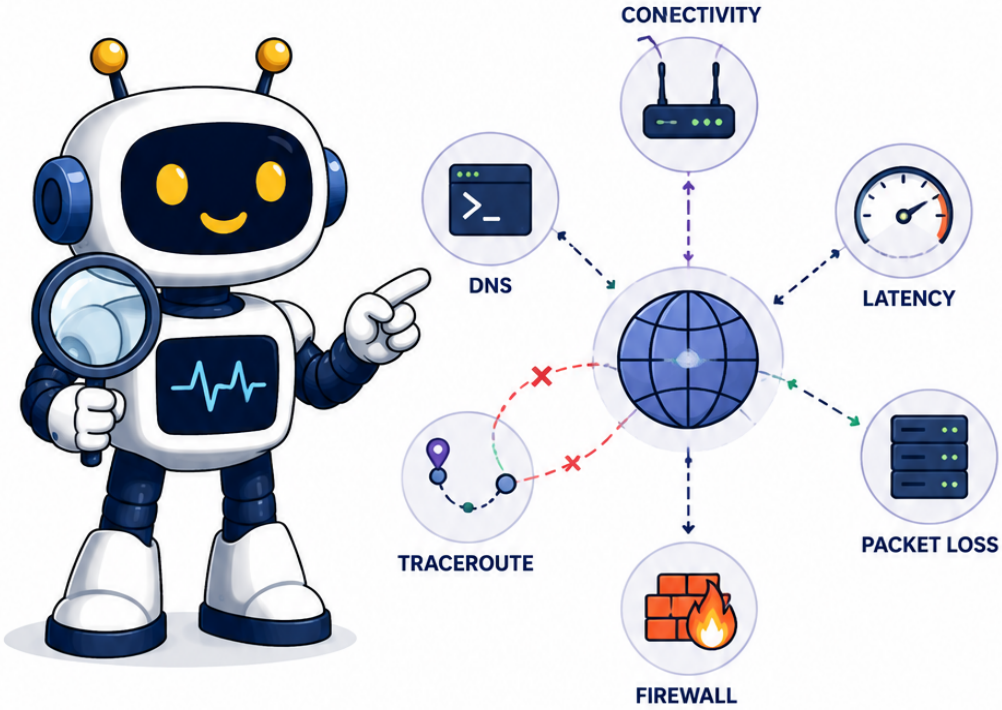
URADICAL

POCKET REFERENCE

NETWORK DIAGNOSTICS

— POCKET REFERENCE —



Alan Bradley

Network Diagnostics Pocket Reference

Command-line tools for diagnosing network failures, from DNS to the firewall

Alan Bradley

uradical.io

Network Diagnostics Pocket Reference

Alan Bradley

Table of Contents

Chapter 1 — DNS & Name Resolution

Diagnostic question: Why isn't DNS resolving?

DNS failures are rarely what they first appear to be. A timeout is not the same as NXDOMAIN. A local resolver returning the wrong answer is not the same as a broken authoritative record. Walk the full resolution chain — from the stub resolver through to the authoritative nameserver — and pinpoint exactly where the failure sits.

Resolution order

Before reaching DNS, the system checks `/etc/hosts`. A name resolving differently on one machine than another, despite identical DNS, is often an `/etc/hosts` entry. Check it first.

```
cat /etc/hosts
getent hosts example.com    # Shows which source answered:
hosts, dns, etc.
```

`/etc/nsswitch.conf` controls the lookup order. The `hosts:` line typically reads `files dns` — local files first, then DNS.

Tools covered

Tool	Purpose
dig	Full-featured DNS query tool. Primary diagnostic instrument.
drill	Lightweight ldns alternative. Default on Alpine and minimal images.

resolvectl	Query and inspect systemd-resolved. Essential on modern systemd distros.
host	Quick lookups. Simpler output than dig.

---

dig

Record types

```
dig example.com A        # IPv4 address
dig example.com AAAA     # IPv6 address
dig example.com MX       # Mail exchange
dig example.com NS       # Nameservers for the zone
dig example.com TXT      # Text records (SPF, DKIM,
verification tokens)
dig example.com CNAME    # Canonical name alias
dig example.com SOA      # Start of authority — zone serial,
refresh intervals
dig -x 93.184.216.34     # Reverse lookup (PTR)
```

Output control

```
dig +short example.com             # Answer values only — one
per line
dig +noall +answer example.com     # Answer section with full
record detail
```

`+short` output format varies by record type — MX includes priority, NS includes the trailing dot. Reliable for human reading; fragile for scripting. Use `+noall +answer` when parsing programmatically.

Targeting

```
dig @8.8.8.8 example.com           # Query a specific resolver
dig @ns1.example.com example.com   # Query the authoritative
server directly
```

Querying the authoritative server directly bypasses resolver cache. If the authoritative answer differs from what your resolver returns, you have a caching or propagation problem.

Tracing

```
dig +trace example.com
```

Starts from root nameservers and follows every referral to the authoritative server. Shows every query made and which server answered. Use when you need to locate exactly where in the delegation chain a failure sits.

TCP fallback

```
dig +tcp example.com
```

DNS defaults to UDP. If the response has the `TC` (truncated) flag set, the answer exceeded the buffer size. Retry over TCP.

DNSSEC

```
dig +dnssec example.com
```

Adds the DNSSEC OK flag and returns RRSIG records alongside the answer.

Key flags in dig output

Flag	Meaning

AA	Authoritative Answer — the responding server owns this zone
TC	Truncated — response cut off, retry with +tcp
AD	Authentic Data — DNSSEC validation passed
CD	Checking Disabled — DNSSEC validation skipped

Response codes

Code	Meaning
NOERROR	Query succeeded — answer section may still be empty
NXDOMAIN	The domain does not exist
SERVFAIL	Server failed to complete the query — often a DNSSEC issue
REFUSED	Server declined to answer — usually a policy decision

`NXDOMAIN` and an empty answer with `NOERROR` are different. `NOERROR` with no records means the record type you asked for does not exist, but the domain does.

---

drill

Syntax is close enough to `dig` that most users switch without friction.

```
drill example.com
drill example.com MX
drill @8.8.8.8 example.com
```

```
drill -x 93.184.216.34        # Reverse lookup
drill -T example.com          # Trace — equivalent to dig
+trace
drill -D example.com          # DNSSEC chain of trust
validation
```

Reach for `drill` on Alpine or minimal container images. For DNSSEC specifically, `drill -D` gives cleaner chain of trust output than `dig +sigchase` on most distro versions — it is the better tool for that task regardless of which image you are on.

---

resolvectl

On systems running `systemd-resolved`, the stub resolver listens on `127.0.0.53`. `resolvectl` queries and inspects its state.

```
resolvectl query example.com              # Query through the
system resolver
resolvectl query --type=MX example.com   # Specific record
type
resolvectl status                         # DNS servers and
settings per interface
resolvectl status eth0                    # Restrict to one
interface
resolvectl flush-caches                   # Clear all cached
records
resolvectl statistics                     # Cache hit/miss
counts
```

`resolvectl status` is the first thing to run when a machine resolves differently than expected — it shows which upstream servers each interface is actually using, which may differ from what `/etc/resolv.conf` claims. Use `resolvectl flush-caches` after a DNS record change when you cannot wait for TTL expiry.

---

host

For quick sanity checks where dig's output depth is not needed.

```
host example.com               # A and AAAA records
host -t MX example.com         # Specific record type
host 93.184.216.34             # Reverse lookup
host -v example.com            # Verbose — closer to dig output
format
```

Diagnostic workflows

A name is not resolving

```
# 1. Check /etc/hosts first
getent hosts example.com

# 2. Check what the system resolver returns and the response
code
dig example.com
dig example.com | grep "status:"

# 3. Bypass the system resolver
dig @8.8.8.8 example.com

# 4. If upstream works but local doesn't, inspect the resolver
resolvectl status
resolvectl flush-caches

# 5. Trace the full delegation chain
dig +trace example.com

# 6. Query the authoritative server directly
dig example.com NS +short
dig @ns1.example.com example.com
```

A DNS change is not propagating

```
# Has the zone serial been updated?
dig example.com SOA +short

# What TTL is left on the cached record?
dig example.com A

# Has the change reached public resolvers?
dig @8.8.8.8 example.com +short
dig @1.1.1.1 example.com +short
dig @9.9.9.9 example.com +short

# Is the new record live on the authoritative server?
dig @ns1.example.com example.com +short
```

Diagnosing email delivery problems

```
# MX records
dig example.com MX +short

# Do the MX hostnames resolve?
dig mail.example.com A +short

# SPF
dig example.com TXT +short | grep "v=spf"

# DKIM — selector is in the email headers
dig selector._domainkey.example.com TXT +short

# DMARC
dig _dmarc.example.com TXT +short
```

Confirming reverse DNS

```
dig example.com +short          # Forward lookup
dig -x 93.184.216.34 +short    # Reverse — does the PTR match
the forward?
```

Forward-confirmed reverse DNS (FCrDNS) is checked by mail servers and security tooling. A mismatch is a common source of delivery problems.

Quick reference

```
# Lookups
dig example.com                          # A record, system
resolver
dig +short example.com                   # Answer values only
dig +noall +answer example.com           # Answer section, full
detail
dig example.com MX +short               # MX records
dig -x 1.2.3.4                          # Reverse lookup
getent hosts example.com                # Check /etc/hosts and
resolver order

# Targeting
dig @8.8.8.8 example.com                # Specific resolver
dig @ns1.example.com example.com        # Authoritative server
directly

# Tracing & DNSSEC
dig +trace example.com                  # Full delegation chain
dig +dnssec example.com                 # Include DNSSEC records

# systemd-resolved
resolvectl query example.com            # Query through system
resolver
resolvectl status                       # Resolver state per
interface
resolvectl flush-caches                 # Clear cached records

# drill
drill -T example.com                    # Trace
drill -D example.com                    # DNSSEC validation
```

Next: Chapter 2 — Reachability & Routing

Chapter 2 — Reachability & Routing

Diagnostic question: Why can't I reach this host?

Reachability failures sit somewhere on a spectrum from "the host does not exist" to "a packet is being dropped six hops away." The tools in this chapter let you establish whether a host is up, trace the path packets take to get there, and inspect the routing decisions your machine makes before a packet even leaves the interface.

Tools covered

Tool	Purpose
ping / ping6	ICMP echo — basic reachability and round-trip time
mtr	Combined ping and traceroute — the production-useful version
traceroute / tracepath	Path analysis — where along the route packets go
ip	Routing table, interfaces, neighbours — the ifconfig and route replacement

---

ping

ICMP echo requests. The first tool to reach for — not because it tells you everything, but because it quickly rules out the obvious. A host that does not respond to ping is not necessarily down; ICMP is frequently filtered. A host that does respond tells you L3 connectivity is intact.

Basic usage

```
ping -c 4 example.com      # Send exactly 4 packets
ping -c 4 -q example.com   # Quiet — summary only, no
per-packet output
ping6 example.com          # IPv6
ping -6 example.com        # On systems where ping handles
both
```

Packet size and MTU testing

```
ping -s 1400 example.com              # Send 1400-byte packets
ping -M do -s 1400 example.com        # Set DF bit — test for
MTU issues
```

If large packets fail but small ones succeed, you have an MTU or fragmentation problem. The `do` flag (Don't Fragment) forces routers to return an ICMP "fragmentation needed" error rather than silently fragmenting. See the MTU diagnostic workflow below.

Reading ping output

```
64 bytes from 93.184.216.34: icmp_seq=1 ttl=55 time=11.2 ms
```

- `ttl` — TTL remaining on arrival. Subtract from the starting value (typically 64 or 128) to estimate hop count.
- `time` — round-trip time. Consistent values mean a stable path. High variance suggests congestion or a flapping route.

The summary on exit:

```
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 10.8/11.2/11.8/0.4 ms
```

`mdev` is mean deviation — low means consistent latency, high means jitter.

---

mtr

`mtr` combines ping and traceroute into a continuously updating view of every hop on the path to a host. It is the tool to use when you need to identify where packet loss or latency is actually occurring.

Basic usage

```
mtr example.com                          # Interactive live
display
mtr -r -c 20 example.com                 # Report mode — 20
packets, print summary
mtr -r -c 20 --json example.com          # JSON output for
scripting
```

Report mode is the one to use when you need to capture results — paste it into a ticket, store it in a log.

TCP and UDP modes

```
mtr --tcp example.com                    # TCP SYN instead
of ICMP
mtr --tcp --port 443 example.com         # TCP SYN to a
specific port
mtr --udp example.com                    # UDP
mtr -6 example.com                       # IPv6
```

Many networks filter ICMP but pass TCP. If ICMP-based mtr shows loss at every hop, switch to TCP mode targeting a port you know is open. Loss that disappears in TCP mode is ICMP filtering, not genuine packet loss.

Reading mtr output

```
Host                          Loss%   Snt   Last   Avg  Best
Wrst StDev
1. router.local                0.0%    20    0.4    0.4   0.3
0.6   0.1
2. 10.0.0.1                    0.0%    20    1.2    1.1   0.9
1.4   0.1
3. 203.0.113.1                20.0%    20    8.4    8.3   8.1
8.6   0.1
4. ???                       100.0%    20    ---    ---   ---
---   ---
5. 198.51.100.1                0.0%    20   11.2   11.1  10.9
11.4   0.1
6. 93.184.216.34               0.0%    20   11.4   11.3  11.0
11.7   0.2
```

Loss at a single hop, none at subsequent hops — the intermediate router is rate-limiting or deprioritising ICMP probes. Not a real problem; the traffic flows normally.

Loss beginning at a hop and continuing through all subsequent hops — the problem is at or just before that hop. That is where to focus.

`???` — the router is not responding to probes but traffic is still flowing through it. Look at the hops either side. If hops before and after show no loss, the `???` hop is simply filtering probes.

Latency jumping sharply at one hop and staying elevated — the bottleneck is at that hop. `StDev` climbing alongside avg latency confirms congestion rather than a routing change.

---

traceroute

Sends probes with incrementing TTL values to map each hop along the path. Each router decrements TTL and returns ICMP Time Exceeded when it hits zero, revealing its address and round-trip time.

Basic usage

```
traceroute example.com
traceroute -n example.com      # No name resolution — faster
output
traceroute -q 1 example.com   # One probe per hop instead of
three
```

Protocol modes

```
traceroute -I example.com                  # ICMP
traceroute -T example.com                  # TCP SYN
traceroute -T --port=443 example.com      # TCP SYN to a
specific port
```

Default mode uses UDP to high-numbered ports. ICMP and TCP modes are useful when UDP is filtered. TCP mode targeting an open port is the most likely to traverse firewalls successfully.

tracepath

`tracepath` requires no root and discovers path MTU alongside the route.

```
tracepath example.com
tracepath -n example.com
```

Prefer `tracepath` when diagnosing MTU-related problems — it surfaces the path MTU directly without needing to manually probe with ping.

Reading traceroute output

```
 1  router.local (192.168.1.1)    0.5 ms   0.4 ms   0.4 ms
 2  10.0.0.1 (10.0.0.1)          1.2 ms   1.1 ms   1.0 ms
 3  * * *
 4  203.0.113.1 (203.0.113.1)    8.4 ms   8.3 ms   8.2 ms
 5  93.184.216.34 (93.184.216.34) 11.2 ms  11.1 ms  10.9 ms
```

`* * *` — the router at this hop did not respond to probes. Does not mean packets are being dropped here; many routers deprioritise or filter probe responses. If routing resumes at the next hop, traffic is flowing.

All three probes at a hop showing very different times — asymmetric probe routing. The three probes may be taking different paths due to load balancing. `mtr` handles this more gracefully with its averaging over many packets.

Probes stopping entirely and not resuming — the path is broken at or just before the last responding hop. Combined with mtr, this confirms where to focus.

Latency increasing steadily hop by hop — normal. A sudden large jump at one hop that persists indicates a bottleneck or a long physical link at that point.

---

ip

`ip` replaces `ifconfig`, `route`, and `arp`. Those tools are deprecated and absent from many modern minimal images. Learn `ip`.

Interfaces

```
ip link show                    # All interfaces — state, MAC,
MTU
ip link show eth0               # Single interface
ip addr show                    # Interfaces with IP addresses
ip addr show eth0               # Single interface with
addresses
```

Interface state flags

Flag	Meaning
UP	Interface is administratively enabled
LOWER_UP	Physical link is up — cable connected, signal present
NO-CARRIER	Administratively up but no physical link
DOWN	Interface is administratively disabled

An interface that shows `UP` but not `LOWER_UP` has no physical link — check the cable or the switch port. `NO-CARRIER` and `LOWER_UP` absent are the same condition; distros render it differently.

Routing table

```
ip route show                       # Full routing table
ip route show table all             # All tables including
policy routes
ip route get 8.8.8.8                # Which route would be used
for this destination
```

`ip route get` is the most useful routing diagnostic command — it tells you exactly which route the kernel selects for a given destination, including the outbound interface and gateway.

```
ip route get 8.8.8.8
# 8.8.8.8 via 192.168.1.1 dev eth0 src 192.168.1.100 uid 1000
```

Adding and removing routes

```
ip route add 10.0.0.0/8 via 192.168.1.254    # Static route
ip route del 10.0.0.0/8                       # Remove it
ip route add default via 192.168.1.1          # Default
gateway
```

Neighbours (ARP/NDP table)

```
ip neigh show                          # ARP table — IP to MAC
mappings
ip neigh show dev eth0                 # Restrict to one interface
```

State	Meaning
REACHABLE	Recently confirmed reachable
STALE	Not refreshed recently — will be confirmed on next use
FAILED	ARP resolution failed — host did not respond
PERMANENT	Statically configured

A neighbour in `FAILED` state means the host at that IP is not responding to ARP. Check whether the IP is correct and the host is up.

IPv6

```
ip -6 addr show
ip -6 route show
ip -6 neigh show
```

Diagnostic workflows

Is the host up?

```
# Basic reachability
ping -c 4 example.com

# No response — is it ICMP filtering or genuine loss?
mtr --tcp --port 80 -r -c 10 example.com
```

```
# Confirm the name resolves to the expected IP
dig example.com +short
```

Where along the path is the problem?

```
# Live path view
mtr example.com

# Report mode for a reproducible capture
mtr -r -c 20 example.com

# ICMP filtered — switch to TCP
mtr --tcp --port 443 -r -c 20 example.com

# Confirm with traceroute
traceroute -n -T --port=443 example.com
```

My machine is not routing traffic correctly

```
# Full routing table
ip route show

# What route would be used for this destination?
ip route get 8.8.8.8

# Is the default gateway present?
ip route show | grep default

# Is the outbound interface up and addressed?
ip addr show eth0
ip link show eth0
```

Diagnosing MTU problems

> **Symptom:** The connection establishes but hangs when transferring data. The TCP handshake (small packets) succeeds; data transfer (large packets) stalls. MTU is the likely cause.

```
# Test whether large packets get through
ping -M do -s 1400 example.com    # Fails if path MTU < 1428
bytes
ping -M do -s 576 example.com     # Likely succeeds on any
path

# tracepath discovers path MTU automatically
tracepath example.com
```

Quick reference

```
# ping
ping -c 4 example.com                          # 4 packets
ping -c 4 -q example.com                      # Quiet, summary
only
ping -M do -s 1400 example.com                # MTU test with DF
bit
ping6 example.com                              # IPv6

# mtr
mtr example.com                                # Live
interactive
mtr -r -c 20 example.com                      # Report mode
mtr --tcp --port 443 -r -c 20 example.com  # TCP mode

# traceroute
traceroute -n example.com                      # No DNS
resolution
traceroute -T --port=443 example.com       # TCP probes
traceroute -n -q 1 example.com             # Fast, one probe
per hop
tracepath example.com                          # No root needed,
finds path MTU

# ip
ip addr show                                   # Interfaces and
addresses
ip link show                                   # Interface state
and MTU
ip route show                                  # Routing table
```

```
ip route get 8.8.8.8                          # Route for a
specific destination
ip neigh show                                  # ARP/NDP table
ip -6 route show                               # IPv6 routing
table
```

Next: Chapter 3 — Connection & Port Testing

Chapter 3 — Connection & Port Testing

Diagnostic question: Connection refused or timing out?

A failed connection is not a single failure mode. Refused means something actively rejected it — a service is not listening, or a firewall sent back a RST. Timeout means nothing responded at all — the packet may never have arrived, or a firewall is silently dropping it. The tools in this chapter let you determine whether a port is open, what is listening on it, and whether something between you and the service is interfering.

Tools covered

Tool	Purpose
ss	Socket statistics — what is listening, what is connected
nc / ncat	Raw TCP/UDP connections — test ports, send data, probe services
curl	HTTP/S testing with full protocol visibility

---

ss

`ss` (socket statistics) is the replacement for `netstat`. It queries the kernel's socket tables directly and is significantly faster on systems with many connections.

What is listening

```
ss -tlnp          # TCP, listening, no name resolution, show
process
ss -ulnp          # UDP listening sockets
ss -tlnp6         # IPv6 TCP listening
```

Flag breakdown: `-t` TCP, `-u` UDP, `-l` listening only, `-n` no name resolution, `-p` show process name and PID.

The most useful invocation for answering "is anything listening on this port":

```
ss -tlnp | grep :8080
```

All established connections

```
ss -tnp           # All established TCP connections with
process
ss -tnp | grep ESTABLISHED
ss -tnp | grep TIME-WAIT     # Connections in TIME-WAIT
```

Connection summary

```
ss -s
```

Prints a summary of socket counts by state — useful for spotting connection exhaustion or a SYN flood at a glance.

Filter by port or address

```
ss -tnp dst :443              # Connections to port 443
ss -tnp src :8080             # Connections from local port
8080
ss -tnp dst 93.184.216.34     # Connections to a specific
remote IP
```

Unix domain sockets

```
ss -xp            # Unix sockets with process info
ss -xlp           # Listening Unix sockets
```

Useful when a service is configured to listen on a socket file rather than a TCP port.

Reading ss output

```
State    Recv-Q  Send-Q  Local Address:Port   Peer
Address:Port  Process
LISTEN   0       128     0.0.0.0:80           0.0.0.0:*
  users:(("nginx",pid=1234))
ESTAB    0       0       192.168.1.10:54321
93.184.216.34:443  users:(("curl",pid=5678))
```

Field	Socket state	Meaning
Recv-Q	LISTEN	Connections waiting to be accepted — persistent non-zero means the app is not accepting fast enough
Recv-Q	ESTAB	Bytes received but not yet read by the application
Send-Q	ESTAB	Bytes sent but not yet acknowledged by the remote

`0.0.0.0:80` — listening on all IPv4 interfaces. `:::80` — all IPv6 interfaces. `127.0.0.1:80` — loopback only.

Socket states

State	Meaning
LISTEN	Accepting incoming connections
ESTAB	Connection established
TIME-WAIT	Connection closed, waiting for late packets to expire
CLOSE-WAIT	Remote closed, local application has not yet closed its end
SYN-SENT	Connection attempt in progress
FIN-WAIT-2	Local closed, waiting for remote to close

A large number of `TIME-WAIT` sockets is normal under high connection volume. A large number of `CLOSE-WAIT` sockets usually means an application is not closing connections properly.

lsof

`lsof -i` is a slower but more universally available alternative for mapping processes to sockets — useful on systems where `ss` is absent or when you want file descriptor context alongside the socket.

```
lsof -i :8080          # What process is using port 8080?
lsof -i tcp            # All TCP sockets with process info
lsof -i -n -P          # No name resolution — faster
```

---

nc / ncat

`nc` (netcat) opens raw TCP and UDP connections. It has no protocol knowledge — it just moves bytes — which makes it the cleanest tool for testing whether a port is open and a service is responding, without application-layer interference.

`ncat` is the nmap project's rewrite. Syntax is broadly compatible with minor differences. On some distros `nc` is a symlink to `ncat`; on others it is OpenBSD netcat. Check which you have:

```
nc --version
```

Test whether a port is open

```
nc -zv example.com 80          # TCP — connect and disconnect
immediately
nc -zv example.com 22          # Test SSH port
nc -zv example.com 80 443 8080 # Test multiple ports
```

`-z` zero I/O mode — connect and exit without sending data. `-v` verbose — prints whether the connection succeeded or was refused.

```
# Connection succeeded
Connection to example.com 80 port [tcp/http] succeeded!

# Connection refused
nc: connect to example.com port 9999 (tcp) failed: Connection
refused

# Timeout (nothing responded)
nc: connect to example.com port 9999 (tcp) failed: Operation
timed out
```

Refused means something is actively rejecting the connection — a RST was returned. Timeout means nothing responded — the packet may be dropped by a firewall or the host is unreachable.

UDP port testing

```
nc -zuv example.com 53         # UDP
```

UDP has no handshake, so there is no connection confirmation. A closed UDP port only reveals itself if the host returns an ICMP port unreachable — which many firewalls suppress. Silence is therefore

ambiguous: the port could be open, closed behind a firewall, or the host unreachable. A positive response confirms open; anything else is inconclusive.

Connect with a timeout

```
nc -zv -w 3 example.com 80     # 3 second timeout
```

Send data to a service

```
echo "GET / HTTP/1.0\r\nHost: example.com\r\n\r\n" | nc
example.com 80
```

Useful for testing a service response without a full client. Works for any line-oriented protocol — SMTP, POP3, FTP, Redis, Memcached.

```
# SMTP banner grab
nc -v mail.example.com 25

# Redis ping
echo "PING" | nc example.com 6379

# HTTP HEAD request
printf "HEAD / HTTP/1.0\r\nHost: example.com\r\n\r\n" | nc
example.com 80
```

Listen for incoming connections

```
nc -lvp 9999                   # Listen on port 9999
```

Useful for testing that traffic is reaching a host — start a listener, send traffic to it from another machine, confirm receipt.

Port scanning a range

```
nc -zv example.com 20-25       # Scan ports 20 through 25
```

Not a replacement for nmap but useful for quick range checks without additional tooling.

---

curl

`curl` has deep protocol support and verbose output that exposes every stage of an HTTP/S request — DNS resolution, connection, TLS handshake, headers, response. It is the tool for testing HTTP services and diagnosing where in the request lifecycle a failure occurs.

Basic request

```
curl https://example.com
curl -s https://example.com          # Silent — suppress
progress meter
curl -o /dev/null https://example.com  # Discard body, just
check the response
```

Response code only

```
curl -s -o /dev/null -w "%{http_code}" https://example.com
```

Returns just the HTTP status code. Useful in scripts and for quick health checks.

Follow redirects

```
curl -L https://example.com          # Follow redirects
curl -L -v https://example.com       # Follow redirects with
full verbosity
```

Verbose output

```
curl -v https://example.com
```

Shows the full request/response cycle: DNS resolution, TCP connection, TLS handshake, request headers sent, response headers received. The most useful diagnostic mode.

```
* Trying 93.184.216.34:443...
* Connected to example.com (93.184.216.34) port 443
* SSL connection using TLSv1.3 / TLS_AES_256_GCM_SHA384
* Server certificate: example.com
> GET / HTTP/2
> Host: example.com
< HTTP/2 200
< content-type: text/html
```

Lines prefixed * are curl status messages. > are request headers sent. < are response headers received.

Timing breakdown

```
curl -s -o /dev/null -w "dns:%{time_namelookup}
connect:%{time_connect} tls:%{time_appconnect}
total:%{time_total}\n" https://example.com
```

Breaks total request time into DNS lookup, TCP connect, TLS handshake, and full response. Pinpoints where latency is accumulating.

Custom headers and methods

```
curl -H "Authorization: Bearer token123"
https://api.example.com
curl -H "Content-Type: application/json" -d '{"key":"value"}'
https://api.example.com
curl -X DELETE https://api.example.com/resource/1
```

Bypass DNS — test against a specific IP

```
curl --resolve example.com:443:93.184.216.34
https://example.com
```

Sends the request to the specified IP while still using the correct `Host` header and SNI. Useful for testing a specific server behind a load balancer, or verifying a new deployment before DNS is updated.

Test with a specific interface or source IP

```
curl --interface eth1 https://example.com
curl --interface 192.168.1.100 https://example.com
```

Ignore TLS certificate errors

`curl -k` disables certificate validation — useful for testing self-signed or expired certs. Covered in full in Chapter 4.

Connection timeout

```
curl --connect-timeout 5 https://example.com   # Abort if no
connection in 5s
curl --max-time 10 https://example.com         # Abort if
total request exceeds 10s
```

Diagnostic workflows

Is a port open and is something listening?

```
# On the local machine — is anything listening on this port?
ss -tlnp | grep :8080

# From a remote machine — can I reach it?
```

```
nc -zv example.com 8080

# Refused vs timeout — what is the failure mode?
# Refused = RST returned, something actively rejected it
# Timeout = nothing responded, likely firewall or host
unreachable
```

A service is running but connections are failing

```
# What is actually listening, on which address, with what
queue state?
ss -tlnp | grep :8080
# LISTEN 0  128  0.0.0.0:8080  . accepting external
connections
# LISTEN 0  128  127.0.0.1:8080  . loopback only, external
connections refused
# LISTEN 32 128  0.0.0.0:8080  . non-zero Recv-Q, app not
accepting fast enough

# Test a raw connection bypassing the application client
nc -zv localhost 8080
```

An HTTP service is returning errors

```
# Full request lifecycle — where does it fail?
curl -v https://example.com

# Just the status code
curl -s -o /dev/null -w "%{http_code}" https://example.com

# Where is the time going?
curl -s -o /dev/null -w "dns:%{time_namelookup}
connect:%{time_connect} tls:%{time_appconnect}
total:%{time_total}\n" https://example.com

# Test against a specific backend, bypassing load balancer DNS
curl --resolve example.com:443:10.0.0.5 https://example.com
```

Testing a non-HTTP service

```
# SMTP
nc -v mail.example.com 25

# Redis
echo "PING" | nc example.com 6379

# PostgreSQL — confirm port is open (will not get a useful
response without auth)
nc -zv db.example.com 5432

# Confirm a listener is reachable from another host
# On the target host:
nc -lvp 9999
# From the source:
nc -v target.host 9999
```

Quick reference

```
# ss
ss -tlnp                              # TCP listeners with
process
ss -tlnp | grep :8080                 # Is anything on this
port?
ss -tnp                               # Established connections
ss -s                                 # Socket count summary
ss -tnp dst :443                      # Connections to port 443

# nc
nc -zv example.com 80                 # Is this port open?
nc -zv -w 3 example.com 80           # With timeout
nc -zv example.com 80 443 8080       # Multiple ports
nc -lvp 9999                          # Listen for incoming
connections
echo "PING" | nc example.com 6379  # Send data to a service

# curl
curl -v https://example.com                      # Full
```

```
request detail
curl -s -o /dev/null -w "%{http_code}" https://example.com   #
Status code only
curl -L https://example.com                      # Follow
redirects
curl --resolve example.com:443:10.0.0.5 https://example.com  #
Target specific IP
curl --connect-timeout 5 https://example.com     # Connection
timeout
curl -s -o /dev/null -w "dns:%{time_namelookup}
connect:%{time_connect} tls:%{time_appconnect}
total:%{time_total}\n" https://example.com  # Timing breakdown
```

Next: Chapter 4 — TLS & Certificate Inspection

Chapter 4 — TLS & Certificate Inspection

> **Diagnostic question:** Something is wrong with the cert or handshake?

TLS failures present in several distinct ways — an expired certificate, a missing intermediate, a hostname mismatch, a cipher negotiation failure, or a broken chain of trust. Each has a different cause and a different fix. The tools in this chapter let you inspect the full certificate chain, walk the handshake, and confirm what a server is actually presenting rather than what you assume it is.

Tools covered

Tool	Purpose
openssl s_client	TLS handshake inspection, certificate chain, cipher negotiation
curl	TLS verification in the context of a full HTTP request

---

openssl s_client

`openssl s_client` connects to a TLS endpoint and prints everything: the certificate chain, the negotiated cipher and protocol version, and the handshake result. It is the primary tool for TLS diagnostics.

Basic connection

```
openssl s_client -connect example.com:443
```

Connects, completes the TLS handshake, and dumps the certificate chain and session details. Send an empty line or press Ctrl+C to close.

The output to focus on:

```
Certificate chain
 0 s:CN=example.com
   i:C=US, O=DigiCert Inc, CN=DigiCert TLS RSA SHA256 2020 CA1
 1 s:C=US, O=DigiCert Inc, CN=DigiCert TLS RSA SHA256 2020 CA1
   i:C=US, O=DigiCert Inc, OU=www.digicert.com, CN=DigiCert
Global Root CA
---
subject=CN=example.com
issuer=C=US, O=DigiCert Inc, CN=DigiCert TLS RSA SHA256 2020
CA1
---
Verify return code: 0 (ok)
```

`Verify return code: 0 (ok)` is the line that matters. Any other code indicates a problem — see the verification codes table below. The certificate chain section shows how many certs the server sent and whether the issuer chain is complete.

SNI — Server Name Indication

Always pass `-servername` when connecting to a shared host or CDN. Without it, the server may present the wrong certificate.

```
openssl s_client -connect example.com:443 -servername
example.com
```

Check certificate expiry

```
echo | openssl s_client -connect example.com:443 -servername
example.com 2>/dev/null \
  | openssl x509 -noout -dates
```

Prints `notBefore` and `notAfter` dates. The `echo` closes the connection immediately after the handshake so the command exits cleanly.

```
notBefore=Sep  1 00:00:00 2024 GMT
notAfter=Sep 30 23:59:59 2025 GMT
```

Check the full certificate details

```
echo | openssl s_client -connect example.com:443 -servername
example.com 2>/dev/null \
  | openssl x509 -noout -text
```

Prints the full certificate — subject, issuer, validity period, Subject Alternative Names, key usage, and extensions.

Check Subject Alternative Names only

```
echo | openssl s_client -connect example.com:443 -servername
example.com 2>/dev/null \
  | openssl x509 -noout -ext subjectAltName
```

Lists every hostname the certificate is valid for. A hostname mismatch error means the name you are connecting to is not in this list.

Force a specific TLS version

```
openssl s_client -connect example.com:443 -tls1_2    # TLS 1.2
only
openssl s_client -connect example.com:443 -tls1_3    # TLS 1.3
only
openssl s_client -connect example.com:443 -no_tls1_3 # Exclude
TLS 1.3
```

Useful when diagnosing compatibility issues between a client and server that support different protocol versions.

Test a specific cipher

```
openssl s_client -connect example.com:443 -cipher AES256-SHA
```

If the server does not support the cipher, the handshake fails with `no shared cipher`. Useful for confirming a server's cipher policy.

STARTTLS protocols

For services that upgrade a plain connection to TLS:

```
openssl s_client -connect mail.example.com:587 -starttls smtp
openssl s_client -connect mail.example.com:143 -starttls imap
openssl s_client -connect ftp.example.com:21  -starttls ftp
```

Get the certificate fingerprint

```
echo | openssl s_client -connect example.com:443 -servername
example.com 2>/dev/null   | openssl x509 -noout -fingerprint
-sha256
```

Useful for verifying a deployed certificate matches what was issued, and for confirming identity in certificate pinning or internal CA workflows.

Check OCSP stapling

```
openssl s_client -connect example.com:443 -servername
example.com -status 2>/dev/null | grep -A 10 "OCSP response"
```

A server supporting OCSP stapling will include the OCSP response in the handshake. `OCSP Response Status: successful` confirms it is working.

Test against a specific IP

```
openssl s_client -connect 93.184.216.34:443 -servername
example.com
```

Connects to the IP directly while sending the correct SNI. Equivalent to `curl --resolve` — useful for testing a specific backend before DNS is updated.

Verify return codes

Code	Meaning
0	Certificate chain validated successfully
2	Unable to get issuer certificate — intermediate missing
10	Certificate has expired
18	Self-signed certificate
19	Self-signed certificate in the chain
20	Unable to get local issuer certificate — root not trusted
21	Unable to verify the first certificate
62	Hostname mismatch

Code `2` — missing intermediate — is one of the most common certificate misconfigurations. The certificate is valid but the server is not sending the full chain. Browsers often paper over this using cached intermediates; curl and openssl do not.

---

curl

`curl` verifies TLS as part of a full HTTP request. Where `openssl s_client` gives you raw handshake detail, `curl -v` shows TLS

verification in the context of the actual request — which is often what you need when diagnosing application-level failures.

Verbose TLS output

```
curl -v https://example.com 2>&1 | grep -E "^\*"
```

The `*` prefixed lines include all TLS detail — certificate subject, issuer, expiry, verification result, and negotiated protocol.

```
* Server certificate:
*  subject: CN=example.com
*  start date: Sep  1 00:00:00 2024 GMT
*  expire date: Sep 30 23:59:59 2025 GMT
*  subjectAltName: host "example.com" matched cert's
"example.com"
*  issuer: C=US; O=DigiCert Inc; CN=DigiCert TLS RSA SHA256
2020 CA1
*  SSL certificate verify ok.
```

Test with certificate validation disabled

```
curl -k https://example.com
```

Disables certificate validation entirely. Use to confirm that the failure is TLS-related — if `-k` succeeds but the normal request fails, the problem is in the certificate or chain. Never use in production scripts.

> **Note:** If `curl -k` still fails, the application may be using certificate pinning — a hardcoded expected certificate or public key that rejects even valid replacements. Pinning failures cannot be bypassed with `-k`; they require updating the pinned value in the application.

Test with a custom CA bundle

```
curl --cacert /path/to/ca-bundle.crt https://example.com
```

Useful when testing against internal services using a private CA. Points curl at a specific CA bundle rather than the system trust store.

Test with a client certificate

```
curl --cert /path/to/client.crt --key /path/to/client.key
https://example.com
```

For services requiring mutual TLS (mTLS). If the server requires a client certificate and none is presented, the handshake fails with a `handshake failure` or `certificate required` error.

Pin to a specific TLS version

```
curl --tlsv1.2 --tls-max 1.2 https://example.com    # TLS 1.2
only
curl --tlsv1.3 https://example.com                   # TLS 1.3
minimum
```

Diagnostic workflows

Certificate is expired or not yet valid

```
echo | openssl s_client -connect example.com:443 -servername
example.com 2>/dev/null \
  | openssl x509 -noout -dates        # notBefore and notAfter
dates
  # Verify return code 10 = expired
```

Hostname mismatch

```
echo | openssl s_client -connect example.com:443 -servername
example.com 2>/dev/null \
  | openssl x509 -noout -ext subjectAltName   # Every name the
cert is valid for
  # Verify return code 62 = hostname not in this list
```

Missing intermediate certificate

> **Symptom:** curl and openssl fail with a chain validation error, but the site loads fine in a browser. Browsers cache intermediate certificates; command-line tools do not.

```
echo | openssl s_client -connect example.com:443 -servername
example.com 2>/dev/null \
  | grep -c "BEGIN CERTIFICATE"
# 1 = server cert only — intermediate missing (verify return
code 2)
# 2 = server cert + intermediate (correct)
# 3 = full chain including root (unnecessary but harmless)
```

TLS version or cipher compatibility

```
openssl s_client -connect example.com:443 -servername
example.com </dev/null 2>/dev/null \
  | grep -E "Protocol|Cipher"         # What was negotiated

openssl s_client -connect example.com:443 -tls1_2 </dev/null
2>/dev/null \
  | grep "Verify return code"         # Does the server
support TLS 1.2?
```

Internal service with a private CA

```
# Does the cert validate against the internal CA bundle?
curl --cacert /etc/ssl/internal-ca.crt
https://internal.example.com

# Raw handshake detail
```

```
openssl s_client -connect internal.example.com:443 \
  -servername internal.example.com \
  -CAfile /etc/ssl/internal-ca.crt </dev/null 2>/dev/null \
  | grep "Verify return code"
```

Is the failure TLS or application?

```
# If -k succeeds but the normal request fails, the problem is
TLS
curl -k https://example.com          # Ignore cert errors
curl https://example.com             # Normal validation

# If both fail, the problem is not TLS
# Move to Chapter 3 (connection) or Chapter 5 (packet
behaviour)
```

Quick reference

```
# openssl s_client
openssl s_client -connect example.com:443 -servername
example.com   # Full handshake
echo | openssl s_client -connect example.com:443 -servername
example.com 2>/dev/null \
  | openssl x509 -noout -dates
       # Expiry dates
echo | openssl s_client -connect example.com:443 -servername
example.com 2>/dev/null \
  | openssl x509 -noout -ext subjectAltName
       # SANs
echo | openssl s_client -connect example.com:443 -servername
example.com 2>/dev/null \
  | grep -c "BEGIN CERTIFICATE"
        # Chain length
openssl s_client -connect example.com:443 -tls1_2
      # Force TLS 1.2
openssl s_client -connect mail.example.com:587 -starttls smtp
      # STARTTLS

# openssl x509 (certificate details from file)
```

```
openssl x509 -noout -fingerprint -in cert.pem      #
Certificate fingerprint

# curl
curl -v https://example.com 2>&1 | grep "^\*"       # TLS
detail only
curl -k https://example.com                          # Skip
cert validation
curl --cacert /path/to/ca.crt https://example.com   # Custom
CA bundle
curl --cert client.crt --key client.key https://example.com  #
mTLS
curl --tlsv1.2 --tls-max 1.2 https://example.com   # Force TLS
1.2
```

Next: Chapter 5 — Performance & Packet Behaviour

Chapter 5 — Performance & Packet Behaviour

> **Diagnostic question:** Connected but something is wrong with throughput or behaviour?

A connection that establishes but performs poorly, drops intermittently, or behaves unexpectedly is a different class of problem from the failures covered in earlier chapters. The tools here operate at the packet level — capturing what is actually on the wire, measuring raw throughput between hosts, and inspecting NIC-level statistics. This is where you go when the application says it is working but something is clearly wrong.

Tools covered

Tool	Purpose
tcpdump	Packet capture — what is actually on the wire
tshark	tcpdump's scriptable sibling — filtering, statistics, protocol dissection
iperf3	Raw bandwidth measurement between two hosts
ethtool	NIC statistics, link speed, offload settings

---

tcpdump

`tcpdump` captures packets on a network interface and prints or saves them. It operates below the application layer — it shows what is actually

being transmitted, not what the application thinks it is transmitting.

Basic capture

```
tcpdump -i eth0                    # Capture all traffic on
eth0
tcpdump -i any                     # Capture on all interfaces
tcpdump -n -i eth0                 # No name resolution —
faster output
tcpdump -nn -i eth0                # No name or port
resolution
```

Always use `-n` in production. Name resolution adds latency to output and can interfere with the capture itself under high traffic.

Capture to file

```
tcpdump -i eth0 -w capture.pcap              # Write raw
packets to file
tcpdump -i eth0 -w capture.pcap -C 100      # Rotate at 100MB
tcpdump -i eth0 -w capture.pcap -G 3600     # Rotate every
hour
tcpdump -r capture.pcap                      # Read and
display a saved capture
```

Saving to a pcap file is the right approach for anything beyond a quick glance — it preserves the full packet data for later analysis and can be opened in Wireshark.

Capture filters (BPF)

Filters use Berkeley Packet Filter syntax. Apply them early — capturing everything and grepping output loses packets under load.

```
tcpdump -i eth0 host 93.184.216.34          # Traffic to or
from this IP
tcpdump -i eth0 src 93.184.216.34           # Traffic from
this IP only
```

```
tcpdump -i eth0 dst 93.184.216.34           # Traffic to this
IP only
tcpdump -i eth0 port 443                    # Traffic on port
443
tcpdump -i eth0 tcp port 80                 # TCP port 80 only
tcpdump -i eth0 udp port 53                 # UDP port 53 only
tcpdump -i eth0 not port 22                 # Exclude SSH
tcpdump -i eth0 host 10.0.0.1 and port 80  # Combine filters
tcpdump -i eth0 'tcp[tcpflags] & tcp-syn != 0'  # SYN packets
only
tcpdump -i eth0 'tcp[tcpflags] & tcp-rst != 0'  # RST packets
only
```

Packet count and verbosity

```
tcpdump -i eth0 -c 100              # Capture exactly 100
packets then exit
tcpdump -i eth0 -v                  # Verbose — TTL, checksum,
IP options
tcpdump -i eth0 -vv                 # More verbose — full
protocol decoding
tcpdump -i eth0 -X                  # Print packet contents in
hex and ASCII
tcpdump -i eth0 -A                  # Print packet contents in
ASCII only
```

`-A` is useful for inspecting plaintext protocol exchanges — HTTP, SMTP, Redis — without the noise of the hex column.

Reading tcpdump output

```
14:32:01.123456 IP 192.168.1.10.54321 > 93.184.216.34.443:
Flags [S], seq 123456789, win 65535, length 0
14:32:01.145678 IP 93.184.216.34.443 > 192.168.1.10.54321:
Flags [S.], seq 987654321, ack 123456790, win 65535, length 0
14:32:01.145700 IP 192.168.1.10.54321 > 93.184.216.34.443:
Flags [.], ack 1, win 502, length 0
```

TCP flags

Flag	Symbol	Meaning
SYN	S	Connection initiation
SYN-ACK	S.	Connection accepted
ACK	.	Acknowledgement
FIN	F	Graceful close
RST	R	Abrupt reset
PSH	P	Push data to application immediately
URG	U	Urgent data

A connection that opens with `S`, `S.`, `.` is a normal three-way handshake. A `R` in response to a `S` means the port is closed or a firewall is sending resets. A stream that ends with `R` rather than `F` means the connection was abruptly terminated rather than gracefully closed.

---

tshark

`tshark` is the command-line version of Wireshark. It understands hundreds of protocols and can filter, decode, and produce statistics from captures. Where `tcpdump` shows raw packet lines, `tshark` understands what is inside them.

Basic capture

```
tshark -i eth0                                  # Capture on
interface
tshark -i eth0 -w capture.pcap                 # Write to file
tshark -r capture.pcap                         # Read a saved
```

```
capture
tshark -r capture.pcap -Y "http"           # Display filter on
a saved capture
```

Display filters

tshark display filters are more expressive than tcpdump BPF filters and operate on decoded protocol fields.

```
tshark -i eth0 -Y "http.response.code == 500"       # HTTP 500
responses
tshark -i eth0 -Y "dns.flags.rcode != 0"            # DNS
errors
tshark -i eth0 -Y "tcp.flags.reset == 1"            # TCP
resets
tshark -i eth0 -Y "tls.handshake.type == 1"         # TLS
ClientHello
tshark -i eth0 -Y "ip.addr == 93.184.216.34"        # Traffic
to/from IP
tshark -i eth0 -Y "tcp.analysis.retransmission"     # TCP
retransmissions
tshark -i eth0 -Y "tcp.analysis.duplicate_ack"      #
Duplicate ACKs
```

Extract specific fields

```
tshark -r capture.pcap -T fields -e ip.src -e ip.dst -e
http.request.uri
tshark -r capture.pcap -T fields -e dns.qry.name -e
dns.resp.addr
tshark -r capture.pcap -T fields -e frame.time -e
tcp.analysis.rtt
```

Extracting fields to tab-separated output makes captures scriptable — pipe to `sort`, `uniq -c`, `awk` for quick analysis.

Protocol statistics

```
tshark -r capture.pcap -q -z io,stat,1         # Bytes per
second
tshark -r capture.pcap -q -z conv,tcp          # TCP
conversation summary
tshark -r capture.pcap -q -z http,stat         # HTTP
request/response stats
tshark -r capture.pcap -q -z dns,tree          # DNS query
breakdown
```

When to reach for tshark over tcpdump

- Protocol-aware filtering — HTTP status codes, DNS rcodes, TLS handshake types
- Extracting specific fields for scripting or reporting
- Analysing a saved capture without Wireshark
- Generating conversation or protocol statistics from a pcap

---

iperf3

`iperf3` measures raw TCP and UDP bandwidth between two hosts. It requires a server process on one end and a client on the other — there is no passive mode. Use it to establish baseline throughput, identify link bottlenecks, and confirm that a connection can sustain the bandwidth a workload requires.

Server

```
iperf3 -s                          # Listen on default port
5201
iperf3 -s -p 9000                  # Listen on a custom port
iperf3 -s -D                       # Run as daemon
```

Client — TCP

```
iperf3 -c server.example.com                    # TCP test, 10
seconds
iperf3 -c server.example.com -t 30              # Run for 30
seconds
iperf3 -c server.example.com -P 4              # 4 parallel
streams
iperf3 -c server.example.com -R                # Reverse —
server sends to client, tests inbound bandwidth
iperf3 -c server.example.com -b 100M           # Limit
bandwidth to 100Mbps
```

`-R` reverses the data flow so the server sends and the client receives. This tests inbound bandwidth independently of outbound — useful for catching asymmetric links, which are common on cloud instances and some ISP connections where upload and download capacity differ significantly.

Client — UDP

```
iperf3 -c server.example.com -u                # UDP test
iperf3 -c server.example.com -u -b 500M       # UDP at 500Mbps
target rate
```

UDP mode reports jitter and packet loss alongside throughput — useful for diagnosing links that serve latency-sensitive traffic.

Reading iperf3 output

```
[ ID] Interval        Transfer     Bitrate         Retr
[  5]  0.00-10.00 sec  1.09 GBytes   935 Mbits/sec    2
sender
[  5]  0.00-10.01 sec  1.09 GBytes   933 Mbits/sec
receiver
```

- `Bitrate` — sustained throughput. Compare against the expected link speed.
- `Retr` — TCP retransmissions during the test. Non-zero indicates packet loss or congestion on the path.

- A large gap between sender and receiver bitrate suggests loss between the two hosts.

JSON output

```
iperf3 -c server.example.com -J               # JSON output
for scripting
iperf3 -c server.example.com -J | jq
'.end.sum_received.bits_per_second'
```

---

ethtool

`ethtool` queries and configures network interface hardware. It surfaces statistics that sit below the kernel's socket layer — NIC errors, dropped packets at the hardware level, link speed, duplex, and offload settings.

Link status and speed

```
ethtool eth0
```

```
Settings for eth0:
    Speed: 1000Mb/s
    Duplex: Full
    Auto-negotiation: on
    Link detected: yes
```

A duplex mismatch — one end full, the other half — causes severe throughput degradation and high collision counts. `Link detected: no` means no physical connection.

NIC statistics

```
ethtool -S eth0                    # Full hardware statistics
ethtool -S eth0 | grep -i drop    # Dropped packet counters
ethtool -S eth0 | grep -i error   # Error counters
```

```
ethtool -S eth0 | grep -i miss    # Missed packet counters
```

Drops at the NIC level — before the kernel even sees the packet — indicate the NIC's receive buffer is overflowing. This points to insufficient interrupt coalescing, CPU affinity misconfiguration, or a NIC that cannot keep up with line rate.

Live counter monitoring

```
watch -n1 'ethtool -S eth0 | grep -i drop'    # Watch drop
counters in real time
```

A single `ethtool -S` snapshot tells you the cumulative count since the interface came up. `watch` makes drops visible as they accumulate — far more useful when you are trying to correlate NIC drops with application errors under load.

Offload settings

```
ethtool -k eth0                    # Show offload features
ethtool -K eth0 tso off            # Disable TCP segmentation
offload
ethtool -K eth0 gro off            # Disable generic receive
offload
```

Offload settings occasionally interfere with packet captures — tcpdump may show packets larger than the MTU when TSO is active because segmentation happens on the NIC, not in the kernel. Disabling TSO during a capture gives accurate per-packet sizes.

Ring buffer sizes

```
ethtool -g eth0                    # Current and maximum ring
buffer sizes
ethtool -G eth0 rx 4096            # Increase receive ring
buffer
```

Increasing the receive ring buffer reduces drops under burst traffic — the NIC has more space to hold packets before the kernel drains them. Check the maximum supported value with `ethtool -g` before setting — not all NICs support 4096, and larger buffers consume more memory.

Driver and firmware information

```
ethtool -i eth0
```

Prints the driver name, version, and firmware version. Useful when chasing a NIC bug or confirming a driver update has taken effect.

Diagnostic workflows

Measuring actual throughput between two hosts

```
# On the destination host
iperf3 -s

# On the source host
iperf3 -c destination.host -t 30 -P 4

# Compare result against expected link speed from ethtool
ethtool eth0 | grep Speed
```

Diagnosing intermittent packet loss

```
# Establish baseline — is loss consistent or intermittent?
mtr -r -c 100 destination.host

# Capture during a loss event — look for retransmissions and
resets
tcpdump -i eth0 -w loss-capture.pcap host destination.host

# Analyse retransmissions in the capture
tshark -r loss-capture.pcap -Y "tcp.analysis.retransmission"
```

```
# Check NIC-level drops — is loss happening before the kernel?
ethtool -S eth0 | grep -i drop
```

Something is wrong but the connection looks fine

```
# What is actually being sent — confirm the application is
doing what you think
tcpdump -i eth0 -A port 8080 -c 50

# Are there unexpected resets?
tcpdump -i eth0 'tcp[tcpflags] & tcp-rst != 0' -nn

# Are retransmissions accumulating?
tshark -i eth0 -Y "tcp.analysis.retransmission" -q -z
io,stat,5
```

NIC is dropping packets before the kernel sees them

> **Symptom:** High traffic load, application sees errors, but no packet loss visible in tcpdump or mtr. NIC-level drops are invisible to kernel tools.

```
# Confirm drops at hardware level
ethtool -S eth0 | grep -i drop

# Check ring buffer — is it undersized?
ethtool -g eth0

# Increase ring buffer if headroom is available
ethtool -G eth0 rx 4096

# Confirm link speed and duplex — a mismatch degrades
throughput severely
ethtool eth0 | grep -E "Speed|Duplex"
```

Capture is showing oversized packets

> **Symptom:** tcpdump shows packets larger than the MTU — common when TCP segmentation offload is active.

```
# Confirm TSO is enabled
ethtool -k eth0 | grep tcp-segmentation-offload

# Disable for the duration of the capture
ethtool -K eth0 tso off
tcpdump -i eth0 -w capture.pcap host destination.host
ethtool -K eth0 tso on
```

Quick reference

```
# tcpdump
tcpdump -nn -i eth0                                # All
traffic, no resolution
tcpdump -nn -i eth0 host 10.0.0.1 port 443      # Filter by
host and port
tcpdump -nn -i eth0 -w capture.pcap              # Save to
file
tcpdump -r capture.pcap                           # Read saved
capture
tcpdump -nn -i eth0 'tcp[tcpflags] & tcp-rst != 0'  # RST
packets
tcpdump -nn -i eth0 -A port 8080 -c 50          # ASCII
payload, 50 packets

# tshark
tshark -i eth0 -Y "tcp.analysis.retransmission"  #
Retransmissions
tshark -i eth0 -Y "http.response.code == 500"    # HTTP 500s
tshark -r capture.pcap -q -z conv,tcp            # TCP
conversation stats
tshark -r capture.pcap -T fields -e ip.src -e ip.dst -e
http.request.uri

# iperf3
iperf3 -s                                         # Server
iperf3 -c host -t 30 -P 4                        # Client, 30s,
```

```
4 streams
iperf3 -c host -u -b 500M                        # UDP at
500Mbps
iperf3 -c host -J | jq '.end.sum_received.bits_per_second'  #
JSON

# ethtool
ethtool eth0                                      # Link speed
and status
ethtool -S eth0 | grep -i drop                   # NIC drop
counters
watch -n1 'ethtool -S eth0 | grep -i drop'    # Live drop
monitoring
ethtool -g eth0                                   # Ring buffer
sizes
ethtool -G eth0 rx 4096                           # Increase rx
ring buffer
ethtool -k eth0                                   # Offload
settings
ethtool -K eth0 tso off                           # Disable TSO
```

Next: Firewall & Filtering — Handoff Section

Firewall & Filtering

> **Diagnostic question:** Are packets being dropped or rewritten somewhere in the stack?

When the previous chapters have not found the problem — the host is reachable, the port is open, TLS is valid, throughput is fine — but connections are still failing or behaving unexpectedly, the firewall and packet filtering layer is the next place to look. This section covers the tools for inspecting that layer. For authoring, auditing, and managing nftables rulesets in depth, see the **nftables Pocket Reference**.

Tools covered

Tool	Purpose
nft list	Inspect the active nftables ruleset
conntrack	Inspect and manage connection tracking state

---

nft list

`nft` is the nftables command-line tool. The `list` subcommand shows the active ruleset — what rules are in place, in what order, and in what chains.

List the full ruleset

```
nft list ruleset
```

Prints every table, chain, and rule currently loaded. This is the starting point — establish what is actually in place before drawing conclusions about what should or should not be matching.

List a specific table

```
nft list table inet filter
nft list table ip nat
nft list table ip6 filter
```

List a specific chain

```
nft list chain inet filter input
nft list chain inet filter forward
nft list chain inet filter output
```

List sets and maps

```
nft list sets                       # All sets across all
tables
nft list set inet filter blocked_ips
nft list maps                       # All maps
```

Sets are named collections of addresses, ports, or other values used in rules. If a rule references a set, listing the set tells you what is actually in it — the rule alone does not.

Watch ruleset changes in real time

```
nft monitor
```

Streams ruleset changes as they happen — rule additions, deletions, and flushes. Useful when debugging dynamic firewall management tools or confirming that a rule change took effect.

Reading nft list output

```
table inet filter {
    chain input {
        type filter hook input priority filter; policy drop;

        ct state established,related accept
        ct state invalid drop
        iif lo accept
        tcp dport 22 accept
        tcp dport { 80, 443 } accept
        counter drop
    }
}
```

Key things to read:

- `policy drop` — the default policy is to drop. Any packet not explicitly accepted will be dropped.
- `policy accept` — the default policy is to accept. Rules are only needed to drop or reject specific traffic.
- `ct state established,related accept` — stateful rule allowing return traffic for established connections. Its absence means return traffic is being dropped.
- `counter drop` — the final rule counts packets before dropping. Checking the counter tells you whether traffic is actually hitting this rule.

Check rule counters

```
nft list ruleset | grep counter
```

Rules with `counter` show hit counts. A counter incrementing on a drop rule confirms that traffic is being matched and dropped there. A counter at zero on a rule you expect to be matching means either the traffic is not arriving or an earlier rule is matching it first.

If no counters are present, check for logging rules:

```
nft list ruleset | grep log     # Are any rules logging
matched packets?
journalctl -k | grep "IN="     # Kernel log entries from
nftables log rules
```

Log rules emit a line to the kernel log for each matched packet — tailing the system log while reproducing the problem confirms whether and where traffic is being matched.

For nftables ruleset authoring, policy assertions, and automated auditing, see the **nftables Pocket Reference** and `nftaudit`.

---

conntrack

`conntrack` inspects and manages the kernel's connection tracking table. The connection tracker is the state engine behind stateful firewall rules — it records active connections so that return traffic can be matched to `established,related` rules. When stateful filtering is misbehaving, this is where to look.

List tracked connections

```
conntrack -L                         # All tracked connections
conntrack -L | grep 443              # Filter by port
conntrack -L | grep 10.0.0.5        # Filter by IP
conntrack -L --proto tcp             # TCP only
conntrack -L --proto udp             # UDP only
```

Connection entry format

```
tcp      6 86394 ESTABLISHED src=192.168.1.10
dst=93.184.216.34 sport=54321 dport=443 \
         src=93.184.216.34 dst=192.168.1.10 sport=443
dport=54321 [ASSURED] mark=0
```

Each entry shows both directions of the connection — the original direction and the reply direction. The reply direction confirms what the connection tracker expects to see coming back. If NAT is active, the reply direction will show the translated addresses.

Connection states

State	Meaning
NEW	First packet of a new connection
ESTABLISHED	Connection is established, packets seen in both directions
RELATED	Related to an existing connection (FTP data, ICMP errors)
INVALID	Does not match any known connection — typically dropped
TIME_WAIT	Connection closed, waiting for late packets
CLOSE_WAIT	One side has closed

Count tracked connections

```
conntrack -C                        # Total number of tracked
connections
cat /proc/sys/net/netfilter/nf_conntrack_count   # Current
count
cat /proc/sys/net/netfilter/nf_conntrack_max     # Maximum
allowed
```

> **Symptom:** Connections are intermittently refused or dropped under load, with no obvious firewall rule causing it. Check whether the connection tracking table is full — when `nf_conntrack_count` reaches `nf_conntrack_max`, new connections cannot be tracked and are dropped.

```
# Is the table full or close to full?
echo "$(cat /proc/sys/net/netfilter/nf_conntrack_count) /
$(cat /proc/sys/net/netfilter/nf_conntrack_max)"
```

Delete a specific connection entry

```
conntrack -D --proto tcp --orig-src 192.168.1.10 --orig-dst
93.184.216.34 --orig-port-dst 443
```

Forces the connection tracker to forget a connection. The next packet will be treated as `NEW`. Useful when a NAT entry is stale and preventing re-establishment.

Watch connection events in real time

```
conntrack -E                          # Stream all connection
events
conntrack -E --proto tcp              # TCP events only
conntrack -E -e NEW                   # New connections only
```

Live connection events are useful for confirming that traffic is actually reaching the connection tracker and being tracked as expected.

NAT inspection

```
conntrack -L | grep DNAT              # Active DNAT translations
conntrack -L | grep SNAT              # Active SNAT translations
```

If NAT rules are in place, the conntrack table shows the actual translation in effect for each connection — the original addresses and the translated addresses. Mismatched NAT translations are a common source of asymmetric routing failures.

Diagnostic workflows

Is a firewall rule dropping this traffic?

```
# What is the current ruleset?
nft list ruleset

# What is the default policy on the relevant chain?
nft list chain inet filter input | grep policy

# Are drop rules accumulating hits?
nft list ruleset | grep counter

# Watch for rule changes — is something modifying the ruleset
dynamically?
nft monitor
```

Traffic arrives but does not reach the application

```
# Is the packet being tracked?
conntrack -L | grep <destination-port>

# Is the connection tracker full?
echo "$(cat /proc/sys/net/netfilter/nf_conntrack_count) /
$(cat /proc/sys/net/netfilter/nf_conntrack_max)"

# Check for INVALID state drops in the ruleset
nft list ruleset | grep invalid

# Capture what is arriving at the interface vs what the
application sees
tcpdump -nn -i eth0 port <port>     # At the interface
ss -tlnp | grep <port>              # What the application is
listening on
```

NAT is not translating correctly

```
# What translations are active?
conntrack -L | grep -E "DNAT|SNAT"

# Full entry for a specific connection
conntrack -L | grep <ip-address>

# Delete a stale NAT entry and force re-establishment
conntrack -D --proto tcp --orig-src <src> --orig-dst <dst>
--orig-port-dst <port>
```

Handoff to nftables Pocket Reference

This section covers inspection — reading the state of the firewall layer to determine whether it is the source of a problem. The following tasks are covered in the **nftables Pocket Reference**:

- Writing and managing nftables rules and chains
- Policy assertion and ruleset validation with `nftaudit`
- Set and map management
- NAT rule authoring
- Logging and audit trails

Quick reference

```
# nft
nft list ruleset                                  # Full active
ruleset
nft list chain inet filter input                 # Specific chain
nft list set inet filter blocked_ips             # Set contents
nft list ruleset | grep counter                  # Rules with hit
counters
nft monitor                                       # Watch ruleset
changes live
nft list ruleset | grep log                       # Find logging
rules
journalctl -k | grep "IN="                       # Kernel log
```

```
entries from log rules

# conntrack
conntrack -L                                  # All tracked
connections
conntrack -L | grep 443                       # Filter by port
conntrack -C                                  # Total
connection count
conntrack -E -e NEW                           # Watch new
connections live
conntrack -D --proto tcp --orig-src 192.168.1.10 --orig-dst
93.184.216.34 --orig-port-dst 443
cat /proc/sys/net/netfilter/nf_conntrack_count   # Current
tracked connections
cat /proc/sys/net/netfilter/nf_conntrack_max     # Maximum
allowed
```

Next: Appendix

Appendix

A — Diagnostic decision tree

Use this to identify which chapter and tool to reach for first.

```
Is the name resolving?
... No . Chapter 1: DNS & Name Resolution
.         dig, drill, resolvectl, host
.
... Yes — Can you reach the host?
    ... No . Chapter 2: Reachability & Routing
    .         ping, mtr, traceroute, ip
    .
    ... Yes — Is the port open and connection accepted?
        ... No . Chapter 3: Connection & Port Testing
        .         ss, nc, curl
        .
        ... Yes — Is TLS failing?
            ... Yes . Chapter 4: TLS & Certificate Inspection
            .          openssl s_client, curl
            .
            ... No — Is throughput or behaviour wrong?
                ... Yes . Chapter 5: Performance & Packet
Behaviour
                .          tcpdump, tshark, iperf3, ethtool
                .
                ... Still no cause found
                          Firewall & Filtering
                          nft list, conntrack
```

B — Connection failure modes

The single most useful diagnostic distinction in the book.

Failure	What it means	Where to look
Connection refused	RST received — port is closed or firewall sent a reset	Is something listening? ss -tlnp
Connection timed out	No response — packet dropped or host unreachable	Path problem or silent firewall drop — mtr, nft list ruleset
No route to host	Kernel has no route for the destination	Routing table — ip route show
Network unreachable	Local interface is down or no default gateway	ip link show, ip route show | grep default
Name or service not known	DNS resolution failed	dig, resolvectl status

---

C — Tool comparison

Overlapping tools and when to pick each.

ping vs mtr

	ping	mtr
Use for	Quick reachability check	Path analysis, loss location
Output	Per-packet RTT	Per-hop loss and latency
ICMP filtering	Shows as 100% loss	Shows as single-hop loss only
Preferred	Confirming a host is up	Diagnosing where loss occurs

traceroute vs mtr

	traceroute	mtr
Probes per hop	3 (default)	Continuous
Output	Single snapshot	Live updating
Load balancer handling	Inconsistent across probes	Averages over many packets
Preferred	Path documentation	Active loss investigation

tcpdump vs tshark

	tcpdump	tshark
Protocol awareness	Layer 3/4 only	Full application layer
Filter syntax	BPF	BPF capture + display filters
Field extraction	No	Yes — -T fields -e
Statistics	No	Yes — -z
Preferred	Quick capture, minimal tooling	Protocol analysis, scripting, saved captures

ss vs lsof -i vs netstat

	ss	lsof -i	netstat
Speed	Fast	Slow	Slow
Availability	Modern Linux	Universal	Deprecated
Process info	Yes (-p)	Yes	Yes (-p)

Preferred	Default choice on modern Linux	Fallback or when file descriptor context needed	Legacy systems only

nc vs curl

	nc	curl
Protocol knowledge	None — raw bytes	HTTP/S, FTP, and more
TLS support	Limited	Full
Use for	Port testing, non-HTTP protocols	HTTP service testing, TLS inspection
Preferred	Is this port open?	Is this HTTP service responding correctly?

D — Common flag reference

Flags that are easy to forget across the tools covered in this book.

dig

Flag	Effect
+short	Answer values only
+noall +answer	Answer section, full detail
+trace	Full delegation chain from root
+dnssec	Include DNSSEC records
+tcp	Force TCP instead of UDP
-x	Reverse lookup

@server	Query a specific resolver
+notcp	Force UDP (override default TCP fallback)
+time=N	Query timeout in seconds

ping

Flag	Effect
-c N	Send N packets then exit
-q	Quiet — summary only, no per-packet output
-s N	Packet size in bytes — use with -M do for MTU testing
-M do	Set DF bit — forces ICMP fragmentation-needed on oversized packets
-i N	Interval in seconds (default 1, minimum 0.2 without root)
-W N	Timeout per packet in seconds

mtr

Flag	Effect
-r -c N	Report mode, N packets
--tcp --port N	TCP mode on port N
-6	IPv6
--json	JSON output

ss

Flag	Effect
-t / -u	TCP / UDP sockets
-l	Listening sockets only
-n	No name resolution — faster output
-p	Show process name and PID
-s	Summary — socket counts by state
-x	Unix domain sockets
dst :N	Filter by destination port
src :N	Filter by source port

tcpdump

Flag	Effect
-i	Interface
-n	No name resolution
-nn	No name or port resolution
-w file	Write to pcap file
-r file	Read from pcap file
-c N	Capture N packets
-A	ASCII payload output
-X	Hex and ASCII payload output
-v / -vv	Verbose protocol decoding

curl

Flag	Effect
-v	Verbose — full request/response detail
-s	Silent — suppress progress meter
-o /dev/null	Discard response body
-w "format"	Write-out — extract timing and status fields
-L	Follow redirects
-k	Disable TLS certificate validation
--resolve host:port:ip	Bypass DNS, target specific IP
--cacert file	Custom CA bundle
--connect-timeout N	TCP connection timeout in seconds
--max-time N	Total request timeout in seconds

openssl s_client

Flag	Effect
-connect host:port	Target host and port
-servername host	SNI hostname
-tls1_2 / -tls1_3	Force specific TLS version
-starttls proto	STARTTLS for smtp, imap, ftp
-status	Request OCSP stapling
-CAfile file	Custom CA bundle

E — TLS verify return codes

Code	Meaning
0	OK — chain validated successfully
2	Unable to get issuer — intermediate missing
10	Certificate expired
18	Self-signed certificate
19	Self-signed certificate in chain
20	Unable to get local issuer — root not trusted
21	Unable to verify first certificate
62	Hostname mismatch

F — /proc and /sys quick reference

Useful kernel state that does not require additional tooling.

```
# Network interfaces
cat /proc/net/dev                    # Interface statistics —
bytes, packets, errors
cat /proc/net/if_inet6               # IPv6 interface addresses

# Routing
cat /proc/net/route                  # IPv4 routing table (hex
format)
cat /proc/net/ipv6_route             # IPv6 routing table

# Connections
cat /proc/net/tcp                    # TCP socket table (hex
format)
cat /proc/net/tcp6                   # IPv6 TCP socket table
```

```
cat /proc/net/udp                    # UDP socket table

# Connection tracking
cat /proc/sys/net/netfilter/nf_conntrack_count   # Current
tracked connections
cat /proc/sys/net/netfilter/nf_conntrack_max     # Maximum
allowed

# DNS (systemd-resolved)
cat /etc/resolv.conf                 # Active resolver
configuration
cat /etc/nsswitch.conf               # Name resolution order

# MTU and interface settings
cat /sys/class/net/eth0/mtu          # Current MTU
cat /sys/class/net/eth0/speed        # Link speed in Mbps
cat /sys/class/net/eth0/operstate    # Operational state: up,
down, unknown
```

G — IPv6 dual-stack gotchas

Dual-stack environments introduce a class of failure that is easy to misdiagnose — the tool appears to work but is testing the wrong protocol.

Tool defaults differ by distro

`ping` on some systems sends ICMP4 only; use `ping6` or `ping -6` explicitly for IPv6. `traceroute` defaults to IPv4 unless `-6` is passed. `curl` prefers IPv6 when both A and AAAA records exist — use `-4` or `-6` to force.

```
curl -4 https://example.com        # Force IPv4
curl -6 https://example.com        # Force IPv6
ping -4 example.com               # Force IPv4
ping -6 example.com               # Force IPv6
dig example.com AAAA               # Does this name have an
IPv6 address?
```

A name resolves on IPv4 but not IPv6

`dig example.com A` succeeds but `dig example.com AAAA` returns NXDOMAIN or empty. The service is IPv4-only. If a client is preferring IPv6 and getting no answer, it may fall back slowly or not at all depending on Happy Eyeballs implementation.

```
# What does the system resolver return for both?
resolvectl query example.com
resolvectl query --type=AAAA example.com
```

Different path on IPv6

`mtr example.com` and `mtr -6 example.com` may show completely different paths and different loss characteristics. If diagnosing a connectivity problem, test both explicitly — do not assume the IPv4 path result applies to IPv6.

ss and IPv6

`ss -tlnp` shows IPv4 listeners. `ss -tlnp6` shows IPv6. A service listening on `:::80` accepts both IPv4 and IPv6 connections on most Linux systems (dual-stack socket). A service listening on `0.0.0.0:80` accepts IPv4 only.

```
ss -tlnp | grep :80          # IPv4 listeners on port 80
ss -tlnp6 | grep :80         # IPv6 listeners on port 80
```

Link-local addresses require a zone ID

IPv6 link-local addresses (`fe80::/10`) require a zone identifier to be routable — the interface name appended with `%`.

```
ping6 fe80::1%eth0            # Ping a link-local address on
eth0
```

Without the zone ID, `ping6 fe80::1` returns `Invalid argument` or `Network unreachable`.

H — Cloud & virtualised environments

Cloud security groups, AWS NACLs, GCP firewall rules, and Azure NSGs operate at the hypervisor or VPC layer — outside the OS entirely. The OS has no visibility into them and no tools that can inspect them. This creates a class of failure where every local diagnostic tool says the configuration is correct, but traffic is still being blocked.

The fundamental problem

```
Internet . [Cloud firewall / security group] . Instance OS . Application
                          .
                  Blocks happen here.
                  ss, nft, tcpdump see nothing.
```

`ss -tlnp` shows the port open. `nft list ruleset` shows no blocking rules. The application is running. But connections from outside time out.

Confirming the block is external

The definitive test is `tcpdump` on the instance while attempting a connection from outside:

```
# On the instance — capture on the relevant port
tcpdump -nn -i eth0 port 8080

# From outside — attempt a connection
nc -zv <instance-public-ip> 8080
```

If tcpdump shows no packets arriving — the block is external to the OS. The packet never reached the instance. Look at the cloud firewall layer, not the OS.

If tcpdump shows the SYN arriving but no SYN-ACK leaving — the OS or a local firewall is blocking it. The tools in the Firewall & Filtering section apply.

If tcpdump shows the full handshake — the connection is reaching the application. The problem is at the application layer.

Where to look for external blocks

- **AWS**: EC2 security groups (stateful, per-instance), VPC Network ACLs (stateless, per-subnet). Both must permit the traffic. A security group allowing inbound port 443 is not enough if the NACL denies it.
- **GCP**: VPC firewall rules (stateful, applied by tag or service account). Check both ingress and egress rules.
- **Azure**: Network Security Groups attached to the NIC or subnet. Effective security rules (the merged view) are in the portal under the NIC's properties.
- **DigitalOcean / Hetzner / Linode**: Cloud firewall rules managed at the provider level, separate from the OS. Check the provider console — no OS tool will show them.
- **OpenStack**: Security groups on the port, plus network policy applied by the Neutron layer. `openstack security group list` and `openstack port show` from outside the instance.
- **On-premise virtualisation (VMware, Proxmox, KVM with OVS)**: Distributed virtual switches and port groups can have their own ACLs. If the VM's OS shows no block but traffic does not flow, check the virtual switch policy on the hypervisor host.

The source IP problem

Cloud instances typically have a private IP on the interface. The public IP is handled by NAT at the cloud provider level — the instance never sees it. This affects:

- `ss` and `tcpdump` — source addresses are the private IPs of the caller after cloud NAT, not the public IPs you may expect
- Security group rules referencing specific source IPs — the rule must reference the actual source as it appears after any NAT in the path

- Logging and audit — VPC flow logs capture the pre-NAT addresses; application logs capture the post-NAT addresses. They will not match without accounting for this

Egress filtering

Outbound connections from an instance can also be blocked externally. If a connection from the instance to an external service times out but the same connection works from elsewhere:

```
# Confirm the block is not local
ip route get <destination-ip>        # Is there a route?
curl --connect-timeout 5 https://destination.example.com   #
Does it time out?

# If tcpdump shows the SYN leaving but no SYN-ACK returning,
# the block is either at the cloud egress rules or the
destination firewall
tcpdump -nn -i eth0 host destination.example.com
```

Check cloud egress rules (security groups apply to outbound traffic too on AWS), and confirm the destination is not blocking the instance's public egress IP.

Instance metadata — finding attached security groups

```
# AWS — instance metadata service
curl http://169.254.169.254/latest/meta-data/security-groups

# GCP
curl
"http://metadata.google.internal/computeMetadata/v1/instance/n
etwork-interfaces/0/access-configs/0/external-ip"   -H
"Metadata-Flavor: Google"
```

The metadata service lets you confirm which security groups are attached to the instance without needing console access — useful when diagnosing from within the instance itself.

I — nftaudit integration

For automated nftables ruleset auditing — confirming that firewall policy matches intent rather than reading rules manually — see `nftaudit`.

`nftaudit` is an open source Go CLI tool with three commands:

```
nftaudit check   # Assert policy rules against the live
ruleset
nftaudit diff    # Compare two rulesets
nftaudit report  # Generate a structured audit report
```

Policy assertions are written in `nftaudit`'s purpose-built DSL, with exit codes designed for use in CI pipelines and automated compliance checks. See the nftables Pocket Reference for integration patterns.

J — Useful one-liners

Patterns that come up repeatedly and are worth keeping to hand.

```
# What process is using this port?
ss -tlnp | grep :8080
lsof -i :8080

# Is this port reachable from here?
nc -zv -w 3 example.com 443

# What IP does this name resolve to right now?
dig +short example.com

# How long until this certificate expires?
echo | openssl s_client -connect example.com:443 -servername
example.com 2>/dev/null \
  | openssl x509 -noout -dates

# What HTTP status is this endpoint returning?
curl -s -o /dev/null -w "%{http_code}" https://example.com
```

```
# Where is the latency on this path?
mtr -r -c 20 example.com

# What route would be used for this destination?
ip route get 8.8.8.8

# Is the connection tracking table close to full?
echo "$(cat /proc/sys/net/netfilter/nf_conntrack_count) /
$(cat /proc/sys/net/netfilter/nf_conntrack_max)"

# What is the path MTU to this host?
tracepath example.com

# Where is the time going in this HTTP request?
curl -s -o /dev/null -w "dns:%{time_namelookup}
connect:%{time_connect} tls:%{time_appconnect}
total:%{time_total}\n" https://example.com

# What certs is this server actually presenting?
echo | openssl s_client -connect example.com:443 -servername
example.com 2>/dev/null \
  | openssl x509 -noout -text | grep -A 1 "Subject Alternative
Name"

# Are there retransmissions in this capture?
tshark -r capture.pcap -Y "tcp.analysis.retransmission" -q -z
io,stat,5

# Watch NIC drops accumulate in real time
watch -n1 'ethtool -S eth0 | grep -i drop'
```

End of Network Diagnostics Pocket Reference