

# **Lesson 01: Packet Filtering: from iptables to nftables**

# Lesson 01 — Packet Filtering

From iptables to nftables, beginner edition.

Linux Networking

In this lesson you will learn how the Linux kernel inspects and filters network packets using the **netfilter** framework, how to write and read **iptables** rules, what **conntrack** is and why it matters, and how the newer **nftables** tool improves on the classic toolchain.

# What is a firewall on Linux?

- **A firewall = packet inspector + decider** — it looks at each packet and decides: allow, drop, or rewrite?
- **On Linux, the inspector lives in the kernel: netfilter** — a set of hooks deep inside the networking stack.
- **iptables/nftables are just front-ends** — they load rules into netfilter. The kernel does the actual filtering.
- No firewall daemon needs to run continuously — the rules are stored in kernel memory.

## Plain English

Think of the Linux kernel as a post office. Every packet is a letter passing through. *Netfilter* is the sorting machine that can inspect, redirect, or shred any letter. *iptables* and *nftables* are the tools you use to program that sorting machine — they just write the instructions, they don't run them.

# The 5 netfilter hooks

A packet's journey through netfilter



- **PREROUTING** — first hook, before routing decision; used for DNAT
- **INPUT** — packets destined for the local machine
- **FORWARD** — packets routed through this machine (router/gateway)
- **OUTPUT** — packets generated by the local machine

- **POSTROUTING** — last hook, after routing decision; used for SNAT/MASQUERADE
-

# Tables vs chains

iptables organises rules into **tables** (purpose) and **chains** (hook attachment points).

| Table               | What it does                                                                   | Typical chains                  | Example rule                                                                   |
|---------------------|--------------------------------------------------------------------------------|---------------------------------|--------------------------------------------------------------------------------|
| <code>filter</code> | Accept, drop, or reject packets — the default firewall table                   | INPUT, FORWARD, OUTPUT          | <code>-A INPUT -p tcp --dport 22</code><br><code>-j ACCEPT</code>              |
| <code>nat</code>    | Rewrite source or destination IP/port (SNAT, DNAT, MASQUERADE)                 | PREROUTING, OUTPUT, POSTROUTING | <code>-t nat -A POSTROUTING -o</code><br><code>eth0 -j MASQUERADE</code>       |
| <code>mangle</code> | Modify packet headers (TTL, TOS, mark for routing) without rewriting addresses | All 5 chains                    | <code>-t mangle -A PREROUTING -j</code><br><code>MARK --set-mark 1</code>      |
| <code>raw</code>    | Bypass conntrack for specific traffic (performance)                            | PREROUTING, OUTPUT              | <code>-t raw -A PREROUTING -p</code><br><code>udp --dport 53 -j NOTRACK</code> |

**Notes:** The `filter` table is the default — when you omit `-t`, you are working in `filter`. Rules are evaluated top-to-bottom; first match wins.

---

# Stateful inspection (conntrack)



- **NEW** — first packet of a connection (e.g. SYN); conntrack creates a tracking entry
- **ESTABLISHED** — packets belonging to an already-opened connection; allowed through without re-inspection
- **RELATED** — packets related to an existing connection (e.g. ICMP errors, FTP data channel)

- **INVALID** — packet doesn't match any known connection and looks malformed; should be dropped
-

# iptables crash course

Three commands every sysadmin needs to know:

**List all rules with packet/byte counters:**

```
iptables -L -v -n
```

The `-v` flag shows interface and packet/byte counters. `-n` prevents slow DNS lookups — always use it.

**Allow inbound SSH:**

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

Appends (`-A`) a rule to the INPUT chain: TCP on port 22 → ACCEPT. The `-A` adds to the end; use `-I` to insert at a specific position.

**Set default policy to drop all inbound:**

```
iptables -P INPUT DROP
```

The policy is the fallback when no rule matches. Set this *after* adding ACCEPT rules for needed services, or you will lock yourself out.

---

# Reading a rule like a sentence

## Anatomy of an iptables rule

```
iptables -A INPUT -p tcp --dport 22 -m conntrack --ctstate NEW,ESTABLISHED -j ACCEPT
```

- `iptables` — the tool that writes rules into the kernel's netfilter tables
- `-A INPUT` — **Append** to the **INPUT** chain (packets destined for this machine)
- `-p tcp` — match only **TCP** packets
- `--dport 22` — match destination port **22** (SSH)
- `-m conntrack --ctstate NEW,ESTABLISHED` — load the conntrack module and match packets in state **NEW** or **ESTABLISHED**
- `-j ACCEPT` — **jump** to target ACCEPT (let the packet through)

Read it aloud: *"For all TCP packets arriving at port 22 that are new or part of an established connection, accept them."*

---

# NAT in action

SNAT (outgoing): POSTROUTING rewrites SOURCE IP

src=10.0.0.5  
dst=8.8.8.8

POSTROUTING

DNAT (incoming): PREROUTING rewrites DEST IP

PREROUTING

src=8.8.8.8 →  
dst=203.0.113.10

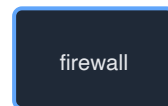
```
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

MASQUERADE is dynamic SNAT — it automatically uses the current IP of the outgoing interface.  
Ideal for dynamic IP addresses (home routers, VMs sharing a host NIC).

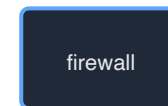
---

# DROP vs REJECT

DROP — silent void



REJECT — ICMP unreachable



## DROP

Packet is silently discarded. The sender gets no response and must wait for a timeout. Harder to fingerprint your firewall, but slower failure for legitimate clients.

## REJECT

Packet is discarded AND an ICMP "port unreachable" (or TCP RST) is sent back. Sender knows immediately the port is closed — faster failure, but reveals firewall presence.

```
iptables -A INPUT -p tcp --dport 23 -j DROP
```

```
iptables -A INPUT -p tcp --dport 23 -j REJECT \  
--reject-with tcp-reset
```

---

# nftables — what changed and why

nftables replaced iptables as the recommended Linux firewall tool starting with kernel 3.13 (2014), and became the default in Debian 10 / RHEL 8.

- **Single binary** — `nft` replaces four tools: `iptables`, `ip6tables`, `arptables`, `ebtables`. One syntax for IPv4, IPv6, ARP, and bridge filtering.
- **Native sets** — you can match against a set of IPs, ports, or prefixes with a single rule instead of a loop of rules: `ip saddr @blocked_ips drop`. Performance scales sub-linearly.
- **Atomic reloads** — the entire ruleset is loaded as one transaction: `nft -f rules.nft`. No window where partial rules leave the machine exposed.
- **Cleaner syntax** — rules read more like prose; the table/chain/rule hierarchy is explicit and user-defined rather than baked into the tool.

## Quick start

```
# List all tables and rules
nft list ruleset
```

```
# Load a complete ruleset from a file (atomic)
nft -f /etc/nftables.conf
```

---

# iptables vs nftables — side by side

| Task                             | iptables                                                          | nftables                                                    |
|----------------------------------|-------------------------------------------------------------------|-------------------------------------------------------------|
| Accept SSH (port 22)             | <code>iptables -A INPUT -p tcp --dport 22 -j ACCEPT</code>        | <code>tcp dport 22 accept</code>                            |
| Drop all ICMP                    | <code>iptables -A INPUT -p icmp -j DROP</code>                    | <code>ip protocol icmp drop</code>                          |
| NAT MASQUERADE                   | <code>iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE</code> | <code>ip saddr 10.0.0.0/8 oifname "eth0" masquerade</code>  |
| Match a set of IPs               | Requires multiple <code>-A</code> rules or ipset module           | <code>ip saddr { 192.168.1.0/24, 10.0.0.0/8 } accept</code> |
| Reload entire ruleset atomically | Not natively supported; use iptables-restore                      | <code>nft -f /etc/nftables.conf</code>                      |
| IPv6 support                     | Separate <code>ip6tables</code> command                           | Same ruleset; use <code>inet</code> table for dual-stack    |

Both tools write rules into the same kernel netfilter subsystem — they can coexist but may conflict; pick one per production system.

---

# Migration & gotchas

- `iptables-translate` — converts individual iptables rules to nftables syntax. Run `iptables-translate -A INPUT -p tcp --dport 22 -j ACCEPT` to get the equivalent nft line. Not 100% coverage, but a great starting point.
  - **Rule order matters** — iptables uses first-match; nftables uses first-match by default but supports `policy accept/drop` per chain. Moving from one to the other can silently change behaviour if chain policies differ.
  - **Default-allow vs default-deny** — fresh installations often have an ACCEPT policy on all chains. Always audit with `iptables -L -v -n` or `nft list ruleset` before assuming a host is hardened.
  - **conntrack on UDP** — UDP is stateless at the protocol level, but conntrack still tracks UDP "connections" (source/dest 4-tuple). Rules like `-m conntrack --ctstate ESTABLISHED` work for DNS, QUIC, and other UDP protocols.
  - **Persistence** — iptables rules are lost on reboot unless saved with `iptables-save` / `iptables-restore` or via `netfilter-persistent`. nftables rules are loaded from `/etc/nftables.conf` by systemd on boot if the service is enabled.
-

# Recap

- **Netfilter is the kernel engine** — iptables and nftables are just front-end tools that write rules into it. Five hooks (PREROUTING → INPUT / FORWARD → OUTPUT → POSTROUTING) define where rules can intercept packets.
- **Conntrack makes rules smarter** — stateful matching (`-m conntrack --ctstate ESTABLISHED,RELATED`) lets you write a single inbound ACCEPT rule instead of one per response port, and is the foundation of NAT.
- **nftables is the future, but iptables is still everywhere** — understand both. Use `iptables-translate` to convert existing rules and `nft list ruleset` to audit nftables configuration.

## Lab time

In the lab you will: add, list, and delete iptables rules on a live VM; observe conntrack state transitions with `conntrack -L`; write an equivalent nftables ruleset; and test DROP vs REJECT behaviour with `netcat`.

**Coming up:** Lesson 02 — How Linux Decides Where Packets Go (routing tables, ip route, policy routing, and Linux as a router).

---

# 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.

+ Send TCP SYN (creates NEW)

Complete handshake (→ ESTABLISHED)

Clear all

| Src → Dst                | State | TTL (s) |
|--------------------------|-------|---------|
| conntrack table is empty |       |         |

## 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

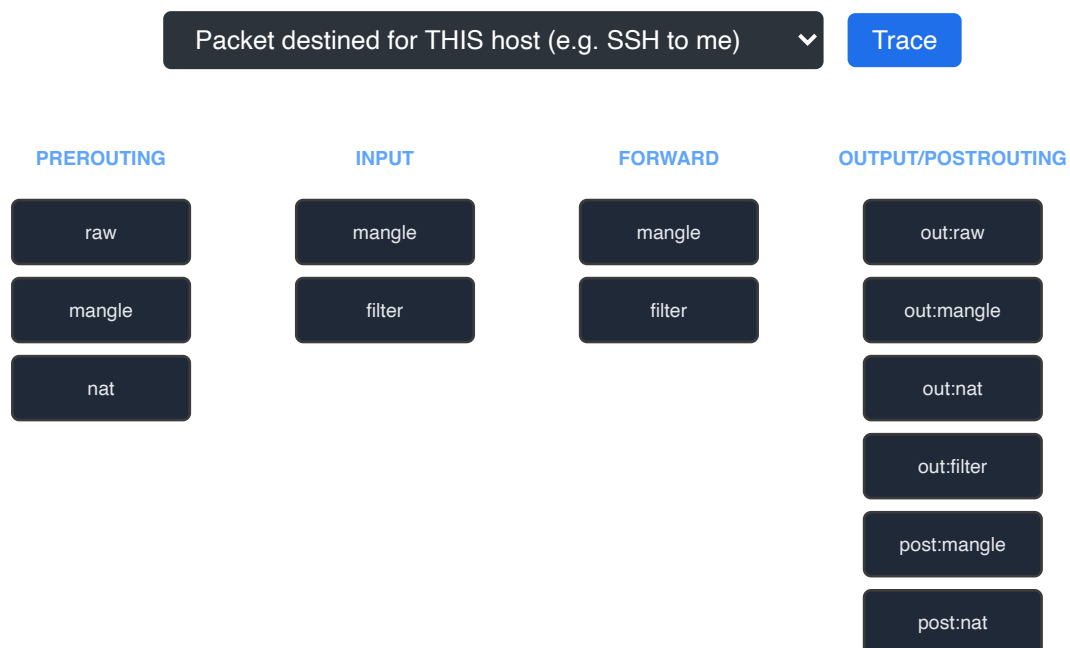
Click "Reload" to see the difference.

## 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.

# Lab — iptables and nftables, hands-on

Lesson 01 · ~10 min · Single host, no extra VMs

## Warning

Every command in this lab requires `sudo` and modifies your live firewall. Run on a throwaway VM if you can. To undo everything, `iptables -F` and `nft flush ruleset`.

## Step 1 — Read your current state

Before you change anything, see what's there:

```
sudo iptables -L -v -n
sudo iptables -t nat -L -v -n
```

Most distros ship with empty default chains and policy `ACCEPT`.

## Step 2 — Drop ICMP, watch ping die

```
sudo iptables -A INPUT -p icmp -j DROP
ping -c 3 127.0.0.1
# Result: 100% packet loss
```

## Step 3 — Remove the rule, ping works again

```
sudo iptables -D INPUT -p icmp -j DROP
ping -c 3 127.0.0.1
# Result: 0% packet loss
```

## Step 4 — Same scenario in nftables

---

```
sudo nft add table inet f
sudo nft 'add chain inet f input { type filter hook input priority 0 ; }'
sudo nft add rule inet f input ip protocol icmp drop
ping -c 3 127.0.0.1
# Result: dropped

sudo nft flush ruleset
ping -c 3 127.0.0.1
# Result: works again
```

## Step 5 — Translate iptables to nftables

---

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

### Cleanup

`sudo iptables -F && sudo nft flush ruleset` resets you to a clean slate.

## Self-check

---

- ✓ You can read the iptables rules table.
- ✓ You can add and remove a rule and observe its effect.
- ✓ You have written the same rule in both iptables and nftables.
- ✓ You have used `iptables-translate`.

# Homework — Lesson 01

~15 min · Two tasks · Bash only

## Task 1 — Allow only SSH and HTTP, drop everything else

Using **iptables**, write rules that:

1. Allow inbound SSH (TCP/22) from anywhere.
2. Allow inbound HTTP (TCP/80) from anywhere.
3. Allow established/related return traffic.
4. Drop everything else inbound.
5. Allow all loopback traffic.

### Self-check:

- `sudo iptables -L -v -n` shows your rules.
- From another host, `nmap -p 22,80,443 <your-ip>` shows 22 and 80 open, 443 filtered.
- Ping (ICMP) is blocked from outside.

## Task 2 — Translate to nftables and persist

Recreate the same firewall using **nftables**:

1. Create a table and an input chain hooked at filter priority 0.
2. Add the four rules from Task 1.
3. Save the ruleset: `sudo nft list ruleset > rules.nft`.

4. Flush, then reload from file: `sudo nft flush ruleset && sudo nft -f rules.nft`.

**Self-check:**

- `sudo nft list ruleset` shows the same effective rules as Task 1.
- The `rules.nft` file is human-readable and can be edited.

[View solutions →](#)

# Solutions — Lesson 01

Worked answers for the homework tasks

Try the homework yourself before reading these. The reasoning matters more than the exact syntax.

## Task 1 — Solution (iptables)

```
# Allow loopback (FIRST — locking out localhost breaks lots of services)
sudo iptables -A INPUT -i lo -j ACCEPT
sudo iptables -A OUTPUT -o lo -j ACCEPT

# Allow ESTABLISHED,RELATED — replies to your outbound traffic come back
sudo iptables -A INPUT -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT

# Allow inbound SSH and HTTP
sudo iptables -A INPUT -p tcp --dport 22 -m conntrack --ctstate NEW -j ACCEPT
sudo iptables -A INPUT -p tcp --dport 80 -m conntrack --ctstate NEW -j ACCEPT

# Default-deny anything else
sudo iptables -P INPUT DROP
sudo iptables -P FORWARD DROP
```

### Why this order matters

- **Loopback first** — many local services bind to `127.0.0.1`. If you set `INPUT` policy to DROP without an explicit accept rule on `lo`, you'll break DNS, package managers, and SSH itself.
- **ESTABLISHED before specific accepts** — every TCP connection's reply packets count as ESTABLISHED. Accepting them globally is faster (single rule) and simpler than thinking through every port.
- **Default-deny is set last** — if you set the default policy first, you'll lose your SSH session before adding the SSH rule.

## Common mistake

Setting `iptables -P INPUT DROP` before adding the SSH accept rule. If you're SSH'd in remotely, your session dies. Always add accept rules *first*.

## Task 2 — Solution (nftables)

```
# Create table and hooked chain
sudo nft add table inet filter
sudo nft 'add chain inet filter input { type filter hook input priority 0 ; po

# Allow loopback
sudo nft add rule inet filter input iif lo accept

# Allow established/related
sudo nft add rule inet filter input ct state established,related accept

# Allow new SSH and HTTP
sudo nft add rule inet filter input tcp dport 22 ct state new accept
sudo nft add rule inet filter input tcp dport 80 ct state new accept

# Save
sudo nft list ruleset > rules.nft
```

## Reload pattern

```
sudo nft flush ruleset
sudo nft -f rules.nft
sudo nft list ruleset    # confirm
```

## Format of `rules.nft`

```
table inet filter {
    chain input {
        type filter hook input priority filter; policy drop;
        iif "lo" accept
        ct state established,related accept
        tcp dport 22 ct state new accept
    }
}
```

```
    tcp dport 80 ct state new accept
}
}
```

### One bonus

nftables lets you set `policy drop` on the chain definition itself — no separate `-P` command needed. That's another concrete win over iptables.

## **Lesson 02: How Linux Decides Where Packets Go**

# Lesson 02 — How Linux Decides Where Packets Go

Every packet that enters or leaves a Linux machine must be routed. The kernel consults a set of rules and routing tables to decide the next hop — or drops the packet if no match exists.

- **Routing table** — the primary data structure: prefix → gateway/interface
- **Longest-prefix match** — most specific route always wins
- **Policy routing** — multiple tables, selected by `ip rule`
- **Netfilter integration** — routing decision splits traffic to INPUT vs FORWARD

**Notes:** This lesson builds directly on Lesson 01's netfilter flow. By the end students will understand every decision the kernel makes before a packet reaches a socket or hits a forwarding chain.

# What Does "Routing" Mean?

For every packet, the kernel asks one question:

"Given this destination IP, which interface do I send out, and who is the next hop?"

- **Locally originated packets** — app writes to socket → kernel routes to outbound interface
- **Forwarded packets** — arrive on interface A, routing decides interface B to send them on
- **Locally destined packets** — routing detects the destination belongs to this host → delivers to a socket
- **No match?** — packet is dropped (or ICMP unreachable is sent)

The kernel resolves this question by walking the **routing table** and applying **longest-prefix-match**.

**Notes:** Emphasise that routing is a purely kernel-space operation. User-space only configures the tables; the kernel does the per-packet lookup at wire speed.

---

# The Routing Table, Line-by-Line

```
$ ip route show
default          via 192.168.1.1 dev eth0    proto dhcp  metric 100
10.0.5.0/24      via 10.0.5.1   dev eth1    proto static
10.0.0.0/16      via 10.0.1.1   dev eth1    proto static
192.168.1.0/24   dev eth0       proto kernel scope link    src 192.168.1.50
```

Each row: **prefix** → **gateway** (via) + **interface** (dev). The `metric` breaks ties between equal-length prefixes. `scope link` means directly reachable — no gateway needed.

Destination 10.0.5.42 arrives — which row wins?

**10.0.5.42**

default via 192.168.1.1 /0 (least specific)

10.0.0.0/16 via 10.0.1.1 /16

10.0.5.0/24 via 10.0.5.1 /24 (more specific — winner!)

10.0.6.0/24 via 10.0.6.1 /24 (doesn't match)

Linux walks the table by specificity (longest prefix first). The first match that covers the destination wins.

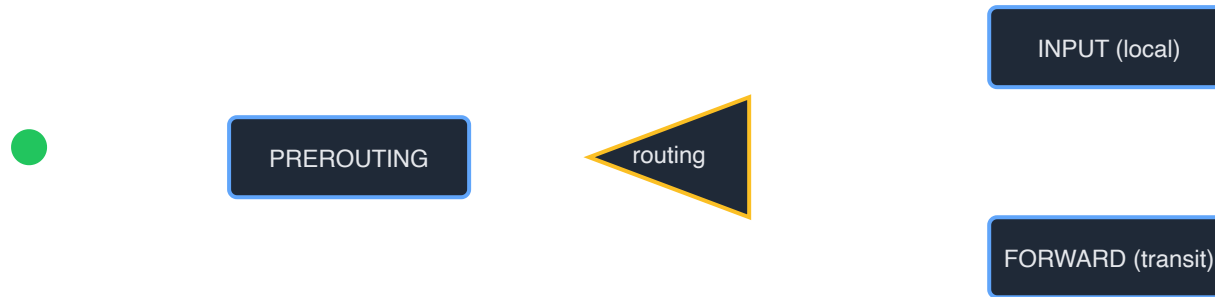


# Routing in the Netfilter Flow

Recall from Lesson 01: packets traverse netfilter hooks in a fixed order. The **routing decision** sits between PREROUTING and the fork to INPUT or FORWARD.

- **Destination = this host** → routing sends packet UP to `INPUT` chain
- **Destination = elsewhere** → routing sends packet ACROSS to `FORWARD` chain
- DNAT in PREROUTING can change the destination *before* the routing decision

Two packets enter — routing forks them by destination.



# Default / Local / Link Routes

The Linux kernel maintains several built-in route types that you will see in every routing table:

| Type      | Example                                         | Meaning                                           | Added by                                   |
|-----------|-------------------------------------------------|---------------------------------------------------|--------------------------------------------|
| link      | <code>192.168.1.0/24 dev eth0 scope link</code> | Directly reachable — ARP, no gateway needed       | Kernel on <code>ip addr add</code>         |
| local     | <code>local 192.168.1.50 dev eth0</code>        | Address belongs to this host → deliver to socket  | Kernel on <code>ip addr add</code>         |
| default   | <code>default via 192.168.1.1 dev eth0</code>   | Last-resort gateway; prefix /0 matches everything | DHCP client or admin                       |
| static    | <code>10.10.0.0/24 via 10.0.0.1 dev eth1</code> | Manually configured persistent route              | <code>ip route add</code> / network config |
| blackhole | <code>blackhole 192.0.2.0/24</code>             | Silently discard matching packets                 | Admin (security / null routing)            |

**Notes:** The `local` table is a separate routing table (table 255) that the kernel always consults first. Students can inspect it with `ip route show table local`.

---

# Adding & Deleting Routes

Use `ip route` to manage routes at runtime. Changes take effect immediately but are **not persistent** across reboots unless written to network configuration files.

```
# Add a static route for a subnet via a specific gateway
sudo ip route add 10.0.0.0/24 via 192.168.1.1 dev eth0

# Add with a metric (lower = preferred)
sudo ip route add 10.0.0.0/24 via 192.168.1.1 dev eth0 metric 50

# Add a blackhole route (silently drop)
sudo ip route add blackhole 192.0.2.0/24

# Delete a route
sudo ip route del 10.0.0.0/24

# Replace an existing route atomically
sudo ip route replace 10.0.0.0/24 via 10.0.0.254 dev eth1
```

**Persistence tip**

On systemd-networkd: add a `[Route]` section to `/etc/systemd/network/*.network`. On NetworkManager: use `nmcli` or the GUI.

---

# Multiple Routing Tables

Linux supports up to **252 user-defined routing tables** (plus built-ins: `local`, `main`, `default`). Named tables are registered in `/etc/iproute2/rt_tables`.

```
# /etc/iproute2/rt_tables
#
# Reserved values
255    local
254    main
253    default
0      unspec
#
# Local
100    isp1
200    isp2
300    vpn
```

- **main** (table 254) — the default table; what `ip route show` displays
- **local** (table 255) — kernel-managed; contains host and broadcast addresses
- **Custom tables** — selected by `ip rule` based on packet attributes

```
# List all routes in a specific table
ip route show table isp1

# Add a route into a named table
sudo ip route add default via 203.0.113.1 table isp1
```

**Notes:** Multi-table routing is the foundation of ISP dual-homing, VPN split routing, and container networking. The policy routing engine (next slide) is what decides which table to use.

---

# Policy Routing with `ip rule`

Before consulting any routing table, the kernel walks the **rule list** (sorted by priority). The first matching rule names the table to use.

```
# View current rules (lower priority number = checked first)
ip rule show

# Route traffic from subnet 10.10.0.0/24 via table 100
sudo ip rule add from 10.10.0.0/24 lookup 100

# Route by fwmark (set by iptables/nftables MARK target)
sudo ip rule add fwmark 0x10 lookup 200

# Route by destination
sudo ip rule add to 8.8.0.0/16 lookup vpn

# Delete a rule
sudo ip rule del from 10.10.0.0/24 lookup 100
```

Packet has src=10.10.0.5 — walk the rule list, first match selects a table.

src=10.10.0.5

0: from all lookup local

100: from 192.168.0.0/16 lookup 100

200: from 10.10.0.0/24 lookup 200 ← matches!

32766: from all lookup main (default)

32767: from all lookup default

---

# Network Namespaces — Your Free Lab

Network namespaces give each namespace its own routing table, interfaces, and firewall rules.

You can build a full multi-router topology with zero hardware.

```
# Create two namespaces
ip netns add red
ip netns add blue

# Create a veth pair (virtual Ethernet cable)
ip link add veth0 type veth peer name veth1

# Move one end into each namespace
ip link set veth0 netns red
ip link set veth1 netns blue

# Assign addresses inside each namespace
ip netns exec red ip addr add 10.0.0.1/24 dev veth0
ip netns exec blue ip addr add 10.0.0.2/24 dev veth1
```

**Lab preview**

Today's lab uses exactly this pattern to build a two-namespace topology and test policy routing between them.

---

# Diagnostic Toolkit

These three tools answer the most common routing questions in production:

```
# ip route get — show EXACTLY which route the kernel will use for a destination
ip route get 8.8.8.8
# 8.8.8.8 via 192.168.1.1 dev eth0 src 192.168.1.50 uid 1000

# Also test from a specific source (policy routing aware)
ip route get 8.8.8.8 from 10.10.0.5

# traceroute — show each hop to a destination
traceroute 8.8.8.8
# Uses UDP by default; add -T for TCP, -I for ICMP

# mtr — combines traceroute + continuous ping; great for diagnosing flapping
mtr --report 8.8.8.8
mtr --report --tcp --port 443 8.8.8.8
```

**ip route get**

**mtr**

Fastest first check — tells you *exactly* what the kernel decides, including which rule matched and which table was consulted.

Real-time hop-by-hop latency and packet loss. Invaluable for spotting asymmetric routing or a bad transit link.

---

# Common Gotchas

- **Asymmetric routing** — packets leave via interface A, replies arrive via interface B. Stateful firewalls and `rp_filter` will silently drop the returning traffic. Fix: ensure return routes mirror outbound paths, or tune `rp_filter=2` (loose mode).
- **Missing return route** — you can reach a remote host but it cannot reply. Classic cause: you added a route on the client but forgot the server's default gateway.
- `rp_filter` (**Reverse Path Filter**) — kernel drops packets whose source IP would not be reachable on the same interface. Check: `sysctl net.ipv4.conf.all.rp_filter`. Set to `0` to disable, `2` for loose validation.
- **Metric collisions** — two routes with the same prefix and metric cause non-deterministic selection (ECMP unless configured). Always set explicit metrics.
- **DNAT vs routing order** — DNAT in PREROUTING fires *before* the routing decision. If you DNAT a packet to a new destination, the routing lookup uses the *new* destination address.
- **IPv6 surprises** — `ip -6 route` is a separate table. A working IPv4 route does not imply a working IPv6 route.

**Notes:** The `rp_filter` gotcha is extremely common in multi-homed servers and VPN setups. Worth a dedicated

live demo.

---

# Recap & Next Steps

- **Routing decision** — kernel asks "which interface + next hop?" for every packet
- **Longest-prefix match** — most specific prefix always wins; metric breaks ties
- **Routing in netfilter** — sits between PREROUTING and INPUT/FORWARD fork
- **Multiple tables** — `local`, `main`, `default`, plus up to 252 user-defined
- **Policy routing** — `ip rule` selects the table by src IP, dst IP, fwmark, or interface
- **Namespaces** — each has its own isolated routing universe; perfect for lab work
- **Diagnostics** — `ip route get` for instant answer; `traceroute` / `mtr` for path analysis

**Coming up:** Apply L1 + L2 together in real scenarios. Lesson 03 covers NAT — how MASQUERADE, SNAT, and DNAT interact with the routing table you now understand. The lab combines netfilter rules with policy routing for a practical dual-ISP setup.



# How Linux Decides Where Packets Go

Lesson 02 · Linux Networking · ~10 min read

## The question Linux asks for every packet

---

Every time your machine generates an outbound packet — whether it comes from a browser, a `curl` command, or a container — the kernel faces the same two-part question: *Which interface should carry this packet out?* and *Should I forward it through a gateway, or is the destination directly reachable on this link?* This decision is routing, and it happens entirely inside the kernel, long before anything touches the wire.

Routing is separate from filtering. In Lesson 1 we saw that netfilter hooks can accept or drop packets; routing happens a step earlier, at the `PREROUTING` and `OUTPUT` hooks, before the `FORWARD` vs `INPUT` split. If the kernel cannot find any route for a destination, the packet is silently discarded with an ICMP "Network unreachable" returned to the sender. No firewall rule is even consulted — the packet never made it that far.

Understanding routing is therefore foundational: a misconfigured route can look exactly like a firewall block, and the fastest way to distinguish them is `ip route get <dst>`, which asks the kernel directly what it would do. We will use that command throughout this lesson.

## Reading the routing table

---

Run `ip route show` on almost any Linux host and you will see output like this:

```
default via 192.168.1.1 dev eth0
10.0.5.0/24 via 10.0.5.1 dev veth-a
10.0.0.0/16 via 10.0.0.1 dev eth1
192.168.1.0/24 dev eth0 proto kernel scope link src 192.168.1.50
```

Each line is a route. A route says: "For packets whose destination matches this network prefix, send them out of this device (`dev`), optionally through this next-hop gateway

(`via`)." The kernel compares the destination IP of every outbound packet against every prefix in the table and picks the *most specific* match — the one with the longest prefix length. A `/24` beats a `/16` beats a `/0` (default). This algorithm is called **longest-prefix match**.

The simulator below lets you type any destination IP and watch the kernel's decision unfold. Each matching route turns green; the winner — the longest-prefix match — is outlined in amber.

Destination:

default via 192.168.1.1 dev eth0

10.0.5.0/24 via 10.0.5.1 dev veth-a

10.0.0.0/16 via 10.0.0.1 dev eth1

192.168.1.0/24 dev eth0 proto kernel scope link src 192.168.1.50

Winner: 10.0.5.0/24 via 10.0.5.1 dev veth-a (longest-prefix /24)

## Default, local, and link routes

---

Three flavors of route appear in almost every Linux routing table, and it is worth understanding what each one means before moving on to more advanced topics.

A **link route** (also called a "connected" route) is added automatically by the kernel whenever you assign an IP address to an interface. If you give `eth0` the address `192.168.1.50/24`, the kernel immediately inserts a route for `192.168.1.0/24 dev eth0 scope link`. This tells the kernel that all addresses in that subnet are directly reachable — no gateway needed, just ARP for the MAC address and send the frame. You will recognize link routes in `ip route show` by the absence of a `via` keyword and the presence of `scope link`.

A **local route** lives in a separate, special routing table (table 255, called `local`) and covers your own IP addresses. When a packet arrives destined for `192.168.1.50` — your own address — the kernel matches the local table first and delivers the packet to a local socket rather than forwarding it. You rarely need to touch this table manually.

A **default route** is simply the route for `0.0.0.0/0` — the prefix of length zero that matches every IP address. Because longest-prefix match always prefers more specific routes, the default route only wins when nothing else matches. It says "if you have no idea where to send this, forward it to my gateway." A host without a default route can only reach networks it has an explicit route for; internet access requires a default route pointing at an upstream gateway.

When you run `ip route add` or `ip route del`, you are directly manipulating the kernel's Forwarding Information Base (FIB). The change takes effect immediately — there is no service to restart, no cache to flush.

## Multiple routing tables — and how to pick one

---

Linux does not limit you to one routing table. The kernel supports up to 252 user-defined tables (numbered 1–252, plus the three built-in tables: `local`, `main`, and `default`). By default, every route you add goes into the `main` table (table 254), and the kernel only consults that one table. But you can configure it to consult different tables based on the source IP, the incoming interface, or a firewall mark — this is called **policy routing**.

The mechanism is `ip rule`. Rules are evaluated in priority order (lower number = higher priority). Each rule says: "if this packet matches this selector, look it up in this routing table." For example:

```
ip rule add from 10.10.0.0/24 lookup 100
ip rule add from 10.10.0.201/29 lookup 100
```

A common use case is a host with two uplinks (say, a corporate VPN and a direct internet connection) where traffic from the management subnet must always exit via the VPN, while regular traffic takes the direct route. You put the VPN default in table 100, the direct default in table 200, and use `ip rule` to steer traffic from each source subnet to the right table. Without policy routing this would be impossible to express in a single routing table.

The widget below visualizes a three-table setup. Drag the slider to change the source IP's last octet and watch the kernel pick the appropriate routing table.



## Network namespaces — building a topology on one host

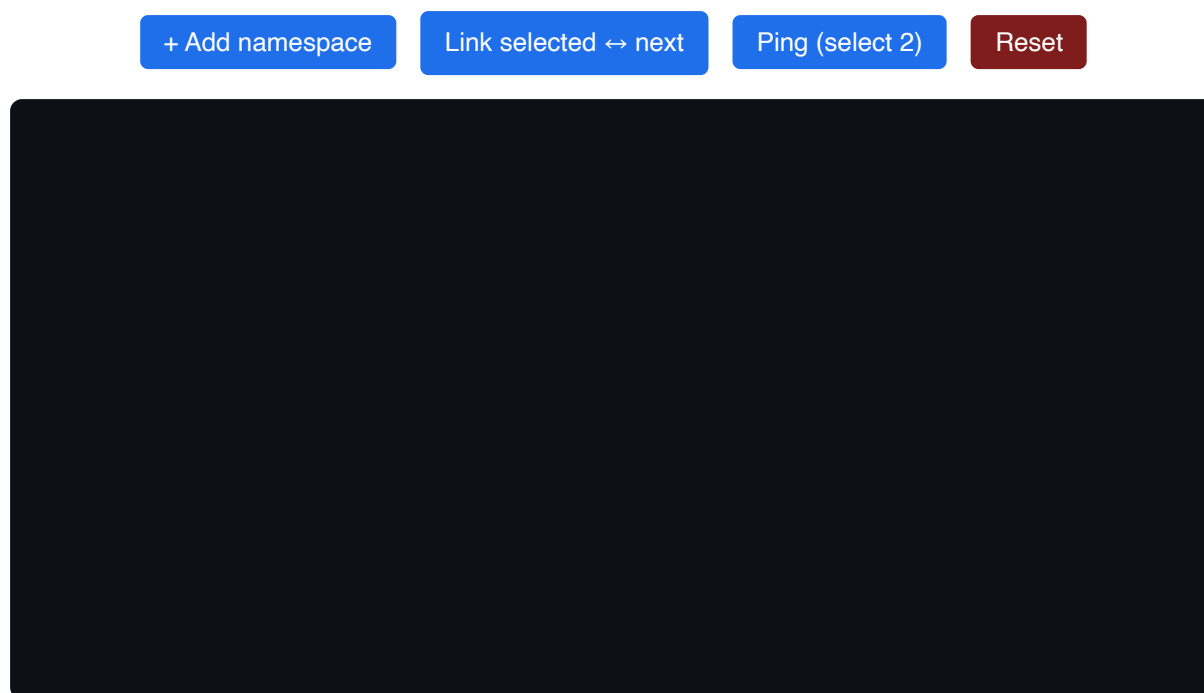
A network namespace is a complete, isolated copy of the Linux networking stack: its own interfaces, its own routing table, its own iptables rules, its own sockets. Processes inside one namespace cannot see or affect the network state of another namespace. The default namespace is simply the one your shell runs in; everything you have seen so far lives there.

Namespaces are not just for containers. They are the standard tool for testing routing topologies on a single developer laptop. Want to simulate a two-router setup? Create three namespaces (client, router, server), connect them with virtual Ethernet pairs (`veth`), assign IPs, enable IP forwarding in the router namespace, and you have a fully functional topology — all without any virtualization overhead, and all gone on reboot.

The essential commands are simple. `ip netns add red` creates a namespace called `red`. `ip netns exec red <cmd>` runs a command inside it. A `veth` pair is a pair of connected virtual interfaces: anything sent into one end comes out the other, as if they were wired together. You create the pair in the default namespace and then move each end into the appropriate namespace with `ip link set <iface> netns <name>`.

One subtle but important point: IP forwarding is per-namespace. Even if the host has `net.ipv4.ip_forward=1` set, a namespace that is meant to act as a router must have forwarding enabled *inside that namespace* via `ip netns exec <router-ns> sysctl -w net.ipv4.ip_forward=1`. Forgetting this is the most common reason a three-namespace topology fails silently.

Use the topology builder below to add namespaces, link them together, and simulate a ping. Namespaces without a veth link between them have no route to each other — the ping will fail with a red cross.



Click "Add namespace" to start.

## Diagnostic toolkit

---

When a packet seems to be going nowhere, the first three tools to reach for are `ip route get`, `traceroute`, and `mtr`. Each answers a different question.

`ip route get <dst>` asks the kernel "what would you actually do with a packet bound for this destination?" It respects policy routing rules and all routing tables, returning the exact interface, gateway, and source address the kernel would use. It is instantaneous and requires no network traffic. If `ip route get` fails or returns the wrong gateway, the problem is in your routing configuration — no firewall investigation needed yet.

`tracert` (or `tracert6` for IPv6) sends probes with incrementing TTL values and listens for ICMP Time Exceeded responses, building a hop-by-hop map from your host to the destination. It is invaluable for pinpointing where in the network path traffic is being dropped or misdirected. Be aware that some routers silently discard ICMP; a missing hop in the output does not always mean the path is broken — only that a particular router chose not to respond.

`mtr` combines `tracert` and `ping` into a live, continuously updating display. It shows per-hop packet loss and round-trip time in real time, making it far easier to spot intermittent problems that a single `tracert` run would miss. Run `mtr --report <dst>` to produce a one-shot summary suitable for pasting into a bug report.

**If a packet seems to be going nowhere, run `ip route get <dst>` first.** It answers exactly what the kernel would do — saving you from guessing.

## Common gotchas

A few mistakes appear so often in routing configurations that they are worth naming explicitly.

**Asymmetric routing** occurs when the path from A to B takes different interfaces than the path from B to A. This is legal in IP but can break stateful inspection: a firewall that only saw the outbound half of a TCP connection will reject the inbound reply as "INVALID" in conntrack terms. The fix is either to make routing symmetric or to tune conntrack not to track certain flows.

**Missing return route** is the single most common reason a ping "works one way." If host A can reach host B but B has no route back to A's subnet, B's ICMP echo-reply has nowhere to go and is dropped. Always check both directions. A quick test: after you confirm A can reach B, run `ip route get <A's IP>` from B's perspective (or from within B's namespace).

**Reverse path filtering** (`rp_filter`) is a kernel feature that drops packets arriving on an interface if the kernel's routing table says the source IP should have arrived on a different interface. This prevents certain spoofing attacks but can cause confusion in asymmetric or multi-homed setups. Check the value with `sysctl net.ipv4.conf.all.rp_filter`; set to `0` to disable or `2` for loose mode.

**Metric collisions** happen when two routes to the same prefix have the same metric (cost). The kernel picks one — which one is implementation-defined. If you add a failover route, give it a higher metric than the primary so the primary always wins when both are present.

## Recap

---

- Linux looks up an interface + gateway for every outbound packet.
- The routing table is consulted by longest-prefix match.
- Multiple routing tables exist; `ip rule` chooses between them.
- Network namespaces let you build multi-router topologies on a single host.
- `ip route get` answers "what would the kernel do?" — your first diagnostic.

# Lab — Linux routing, hands-on

Lesson 02 · ~12 min · Single host, uses ip netns

This lab modifies kernel routing. Run on a throwaway VM if you can. All namespaces and veth pairs vanish on reboot.

## Step 1 — Read your current routing table

```
ip route show
ip route show table all      # all tables
ip rule show
```

## Step 2 — Ask the kernel a "what would you do?" question

```
ip route get 8.8.8.8
# Example output: 8.8.8.8 via 192.168.1.1 dev eth0 src 192.168.1.50 ...
```

## Step 3 — Add a bogus route, watch the answer change

```
sudo ip route add 10.20.0.0/16 via 192.168.1.254 dev eth0
ip route get 10.20.5.5      # should now resolve via the new route
sudo ip route del 10.20.0.0/16
```

## Step 4 — Build a 2-namespace topology

```
# Create two namespaces
sudo ip netns add red
sudo ip netns add blue
```

```
# Connect them with a veth pair
sudo ip link add veth-red type veth peer name veth-blue
sudo ip link set veth-red netns red
sudo ip link set veth-blue netns blue

# Assign IPs and bring up
sudo ip -n red addr add 10.10.0.1/24 dev veth-red
sudo ip -n blue addr add 10.10.0.2/24 dev veth-blue
sudo ip -n red link set veth-red up
sudo ip -n blue link set veth-blue up
sudo ip -n red link set lo up
sudo ip -n blue link set lo up

# Ping across!
sudo ip netns exec red ping -c 3 10.10.0.2
```

## Step 5 — Add a policy rule, observe table selection

---

```
# Create a separate routing table (just memory – no /etc edits needed)
sudo ip route add default via 10.10.0.2 dev veth-red table 100
sudo ip rule add from 10.10.0.0/24 lookup 100

# Verify
sudo ip rule show
sudo ip route show table 100
```

## Cleanup

---

```
sudo ip netns del red
sudo ip netns del blue
sudo ip rule del from 10.10.0.0/24 lookup 100
sudo ip route flush table 100
```

## Self-check

---

- ✓ You can read the routing table and a policy rule list.
- ✓ You have used `ip route get` to ask the kernel.
- ✓ You have built and pinged across a two-namespaces topology.

- ✓ You have added an `ip rule` entry that selects a non-main routing table.

# Homework — Lesson 02

~20 min · Two tasks · Uses ip netns

## Task 1 — Static route to a phantom network

Write commands to:

1. Add a route to network `10.20.0.0/16` via gateway `192.168.1.254` through interface `eth0`.
2. Verify the kernel will use it for destination `10.20.5.5` by running `ip route get`.
3. Remove the route.

### Self-check:

- `ip route get 10.20.5.5` outputs a line containing `via 192.168.1.254` while the route is present.
- After `ip route del`, the same command resolves through the default route.

## Task 2 — Three-namespace topology (client → router → server)

Build the following on a single host:

```
ns-client (10.0.1.10) —veth0/veth1— ns-router (10.0.1.1 + 10.0.2.1)
```

1. Create three namespaces: `ns-client`, `ns-router`, `ns-server`.

2. Create two veth pairs and assign one end to each pair of adjacent namespaces.
3. Assign IPs as shown. Bring all interfaces up.
4. Enable IPv4 forwarding inside `ns-router`: `sysctl -w net.ipv4.ip_forward=1`.
5. Add default routes in `ns-client` and `ns-server` pointing to the router.
6. Ping `ns-server` from `ns-client`.
7. While the ping is running, in a second terminal capture on the router: `sudo ip netns exec ns-router tcpdump -i any icmp`.

**Self-check:**

- The ping succeeds with 0% loss.
- The tcpdump shows ICMP echo requests in *both* directions: from veth1 (client side) into the router AND out veth2 (server side) — and the corresponding replies coming back. If you only see one direction, you forgot the return route.

[View solutions →](#)

# Solutions — Lesson 02

Worked answers for the homework tasks

Try the homework first. Networking labs reward muscle memory.

## Task 1 — Solution

```
sudo ip route add 10.20.0.0/16 via 192.168.1.254 dev eth0
ip route get 10.20.5.5
# Expected: 10.20.5.5 via 192.168.1.254 dev eth0 src 192.168.1.50 ...
sudo ip route del 10.20.0.0/16
ip route get 10.20.5.5
# Expected: now resolves via the default route
```

## Task 2 — Solution (client → router → server)

```
# 1. Namespaces
sudo ip netns add ns-client
sudo ip netns add ns-router
sudo ip netns add ns-server

# 2. Two veth pairs
sudo ip link add veth0 type veth peer name veth1
sudo ip link add veth2 type veth peer name veth3

# Move ends into the right namespaces
sudo ip link set veth0 netns ns-client
sudo ip link set veth1 netns ns-router
sudo ip link set veth2 netns ns-router
sudo ip link set veth3 netns ns-server

# 3. IPs and link state
sudo ip -n ns-client addr add 10.0.1.10/24 dev veth0
```

```
sudo ip -n ns-router addr add 10.0.1.1/24 dev veth1
sudo ip -n ns-router addr add 10.0.2.1/24 dev veth2
sudo ip -n ns-server addr add 10.0.2.10/24 dev veth3
for ns in ns-client ns-router ns-server; do
    sudo ip -n $ns link set lo up
done
sudo ip -n ns-client link set veth0 up
sudo ip -n ns-router link set veth1 up
sudo ip -n ns-router link set veth2 up
```

## Common mistakes

- **Forgot to enable forwarding** — `net.ipv4.ip_forward=1` must be set *inside the router namespace*, not on the host. Without it, the router drops transit packets.
- **Forgot the return route** — `ns-server` needs a default route back to `10.0.2.1`. Without it, the echo-reply has nowhere to go and you'll see only one-way traffic in `tcpdump`.
- **Forgot to bring up lo** — required for many local sockets; safest to do it in every namespace.
- **Used the host's `ip route` instead of `ip -n ns-... route`** — easy slip; the host's table doesn't apply inside a namespace.

### Pro tip:

`ip netns exec ns-router bash` drops you into a shell where every `ip` command runs inside that namespace by default. Less ceremony than `-n ns-router` on every command.

## **Lesson 03: Policy-Based Routing**

# Lesson 03 — Policy-Based Routing

How Linux picks between multiple routing tables.

Linux Networking

**Notes:** This lesson builds on Lesson 02's routing table foundations. Students should already know `ip route`, longest-prefix-match, and the netfilter flow. Today we introduce the rule list that sits in front of every routing table lookup.

# The routing table is just one table

Lesson 02 showed how the kernel consults the **main** routing table for every packet. But Linux actually ships with three built-in tables:

- **local** (table 255) — kernel-managed; host addresses and broadcasts. Always consulted first.
- **main** (table 254) — what `ip route show` displays. Your static routes live here.
- **default** (table 253) — historically for postrouting; almost never populated today.

Plus up to **252 user-defined tables**, each with its own independent prefix set. The question is: *who decides which table to use?*

Answer: `ip rule`

A sorted list of *rules* maps packet attributes → table IDs, evaluated before any table lookup.

**Notes:** Quick recap slide — move through it fast if students remember Lesson 02. The key hook is the question at the end: "who decides which table?" That's what this whole lesson answers.

---

# Two lookups, not one

When the kernel needs to route a packet, it performs two distinct lookups in sequence:

## Step 1: `ip rule` picks a table

Walk the rule list by priority. The first matching rule names a routing table. If no user rule matches, fall through to `main` (priority 32766).

## Step 2: that table is consulted

Perform a normal longest-prefix-match lookup inside the selected table. If the table has no match, the rule list continues to the next rule (not necessarily `main`).

This two-step process lets you route **different traffic classes through entirely different forwarding paths** — without touching the main table at all.

**Notes:** Emphasise the asymmetry: Step 1 is a linear scan (short list,  $O(N)$ ), Step 2 is a trie lookup (large table,  $O(\log N)$ ). We'll see why this matters in slide 11.

# Anatomy of an `ip rule`

Every rule has three parts: a **priority**, a **selector**, and a **target table**.

```
ip rule add pref 100 from 10.10.0.0/24 lookup 200
```

- `pref 100` — priority; lower numbers are evaluated first
- `from 10.10.0.0/24` — selector: match packets whose *source* address falls in this subnet
- `lookup 200` — target: if this rule matches, consult routing table 200

Packet metadata: src=10.10.0.5 fwmark=0x42 iif=eth0

**src=10.10.0.5 fwmark=0x42 iif=eth0**

pref 0: from all lookup local

pref 50: from 192.168.0.0/16 lookup 50

pref 80: iif wlan0 lookup 80

pref 100: from 10.10.0.0/24 fwmark 0x42 lookup 200 ← match!

pref 32766: from all lookup main

---

# Selectors at a glance

A single rule can combine multiple selectors — all must match for the rule to fire.

| Selector            | Example                        | Matches when...                                                                       |
|---------------------|--------------------------------|---------------------------------------------------------------------------------------|
| <code>from</code>   | <code>from 10.10.0.0/24</code> | Source address is within the given prefix                                             |
| <code>to</code>     | <code>to 8.8.0.0/16</code>     | Destination address is within the given prefix                                        |
| <code>fwmark</code> | <code>fwmark 0x42</code>       | Packet carries the given netfilter mark (set by iptables/nftables <code>MARK</code> ) |
| <code>iif</code>    | <code>iif eth1</code>          | Packet arrived on the named interface                                                 |
| <code>oif</code>    | <code>oif eth0</code>          | Packet is leaving on the named interface (locally-originated traffic)                 |
| <code>tos</code>    | <code>tos 0x10</code>          | IP Type of Service / DSCP field matches                                               |

**Notes:** `fwmark` is the most powerful — it lets netfilter "tag" packets with arbitrary 32-bit values before the routing decision. This is the key to VPN policy routing, QoS, and split tunnels.

---

# Use case 1 — Source-based routing

A server has two uplinks (eth0 → ISP-A, eth1 → ISP-B). Traffic from subnet `10.10.0.0/24` must always leave via ISP-B, regardless of the destination.

```
# Table 200 gets its own default route via ISP-B
ip route add default via 203.0.113.1 dev eth1 table 200

# Rule: any packet sourced from 10.10.0.0/24 uses table 200
ip rule add from 10.10.0.0/24 lookup 200 pref 100

# All other traffic falls through to table main → ISP-A
ip route add default via 198.51.100.1 dev eth0
```

## Why not just change the main table?

Because a single default route can't express "use this gateway *for some sources* and that gateway *for others*." PBR handles this elegantly without NAT or complex iptables rules.

**Notes:** Classic ISP dual-homing scenario. Works for outbound traffic. Inbound traffic is handled by BGP or DNS-based load balancing — out of scope for this lesson.

---

## Use case 2 — fwmark-driven routing

Netfilter's `MARK` target paints a 32-bit tag onto a packet in the `mangle` table. The `ip rule fwmark` selector then routes by that tag.

```
# Mark packets from VPN clients with 0x42
iptables -t mangle -A PREROUTING -i tun0 -j MARK --set-mark 0x42

# Route all marked packets through table 200
ip rule add fwmark 0x42 lookup 200 pref 100

# Table 200's default: force through the secondary WAN
ip route add default via 10.99.0.1 dev eth1 table 200
```

Netfilter MANGLE paints a mark, ip rule routes by the mark



# Persisting marks across a flow — CONNMARK

**MARK paints one packet. CONNMARK makes the mark sticky.**

CONNMARK saves a mark to the conntrack entry, so every subsequent packet in the flow inherits it — even before the mangle rule fires again.

```
# Three-rule mangle chain for CONNMARK
# 1. Restore saved mark for ESTABLISHED/RELATED packets
iptables -t mangle -A PREROUTING -m conntrack --ctstate ESTABLISHED,RELATED \
-j CONNMARK --restore-mark

# 2. Mark NEW packets
iptables -t mangle -A PREROUTING -m conntrack --ctstate NEW \
-i tun0 -j MARK --set-mark 0x42

# 3. Save the mark back to conntrack for future packets
iptables -t mangle -A PREROUTING -j CONNMARK --save-mark
```

Packet 1 sets the mark and saves it to conntrack. Packets 2-4 inherit it.

MARK --set-mark

--save-mark

--restore-mark

ip rule fwmark

P1 EW

P2 EST

P3 EST

P4 EST

conntrack entry  
10.0.0.5 → 1.1.1.1:80

---

## Use case 3 — Multi-WAN with metrics

Two default routes with different metrics provide automatic failover. Lower metric = preferred.

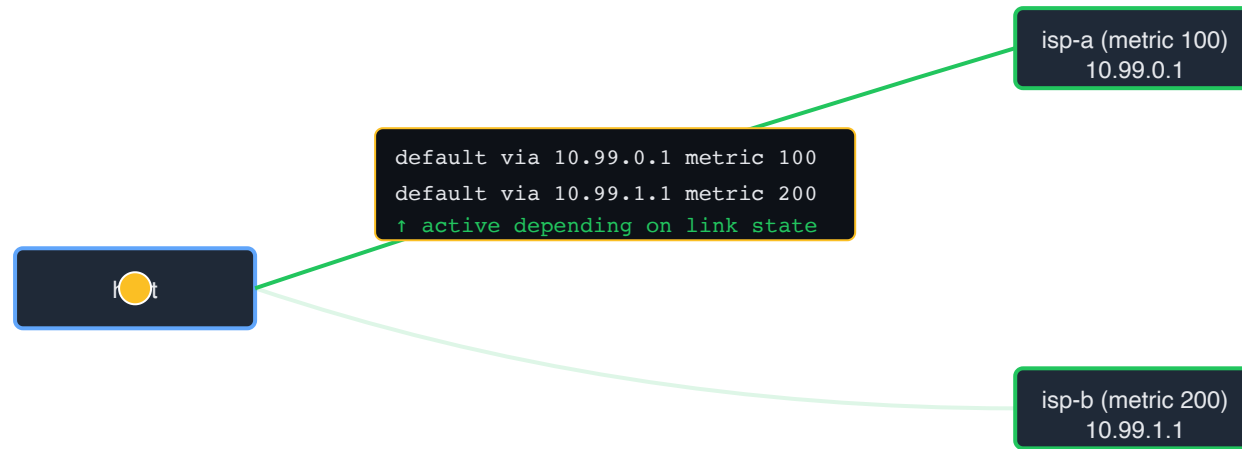
When the primary link goes down, the kernel falls back to the higher-metric route automatically.

```
# Primary default (metric 100 — preferred)
ip route add default via 10.99.0.1 dev eth0 metric 100

# Backup default (metric 200 — used only if eth0 fails)
ip route add default via 10.99.1.1 dev eth1 metric 200

# Optional: source-policy overlay to guarantee return-path symmetry
ip rule add from 10.99.0.0/24 lookup isp-a pref 100
ip rule add from 10.99.1.0/24 lookup isp-b pref 110
```

Primary WAN fails; traffic re-routes to backup.



# The `rp_filter` gotcha

When you add policy routing, return packets may arrive on an interface that *doesn't match* the route the kernel would use to reach the sender. The **Reverse Path Filter** will silently drop these packets.

## Symptom

You can ping out, but replies never arrive. `ip route get` shows the right table. `conntrack -L` shows the flow. But packets vanish.

```
# Check current rp_filter values
sysctl net.ipv4.conf.all.rp_filter
sysctl net.ipv4.conf.eth0.rp_filter

# 0 = off   |  1 = strict (default)  |  2 = loose
# For PBR, use loose mode (2) on the relevant interface
sysctl -w net.ipv4.conf.eth0.rp_filter=2
sysctl -w net.ipv4.conf.all.rp_filter=2
```

```
# Persist in /etc/sysctl.d/  
echo "net.ipv4.conf.all.rp_filter=2" >> /etc/sysctl.d/99-pbr.conf
```

### Why loose, not off?

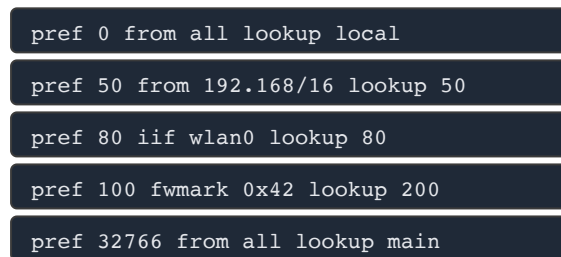
Loose mode (2) accepts packets if *any* route leads back to the source — catching obvious spoofs. Strict mode (1) requires the packet to arrive on the *same* interface the kernel would use to reach the source, which breaks asymmetric PBR setups.

---

# Under the hood — iproute2 internals

Understanding the data structures behind `ip rule` and `ip route` helps you reason about performance and design rule lists correctly.

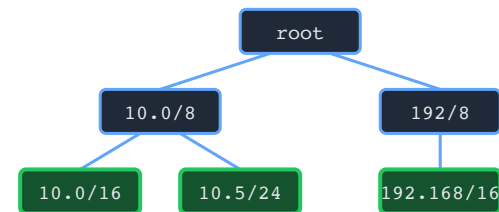
**ip rule: sorted linear array**



**$O(N)$  rules) per packet**

walked linearly, top-to-bottom  
keep the list short

**FIB: LC-trie (longest prefix)**



**$O(\log N)$  routes) per packet**

LC-trie + per-CPU FIB  
(per-flow route cache removed  
in kernel 3.6, year 2012)

Rules are linear, FIB is a tree.  
Order matters in ip rule. Specificity matters in ip route.

# Toolkit

Five commands cover 90% of PBR debugging:

```
# 1. Show the rule list (and which table each rule points to)
ip rule show

# 2. Show routes in a specific table
ip route show table 200

# 3. Show all tables at once (filter noise)
ip route show table all | grep -v "^broadcast\|^local\|^multicast"

# 4. Simulate the full two-step lookup for a given source/destination
ip route get 8.8.8.8 from 10.10.0.5

# 5. Watch conntrack marks on live traffic
conntrack -L -o extended | grep mark
```

**Notes:** `ip route get ... from ...` is the most underused command in Linux networking. It tells you exactly which table the kernel selected and which route it matched, in one call.

---

# Common mistakes

- **Forgetting the return route.** Adding a rule for outbound traffic (src-based) without also ensuring replies come back on the same interface creates asymmetric routing. Always add a matching rule for the source subnet of each WAN gateway.
- **rp\_filter=1 in strict mode.** Strictly correct packets dropped silently. Check with `sysctl net.ipv4.conf.all.rp_filter` first before spending hours in tcpdump.
- **Rule list explosion.** Each rule is walked linearly per packet. 50+ rules cause measurable latency on high-pps routers. Consolidate by using fwmark instead of many source-IP ranges.
- **Forgetting CONNMARK for stateful flows.** MARK alone only marks a single packet. If you need all packets in a TCP flow to use the same path, add `--save-mark` / `--restore-mark`.
- **Rules not persistent across reboots.** `ip rule add` is ephemeral like `ip route add`. Write rules to `/etc/iproute2/rt_rules` or your distro's network configuration (NetworkManager, systemd-networkd, ifupdown).

**Notes:** The first two bullets account for the majority of "PBR isn't working" issues in production. Spend extra time on them during the live session.

---

# Recap

- **Two-step routing** — `ip rule` picks a table; that table is consulted for the longest prefix match.
- **Selectors** — `from`, `to`, `fwmark`, `iif`, `oif`, `tos`; combine freely, lower `pref` wins.
- **CONNMARK** — `--save-mark` / `--restore-mark` makes marks sticky to the entire conntrack flow.
- **Watch out for** `rp_filter` (use loose mode=2), return-route symmetry, and rule list length.

**Where next:** VRF-lite ( `ip link add vrf0 type vrf table 100` ), IPv6 policy routing ( `ip -6 rule` ), and traffic shaping with `tc` using fwmark as a classifier.



# Policy-Based Routing

Lesson 03 · Linux Networking · ~10 min read

## Why a second routing decision exists

---

In Lesson 02 you learned that every outbound packet goes through a longest-prefix match against the `main` routing table. That model works perfectly when you have one uplink and every host in every subnet should exit through the same gateway. The moment you have two ISPs, a management network that must always go through a VPN, or application traffic that needs to be steered by firewall mark rather than destination — the single-table model breaks down.

Consider a dual-homed host with one interface to ISP-A and one to ISP-B. You want traffic from your internal `10.10.0.0/24` subnet to exit ISP-A, and traffic from a second subnet `10.20.0.0/24` to exit ISP-B. You cannot express this in a single routing table because both subnets share the same default route slot (`0.0.0.0/0`), and only one default wins at a time.

Linux solves this with a two-step lookup. Before the kernel consults a routing table, it walks a priority-sorted list of **rules**. Each rule says "if this packet matches these selectors, go look in *this* routing table." Only when a rule matches does the kernel actually touch a routing table. The rules themselves contain no route information — they just map packets to table IDs. This two-step design means you can have arbitrarily many routing tables and express almost any forwarding policy using combinations of source address, firewall mark, incoming interface, TOS bits, and other metadata.

## Anatomy of an ip rule

---

A rule has three parts: a **preference** (priority number), one or more **selectors**, and a **target table**. Lower preference numbers are evaluated first. The first rule that matches wins — the kernel stops walking the list and uses the named table for the actual route lookup.

```
ip rule add pref 100 from 10.10.0.0/24 lookup 200
```

This says: for any packet whose source address falls inside `10.10.0.0/24`, look up the route in table `200`. Table `200` is a user-defined table that you populate separately with `ip route add ... table 200`. If the source address does not match, the rule is skipped and the next one (by preference) is tried.

Every Linux system starts with three built-in rules that you cannot delete:

- **pref 0** — `from all lookup local`: handles your own IP addresses and broadcast; always runs first.
- **pref 32766** — `from all lookup main`: the fallback that causes normal routing to work by default.
- **pref 32767** — `from all lookup default`: a last-resort empty table; almost never used in practice.

Your custom rules go between preference 1 and 32765. The selectors available to you include: `from` (source CIDR), `to` (destination CIDR), `fwmark` (Netfilter mark), `iif` (incoming interface), `oif` (outgoing interface), and `tos` (IP TOS field). You can combine selectors in a single rule — all of them must match for the rule to fire.

The widget below lets you build a rule list and send a test packet through it. Add rules, set the packet's source IP, fwmark, and incoming interface, then click Send to see which rule matches first.

Add up to 5 rules. Send a packet. See which one matches first.

pref: 100

selector: from CIDR ▼

e.g. 10.10.0.0/24

→ table: main ▼

+ Add rule

No rules. Click "+ Add rule".

test packet — src: 10.10.0.5

fwmark: 0x42

iif: eth0

Send

Clear all

Add some rules, then click Send.

## Three use cases you will actually meet

---

### Source-based routing

The most common PBR scenario is a host or router with two uplinks where each source subnet must always exit via a specific gateway. You create one routing table per uplink, populate each table with a default route pointing at the appropriate gateway, then add `ip rule` entries that steer each source subnet to the right table.

```
# Table 200: traffic from 10.10.0.0/24 exits via ISP-A
ip route add default via 203.0.113.1 dev eth0 table 200
ip rule add pref 100 from 10.10.0.0/24 lookup 200

# Table 201: traffic from 10.20.0.0/24 exits via ISP-B
ip route add default via 198.51.100.1 dev eth1 table 201
ip rule add pref 101 from 10.20.0.0/24 lookup 201
```

Traffic from any other source still falls through to the default pref 32766 rule and uses the main table as normal. Source-based routing is the lightest form of PBR — no Netfilter involvement, no conntrack, just `ip rule` and `ip route`.

### fwmark-driven routing

Sometimes you cannot classify traffic by source address alone. You want to route all DNS queries via one path, or steer SSH traffic through a VPN regardless of where it originates. In these cases you use Netfilter's `MARK` target to paint a numeric mark onto matching packets in the mangle table, then write an `ip rule` that matches on that mark.

```
# Mark all TCP traffic destined for port 22
iptables -t mangle -A OUTPUT -p tcp --dport 22 -j MARK --set-mark 0x42

# Route all marked packets through table 300 (VPN)
ip route add default via 10.99.0.1 dev tun0 table 300
ip rule add pref 90 fwmark 0x42 lookup 300
```

The mark only exists in kernel memory for the lifetime of the packet — it is not part of the IP header. Once the packet leaves the machine the mark is gone. For most use cases this is fine; for TCP flows spanning many packets you need CONNMARK, covered in the next section.

## Multi-WAN with metrics

When you have two uplinks and want automatic failover rather than strict partitioning, you install two default routes in the main table with different metrics. The kernel always prefers the lower metric when both routes are present. When ISP-A's link drops, you remove or invalidate its route and the kernel automatically falls over to the higher-metric ISP-B route.

```
ip route add default via 203.0.113.1 dev eth0 metric 100    # primary
ip route add default via 198.51.100.1 dev eth1 metric 200   # backup
```

You can also overlay `ip rule` on top of this: add a rule that forces a specific source CIDR to a dedicated table, while leaving the two-metric default routes in main for everything else. This gives you both source-based partitioning and metric-based failover simultaneously.

## CONNMARK and stateful PBR

The `MARK` target paints a single packet. The moment that packet clears the mangle table, the mark exists only as long as the packet is in flight. The next packet in the same TCP flow — say, the second segment of an HTTP response — arrives with mark zero. If your `ip rule fwmark` only sees marked packets it will route the first packet to table 300 via the VPN, and the second packet will fall through to the main table via the regular gateway. The result is an asymmetric flow where replies and retransmits take different paths, which almost always causes TCP to stall or reset.

The fix is **CONNMARK**. CONNMARK stores the mark in the conntrack entry for the flow, not just on the individual packet. Two iptables targets handle this:

- `CONNMARK --save-mark`: copies the packet's fwmark into the conntrack entry. Run this after `MARK --set-mark` on the first (NEW) packet.
- `CONNMARK --restore-mark`: copies the mark from the conntrack entry back onto every subsequent packet in the flow. Run this early in PREROUTING so the mark is present before `ip rule` is consulted.

A complete three-rule mangle chain for a PBR scenario looks like this:

```
iptables -t mangle -A PREROUTING -j CONNMARK --restore-mark
iptables -t mangle -A PREROUTING -m conntrack --ctstate NEW -j MARK --set-mark
```

```
iptables -t mangle -A PREROUTING -m mark ! --mark 0 -j CONNMARK --save-mark
```

Rule order matters. The restore runs first so that established packets inherit their saved mark immediately. Only packets in state NEW then go through the classifier. The save runs last to ensure the mark is written back to conntrack whenever a new mark is set. With this chain in place every packet in a flow — from SYN to FIN — carries the same mark and takes the same routing path.

Use the playground below to see what happens when you send a four-packet flow with and without CONNMARK rules enabled.

#### iptables -t mangle -A PREROUTING

☒ -m conntrack --ctstate NEW -j MARK --set-mark 0x42  
☐ -j CONNMARK --save-mark  
☐ -j CONNMARK --restore-mark

#### ip rule

☒ from all fwmark 0x42 lookup 200  
ip route show table 200:  
default via 10.99.0.1

#### conntrack entry

(empty — no flow yet)

P1 NEW P2 EST P3 EST P4 EST

P1

P2

P3

P4

Send 4-packet flow

Toggle CONNMARK rules. Then send a flow.

## Multi-WAN in practice

Running two uplinks reliably is harder than it looks. The metric-based failover approach described above requires something external — a health-check daemon, [keepalived](#), or a custom script — to detect when a link dies and remove its route. Until the route is removed the kernel keeps sending packets out the dead interface, where they are silently lost.

A production multi-WAN setup typically combines three pieces: metric-based routes in the main table for automatic failover, ECMP (equal-cost multi-path) or weighted routing

for load-sharing when both links are healthy, and `ip rule` with source-based rules to pin specific services to a specific WAN regardless of metrics. For example, a VoIP server might always exit via the low-latency fibre link even if the cable modem temporarily has a lower metric because its monitor script hasn't run yet.

Stateful services — anything using TCP — need CONNMARK if you want all packets in a flow to take the same path. Without it, the first packet of a TCP handshake might go out ISP-A but the ACK from the remote server comes back via ISP-B's IP. The remote host sees a reply from an unexpected address and drops the connection.

The simulator below gives you two WANs with adjustable metrics. Kill either WAN, toggle policy mode, and send packets to see how the routing decision changes.

WAN-A metric: 100

WAN-B metric: 200

Kill WAN-A

Kill WAN-B

Policy: OFF

Send packet from 10.10.1.5

host (10.10.1.5)

WAN-A  
metric 100

WAN-B  
metric 200

Try killing a WAN or toggling policy mode, then send a packet.

## The rp\_filter trap

Reverse-path filtering ( `rp_filter` ) is a kernel anti-spoofing mechanism. When enabled, the kernel checks whether the source address of an arriving packet is reachable via the same interface the packet arrived on. If the routing table says the source IP should have come from a different interface, the packet is dropped silently. For a standard single-homed host this is a useful protection. For a host doing PBR it is a frequent source of mysterious connectivity failures.

The scenario: you have set up a rule that routes replies from table 200 out through `eth1`, but the packet arrived on `eth0`. The kernel's strict `rp_filter` check (value `1`) sees that the main table routes back to that source via `eth0`, but the packet came in on `eth1`, so it drops the packet before any routing decision is made. Your PBR configuration is correct but you see no traffic.

**Symptom:** ping replies vanish when using PBR with non-default interfaces.

**Cure:** `sysctl -w net.ipv4.conf.<iface>.rp_filter=2` (loose mode).

Loose mode (value `2`) only requires that the source address be reachable via *any* interface, not necessarily the one the packet arrived on. This is almost always the right setting for interfaces participating in PBR. You can also set `net.ipv4.conf.all.rp_filter=2` to apply it globally, but prefer per-interface settings to minimize the attack surface.

To make the change persistent across reboots, add it to `/etc/sysctl.d/`:

```
echo "net.ipv4.conf.eth1.rp_filter=2" | sudo tee /etc/sysctl.d/99-pbr.conf
sudo sysctl -p /etc/sysctl.d/99-pbr.conf
```

## Under the hood

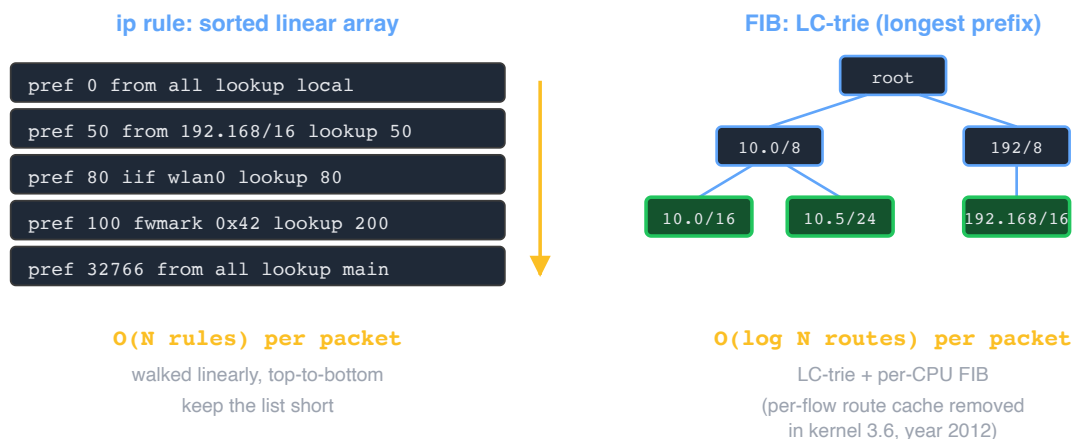
It is useful to understand what the kernel actually does when it evaluates rules and routes, because the performance characteristics differ between the two structures.

The rule list (`ip rule show`) is stored as a sorted linked list in the kernel. When a packet arrives, the kernel walks it from the lowest preference number upward, checking each rule's selectors against the packet's metadata. This is an  $O(N)$  linear scan — if you have 200 rules, the worst-case packet may check all 200 before finding a match or falling through to main. Keep the rule list short. Dozens of rules are fine; thousands are not.

The routing table itself — the Forwarding Information Base (FIB) — is stored as an LC-trie (level-compressed trie), which allows longest-prefix lookups in  $O(\log N)$  time. The kernel also maintains a per-CPU cache of recent lookups to amortize repeated lookups for the same flow. Note that the per-flow route cache that existed in older kernels was

removed in kernel 3.6 (2012); modern kernels do the trie walk on every packet, relying on CPU caches for speed rather than a software route cache.

The diagram below shows both structures side by side.



Rules are linear, FIB is a tree.  
Order matters in ip rule. Specificity matters in ip route.

## Recap

- Linux looks up an *ip rule* before consulting any routing table.
- Selectors include `from`, `to`, `fwmark`, `iif`, `oif`, `tos`.
- The first matching rule wins; lower `pref` = higher priority.
- CONNMARK `--save-mark` / `--restore-mark` persists a mark across every packet in a flow.
- Rule lists are walked linearly; keep them short. FIB is a trie — specificity matters.

# Lab — Policy-Based Routing, hands-on

Lesson 03 · ~15 min · Single host, uses ip netns + iptables mangle

## Caution

Modifies kernel routing rules and iptables mangle table. Run on a throwaway VM if you can. The cleanup section at the end reverses every change.

## Step 1 — Read your current rule list

```
ip rule show
# Default kernel rules:
# 0:    from all lookup local
# 32766: from all lookup main
# 32767: from all lookup default
```

## Step 2 — Add a source-based rule + populate a new table

```
# Pretend traffic from 10.10.0.0/24 should use a different gateway.
sudo ip rule add pref 100 from 10.10.0.0/24 lookup 200
sudo ip route add default via 10.99.0.1 dev eth0 table 200

ip rule show          # confirm pref 100 now appears
ip route show table 200
```

## Step 3 — Ask the kernel which table it would consult

```
ip route get 8.8.8.8 from 10.10.0.5
# Expected: shows table 200 selected, gateway 10.99.0.1
```

## Step 4 — fwmark-based rule

---

```
# Add a fwmark rule (higher priority than the source-based one)
sudo ip rule add pref 90 fwmark 0x42 lookup 300
sudo ip route add default via 10.99.1.1 dev eth0 table 300

# Mark outbound SSH (port 22) with iptables
sudo iptables -t mangle -A OUTPUT -p tcp --dport 22 -j MARK --set-mark 0x42

# See the rule fire (no actual SSH needed — just check the kernel agrees)
ip route get 8.8.8.8 mark 0x42
# Expected: table 300 selected, via 10.99.1.1
```

## Step 5 — Verify with conntrack (no save/restore yet)

---

```
# Without --save-mark, only the first packet of a NEW connection gets marked by
# the MARK rule. Subsequent packets in the same flow won't reliably hit it.
conntrack -L 2>/dev/null | head
```

## Step 6 — Sticky marks with CONNMARK

---

```
# The canonical "mark once, persist" pattern.
sudo iptables -t mangle -A PREROUTING -m conntrack --ctstate NEW -j MARK --set-mark 0x42
sudo iptables -t mangle -A PREROUTING -j CONNMARK --save-mark
sudo iptables -t mangle -A PREROUTING -j CONNMARK --restore-mark

# Now: every packet in the flow inherits the mark from conntrack.
# Open a long-lived TCP connection between two namespaces, then:
conntrack -L | grep mark
```

## Step 7 — Cleanup

---

```
sudo ip rule del pref 90 2>/dev/null
sudo ip rule del pref 100 2>/dev/null
sudo ip route flush table 200 2>/dev/null
sudo ip route flush table 300 2>/dev/null
sudo iptables -t mangle -F
```

## Self-check

---

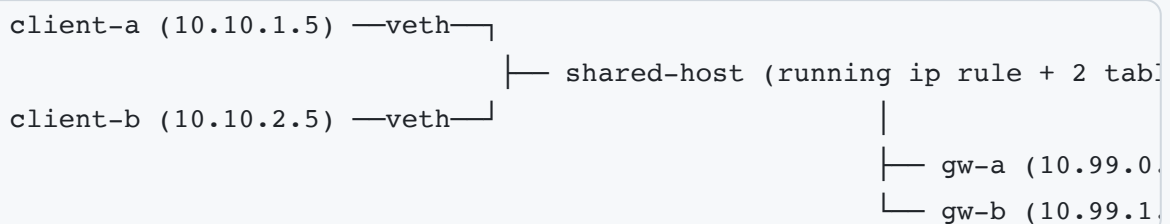
- ✓ You can list and add `ip rule` entries with custom priorities.
- ✓ You have populated a non-main routing table.
- ✓ You have used `ip route get ... from ...` to verify a policy match.
- ✓ You have wired iptables MARK to an `ip rule fwmark` entry.
- ✓ You have used the three-rule CONNMARK pattern ( `--set-mark` + `--save-mark` + `--restore-mark` ).

# Homework — Lesson 03

~30 min · Three tasks · Uses ip netns + iptables mangle

## Task 1 — Source-based routing across 3 namespaces

Build the following topology on a single host:



1. Create 5 namespaces: `client-a`, `client-b`, `shared-host`, `gw-a`, `gw-b`.
2. Connect them with veth pairs. Assign IPs as shown.
3. On `shared-host`, enable IPv4 forwarding.
4. Add `ip rule from 10.10.1.0/24 lookup 100` and `ip rule from 10.10.2.0/24 lookup 200`.
5. Populate table 100 with a default via `gw-a`; table 200 via `gw-b`.
6. Ping a "remote" IP from each client through the shared host.

### Self-check:

- `ip route get 8.8.8.8 from 10.10.1.5` resolves via `gw-a`.
- `ip route get 8.8.8.8 from 10.10.2.5` resolves via `gw-b`.
- `tcpdump` on each gateway sees only its expected client.

## Task 2 — fwmark-based routing

---

On a single namespace (or your host):

1. Add an iptables MANGLE rule in OUTPUT: mark all outbound SSH (tcp/22) traffic with `0x42`.
2. Add an `ip rule fwmark 0x42 lookup 300`.
3. Populate table 300 with a default via a different gateway than your normal one.
4. Verify with `ip route get 8.8.8.8 mark 0x42`.

### Self-check:

- `iptables -t mangle -L OUTPUT -v -n` shows your MARK rule.
- `ip rule show` shows your fwmark rule.
- `ip route get 8.8.8.8 mark 0x42` resolves through table 300.

## Task 3 — CONNMARK persistence verification

---

Building on Task 2's setup, add the three-rule CONNMARK chain:

1. Add `-j MARK --set-mark 0x42` on NEW packets (conntrack state NEW).
2. Add `-j CONNMARK --save-mark`.
3. Add `-j CONNMARK --restore-mark` BEFORE the MARK rule so subsequent packets get the mark before routing.
4. Open a long-lived TCP connection (e.g. `nc -lk 9999` on one namespace and `nc <ip> 9999` on another).
5. Run `conntrack -L | grep mark` while the connection is open.

### Self-check:

- `conntrack -L` shows the connection entry with `mark=0x42`.

- After typing some data into `nc` on the client side, the response on the server side travels through the policy-routing table — verified by `tcpdump` on the alternate gateway namespace.
- Restarting `nc` creates a new conntrack entry which also gets the mark.

[View solutions →](#)

# Solutions — Lesson 03

Worked answers for the homework tasks

Try the homework yourself first. PBR makes much more sense when you've seen it not work and figured out why.

## Task 1 — Source-based routing

```
# 1. Namespaces
for ns in client-a client-b shared gw-a gw-b; do sudo ip netns add $ns; done

# 2. Veth pairs (4 pairs: each client↔shared, shared↔each gw)
sudo ip link add veth-ca type veth peer name veth-sa
sudo ip link add veth-cb type veth peer name veth-sb
sudo ip link add veth-sga type veth peer name veth-gua
sudo ip link add veth-sgb type veth peer name veth-gub

# Move ends into namespaces
sudo ip link set veth-ca netns client-a
sudo ip link set veth-cb netns client-b
sudo ip link set veth-sa netns shared
sudo ip link set veth-sb netns shared
sudo ip link set veth-sga netns shared
sudo ip link set veth-sgb netns shared
sudo ip link set veth-gua netns gw-a
sudo ip link set veth-gub netns gw-b

# 3. IPs and link up
sudo ip -n client-a addr add 10.10.1.5/24 dev veth-ca
sudo ip -n shared addr add 10.10.1.1/24 dev veth-sa
sudo ip -n client-b addr add 10.10.2.5/24 dev veth-cb
sudo ip -n shared addr add 10.10.2.1/24 dev veth-sb
sudo ip -n shared addr add 10.99.0.2/24 dev veth-sga
sudo ip -n gw-a addr add 10.99.0.1/24 dev veth-gua
```

## Task 2 — fwmark-based routing

```
sudo iptables -t mangle -A OUTPUT -p tcp --dport 22 -j MARK --set-mark 0x42
sudo ip rule add pref 90 fwmark 0x42 lookup 300
sudo ip route add default via 10.99.1.1 dev eth0 table 300

ip rule show          # pref 90 fwmark 0x42 lookup 300 present
ip route show table 300
ip route get 8.8.8.8 mark 0x42  # → via 10.99.1.1
```

## Task 3 — CONNMARK persistence

```
# Order matters: restore BEFORE the conditional --set-mark.
# Otherwise the first packet works (NEW state matches), but you've
# already overwritten the mark before save can capture it.
sudo iptables -t mangle -A PREROUTING -j CONNMARK --restore-mark
sudo iptables -t mangle -A PREROUTING -m conntrack --ctstate NEW -j MARK --set-mark
sudo iptables -t mangle -A PREROUTING -j CONNMARK --save-mark

# Long-lived flow (run in two terminals or two namespaces):
#   server: nc -lk 9999
#   client: nc 10.x.x.x 9999

# Inspect:
sudo conntrack -L | grep mark
# Expected output line includes "mark=66" (0x42 = 66 decimal)
```

### Why the ordering matters

- **restore first:** if a non-NEW packet arrives with no mark, this pulls the mark from conntrack so routing sees it.
- **set-mark second (NEW only):** NEW packets (no conntrack entry yet) get the initial mark.
- **save last:** stamps the now-set mark onto the conntrack entry so future packets can restore from it.

**Common mistake:** putting `--save-mark` before `--set-mark` saves a zero mark to conntrack, then every "restore" pulls back zero. Symptom: only the first packet

of every new flow takes the policy path.

## Common mistakes across all three tasks

---

- **Priority collisions.** Two `ip rule` entries with the same `pref` are valid but undefined-ordered. Use distinct priorities.
- **Forgot the return route.** Building three-namespace topologies, both directions need routes. Symptom: tcpdump shows only one direction.
- **Forgot forwarding on the router namespace.** `net.ipv4.ip_forward=1` must be set inside the router namespace, not on the host.
- **rp\_filter dropping replies.** If you use PBR with non-default interfaces, set `rp_filter=2` (loose mode) on the relevant interfaces.
- **Flushing the wrong table.** `ip route flush table main` is catastrophic. Always specify the non-main table number explicitly.