Kernel Korner

A NATural Progression

David continues his series on the Netfilter framework with a look at NAT and how to avoid common errors when constructing iptables rules.

by David A. Bandel

Editor's Note: Due to a printer error, David Bandel's article on iptables building was not complete in the magazine. We present it here in its entirety.

One of the best tools at our disposal within the Netfilter framework is NAT. NAT allows us to obfuscate (but not hide) our true network, forcing would-be black hats to work harder (and possibly go after easier targets). It also permits us to make the best use of limited IPs. Last month [see "Netfilter 2: in the POM of Your Hands" in the May 2002 issue of LJ] we looked at extending iptables to include experimental or beta matches and targets. This month we look at doing NAT the correct way, take a closer look at one or two other matches, then see what some of the more common errors are when constructing iptables rules and how to avoid them.

SNAT, DNAT and Managing Services

Given the shortage of usable IPv4 addresses left today, it's likely your ISP didn't provide you enough IPs to run all your systems. If you're lucky, you got more than half a dozen you could actually use. But if you don't use them, you'll lose them. And you don't want that, or there's no room for expansion tomorrow. So you're going to make your firewall look like as many systems as you have IPs to use. How to do that? The easiest way is to assign all your IPs to one NIC (the one connected to your ISP) and SNAT connections so they look like they come from each IP in turn (this example assumes your internal network is 192.168.0.0/24, which is bound to eth1, and your usable IPs are 209.127.112.26-209.127.112.30, which are bound to eth0):

```
iptables -t nat -A POSTROUTING
-o eth0 -s
--to-source 209.127.112.26-209.127.112.30
```

Now iptables will NAT the first connection to .26, the second to .27, the third to .28 and so on, wrapping around to .26 after the connection to .30.

One word of caution: test this before you deploy it. I've had one router that didn't like seeing multiple IPs sourced from the same MAC address. It would pass the first connection, but subsequent connections would time out. The router's built-in firewall (which couldn't be turned off by the client) most likely thought the other packets were spoofed and was silently dropping them.

Let's make sure we accept all outgoing connections but only accept incoming connections that are related to these outgoing connections:

```
ipables -t filter -A FORWARD -i ! eth0 -m state
--state NEW,ESTABLISHED,RELATED -j ACCEPT
iptables -t filter -A FORWARD -i eth0 -m state
--state ESTABLISHED,RELATED -j ACCEPT
iptables -t filter -A FORWARD -i eth0 -m state
--state NEW,INVALID -j DROP
```
Now that we've managed outgoing traffic, let's assume we've moved all our services inside this firewall. We'll further assume they are all inside eth1 on the 192.168.0.0/24 network. Each service has two IPs associated with it: an external IP that the world sees and an internal IP that we see. Specifically, we'll assign the following:

- Apache web server (serves both insecure and secure connections): 209.127.112.26 and 192.168.0.4
- FTP server: 209.127.112.27 and 192.168.0.5
- Primary DNS server: 209.127.112.26 and 192.168.0.6
- Secondary DNS server: 209.127.112.28 and 192.168.0.7
- Primary mail server: 209.127.112.28 and 192.168.0.8
- Secondary mail server: 209.127.112.29 and 192.168.0.9

Because of our iptables state table rules above, each service (or more specifically, each port that corresponds to that service) will not only have to be forwarded through the firewall to the correct IP inside, but we'll need a rule to accept that NEW traffic. Starting with Apache, which in our case uses both ports 80 and 443 (for SSL), we have:

```
iptables -t filter -I FORWARD -i eth0 -d
iptables -t nat -A PREROUTING -d -p tcp --dport 80 -j DNAT --to-destination
iptables -t filter -I FORWARD -i eth0 -d 209.127.112.26 -p tcp --dport 443 -j ACCEPT
iptables -t nat -A PREROUTING -d --dport 443 -j DNAT --to-destination 192.168.0.4
```

Notice that we had to insert a rule in the FORWARD chain. This is because we already have a more general rule that would have dropped NEW connections. We can insert a rule anywhere in a chain, but if we don't specify where, the default is to insert it as the first rule. Normally, this will not be a problem and will put our specific rules ahead of our general rules.

Note that we also specified the IP on which this connection should show. This is not necessary because connections showing up on other IPs will be handled by the state table and dropped, unless we've done something really questionable and started a web server on our firewall. If no ports on our firewall are open, we're okay. We always can protect them just in case by ensuring we also have stateful rules for our INPUT chain:

```
iptables -t filter -A INPUT -i ! eth0 -m state --state NEW,RELATED,ESTABLISHED -j ACCEPT
iptables -t filter -A INPUT -i eth0 -m state --state RELATED,ESTABLISHED -j ACCEPT
iptables -t filter -A INPUT -i eth0 -m state --state NEW,INVALID -j DROP
```

The first rule above allows NEW, ESTABLISHED and RELATED connections from lo (the localhost interface) as well as any internal devices, omitting only our external device, which is dealt with in the next two rules.

Next we look at the FTP connection. This is straightforward and exactly the same as the above rules, but for
port 21:

```bash
iptables -t filter -I FORWARD -i eth0 -p tcp --dport 21 -j ACCEPT
iptables -t nat -A PREROUTING -i eth0 -d --to-destination 192.168.0.5
```

We don't have to worry about the FTP-data channel (port 20) because our FTP server opens it outgoing, and our state rules will pass this new connection out.

Now it gets a little more difficult. DNS works on both UDP for normal queries and TCP for zone transfers. If we don't want to allow zone transfers to the outside, we only open UDP. If we want to allow zone transfers, then we have to allow both. Assuming we want to allow both, we know that we can specify UDP, TCP or ICMP as protocols. You must specify `-p` (protocol) in order to specify a port. If you want both UDP and TCP, you should be able to say `not ICMP`, and the other two are assumed automatically. Unfortunately, it doesn't work that way.

When you specify a protocol, even if you say `-p ! ICMP`, it's the ICMP match module that is loaded, not the TCP and UDP match modules. So you'll get an error message when you specify a port. This is a danger with using a negative match; the match module that is loaded is the module specified, not the modules you may assume are loaded. You must specify positively each match you want so the corresponding match module is loaded.

For now, let's assume you are only interested in opening the UDP port:

```bash
iptables -t filter -I FORWARD -i eth0 -d
iptables -t nat -A POSTROUTING -i eth0 -d --to-destination
iptables -t filter -I FORWARD -i eth0 -d
iptables -t nat -A POSTROUTING -i eth0 -d --to-destination 192.168.0.8
```

And finally, we need to deal with our mail host:

```bash
iptables -t filter -I FORWARD -i eth0 -d 209.127.112.28 -i eth0 -p TCP --dport 25 -j ACCEPT
iptables -t nat -A POSTROUTING -i eth0 -d --to-destination
iptables -t filter -I FORWARD -i eth0 -d 209.127.112.29 -i eth0 -p UDP --dport 25 -j ACCEPT
iptables -t nat -A POSTROUTING -i eth0 -d --to-destination 192.168.0.9
```

We can now accept incoming mail. Does anyone see a problem here?

If we test our outgoing mail using `mail -v user@another.dom`, our firewall will grab one of 209.127.112.25-209.127.112.30. If our DNS records say our MX host is 209.127.112.28, then we have only a 20% chance of grabbing that IP and an 80% chance that upstream mail hosts will bounce our mail as not being from a host with a DNS MX RR--not good.
So how do we fix this? If we have the luxury of adding all our IPs as MX hosts, that would solve part of the problem, but then upstream hosts might spend time connecting to IPs of ours that don't DNAT the mail through. And we really don't want all those IPs to appear as MX IPs.

The correct response is to add a more specific SNAT rule ahead of the general SNAT rules that will handle outgoing traffic on port 25. The danger here is that if we can't trust our internal users, we also must make sure that internal users can't abuse port 25. So we'll add three rules for outgoing traffic, one to SNAT port 25 traffic coming from our primary mail server (192.168.0.8) only out through 209.127.112.28, and two to block port 25 traffic from all other internal addresses except our true mail host:

```bash
iptables -t filter -I FORWARD -i eth1 -s
iptables -t filter -I FORWARD -i eth1 -p tcp --dport 25 -s
-I POSTROUTING -o eth0 -p tcp --dport 25 -s 192.168.0.8 -j SNAT --to-source 209.127.112.28
```

Some of you may think: hah! caught him. The first two rules above are reversed. Well, yes, they are. But that's because we're inserting them one at a time as the first rule, so rule two above will really be rule one in the FORWARD chain after running them both. The SNAT rule is in another chain, so it could have gone anywhere. Also, I suggest you make sure the first rule above is correct for your system. If the untrusted network is 192.168.0.0/24, and the trusted network is 192.168.1.0/24, you may need the source (-s) to be 192.168.0.0/23 to cover both. Or, perhaps just drop the -s option and match on the inbound interface (-i).

I suggest that the best way to build a firewall is to walk through each chain to see where and how (or even if) a particular packet will be handled. Don't do as we've been doing here inserting rules seemingly willy-nilly. Build your chain on paper with all the rules in the correct order. Then you won't make mistakes. You can always check afterward to make sure the rules are as you think you wanted them with: `iptables -t <table> -L -nv`. The above rule with the -v included will show you how many packets and how many bytes are affected by this rule. If, after a week has gone by, you still have rules with 0 bytes affected by it, you might want to relook at that rule's position in the chain. But just because a rule has affected packets doesn't mean it's in the right place. It may have only affected half the packets that really should have been affected.

I'm waiting for someone to write ``the killer app'' for Netfilter, and that would be a utility that runs tests, analyzes the rules and allows you to move them around and test again. But until that day comes, you'll have to do it by hand.

Some of you may have noticed that I make heavy use of -i eth0, or -i ! eth0, but in general match an interface. Often, you can probably see that this isn't necessary because I've limited the source IP address or some other part of the packet header that pretty much ensures matching what we want. But I do this for a particular reason. I turn off rp_filters (reverse path filters). These tend to interfere with legitimate VPN packets. Besides, Linux's rp_filter is nowhere near as granular as iptables.

### Protecting Your Data

It's very difficult to make sure someone on the inside isn't passing data through your firewall that shouldn't go out. And maybe some folks have a legitimate reason for passing company data out through the firewall. But let's assume for now that that's not the case. You want to stop certain data from leaving (or at least attempt to do so).

We can try to prevent certain data from leaving by marking that data, then looking for that mark using the string match. Here I suggest a policy of putting a string, such as ```Copyright, foo.corp, not for publication``` at the top of those files you don't want sent through the firewall. Then, on the inner firewall, or as a rule on eth2
(where eth0 is the Internet, eth1 is the untrusted LAN and eth2 is the trusted LAN) on your outer firewall, you might want something like this:

```bash
iptables -t filter -I FORWARD -i eth2 -m string
--string="Copyright, foo.corp, not for publication"
-j DROP
```

A few words about this particular solution. First, ensure you have the ipconntrack module loaded. This will defragment packets and result in a much higher likelihood of seeing the string. Second, don't expect this to catch everything. Particularly, if a file has been compressed, the phrase will not be recognizable as such. So this does have limitations.

It will work very nicely, however, if you are running an IIS server and want to drop packets with the string root.exe, for example. The rule might look like this:

```bash
iptables -t filter -I FORWARD -m string
--string=root.exe -j DROP
```

While it might be amusing to use the MIRROR target and turn the attack back on the attacker, this would be an ethically questionable thing to do.

You also have the PSD (port-scan detection) match if you're still subject to this kind of activity. I don't see so much port scanning anymore as I see script kiddies that have a particular tool; they aim it at my systems and fire. Usually it's an FTP attack designed to compromise an IIS server running FrontPage extensions. I see an FTP in, then lots of activity trying to create _vti_private files and the like. We can stop this with:

```bash
iptables -t filter -I FORWARD -i eth0 -p tcp
--dport 21 -m string
--string="_vti_private" -j REJECT
```

Obviously, if you're running a FrontPage server and folks aren't "publishing" to it (which uses port 80) but moving their sites via FTP, the above won't work.

### Targets and Matches and More, Oh My!

This article has not touched on a large number of extensions and targets. Some of you with very specific routing requirements might want to look at the MARK target, with or without the realm match, to do some really funky routing tricks. This will require use of iproute2 in conjunction with iptables. This is a very powerful combination for ISPs or others with very specific routing and bandwidth-limiting requirements.

Others of you probably wanted to see some ULOG target examples or iplimit or mport examples. But these are very similar to other matches or targets and are handled in the same way. Often the help in the kernel configuration will show you enough of an iptables rule fragment to make use of these extensions.

Just remember, only ACCEPT, DROP or REJECT are final targets for a packet and stop iptables processing of a packet. The RETURN target only terminates a chain, but not iptables processing.

I also haven't touched on the MANGLE table. But this table works in the same manner as the mangle target in ipchains. Try it out if you're so inclined. You may find you won't be able to use the numeric (hex) targets but have to use the descriptive values. If you can't remember what they are, try:

```bash
iptables -j TOS -h
```

This trick also works if you need a list of the ICMP types because you want to handle a particular ICMP type with iptables (such as permitting pings, which will be dropped by a firewall with `-m state --state ESTABLISHED,RELATED`):
iptables -p icmp -h

Armed with the correct ICMP name, echo-request, you can permit pings:

```
iptables -t filter -I INPUT -i eth0 -p icmp
--icmp-type echo-request -j ACCEPT
```

You also can rate-limit this using either iptlimit or limit if you're concerned about this. But note that limiting pings doesn't limit the amount of traffic on your link, just the rate at which you'll respond to this traffic. Anyway, standard ping packets are so small and normally sent only once a second by any given host that they're barely noticeable as traffic.

**Errors**

Some common errors I've seen with iptables scripts include choosing an inappropriate interface for packets. This includes not selecting all interfaces that might be affected. Often lo, the localhost interface, is forgotten about on systems used as both firewall and host (usually a system used in a home). I've also seen outgoing packets using the MANGLE table, the OUTPUT chain or the SNAT target that have `-i <interface>` rather than `-o <interface>`.

Sometimes rules get so specific nothing matches them. Try the most general rule you can get away with, adding match extensions only as required. Just be careful where these rules are located in relation to other rules, so they're not picking up packets you don't want them to.

Ensure you're using the right case: ipchains uses lowercase for its built-in chains, but iptables uses uppercase. Targets are also uppercase. Almost everything else is lowercase. If you're using the short options (as I did in this article), the chain action (Insert, Append, Delete, etc.) uses uppercase.

**Conclusion**

Netfilter and iptables make an extremely powerful firewall. But to take advantage of it, you need to master the basic syntax as explained in my first article (``Taming the Wild Netfilter'', published in the September 2001 issue of *LJ*), have an understanding of the modules and matches available to you and have an understanding of what a particular system can know about a packet. Armed with these three things, you can build highly complex, tailored firewall solutions for whatever problem you might have.

Know how to take advantage of new and experimental matches and targets (and always test them). You learned this in last month's article with the iptables build targets of pending-patches, most-of-pom and patch-o-matic. Testing by creating and sending specific packets to a firewall interface is beyond the scope of this particular article, but a number of utilities exist to assist you here (sendip or ipmagic come to mind).

Build the missing modules for the kernel (make sure they're selected).

Build your rule chains, more specific rules first followed by more general rules. If it helps you organize things, go ahead and build custom user chains that can be called from another chain. While this was not covered specifically in this article, it was addressed in the September 2001 article. Use everything at your disposal, including the LOG target to help you see if particular rules were applied to which packets.

With just some basic knowledge, iptables are not difficult to use. Read some iptables scripts on the Internet. I don't recommend using them as they are; they almost certainly won't work for you without a lot of tweaking, but they will show you syntax, rules (from which you can grab fragments), thought processes, etc.

David A. Bandel ([david@pananix.com](mailto:david@pananix.com)) is a Linux/UNIX consultant currently living in
the Republic of Panama. He is coauthor of Que Special Edition: Using Caldera OpenLinux.