iptables debugging

Posted by waldner on 11 June 2010, 9:15 pm

Has it ever happened to you that **iptables** was apparently not working as expected, and, in an effort to find out what's going on, you littered your ruleset with logging rules all over the place, or some other awkward kludge? Now, it turns out that there's a much more convenient and cleaner way to find out which chains a packet traverses. This is based on using ip{,6}tables' **raw** table, with the **TRACE** target. For the following discussion, it helps if you keep an eye on this excellent iptables flow diagram.

In the diagram, we can see that the **raw** table has two built-in chains: **PREROUTING** and **OUTPUT**, which together cover both the input and output of packets. In other words, *any* packet in the system, be it for the local system, locally generated, or forwarded, traverses one or both these chains in the raw table. These chains support a special target, namely **TRACE**. Here's what the man page says about **TRACE**:

> **TRACE**
> This target marks packes so that the kernel will log every rule which match the packets as those traverse the tables, chains, rules. (The ipt_LOG or ip6t_LOG module is required for the logging.) The packets are logged with the string prefix: "TRACE: tablename:chainname:type:rulenum " where type can be "rule" for plain rule, "return" for implicit rule at the end of a user defined chain and "policy" for the policy of the built in chains. It can only be used in the **raw** table.

If you're building your own kernel, you need to enable or build as module at least **CONFIG_NETFILTER_XT_TARGET_TRACE**, one or both of **CONFIG_IP_NF_RAW** and **CONFIG_IP6_NF_RAW** to get the **raw** table, and again one or both of **CONFIG_IP_NF_TARGET_LOG** and **CONFIG_IP6_NF_TARGET_LOG** for logging (that's where the debug info will be written to). The **TRACE** target should be available from kernel 2.6.23 onwards.

Configuration

Configuration is really straightforward. Let's trace just **ICMP** echo packets for simplicity (of course, you'll trace whatever packets are relevant for your situation). First, we add rules to trace the relevant packets to the **raw** table's **OUTPUT** and **PREROUTING** chains, to catch all possible packet paths:

```
# for IPv4
# iptables -t raw -A OUTPUT -p icmp -j TRACE
# iptables -t raw -A PREROUTING -p icmp -j TRACE
# for IPv6
# ip6tables -t raw -A OUTPUT -p icmpv6 --icmpv6-type echo-request -j TRACE
# ip6tables -t raw -A OUTPUT -p icmpv6 --icmpv6-type echo-reply -j TRACE
# ip6tables -t raw -A PREROUTING -p icmpv6 --icmpv6-type echo-request -j TRACE
# ip6tables -t raw -A PREROUTING -p icmpv6 --icmpv6-type echo-reply -j TRACE
```

Yes, I should have specified the ICMP type for IPv4, too. But you get the idea, and for our purposes that works fine. For IPv6, where ICMP is used for many other things like neighbor discovery etc., tracing *all* ICMP types would pollute the traces with extra traffic.

Now, where is the debug information going to be logged? This info comes from the kernel, so it will use the LOG_KERN facility (kern.* in syslog speak). Check your logging daemon config to see where those messages are logged (they may end up in more than on file). On most machines, they end up in **/var/log/kern.log**.
Let's load the logging module (if it's not built-in)

```
# for IPv4
# modprobe ipt_LOG
# for IPv6
# modprobe ip6t_LOG
```

Some examples

Now we're ready to see tracing in action. Let's generate some ICMP traffic.

Local traffic, IPv4

```
# ping -c 1 www.example.com
PING www.example.com (192.0.32.10) 56(84) bytes of data.
64 bytes from www.example.com (192.0.32.10): icmp_seq=1 ttl=243 time=156 ms

--- www.example.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 156.370/156.370/156.370/0.000 ms
```

Here's the trace in the log (line numbers and spacing added for readability):

```
 1 Jun 10 21:38:18 rowlf klogd: [39964.568632] TRACE: raw:OUTPUT:policy:2 IN= OUT=br
 2 Jun 10 21:38:18 rowlf klogd: [39964.568645] TRACE: mangle:OUTPUT:policy:1 IN= OUT
 3 Jun 10 21:38:18 rowlf klogd: [39964.568653] TRACE: nat:OUTPUT:policy:1 IN= OUT=br
 4 Jun 10 21:38:18 rowlf klogd: [39964.568662] TRACE: filter:OUTPUT:policy:1 IN= OUT
```

```
 5 Jun 10 21:38:18 rowlf klogd: [39964.568672] TRACE: mangle:POSTROUTING:policy:1 IN
 6 Jun 10 21:38:18 rowlf klogd: [39964.568679] TRACE: nat:POSTROUTING:policy:2 IN= O

 7 Jun 10 21:38:18 rowlf klogd: [39964.725001] TRACE: raw:PREROUTING:policy:2 IN=br0
 8 Jun 10 21:38:18 rowlf klogd: [39964.725024] TRACE: mangle:PREROUTING:policy:1 IN=
 9 Jun 10 21:38:18 rowlf klogd: [39964.725051] TRACE: mangle:INPUT:policy:1 IN=br0 O
10 Jun 10 21:38:18 rowlf klogd: [39964.725066] TRACE: filter:INPUT:policy:5 IN=br0 O
```

There is a lot to see here. Looking at the iptables diagram mentioned at the beginning, it's possible to make sense of what we're seeing. Also note that after each **table:CHAIN** the matching rule(s) or policy in that chain are shown, to make debugging easier. The rules can be printed with **iptables-save** or **iptables -S**.
The first packet is a locally generated packet, and similarly the return packet is addressed to the local machine. It can clearly be seen that lines 1 to 6 trace the outgoing echo request packet, while lines from 7 to 10 trace the echo reply return packet. Note that the chains in the **nat** table are NOT traversed by the return packet. This is by design; only packets in the "NEW" state go through **nat** chains. Also note that the interface is a bridge, and iptables differentiates between **IN** (br0) and **PHYSIN** (eth0).

Transit traffic, IPv4

And now let's check out some transit traffic, IPv4:

```
 1 Jun 10 21:54:23 rowlf klogd: [40929.881158] TRACE: raw:PREROUTING:policy:2 IN=br2
 2 Jun 10 21:54:23 rowlf klogd: [40929.881181] TRACE: mangle:PREROUTING:policy:1 IN=
 3 Jun 10 21:54:23 rowlf klogd: [40929.881197] TRACE: nat:PREROUTING:policy:1 IN=br2
 4 Jun 10 21:54:23 rowlf klogd: [40929.881224] TRACE: mangle:FORWARD:policy:1 IN=br2
 5 Jun 10 21:54:23 rowlf klogd: [40929.881237] TRACE: filter:FORWARD:rule:2 IN=br2 O
 6 Jun 10 21:54:23 rowlf klogd: [40929.881246] TRACE: mangle:POSTROUTING:policy:1 IN
 7 Jun 10 21:54:23 rowlf klogd: [40929.881256] TRACE: nat:POSTROUTING:rule:1 IN= OUT

 8 Jun 10 21:54:23 rowlf klogd: [40930.037578] TRACE: raw:PREROUTING:policy:2 IN=br0
 9 Jun 10 21:54:23 rowlf klogd: [40930.037598] TRACE: mangle:PREROUTING:policy:1 IN=
10 Jun 10 21:54:23 rowlf klogd: [40930.037623] TRACE: mangle:FORWARD:policy:1 IN=br0
11 Jun 10 21:54:23 rowlf klogd: [40930.037630] TRACE: filter:FORWARD:rule:1 IN=br0 O
12 Jun 10 21:54:23 rowlf klogd: [40930.037638] TRACE: mangle:POSTROUTING:policy:1 IN
```

This is again a echo request/echo reply exchange, sent by a machine in the network through this box. Incidentally, this computer also does NAT. Lines 1 to 7 trace the echo request packet, lines 8 to 12 trace the echo reply. Looking at the various **IN=**, **OUT=**, **PHYSIN=**, etc. fields it can be observed when decisions about which interfaces to use are made. NAT for outgoing packets happens in the **nat:POSTROUTING** chain, so the rewriting of the source address is not visible above (it could be seen using **tcpdump**, though); however, the reverse process is visible in the return packet: after line 9, the destination address changes from the local box (**DST=192.168.6.131**) to the original sender host (**DST=10.0.2.199**); also at the same time the output interface **OUT=br2** is selected. Again note that, although the kernel does the

reverse rewriting process as we just saw, no **nat** chain is traversed by the return packet, because the connection is no longer NEW.

Local traffic, IPv6

Now with IPv6 (**2a00:1450:8002::63** is one of www.google.com's IPv6 addresses):

```
# ping6 -c1 2a00:1450:8002::63
PING 2a00:1450:8002::63(2a00:1450:8002::63) 56 data bytes
64 bytes from 2a00:1450:8002::63: icmp_seq=1 ttl=57 time=27.2 ms

--- 2a00:1450:8002::63 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 27.217/27.217/27.217/0.000 ms
```

The machine is using an IPv6-in-IPv4 tunnel, so the trace is particularly interesting. Here it is (make sure you scroll all the way to the right, there's a lot of info):

```
 1 Jun 11 09:57:34 scooter kernel: TRACE: raw:OUTPUT:policy:3 IN= OUT=ip6tun SRC=200
 2 Jun 11 09:57:34 scooter kernel: TRACE: mangle:OUTPUT:policy:1 IN= OUT=ip6tun SRC=
 3 Jun 11 09:57:34 scooter kernel: TRACE: filter:OUTPUT:policy:1 IN= OUT=ip6tun SRC=
 4 Jun 11 09:57:34 scooter kernel: TRACE: mangle:POSTROUTING:policy:1 IN= OUT=ip6tun

 5 Jun 11 09:57:34 scooter kernel: TRACE: raw:OUTPUT:policy:1 IN= OUT=eth0 SRC=10.12
 6 Jun 11 09:57:34 scooter kernel: TRACE: mangle:OUTPUT:policy:1 IN= OUT=eth0 SRC=10
 7 Jun 11 09:57:34 scooter kernel: TRACE: nat:OUTPUT:policy:1 IN= OUT=eth0 SRC=10.12
 8 Jun 11 09:57:34 scooter kernel: TRACE: filter:OUTPUT:policy:1 IN= OUT=eth0 SRC=10
 9 Jun 11 09:57:34 scooter kernel: TRACE: mangle:POSTROUTING:policy:1 IN= OUT=eth0 S
10 Jun 11 09:57:34 scooter kernel: TRACE: nat:POSTROUTING:policy:4 IN= OUT=eth0 SRC=

11 Jun 11 09:57:34 scooter kernel: TRACE: raw:PREROUTING:policy:3 IN=ip6tun OUT= MAC
12 Jun 11 09:57:34 scooter kernel: TRACE: mangle:PREROUTING:policy:1 IN=ip6tun OUT=
13 Jun 11 09:57:34 scooter kernel: TRACE: mangle:INPUT:policy:1 IN=ip6tun OUT= MAC=0
14 Jun 11 09:57:34 scooter kernel: TRACE: filter:INPUT:policy:1 IN=ip6tun OUT= MAC=0
```

The different phases of tunnelling can be observed (**ip6tun** is the tunnel interface): lines 1-4 are for the outgoing IPv6 packet, lines 5-10 for the outgoing encapsulating IPv4 packet (and they are printed even if IPv4 tracing is not enabled). Note that in the case of the return packet (lines 11-14), things are not symmetrical (or I'm missing something, which is also possible); however there is a **TUNNEL=172.21.17.6->10.127.68.105** specification as part of the debugging information.
Also note that the IPv4 packet DOES traverse the **nat** chains, because this was explicitly set up to be the very first IPv4 packet in the tunnel connection (hence the connection was in state NEW); in practice, if the tunnel is in constant use, the connection will not be NEW and the **nat** chains will not usually be traversed. Lines 7 and 10 above will be missing from the traces.

Transit traffic, IPv6

Finally, here's a trace of transit IPv6 traffic (native, no IPv4 tunnelling):

```
 1 Jun 11 20:54:18 arch kernel: [ 1216.851243] TRACE: raw:PREROUTING:policy:3 IN=eth
 2 Jun 11 20:54:18 arch kernel: [ 1216.855156] TRACE: mangle:PREROUTING:policy:1 IN=
 3 Jun 11 20:54:18 arch kernel: [ 1216.860565] TRACE: mangle:FORWARD:policy:1 IN=eth
 4 Jun 11 20:54:18 arch kernel: [ 1216.863630] TRACE: filter:FORWARD:policy:1 IN=eth
 5 Jun 11 20:54:18 arch kernel: [ 1216.866650] TRACE: mangle:POSTROUTING:policy:1 IN

 6 Jun 11 20:54:18 arch kernel: [ 1216.872808] TRACE: raw:PREROUTING:policy:3 IN=eth
 7 Jun 11 20:54:18 arch kernel: [ 1216.878469] TRACE: mangle:PREROUTING:policy:1 IN=
 8 Jun 11 20:54:18 arch kernel: [ 1216.884192] TRACE: mangle:FORWARD:policy:1 IN=eth
 9 Jun 11 20:54:18 arch kernel: [ 1216.888587] TRACE: filter:FORWARD:policy:1 IN=eth
10 Jun 11 20:54:18 arch kernel: [ 1216.892593] TRACE: mangle:POSTROUTING:policy:1 IN
```

Conclusion

When we're done with our debugging, all we have to do is just delete the tracing rules:

```
# iptables -t raw -D OUTPUT -p icmp -j TRACE
# iptables -t raw -D PREROUTING -p icmp -j TRACE
# ip6tables -t raw -D OUTPUT -p icmpv6 --icmpv6-type echo-request -j TRACE
# ip6tables -t raw -D OUTPUT -p icmpv6 --icmpv6-type echo-reply -j TRACE
# ip6tables -t raw -D PREROUTING -p icmpv6 --icmpv6-type echo-request -j TRACE
# ip6tables -t raw -D PREROUTING -p icmpv6 --icmpv6-type echo-reply -j TRACE
```

Hopefully, using TRACE makes iptables debugging easier.

Filed under linux, networking, tips, worksforme Tagged ip6tables, iptables, IPv4, IPv6, raw, trace
| Permalink

5 Comments

1. *noname* says:
   May 29, 2018 at 10:22

   On 4.4.0 I had to do:

   modprobe nf_log_ipv4
   sysctl net.netfilter.nf_log.2=nf_log_ipv4

   And if you get nothing in /var/log/kern.log, check if you have kernel logging enabled in syslog and if LOG_KERN facility exists.

2. *Chris* says:
   May 29, 2015 at 00:08

With Kernel 4.0.4, I had to use:

sysctl 'net.netfilter.nf_log.2=nf_log_ipv4

3. *tudor* says:
   November 6, 2013 at 11:33

   Works after configure this:
   sudo sysctl net.netfilter.nf_log.2=ipt_LOG

   Thanks!

4. *Yanis Guenane* says:
   May 30, 2013 at 18:12

   Thank you for this post. Really informative and actually straigh to the point. Exactly what I needed

5. *Saibaba Telukunta* says:
   October 23, 2011 at 13:19

   You saved my day! Thank you.