VPN Course

Flavio Copes

Updated August 3, 2026

Contents

Learn how VPN tunnels, encryption, routing, DNS, and access control work, then build a WireGuard VPN and a Cloudflare private network.

What you'll learn: Trace traffic through a VPN, build and verify a small WireGuard setup, and create a VPN replacement with Cloudflare Tunnel and the Cloudflare One Client.

VPN foundations

Understand the problems VPNs solve, the paths they create, and the trust they move.

What a VPN is

See a VPN as a private path across another network and identify the two endpoints that protect and forward its traffic.

A **Virtual Private Network**, or VPN, creates a logical private path across another network.

"Virtual" is the key word. There is no dedicated cable between you and the network you want to reach. There is only the ordinary Internet. The VPN builds something that behaves like a private link on top of it.

Here is the mechanism. Your device sends selected packets into a tunnel. The VPN software protects them, wraps them inside other packets, and sends them to the other endpoint. The endpoint removes the outer wrapper and forwards the original packets. Replies travel back through the tunnel.

Two endpoints do all the work. One runs on your device. The other runs at the edge of the network you want to reach.

```
laptop  (protected outer packets, across the Internet)  VPN endpoint   private network
```

On your device the tunnel appears as a network interface, right next to the physical ones:

```
ip -brief address
# lo     UNKNOWN  127.0.0.1/8
# eth0   UP       192.168.1.34/24
# wg0    UNKNOWN  10.14.0.2/24
```

`wg0` is a virtual interface. Packets the operating system routes into it get protected and wrapped. Packets arriving through it get unwrapped and delivered like any other traffic.

That interface is the whole trick. The rest of the system does not need to know a VPN exists. It sees one more interface, with routes pointing at it, and uses it like any other.

Why does this arrangement exist? Because networks route on addresses, and a private address like `10.14.0.7` means nothing on the public Internet. No router out there will carry a packet toward it. The tunnel gives those packets a routable outer envelope, so they can cross networks that would never carry them directly.

One warning before you go further. A VPN is a path, not a promise that every system behind it is safe. Connecting to a network does not make that network trustworthy, and a VPN endpoint forwards malicious packets exactly as well as honest ones.

Follow a packet through a VPN

Trace one request from an application through the virtual interface, encrypted tunnel, VPN endpoint, and final destination.

Suppose your browser sends a request while a full-tunnel VPN is active. We can follow one packet the whole way.

First, the operating system consults its routing table. With a full tunnel active, the best route for almost every destination points at the virtual network interface. You can ask the kernel directly which route a destination would take:

```
ip route get 142.250.180.100
# 142.250.180.100 dev wg0 table 51820 src 10.14.0.2
```

The answer names `wg0`. The packet is handed to the virtual interface, still unencrypted, carrying your VPN address `10.14.0.2` as its source.

The VPN client now encrypts and encapsulates it. The original packet becomes the payload of a new packet addressed to the VPN endpoint, and that outer packet leaves through your real interface.

This is the part your normal network can see: a stream of encrypted packets going to the VPN endpoint. It cannot read the protected inner packet, and it cannot see the real destination.

The VPN endpoint decrypts the outer packet and forwards the inner one toward the destination. When the endpoint also provides Internet egress, it rewrites the source address, so the destination sees the endpoint address instead of your current public address.

You can verify that last step yourself:

```
curl https://ifconfig.me
# 203.0.113.10
```

If that prints the VPN endpoint's address, egress traffic is really using the tunnel. If it prints your home connection's address, your routes are not doing what you think.

The reply retraces the path. The destination answers the endpoint, the endpoint wraps the reply, your client unwraps it, and the browser receives bytes as if nothing unusual happened.

One detail surprises people: traffic to the VPN endpoint itself never goes through the tunnel. It cannot, because the outer packets need a direct path to reach it. The operating system keeps one ordinary route to the endpoint via your local gateway. That route is correct behavior, not a leak.

Understand the new trust boundary

Know which observers a VPN can hide traffic from and why the VPN operator becomes part of your trust model.

A VPN does not remove trust from your network path. It moves it somewhere else.

Start with what it genuinely does. A VPN can protect traffic from the local Wi-Fi operator and other observers between you and the VPN endpoint. The coffee shop router sees encrypted packets flowing to one address, and nothing else. Against that specific observer, the protection is real.

Now look at the other end. Every packet you tunnel gets decrypted at the VPN endpoint and forwarded from there. The VPN operator handles the forwarded traffic and can see metadata: which destinations you contact, when, and how much data moves. If a connection uses plain HTTP, the operator can read the content too.

```
observer between you and the endpoint:
  sees:  your IP   endpoint IP, packet sizes, timing
  can't: read content or learn destinations

the VPN operator, at the endpoint:
  sees:  your IP, every destination you contact, timing, volume
  can't: read content protected by HTTPS
```

This is why "no logs" marketing claims deserve skepticism. You cannot verify them from the outside. You are choosing to believe the operator.

A VPN also does not make you anonymous. Destinations can still identify you through accounts, cookies, browser fingerprints, and application data. Log in to your email through a VPN and your email provider knows exactly who you are, tunnel or not.

HTTPS still matters. It protects the application connection beyond the VPN endpoint, all the way to the destination. The two protections layer: the VPN covers the first leg against local observers, HTTPS covers the content end to end.

My advice is to ask one question first: who are you moving trust away from, and who receives it instead? "Away from an airport Wi-Fi operator, toward a server I run" is a clear trade. "Away from my ISP, toward a company I know nothing about" is also a trade — just make sure you noticed you made it.

Recognize common VPN jobs

Separate remote access, site-to-site connectivity, Internet egress, and consumer privacy because each job needs a different design.

Several different jobs hide behind the word VPN. Naming the job you actually have is the most useful thing you can do before building anything.

A **remote-access VPN** connects one device to a private network. The classic example: your laptop at home reaching the internal wiki at `10.0.1.100`. Each device runs a client and holds its own credentials.

A **site-to-site VPN** joins two networks. The office network and the datacenter network behave like one routed system. The tunnel runs between two gateways, and the individual devices behind them never know it exists.

An **egress VPN** sends Internet traffic through another location. You are not trying to reach anything private; you are changing where your traffic exits and who can observe it locally. A consumer VPN usually focuses on this job.

```
job             connects               the question it answers
remote access   one device   network   "can my laptop reach the internal wiki?"
site-to-site    network   network      "can the office reach the datacenter?"
egress          device   Internet      "which address do websites see, and who watches local
app access      one user   one app     "can Ana open the admin panel, and nothing else?"
```

That last row matters more every year. A company may need identity-based access to a few internal applications instead of broad network access. If the requirement is "Ana can use the dashboard", handing Ana a route to an entire subnet is far more access than the job needs.

The designs differ, which is why the label alone misleads. Remote access needs per-device keys and enrollment. Site-to-site needs gateways and agreed routes on both sides. Egress needs NAT and careful DNS handling. App access might not need a network tunnel at all.

Do not choose a product from the word VPN alone. Start from the traffic that must move and the resources that must become reachable. Write it as one sentence: "this device must reach that resource." The design follows from the sentence.

The common mistake: buying a consumer egress VPN to reach an office network. It changes where your Netflix traffic exits. It brings you no closer to `10.0.1.100`.

Check your understanding: VPN foundations

Check tunnels, endpoints, packet paths, trust boundaries, HTTPS, remote access, site-to-site links, and egress.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Tunnels and protocols

Separate tunneling from encryption and compare the main protocol designs.

Separate tunneling from encryption

Understand encapsulation and encryption as different operations that a secure VPN normally combines.

Tunneling wraps one packet inside another packet so it can cross a network that would not route the original directly.

Encryption turns readable data into protected data for parties holding the right keys.

These are different operations solving different problems. Tunneling solves reachability: a packet addressed to `10.14.0.1` cannot cross the public Internet on its own, but a packet carrying it as payload can. Encryption solves confidentiality: whoever sees the bytes in transit learns nothing from them.

A wrapped, protected packet looks like this on the wire:

```
outer IP header        public addresses, routable everywhere
  UDP header           port 51820
    encrypted payload  opaque to observers
      inner IP header  10.14.0.2 → 10.14.0.1
      inner data       your actual traffic
```

A tunnel can exist without encryption. GRE, an old and still common tunneling protocol, wraps packets and moves them in the clear. Anyone on the path can read the inner packet. That can be acceptable inside infrastructure you already trust, and reckless across the Internet.

Encryption can protect an application without creating a network tunnel. HTTPS encrypts your connection to one website. No new interface appears, no routes change — exactly one TCP connection is protected, and nothing else.

A secure VPN normally combines both: it creates a routable outer packet and protects the inner packet. You can watch the combination on the real interface while a WireGuard tunnel carries traffic:

```
sudo tcpdump -n -i eth0 udp port 51820 -c 2
# IP 192.168.1.34.51423 > 203.0.113.10.51820: UDP, length 148
# IP 203.0.113.10.51820 > 192.168.1.34.51423: UDP, length 92
```

Ordinary UDP packets between two addresses. The payload is noise. That is tunneling and encryption working together.

Keeping the two ideas separate pays off when things break. Tunnel problems look like routing problems: unreachable destinations, packets taking the wrong path. Encryption problems look like identity problems: handshakes that never complete, peers that stay silent. Knowing which half failed cuts your search in half.

Compare IPsec, OpenVPN, and WireGuard

Recognize the different shapes of three common VPN protocol families without treating them as interchangeable configuration files.

Three protocol families dominate real deployments. They are not three flavors of the same thing. They have different shapes.

IPsec is a family of standards for protecting IP traffic. IKE negotiates peers, algorithms, and keys. It lives in operating system kernels and in network hardware, which makes it the usual answer when two organizations connect routers from different vendors. The cost is surface area: many components, many possible algorithm combinations, and debugging sessions that involve reading two vendors' logs side by side.

OpenVPN is a userspace VPN that uses TLS and can run over UDP or TCP. The TCP option matters in practice: on networks that block unfamiliar UDP, OpenVPN on TCP port 443 looks like ordinary HTTPS and often gets through. Configuration revolves around certificates and a long list of directives. It is mature and portable, and its userspace design costs some throughput.

WireGuard presents a small encrypted network interface built around public-key peers and UDP. There is no algorithm negotiation — the protocol fixes one modern set of cryptographic primitives. It runs inside the Linux kernel, roaming between networks is built in, and a working configuration fits on one screen.

```
IPsec      standards family, IKE negotiation, kernel and hardware support
OpenVPN    TLS-based, UDP or TCP (even 443), certificates, userspace
WireGuard  fixed modern crypto, UDP only, public-key peers, tiny config
```

Those shapes decide real outcomes. Need to interoperate with a partner's firewall appliance? That points to IPsec. Need to work from networks that strangle UDP? OpenVPN over TCP. Building your own infrastructure and want the smallest thing you can fully understand? WireGuard — and it is what we build in this course.

WireGuard's UDP-only design is also its clearest limitation. On a network that blocks UDP entirely, the handshake never completes, and no configuration change on your side fixes that.

The protocol affects deployment, compatibility, roaming, debugging, and policy. The name alone does not decide whether the complete setup is safe. A flawless protocol with a careless peer list is still an open door.

Keys, handshakes, and peers

See how peers authenticate each other, establish fresh session keys, and keep long-term private keys out of configuration sharing.

Each WireGuard peer has a private key and a public key. You share the public key and protect the private key. There are no usernames, no certificates, no accounts. A peer *is* its key pair.

If you have used SSH keys, this model is familiar. The public key can travel anywhere — chat, email, a config file in a repo. The private key never leaves the device it was generated on.

When two peers first exchange traffic, they perform a **handshake**. Each side proves it holds its private key, and together they derive fresh symmetric **session keys** that protect the actual traffic. The peers rotate session keys while the long-term identity remains stable — under active traffic, WireGuard performs a new handshake roughly every two minutes. A stolen session key exposes only a small window of traffic, never the whole history.

You can watch this machinery work:

```
sudo wg show
# interface: wg0
#   public key: hIhpm5DfIhSNQvvBpG0fWo6yQqYamOoO70QI0DGkfBM=
#   listening port: 51820
#
# peer: xTIBA5rboUvnH4htodjb6e697QjLERt1NAB4mZqp8Dg=
#   endpoint: 198.51.100.44:53200
#   latest handshake: 34 seconds ago
#   transfer: 1.24 MiB received, 861.32 KiB sent
```

The `latest handshake` line is the most useful diagnostic in all of WireGuard. A recent handshake proves both sides hold the right keys and can exchange UDP. No handshake at all means identity or connectivity failed — nothing else is worth checking until that line appears.

WireGuard does not distribute keys or decide who should receive them. There is no login server, no revocation list. That operational work still belongs to you: generate keys on each device, copy public keys between configs, and remove them when access should end.

Treat a peer key like an access credential. Remove it when the device is lost or no longer authorized:

```
sudo wg set wg0 peer xTIBA5rboUvnH4htodjb6e697QjLERt1NAB4mZqp8Dg= remove
```

One mistake to avoid: copying a single private key to a second device because it seems convenient. Two devices then claim the same identity, and the endpoint flips between their addresses as each one talks. Connections stall in confusing ways. Every device gets its own pair, always.

Account for encapsulation and MTU

Understand why tunnel headers reduce usable packet size and how fragmentation or dropped discovery messages can break selected connections.

Every tunnel adds outer headers. The wrapped packet is larger than the original packet. That

small piece of arithmetic produces some of the strangest bugs you will ever debug.

A network interface has a **Maximum Transmission Unit**, or MTU: the largest packet it agrees to carry. Ethernet links commonly use 1500 bytes. WireGuard's outer headers consume about 60 of those bytes on IPv4, so the tunnel interface must advertise a smaller MTU. `wg-quick` handles this for you, which is why you typically see 1420:

```
ip link show wg0
# 4: wg0: <POINTOPOINT,NOARP,UP,LOWER_UP> mtu 1420 qdisc noqueue ...
```

If a protected packet becomes too large for some link on the path, the path must fragment it or tell the sender to use a smaller size. Modern senders set the Don't Fragment flag and rely on ICMP "packet too big" messages coming back — that mechanism is **Path MTU Discovery**. Firewalls that silently drop ICMP break it.

Broken Path MTU Discovery has a signature: small requests work while larger transfers stall. The login page loads, the file download hangs at zero. Ping succeeds, HTTPS to the same host freezes mid-response. Nothing errors — packets just vanish.

You can probe packet sizes deliberately. `-M do` forbids fragmentation, so failures reveal exactly where the limit sits:

```
ping -c 1 -M do -s 1392 10.14.0.1
# 1392 data bytes + 28 header bytes = 1420: fits, replies arrive

ping -c 1 -M do -s 1400 10.14.0.1
# ping: local error: message too long, mtu=1420
```

If sizes fail well below the interface MTU, some link on the path is smaller than everyone assumed — a PPPoE connection at 1492 is a classic cause.

Do not guess first. Compare working and failing packet sizes, inspect the interface MTU, and change it only with evidence. When the evidence points to a smaller path, set `MTU = 1380` in the `[Interface]` section and test again.

The mistake to avoid is the reflex fix: lowering MTU to some tiny value at the first sign of trouble. It often masks the symptom, costs throughput on every packet, and leaves the real cause — usually an ICMP-dropping firewall — in place.

Check your understanding: tunnels and protocols

Check encapsulation, encryption, IPsec, OpenVPN, WireGuard, peer keys, handshakes, packet overhead, and MTU.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Routing, DNS, and privacy

Control traffic paths, private ranges, name resolution, egress, and leak protection.

Read private addresses and CIDR routes

Identify private IPv4 ranges and read a CIDR prefix as the set of destination addresses a route can match.

Private IPv4 networks commonly use `10.0.0.0/8`, `172.16.0.0/12`, or `192.168.0.0/16`. RFC 1918 reserves these ranges for private use, and Internet routers do not carry them. That fact is why VPNs exist at all: the only way to reach a private address from outside is through some kind of tunnel.

The notation is **CIDR**. The number after the slash says how many leading bits identify the network. The remaining bits identify hosts inside it:

```
10.14.0.0/24   →  10.14.0.0 - 10.14.0.255       (256 addresses)
10.14.0.0/16   →  10.14.0.0 - 10.14.255.255     (65,536 addresses)
10.0.0.0/8     →  10.0.0.0  - 10.255.255.255    (16,777,216 addresses)
```

A `/24` contains a much smaller range than a `/8`. The bigger the number after the slash, the smaller the network. That inversion trips up everyone once.

Routes are CIDR prefixes with a direction attached. When several routes match a destination, the kernel picks the most specific one — the longest prefix wins. You can ask which route a given address would take:

```
ip route get 10.14.0.7
# 10.14.0.7 dev wg0 src 10.14.0.2

ip route get 10.15.0.7
# 10.15.0.7 via 192.168.1.1 dev eth0 src 192.168.1.34
```

The first address matched the tunnel's `/24` route. The second matched nothing specific and fell through to the default route.

VPN routes often point one private prefix into the tunnel. Use the narrowest prefix that covers the intended resources. Routing all of `10.0.0.0/8` into a tunnel to reach one server at `10.0.1.100` drags sixteen million addresses along for no reason, and every one of them is a potential collision with some other network.

Collisions are the classic failure. Overlapping home and company ranges cause ambiguity: if both use `192.168.1.0/24`, the operating system cannot know which `192.168.1.20` you meant. Longest-prefix matching resolves it mechanically, and you get the printer when you wanted the server. The only clean fixes are renumbering one side or picking unusual ranges from the start — which is exactly why our lab uses `10.14.0.0/24` instead of something popular.

Choose full or split tunneling

Decide whether every destination or only selected private ranges should use the VPN.

The same tunnel, the same keys, the same endpoint — what makes a VPN ”full” or ”split” is nothing but the routes you install.

A **full tunnel** installs routes that send normal Internet traffic through the VPN endpoint. A **split tunnel** sends selected destinations through the VPN and leaves other traffic on the local connection.

In WireGuard the choice is one line in the client's `[Peer]` section:

```
AllowedIPs = 10.14.0.0/24
# split tunnel: only the private subnet uses wg0

AllowedIPs = 0.0.0.0/0, ::/0
# full tunnel: every IPv4 and IPv6 destination uses wg0
```

When `AllowedIPs` contains `0.0.0.0/0`, `wg-quick` does not overwrite your default route. It installs special policy-routing rules that take precedence, so bringing the tunnel down restores the previous state cleanly.

Now the trade-offs. Full tunneling centralizes egress and filtering: every packet gets the endpoint's protections, its logging, and its network position. It also adds latency, bandwidth cost, and a larger failure domain — when the endpoint dies, the entire Internet dies with it for that device.

Split tunneling reduces unnecessary traffic. Your video call to a public service takes the short path instead of a detour through the company gateway. It needs careful routes and DNS behavior so private destinations still use the correct path: a private hostname resolved through public DNS returns nothing, no matter how correct your routes are.

Verify which mode you actually got by testing a public destination:

```
ip route get 1.1.1.1
# split:  1.1.1.1 via 192.168.1.1 dev eth0 src 192.168.1.34
# full:   1.1.1.1 dev wg0 table 51820 src 10.14.0.2
```

Choose from the job. Reaching private resources calls for a split tunnel with narrow routes. Protecting all traffic on hostile networks, or enforcing central filtering, calls for a full tunnel — and you accept its costs deliberately.

The failure mode worth knowing: switching a client to `0.0.0.0/0` when the server was never configured to forward and NAT Internet traffic. Every public destination goes dark at once. The next lesson covers the server side that makes full tunnels actually work.

Understand forwarding, NAT, and egress

Follow a packet across a VPN gateway and know why forwarding and address translation may be required for Internet access.

A VPN endpoint that routes traffic between interfaces acts as a gateway. Packets arrive on `wg0` and need to leave through `eth0`. By default, Linux refuses to do that — it behaves as a host, not a router. The operating system must permit IP forwarding:

```
sudo sysctl -w net.ipv4.ip_forward=1
echo 'net.ipv4.ip_forward=1' | sudo tee /etc/sysctl.d/99-wireguard.conf
```

The first command applies the setting now. The second makes it survive reboots.

Forwarding alone is not enough for Internet egress. Think about the packet the client sends: its source address is `10.14.0.2`, a private address. The web server's reply, addressed to `10.14.0.2`, would be dropped by the first Internet router that saw it. There is no return path.

So for Internet egress, the gateway often replaces the client source address with its own public address. This is source NAT, commonly implemented with masquerading:

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

Every packet leaving through `eth0` now carries the server's address. Replies return to the gateway,

which maps them back to the VPN client using its connection-tracking table. The rewriting is invisible to both ends.

Verify the whole chain from the client:

```
curl https://ifconfig.me
# 203.0.113.10   ← the gateway's public address, not yours
```

Private-network access is a different story. If the office network knows a route back to `10.14.0.0/24`, no address rewriting is needed. Real routes carry traffic in both directions. Prefer routing when both sides can learn the return path: destination logs keep real client addresses, and you carry no NAT state that can fill up or expire mid-connection.

The classic failure: the handshake works, pinging the gateway's `10.14.0.1` works, and nothing beyond it responds. That pattern almost always means forwarding is off or the masquerade rule vanished — iptables rules do not survive reboots unless you persist them. Check `sysctl net.ipv4.ip_forward` and `iptables -t nat -L POSTROUTING -n` before touching anything else.

Keep DNS on the intended path

Route name resolution deliberately and understand why DNS, IPv6, and disconnect behavior can reveal or break traffic.

Routing packets is half the job. Name resolution decides which packets exist in the first place, and it deserves the same attention.

Two situations force you to care where DNS goes. A private service may need a private DNS resolver: `wiki.internal` exists only in the company's DNS, and no public resolver has ever heard of it. And an Internet egress VPN may want DNS queries to use the same trusted path as the traffic — otherwise the local network still sees every hostname you look up, which is most of what you were trying to hide.

A **DNS leak** happens when queries take a path you did not intend. The tunnel dutifully carries your traffic while your queries still flow to the home router's resolver. The observer you were avoiding reads your browsing intentions from the query stream.

`wg-quick` can point the system at a resolver while the tunnel is up:

```
[Interface]
Address = 10.14.0.2/24
PrivateKey = LAPTOP_PRIVATE_KEY
DNS = 10.14.0.1
```

Then verify where queries actually go, because intent and reality diverge here constantly:

```
resolvectl status wg0
#   DNS Servers: 10.14.0.1

dig example.com | grep SERVER
# ;; SERVER: 10.14.0.1#53(10.14.0.1)
```

If the `SERVER` line names your home router instead, you have a leak, whatever the VPN app's interface claims.

IPv6 can also bypass an IPv4-only tunnel. When a destination publishes an IPv6 address and your device has working IPv6, the operating system may prefer it — sailing straight past your carefully

built IPv4 routes. Either carry IPv6 in the tunnel too (`::/0` in `AllowedIPs`) or decide explicitly how IPv6 should behave.

The third trap is disconnection. A **kill switch** blocks selected traffic when the tunnel disappears. Without one, the operating system may return to its ordinary default route — silently, mid-session, while you keep working under assumptions that stopped being true.

Test addresses, DNS answers, IPv4, IPv6, and disconnect behavior. A connected icon proves very little.

Check your understanding: routing, DNS, and privacy

Check private ranges, CIDR, route overlap, full and split tunnels, forwarding, NAT, DNS, IPv6, and kill switches.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Build a WireGuard VPN

Create keys, configure both peers, route traffic, verify the tunnel, and remove it safely.

Plan the WireGuard lab

Define a small two-peer lab, its addresses, reachable resources, public endpoint, and recovery path before changing routes or firewalls.

We will connect a laptop to an Ubuntu server. Before touching a single config file, we decide everything on paper. VPN debugging is miserable when you are guessing what the setup was supposed to look like.

Here is the plan for the whole module:

```
private VPN network    10.14.0.0/24
server VPN address     10.14.0.1
laptop VPN address     10.14.0.2
server listens on      UDP port 51820
public endpoint        203.0.113.10:51820
routed via the VPN     10.14.0.0/24 only - no Internet egress yet
```

Each choice has a reason. `10.14.0.0/24` is an RFC 1918 range that home routers rarely use, so it will not collide with the Wi-Fi networks the laptop visits. Port `51820` is the conventional WireGuard port, though any free UDP port works. And this first setup routes only the VPN subnet — a split tunnel. You can add full Internet egress after the private tunnel works. Debugging a broken default route on a machine you can only reach through that route is a special kind of pain.

Two safety rails before changing anything on the server.

First, keep the current SSH session open while changing the server. An established session keeps working through firewall mistakes that would refuse any new connection. Open it now, in a terminal you will not close:

```
ssh flavio@203.0.113.10
```

```
# leave this session running for the entire lab
```

Second, confirm the provider console works before touching the firewall. That out-of-band console is your recovery path if you lock yourself out of SSH entirely. Log in to it once today, so you are not discovering a broken console password during an emergency.

While you are in the provider dashboard, check whether a cloud firewall or security group sits in front of the server. UDP `51820` will need to pass through it later, and forgotten provider firewalls cause more ”WireGuard doesn't work” reports than WireGuard does.

Write the plan down somewhere you can see it. The mistake this prevents is real: picking a VPN subnet that overlaps a network one of the peers already sits on, then wondering why routes behave strangely. You chose `10.14.0.0/24` deliberately. When something fails later, you will compare reality against this plan instead of against memory.

Install WireGuard and create keys

Install the tools on Ubuntu and generate one protected private and public key pair for each peer.

Install WireGuard on the Ubuntu server:

```
sudo apt update
sudo apt install wireguard
```

Create the server keys with restrictive file permissions:

```
umask 077
wg genkey | tee server-private.key | wg pubkey > server-public.key
```

On an Ubuntu laptop, install the same package and create a separate pair. On another system, install the official WireGuard app first. Never copy a private key to the other peer.

After creating the keys, inspect the files before moving on:

```
ls -l *-private.key *-public.key
wc -c *-private.key *-public.key
```

The private files should only be readable by your user. Do not compare keys by printing the private value into a terminal recording or chat. Compare public keys instead. Delete this practice pair and create it again once, so key rotation feels like a normal operation rather than an emergency.

Configure the WireGuard peers

Give each interface its private address and key, then authorize the exact address owned by the other peer.

Each peer needs a config with two parts: an `[Interface]` section describing itself, and a `[Peer]` section describing who it talks to.

Create `/etc/wireguard/wg0.conf` on the server:

```
[Interface]
Address = 10.14.0.1/24
ListenPort = 51820
PrivateKey = SERVER_PRIVATE_KEY

[Peer]
```

```
PublicKey = LAPTOP_PUBLIC_KEY
AllowedIPs = 10.14.0.2/32
```

The server describes itself with its VPN address, its listening port, and its private key. The `[Peer]` block authorizes the laptop: traffic from that public key may claim the source address `10.14.0.2`, and traffic destined to `10.14.0.2` goes to that peer.

On the laptop, create the matching peer configuration:

```
[Interface]
Address = 10.14.0.2/24
PrivateKey = LAPTOP_PRIVATE_KEY

[Peer]
PublicKey = SERVER_PUBLIC_KEY
Endpoint = 203.0.113.10:51820
AllowedIPs = 10.14.0.0/24
PersistentKeepalive = 25
```

The asymmetries are worth reading twice. Only the laptop has an `Endpoint`, because only the laptop knows where to find the other side — the server learns the laptop's address from its first authenticated packet, which is what lets the laptop roam between networks. The laptop's `AllowedIPs` covers the whole `10.14.0.0/24`, sending the entire VPN subnet through the tunnel. And `PersistentKeepalive = 25` sends a small packet every 25 seconds so the NAT mapping in the laptop's home router never expires.

AllowedIPs is the setting people misread. It does two jobs at once: it is a routing rule (which destinations go to this peer) and a source filter (which addresses this peer is allowed to use). Narrow on the server — exactly `/32` per client — and wider on the client.

Replace the documented placeholders locally with the keys you generated. Then set private configuration permissions to `600`, because the file contains a private key:

```
sudo chmod 600 /etc/wireguard/wg0.conf
```

The classic mistake here is swapped keys. Each `[Peer]` block must contain the *other* side's public key. Paste a peer's own public key into its config and the handshake never completes — `wg show` will report the peer with no `latest handshake` line, which is your cue to re-check every key.

Start, test, and remove WireGuard

Open the narrow UDP path, start both interfaces, inspect routes and handshakes, then practice a complete cleanup.

Open the narrow UDP path first. Allow UDP port `51820` in the host and provider firewalls:

```
sudo ufw allow 51820/udp
```

Remember the provider side too — a cloud security group that drops UDP will make everything below fail silently.

Start the server interface with `sudo systemctl enable --now wg-quick@wg0`. That starts it now and at every boot. `wg-quick` reads `/etc/wireguard/wg0.conf`, creates `wg0`, assigns the address, and installs the routes.

Start the laptop interface, then inspect what got created with `wg show` and `ip route`:

```
sudo wg-quick up wg0
sudo wg show
# peer: SERVER_PUBLIC_KEY
#   endpoint: 203.0.113.10:51820
#   latest handshake: 12 seconds ago
#   transfer: 3.41 KiB received, 4.72 KiB sent

ip route | grep wg0
# 10.14.0.0/24 dev wg0 proto kernel scope link src 10.14.0.2
```

Two lines matter: a recent `latest handshake`, and a route sending `10.14.0.0/24` into `wg0`.

Now the real test. Ping `10.14.0.1` from the laptop:

```
ping -c 3 10.14.0.1
# 64 bytes from 10.14.0.1: icmp_seq=1 ttl=64 time=18.3 ms
```

Replies prove the whole chain at once: routing into the tunnel, encryption in both directions, and a live interface on the server. If SSH to the server's VPN address works too, you have a private management path that no longer depends on the public one.

If it fails, check the endpoint, keys, `AllowedIPs`, firewall, routes, and latest handshake in that order. The handshake line splits the problem in half: no handshake means the first four items, a handshake without ping replies means the last two. Resist the urge to change several things between attempts — you will fix it and not know which change mattered.

Removal is part of the exercise, because revoking access cleanly is a skill you will need in production. To remove the lab, stop and disable the service, remove the narrow firewall rule, and delete the peer configuration and keys securely:

```
sudo wg-quick down wg0                          # on the laptop
sudo systemctl disable --now wg-quick@wg0       # on the server
sudo ufw delete allow 51820/udp
sudo shred -u /etc/wireguard/wg0.conf
```

Then remove the server when it was created only for this exercise. A forgotten VPN server with valid peer keys is standing access nobody remembers granting.

Check your understanding: building with WireGuard

Check lab addresses, UDP, key handling, peer configuration, AllowedIPs, startup, verification, troubleshooting, and cleanup.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Cloudflare VPN replacement

Understand how Cloudflare Tunnel, the Cloudflare One Client, identity, and policies fit together.

Understand Cloudflare private access

See how a remote device reaches a private resource through Cloudflare without opening an inbound port on the private network.

Cloudflare calls this a VPN replacement. The result is remote access to private resources, but the architecture differs from one VPN server you operate — and the difference is worth understanding before you build it.

Your WireGuard server had a public address and listened for inbound UDP. The whole design depended on that open door. Cloudflare's model inverts it: nothing on the private network listens for anything.

A connector named `cloudflared` creates outbound connections from the private network to Cloudflare. It runs on any machine inside that network that can reach the private resource. It dials out to Cloudflare's edge and keeps those connections alive, waiting for work.

The Cloudflare One Client on the remote device sends matching traffic to Cloudflare. Cloudflare routes it through the connector to the private destination.

```
laptop (Cloudflare One Client)
      outbound connection to Cloudflare

Cloudflare edge   ← identity and policy checks happen here

      outbound connection from inside the private network
cloudflared connector    10.0.1.100 (the private resource)
```

Both arrows point outward. No public inbound port is required on the private network. The firewall stays closed to inbound traffic entirely — there is no UDP `51820` to expose, and nothing for an Internet scanner to find.

Two properties follow from this shape. First, the attack surface changes: a listening VPN port is discoverable and probeable by anyone, while an outbound-only connector offers nothing to connect to. Second, every connection passes Cloudflare's edge, which is where identity and policy get evaluated — access decisions can consider who is asking, not just who holds a key.

The price is that Cloudflare now sits in the path of every connection. We look at that trade honestly in a later lesson.

On the connector machine, health is easy to check:

```
systemctl status cloudflared
#   cloudflared.service - cloudflared
#       Active: active (running)
```

The misconception to drop now: `cloudflared` does not ”open a port” or ”expose the server”. If you find yourself adding an inbound firewall rule for it, you have misread the model — it needs outbound connectivity and nothing else.

Separate the client, Tunnel, and Workers

Give the Cloudflare One Client, Cloudflare Tunnel, and Workers their correct roles instead of calling each one a VPN server.

Three Cloudflare products show up around this lab, and people casually call each of them ”the

VPN". They do different jobs, and mixing them up leads to designs that cannot work.

The **Cloudflare One Client**, formerly WARP, runs on user devices. It creates the device-side path into Cloudflare: it registers the device with your organization, applies your enrollment rules, and forwards the device's matching traffic to Cloudflare's edge. It is the piece your users install.

Cloudflare Tunnel runs near private resources. The `cloudflared` daemon creates the outbound path from Cloudflare into that network. It is the resource-side half of the connection — the counterpart to the client.

Cloudflare Workers runs request-handling code at the edge. Workers respond to requests: useful for APIs, sites, and glue logic. A Worker cannot accept arbitrary inbound TCP connections and is not a general-purpose VPN endpoint. Its TCP socket API opens *outbound* connections from your code; nothing in the platform lets a Worker sit there listening for VPN clients.

```
Cloudflare One Client   device side      "my laptop's traffic enters Cloudflare"
Cloudflare Tunnel       resource side    "Cloudflare can reach into my private network"
Workers                 edge compute     "my code runs on requests" -
not a VPN component
```

Why insist on this? Because "can I run a WireGuard server on Workers?" comes up constantly, and the answer is no by design. WireGuard needs a listening UDP port and raw packet handling. A Worker has neither. Knowing each product's shape saves you from architectures that fail at the whiteboard stage.

For this course lab we need the client and Tunnel. We do not need a Worker.

The role split also tells you where to look when things break. Device-side problems show up in the client:

```
warp-cli status
# Status update: Connected
```

Resource-side problems show up in the connector — `systemctl status cloudflared` on the machine running it, and the Tunnel health indicator in the dashboard. When a private connection fails, checking each half separately beats staring at the whole system at once.

Add identity, devices, and policy

Understand why enrollment is not authorization and restrict private access by user, device, destination, and protocol.

Three separate layers control who reaches what in this architecture. Collapsing them into one mental bucket is how over-broad access happens.

Device enrollment decides who may connect a device to your Zero Trust organization. Passing enrollment means the device is inside the front door. It says nothing about what the device may do next.

A network route says where matching traffic can go. Advertising `10.0.1.0/24` through a Tunnel tells enrolled clients that Cloudflare can carry traffic toward that range. It does not say every enrolled user should reach every destination in it.

Authorization is the third layer, and it is the one people skip. Gateway and Access policies can evaluate identity, destination, port, and device posture — for every connection, at the edge, before traffic reaches your network.

The pattern to copy: start with a catch-all block for the private range. Add narrow allow rules above it for the users and services that need access.

```
priority  action  who                destination      port
1         allow   ana@example.com    10.0.1.100/32    22
2         allow   dev-team group     10.0.2.0/24      443
3         block   everyone           10.0.0.0/8       any
```

Policies evaluate in priority order and the first match wins, so the specific allows fire before the blanket block. Anything you never explicitly allowed gets denied by rule 3.

Device posture deserves a sentence. A policy can require an up-to-date OS version or a corporate certificate before allowing a connection. Access follows the person *and* the machine's state — something a raw WireGuard key can never express, because a key only proves possession.

The failure mode without that block rule is quiet and common: enrollment plus a route means every enrolled device reaches the entire advertised range. The intern's laptop can reach the database port. Nothing errors, nothing warns. It surfaces later, in an audit or an incident.

Verification comes in the lab lesson, but the principle is worth stating now: test one allowed identity and one denied identity, and read the policy logs to see which rule matched each connection. A policy nobody has watched matching is a policy nobody understands.

Know what Cloudflare does not build here

Set honest expectations about egress location, provider trust, server control, product limits, and the difference from a consumer VPN.

This lab gives a remote device access to a private network. Before building it, be precise about what it does not give you — the marketing around ”VPN replacement” invites confusion.

It does not create an anonymous consumer VPN service. Your organization's administrator can see traffic logs, because visibility is the product's point. If your goal is hiding activity from an observer, this tool is aimed in the opposite direction: it exists so that someone — you, the admin — sees more, not less.

It does not create a Worker-based WireGuard server either. The previous lesson covered why: Workers cannot terminate inbound VPN connections, whatever creative blog posts suggest.

Cloudflare becomes part of the traffic path and trust model. You depend on its account, client, routing, policies, and service availability. When Cloudflare has an outage, your people cannot reach internal tools, and there is no server of yours to restart. That is the same shape of trade a consumer VPN customer makes — trusting an operator — with a different company and an actual contract.

You also give up server-level control. There is no root shell on Cloudflare's edge. You cannot capture packets in the middle of the path. You debug through the dashboards and logs Cloudflare chooses to expose.

```
you still own       the private network, the resources, the policy decisions
you now delegate    transport, client software, policy enforcement, availability
you cannot get      anonymity, protocol control, root access to the middle
```

Product limits are real and they move. The Zero Trust Free plan currently targets teams under 50 users. Plan details and dashboard labels can change, so check the current pricing and documentation rather than trusting any course or blog post — including this one.

So when does each design win? Use traditional WireGuard when you want to own the endpoint and protocol configuration, or when a third party in the path is unacceptable. Use Cloudflare private access when identity policy and outbound-only connectivity fit the job better than key distribution does.

The mistake to avoid: treating today's free tier as a permanent architectural guarantee. If your access design only works because a plan is currently free, revisit that assumption on a schedule.

Check your understanding: Cloudflare VPN replacement

Check connector roles, outbound-only tunnels, Workers limits, enrollment, policy, trust, plan limits, and appropriate use cases.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Build a Cloudflare private network

Connect a private network, enroll a remote device, restrict access, and test the complete path.

Create the Zero Trust organization

Create the account boundary, choose the free plan deliberately, record the team name, and restrict who may enroll a device.

Everything in this module lives inside one container: the Zero Trust organization. Devices enroll into it, tunnels belong to it, policies apply within it. Creating it carefully now saves confusion later.

Create a Cloudflare account, then open the Zero Trust dashboard at `one.dash.cloudflare.com` and follow the onboarding. You will pick a plan along the way. The free plan may still ask for payment details even when the selected plan costs nothing — expected, not a dark pattern you triggered by mistake.

During onboarding you choose the **team name**. Record it. It becomes your organization's identity, visible in the enrollment domain, and remote devices use it when they log in through the Cloudflare One Client. Pick something short you can type on a phone keyboard.

```
team name         flavio-lab
org login domain  flavio-lab.cloudflareaccess.com
enrollment rule   allow only flavio@example.com
```

If you lose the team name later, it is in the dashboard settings — but the client login screen asks for it before anything else, so keeping it written down avoids a silly roadblock.

Next, before enrolling anything, open the device enrollment permissions and allow only the email identity you will use for the lab. The default One-time PIN login method emails a code to the address a rule allows, so you do not need to configure an identity provider for this exercise.

Verify the boundary works by visiting your enrollment domain in a browser:

```
open https://flavio-lab.cloudflareaccess.com
# the login page should offer a one-time PIN for your email
```

An email outside your rule should fail to get in. That failed attempt is a test worth running on purpose.

Do not use an Everyone rule for a real deployment. Enrollment is the front door to your organization, and the team name is not a secret — it appears in login URLs and can be guessed. An Everyone rule means anyone who learns the name can join a device to your organization and start knocking on your policies from the inside.

Create the Tunnel and private route

Run the outbound connector beside a private resource and route one narrow IP or CIDR through it.

In the Cloudflare One onboarding flow, choose the device-to-network VPN replacement path and create a Tunnel. A Tunnel is a named connection slot in your organization; the `cloudflared` daemon fills it with live outbound connections.

Placement is the decision that matters. Install `cloudflared` on a Linux, macOS, or Windows device that can already reach the private resource. The connector is a bridge: Cloudflare hands it traffic, and it delivers that traffic over the local network. If the connector cannot reach the resource, nothing upstream can compensate.

Prove local reachability before involving Cloudflare at all:

```
# on the machine that will run cloudflared
ping -c 2 10.0.1.100
curl -I http://10.0.1.100
# HTTP/1.1 200 OK
```

Use the install command generated by the dashboard. It embeds a token that binds this daemon to your Tunnel:

```
sudo cloudflared service install eyJhIjoiN2Q4M2M1...
systemctl status cloudflared
#   cloudflared.service - cloudflared
#       Active: active (running)
```

Then define what this Tunnel carries. Add a narrow route such as `10.0.1.100/32` for one lab host. The route tells Cloudflare: traffic for this prefix belongs to this Tunnel. Choose a larger CIDR only when every address in it should be reachable through this connector — a route is a reachability promise, and promising an entire `/8` to reach one machine is the network equivalent of handing out a master key.

Wait until the dashboard reports the connector as healthy before configuring the remote device. Healthy means `cloudflared` has established its outbound connections to Cloudflare's edge. Debugging device enrollment against a connector that was never up wastes an evening, because every symptom looks identical from the laptop side.

The classic placement mistake: installing `cloudflared` on a machine in the wrong network segment — a different VLAN, or behind an internal firewall that blocks it from the resource. The symptom appears much later: the remote device enrolls fine, the Tunnel shows healthy, and connections to `10.0.1.100` time out anyway. The `ping` test above, run from the connector's own shell, is what splits that problem in half.

Enroll the remote device and route traffic

Connect the Cloudflare One Client to the organization and make the private CIDR use its tunnel without breaking local private networks.

Install the Cloudflare One Client on the remote device — the laptop playing the role of your remote worker. Log in to Cloudflare Zero Trust with the team name and allowed identity. The client asks for the team name first, then opens a browser window for the identity check. Only the email your enrollment rule allows gets through, which is the rule doing its job.

Use **Traffic and DNS mode** for the complete lab. The client has several operating modes, and some of them only filter DNS queries without carrying packets. This lab needs both traffic and name resolution flowing through Cloudflare, so anything less will produce confusing half-working behavior.

Now the routing question. Cloudflare excludes RFC 1918 private ranges by default. That default exists for a good reason: it keeps home printers, NAS boxes, and local networks working for every enrolled device. But your lab resource lives inside one of those excluded ranges, so right now its traffic never leaves the laptop's local network.

Adjust Split Tunnels so the exact lab IP or CIDR goes through the client. In the default Exclude mode, that means editing the entry covering your lab range — remove the broad `10.0.0.0/8` exclusion and re-add the narrower pieces you still want excluded, leaving your lab address routed.

Then verify from the remote device, ideally on a different network than the resource — a phone hotspot makes the test honest:

```
ping -c 2 10.0.1.100
# 64 bytes from 10.0.1.100: icmp_seq=1 ttl=63 time=42.1 ms
```

Keep the routed range narrow. A broad private range can overlap the user's home network and send local devices to the wrong place — the laptop suddenly cannot print at home because `192.168.1.9` is being shipped into the tunnel.

When a routed IP is unreachable from an enrolled device, check the Split Tunnels list before anything else. Forgetting this adjustment is the single most common failure in the whole lab: the device is connected, the connector is healthy, every dashboard light is green, and the packets quietly stay local.

Restrict and verify Cloudflare access

Add a narrow allow policy, block the rest of the private range, and prove both the permitted and denied paths.

Everything connects now. This lesson is about making sure not everything is *allowed* — and proving it.

Enable the Gateway proxy for the protocols your lab needs. Network policies only apply to traffic the proxy handles, so switch on TCP at minimum, plus UDP and ICMP if you want your ping tests governed by policy too.

Create an allow policy for the test identity, destination, and port. Then add a lower-priority catch-all block for the private range. This prevents every enrolled device from receiving broad network access:

```
priority  action  who              destination     port
```

```
1          allow    you@example.com    10.0.1.100/32    22, 80
2          block    everyone           10.0.0.0/8       any
```

Now prove the allowed path. From the remote device, open the private HTTP service or run `ssh 10.0.1.100`:

```
ssh flavio@10.0.1.100
curl -I http://10.0.1.100
# HTTP/1.1 200 OK
```

Then test a destination or port that should be denied. Port `2222` on the same host falls outside your allow rule, so the catch-all should swallow it:

```
ssh -o ConnectTimeout=5 -p 2222 flavio@10.0.1.100
# ssh: connect to host 10.0.1.100 port 2222: Connection timed out
```

The denied test matters as much as the allowed one. A policy that permits correctly but fails to block is broken in the dangerous direction, and nothing on the happy path will ever tell you.

Read the Gateway network logs after both tests. Each connection shows which policy matched it — your allow rule for the first pair, your block rule for the third. If results confuse you, check the device connection, Tunnel health, routes, policy logs, and destination service, in that order.

Save evidence for both the allowed and blocked tests: log excerpts or screenshots. When someone later asks "who can reach this system?", you answer with evidence instead of belief.

The classic misconfiguration is inverted priority — the block rule sitting above the allows. Everything gets denied, and the logs show the block rule matching your own test connection. The logs make the fix obvious, which is one more reason to read them now, while nothing is on fire.

Check your understanding: building with Cloudflare

Check team names, enrollment, connectors, private routes, client modes, split tunnels, policy order, tests, and evidence.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Operate and choose

Troubleshoot by layer, maintain access safely, and choose the right design for each job.

Troubleshoot the path by layer

Locate a failure by checking the device, route, Cloudflare edge, connector, destination network, and service in order.

Start on the remote device. Confirm the client is connected to the intended team and the destination matches a routed prefix.

Check Tunnel health and whether `cloudflared` can reach the destination locally.

Review Gateway or Access policy logs for an explicit block. Then check the destination firewall, listening address, and service logs.

Change one layer at a time. A broad allow rule can hide the real problem and leave permanent access behind.

Start with evidence from both peers:

```
sudo wg show
ip address show wg0
ip route get 10.14.0.1
ping -c 3 10.14.0.1
```

No recent handshake points toward endpoint, key, UDP, or firewall trouble. A handshake with no useful traffic points toward AllowedIPs, routes, forwarding, DNS, or MTU. Change one layer at a time. Save the before and after output so you can explain which change fixed the path.

Diagnose routes, DNS, and MTU

Recognize the common patterns produced by overlapping prefixes, missing private DNS, and packets that are too large for the tunnel path.

Three failure patterns cover most VPN support tickets. Learn their shapes and you diagnose in minutes what others chase for hours.

Pattern one: the same private IP works on one network but not another. That is the signature of an address overlap. Compare local and tunnel routes for overlap — the hotel Wi-Fi happens to use the same range as your company, and longest-prefix matching hands your packets to the local network:

```
ip route get 10.0.1.100
# 10.0.1.100 via 192.168.1.1 dev wlan0 src 192.168.1.34
# a physical interface answered - the tunnel never saw this packet
```

When the output names a physical interface instead of the tunnel, the local network won the routing decision. The fix is a more specific tunnel route, or moving to a network that does not collide.

Pattern two: an IP works but a private hostname fails. Routing is fine; name resolution took the wrong path. Inspect DNS resolver policy and query logs:

```
dig wiki.internal
# status: NXDOMAIN   ← a public resolver answered; it has never heard of this name

resolvectl status
# check which DNS server each interface is actually using
```

`NXDOMAIN` from a public resolver means your queries are not reaching the private DNS server that knows the name.

Pattern three: small requests work but larger transfers stall. The login page loads and the download freezes — that is MTU territory. Test packet sizes and inspect MTU rather than changing application code:

```
ping -c 1 -M do -s 1300 10.0.1.100   # replies arrive
ping -c 1 -M do -s 1450 10.0.1.100   # silence: the path limit sits between the two
```

Narrow the range until you find the boundary, then compare it with the tunnel interface's MTU.

Write down the expected path before debugging: one sentence like ”packets to 10.0.1.100 leave via

the tunnel, names resolve at 10.0.1.53". Then compare the route table and logs with that picture. Without the expected picture, every command output looks plausible, and debugging degrades into running commands and hoping.

Maintain access as a security system

Review peers, enrolled devices, policies, connector versions, logs, recovery access, and removal procedures as one system.

A VPN you built once and never touched again drifts toward being a liability. Access systems need operation, not just installation.

Start with the population of things that can connect. Remove lost or retired devices quickly. Revoke WireGuard peers and Cloudflare device registrations that no longer belong. On the WireGuard side, the interface itself tells you which peers are alive:

```
sudo wg show wg0 latest-handshakes
# xTIBA5rboUvnH4htodjb6e697QjLERt1NAB4mZqp8Dg=   1785171243
# mAbc9xkQGVpDzR7wFJ2sLuNvTq81cYheKf0iW5nBOEU=   0
```

The number is a Unix timestamp of the last handshake. A `0` means that peer has never connected — so why is it still configured? A peer silent for months is a removal candidate: delete its `[Peer]` block from `wg0.conf` and restart the interface. On the Cloudflare side, review the device list in the dashboard and revoke stale registrations, and re-check that the enrollment rule still lists only the identities you intend.

Patch the VPN endpoint and connector host. These machines process input from the Internet by design; running last year's `cloudflared` or an unpatched kernel on the WireGuard server undermines everything the tunnel protects.

Review who can edit routing and policy. A dashboard admin is a network admin, whatever their job title says.

Keep logs long enough for the risks and plan you chose. Handshake history, policy logs, and connector logs are what turn "something looks wrong" into an answer.

Test recovery without weakening access permanently. The classic failure: an outage, someone adds a broad allow rule "temporarily", and the rule outlives the incident by years. Time-box exceptions and put the removal date in the rule's name, where the next reviewer will trip over it.

A tunnel that still connects is not necessarily healthy. Re-run both allowed and denied tests after meaningful changes — a routing edit can widen access quietly while every existing connection keeps working.

My advice: a fifteen-minute monthly review. Peer list, device list, policies, versions, one allowed test, one denied test. Written down, every time. Boring is exactly what you want from an access system.

Choose and clean up the right design

Select traditional VPN, Cloudflare private access, or direct application access from the actual requirement, then remove the lab completely.

You built the same capability twice in this course: once with WireGuard, once with Cloudflare. Neither is "the right one". Each fits different requirements.

Choose WireGuard when you want a small protocol, control of both peers, and your own endpoint. Nothing sits between you and your traffic. The cost is that everything is yours: key distribution, server patching, routing, DNS, and firewall design.

Choose Cloudflare private access when outbound-only connectivity, managed clients, identity, and destination policy are central requirements. You trade a third party in the path for policies that understand who is connecting, not just which key they hold.

And sometimes neither is right. For one web application, Cloudflare Access through a browser may be narrower than network access — the user reaches one app and touches no network at all. For two networks that must behave as one, a site-to-site design fits better than per-device clients ever will.

```
one device → your network, full control        WireGuard remote access
workforce → internal apps, identity rules      Cloudflare private access
a few people → one web app                     Access in front of the app, no tunnel
network   network, permanently                 site-to-site design
```

Decide from the sentence you wrote back in the first module: who must reach what. The design falls out of the sentence.

Now finish the course by removing the lab completely: lab peers, keys, routes, policies, enrolled devices, tunnels, firewall rules, and any paid server. Leftover access paths are a real risk — a forgotten peer key is a credential, and a forgotten tunnel is a door.

Then confirm the private resource is unreachable afterward. The verification of a cleanup is a failed connection:

```
ping -c 2 -W 2 10.0.1.100
# 2 packets transmitted, 0 received, 100% packet loss

ssh -o ConnectTimeout=5 flavio@10.0.1.100
# ssh: connect to host 10.0.1.100 port 22: Operation timed out
```

Here, the timeout is the passing test.

The cleanup mistake people make is walking only one list. Deleting the cloud server while the Zero Trust organization still holds routes and enrolled devices leaves config pointing at nothing — or worse, at whatever gets that IP next. Walk both inventories: the cloud provider's resources and the Cloudflare organization's configuration. Done means both are empty.

Check your understanding: operating and choosing

Check layered troubleshooting, overlap, DNS, MTU, revocation, maintenance, design selection, denied tests, and complete cleanup.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/vpn/.