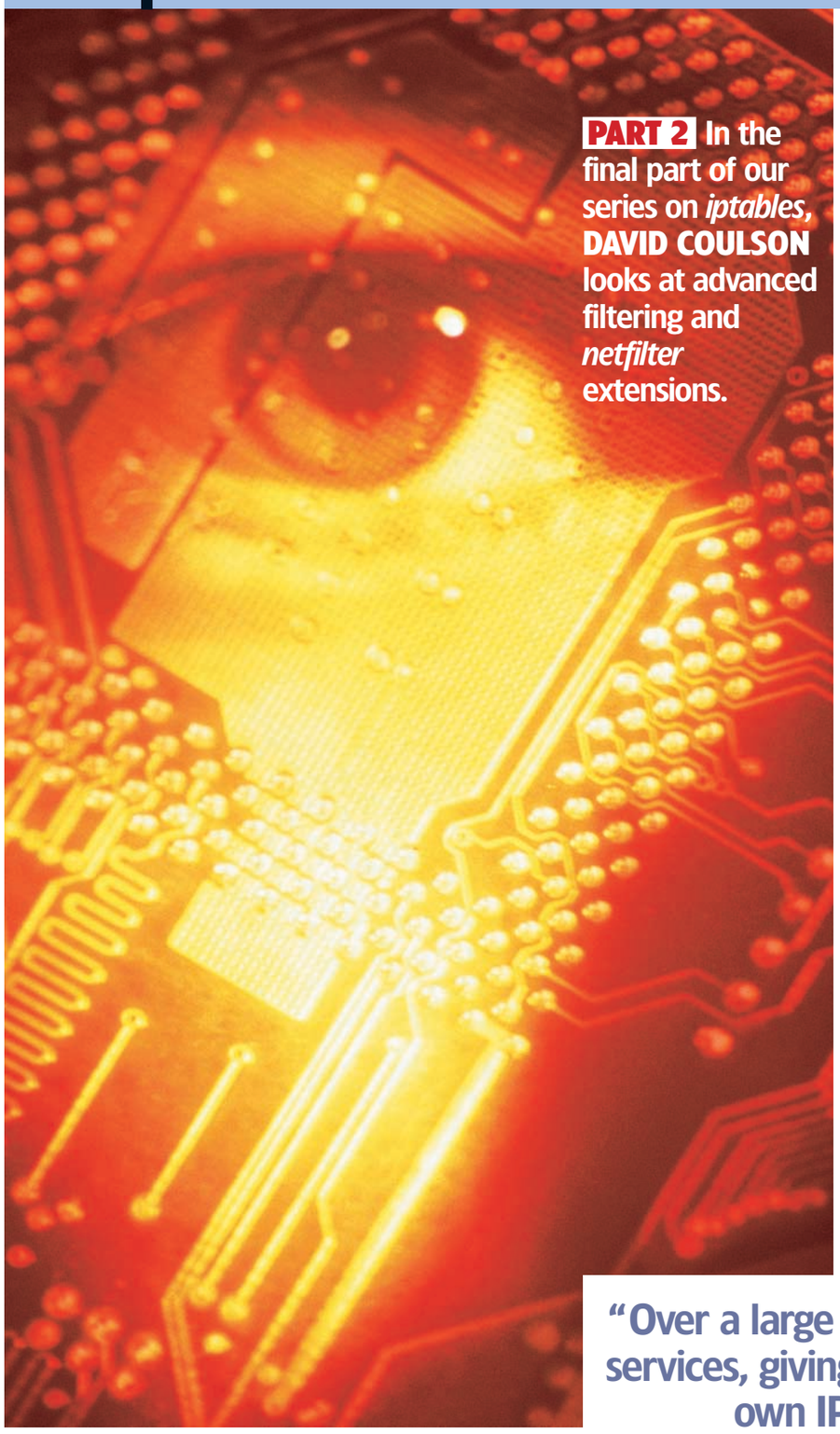
Network security

iptables



PART 2 In the final part of our series on *iptables*, **DAVID COULSON** looks at advanced filtering and *netfilter* extensions.

Last month we looked at the basics of *iptables* with the Linux 2.4 kernel, as well as example configurations for simple firewalls and gateways. However, for a production firewall implementation, or situations where we need to finely tune the packet filtering, we have to look at some of the more advanced capabilities of the netfilter system and how we can make use of those with *iptables*.

Destination NAT

On a large network with many systems running public-facing services, such as web and mail servers, giving each front-end system its own IP is often rather expensive. Each system may be running one or two services, but requires a single IP to perform that function.

Instead, another option is to use the Destination NAT, or DNAT, capabilities of iptables in order to redirect packets from the front-end IPs onto the servers using internal IP addressing. This permits us to run many services on a single IP address, even if these services are actually provided by more than one physical box. Each system will have an IP in the 10.0.0.0/8 range, allowing structured networks to be created, offering a DMZ and secure back end networks for databases and file serving. On the outside, all we need to do is have our gateway system ARP for the external IP addresses, then we can proceed to modify the packets using DNAT.

If we assume that our public range is 192.168.1.0/27, we can use a number of DNAT rules to rewrite the packets so that they hit our internal systems:

```
iptables -t nat -A PREROUTING -i eth0 -d 192.168.1.5 -p
tcp —dport 80 -j DNAT --to 10.1.1.2
```

This does, of course, assume that our web server is listening on 10.1.1.2, using port 80, and that the eth0 device on our gateway is the interface connected to the public Internet. By expanding upon this rule, we can easily forward packets from external IP addresses:

```
iptables -t nat -A PREROUTING -i eth0 -d 192.168.1.5 -p
tcp --dport 80 -j DNAT --to 10.1.1.2
iptables -t nat -A PREROUTING -i eth0 -d 192.168.1.5 -p
tcp --dport 25 -j DNAT --to 10.1.1.2
iptables -t nat -A PREROUTING -i eth0 -d 192.168.1.5 -p
tcp --dport 53 -j DNAT --to 10.1.1.8
```

"Over a large network with public-facing services, giving each front-end system its own IP is often rather expensive"

```
iptables -t nat -A PREROUTING -i eth0 -d 192.168.1.5 -p udp --dport 53 -j DNAT --to 10.1.1.8
iptables -t nat -A PREROUTING -i eth0 -d 192.168.1.5 -p tcp --dport 22 -j DNAT --to 10.2.1.3
```

If we don't specify a port following the – to argument to DNAT, *iptables* will rewrite the destination IP and keep the port the same. Now, rather than wasting four IPs to run services on different systems, we have one IP which handles four different services, for four different boxes. We can also quite easily modify the external IP address block of the network without having to reconfigure any of the internal systems.

This type of setup does present a number of rather interesting issues. While it will work fine with an external connection, if we want a system on the internal network to access services via their external IP address, we need to add a few extra iptables options. The problem occurs when a system, such as 10.2.1.8 makes a connection to 192.168.1.5:80. Our firewall will modify the destination of the packet to 10.1.1.5 and will then send a packet back to the source. However, the source address is still 10.2.1.8, so rather than routeing the packet back via the gateway machine for correct reversal of the DNAT operation. 10.2.1.8 will see a packet coming back to it from 10.1.1.5, rather than 192.168.1.5, and will assume it's a bogus packet and drop it.

This can be solved by performing a SNAT on the packet when it enters our gateway, so that the source address is set to the internal IP address of our gateway. The returning packets form the server will be routed via the gateway and will end up back to our client correctly. The only issue with this is that all clients will appear to come from 10.1.1.1, rather than the unique IP address of each system on the internal network. This can present some problems, although by using a proxy server such as *Squid*, one can get the original IP address via the HTTP_VIA or HTTP_FORWARDED_FOR headers from within a CGI or PHP script. To allow systems from our 10.0.0.0/8 range to access external IPs correctly, we would use a iptables configuration as follows:

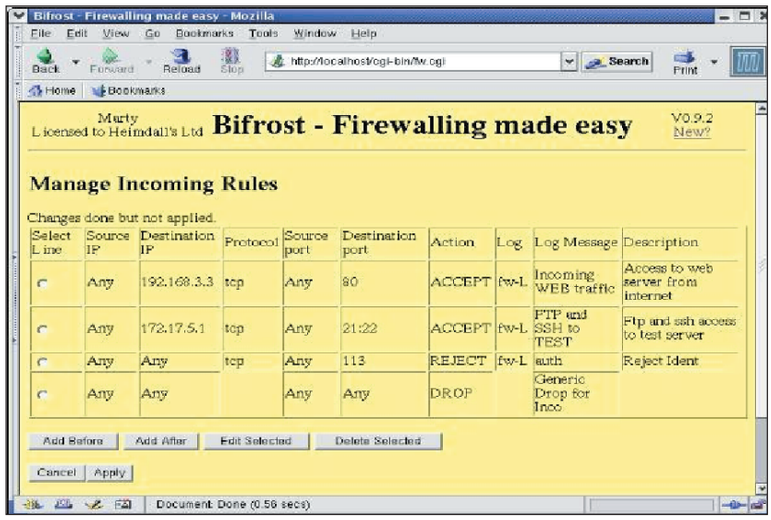
```
iptables -t nat -A POSTROUTING -s 10.0.0.0/8 -d 192.168.1.0/27 -j SNAT —to 10.1.1.1
```

Of course, if we have both a front-end and back-end internal network, we can DNAT the packets to an IP from either range, as long as the IP belongs to the gateway:

```
iptables -t nat -A POSTROUTING -s 10.1.0.0/16 -d 192.168.1.0/27 -j SNAT —to 10.1.1.1
iptables -t nat -A POSTROUTING -s 10.2.0.0/16 -d 192.168.1.0/27 -j SNAT —to 10.2.1.1
```

Note that we have not specified a interface, since if we have many internal interfaces,

With an internal network with this type of configuration, we also need to ensure that packets appear to come from the correct IP. This is especially important with services such as DNS, which do IP-based authorisation for notification and transfers. With a master/slave DNS configuration, we would configure our slave to use 192.168.1.5 as the master, since that is the IP which our DNS server is accessible via. However, when our system sends a notify following a zone update, if it comes from the wrong IP address and the slave will ignore it. Naturally, if DNS servers out of sync for the refresh interval, then it might cause problems. We can perform SNAT on specific packets, based on what TCP or UDP service it applies to:



Bifrost handles iptables configuration over HTTP, but it's not freely distributable so is unlikely to become hugely popular.

```
iptables -t nat -A POSTROUTING -o eth0 -s 10.1.1.8 --dport 53 -p tcp -j SNAT --to 192.168.1.5
iptables -t nat -A POSTROUTING -o eth0 -s 10.1.1.8 --dport 53 -p udp -j SNAT --to 192.168.1.5
iptables -t nat -A POSTROUTING -o eth0 -s 10.0.0.0/8 -j SNAT --to 192.168.1.2
```

In this configuration, we have set our default SNAT address to be 192.168.1.2, however any packets heading to port 53 will have it's source address set to 192.168.1.5, in line with our incoming DNS configuration.

Mangling

As well as performing filtering and NAT, we can also 'mangle' packets. Mangling simply allows us to modify packet options, such as the TOS field or to mark the packet with a value. The 'mangle' table in iptables has five different chains, so we can mangle packets at almost every stage of their processing and routeing. However, to actually make use of mangled packets in something other than iptables, we need to compile in certain options, such as the ability to route based on **fw mark** or to perform QoS matching based on the packet mark.

If we want to mark all packets which belong to ssh sessions so that we can route them out over another link, we need a mangle entry:

```
iptables -t mangle -A PREROUTING -p tcp --dport 22 -j MARK --set-mark 1
```

This won't mark packets which head out of our firewall box, so we need to duplicate this in the OUTPUT chain:

```
iptables -t mangle -A OUTPUT -p tcp --dport 22 -j MARK --set-mark 1
```

We can then perform routeing, or QoS, based on this mark. As neither of these support packet matching based on protocol or port, we have to use marking to handle routeing. This type of routeing is known as 'policy routeing', as it is based on something other than packet destination, and we need to use the iproute2

packet in order to configure the correct rules and create a separate routeing table for these packets:

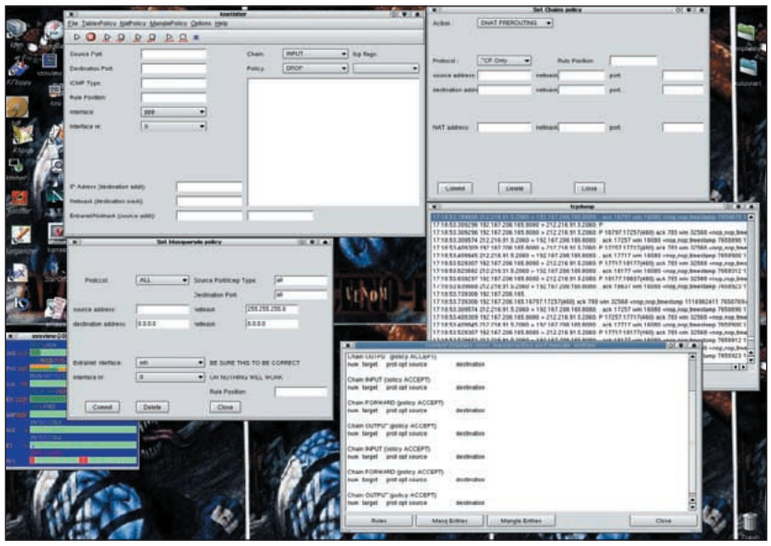
```
ip ru add table 10 fw 1
ip ro add table 10 default via 192.168.2.1
```

Of course, this will require an appropriate SNAT rule to modify the source address appropriate to the outgoing interface. We can send packets out into the Internet with the incorrect source IP, but this may cause issues of the ISP blocks packets not belonging to its own network to avoid spoofed packets heading onto the Internet.

Graphical IPTables Configuration

The command line is always nice and welcoming for die-hard Unix users and those who really want to know what's happening behind the scenes. However, if you've just got one or two boxes and it's more important to get it all working quickly and easily, there are a number of different GUI front-ends to *Netfilter* available. Of course, one still needs an understanding of firewalling, as it's still just as easy to create a completely broken configuration where the firewall isn't of any use.

Possibly one of the most popular GUIs for *netfilter* is *knetfilter*, from the KDE project. This is a fairly simple, yet comprehensive, tool which allows all aspects of the iptables system to be configured. The name *knetfilter* is somewhat inappropriate, as it won't work with anything other than the *iptables* modules, so if we're trying to use *ipchains* or *ipvs* with *netfilter*, it's not going to work with those. More information on *knetfilter*, along with documentation and source code, can be found for your perusal at **http://expansa.sns.it/knetfilter/**



***Knetfilter* is a simple KDE-based *iptables* configuration tool for those who don't want to get their hands dirty with the command line.**

For headless servers, having a non-X GUI interface is very useful, and more often than not, it's a web interface

NETFILTER PATCH-O-MATIC

Getting the latest firewall capabilities for the Linux kernel can prove complex, but *patch-o-matic* makes life easier.

As *netfilter* has its capabilities provided by loadable kernel modules, there are a great number of third-party kernel modules available which allow our firewall to do somewhat obscure things. Of course, among the nonsense, there are a number of very valuable contributions to the *netfilter* project. All of these patches can be downloaded from the **iptables.org** site, and we can happily apply each of them in turn. However, this is very tiring and boring, not to mention if a patch fails, we have to decide if we should continue with other patches, or just give up. Fortunately, the *netfilter* project has a very useful suite of utilities which makes all of this much easier.

Netfilter provides a system known as *patch-o-matic* which has a nifty system to manage patches and ensure that you don't end up with a half patched kernel that won't compile properly. *Patch-o-matic* can be downloaded either as a tarball or from netfilter cvs, depending upon how stable you want your firewall to be and if you need some of the new exciting netfilter modules. Both tarball releases and cvs checkouts contain the two main patch groups:

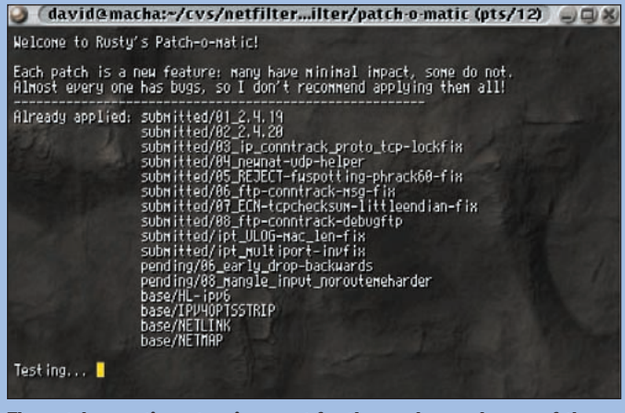
base – The current 'stable' netfilter module release

extra – Development modules

Patch filter also includes the netfilter patches for the last few stable kernels, so if you're still using 2.4.18, it will automatically upgrade you to the current netfilter release in the stable kernel tree before applying lots of other things. To apply the patch-o-matic patches to our kernel tree, we use the aptly named **runme** command passing it our kernel source path in the KERNEL_DIR environmental variable:

```
KERNEL_DIR=/usr/src/linux ./runme extra
```

This will step through each patch in turn and attempt to install it. If for some reason the patch won't apply, *patch-o-matic* gives you the option of testing to see if it works if it is applied anyway.



The patch-o-matic system is a must for those who need some of the more advanced *iptables* capabilities.

Frequently *patch-o-matic* will complain about missing files, but the patch will actually apply perfectly fine. For a 'base' patch, this isn't much of a problem, but with an 'extra' patch, it gets very frustrating pressing 'Y' and 'Y' every few patches. So that *patch-o-matic* is useful to people who don't have time to watch what it's doing, there is a **--batch** option which will test patches which it has problems with. It will stick as about patches which cause rejects, but these are rare, and generally depend upon the state of your kernel tree, rather than the *netfilter* patches themselves.

Patch-o-matic contains a wide variety of patches, so here's a quick rundown of the best:

fuzzy

This option adds CONFIG_IP_NF_MATCH_FUZZY, which allows you to match

which proves popular. *Bifrost* fits this need, by having a very simple web interface which runs via *Apache*'s mod_cgi system. Unfortunately, it is not available under a free license, so if we want full control of your firewall, cash has to be duely coughed up. It's open to the individual as to whether *Bifrost* is worth the cash, but it does what it does well. *Bifrost* can be downloaded from **http://bifrost.heimdalls.com/**

> **"Administrators can save a great deal of time by using *Firewall Builder*, as opposed to handling each system individually"**

However, the *LXF* vote goes to *Firewall Builder*, which is a cross-platform configuration builder for firewalls, which supports iptables to Cisco PIX, with a whole selection of things in between. It is a vice nicely implemented system, which allows multiple firewalls to be configured from the same front-end. While it may be a little more complex to install than other systems, such as *knetfilter*, for a large network where nearly all machines are firewalled, administrators can save a great deal of time by using *Firewall Builder,* as opposed to handling each system individually. Plus, as an added advantage for large networks, *Firewall Builder* integrates with ucd-snmp, so that monitors and logging software can be developed to watch and tune the firewall in real-time. *Firewall Builder* can be downloaded from **www.fwbuilder.org/** As it can handle a large number of other platforms as well as *iptables*, it's certainly worth trying to see if it fits your needs.

Going further with netfilter

In this series we've only looked at the iptables modules which can be used with netfilter. As *netfilter* is a completely modular system, there are a wide variety of netfilter modules to provide varying network capabilities to a Linux system. While most people will know of the ipfwadm and ipchains modules for backwards compatibility with 2.0 and 2.2 kernel firewall systems, the popular Linux Virtual Server has been ported to 2.4 using the netfilter subsystem. This makes it somewhat more attractive, as it does not require any patches to be applied to the kernel. As always, the more patches which a kernel is using, the longer it takes to get the kernel up and running after a new kernel release is made.

As always, all the latest information on *netfilter* and *iptables* can be found at **http://netfilter.org/**. There are also some excellent mailing lists there covering *netfilter* and *iptables* configuration, so anyone wanting to seriously use *iptables* should subscribe. ■

packets according to a dynamic profile implemented by means of a simple Fuzzy Logic Controller (FLC) .
Suppported options are:
--upper-limit => Desired upper bound for traffic rate
--lower-limit => Lower bound over which the FLC starts to limit traffic

iplimit

This adds CONFIG_IP_NF_MATCH_IPLIMIT match allows you to restrict the number of parallel TCP connections to a server per client IP address (or address block).
Examples:

```
# allow 2 telnet connections per client host
iptables -p tcp --syn --dport 23 -m iplimit --
iplimit-above 2 -j REJECT
# you can also match the other way around:
iptables -p tcp --syn --dport 23 -m iplimit ! --
iplimit-above 2 -j ACCEPT
# limit the nr of parallel http requests to 16 per
class C sized
# network (24 bit netmask)
iptables -p tcp --syn --dport 80 -m iplimit --
iplimit-above 16                    \
  --iplimit-mask 24 -j REJECT
```

NETMAP

This adds CONFIG_IP_NF_TARGET_NETMAP option, which provides a target for the nat table. It creates a static 1:1 mapping of the network address, while keeping host addresses intact. It can be applied to the PREROUTING chain to alter the destination of incoming connections, to the POSTROUTING chain to alter the source of outgoing connections, or both (with separate rules).
Examples:

```
1.2.3.0/24 -j NETMAP --to 5.6.7.0/24 -j
NETMAP --to 5.6.7.0/24
iptables -t nat -A POSTROUTING -s 5.6.7.0/24
-j NETMAP --to 1.2.3.0/24
```

mport

This module is an enhanced multiport match. It has support for byte ranges as well as for single ports.
Examples:

```
# iptables -A FORWARD -p tcp -m mport --ports
23:42,65
```

Up to 15 ports are allowed. Note that a portrange uses up 2 port values.

quota

This option adds CONFIG_IP_NF_MATCH_QUOTA, which implements network quotas by decrementing a byte counter with each packet. Supported options are:

```
-quota
```

The quota in bytes.

ROUTE (Development/Works for me)

This option adds a `ROUTE' target, which enables you to setup unusual routes not supported by the standard kernel routing table.

For example, the ROUTE lets you directly route a received packet through an interface or towards a host, even if the regular destination of the packet is the router itself. The ROUTE target is also able to change the incoming interface of a packet. This target does never modify the packet and is a final target. It has to be used inside the mangle table.

ROUTE target options:

```
--oif   ifname
```

Send the packet out using the `**ifname'** network interface.

```
--iif   ifname
```

Change the packet's incoming interface to `**ifname'**.

```
--gw    ip
```

Route the packet via this gateway.

SAME

This adds CONFIG_IP_NF_TARGET_SAME option, which is similar to SNAT: it takes a range of addresses (`--to 1.2.3.4-1.2.3.7') and gives a client the same address for each connection. It has a **--nodst** option to make it not use the destination-ip in the calculations when selecting the new source-ip.