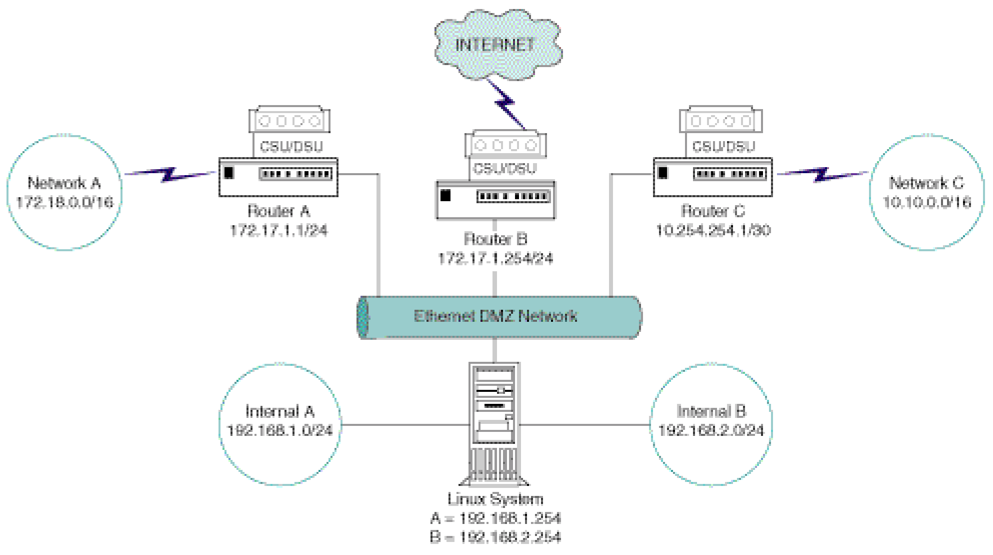
Policy Routing in Linux

Matthew G. Marsh



The classic TCP/IP routing algorithms used today make their routing decisions based only on the destination address of IP packets. However, we often find ourselves wanting to route IP packets depending not only on the destination addresses, but also on other packet fields such as the source address, the IP protocol, the transport protocol ports, or even data within the packet payload. This type of IP routing is referred to as “policy routing”.

Within Linux, as of the Kernel 2.1 series and above, this problem of providing policy routing is solved by replacing the conventional destination-based routing table with the “routing policy database”, or RPDB, which selects the appropriate IP route by executing a set of rules. These rules may contain many keys of various types and therefore, they can have no “natural” ordering. Any ordering or precedence must be imposed by the network or systems administrator.

The RPDB within Linux is currently implemented as a linear list of rules ordered by a numeric priority value. The RPDB itself can explicitly match packet source address, packet destination address, TOS, incoming interface (which is packet metadata, rather than a packet field), and fwmark values. Each routing policy rule consists of a selector and an action. The RPDB is scanned in order of increasing priority with the selector of each rule applied to the source address, destination address, incoming interface, TOS, and fwmark. If the packet matches, then the action is performed. If the action returns success, then the rule output will provide either a valid route or a route lookup failure indication, and RPDB lookup is then terminated. Otherwise, the RPDB lookup continues on to the next rule.

What action should be performed when the selector matches our packet? The standard action, as performed by router software (such as Cisco IOS), is to select the next hop address and the output device. I will refer to this action as a "match & set" style of action. However, Linux takes a much more flexible approach. In Linux, there are several actions to choose from. The default action performs a route lookup from a specified destination-based routing table. The match & set action then becomes the simplest case of Linux route selection, which is realized when the specified destination-based routing table contains only a single default route. Linux supports multiple routing tables, containing multiple standard destination routes. Bear in mind that each of these routing tables is the same as the entire routing table for any other OS. Linux effectively provides 255 Ciscos to choose from. (For number freaks, Linux 2.2.12 supports 255 routing tables, 255 aggregate realms, and 232 (4294967296 decimal) policy rule priorities.

Look at some defaults for Linux 2.1/2.2. At startup, the Linux kernel configures a default RPDB consisting of three policy rules. One way of looking at these default rules is to issue the command that lists all rules on a system that has just been started:

```
root@netmonster# ip rule list
0:      from all lookup local
32766:  from all lookup main
32767:  from all lookup default
```

The following default rules are very important to remember when you start considering complicated systems.

First, is the highest priority rule, Rule Priority 0.

Rule 0: Priority 0 Selector = match every packet

Action = lookup routing table local (ID 255).

Table local is the reserved routing table containing high priority control routes for local and broadcast addresses. Rule 0 is special and cannot be deleted or changed.

Rule 32766: Priority 32766 Selector = match every packet

Action = lookup routing table main (ID 254)

The table main is the default standard routing table containing all non-policy routes. Table main is where routes created with the old route command are placed. Additionally, any routes created by ip route that do not specify an explicit table are placed into this table. This rule may be deleted or overridden by other rules.

Rule 32767: Priority 32767 Selector = match every packet

Action = lookup routing table default (ID 253).

Table default is empty and reserved for post-processing if previous default rules did not select the packet. This rule also may be deleted.

Do not mix routing tables and rules. Rules point to routing tables. You may have several rules referring to one routing table, and some routing tables may not have rules referring to them. If you delete all the rules referring to a table, then the table is not used but the table still exists. A routing table will disappear only after all the routes contained within it are deleted.

As I mentioned previously, Linux policy rules can perform several actions besides pointing to a routing table. When creating a policy rule, you can set the following types of action:

unicast — Standard route lookup within the table referenced by the rule. This is the default action when a table is specified.

blackhole — Rule action drops the packet without any messages.

unreachable — Rule action generates a “Network is unreachable” error. An ICMP Type 3 Code 0 packet is returned to the sender.

prohibit — Rule action generates a “Communication is administratively prohibited” error. ICMP Type 3 Code 13 returned to sender.

Other action types can be used, but none of them are relevant to policy routing. They are used for other advanced traffic control and packet manipulation within the Linux kernel. Because there is only one utility, ip, all of these types are available but we will only use the ones listed above. You can take several actions besides returning a route.

Before I illustrate examples, I’ll run through the relevant ip utility command syntax. The lp utility itself can be used for many functions in addition to the ones I address here. Note that all of these commands are run by typing them at the root prompt on your Linux system:

First, consider the ip addressing:

```
root@netmonster# ip addr help
Usage: ip addr {add|del} IFADDR dev STRING
       ip addr {show|flush} [ dev STRING ] [ scope SCOPE-ID ]
                            [ to PREFIX ] [ FLAG-LIST ] [ label PATTERN
]
IFADDR := PREFIX | ADDR peer PREFIX
          [ broadcast ADDR ] [ anycast ADDR ]
          [ label STRING ] [ scope SCOPE-ID ]
SCOPE-ID := [ host | link | global | NUMBER ]
FLAG-LIST := [ FLAG-LIST ] FLAG
FLAG  := [ permanent | dynamic | secondary | primary |
           tentative | deprecated ]

Example - ip addr add 192.168.1.1/24 dev eth0
```

This will add the TCP/IP address 192.168.2.2 with netmask 255.255.255.0 to the eth0 network card.

Second, consider the ip route command:

```
root@netmonster# ip route help

Usage: ip route { list | flush } SELECTOR
```

```
       ip route get ADDRESS [ from ADDRESS iif STRING ]
                              [ oif STRING ]  [ tos TOS ]
       ip route { add | del | replace | change | append | replace | \
                  monitor} ROUTE
SELECTOR := [ root PREFIX ] [ match PREFIX ] [ exact PREFIX ]
            [ table TABLE_ID ] [ proto RTPROTO ]
            [ type TYPE ] [ scope SCOPE ]
ROUTE := NODE_SPEC [ INFO_SPEC ]
NODE_SPEC := [ TYPE ] PREFIX [ tos TOS ]
             [ table TABLE_ID ] [ proto RTPROTO ]
             [ scope SCOPE ] [ metric METRIC ]
INFO_SPEC := NH OPTIONS FLAGS [ nexthop NH ]...
NH := [ via ADDRESS ] [ dev STRING ] [ weight NUMBER ] NHFLAGS
OPTIONS := FLAGS [ mtu NUMBER ] [ advmss NUMBER ]
           [ rtt NUMBER ] [ rttvar NUMBER ]
           [ window NUMBER] [ cwnd NUMBER ] [ ssthresh REALM ]
           [ realms REALM ]
TYPE := [ unicast | local | broadcast | multicast | throw |
          unreachable | prohibit | blackhole | nat ]
TABLE_ID := [ local | main | default | all | NUMBER ]
SCOPE := [ host | link | global | NUMBER ]
FLAGS := [ equalize ]
NHFLAGS := [ onlink | pervasive ]
RTPROTO := [ kernel | boot | static | NUMBER ]

Example - ip route add 192.168.2.0/24 via 192.168.1.254
```

This will add a route to network 192.168.2.0 netmask 255.255.255.0 through the router at 192.168.1.254.

Finally, consider the ip rule command:

```
root@netmonster# ip rule help
Usage: ip rule [ list | add | del ] SELECTOR ACTION
SELECTOR := [ from PREFIX ] [ to PREFIX ] [ tos TOS ] [ fwmark FWMARK ]
            [ dev STRING ] [ pref NUMBER ]
ACTION := [ table TABLE_ID ] [ nat ADDRESS ]
          [ prohibit | reject | unreachable ]
          [ realms [SRCREALM/]DSTREALM ]
TABLE_ID := [ local | main | default | NUMBER ]

Example - ip rule add from 192.168.2.0/24 prio 32777 reject
```

This will drop all packets coming into the machine from any address in the 192.168.2.0 netmask 255.255.255.0 network.

Now that I’ve covered the command syntax, here are some examples that show what all this verbiage means.

Example 1: Denying All Access to the Internet

Suppose there is a Linux firewall connecting you to the Internet. You wish to prevent any access to the Internet to an entire subnet within your network. While this action can be performed within the Linux packet filtering firewall, I will illustrate an equivalent method. First, look at the setup of this network.

```
Internal network     192.168.0.0/16
Denied Subnet        192.168.2.0/24

Current Routing Table Main (Table 254):

root@netmonster# ip route list table 254

default via 192.168.254.254 dev eth0 proto static
```

Now, I will create a policy rule for this special subnet. From the root command prompt, type:

```
ip rule add from 192.168.2.0/24 priority 5000 prohibit
```

At this point, anyone who comes into this machine from any IP address within the 192.168.2.0/24 network will be sent an ICMP Type 3 Code 13 packet, and the packets will be dropped.

Note that after you run any of these commands, you will want to also issue an ip route flush cache command to flush out the routing cache. Otherwise, you can enter one of these examples but it may not take effect for a period of time depending on the load and size of the ip routing structure. This is especially true when you delete a rule or route.

If I were doing the above example, I would type in all of the following command lines as root.

```
ip rule del priority 5000
ip rule add from 192.168.2.0/24 priority 5000 \
  prohibit
ip route flush cache
```

This ensures that I do not have a rule 5000 by attempting to delete it. If the rule does not exist, I will get a harmless error message. I then install a rule 5000 and reset the RPDB by flushing the runtime cache. If you do not issue the last command to reset the RPDB, your rule will still take effect, but it may be delayed for several minutes.

Multiple Route Tables and IP Addresses

To fully understand the use of policy-based routing, you need to be able to use the multiple routing tables and IP address assignments available in Linux. There are several facets to this, and I will illustrate them with examples.

When you obtain the ip utility, you may notice that there is a subdirectory within the distribution called etc, and within this directory is a subdirectory called iproute2. Copy this subdirectory to your /etc directory or create a /etc/iproute2 directory. This directory contains files that allow you to name your tables and other facets of the policy routing structure. Use or create the rt_tables file in this directory. The sample file already reserves some numbers and provides an example name for table 1.

I will first edit this file and create several tables for use in these examples. So, open the /etc/iproute2/rt_tables file in the editor you prefer and make it look like the following:

```
# reserved values
#
255     local
254     main
253     default
0       unspec
#
# local
#
1       goodnet1
2       goodnet2
3       badnet1
4       badnet2
5       internet
```

Note that you now can use either these names or the table numbers when specifying which routing table you want. For example, the following two commands will both look at the same routing table, assuming you have performed the editing above:

```
ip route list table 1
ip route list table goodnet1
```

It is much easier to understand which table you are referencing with the table name.

Example 2: Create Multiple Routing Tables

When you have entered the rt_table information above, and saved the file, take a look at the contents of the various tables with the following command:

```
ip route list table <name>
```

Note that if you have not edited the rt_table file, you can still refer to all 255 routing tables by number because all 255 routing tables exist, even if they have no data inside them. In the example above, I edited the rt_tables file so that goodnet1 refers to table 1.

Since I entered ip route list table goodnet1, the output will be a new command line. There is nothing in the table to list. Now, populate each of the tables with a route. I will use the local interface address 192.168.1.1/24 in these examples. Note that I could also list this table by using ip route list table 1:

```
ip route add 10.10.10.0/24 via 192.168.1.2 table goodnet1
ip route add 10.10.11.0/24 via 192.168.1.2 table goodnet2
ip route add 10.10.12.0/24 via 192.168.1.2 table badnet1
ip route add 10.10.13.0/24 via 192.168.1.2 table badnet1
ip route add default via 192.168.1.254 table internet
```

By using the ip route list table <name> command, you can see these routes in their tables.

Example 3: Create Multiple IP Addresses

Note that this is not IP aliasing! IP aliasing using the ":#" system is deprecated in Linux 2.1 and higher. This is the new way to use multiple IP addresses.

The eth0 outgoing interface should respond to three different IP addresses. Two of these addresses could be considered as belonging to the same subnet, but should be entered independently. This also illustrates the method of disabling the auto-route creation in Linux 2.2 and higher. Remember that in Linux 2.2 and higher, when you enter an ip address for an interface, the kernel will automatically install a route to the appropriate network. If you do not want this behavior, then you must add the address as a host address. Because you will be entering two addresses that belong to the same subnet, you don't want the kernel to create the routes. That would lead to conflicts. Instead, specify the addresses with full host masks and then enter the necessary routes manually.

Assign the following IP addresses to the interface:

```
192.168.1.1
192.168.1.128
192.168.3.1
```

Deactivate the auto-route creation for both addresses in the 192.168.1.0/24 range, and allow auto-route creation for 192.168.3.0/24.

```
ip addr add 192.168.1.1/32 dev eth0
ip addr add 192.168.1.128/32 dev eth0
ip addr add 192.168.3.1/24 dev eth0
```

Look at the main routing table with ip route list table main. Note that there is a route for 192.168.3.0/24, but not for 192.168.1.0/24, because you didn't allow the kernel to auto-create the routes. I will return to creating the route later, and show how to look at the addresses.

Typing in ip addr by itself on the command line will list the addresses. This will provide a listing of all the addresses assigned to all of the interfaces.

```
root@netmonster# ip addr
1: lo: <LOOPBACK,UP> mtu 3924 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 brd 127.255.255.255 scope host lo
2: eth0: <BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast qlen 100
    link/ether 00:00:49:61:32:bc brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.1/32 scope global eth0
    inet 192.168.1.128/32 scope global eth0
    inet 192.168.3.1/24 scope global eth0
```

Now that I’ve covered the basics, I’ll move on to a serious example.

Example 4: Multiple Route Tables and IP Addresses

Arguably the most powerful feature in the Linux kernel routing code is the use of multiple addresses and routing tables combined with policy-based routing. The following text shows an example of a system acting as a router for three disparate networks.

Look at the diagram of the network under consideration in Figure 1. Note the three external networks attached to the external Ethernet interface. Each of these networks has its own router and its own IP address space that you need to use. Note that two of these address spaces overlap, thus adding a degree of complexity. Set up the routing tables to allow the following connectivity:

• All traffic from any internal network may go to the Internet.

• Traffic from Internal B may go to Network A.

• Traffic from Internal A may go to Network C.

• Traffic from Internal A Hosts 33-62 may go to Network A.

• Traffic from Internal B Hosts 65-78 may go to Network C.

First, set up the external IP addressing — two addresses on the DMZ ethernet interface, eth0:

```
ip addr add 10.254.254.2/30 dev eth0
ip addr add 172.17.1.128/24 dev eth0
```

Next, I will cover which route tables to create. One of the best ways to look at this is to consider that policy routing enables you to determine which routing table to use for source addresses. The rules should enable you to segment the internal networks. Then you can set up normal destination-based routes within the tables. I will use the new route tables that were created previously.

When you set up the routes within tables, the following approach can help clarify the steps. Imagine that you are configuring a router with only two interfaces — the outgoing interface attaches to any outbound router and the incoming interface already has the packets you want to route. It is simply a matter of setting up the routing that you want within that scope. To illustrate, run through the set up for table 1, goodnet1.

```
ip route add 10.10.0.0/16 via 10.254.254.2 table goodnet1 proto static
ip route add default via 172.17.1.254 table goodnet1 proto static
```

What is the command sequence for table goodnet2?

```
ip route add 172.18.0.0/16 via 172.17.1.1 table goodnet2 proto static
ip route add default via 172.17.1.254 table goodnet2 proto static
```

Notice that there are only two tables with three destinations. I have duplicated the destination default for the Internet into both tables. Why not place this into a third table that will be referred to? Consider the interaction between the rules and the tables, and remember that the rules define the policy-based routing structure. Multiple rules can point to the same table. However, once you are in a table, you need to either obtain a route or be returned via a throw route to the rule list. If you have matched a rule, you want to assume that you have a correct match of the policy. Thus, all possible routes for that packet should be present in the routing table where the packet is sent.

In the case where there are three tables, then you need additional rules that look at the destination of the packet. But, looking at the packet destination is the function of a standard route. Why have rules for every possible combination of source and destination? By using the table, you can create a few rules that will serve your purpose. Of course, the flexibility of the system allows many ways to perform this routing. You should work through all of the scenarios yourself and decide what works best.

```
ip rule add from 192.168.1.32/27 to 172.18.0.0/16 pref 15000 table
goodnet1
ip rule add from 192.168.2.64/28 to 10.10.0.0/16 pref 15001 table
goodnet2
ip rule add from 192.168.1.0/24 pref 15002 table goodnet2
ip rule add from 192.168.2.0/24 pref 15003 table goodnet1
```

Note that I have used the preference settings to run the rules from detailed to most general.

Now consider what will happen to a packet incoming from an internal network. First, it will be passed through the rule priority 0 that will pass it on. It then hits rule priority 15000. If it matches, it will be routed according to table goodnet1. If not, it runs through rule 15001, then 15002, and finally 15003. Will such a packet ever continue beyond rule priority 15003?

Now, I'm going to confuse the issue by setting up the exact same routing scenario from a different angle, just to illustrate the range of flexibility available when specifying the routing structure. Here are some details about the Linux server:

```
eth0 - DMZ ethernet - addresses: 10.254.254.2/30, 172.17.1.128/24
eth1 - Internal A - addresses: 192.168.1.254/24
eth2 - Internal B - addresses: 192.168.2.254/24
```

I will run through the route and rule creation, assuming we are starting from the beginning. First, edit /etc/iproute2/rt_tables:

```
# reserved values
#
255     local
254     main
253     default
0       unspec
#
# Local Tables
#
1       int1
2       int2
```

Create the routes and rules:

```
ip route add 10.10.0.0/16 via 10.254.254.1 table int1 proto static
ip route add throw 0/0 table int1 proto static
ip route add 172.18.0.0/16 via 172.17.1.1 table int2 proto static
ip route add throw 0/0 table int2 proto static
ip route add 0/0 via 172.17.1.254 table main proto static
ip rule add pref 15000 table int1 iif eth1
ip rule add pref 15001 table int2 iif eth2
ip rule add pref 15002 to 10.10.0.0/16 table int1
ip rule add pref 15003 to 172.18.0.0/16 table int2
```

This set of routes and rules will perform the same operations as the previous set. Study them until you see why. Hint: Do not forget about the three default rules.

Advanced Policy Routing with ipchains

One of the options available in specifying policy rules is to have a rule match on a fwmark value. fwmark is a numeric tag that can be attached to a packet with the packet filter tool ipchains. If you are not familiar with Linux 2.2 packet filtering, I recommend that you read about ipchains before trying these examples. There is an excellent ipchains HOWTO available through your local Linux Documentation Project mirror site. There is one at: http://ldp.pakuni.net, or the master page can be found at: http://www.linuxdoc.org.

Example 5: Simple fwmark Policy Routing

I will Start with a simple example — using the multiple tables network above, I want to send all traffic from Internal_B destined for tcp port 80 out to the Internet. But, all traffic from Internal_A destined for tcp port 80 should be administratively prohibited. I will assume that we have blank routing tables. Here are the commands to flush out those routing tables:

```
ip route flush table goodnet1

ip route flush table goodnet2

ip route flush table badnet1

ip route flush table badnet2

ip route flush table internet

ip route flush cache
```

Unfortunately, the only current method of deleting policy rules is to list them out and manually delete them. To do this, first list them with ip rule list and then delete the ones you do not want with ip rule del priority <#>. However you do it, I will assume there is nothing in the tables or rules.

To use the fwmark facility, you have to specify the packets you wish to mark using ipchains. Then, use the mark value to specify a policy rule to deal with those packets.

Be forewarned that ipchains automatically converts decimal number marks to hex. ip rule expects a hex value without markers. I will illustrate this, but be careful when specifying fwmarks.

First, set up your ipchains rules to mark incoming packets with the appropriate values. Assume that there are no other firewall rules:

```
ipchains -I input -p tcp -s 192.168.2.0/24 -d 0/0 80 -m 2
ipchains -I input -p tcp -s 192.168.1.0/24 -d 0/0 80 -m 16
```

Now, set up the policy rules. Note that the mark value for Internal_A is decimal 16. Look carefully at the related policy rule to understand the warning:

```
ip rule add fwmark 2  table goodnet1
ip rule add fwmark 10 prohibit
```

And finally, the route for table goodnet1 is as follows:

```
ip route add default via 172.17.1.254 table goodnet1
```

One of the main questions about policy routing concerns the interactions between the policy routes and IP masquerading. While I do not have the space here to cover this in detail, I can offer a tip and quick example. Always remember that the routing tables are consulted just before the forward chain. This means that any source address returned by the route selector will be used as the IP masq address, if you use IP masquerading.

Example 6: Multiple IP Address IP Masquerade

Using the network figure above, I will IP masquerade the connections to the three networks. I want the following outputs from the system:

1. From Internal A to Network C Masq as 10.254.254.2

2. From Internal B to Network A Masq as 172.17.1.2

3. From either Internal to Internet Masq as 172.17.1.128

The addresses on eth0 are:

```
10.254.254.2/30
172.17.1.128/24
```

So, I add the address needed to satisfy condition 2 with:

```
ip addr add 172.17.1.2/32 dev eth0
```

Assume that the system is set up to IP masquerade all outgoing packets. You need only to set up the routes and rules. First, flush the tables and old rules, then set up the following new ones:

```
ip route add 10.10.0.0/16 via 10.254.254.2 src 10.254.254.2 \
    table goodnet1 proto static
ip route add default via 172.17.1.254 src 172.17.1.128  \
    table goodnet1 proto static
ip route add 172.18.0.0/16 via 172.17.1.1 src 172.17.1.2 \
    table goodnet2 proto static
ip route add default via 172.17.1.254 src 172.17.1.128 \
    table goodnet2 proto static
ip rule add from 192.168.1.0/24 pref 15000 table goodnet2
ip rule add from 192.168.2.0/24 pref 15001 table goodnet1
```

Then you are done.

For the grand finale example, I will take this previous example and add in fwmarking and policy blackholes:

Example 7: All Together Now

Assuming that the routes, rules, and addresses from Example 6 are all still in force, I want to allow for the following set of functions:

```
•  Internal A Hosts 33-62 to Network A Masq as 172.17.1.3
•  Internal A Hosts 65-78 to tcp port 80 on Network A Masq as
172.17.1.4
•  Internal B Hosts 33-62 to tcp port 80 on Network A Deny Access
•  Internal B Hosts 65-78 to tcp port 80 on Network C Masq as
10.254.254.2
```

Keep in mind that we are still allowing the connectivity from Example 6. Here is the solution:

```
ip addr add 172.17.1.3/32 dev eth0
ip addr add 172.17.1.4/32 dev eth0
ip route del default table goodnet1
ip route del default table goodnet2
ip route add throw 0/0 table goodnet1 proto static
ip route add throw 0/0 table goodnet2 proto static
ip route add default via 172.17.1.254 src 172.17.1.128 \
    table internet proto static
ip route add 172.18.0.0/16 via 172.17.1.1 src 172.17.1.3 \
    table badnet1 proto static
ip route add 172.18.0.0/16 via 172.17.1.1 src 172.17.1.4 \
    table badnet2 proto static
ip rule add from 192.168.1.32/27 to 172.18.0.0/16 pref 14999 table
badnet1
ip rule add fwmark 1 pref 14998 table badnet2
ip rule add fwmark 2 pref 14997 table goodnet1
ip rule add fwmark 3 pref 14996 blackhole
ip rule add pref 15003 table internet
ipchains -I input -p tcp -s 192.168.1.64/28 -d 172.18.0.0/16 80 -m 1
ipchains -I input -p tcp -s 192.168.2.64/28 -d 10.10.0.0/16 80 -m 2
ipchains -I input -p tcp -s 192.168.2.32/27 -d 172.18.0.0/16 80 -m 3
```

There are many valid methods. Note that while I used the tables badnet1 and badnet2, the names are meaningless as I merely used those names to refer to tables 3 and 4.

Summary

I hope you enjoyed this romp through the world of policy routing in Linux 2.2. The current state of affairs in Linux IP networking provides a very powerful system that is very rarely matched in capability, especially when considering the price/performance ratio. In other words, the capabilities of Example 7 above could easily be done on a 486/33 with 16 megs of memory.

For more information about many of the topics mentioned here, check out the various HOWTOs. Currently, there is an ipchains HOWTO as part of the LDP. The Policy Routing HOWTO was in alpha as of this writing, but should be available by the time you read this, at http://www.linuxgrill.com. Alexey Kuznetsov’s IPROUTE2 and other utilities are also available at LinuxGrill. The reference for the ip utility is contained in the documentation directory of the IPROUTE2 source directory.

Postscript on Netfilter and Kernel 2.4

Netfilter is the successor to ipchains. As of netfilter 0.1.9 on kernel 2.3.18, there are no facilities for providing fwmark functions. However, the backward compatibility feature of netfilter will allow you to use ipchains fwmark commands. By the time you read this, I hope there will be a utility to provide native fwmark capabilities under netfilter, because the native capabilities of netfilter in all other areas of Linux networking are superb. n

About the Author

Matthew Marsh has been working with Linux since 1993. He is the author of “Linux Networking and Security Unleashed” from SAMS, cofounder of the NECERT, and is President of Paktronix Systems LLC, an Omaha, Nebraska based security consulting firm. He remembers when he had to specify all the mail system gateways in order to send email from BITNET to his friend at Brown. Currently, when not grokking Linux, he is addicted to Civ:CTP.