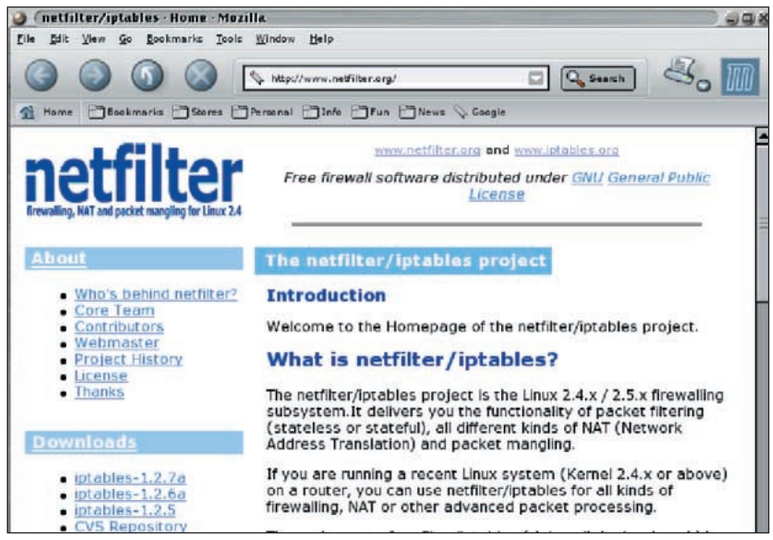
```
iptables -D INPUT 1
```

Or to delete the above rule which we inserted:

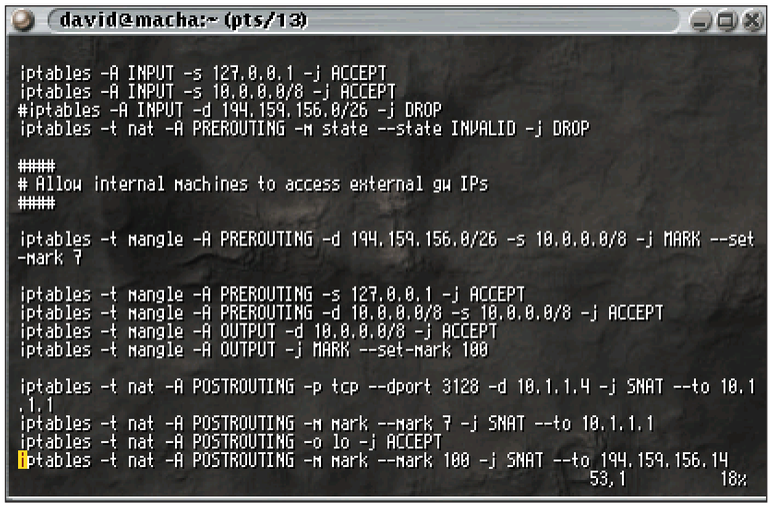
```
iptables -D INPUT -s 10.0.0.0/8 -j ACCEPT
```

If for some reason there are two rules within a chain with exactly the same options, then **-D** will delete the first one it finds in the chain.

There are three main targets for a rule within the filter table. The first we've already looked at, which is the ACCEPT rule. This allows the packet to be passed through the firewall without any noticeable interaction with the firewall. Of course, we want to block packets with our firewall, and *iptables* supports two different ways to do this. The first is the DROP target which simply drops the packet



www.iptables.org/ contains all the information on installing the user space utilities required to properly configure your firewall.



Creating a bash script containing firewall rules is the easiest way to maintain a system with huge rule sets.

to the floor, as if our system was not even there. This type of packet blocking is known as 'stealth' firewalling, as it the firewall does not respond to the client when it blocks a packet. The other way we can block a packet is with the REJECT target, which drops the packet then sends a ICMP reply back to the client telling it why the connection failed. The default response is icmp-port-unreachable, although by using the **--reject-with** option we can send an alternative ICMP packet back, such as icmp-net-unreachable, icmp-host-prohibited, and in the case of a TCP connection, we can send a tcp-reset back which causes the client to reset the TCP connection to our system. Which ICMP packet we choose to respond with depends upon the impression of our network we want to supply to the remote system..

Matching packets

Matching packets is, of course, the most important aspect of our packet filtering setup, so we need to be able to clearly define which packets we want to block and which we want to allow through. The two most basic match conditions are source and destination address of the packet, the first of which we looked at earlier. These can either be individual IP addresses or a whole netblock, depending upon what we're trying to achieve. If we wanted to block packets heading to 192.168.1.2 from anything on the 10.0.0.0/8 network, we would do:

```
iptables -A INPUT -s 10.0.0.0/8 -d 192.168.1.2 -j DROP
```

We can also match based on protocol used, such as TCP, UDP, ICMP and so forth, as well as the specific port or service type used by that protocol. As an example, a common usage is to block connections to port 113 via TCP, which is used by identd:

```
iptables -A INPUT -p tcp --dport 113 -j REJECT --reject-with tcp-reset
```

Of course, we can mix the protocol and source or destination address into one whole rule, as appropriate.

```
iptables -I INPUT -p tcp --dport 113 -s 10.0.0.0/8 -j ACCEPT
```

When we specify a protocol we can either use the abbreviated name, such as **tcp**, **udp** or **icmp**, or we can use it's numeric reference, 6, 17 and 1 respectively. If for some reason *iptables* complains about a protocol name,

ensure that it is defined in /etc/protocols, as the system uses that file to associate protocol names with the numeric assignments. Ports are setup the same way, in /etc/services, although it's worth checking the port number against the service name to ensure that nothing has been changed. We don't want to block 'auth' expecting that to refer to '113/tcp', when in fact someone has changed it to refer to '1113/tcp'.

We can also specific a 'match' option, using the **-m** flag. This allows us to use a kernel module to provide extra packet matching capabilities, the most popular usage of which is for connection tracking matching. By matching based upon the state of the actual connection, rather than simply the packet, we can ensure that connections which originated within our network are permitted, even if everything else is dropped. It can also prove useful for UDP packets, as UDP is a connectionless protocol, as it will track what we sent out and allow that system to send a reply to us through the firewall without opening it up.

The 'state' match has four different types of connection which we can match against. The most useful is ESTABLISHED, which corresponds to a connection which is already up and running. If the connection originated within our network, as soon as the packet passes through our firewall on its way to the Internet, it is tracked as ESTABLISHED. We also have a RELATED connection state, which is provided by a protocol helper module. The most common use for this is with FTP by using the ip_conntrack_ftp.o module, which allows us to track FTP connections back into our network properly, as when we download from a FTP server, it will try to make a TCP connection back to our system. There is also a 'NEW' state, which means that the packet is part of a new connection, meaning that it has not yet been tracked by the connection tracking system. Finally, we have the INVALID state which means that the connection is in an invalid state, so generally these should be dropped.

As a basic rule, we want to allow all ESTABLISHED and RELATED packets into our network, and selectively allow NEW packets through depending on the destination port.

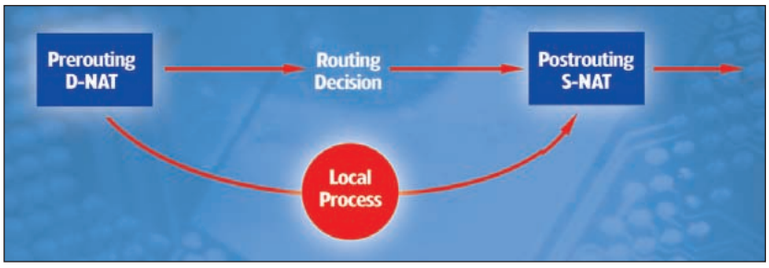
```
iptables -A INPUT -m state --state INVALID -j DROP
iptables -A INPUT -m state --state NEW -j DROP
iptables -A INPUT -p tcp --dport 22 -m state --state
NEW -j DROP
iptables -A INPUT -m state --state RELATED,
ESTABLISHED -j ACCEPT
```

The above is a simple firewall configuration, which allows everything which originated within our network to be handled properly, plus it allows ssh in. However, we only want to drop NEW connections if they originated from the Internet, as we want to use other services internally, so we need a way to match based on the network connection the packet originated on.

This is done using the **-i** and **-o** flags, which refer to the 'in' and 'out' interfaces. The INPUT and OUTPUT chains can only use **-i** and **-o** respectively, but 'FORWARD' can use both **-i** and **-o**, as the packet is being passed between two network interfaces.

If we just want to block NEW packets which came from our PPP connection to the Internet, we could do:

```
iptables -A INPUT -i ppp0 -m state --state NEW -j DROP
```



The three NAT chains try to match packets at various stages of the kernel's routeing decision for the packet.

or, if the PPP connection may have a name other than ppp0, we can drop NEW connections on all PPP interfaces:

```
iptables -A INPUT -i ppp+ -m state --state NEW -j DROP
```

When we're dropping packets, it's often important to make a log of these so they can be inspected for any attack against the network. To log packets which are matched by a particular rule, we need to set the target to LOG. If the rule already exists, we need to duplicate it and set the target of the first rule to LOG, as the LOG target will log the rule then allow the system to continue to search within the chain for another rule which matches.

```
iptables -A INPUT -i ppp+ -m state --state NEW -j LOG
iptables -A INPUT -i ppp+ -m state --state NEW -j DROP
```

Network Address Translation

NAT allows us to modify the source or destination address or port of a packet, allow it to be redirected, or so that it appears to come from another system. The most popular use of NAT is for IP Masquerading, where we have a LAN of systems using private IP addresses, which have network connectivity through a gateway which performs NAT on their packets so that all connections appear, at least to the outside world, to have come from the gateway system.

The nat table has three different chains, each of which is checked at a particular position in the routeing of a packet. The first is PREROUTING, who's rules are checked for a match before the system attempts to route the packet anywhere. PREROUTING is where we perform destination NAT, or DNAT. We can also perform DNAT within the OUTPUT chain, which is for packets which originated locally, as they are never checked by PREROUTING. Finally we have POSTROUTING, which is the last chain checked for a rule match, which is where source NAT, or SNAT is performed.

With a static IP address, we can easily perform IP Masquerading using the SNAT target, by modifying all outgoing packets with a source address matching the external IP of our gateway.

```
# iptables -t nat -A POSTROUTING -o eth0 -j SNAT --to
194.159.156.1
```

For dynamic IPs, there is the 'MASQUERADE' target, which is a special type of SNAT. Not only does it mean we don't have to provide an IP address, but whenever the interface goes down, tracked masqueraded connections across that interface are dropped. For static IP addresses, if we use masquerade, if the connection goes down, when we come to reconnect our existing TCP connections will be dead, making the use of a static IP somewhat limited. We setup masquerading with:

```
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

Comments on this article to the usual address please. ■

NEXT MONTH

We're going to take a look at some more complex uses of NAT, the mangle table as well as GUIs aiding in the configuration of *iptables*. More information, including HOWTOs and other documentation, on *iptables* and *netfilter* can be found at http://iptables.org/