Towards the perfect ruleset

Jan Engelhardt

v. May 2011

Abstract

Rulesets created by beginners to the iptables Linux firewalling software often are sub-optimal with regard to performance. Larger rulesets also mean a harder time for anybody to look after them. This document shall describe some optimizations and transformations that can be applied to improve processing.

Familiary with the iptables syntax and available options is required.

Contents

1 Prolog

New users continuously hit the IRC channel and most likely also other forums with their gory ad-hoc iptables scripts. (Scripts even, bwa!) Many of these have bad quality just from the looks of it, let alone performance.

This e-book shall serve as a reference on how to clean up your rules to make it easier for the both of us (you, and the people you are seeking help from) have to look through a fewer, and less complex number of rules.

Unless otherwise stated, iptables will herein refer to all of iptables, ip6tables, arptables and ebtables.

2 Debugging rulesets

Can't figure out where a packet gets lost or where a rule does not trigger? Turn on tracing for packets of interest:

```
iptables -t raw -A PREROUTING/OUTPUT [-m ...] -j TRACE
```

Be sure to have a logging backend loaded. Possible backends are ipt_LOG/ip6t_LOG (direct-to-syslog) or nfnetlink_log (for userspace logging via, e. g. ulogd). To activate direct-to-syslog, all you need to do is modprobing the appropriate LOG module.

3 Use iptables-restore

As of June 2009, iptables and its relatives are still bound by the kernel interfaces of the ip_tables etc. modules, which only provide means to get and set entire tables. Old-fashioned scripts execute `/sbin/iptables` over and over — once for each policy they want to set, once for each flush operation they attempt, and once for each rule they are adding to the system — spend a much higher time finishing execution than would normally be required. Every iptables invocation will download the ruleset from the kernel, perform the modification — in case of `-A` this is adding just one rule — and upload it back into the kernel. Repeat that n times, and you get a cost factor of $\mathcal{O}(n^2)$ to perform n operations.

There is also a security risk, a window of opportunity where packets might pass halfway through while a ruleset is copied back and forth. Setting the policy to DROP before will not solve this problem, as this novice example shows:

```
iptables -P INPUT DROP;
# 2. Allow VPNs
iptables -A INPUT -p esp -j ACCEPT;
iptables -P FORWARD DROP;
# 4. Only allow forwarding for certain endpoints
iptables -A FORWARD -m policy --tunnel-src 1.2.3.4 -j ACCEPT;
```

In this example, there is a window of opportunity between the time rule 2 and 3 get active. Of course you can reorder this specific example to be free of possible attack windows, but my mathematical gut feeling tells me there you can always find a way to thwart a given ruleset.

For these reasons, it is much more efficient to use iptables-save/iptables-restore instead for the initial setup of the ruleset. iptables-restore also allows you to flush the tables at once, set the chains' policies, together with specifying the ruleset, all in *one atomic operation*. By smart application of shell knowledge, iptables-restore can even be used with variables:

```
#!/bin/sh

management=85.213.68.203/27

iptables-restore <<-EOF;
        *filter
        :INPUT DROP [0:0]
        :FORWARD DROP [0:0]
        :OUTPUT ACCEPT [0:0]

        # we can also do comments - and even variables!
        -A INPUT -d $management -p tcp --dport 22 -j ACCEPT
        COMMIT
EOF
```

Comments and blank lines are there for convenient documentation issues. The indent you see here is a shell feature (the minus in "-EOF"[1]), it will be stripped before passing on to iptables. Continuing lines with backslashes should be working too, as the shell expands them.

4 Redundant arguments

Redundant default arguments can be removed from your ruleset. These include, among others: `-p all`, "0/0" or "::/0" for addresses (`-s`, `-d`, and elsewhere), "0:65535" for TCP/UDP/SCTP/DCCP port specifications.

5 Modern extensions

Obsolete extensions:

- `-m state`: replaced by `-m conntrack`
- `-j NOTRACK`: replaced by `-j CT --notrack` (from 2.6.35)

6 Path MTU Discovery/TCPMSS

Do not block the ICMP type "parameter-problem", or you will be breaking Path MTU Discovery (PMTUD) for everybody. It also makes the use of the TCPMSS target redundant, and thus saves you rules in your tables.

7 The loopback interface

Try avoiding blocking loopback traffic in the INPUT and OUTPUT chains. Local processes may rely heavily on it, or in fact, may even depend on it.

[1] See bash(1) manual for details.

7.1 Loopback is not just 127.0.0.1

```
-A INPUT -s 127.0.0.1 -j ACCEPT
```

This is wrong for two reasons. The first is that it can allow spoofed packets sent from another machine (and thus would warrant adding `-i lo`). The second is that 127.0.0.1 is not the only IPv4 address your machine has. God knows what made the people at IANA devote an entire /8 subnet to localhost**(author?)** [RFC3330], but that is how it is. Despite this oddity, Linux distributions do sometimes make use of additional loopback network addresses, such as 127.0.0.2, to counter some stupidities of broken software, or for convenience[2].

7.2 Loopback is not just 127/8, either

There is more however. All packets that are destined for an IP address that an interface on the local machine has[3] will be routed over "lo". Connecting to yourself with '`ssh 192.168.1.1`' (assuming that is yours) will make the byte/packet counters for "lo" rise. The correct way is therefore:

```
-A INPUT -i lo -j ACCEPT
```

8 Reverse Path Filtering

(Some call it Reverse Path Forwarding, but since we want to filter something, Filtering comes in good.)

Many a rulesets show to have

```
-A INPUT -i ppp0 -s 192.168.0.0/16 -j DROP
```

Assuming ppp0 is the public Internet and not a VPN (where 192.168/16 could be legitimate), such rules can be eliminated if RPF is activated. (See `/proc/sys/net/ipv4/conf/all/rp_filter`.) Some setups, notably with asymmetric routing, may not be eligible to use RPF however.

9 TCP flag checks

If you happen to make use of connection state tracking (it incurs a runtime cost of course, but is very valuable), you can also use the light-weight TCP flag combination it ships with. Instead of spending oodles of rules trying to determine what is good and what is bad, such as in this posted example:

```
# Drop those nasty packets! These are all TCP flag
# combinations that should never, ever occur in the
# wild. All of these are illegal combinations that
# are used to attack a box in various ways, so we
# just drop them and log them here.
-A INPUT -p tcp --tcp-flags ALL FIN,URG,PSH -j badflags
```

[2] See `https://bugzilla.novell.com/show_bug.cgi?id=416964`.

[3] Actually, interface addresses do not matter. If there is a route entry for local delivery, it will be done so. See '`ip route list table local`'. Adding an address to an interface usually instantiates a number of route entries automatically alongside it.

```
-A INPUT -p tcp --tcp-flags ALL ALL -j badflags
-A INPUT -p tcp --tcp-flags ALL SYN,RST,ACK,FIN,URG -j badflags
-A INPUT -p tcp --tcp-flags ALL NONE -j badflags
-A INPUT -p tcp --tcp-flags SYN,RST SYN,RST -j badflags
-A INPUT -p tcp --tcp-flags SYN,FIN SYN,FIN -j badflags
```

You could just simply use the INVALID state match (limted to tcp, or used for all protocols — it's up to you):

```
-A INPUT (-p tcp) -m conntrack --ctstate INVALID -j badflags
```

The "badflags" chain is just a placeholder in this example — a user could have placed `-j DROP` instead, or anything else he desired to (or not to) handle a bad flag combination.

10 Combination of matches

10.1 Aggregation of prefixes

A prominent example of suboptimal rules is a list of IP addresses or groups, such as:

```
-A INPUT -d 224.0.0.0/8 -j DROP
-A INPUT -d 225.0.0.0/8 -j DROP
...
-A INPUT -d 239.0.0.0/8 -j DROP
```

The intent in this example is to match traffic with a multicast destination address. Similar patterns may be encountered with different options or targets.

Adjacent networks can be combined into a netmask with higher-order prefix if the higher-order netmask contains all of the addresses that we are trying to combine. The stepwise procedure is to combine /8s into /7s, those into /6s, those into /5s, and those into a /4, yielding 224/4 in the end. (Gifted thinkers can do it in one step.)

$$
224_4 \left\{ \begin{array}{ll}
224_5 \left\{ \begin{array}{ll}
224_6 \left\{ \begin{array}{ll} 224_7 \left\{ \begin{array}{l} 224_8 \\ 225_8 \end{array} \right. \\ 226_7 \left\{ \begin{array}{l} 226_8 \\ 227_8 \end{array} \right. \end{array} \right. \\
\ldots \\
\ldots \\
236_6 \left\{ \begin{array}{ll} 236_7 \left\{ \begin{array}{l} 236_8 \\ 237_8 \end{array} \right. \\ 238_7 \left\{ \begin{array}{l} 238_8 \\ 239_8 \end{array} \right. \end{array} \right.
\end{array} \right. \\
232_5
\end{array} \right.
$$

Not all networks can be directly combined. For example, had you only 225/8 and 226/8, there would be no corresponding /7 network for it.

10.2 Address range matching

Now, of course it is easy to construct a hypotetical case where mask grouping is not performing perfect either. Consider this 16-rule sample:

```
-A INPUT -s 10.10.96.255/32 -j ACCEPT
-A INPUT -s 10.10.97.0/32 -j ACCEPT
-A INPUT -s 10.10.97.1/32 -j ACCEPT
...
-A INPUT -s 10.10.97.14/32 -j ACCEPT
```

By use of section 10.1's prefix aggregation transformation, one would get a 5-rule output:

```
-A INPUT -s 10.10.96.255/32 -j ACCEPT
-A INPUT -s 10.10.97.0/29 -j ACCEPT
-A INPUT -s 10.10.97.8/30 -j ACCEPT
-A INPUT -s 10.10.97.12/31 -j ACCEPT
-A INPUT -s 10.10.97.14/32 -j ACCEPT
```

These are the maximum possible subnets that can be specified in rules. Using just one order higher, e. g. 10.10.97.**0/28**, 10.10.97.**8/29**, 10.0.97.**12/30** would have caused 10.10.97.15 to get matched too — but that address was not listed in the original set!

So, can we do better than five rules? Yes, in fact, by large. The iprange match allows to make this stunningly simple:

```
-A INPUT -m iprange --src-range 10.10.96.255-10.10.97.14 -j ACCEPT
```

10.3 Arbitrary address mask matching

One of the more obscure, hidden, side features of iptables is arbitrary mask matching. It is not obvious at first, because it need not have a corresponding CIDR-conforming "/n" mask. Here is a 4-rule sample that would defeat range grouping as per section 10.2:

```
-A INPUT -s 10.10.97.1/32 -j ACCEPT
-A INPUT -s 10.10.97.3/32 -j ACCEPT
-A INPUT -s 10.10.97.5/32 -j ACCEPT
-A INPUT -s 10.10.97.7/32 -j ACCEPT
```

The "holes" between 10.10.97.1 and 10.10.97.7 — that were deliberately chosen for this case — make range grouping absolutely meaningless, because you would still need four iprange rules. iptables allowing arbitrary masks has a solution for this too:

```
-A INPUT -s 10.10.97.1/255.255.255.249 -j ACCEPT
```

The /255.255.255.249 mask would succeed for all hosts that have bits 1 and 2 (counting from 0) cleared. Combining this with the iprange module allows, for example, to only match hosts .3 and .5 when a mask of .249 is used.

Using such a non-prefix mask is only possible in a few spots.

10.4 Arbitrary address sets

And still, one may be having to test for lots of addresses where none of the above solution produces an acceptable result with few enough rules. In this case, specially-optimized set matching through the set match, part of iptables/ipset, is recommended, but is currently beyond the scope of this document.

10.5 Port lists with "multiport"

```
-A INPUT -p tcp --dport 21 -j ACCEPT
-A INPUT -p tcp --dport 22 -j ACCEPT
-A INPUT -p tcp --dport 6881:6882 -j ACCEPT
```

The multiport match can accomodate port numbers and port ranges, up to 15 "numbers" in total:

```
-A INPUT -p tcp -m multiport --dports 21,22,6881:6882 -j ACCEPT
```

Not only is this clearer, but also much faster[4] in execution than writing separate rules.

Again, ipset has modules for faster large-scale port matching (usually at the cost of a large bitmap).

11 Hit frequency

Because iptables processes rules in linear order, from top to bottom within a chain, it is advised to put frequently-hit rules near the start of the chain. Of course there is a limit, depending on the logic that is being implemented. Also, rules have an associated runtime cost, so rules should not be reordered solely based upon empirical observations of the byte/packet counters. While I do not have any hard data on this (one of the many kernel profilers can probably tell), a large number of matches within a rule is surely a candidate.

11.1 State tracking

Unless you are taking fancy random actions on your packets, every standard setup has a line like

```
-A INPUT -m conntrack --ctstate ESTABLISHED -j ACCEPT
```

It may also appear in FORWARD, or sometimes in OUTPUT, and it may also occur with the RELATED state added as an extra state that is to be let through too. Such a rule will let through the majority of the traffic. On a multi-purpose dedicated server of ours, 99.02% of all past-37 days traffic (41134 MB) was matched by such a rule.

If you unconditionally accept established traffic, your remaining rules only need to deal with new connections (and possibly related, unless you have that unconditionally allowed too).

12 Further DOs and DON'Ts

12.1 Should Not

- Should not block received "destination-unreachable" ICMP messages to ease detection on your side that there is a remote problem (such as a closed port, etc.).
- Should not block received "time-exceeded" ICMP messages, so that your traceroute instance can actually work.

[4] Both rule processing and multiport list processing are of same complexity $\mathcal{O}(n)$, but multiport has a "smaller $\mathcal{O}$".

12.2 Must Not

- Must not use the DROP target in the "nat" table. DROP is a filter, while the nat table is meant to be exclusively used as a NAT transformation setup database. This table will not see all packets, and the commonly-heard lemma that "it sees only the first packet" is only half the truth, and ambiguous at that.

12.3 FTP

A fair number of rulesets contain something to allow port 20. That is however quite useless, because ftp-data does not always run on 20. The ports can be, and generally are, dynamically chosen, on both sides, in active and passive mode.

Consider client-passive mode:

```
client> EPSV
server> 229 Entering Extended Passive Mode (|||51458|)
```

Client-active mode:

```
client> PORT 134,76,2,119,123,45
server> 200 PORT command successful.
```

No port 20 in there. If you already have a `--ctstate RELATED` rule, it will match initial packets on the used data ports automatically, given the nf_conntrack_ftp kernel module is loaded. The use of `--ctstate ESTABLISHED` then makes sure the rest is also accepted.

There is also `-m helper --helper ftp`, but I would not recommend using it on the plentiful established traffic (it does a string comparison after all, seems costly), and instead only use it with rules dealing with RELATED.

13 Epilogue

I very much recommend the reader to get familiar with iptables and how to write rules on his/her own. Premade scripts often try to accomodate for many use cases, and as a result get more blown up than would technically be needed.

Noteworthy programs that fall under this category is fwbuilder, which only knows "inbound" and "outbound" states in its GUI, and bloats the INPUT and FORWARD chains with rules from each another.

References

[RFC3330] RFC 3330: Special-Use IPv4 Addresses
`http://tools.ietf.org/html/rfc3330`
IANA, September 2002 5

[2]