Linux 2.4 NAT HOWTO

Rusty Russell, mailing list netfilter@lists.samba.org $Revision: 487 $ $Date: 2002-01-14 10:35:13 +0100 (Mon, 14 Jan 2002) $

This document describes how to do masquerading, transparent proxying, port forwarding, and other forms of Network Address Translations with the 2.4 Linux Kernels.

Contents

1 Introduction

Welcome, gentle reader.

You are about to delve into the fascinating (and sometimes horrid) world of NAT: Network Address Translation, and this HOWTO is going to be your somewhat accurate guide to the 2.4 Linux Kernel and beyond.

In Linux 2.4, an infrastructure for mangling packets was introduced, called 'netfilter'. A layer on top of this provides NAT, completely reimplemented from previous kernels.

2 Where is the official Web Site and List?

There are three official sites:

- Thanks to *Filewatcher* `http://netfilter.filewatcher.org/` .
- Thanks to *The Samba Team and SGI* `http://netfilter.samba.org/` .
- Thanks to *Harald Welte* `http://netfilter.gnumonks.org/` .

You can reach all of them using round-robin DNS via

`http://www.netfilter.org/` and `http://www.iptables.org/`

For the official netfilter mailing list, see

netfilter List `http://www.netfilter.org/contact.html#list` .

2.1 What is Network Address Translation?

Normally, packets on a network travel from their source (such as your home computer) to their destination (such as www.gnumonks.org) through many different links: about 19 from where I am in Australia. None of these links really alter your packet: they just send it onward.

If one of these links were to do NAT, then they would alter the source or destinations of the packet as it passes through. As you can imagine, this is not how the system was designed to work, and hence NAT is always something of a crock. Usually the link doing NAT will remember how it mangled a packet, and when a reply packet passes through the other way, it will do the reverse mangling on that reply packet, so everything works.

2.2 Why Would I Want To Do NAT?

In a perfect world, you wouldn't. Meanwhile, the main reasons are:

Modem Connections To The Internet

Most ISPs give you a single IP address when you dial up to them. You can send out packets with any source address you want, but only replies to packets with this source IP address will return to you. If you want to use multiple different machines (such as a home network) to connect to the Internet through this one link, you'll need NAT.

This is by far the most common use of NAT today, commonly known as 'masquerading' in the Linux world. I call this SNAT, because you change the **source** address of the first packet.

Multiple Servers

Sometimes you want to change where packets heading into your network will go. Frequently this is because (as above), you have only one IP address, but you want people to be able to get into the boxes behind the one with the 'real' IP address. If you rewrite the destination of incoming packets, you can manage this. This type of NAT was called port-forwarding under previous versions of Linux.

A common variation of this is load-sharing, where the mapping ranges over a set of machines, fanning packets out to them. If you're doing this on a serious scale, you may want to look at

Linux Virtual Server http://linuxvirtualserver.org/ .

Transparent Proxying

Sometimes you want to pretend that each packet which passes through your Linux box is destined for a program on the Linux box itself. This is used to make transparent proxies: a proxy is a program which stands between your network and the outside world, shuffling communication between the two. The transparent part is because your network won't even know it's talking to a proxy, unless of course, the proxy doesn't work.

Squid can be configured to work this way, and it is called redirection or transparent proxying under previous Linux versions.

3 The Two Types of NAT

I divide NAT into two different types: **Source NAT** (SNAT) and **Destination NAT** (DNAT).

Source NAT is when you alter the source address of the first packet: i.e. you are changing where the connection is coming from. Source NAT is always done post-routing, just before the packet goes out onto the wire. Masquerading is a specialized form of SNAT.

Destination NAT is when you alter the destination address of the first packet: i.e. you are changing where the connection is going to. Destination NAT is always done before routing, when the packet first comes off the wire. Port forwarding, load sharing, and transparent proxying are all forms of DNAT.

4 Quick Translation From 2.0 and 2.2 Kernels

Sorry to those of you still shell-shocked from the 2.0 (ipfwadm) to 2.2 (ipchains) transition. There's good and bad news.

Firstly, you can simply use ipchains and ipfwadm as before. To do this, you need to insmod the 'ipchains.o' or 'ipfwadm.o' kernel modules found in the latest netfilter distribution. These are mutually exclusive (you have been warned), and should not be combined with any other netfilter modules.

Once one of these modules is installed, you can use ipchains and ipfwadm as normal, with the following differences:

- Setting the masquerading timeouts with ipchains -M -S, or ipfwadm -M -s does nothing. Since the timeouts are longer for the new NAT infrastructure, this should not matter.
- The init_seq, delta and previous_delta fields in the verbose masquerade listing are always zero.
- Zeroing and listing the counters at the same time '-Z -L' does not work any more: the counters will not be zeroed.
- The backward compatibility layer doesn't scale very well for large numbers of connections: don't use it for your corporate gateway!

Hackers may also notice:

- You can now bind to ports 61000-65095 even if you're masquerading. The masquerading code used to assume anything in this range was fair game, so programs couldn't use it.
- The (undocumented) 'getsockname' hack, which transparent proxy programs could use to find out the real destinations of connections no longer works.
- The (undocumented) bind-to-foreign-address hack is also not implemented; this was used to complete the illusion of transparent proxying.

4.1 I just want masquerading! Help!

This is what most people want. If you have a dynamically allocated IP PPP dialup (if you don't know, this is you), you simply want to tell your box that all packets coming from your internal network should be made to look like they are coming from the PPP dialup box.

```
# Load the NAT module (this pulls in all the others).
modprobe iptable_nat

# In the NAT table (-t nat), Append a rule (-A) after routing
# (POSTROUTING) for all packets going out ppp0 (-o ppp0) which says to
# MASQUERADE the connection (-j MASQUERADE).
iptables -t nat -A POSTROUTING -o ppp0 -j MASQUERADE

# Turn on IP forwarding
echo 1 > /proc/sys/net/ipv4/ip_forward
```

Note that you are not doing any packet filtering here: for that, see the Packet Filtering HOWTO: 'Mixing NAT and Packet Filtering'.

4.2 What about ipmasqadm?

This is a much more niche user base, so I didn't worry about backward compatibility as much. You can simply use 'iptables -t nat' to do port forwarding. So for example, in Linux 2.2 you might have done:

```
# Linux 2.2
# Forward TCP packets going to port 8080 on 1.2.3.4 to 192.168.1.1's port 80
ipmasqadm portfw -a -P tcp -L 1.2.3.4 8080 -R 192.168.1.1 80
```

Now you would do:

```
# Linux 2.4
# Append a rule before routing (-A PREROUTING) to the NAT table (-t nat) that
# TCP packets (-p tcp) going to 1.2.3.4 (-d 1.2.3.4) port 8080 (--dport 8080)
# have their destination mapped (-j DNAT) to 192.168.1.1, port 80
# (--to 192.168.1.1:80).
iptables -A PREROUTING -t nat -p tcp -d 1.2.3.4 --dport 8080 \
        -j DNAT --to 192.168.1.1:80
```

5 Controlling What To NAT

You need to create NAT rules which tell the kernel what connections to change, and how to change them. To do this, we use the very versatile `iptables` tool, and tell it to alter the NAT table by specifying the '-t nat' option.

The table of NAT rules contains three lists called 'chains': each rule is examined in order until one matches. The two chains are called PREROUTING (for Destination NAT, as packets first come in), and POSTROUTING (for Source NAT, as packets leave). The third (OUTPUT) will be ignored here.

The following diagram would illustrate it quite well if I had any artistic talent:

```
      -----                                   -----
     /     \                                 /     \
   PREROUTING -->[Routing ]----------------->POSTROUTING----->
     \D-NAT/     [Decision]                  \S-NAT/
                     |                            ^
                     |                            |
                     |                            |
                     |                            |
                     |                            |
                     |                            |
                     |                            |
                     --------> Local Process ------
```

At each of the points above, when a packet passes we look up what connection it is associated with. If it's a new connection, we look up the corresponding chain in the NAT table to see what to do with it. The answer it gives will apply to all future packets on that connection.

5.1 Simple Selection using iptables

`iptables` takes a number of standard options as listed below. All the double-dash options can be abbreviated, as long as `iptables` can still tell them apart from the other possible options. If your kernel has iptables support as a module, you'll need to load the ip_tables.o module first: 'insmod ip_tables'.

The most important option here is the table selection option, '-t'. For all NAT operations, you will want to use '-t nat' for the NAT table. The second most important option to use is '-A' to append a new rule at the end of the chain (e.g. '-A POSTROUTING'), or '-I' to insert one at the beginning (e.g. '-I PREROUTING').

You can specify the source ('-s' or '–source') and destination ('-d' or '–destination') of the packets you want to NAT. These options can be followed by a single IP address (e.g. 192.168.1.1), a name (e.g. www.gnumonks.org), or a network address (e.g. 192.168.1.0/24 or 192.168.1.0/255.255.255.0).

You can specify the incoming ('-i' or '–in-interface') or outgoing ('-o' or '–out-interface') interface to match, but which you can specify depends on which chain you are putting the rule into: at PREROUTING you can only select incoming interface, and at POSTROUTING you can only select outgoing interface. If you use the wrong one, `iptables` will give an error.

5.2 Finer Points Of Selecting What Packets To Mangle

I said above that you can specify a source and destination address. If you omit the source address option, then any source address will do. If you omit the destination address option, then any destination address will do.

You can also indicate a specific protocol ('-p' or '–protocol'), such as TCP or UDP; only packets of this protocol will match the rule. The main reason for doing this is that specifying a protocol of tcp or udp then allows extra options: specifically the '–source-port' and '–destination-port' options (abbreviated as '–sport' and '–dport').

These options allow you to specify that only packets with a certain source and destination port will match the rule. This is useful for redirecting web requests (TCP port 80 or 8080) and leaving other packets alone.

These options must follow the '-p' option (which has a side-effect of loading the shared library extension for that protocol). You can use port numbers, or a name from the /etc/services file.

All the different qualities you can select a packet by are detailed in painful detail in the manual page (`man iptables`).

6 Saying How To Mangle The Packets

So now we know how to select the packets we want to mangle. To complete our rule, we need to tell the kernel exactly what we want it to do to the packets.

6.1 Source NAT

You want to do Source NAT; change the source address of connections to something different. This is done in the POSTROUTING chain, just before it is finally sent out; this is an important detail, since it means that anything else on the Linux box itself (routing, packet filtering) will see the packet unchanged. It also means that the '-o' (outgoing interface) option can be used.

Source NAT is specified using '-j SNAT', and the '–to-source' option specifies an IP address, a range of IP addresses, and an optional port or range of ports (for UDP and TCP protocols only).

```
## Change source addresses to 1.2.3.4.
# iptables -t nat -A POSTROUTING -o eth0 -j SNAT --to 1.2.3.4

## Change source addresses to 1.2.3.4, 1.2.3.5 or 1.2.3.6
# iptables -t nat -A POSTROUTING -o eth0 -j SNAT --to 1.2.3.4-1.2.3.6

## Change source addresses to 1.2.3.4, ports 1-1023
# iptables -t nat -A POSTROUTING -p tcp -o eth0 -j SNAT --to 1.2.3.4:1-1023
```

6.1.1 Masquerading

There is a specialized case of Source NAT called masquerading: it should only be used for dynamically-assigned IP addresses, such as standard dialups (for static IP addresses, use SNAT above).

You don't need to put in the source address explicitly with masquerading: it will use the source address of the interface the packet is going out from. But more importantly, if the link goes down, the connections (which are now lost anyway) are forgotten, meaning fewer glitches when connection comes back up with a new IP address.

```
## Masquerade everything out ppp0.
# iptables -t nat -A POSTROUTING -o ppp0 -j MASQUERADE
```

6.2 Destination NAT

This is done in the PREROUTING chain, just as the packet comes in; this means that anything else on the Linux box itself (routing, packet filtering) will see the packet going to its 'real' destination. It also means that the '-i' (incoming interface) option can be used.

Destination NAT is specified using '-j DNAT', and the '–to-destination' option specifies an IP address, a range of IP addresses, and an optional port or range of ports (for UDP and TCP protocols only).

```
## Change destination addresses to 5.6.7.8
# iptables -t nat -A PREROUTING -i eth0 -j DNAT --to 5.6.7.8

## Change destination addresses to 5.6.7.8, 5.6.7.9 or 5.6.7.10.
# iptables -t nat -A PREROUTING -i eth0 -j DNAT --to 5.6.7.8-5.6.7.10

## Change destination addresses of web traffic to 5.6.7.8, port 8080.
# iptables -t nat -A PREROUTING -p tcp --dport 80 -i eth0 \
        -j DNAT --to 5.6.7.8:8080
```

6.2.1 Redirection

There is a specialized case of Destination NAT called redirection: it is a simple convenience which is exactly equivalent to doing DNAT to the address of the incoming interface.

```
## Send incoming port-80 web traffic to our squid (transparent) proxy
# iptables -t nat -A PREROUTING -i eth1 -p tcp --dport 80 \
        -j REDIRECT --to-port 3128
```

Note that squid needs to be configured to know it's a transparent proxy!

6.3 Mappings In Depth

There are some subtleties to NAT which most people will never have to deal with. They are documented here for the curious.

6.3.1 Selection Of Multiple Addresses in a Range

If a range of IP addresses is given, the IP address to use is chosen based on the least currently used IP for connections the machine knows about. This gives primitive load-balancing.

6.3.2 Creating Null NAT Mappings

You can use the '-j ACCEPT' target to let a connection through without any NAT taking place.

6.3.3 Standard NAT Behavior

The default behavior is to alter the connection as little as possible, within the constraints of the rule given by the user. This means we won't remap ports unless we have to.

6.3.4 Implicit Source Port Mapping

Even when no NAT is requested for a connection, source port translation may occur implicitly, if another connection has been mapped over the new one. Consider the case of masquerading, which is rather common:

1. A web connection is established by a box 192.1.1.1 from port 1024 to www.netscape.com port 80.
2. This is masqueraded by the masquerading box to use its source IP address (1.2.3.4).
3. The masquerading box tries to make a web connection to www.netscape.com port 80 from 1.2.3.4 (its external interface address) port 1024.
4. The NAT code will alter the source port of the second connection to 1025, so that the two don't clash.

When this implicit source mapping occurs, ports are divided into three classes:

- Ports below 512
- Ports between 512 and 1023
- Ports 1024 and above.

A port will never be implicitly mapped into a different class.

6.3.5 What Happens When NAT Fails

If there is no way to uniquely map a connection as the user requests, it will be dropped. This also applies to packets which could not be classified as part of any connection, because they are malformed, or the box is out of memory, etc.

6.3.6 Multiple Mappings, Overlap and Clashes

You can have NAT rules which map packets onto the same range; the NAT code is clever enough to avoid clashes. Hence having two rules which map the source address 192.168.1.1 and 192.168.1.2 respectively onto 1.2.3.4 is fine.

Furthermore, you can map over real, used IP addresses, as long as those addresses pass through the mapping box as well. So if you have an assigned network (1.2.3.0/24), but have one internal network using those addresses and one using the Private Internet Addresses 192.168.1.0/24, you can simply NAT the 192.168.1.0/24 source addresses onto the 1.2.3.0 network, without fear of clashing:

```
# iptables -t nat -A POSTROUTING -s 192.168.1.0/24 -o eth1 \
        -j SNAT --to 1.2.3.0/24
```

The same logic applies to addresses used by the NAT box itself: this is how masquerading works (by sharing the interface address between masqueraded packets and 'real' packets coming from the box itself).

Moreover, you can map the same packets onto many different targets, and they will be shared. For example, if you don't want to map anything over 1.2.3.5, you could do:

```
# iptables -t nat -A POSTROUTING -s 192.168.1.0/24 -o eth1 \
        -j SNAT --to 1.2.3.0-1.2.3.4 --to 1.2.3.6-1.2.3.254
```

6.3.7 Altering the Destination of Locally-Generated Connections

The NAT code allows you to insert DNAT rules in the OUTPUT chain, but this is not fully supported in 2.4 (it can be, but it requires a new configuration option, some testing, and a fair bit of coding, so unless someone contracts Rusty to write it, I wouldn't expect it soon).

The current limitation is that you can only change the destination to the local machine (e.g. 'j DNAT –to 127.0.0.1'), not to any other machine, otherwise the replies won't be translated correctly.

7 Special Protocols

Some protocols do not like being NAT'ed. For each of these protocols, two extensions must be written; one for the connection tracking of the protocol, and one for the actual NAT.

Inside the netfilter distribution, there are currently modules for ftp: ip_conntrack_ftp.o and ip_nat_ftp.o. If you insmod these into your kernel (or you compile them in permanently), then doing any kind of NAT on ftp connections should work. If you don't, then you can only use passive ftp, and even that might not work reliably if you're doing more than simple Source NAT.

8 Caveats on NAT

If you are doing NAT on a connection, all packets passing **both** ways (in and out of the network) must pass through the NAT'ed box, otherwise it won't work reliably. In particular, the connection tracking code reassembles fragments, which means that not only will connection tracking not be reliable, but your packets may not get through at all, as fragments will be withheld.

9 Source NAT and Routing

If you are doing SNAT, you will want to make sure that every machine the SNAT'ed packets goes to will send replies back to the NAT box. For example, if you are mapping some outgoing packets onto the source address 1.2.3.4, then the outside router must know that it is to send reply packets (which will have **destination** 1.2.3.4) back to this box. This can be done in the following ways:

1. If you are doing SNAT onto the box's own address (for which routing and everything already works), you don't need to do anything.
2. If you are doing SNAT onto an unused address on the local LAN (for example, you're mapping onto 1.2.3.99, a free IP on your 1.2.3.0/24 network), your NAT box will need to respond to ARP requests for that address as well as its own: the easiest way to do this is create an IP alias, e.g.:

   ```
   # ip address add 1.2.3.99 dev eth0
   ```

3. If you are doing SNAT onto a completely different address, you will have to ensure that the machines the SNAT packets will hit will route this address back to the NAT box. This is already achieved if the NAT box is their default gateway, otherwise you will need to advertise a route (if running a routing protocol) or manually add routes to each machine involved.

10 Destination NAT Onto the Same Network

If you are doing port forwarding back onto the same network, you need to make sure that both future packets and reply packets pass through the NAT box (so they can be altered). The NAT code will now (since 2.4.0-test6), block the outgoing ICMP redirect which is produced when the NAT'ed packet heads out the same interface it came in on, but the receiving server will still try to reply directly to the client (which won't recognize the reply).

The classic case is that internal staff try to access your 'public' web server, which is actually DNAT'ed from the public address (1.2.3.4) to an internal machine (192.168.1.1), like so:

```
# iptables -t nat -A PREROUTING -d 1.2.3.4 \
        -p tcp --dport 80 -j DNAT --to 192.168.1.1
```

One way is to run an internal DNS server which knows the real (internal) IP address of your public web site, and forward all other requests to an external DNS server. This means that the logging on your web server will show the internal IP addresses correctly.

The other way is to have the NAT box also map the source IP address to its own for these connections, fooling the server into replying through it. In this example, we would do the following (assuming the internal IP address of the NAT box is 192.168.1.250):

```
# iptables -t nat -A POSTROUTING -d 192.168.1.1 -s 192.168.1.0/24 \
        -p tcp --dport 80 -j SNAT --to 192.168.1.250
```

Because the **PREROUTING** rule gets run first, the packets will already be destined for the internal web server: we can tell which ones are internally sourced by the source IP addresses.

11 Thanks

Thanks first to WatchGuard, and David Bonn, who believed in the netfilter idea enough to support me while I worked on it.

And to everyone else who put up with my ranting as I learnt about the ugliness of NAT, especially those who read my diary.

Rusty.