Network Troubleshooting

A Complete 10-Chapter Technical Support Course

Topics covered:

The "user can't reach X" diagnostic mindset · a practical OSI ladder

Local connectivity, DNS resolution, ping & traceroute

Port/service checks · firewalls · proxies, VPNs & NAT

Reading HTTP & TLS failures at the application layer

Capstone: three real connectivity tickets, triaged end to end

Exercises: 30 hands-on scenarios with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

01 The "User Can't Reach X" Mindset

02 A Practical OSI Model for Diagnosis

03 Local Connectivity: Interfaces, IP Configuration & the Default Gateway

04 DNS Resolution Troubleshooting

05 Testing the Path: ping and traceroute/tracert

06 Port & Service-Level Checks: telnet, curl & netcat

07 Firewalls & Blocked Traffic

08 Proxies, VPNs & NAT: When the Path Isn't What You Expect

09 Application-Layer Symptoms: HTTP & TLS-Specific Failures

10 Capstone: Triaging Three Real Connectivity Tickets

CHAPTER 1 OF 10

The "User Can't Reach X" Mindset

Network Troubleshooting

Chapter 1 · The "User Can't Reach X" Mindset

A ticket comes in: "I can't get to the app." One support engineer opens a terminal and starts working up from the bottom — check the network interface, check the cable, check the switch port, check the router — methodically confirming every layer is healthy before moving to the next. The other starts from the complaint itself: *what exactly can't this person reach, and what's actually different about their situation right now compared to everyone else's?* Both engineers might get to the right answer eventually. Only one of them gets there before the ticket queue backs up. This course is about becoming the second engineer.

Two Ways to Approach a Network

If you've worked through **Networking Fundamentals** (`net1`) on this site, you've already spent real time with the OSI model, IP addressing, routing, and how a network gets built and configured from the ground up. That knowledge is genuinely useful here — but it's not the same job as the one this course teaches.

NETWORKING FUNDAMENTALS (`NET1`)	NETWORK TROUBLESHOOTING (THIS COURSE)
Starts from	A blank network that needs designing and building
Starts from	A working network that has just stopped working for someone
Direction	Bottom-up — physical layer, then link, then network, then up
Direction	Symptom-out — start at the reported complaint, narrow from there
Goal	Understand every layer well enough to configure it correctly
Goal	Find which single layer is broken, as fast as the evidence allows

Neither approach is "better" in the abstract — they're built for different jobs. A network engineer designing a new office deliberately works bottom-up, because getting a layer wrong early causes problems in everything built on top of it. A support engineer staring at a single broken connection doesn't have that luxury: the network already works for almost everyone else, and starting a full bottom-up review of every layer for one user's complaint would be enormously wasteful. The faster path is narrowing down which layer is actually the problem *this time*, and ruling out everything else without checking it in full.

The Shape of a Connectivity Complaint

"I can't reach the app" is not one problem — it's a category that covers several genuinely different problems, each with a different fix. Before touching any diagnostic tool, the first job is narrowing down which shape this particular complaint actually has.

QUESTION TO ASK	WHY IT MATTERS
One user, or many?	One user points toward that user's own machine, credentials, or local network; many users pointing toward a shared cause — the server, a shared firewall rule, or DNS
One destination, or everywhere?	Can't reach one specific site/app but everything else works fine narrows the search dramatically compared to "the internet doesn't work at all"
Total failure, or degraded?	"Nothing loads at all" and "it loads, just very slowly" point at different layers — a total failure often means routing/DNS/firewall; slowness often means something further up the stack
Just started, or always broken?	Something that worked yesterday and doesn't today means *something changed* — a config push, an expired certificate, a new firewall rule — which is a genuinely different search than something that's simply never worked

These four questions alone often narrow a vague "it's broken" complaint down to one or two plausible causes before a single diagnostic command has been run — which is the entire point of asking them first.

> **DON'T NAME A CAUSE BEFORE YOU'VE SCOPED THE COMPLAINT**
>
> It's tempting to hear "can't reach the app" and jump straight to "sounds like a firewall issue" or "probably DNS." That's a guess dressed up as a diagnosis, and it sends you down one specific path before you know whether it's even the right path. Scope the complaint first — one user or many, one destination or everywhere, total or degraded, new or persistent — and let the answers point you toward a cause, rather than picking a cause and looking for evidence to support it.

A Concrete Example: One Symptom, Several Different Causes

Take the exact same complaint — "I can't reach the app" — and see how differently it resolves depending on what the scoping questions reveal:

- **One user, everywhere else fine, just started:** often a local DNS cache issue, a VPN misconfiguration, or a stale cached credential — see Chapter 4
- **Many users, one destination, just started:** often points at the server itself being down, or a firewall/routing change that just went live — see Chapters 3 and 7
- **One user, total failure, always broken for them:** often a routing or gateway misconfiguration on that user's own segment — see Chapter 3

- **Many users, degraded rather than total, no recent change:** often a proxy, VPN, or NAT path introducing latency rather than blocking traffic outright — see Chapter 8
- **One destination reachable by IP but not by name:** a DNS resolution problem specifically, not a connectivity problem at all — see Chapter 4

Five genuinely different root causes, all hiding behind the same three words. The scoping questions from the previous section are what separate them — and every later chapter in this course builds out one of these branches in real depth.

Here's a first, deliberately small piece of evidence — you'll learn to run and properly interpret this in Chapter 5, but even without the tool knowledge yet, notice how much a single line already narrows things down:

```
$ ping app.example.com
PING app.example.com (203.0.113.42): 56 data bytes
Request timeout for icmp_seq 0
Request timeout for icmp_seq 1
Request timeout for icmp_seq 2
```

This alone tells you the name resolved to an IP address (so DNS isn't the immediate problem) but no reply came back. It does *not* yet tell you whether the server is down, a firewall is silently dropping the traffic, or the path is simply congested — three different causes that all produce this exact same output. Chapter 5 covers exactly why `ping` can't distinguish between them on its own, and what to check next.

> **ASK "CAN THEY REACH ANYTHING AT ALL?" BEFORE "WHY CAN'T THEY REACH THIS ONE THING?"**
> A single extra question — can the user load any other website, or reach any other internal service? — often does more scoping work than any diagnostic tool. If everything else works, the problem is almost certainly specific to the one destination or one credential. If nothing works, the problem is almost certainly upstream of that destination entirely — their local connection, DNS, or a shared gateway.

What This Course Covers

A practical, layer-by-layer diagnostic toolkit built around real support tickets rather than protocol theory for its own sake: a triage-oriented way of thinking about the OSI model, checking local connectivity and IP configuration, DNS resolution troubleshooting, reading `ping` and `traceroute` correctly, port and service-level checks, diagnosing firewalls you can't see the rules for, proxies/VPNs/NAT complicating the picture, and application-layer symptoms that look like a network problem but aren't. The capstone applies all of it to three realistic connectivity tickets, start to finish.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why this chapter argues that a support engineer's "symptom-out" approach is a genuinely different job from a networking course's "bottom-up" approach, rather than just a faster version of the same thing.

View solution

EXERCISE 2

A ticket says "the app is down." List the four scoping questions from this chapter, and explain what a "yes" or "no" answer to each one would point you toward.

View solution

EXERCISE 3

A colleague sees the `ping` output in this chapter's example and says "the server must be down." Explain why this chapter says that conclusion is unsafe on its own, and name at least two other causes that would produce the exact same output.

View solution

CHAPTER 1 QUICK REFERENCE

- **Symptom-out, not bottom-up** — start from the reported complaint and narrow down, rather than reviewing every network layer from the physical wire up
- Distinct from **Networking Fundamentals** (`net1`), which builds and configures a network; this course diagnoses one that's already built and already broken
- Scope every complaint with four questions: **one user or many? one destination or everywhere? total failure or degraded? just started or always broken?**
- Don't name a likely cause before scoping the complaint — that's a guess wearing a diagnosis's clothes
- The same three words ("can't reach it") can hide genuinely different root causes — DNS, routing, firewall, server, or a slow path — that each need a different fix
- **Next chapter:** A Practical OSI Model for Diagnosis

CHAPTER 2 OF 10

A Practical OSI Model for Diagnosis

Network Troubleshooting

Chapter 2 · A Practical OSI Model for Diagnosis

Chapter 1's four scoping questions narrow a vague complaint down to a rough shape — one user or many, one destination or everywhere. They don't yet tell you *which layer of the network is actually broken*. That's this chapter's job: a version of the OSI model built specifically for triage, not for an exam — fewer layers, a deliberately practical starting point, and one honest deviation from the textbook order that matters more than it might seem.

Why Not Just Use the Textbook Seven Layers?

The full OSI model — Physical, Data Link, Network, Transport, Session, Presentation, Application — is precise and genuinely useful for understanding how networking software is built, which is exactly why **Networking Fundamentals** (`net1`) teaches it in full. But precision isn't what triage needs. In practice, Session and Presentation rarely come up as a distinct troubleshooting step, and Physical is often something you can't even check directly on a remote user's machine. Support diagnosis collapses the seven layers down to five practical categories, each tied to one concrete question and one or two tools you'll cover in depth over the next several chapters.

PRACTICAL LAYER	THE ONE QUESTION IT ANSWERS	COVERED IN
Local connectivity	Does this machine have a working network connection at all — interface up, IP assigned, gateway reachable?	Chapter 3
DNS resolution	Does the name resolve to the IP address it's actually supposed to?	Chapter 4
Network reachability	Can a packet get from here to that IP address at all?	Chapter 5
Port / service reachability	Is something actually listening and accepting connections on the specific port?	Chapter 6
Application response	Does the service reply with the right thing once a connection is made?	Chapters 7–9

The One Honest Deviation: Why DNS Comes Before "Network"

Strictly by OSI layer number, DNS is an application-layer protocol (it runs over the network, same as any other service) — so a purist ordering would test raw network reachability before ever touching DNS. This course's practical ladder puts DNS second anyway, right after local connectivity, and it's worth being upfront about why: almost every later test in this ladder needs a correct IP address to test *against*. If DNS resolves a name to the wrong IP, every network-reachability, port, and application test that follows will faithfully test the wrong destination and produce confusing, misleading results — not because those layers are broken, but because the address itself was wrong from the start. Checking DNS early isn't about OSI purity; it's about not wasting the rest of the ladder on a bad starting address.

TWO THINGS DELIBERATELY DON'T FIT IN THE LADDER AT ALL

Firewalls (Chapter 7) and proxies/VPNs/NAT (Chapter 8) aren't a single layer — a firewall can filter at the network layer, the transport layer, or inspect all the way up at the application layer, and a proxy or VPN can quietly change which path your traffic even takes before it reaches any of the layers above. Both get their own dedicated chapter specifically because they sit *beside* this ladder, capable of interfering with more than one rung of it at once, rather than living cleanly on one.

Working the Ladder: Where to Actually Start

A textbook bottom-up review starts at the physical layer — but as a support engineer, you usually can't see a user's cable or link light from where you're sitting, so that step is often not available to you at all. The practical starting point is one rung up: confirm local connectivity is basically sane (Chapter 3 covers exactly how, remotely), then work upward one rung at a time, stopping the moment you find the first rung that fails. The first failure you find is very often the actual problem — not always, but often enough that it's the right place to start looking closely before moving further up.

DON'T TEST THE TOP OF THE LADDER BEFORE THE BOTTOM

It's tempting to jump straight to the application — open a browser, try the exact thing the user tried, see it fail, and start reading application logs or debugging application code. If DNS or network reachability was the actual problem, none of that time was wasted on the real cause, because the application was never going to work regardless of what's inside it. Confirm each rung below "application" is fine first; it usually takes less time than one detour into the wrong codebase.

Working Example: Walking the Ladder on Chapter 1's Complaint

Chapter 1 left a ticket open: "I can't reach the app," with a `ping` that timed out. Suppose this time the ping actually succeeds — walking the ladder from the bottom:

```
$ ping app.example.com
PING app.example.com (203.0.113.42): 56 data bytes
64 bytes from 203.0.113.42: icmp_seq=0 ttl=54 time=11.204 ms
64 bytes from 203.0.113.42: icmp_seq=1 ttl=54 time=10.887 ms

$ telnet app.example.com 443
Trying 203.0.113.42...
telnet: Unable to connect to remote host: Connection refused
```

Local connectivity: fine, since the ping left this machine and came back. DNS: fine, since the name resolved to the expected IP. Network reachability: fine, since the ping actually got a reply this time, unlike Chapter 1's example. The first rung that fails is port/service reachability — something on the far end is actively refusing the connection on port 443, rather than the packet simply going nowhere. That's a genuinely different, much narrower problem than the one in Chapter 1, and Chapter 6 covers exactly what "connection refused" does and doesn't tell you about what's happening on the other end.

"REFUSED" AND "TIMED OUT" ARE NOT THE SAME FAILURE

A quick preview worth holding onto until Chapter 6: a refused connection means something on the far end actively responded to say no — usually nothing is listening on that port. A timeout means no response came back at all, which is more consistent with a firewall silently dropping the traffic or the packet never arriving. The two look similar at a glance but point in different directions.

What This Buys You

The practical ladder doesn't replace the four scoping questions from Chapter 1 — it comes after them. Scoping tells you roughly what kind of problem you're looking at (one user vs. many, new vs. persistent); the ladder tells you specifically which layer is responsible, by testing from the bottom and stopping at the first failure. Together they turn "I can't reach the app" into a precise, evidence-backed statement like "DNS and network reachability are both fine, but port 443 is refusing connections" — which is a genuinely different, far more actionable ticket than the one that came in.

Hands-On Exercises

EXERCISE 1

Explain why this chapter's practical ladder checks DNS resolution before raw network reachability, even though DNS is formally an application-layer protocol sitting above the network layer in the textbook OSI model.

View solution

EXERCISE 2

A colleague's first troubleshooting step for "I can't reach the app" is opening the app directly and reading its logs. Explain why this chapter would consider that risky as a first step, and what to check before it.

View solution

EXERCISE 3

In this chapter's worked example, DNS resolves correctly and the ping succeeds, but a connection to port 443 comes back "connection refused." Explain which rung of the ladder this points to, and why it's a narrower, different problem than Chapter 1's timed-out ping.

View solution

CHAPTER 2 QUICK REFERENCE

- The practical ladder: **local connectivity → DNS resolution → network reachability → port/service reachability → application response**
- Session and Presentation are collapsed away; Physical is skipped as a starting point because it's rarely checkable remotely
- **DNS deliberately comes before raw network reachability** — every later test needs a correct IP to test against
- Firewalls and proxies/VPNs/NAT don't fit cleanly into one rung — they can interfere with several at once, hence their own dedicated chapters (7 and 8)
- Work the ladder bottom-up and **stop at the first failing rung** — don't debug the application before confirming everything below it is fine
- **"Refused" ≠ "timed out"** — refused means an active no; timed out means no response at all (Chapter 6 covers the difference in full)
- **Next chapter:** Local Connectivity: Interfaces, IP Configuration & the Default Gateway

CHAPTER 3 OF 10

Local Connectivity: Interfaces, IP Configuration & the Default Gateway

Network Troubleshooting

Chapter 3 · Local Connectivity: Interfaces, IP Configuration & the Default Gateway

Chapter 2 named local connectivity as the practical starting rung of the ladder — the one closest to physical, but the first one you can actually check remotely. This chapter is entirely about that one rung: reading a machine's own IP configuration, recognizing the signature of a machine that never got a proper address at all, and understanding exactly what the default gateway is and why so many "nothing works" complaints trace back to it.

What "Local Connectivity" Actually Confirms

This rung isn't about the cable or the Wi-Fi signal itself — as Chapter 2 noted, that's rarely something you can check remotely. It's about everything the operating system's own network stack can report: whether the interface is up, whether it has a valid IP address and subnet mask, and whether it has a default gateway configured at all. Three commands cover almost every case you'll see:

PLATFORM	COMMAND	WHAT IT SHOWS
Windows	`ipconfig /all`	Every adapter, its IP, subnet mask, default gateway, and DNS servers
Linux	`ip addr` and `ip route`	Interface addresses (`ip addr`) and the default gateway (`ip route`) separately
macOS	`ifconfig` and `netstat -nr`	Same information, older BSD-style tooling

Reading a Healthy Configuration

```
C:\> ipconfig /all

Ethernet adapter Ethernet:
   Connection-specific DNS Suffix  . : corp.example.com
   IPv4 Address. . . . . . . . . . . : 192.168.1.42
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.1.1
```

Three fields matter most here: a real IP address in the range the local network actually uses, a subnet mask consistent with that network, and — critically — a default gateway that's actually populated. All three present and sensible-looking is a good sign, but "sensible-looking" isn't the same as "confirmed working" — the next section covers the one extra step that turns this from a good sign into an actual confirmation.

The Telltale Sign of a Broken Configuration: Link-Local Addresses

When a machine can't reach a DHCP server to be assigned a real address, most operating systems fall back to self-assigning one from the **169.254.0.0/16** range — commonly called a link-local or APIPA (Automatic Private IP Addressing) address. It's designed to let machines on the same broken segment still talk to each other, but it means the machine has no real address, no gateway, and no way to reach anything outside its own local segment:

```
Ethernet adapter Ethernet:
   IPv4 Address. . . . . . . . . . . : 169.254.23.107
   Subnet Mask . . . . . . . . . . . : 255.255.0.0
   Default Gateway . . . . . . . . . : (none)
```

Seeing a `169.254.x.x` address is one of the most immediately conclusive pieces of evidence you'll find at this rung: it means DHCP failed, full stop — the cause could be a disconnected cable, a misconfigured switch port, a DHCP server outage, or a wireless authentication failure, but the fix is squarely at this layer, and there's no point testing DNS, ping, or anything further up the ladder until it's resolved.

The Default Gateway: The Machine's Only Way Out

The default gateway is the router a machine sends every packet through when the destination isn't on its own local subnet — which, for almost any real complaint ("I can't reach the app," "the internet's down"), is nearly everything. If the gateway itself can't be reached, nothing beyond the local network can be either, regardless of how healthy DNS or the destination server are. That makes the gateway worth testing directly, separately from testing the actual destination:

```
$ ping 192.168.1.1
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=0.412 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.398 ms
```

A gateway that responds to a ping like this is a genuinely useful data point: it separates "something is wrong with my own local segment" from "something is wrong further out." If the gateway responds fine but the actual destination in Chapter 5's tests doesn't, the problem is somewhere beyond this machine's own local network — a routing issue further upstream, the destination itself, or DNS. If the gateway doesn't respond at all, the problem is much more likely to be local: a cable, a switch port, a VLAN misconfiguration, or the gateway device itself being down.

> **A GATEWAY THAT DOESN'T ANSWER A PING ISN'T AUTOMATICALLY DOWN**
>
> Plenty of routers are deliberately configured to ignore ICMP ping requests as a security measure, even while routing traffic perfectly normally. A silent gateway is a real data point worth noting, but on its own it's not fully conclusive — the same "refused vs. timed out" ambiguity from Chapter 2 applies here too. If the gateway stays silent, the next real test is whether traffic actually routed through it succeeds or fails (Chapter 5), not just whether it answers ping.

WHAT YOU SEE	WHAT IT MEANS
Real IP, gateway set, gateway pings	Local connectivity confirmed — move to Chapter 4 (DNS)
169.254.x.x address, no gateway	DHCP failure — a genuinely local problem, stop here and investigate the cause
Real IP, gateway set, gateway silent	Inconclusive on its own — check whether traffic actually routes through it before assuming the gateway is down
No IP address, interface reports "down"	Physical/link-layer problem — often needs someone with physical access to the machine

> **YOU OFTEN CAN'T RUN THESE COMMANDS YOURSELF**
>
> Unlike checking a server you have shell access to, a user's own laptop usually means asking them to run `ipconfig /all` (or `ip addr` / `ifconfig`) themselves and read the output back to you, or send a screenshot. Give them the exact command rather than a vague "check your network settings" — it's the difference between getting the three fields that actually matter and getting a description of whatever the user's operating system happened to show them.

Working Example: Continuing the Ticket

Back to the running ticket from Chapters 1 and 2. Before even reaching for `ping` against the app itself, the user reads back their own `ipconfig /all` output over the phone: a real-looking `192.168.1.42` address, subnet mask `255.255.255.0`, gateway `192.168.1.1` — no `169.254.x.x` address in sight. A quick ping to

that gateway from their machine gets replies. Local connectivity confirmed, in well under a minute, with no need to touch the application at all — exactly the kind of fast, evidence-based elimination this course is built around. The ladder moves up to Chapter 4: does the app's name actually resolve to the right address?

Hands-On Exercises

EXERCISE 1

A user reads you their IP configuration: address `169.254.87.12`, subnet mask `255.255.0.0`, no default gateway listed. Explain what this specific pattern means, and why it would be a waste of time to test DNS or ping the app at this point.

View solution

EXERCISE 2

Explain why this chapter treats "the default gateway doesn't respond to a ping" as inconclusive rather than as proof the gateway is down, and what the next step should be when that happens.

View solution

EXERCISE 3

Explain why this chapter recommends testing the default gateway separately from testing the actual destination (e.g. the app server), rather than only testing the destination directly.

View solution

CHAPTER 3 QUICK REFERENCE

- Check with `ipconfig /all` (Windows) or `ip addr` + `ip route` (Linux) — interface state, IP address, subnet mask, default gateway
- A **169.254.x.x** (link-local/APIPA) address means DHCP failed — a genuinely local problem, no point testing further up the ladder yet
- The **default gateway** is the only way out of the local subnet — if it's unreachable, nothing beyond the local network is reachable either
- Ping the gateway **separately** from the actual destination — it distinguishes "problem is local" from "problem is further out"
- A silent gateway isn't automatically down — many routers deliberately ignore ICMP; treat it as inconclusive, not proof
- You usually can't run these commands yourself on a user's machine — give them the exact command to run and read back

- **Next chapter:** DNS Resolution Troubleshooting: nslookup, dig & Common Failure Patterns

CHAPTER 4 OF 10

DNS Resolution Troubleshooting

Network Troubleshooting

Chapter 4 · DNS Resolution Troubleshooting

Local connectivity confirmed in Chapter 3 means this machine can reach its gateway and, through it, the rest of the network. The next rung, per Chapter 2's ladder, is DNS: does the name the user is actually typing resolve to the IP address it's supposed to? This chapter covers the two tools that answer that question, the handful of failure patterns that cover almost every real DNS ticket, and one technique — bypassing local DNS entirely — that's often the single fastest way to tell a local infrastructure problem apart from a genuine, global DNS issue.

How Resolution Actually Happens

When a name is looked up, the request doesn't necessarily go out to "the internet" fresh every time. It typically passes through several layers, any of which might already hold — or wrongly hold — an answer: the application's own cache, the operating system's resolver cache, the configured DNS server (often a local corporate resolver), and only then, if nothing already has an answer, a chain of recursive lookups out to the authoritative servers for that domain. Every one of those layers can cache a result for a period of time set by that record's **TTL** (time to live) — which matters a great deal diagnostically, and gets its own warning later in this chapter.

Two Tools, One Underlying Answer

TOOL	PLATFORM	NOTES
`nslookup`	Windows (also available on Linux/macOS)	Simple, widely available, but its plain-English phrasing hides the underlying DNS response code
`dig`	Linux, macOS	More verbose, shows the actual DNS response status directly (`NOERROR`, `NXDOMAIN`, `SERVFAIL`, etc.)
`Resolve-DnsName`	Windows PowerShell	PowerShell's own modern equivalent, with structured output — worth knowing if you've been through this site's own PowerShell Fundamentals course

SAME UNDERLYING ANSWER, GENUINELY DIFFERENT-LOOKING OUTPUT

A single DNS response code can look completely different depending on which tool reports it. A name that doesn't exist produces a literal `status: NXDOMAIN` line in `dig`'s output, but shows up in `nslookup` as a plain-English sentence: `"can't`

`find ... : Non-existent domain."` Recognizing that both are reporting the exact same underlying DNS failure — not two different problems — matters more than memorizing either tool's specific wording.

Reading a Healthy Resolution

```
C:\> nslookup app.example.com
Server:   dns.corp.example.com
Address:  10.0.0.53

Non-authoritative answer:
Name:     app.example.com
Address:  203.0.113.42
```

Two things worth noting even in a healthy result: which server answered (`dns.corp.example.com` here — the locally configured resolver, not necessarily the domain's own authoritative server), and that the answer is marked "non-authoritative" — meaning it came from a cache somewhere along the chain, not fresh from the domain's own DNS servers. Neither is a problem on its own; both become relevant the moment the answer looks wrong.

The Four Failure Patterns

PATTERN	WHAT IT MEANS
NXDOMAIN	The name genuinely doesn't exist as far as this DNS server knows — a typo, a domain that was never registered, or a record that was deleted
SERVFAIL	The DNS server tried to answer and failed — often a misconfigured zone on the authoritative server, or a DNSSEC validation failure; the name might be entirely valid
Timeout / no response	The DNS server itself couldn't be reached at all — this is actually a network-reachability problem to the DNS server, not a DNS problem in the strict sense
Resolves, but to an unexpected IP	Often a stale cached answer that hasn't expired yet, or a genuine split-horizon DNS setup returning a different, equally "correct" answer depending on where the query came from

```
$ dig app.example.com

;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 4821
;; QUESTION SECTION:
;app.example.com.        IN  A

;; AUTHORITY SECTION:
example.com.    3600  IN  SOA  ns1.example.com. admin.example.com.
```

The line to look for is always `status:` — in `dig`'s output it's stated explicitly, which is exactly why it's the more useful tool once you already suspect a DNS problem rather than just doing a routine lookup.

Bypassing Local DNS to Isolate the Problem

One of the most useful single techniques in this chapter: query a known public DNS server directly, instead of whatever's configured by default, and compare the two answers.

```
$ dig app.example.com +short
203.0.113.77

$ dig @8.8.8.8 app.example.com +short
203.0.113.42
```

Two different answers for the same name is a genuinely strong signal: the local resolver is holding a stale or otherwise incorrect answer, while the public internet's view of that domain is already correct — or, less often, the reverse, if a change was only made locally and hasn't propagated out yet. Either way, this single comparison usually tells you within seconds whether the problem is specific to local DNS infrastructure or a real, global issue with the domain itself.

"IT'S FIXED" DOESN'T MEAN EVERYONE SEES IT FIXED YET

A DNS record has a TTL — a length of time other resolvers are allowed to keep serving their own cached copy of it before checking again. Updating a record doesn't retroactively update every cache that already has the old answer; those caches keep serving the stale value until their own TTL expires. "I already fixed it" and "the user still can't reach it" are both true at the same time far more often than it seems like they should be — this isn't a sign the fix didn't work, it's the TTL doing exactly what it's designed to do.

SPLIT-HORIZON DNS: TWO CORRECT ANSWERS, NOT ONE WRONG ONE

Some organizations deliberately configure DNS to return different answers depending on where the query comes from — an internal IP for requests from inside the corporate network, a public IP for everyone else. A ticket like "it

works from home but not from the office" (or the reverse) can be exactly this working as designed, not a fault at all. Worth ruling in or out early, since chasing it as a bug wastes real time on something that isn't broken.

Working Example: Continuing the Ticket

The ticket from Chapters 1–3 continues: local connectivity confirmed, so the ladder moves to DNS. `nslookup app.example.com` from the user's machine returns `203.0.113.77` — but a quick `dig @8.8.8.8 app.example.com +short` from your own machine returns `203.0.113.42`, a different address. That mismatch alone is the finding: this user's local DNS resolver is serving a stale answer, most likely a caching issue on the corporate resolver or a TTL that hasn't yet expired since a recent change. The fix belongs to DNS infrastructure, not the application — and the ladder can move to Chapter 5 once a correct answer is confirmed, to check whether the right IP is actually reachable.

Hands-On Exercises

EXERCISE 1

A colleague sees an `nslookup` failure message and a `dig` failure message that look completely different and assumes they're two different problems. Explain why this chapter says that assumption can be wrong, using the specific example from this chapter.

View solution

EXERCISE 2

Explain what it means when `dig app.example.com +short` and `dig @8.8.8.8 app.example.com +short` return two different IP addresses, and why running both is more useful than running only the first one.

View solution

EXERCISE 3

A user says "you told me this was fixed yesterday, but I still can't reach it correctly." Explain two genuinely different, non-buggy explanations from this chapter that could both make this true at the same time.

View solution

CHAPTER 4 QUICK REFERENCE

- **nslookup** (Windows/cross-platform) and **dig** (Linux/macOS, more verbose) both report the same underlying DNS response — just phrased differently
- Four failure patterns: **NXDOMAIN** (doesn't exist), **SERVFAIL** (server-side failure), **timeout** (DNS server unreachable — a network problem, not a DNS problem), **wrong-looking answer** (stale cache or split-horizon)
- **Bypass local DNS** — query a public resolver directly (`dig @8.8.8.8 ...`) and compare — a fast way to isolate local DNS infrastructure from a genuine global issue
- **TTL means a fix isn't instantly visible everywhere** — stale caches keep serving the old answer until their own TTL expires, by design
- **Split-horizon DNS** can make "different answer from a different location" correct behavior, not a bug
- **Next chapter:** Testing the Path: ping and traceroute/tracert — What They Actually Prove

CHAPTER 5 OF 10

Testing the Path: ping and traceroute/tracert

Network Troubleshooting

Chapter 5 · Testing the Path: ping and traceroute/tracert — What They Actually Prove

Both tools in this chapter have already made cameo appearances — Chapter 1's timed-out ping, Chapter 3's gateway ping. This chapter goes deeper on what they're actually doing under the hood, because a surprising amount of wasted diagnostic time comes from treating a ping reply as proof of more than it actually proves, or reading a broken-looking traceroute as more conclusive than it really is.

What Ping Actually Tests

`ping` sends an ICMP echo request and waits for an ICMP echo reply. That's the entire test — it says nothing directly about whether any actual service on that machine is working. A web server can be completely healthy and serving every request correctly while its host doesn't answer ping at all (ICMP disabled by policy, same as Chapter 3's gateway example); equally, a host can answer ping instantly while the one service you actually care about has crashed. Ping tests reachability at the network layer, not health at the application layer — conflating the two is one of the most common false conclusions in network troubleshooting.

Reading Ping Output In Depth

```
$ ping -c 4 app.example.com
PING app.example.com (203.0.113.42): 56 data bytes
64 bytes from 203.0.113.42: icmp_seq=0 ttl=54 time=11.204 ms
64 bytes from 203.0.113.42: icmp_seq=1 ttl=54 time=10.887 ms
64 bytes from 203.0.113.42: icmp_seq=2 ttl=54 time=12.013 ms
64 bytes from 203.0.113.42: icmp_seq=3 ttl=54 time=11.442 ms

--- app.example.com ping statistics ---
4 packets transmitted, 4 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 10.887/11.387/12.013/0.421 ms
```

- **Packet loss %** — the single most important summary line; any sustained loss above 0% on a healthy path is worth investigating further, even if most packets get through
- **Round-trip time (min/avg/max)** — high or wildly varying latency (a large gap between min and max, called jitter) can point toward congestion somewhere on the path, even when every packet technically

arrives

- **TTL in the reply** — a rough hint at how many hops away the destination is; a TTL value much lower than expected for a "local" destination can be a sign traffic is taking an unexpectedly long path

PLATFORM	DEFAULT BEHAVIOR	TO LIMIT THE COUNT
Windows	Sends exactly 4 packets and stops automatically	`ping -n 10 app.example.com`
Linux / macOS	Runs continuously until interrupted with Ctrl+C	`ping -c 4 app.example.com`

Traceroute: Watching the Path Hop by Hop

Where ping only reports the final result, `traceroute` (Linux/macOS) and `tracert` (Windows) reveal every router along the way. The mechanism is a genuinely clever trick with a packet's TTL field: the first probe is sent with TTL set to 1, so the very first router along the path decrements it to 0 and sends back an ICMP "time exceeded" message — revealing itself — instead of forwarding the packet further. The next probe uses TTL 2, revealing the second router, and so on, one hop further with each round, until a probe finally reaches the actual destination.

WINDOWS AND LINUX DON'T USE THE SAME UNDERLYING PROTOCOL BY DEFAULT

`tracert` on Windows sends ICMP echo requests, the same protocol family as ping. Traditional `traceroute` on Linux/macOS sends UDP packets to a high, normally-unused port by default, relying on a "port unreachable" ICMP response at the final hop to detect arrival. This matters in practice: a firewall that blocks UDP but allows ICMP (or the reverse) can make one tool succeed and the other fail on the exact same path, for reasons that have nothing to do with the actual destination being reachable.

Reading Traceroute Output

```
$ traceroute app.example.com
 1  192.168.1.1 (192.168.1.1)  0.412 ms  0.389 ms  0.375 ms
 2  10.20.0.1 (10.20.0.1)  2.104 ms  2.041 ms  1.998 ms
 3  * * *
 4  203.0.42.9 (203.0.42.9)  8.221 ms  8.109 ms  8.033 ms
 5  203.0.113.42 (203.0.113.42)  11.204 ms  11.187 ms  11.093 ms
```

Each numbered line is one hop, with up to three round-trip times from three separate probes sent to it. Hop 3 shows three asterisks — no response at all from that specific router — yet hop 4 responds normally and the trace reaches the actual destination at hop 5. That's a genuinely common, non-broken pattern: many routers are configured to rate-limit or simply not generate the ICMP "time exceeded" messages traceroute depends on, even while forwarding the actual traffic through them perfectly normally.

A SILENT HOP IN THE MIDDLE ISN'T PROOF THE PATH IS BROKEN THERE

The same "refused vs. timed out" and "silent gateway" ambiguity from earlier chapters applies to traceroute too, and it's worth being extra careful here because a wall of asterisks in the middle of a trace *looks* dramatic. What actually matters is whether the trace reaches the destination at all, and whether packet loss or timeouts persist consistently at or beyond a specific hop across repeated runs — not a single silent hop with normal hops on either side of it.

What Neither Tool Proves

Two honest limits worth carrying into every later chapter. First, neither tool says anything about the application layer — a perfect ping and a clean traceroute both stop exactly at "the network path works," which is Chapter 6 and 7's job to build on, not this chapter's. Second, the path a traceroute reveals is only the *outbound* path — return traffic can genuinely take a different route back (asymmetric routing is common on real networks), so a trace that looks broken partway through doesn't necessarily mean the return path is broken in the same place, or at all.

A TRACEROUTE IS GOOD EVIDENCE TO HAND OFF, EVEN MID-INVESTIGATION

If a trace consistently stops or starts losing packets at a specific hop that belongs to another team or provider's network, that's concrete, specific evidence worth escalating with — "traffic reaches hop 4 fine every time but never gets a response past hop 5, which resolves to a range owned by [provider]" is a far more useful handoff than "the network seems slow," and it points the next person directly at where to start looking.

Working Example: Continuing the Ticket

Chapter 4 confirmed the correct IP is `203.0.113.42` once the user's DNS cache issue was accounted for. A ping directly to that IP now returns clean, consistent replies with 0% loss — network reachability confirmed. A traceroute shows one silent hop partway through, but reaches the destination normally, consistent with this chapter's own "silent hop, normal arrival" pattern rather than a real problem. The ladder moves to Chapter 6: with the network path confirmed healthy, is the actual application port open and accepting connections?

Hands-On Exercises

EXERCISE 1

A colleague says "the server must be healthy, it replies to ping instantly." Explain, using this chapter's own reasoning, why that conclusion goes further than the evidence actually supports.

View solution

EXERCISE 2

Explain why Windows' `tracert` and Linux's `traceroute` can produce different results for the exact same network path, even when nothing about the destination itself has changed.

View solution

EXERCISE 3

A traceroute shows asterisks at hop 6, but hops 5 and 7 respond normally and the trace reaches the destination. Explain why this chapter says that alone isn't proof of a problem at hop 6, and what would actually be more convincing evidence.

View solution

CHAPTER 5 QUICK REFERENCE

- **Ping tests reachability, not application health** — a host can be fully healthy and ignore ping, or answer ping while its actual service is down
- Windows ping sends 4 packets and stops; Linux/macOS ping runs until interrupted (`-c N` to limit)
- **Traceroute reveals each hop via the TTL-increment trick** — each router along the path is exposed one at a time as TTL expires
- **Windows `tracert` uses ICMP; Linux/macOS `traceroute` uses UDP by default** — a firewall can make one succeed and the other fail on the identical path
- A silent hop in the middle of a trace, with normal hops before and after it, usually just means that router doesn't generate "time exceeded" replies — not that the path is broken there
- Traceroute only shows the **outbound** path — the return path can genuinely differ (asymmetric routing)
- **Next chapter:** Port & Service-Level Checks: telnet, curl & netcat

CHAPTER 6 OF 10

Port & Service-Level Checks: telnet, curl & netcat

Network Troubleshooting

Chapter 6 · Port & Service-Level Checks: telnet, curl & netcat

Chapter 5 closed out with the network path confirmed clean — a healthy ping, a traceroute that reaches the destination. That confirms packets can get to the right machine. It says nothing about whether anything on that machine is actually listening on the specific port the application needs. This chapter is entirely about that one narrow question, and about reading the difference between an active "no" and total silence — a distinction that's come up in passing since Chapter 2, and finally gets its full treatment here.

Why the Network Layer Passing Isn't the Whole Answer

A server can respond to ping perfectly while the specific service you care about — a web server on port 443, a database on port 3306 — has crashed, was never started, or is deliberately firewalled off from where you're testing. Confirming the network path is healthy narrows the search meaningfully, but the only way to confirm the actual port is to try connecting to it directly, which is what all three tools in this chapter do, in slightly different ways.

Three Tools for One Job

TOOL	WHAT IT'S GOOD FOR	WORTH KNOWING
`telnet host port`	The classic, simplest way to attempt a raw TCP connection to one port	No longer installed by default on Windows or many minimal Linux distributions — often needs enabling or installing first
`nc` (netcat)	The same job as telnet, plus fast scriptable checks across many ports with `-z` (zero-I/O, no data sent)	Usually available by default on Linux/macOS; a genuinely flexible, scriptable alternative
`curl -v`	Shows the same TCP connect step explicitly, as the first phase of an HTTP request, before anything HTTP-specific happens	Useful when the target is a web service specifically — Chapter 9 goes much further into reading the rest of curl's output

TELNET'S CLIENT IS OFTEN MISSING ENTIRELY TODAY

Modern Windows ships the telnet client as an optional feature, disabled by default, and a number of minimal Linux distributions leave it out entirely to reduce their attack surface. Don't assume it's there — `nc` or `curl` are, in practice, more reliably available on a random machine today.

Reading the Three Outcomes

```
$ nc -zv app.example.com 443
Connection to app.example.com 443 port [tcp/https] succeeded!

$ nc -zv app.example.com 8080
nc: connect to app.example.com port 8080 (tcp) failed: Connection refused

$ nc -zv app.example.com 22
(hangs with no response — Ctrl+C to cancel)
```

OUTCOME	WHAT IT MEANS
Connects immediately	Something is actively listening and accepting connections on that port — the port/service rung is confirmed
"Connection refused"	An active reply came back saying no — either nothing is listening on that port, or something explicitly rejected the connection
Hangs / times out	No response of any kind — consistent with a firewall silently dropping the connection attempt, or the packet never arriving at all

"REFUSED" NARROWS IT TO TWO CANDIDATES, NOT ONE

A refused connection is a genuinely useful, active signal — but on its own it doesn't fully distinguish between "nothing is listening on this port" (the operating system itself replies, since no process owns that port) and "a firewall is explicitly configured to reject rather than silently drop this traffic" (which can produce the exact same client-side message). Both are more informative than a silent timeout, but telling them apart usually needs a second data point — often access to the machine itself, or the firewall's own logs, which Chapter 7 covers.

curl's Connection Phase

For a web service specifically, `curl -v` shows the raw TCP connect step explicitly, before anything HTTP-related begins:

```
$ curl -v https://app.example.com
*   Trying 203.0.113.42:443...
* Connected to app.example.com (203.0.113.42) port 443
* ALPN: offers h2,http/1.1
...
```

Even without reading any further, the "Trying..." and "Connected to..." lines alone confirm the exact same thing `nc -zv` would — this chapter's own narrow question. Everything curl prints after that point — the TLS handshake, the request headers, the response status code — belongs to Chapter 9, once the port itself is confirmed open.

CHECK SEVERAL COMMON PORTS AT ONCE WITH A LOOP

`nc -z`'s speed makes it easy to script a quick sweep instead of testing one port at a time by hand:

```
$ for port in 22 80 443 3306; do
    nc -zv -w 2 app.example.com $port
done
```

Working Example: Closing Out the Ticket

Back to the ticket that's run since Chapter 1. With DNS corrected (Chapter 4) and the network path confirmed clean (Chapter 5), a final check confirms port 443 as well:

```
$ nc -zv 203.0.113.42 443
Connection to 203.0.113.42 443 port [tcp/https] succeeded!
```

Every rung of the ladder — local connectivity, DNS, network reachability, and now port reachability — checks out clean. Which is, in its own way, the actual finding: this ticket's real root cause was the stale DNS answer identified back in Chapter 4. Once that single rung was corrected, everything above it was already fine, and there was never a firewall or application-layer problem to chase at all. That's exactly the payoff the ladder is built for — it would have been easy to spend far longer investigating firewalls or application logs for a problem that was actually settled two chapters ago.

Hands-On Exercises

EXERCISE 1

Explain why confirming a clean ping and a complete traceroute (Chapter 5) doesn't yet confirm that a specific application on that server is working, and what this chapter's checks add on top of that.

View solution

EXERCISE 2

A port check comes back "connection refused." Explain why this chapter says that doesn't fully pinpoint the cause on its own, and name the two candidate explanations it gives.

View solution

EXERCISE 3

Explain what this chapter identifies as the real root cause of the running ticket from Chapters 1 through 6, and why confirming every later rung of the ladder was still worth doing even after that root cause was already found in Chapter 4.

View solution

CHAPTER 6 QUICK REFERENCE

- A healthy network path (Chapter 5) doesn't confirm the specific port is open — that needs a direct connection attempt
- **telnet, nc, and curl -v** all attempt the same raw TCP connection; telnet is increasingly not installed by default
- **Connects immediately** = something is listening and accepting; **refused** = an active "no"; **hangs/times out** = no response at all
- A refused connection narrows the cause to two candidates — nothing listening, or an explicit firewall rejection — not one; telling them apart usually needs Chapter 7
- `nc -z` scripts easily into a quick sweep across several ports at once
- **Next chapter:** Firewalls & Blocked Traffic: Diagnosing a Block You Can't See the Rules For

CHAPTER 7 OF 10

Firewalls & Blocked Traffic

Network Troubleshooting

Chapter 7 · Firewalls & Blocked Traffic: Diagnosing a Block You Can't See the Rules For

Chapter 6 left one question open: a refused connection could mean nothing is listening on that port, or it could mean a firewall explicitly rejected it — and from a plain client-side test, the two can look identical. This chapter is about diagnosing firewall involvement specifically, under the constraint most support engineers actually work under: you very often can't see the firewall's own rules or logs at all, because it belongs to another team, a cloud provider's own console, or an ISP.

Two Ways a Firewall Can Say No

A firewall rule that blocks traffic can do it two genuinely different ways, and which one it uses shapes what you'll actually see from the client side:

FIREWALL BEHAVIOR	WHAT THE CLIENT SEES
DROP (silent)	The packet is discarded with no response of any kind — indistinguishable, from the client's side, from a timeout caused by anything else (Chapter 5's "no reply" scenarios)
REJECT (explicit)	An active reply is sent back — typically a TCP reset or an ICMP "administratively prohibited" message — which looks the same as Chapter 6's "connection refused" from a port with nothing listening on it

Both behaviors are entirely legitimate, deliberate configuration choices — DROP is often preferred specifically because it gives an attacker less information (a silent void looks the same whether a port is closed or actively blocked), while REJECT gives a faster, cleaner failure for legitimate traffic that isn't allowed. Distinguishing which one actually happened, and telling a firewall REJECT apart from a genuinely closed port, needs a packet-capture tool like `tcpdump` or Wireshark to inspect the actual ICMP response — deliberately out of scope for this course, but worth knowing the limit exists rather than assuming a simple client tool can always tell you.

Symptom Patterns That Point at a Firewall

PATTERN	WHY IT POINTS AT A BOUNDARY RULE
Works from one network, not another	The service itself is unlikely to behave differently by source — a rule scoped to where the traffic originates is a much more likely explanation

PATTERN	WHY IT POINTS AT A BOUNDARY RULE
One port blocked, others on the same host fine	A blanket host outage would affect every port; a single port failing while others succeed points at a port-specific rule
Worked yesterday, no application deploy happened	A firewall or security-group rule change is a common, easy-to-miss cause precisely because it never shows up in an application's own deploy history
Fails consistently, everywhere, always	Less diagnostic on its own — equally consistent with a firewall block or genuinely nothing listening (Chapter 6's own ambiguity)

Testing From Multiple Vantage Points

The single most useful technique available when you can't see the rules yourself is testing the exact same port from more than one location and comparing the results — the same instinct as Chapter 1's own scoping questions ("one user or many, one destination or everywhere"), now applied specifically to isolating a boundary rule. A port that opens cleanly from inside the same network as the server, but times out from outside it, is strong evidence for a boundary firewall or a cloud security group specifically — not the service itself, which is demonstrably working for at least one source.

```
# From inside the same network as the server
$ nc -zv 10.0.4.15 8443
Connection to 10.0.4.15 8443 port [tcp/*] succeeded!

# From outside that network
$ nc -zv 10.0.4.15 8443
(hangs with no response — Ctrl+C to cancel)
```

Cloud environments make this pattern especially common: a AWS Security Group or Azure NSG can silently restrict a port to a specific source range, and a support engineer without console access to that layer may have no visibility into the rule at all — only its effect, which is exactly why comparing results from multiple vantage points is often the fastest available path to a confident answer.

"I CAN'T FIND A FIREWALL RULE" ISN'T THE SAME AS "THERE ISN'T ONE"

Real environments frequently stack several independent firewall layers — a host-based firewall on the server itself, a network appliance, and a cloud security group, any of which could be the one actually blocking traffic. Having checked one layer and found nothing doesn't rule out a block at a layer you don't have visibility into at all. Absence of evidence, in a layer you can't actually see, is not evidence of absence.

WHEN YOU CAN'T SEE THE RULES, GIVE THE FIREWALL ADMIN EXACT EVIDENCE INSTEAD

A vague "it's blocked somewhere" ticket is slow for someone else to act on. Handing off the exact source IP, destination IP, port, protocol, and timestamp of a failed attempt lets whoever owns the firewall search their own logs

for that precise combination directly — turning a vague investigation into a quick lookup, the same principle as Chapter 5's traceroute handoff.

Working Example: A Fresh Ticket

A new ticket, unrelated to the one resolved in Chapter 6: users report an internal reporting app is unreachable when connecting from home, but works fine in the office. DNS and network reachability both check out from every location tested. Testing the port itself from multiple vantage points narrows it quickly:

- From the office LAN: port opens immediately
- Connected via the corporate VPN, from home: port opens immediately
- Connecting directly from home, no VPN: the port times out every time

This pattern points precisely at a boundary rule that permits traffic from the office network and the VPN's own address range, but not from the general internet — consistent with a port-specific, source-scoped firewall rule. Worth being honest about the next step: this could be a genuine misconfiguration, or it could be entirely intentional access control — this app may simply have never been meant to be reachable without the VPN. The right next step is confirming which one it is with whoever owns that access policy, not assuming either answer and proceeding to "fix" something that might not be broken at all.

Hands-On Exercises

EXERCISE 1

Explain the difference between a firewall's DROP and REJECT behavior, and why DROP is a deliberate security choice rather than a misconfiguration.

View solution

EXERCISE 2

A colleague checks the server's own host firewall, finds no blocking rule, and concludes the connection can't be firewall-related. Explain why this chapter says that conclusion isn't safe.

View solution

EXERCISE 3

In this chapter's worked example, the port opens from the office and over VPN but not from a direct home connection. Explain what this pattern points to, and why the chapter says it might not actually be a bug at all.

📄 View solution

CHAPTER 7 QUICK REFERENCE

- **DROP** = silent, indistinguishable from a plain timeout; **REJECT** = an active reply, indistinguishable from "nothing listening" (Chapter 6)
- Telling a firewall reject apart from a genuinely closed port needs packet-capture tooling (`tcpdump`/Wireshark) — out of scope here, but worth knowing the limit
- Symptom patterns pointing at a firewall: works from one network but not another, one port blocked while others work, or a sudden break with no application deploy behind it
- **Test the same port from multiple vantage points** — the single most useful technique when you can't see the rules yourself
- Not finding a rule at one layer (e.g. the host firewall) doesn't rule out a block at another layer (network appliance, cloud security group) you may have no visibility into
- When you can't see the rules, hand off exact evidence — source IP, destination IP, port, protocol, timestamp — rather than a vague report
- A source-scoped block found via testing may be **intentional access control**, not a bug — confirm before "fixing" it
- **Next chapter:** Proxies, VPNs & NAT: When the Path Isn't What You Expect

CHAPTER 8 OF 10

Proxies, VPNs & NAT: When the Path Isn't What You Expect

Network Troubleshooting

Chapter 8 · Proxies, VPNs & NAT: When the Path Isn't What You Expect

Chapter 7 found that traffic behaved differently depending on whether it came from the office, over VPN, or directly from home — and left the actual mechanism unexplained. This chapter covers the three things most likely to be quietly rewriting the path itself: NAT changing which address the far end actually sees, a VPN routing only some traffic (or all of it) somewhere unexpected, and a proxy sitting in the middle without the client necessarily even knowing it's there.

NAT: Why "My IP" and "The IP the Server Sees" Aren't the Same

Chapter 3's `ipconfig` output showed a private address like `192.168.1.42` — an address that's only meaningful on the local network. The router performing Network Address Translation (NAT) rewrites that private address to its own public one before traffic leaves the local network, and rewrites the reply back on the way in. This matters diagnostically more than it might seem: a server's own logs, or a firewall rule written to allow a specific "user's IP," will reference the public, NAT'd address — not the private one the user's own machine reports. Comparing the two, when they're expected to match and don't, is a genuinely common source of confusion.

```
$ curl ifconfig.me
198.51.100.17

# vs. the machine's own local address, from Chapter 3
$ ip addr show eth0 | grep inet
    inet 192.168.1.42/24
```

A quick "what's my public IP" check like this is worth running before comparing a user's own reported address against anything in server logs or a firewall rule — otherwise you're comparing two genuinely different addresses and drawing conclusions from a mismatch that was never meaningful in the first place.

A SECOND LAYER: DOUBLE NAT AND CARRIER-GRADE NAT

A home router performs NAT once; some ISPs additionally run their own NAT layer for an entire neighborhood of customers (carrier-grade NAT, common on mobile networks and some residential ISPs). The practical effect worth knowing: the public IP a user's machine reports as "theirs" can be shared with many other unrelated customers at the same time, which makes

IP-based access rules and abuse-tracking genuinely less reliable for those users — not a bug on the user's end, just a real limitation of how their connection is provisioned.

VPN: Full Tunnel vs. Split Tunnel

Chapter 7's ticket resolves here: a **full-tunnel** VPN routes every packet through the VPN, internal and general internet traffic alike — while a **split-tunnel** VPN routes only traffic destined for specific internal ranges through the tunnel, leaving everything else to go directly out the local internet connection as normal. Whether a given connection actually goes through the tunnel or not depends entirely on which mode is configured — "connected to VPN" alone doesn't tell you which one you're dealing with.

"CONNECTED" DOESN'T MEAN "ROUTED THE WAY YOU EXPECT"

It's easy to assume a connected VPN means all traffic is tunneled, or conversely to assume only the intended internal traffic is — and act on either assumption without checking. Two users both showing "VPN: Connected" can have genuinely different effective routing if one has a full-tunnel profile and the other a split-tunnel one, or if a split-tunnel configuration's own included ranges don't cover the specific destination in question.

Proxies: Explicit and Transparent

An **explicit** proxy is one the client is deliberately configured to use — a browser or application setting pointing traffic at a specific proxy server. A **transparent** (or intercepting) proxy redirects traffic without any client-side configuration at all, commonly used on corporate networks for content filtering or security scanning — which means a support engineer troubleshooting a client that has no proxy settings configured can still be dealing with one, invisibly, at the network level.

One specific transparent-proxy behavior is worth knowing before Chapter 9 goes deep on TLS: some corporate security proxies perform **TLS inspection** — decrypting HTTPS traffic, inspecting it, then re-encrypting it with the proxy's own internal certificate before passing it on. A browser that already trusts that internal certificate authority (often pushed out by IT via group policy) sees nothing unusual. A script, a command-line tool, or any client that doesn't already trust that same internal CA will see the connection as presenting an untrusted certificate — because, from that client's point of view, it is one.

COMPARE THE EFFECTIVE PUBLIC IP ACROSS SCENARIOS

Running `curl ifconfig.me` (or an equivalent) with a VPN connected, then again disconnected, is a fast way to confirm whether a given connection actually goes through the VPN at all — a different result each time confirms the traffic is genuinely tunneled; an identical result on a supposedly-connected VPN is a strong sign it's split-tunnel and isn't routing this particular traffic through the tunnel.

Working Example: A Script That Fails Where a Browser Doesn't

A ticket: an automation script calling an external API fails with a certificate validation error, but a colleague can reach the exact same API fine from their browser, from the same corporate network. Both facts turn out to be true, and neither is really about the API itself:

```
$ curl -v https://api.partner-service.com
* Connected to api.partner-service.com (203.0.113.90) port 443
* TLS handshake, Server Hello
* SSL certificate problem: unable to get local issuer certificate
```

The browser trusts the corporate network's own internal certificate authority, pushed out to every managed machine — so it sees the TLS-inspecting proxy's re-issued certificate as perfectly valid and never shows a problem. The script's HTTP client, running with its own default trust store, has never heard of that internal CA, and correctly refuses to trust a certificate signed by an authority it doesn't recognize. Neither the API nor the network connection is actually broken; the fix belongs to the script's own environment — installing the corporate CA certificate into its trust store — not to anything on the network path itself. Chapter 9 covers reading a TLS handshake failure like this one in full.

Hands-On Exercises

EXERCISE 1

A user reads you their local IP address from `ipconfig`, and you compare it directly against an address recorded in a server's access log for the same time window — they don't match. Explain why this mismatch might not mean anything is wrong.

View solution

EXERCISE 2

Two users both show "VPN: Connected" but report different results reaching the same internal resource. Explain what this chapter says could cause that, even though both are genuinely connected.

View solution

EXERCISE 3

In this chapter's worked example, explain why the browser succeeds and the script fails against the exact same API from the same network, and why the fix belongs to the script's environment rather than the network path.

📄 View solution

CHAPTER 8 QUICK REFERENCE

- **NAT** means a user's private local IP and the public IP a server actually sees are two different addresses — don't compare them directly without accounting for this
- **Double/carrier-grade NAT** can mean many unrelated users share one visible public IP — a real limit on IP-based tracking, not a bug
- **Full-tunnel VPN** routes everything through the tunnel; **split-tunnel VPN** routes only specific traffic — "connected" alone doesn't say which
- **Transparent proxies** can intercept traffic with zero client-side configuration — absence of proxy settings doesn't mean absence of a proxy
- **TLS-inspecting proxies** re-issue their own certificate — a browser trusting the internal CA sees nothing wrong; a script without that CA correctly rejects it
- Compare the **effective public IP** with a VPN connected vs. disconnected to confirm whether traffic is actually tunneled
- **Next chapter:** Application-Layer Symptoms: HTTP & TLS-Specific Failures

CHAPTER 9 OF 10

Application-Layer Symptoms: HTTP & TLS-Specific Failures

Network Troubleshooting

Chapter 9 · Application-Layer Symptoms: HTTP & TLS-Specific Failures

This is the top rung of Chapter 2's ladder. Chapter 8 gave a first taste of a TLS-specific failure; this chapter reads TLS handshake problems and HTTP status codes properly, as real diagnostic signal rather than just "it errored." It's also worth naming up front what's different about this rung compared to every one before it: for the very first time, getting a response — *any* response, including an error — is itself good news about everything below it.

AN HTTP ERROR IS PROOF THE NETWORK WORKED

Receiving an HTTP status code back at all — even a 500 — is only possible once DNS resolved (Chapter 4), the network path worked (Chapter 5), the port accepted a connection (Chapter 6), and, for HTTPS, the TLS handshake completed successfully (this chapter). None of that could have happened if any lower rung had actually failed. A user reporting "I got an error page" is, whether they realize it or not, handing you confirmation that the first four rungs of the ladder are already fine — the remaining question is narrower than it sounds.

Reading a TLS Handshake Failure

Chapter 8 covered one TLS failure — an untrusted certificate issuer, caused by a transparent proxy's own re-issued certificate. That's one of several distinct TLS failure signatures worth telling apart, since each points at a genuinely different fix:

FAILURE	WHAT IT MEANS
Unable to get local issuer certificate	The client doesn't trust the certificate's issuing authority — Chapter 8's transparent-proxy scenario is one cause; a genuinely misconfigured server chain is another
Certificate has expired	The certificate's own validity window has passed — an operational/renewal problem, not a trust or network issue
Hostname mismatch	The certificate presented doesn't cover the name being requested — common behind a load balancer or reverse proxy serving the wrong certificate for a given hostname (SNI)
Handshake failure / no shared cipher	The client and server couldn't agree on a TLS version or cipher — often an old client library that doesn't support a server's minimum required TLS version

This site's own **HTTPS/TLS Fundamentals** (`https1`) course covers the handshake mechanics and certificate chain of trust behind each of these in full — this chapter focuses specifically on recognizing which one you're looking at from a failure message.

HTTP Status Codes as Diagnostic Signal

RANGE	WHAT IT TELLS YOU
2xx	Success — the request was received and handled as expected
3xx	A redirect — worth following explicitly (`curl -L`) rather than assuming; a redirect loop or an unexpected destination is itself a real, diagnosable symptom
4xx	Reported as a client-side problem, but not always literally the client's fault — a 401/403 can come from the application's own authorization logic, or from a WAF or reverse proxy blocking the request before it ever reaches the application
5xx	A server-side failure — the request was received and something failed while handling it; which specific 5xx code narrows down where in the server-side chain it happened

A 403 DOESN'T TELL YOU, ON ITS OWN, WHO BLOCKED YOU

An authorization failure and a security block can look identical from the client side. The application itself might be correctly enforcing a permissions rule, or a Web Application Firewall (WAF) — itself a specialized, application-aware relative of Chapter 7's own firewalls — might be rejecting the request based on its IP, pattern, or rate before it ever reaches the actual application. Telling the two apart from outside usually needs the same thing Chapter 7 needed for a plain network-layer firewall: server-side or WAF logs, not just the client's own response.

The 5xx Family, Specifically

These four get conflated constantly, but each says something genuinely different about where the failure happened:

CODE	WHAT IT SPECIFICALLY MEANS
500 Internal Server Error	A generic failure inside the application itself — no more specific information is being given
502 Bad Gateway	A reverse proxy or gateway received an invalid or malformed response from the backend it forwarded the request to — the proxy itself is fine; its upstream isn't
503 Service Unavailable	The server is deliberately not handling requests right now — overloaded, or in maintenance mode; often paired with a `Retry-After` header
504 Gateway Timeout	A reverse proxy or gateway's upstream never responded within the allowed time — distinct from 502, where the upstream did respond, just invalidly

Reading curl -v's Full Timeline

Put together, one verbose `curl` request shows every phase this course has built toward, in order:

```
$ curl -v https://app.example.com/api/status
*   Trying 203.0.113.42:443...
* Connected to app.example.com (203.0.113.42) port 443   # Chapter 6: port confirmed open
* TLS handshake, Client Hello
* TLS handshake, Server Hello, Finished             # This chapter: TLS confirmed
> GET /api/status HTTP/1.1
> Host: app.example.com
<
< HTTP/1.1 502 Bad Gateway                            # This chapter: application-layer result
```

Every line above the status code is confirmation, not failure — the connection, the TLS handshake, and the request itself all completed. The 502 specifically says the reverse proxy fronting this app received a response from its own backend, and that response was invalid — narrowing the problem to the backend service itself, or the connection between the proxy and that backend, rather than anything this course's earlier chapters cover.

FROM HERE, THIS BECOMES LOG1'S TERRITORY

Once a genuine application-layer failure is confirmed — as opposed to a network, DNS, port, or TLS problem — actually diagnosing *why* the backend returned an invalid response belongs to this site's own **Logging & Log Analysis** (`log1`) course, whose Chapters 6 and 7 build exactly this kind of walkthrough in depth. This course's ladder exists to get you confidently to that handoff point — confirming which rung actually failed — not to replace the log-reading skill needed once it has.

Hands-On Exercises

EXERCISE 1

A user reports "I got a 500 error page." Explain why this chapter treats that report as good evidence, not just bad news, and specifically what it already confirms.

View solution

EXERCISE 2

Explain the specific difference between a 502 Bad Gateway and a 504 Gateway Timeout, and what each one tells you about where in the request chain the failure happened.

View solution

EXERCISE 3

A request returns a 403 Forbidden. Explain why this chapter says that alone doesn't tell you whether the application or a WAF produced it, and what Chapter 7 concept this connects to.

View solution

CHAPTER 9 QUICK REFERENCE

- **Any HTTP status code, even an error, proves DNS, network reachability, port, and TLS (if applicable) all already worked**
- TLS failures: **untrusted issuer** (Chapter 8), **expired certificate**, **hostname mismatch**, **no shared cipher/version** — each points at a different fix; see `https1` for the full mechanics
- 2xx = success; 3xx = redirect, worth following explicitly; 4xx = reported client-side, but a 401/403 could be the app or a WAF; 5xx = server-side
- **500** = generic app failure; **502** = proxy got an invalid response from its backend; **503** = deliberately unavailable; **504** = the backend never responded in time
- A 403 doesn't reveal, on its own, whether the application or a WAF produced it — same ambiguity as Chapter 7's plain firewall block
- Once the ladder confirms a genuine application-layer failure, actually diagnosing it becomes **Logging & Log Analysis** (`log1`) territory
- **Next chapter:** Capstone: Triaging Three Real Connectivity Tickets

CHAPTER 10 OF 10

Capstone: Triaging Three Real Connectivity Tickets

Network Troubleshooting

Chapter 10 · Capstone — Triaging Three Real Connectivity Tickets

Nine chapters built the pieces — scoping questions, the practical ladder, local connectivity, DNS, path testing, port checks, firewalls, VPN/NAT/proxy path changes, and reading the application layer itself. This capstone applies all of it to three fresh tickets, worked more briskly than earlier chapters' own dedicated walkthroughs, since the underlying discipline should already feel familiar by now.

Ticket 1: "The new hire can't reach anything"

A new employee's first-day laptop can't load any internal site or reach the internet at all — freshly imaged, connected today for the first time.

APPLYING CHAPTER 1'S SCOPING QUESTIONS

One user, not many — everywhere, not one destination — total failure, not degraded — and it has never worked, since this is the machine's first connection. That combination points squarely at something local to this one machine, before DNS or any specific server is even worth testing.

APPLYING CHAPTER 3'S LOCAL CONNECTIVITY CHECK

```
Ethernet adapter Ethernet:
   IPv4 Address. . . . . . . . . . . : 169.254.12.203
   Subnet Mask . . . . . . . . . . . : 255.255.0.0
   Default Gateway . . . . . . . . . : (none)
```

A textbook link-local address with no gateway — DHCP never succeeded on this machine at all. A loose network cable, reseated, immediately resolves it; a fresh `ipconfig /all` shows a real address and gateway, and a ping to the gateway confirms it.

Total time to resolution: well under the time it would have taken to start troubleshooting DNS or a specific application — exactly why local connectivity is the ladder's own starting rung.

Ticket 2: "The reporting app doesn't work on the new Wi-Fi"

Since this week's new office Wi-Fi rollout, multiple users report the internal reporting app is unreachable over Wi-Fi, while it still works fine from a wired connection. Every other internal app works fine on both.

APPLYING CHAPTER 1'S SCOPING QUESTIONS

Many users, one destination, just started — since the new Wi-Fi went live this week. That specific combination (one app, one network, everyone affected) points toward something scoped by network and by destination — a strong early hint toward a firewall rule rather than DNS or the app itself.

APPLYING CHAPTERS 4 AND 5: RULING OUT THE LOWER RUNGS

From the new Wi-Fi network, DNS resolves the reporting app's name correctly, and a ping to its host succeeds cleanly. Both rungs check out — the problem sits higher up the ladder.

APPLYING CHAPTER 6 AND 7'S VANTAGE-POINT COMPARISON

```
# From the wired network
$ nc -zv 10.0.9.20 8443
Connection to 10.0.9.20 8443 port [tcp/*] succeeded!

# From the new Wi-Fi network
$ nc -zv 10.0.9.20 8443
(hangs — Ctrl+C to cancel)
```

Same destination, same port, two different networks, two different results — exactly the pattern Chapter 7 flagged as pointing at a boundary rule. Handing off the wired subnet, the new Wi-Fi subnet, the destination, and the port to the network team confirms it directly: the reporting app's security-group rule was scoped to the old wired subnet during the original rollout, and was simply never updated to include the new Wi-Fi range when it launched — an oversight, not an intentional restriction this time.

Ticket 3: "The partner API script times out sometimes, but I can never reproduce it in a browser"

A scheduled automation script calling a partner's API intermittently fails with a 504 error. A colleague testing the same API by hand, in a browser, from the same corporate network, has never once seen it fail.

APPLYING CHAPTER 9: READING THE FAILURE, NOT JUST REACTING TO IT

The first instinct is to assume this is Chapter 8's own certificate-trust issue again — a script failing where a browser succeeds. Checking the actual failure rules that out immediately: `curl -v` shows the TLS handshake completing cleanly every time, and the failure itself is a genuine **504 Gateway Timeout** — a real HTTP response, which per Chapter 9 already proves DNS, network reachability, the port, and TLS all worked on that specific attempt. This is a different problem wearing a familiar disguise.

APPLYING CHAPTER 5'S TRACEROUTE TECHNIQUE FROM THE ACTUAL FAILING CONTEXT

Run from the batch server that actually executes the script — not from the colleague's own workstation — a traceroute to the partner API shows something the interactive browser test never would have surfaced:

```
$ traceroute partner-api.example.com
 1  10.0.9.1        0.3 ms
 2  10.0.2.1        1.1 ms
 3  * * *
 4  * * *
 5  198.51.100.9    210 ms   # high, inconsistent latency
 6  203.0.113.90    340 ms
```

The batch server's own traffic takes a genuinely different path than an ordinary workstation's — routed through a secondary, backup VPN tunnel reserved for server infrastructure, with real, measurable latency and packet loss under load. That path occasionally exceeds the partner API's own gateway timeout window, exactly as Chapter 9's 504 definition describes — not because the script or the API is broken, but because of where the traffic is actually routed from, per Chapter 8's own "the path isn't always what you'd expect" theme.

> **THE SAME "SCRIPT VS. BROWSER" QUESTION, AN HONESTLY DIFFERENT ANSWER**
>
> Chapter 8's own script-vs-browser ticket was a certificate-trust issue. This one looks identical on the surface but isn't — checking rather than assuming led straight past the familiar-looking explanation to the real one: a routing difference specific to where the script actually runs from.

The fix belongs to infrastructure — routing the batch server's traffic over the same primary path interactive users take, or raising the timeout on that specific backup path — handed off with the exact traceroute evidence above, per Chapter 7's own handoff principle.

Chapter Attribution

TECHNIQUE USED ABOVE	SOURCE CHAPTER
Scoping questions (one user/many, one destination/everywhere, total/degraded, new/persistent)	Chapter 1
The overall bottom-up ladder, stopping at the first failing rung	Chapter 2
Reading a link-local (169.254.x.x) address and missing gateway (Ticket 1)	Chapter 3
Confirming DNS resolution before testing further up (Ticket 2)	Chapter 4
Ping to confirm network reachability; traceroute revealing a real routing difference (Ticket 3)	Chapter 5
`nc -zv` port checks, compared across vantage points (Ticket 2)	Chapter 6
The multi-vantage-point technique and exact-evidence handoff (Tickets 2 and 3)	Chapter 7
Ruling out a certificate-trust explanation and recognizing a genuine path/routing difference (Ticket 3)	Chapter 8
Reading a 504 correctly, and using an HTTP response as proof of the lower rungs (Ticket 3)	Chapter 9

Honest Scope Note

WHAT THIS COURSE DELIBERATELY DOESN'T COVER

- No packet-capture tooling in depth (`tcpdump`, Wireshark) — Chapter 7 named the limit where these would be needed; a genuinely separate, deeper skill
- No routing-protocol theory (BGP, OSPF, and similar) — this course diagnoses an existing network from the outside, not the internals of how routes are built and propagated
- No RF-level wireless troubleshooting (signal strength, channel interference, antenna placement) — Ticket 2 touched Wi-Fi only at the IP/firewall layer, not the radio layer
- No IPv6-specific troubleshooting — every example in this course used IPv4; IPv6's own addressing and neighbor-discovery quirks are a real, separate topic
- No cloud-native networking specifics (service meshes, ingress controllers, container networking) — the ladder's principles still apply, but the concrete tooling genuinely differs

Each is a legitimate, separate topic — not silently assumed solved by what this course actually covers.

Hands-On Exercises

EXERCISE 1

Explain why Ticket 1's scoping answers (one user, everywhere, total failure, never worked) pointed toward local connectivity before DNS or any specific application was even tested.

View solution

EXERCISE 2

Explain what specifically confirmed Ticket 2 was a firewall/security-group issue rather than a problem with the reporting app itself, and why testing from both networks was the key step.

View solution

EXERCISE 3

Explain why Ticket 3's "script fails, browser doesn't" symptom looked like Chapter 8's certificate-trust ticket at first, and what specific evidence ruled that explanation out before the real cause was found.

View solution

CHAPTER 10 QUICK REFERENCE — COURSE COMPLETE

- Ticket 1: a link-local address with no gateway — a local connectivity problem, found and fixed in under a minute
- Ticket 2: a firewall/security-group rule scoped to the wrong subnet — found by comparing results across vantage points
- Ticket 3: a 504 caused by a genuine routing difference for the script's own traffic — not the certificate-trust issue it first resembled
- The recurring theme across all ten chapters: **scope first, work the ladder, and check rather than assume** — whichever specific technique that means in the moment
- This closes *Network Troubleshooting*, 10/10 chapters — the second course under the Technical Support subject, alongside *Logging & Log Analysis*