

Packet Filtering: from iptables to nftables

Lesson 01 · Linux Networking · ~10 min read

What a Linux firewall actually is

When people say "Linux firewall", they usually mean *netfilter* — a framework built directly into the Linux kernel since version 2.4. Netfilter intercepts every packet that the kernel processes, whether the packet arrived from the network, was generated by a local process, or is being forwarded between two interfaces. It does this by registering callback functions at five fixed points in the kernel's network stack, called **hooks**.

`iptables` and `nftables` are *not* the firewall itself — they are command-line front-ends that load rules into netfilter. Think of netfilter as the engine and iptables or nftables as the configuration language you use to program it. When you run `iptables -A INPUT -j DROP`, you are inserting a rule into a data structure that the kernel reads for every incoming packet. The rule evaluation happens in the kernel; the CLI tool just writes the rules and exits.

This distinction matters when you troubleshoot. If packets are being dropped and you don't know why, you inspect the rules that were loaded into the kernel — not the command you typed last week. The commands `iptables -L -v -n` and `nft list ruleset` both read live kernel state.

The five netfilter hooks

Every packet the kernel touches passes through a subset of five hooks, depending on where the packet is going. Understanding which hook fires for which traffic is the mental model that makes all other firewall concepts fall into place.

- **PREROUTING** — fires the instant a packet arrives on any interface, before the kernel has decided whether to deliver it locally or forward it. This is where DNAT (destination NAT) lives.

- **INPUT** — fires for packets that are addressed to this machine (the kernel routing decision found a local destination). This is where you filter inbound traffic for local processes.
- **FORWARD** — fires for packets the kernel is routing from one interface to another. This hook only sees traffic when `ip_forward` is enabled and the machine acts as a router.
- **OUTPUT** — fires for packets generated by local processes before they leave. You can filter outbound traffic here.
- **POSTROUTING** — fires last, just before a packet leaves any interface. This is where SNAT and MASQUERADE live.

Use the interactive widget below to walk a packet through the hooks step by step. Pick a rule and a packet type, then click **Send** to see which verdict the rule produces.

Active rule:

-A INPUT -p tcp --dport 22 -j ACCEPT

Send packet:

New SSH (tcp/22, NEW)

Send



-p tcp --dport 22 -j ACCEPT

Pick a rule and a packet, then click Send.

Tables and chains

Netfilter organises rules into **tables**, and each table contains one or more **chains**. A chain is simply an ordered list of rules. When a packet reaches a hook, the kernel evaluates the rules in the relevant chain from top to bottom until one matches. The matching rule issues a *verdict*: `ACCEPT` (let it through), `DROP` (silently discard), `REJECT` (discard and send an error back), or `LOG` (record and continue).

The three most important tables are:

- **filter** — the default table. Attached to INPUT, FORWARD, and OUTPUT. This is where your allow/deny rules live.
- **nat** — attached to PREROUTING, OUTPUT, and POSTROUTING. Used for rewriting source or destination addresses (SNAT, DNAT, MASQUERADE).
- **mangle** — attached to all five hooks. Used for modifying packet headers (TTL, TOS, etc.) without altering addresses.

When you type `iptables -A INPUT -p tcp --dport 22 -j ACCEPT`, you are appending a rule to the `INPUT` chain of the `filter` table. The `-A` flag means append (add to the end). If you need a rule to be checked before others, use `-I` (insert at position 1 by default).

Rule order matters

In iptables, rules are evaluated in the order they appear in the chain. The **first match wins**: once a packet matches a rule and receives a terminating verdict, the remaining rules are not checked. This means the order you add rules is not cosmetic — it directly controls behaviour.

A classic mistake is adding a permissive rule after a blanket DROP. If you set the default policy to DROP and then add `-A INPUT -p tcp --dport 22 -j ACCEPT`, SSH works. But if you accidentally add `-A INPUT -j DROP` as a rule *before* the SSH rule, that explicit DROP will catch every packet first and SSH will be locked out — even though the ACCEPT rule exists.

Drag the rules in the widget below to see this effect directly:

Drag rules to reorder. Then send a test packet.

`-A INPUT -p tcp --dport 22 -j ACCEPT`

`-A INPUT -j DROP`

`-A INPUT -p tcp --dport 80 -j ACCEPT`

Send tcp/22

Send tcp/80

Reset

Drag to reorder, then send.

Stateful inspection: conntrack

Early firewalls were purely stateless: every packet was evaluated in isolation, with no memory of previous packets. This is cumbersome — you need explicit rules to allow return traffic for every connection you permit outbound. Netfilter's **connection tracking** subsystem (conntrack) solves this by keeping a table of known connections in the kernel.

When a packet arrives, conntrack classifies it as one of four states:

- **NEW** — the first packet of a new connection (e.g. a TCP SYN).
- **ESTABLISHED** — part of a connection the kernel has already seen in both directions.
- **RELATED** — related to an existing connection (e.g. an ICMP error referencing a tracked TCP flow, or an FTP data channel spawned from a control connection).
- **INVALID** — does not match any known connection and cannot be classified.

With conntrack, you can write one rule that allows all return traffic instead of crafting per-port rules:

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

This single rule keeps your firewall stateful and concise. The widget below lets you simulate creating connections and watch the conntrack table update in real time.

Why nftables exists: atomic reloads

`iptables` has been the standard since the early 2000s, but it accumulated several operational problems over time. It uses a separate binary for each protocol family (`iptables`, `ip6tables`, `arptables`). Listing rules requires a different flag combination depending on which table you want. Sets and maps — useful for whitelisting large numbers of IPs — had to be bolted on via `ipset` rather than being a first-class concept.

The most significant operational problem was reloads. When you update firewall rules on a live server, you typically replace the full ruleset using `iptables-restore`. Under the hood, `iptables-restore` flushes all existing rules and reloads them one by one. During the brief window between flush and reload, the kernel is running with an empty ruleset — any packet that arrives in that window passes through unchecked. On a busy server, that window produces at least one leaked packet per reload.

`nftables`, introduced in kernel 3.13 (2014) and stable by kernel 4.x, solves this with **atomic reloads**. When you load a ruleset with `nft -f rules.nft`, the entire new ruleset is compiled into a kernel transaction and swapped in as a single atomic operation. There is no flush-then-reload window. Either the whole new ruleset takes effect, or none of it does.

Click **Reload** in the widget below to see the difference side by side:

Reload rules on both

iptables-restore (racy)

iptables

nft -f (atomic)

nftables

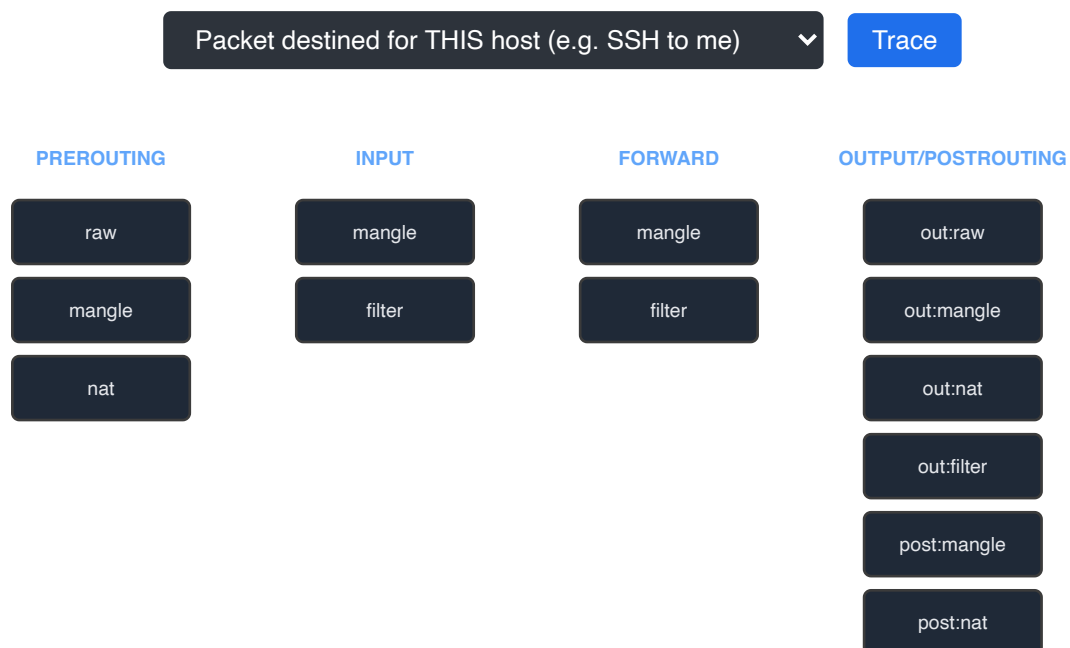
Click "Reload" to see the difference.

Following a packet through the full chain order

Now that you know what tables, chains, hooks, and conntrack are, you can put them together into the full picture: what sequence of chains does a given packet traverse? The answer depends on whether the packet is arriving from outside (inbound), being forwarded by this host as a router, or being sent by a local process (outbound).

For a packet arriving and destined for this host, the order is: **PREROUTING:raw** → **PREROUTING:mangle** → **PREROUTING:nat** → **INPUT:mangle** → **INPUT:filter**. For a forwarded packet, the INPUT chains are replaced by FORWARD chains, and POSTROUTING is appended at the end. For a locally-originated packet, the path starts at OUTPUT and ends at POSTROUTING.

Use the widget below to trace each scenario. Click **Trace** to watch the chain traversal animate step by step. Chains currently being evaluated highlight in yellow; completed chains turn green.



Pick a scenario and click Trace.

Migration: iptables to nftables

Most Linux distributions now ship nftables as the default and provide compatibility shims that route `iptables` commands through an nftables backend. If you are maintaining existing iptables rule sets, the `iptables-translate` utility converts

individual rules to their nftables equivalents. The companion tool `iptables-restore-translate` converts an entire saved iptables ruleset file at once.

```
iptables-translate -A INPUT -p tcp --dport 22 -j ACCEPT
# nft add rule ip filter INPUT tcp dport 22 counter accept
```

A few things to watch during migration: nftables uses different syntax for conntrack state matching (`ct state established,related accept` instead of `-m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT`), and sets are declared separately from rules rather than being inlined as `ipset` objects. Rule ordering within a chain behaves the same way — first match wins — so the logical structure of your ruleset carries over directly.

Quick start: If you only learn one nftables command, learn `nft list ruleset` — it shows everything loaded in all tables, all families, in one shot.

Recap

- Netfilter is the kernel subsystem; iptables/nftables are front-ends that load rules into it.
- Rules live in tables (`filter` / `nat` / `mangle`), grouped into chains attached to hooks (PREROUTING, INPUT, FORWARD, OUTPUT, POSTROUTING).
- Order matters: first match wins. A DROP rule above an ACCEPT rule beats the ACCEPT.
- Conntrack makes stateful rules possible — one `ESTABLISHED,RELATED` rule covers all return traffic.
- nftables fixed iptables' biggest operational pain: atomic reloads eliminate the flush window where packets leak through.