VPN

A Complete Practical Course

From understanding what a VPN actually protects you from, through building WireGuard and OpenVPN servers from scratch, connecting every platform, using commercial providers intelligently, and verifying nothing is leaking.

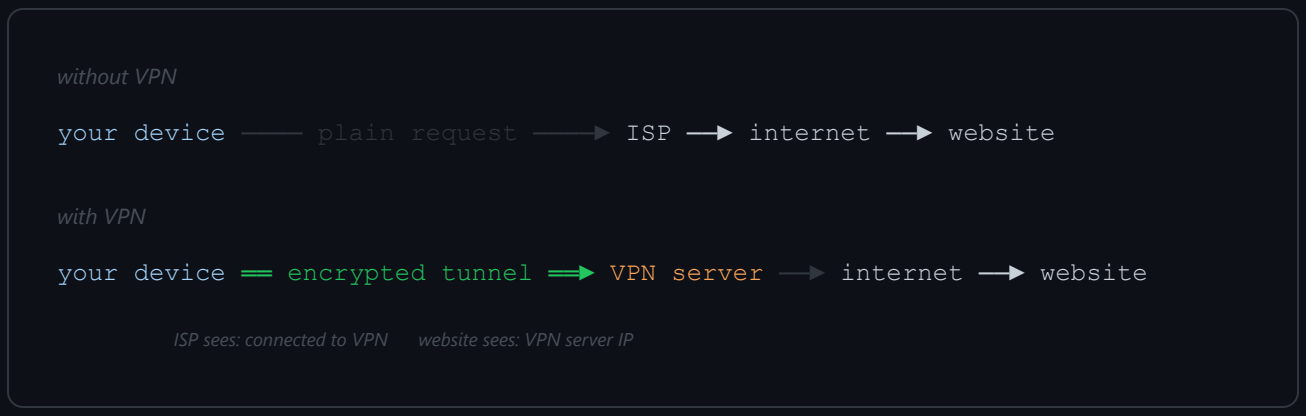
8 Chapters | WireGuard · OpenVPN · ProtonVPN · Mullvad

Chapter 1 — What is a VPN and Why?

A **Virtual Private Network** creates an encrypted tunnel between your device and a server somewhere else on the internet. All your network traffic travels through that tunnel — your ISP sees only that you're connected to a VPN server, and the websites you visit see the VPN server's IP address, not yours.

That's the core promise. But VPNs are surrounded by more marketing noise than almost any other technology, so this chapter focuses on something more important than the setup instructions: understanding precisely what a VPN actually does and doesn't protect, and what problem you're solving before you reach for one.

How It Works — The Tunnel



The encrypted tunnel is established using a VPN protocol — **WireGuard**, **OpenVPN**, or **IKEv2/IPsec** being the most common today. Each protocol handles key exchange, encryption, and packet encapsulation differently. Chapter 2 covers the protocol comparison in detail. For now, the important thing is what the tunnel achieves:

- **Your ISP** can no longer see which sites you visit or what you send — only that you have a VPN connection open.
- **Anyone on your local network** (a café Wi-Fi operator, a hotel network) cannot inspect your traffic.
- **Websites and services** see the VPN server's IP address and country, not yours.

Legitimate Use Cases

Public Wi-Fi protection

Coffee shops, hotels, airports — any shared network. Encrypts your traffic so the network operator (or anyone running a packet sniffer on the same network) can't read it. The single most practical use case.

Accessing your home network remotely

A self-hosted WireGuard or OpenVPN server at home lets you access your NAS, home server, printers, and local devices as if you were on the LAN — from anywhere. This is the most compelling use case for self-hosting a VPN.

Corporate remote access

Companies use VPNs to let remote employees access internal systems — intranet, internal APIs, file servers — that aren't exposed to the internet. The reason "VPN" often means "work VPN" to most people.

Bypassing geographic restrictions

Streaming services, news sites, and some government services restrict content by IP location. Connecting through a VPN server in another country appears to come from that country. Works until the service blocks the VPN's IP range.

ISP traffic shaping avoidance

Some ISPs throttle specific traffic (streaming, torrents). Since the ISP can't see what's inside the VPN tunnel, traffic shaping based on content type no longer works.

Hiding activity from your ISP

In countries where ISPs are required to log browsing history, a VPN moves trust from the ISP to the VPN provider. Only worthwhile if the VPN provider is more trustworthy than the ISP — and subject to their jurisdiction.

What a VPN Does and Doesn't Protect

A VPN protects against...	A VPN does NOT protect against...
✓ Your ISP logging the sites you visit	✗ Tracking cookies and browser fingerprinting
✓ Packet sniffing on shared/public Wi-Fi	✗ Malware already on your device
✓ Websites seeing your real IP address	✗ Phishing — clicking a bad link still works

- ✓ IP-based geographic blocking
- ✓ Basic traffic analysis by a local network operator
- ✓ ISP selling your browsing data (where legal)

- ✗ Google/Facebook tracking you when logged in
- ✗ A VPN provider who logs your traffic
- ✗ DNS leaks if misconfigured
- ✗ WebRTC IP leaks in browsers
- ✗ Account takeover or data breaches at services you use

> **The trust shift, not the trust removal:** A VPN doesn't make you anonymous — it moves trust from your ISP to your VPN provider. Your VPN provider can see everything your ISP previously could. The relevant question is: who do you trust more? For a commercial VPN, that means reading the provider's logging policy carefully and understanding which country's laws they operate under.

Common VPN Myths

Myth: "A VPN makes me anonymous online."
False. You're authenticated with the VPN provider, and if you're logged into any account (Google, Facebook, your email), the service knows exactly who you are regardless of which IP you connect from. A VPN hides your IP from the sites you visit; it does not hide your identity.

Myth: "A VPN protects me from hackers."
Mostly false. A VPN encrypts traffic between your device and the VPN server — it doesn't protect against malware, unpatched software, weak passwords, or phishing. Most "hacking" happens at the application layer, which the VPN doesn't touch.

Myth: "I need a VPN to be safe on public Wi-Fi."
Largely outdated. In 2026, almost all websites use HTTPS, which already encrypts traffic end-to-

end. A packet sniffer on café Wi-Fi sees encrypted HTTPS data, not your passwords. A VPN adds a second layer, but the threat it solves here is much smaller than it was in 2010.

Myth: "A free VPN is fine."
Almost never. Free VPN services need to monetise somehow — commonly by logging and selling your traffic data, which is exactly what you're trying to avoid. Some have been caught injecting ads, harvesting credentials, or running malware through the tunnel.

Matching a VPN to Your Threat Model

The right question isn't "should I use a VPN?" but "what am I protecting against, and is a VPN the right tool for it?" These scenarios show where a VPN helps and where it doesn't:

Using hotel Wi-Fi for work
Risk: network operator or another guest sniffing traffic.
VPN helps — encrypts the local hop

Hiding browsing from your ISP
Risk: ISP logging habits and selling data or complying with government requests.
VPN helps — but VPN provider now holds the same data

Accessing home server remotely
Need: reach internal devices without exposing ports.
Self-hosted VPN is ideal

Avoiding targeted advertising
Risk: ad networks tracking you across sites.
VPN doesn't help — cookies and fingerprinting work regardless

Watching geo-blocked content
Need: appear to be in a different country.
Works until the service blocks VPN IP ranges

Staying safe from malware
Risk: clicking bad links, drive-by downloads.
VPN doesn't help — use updated AV and a browser with safe-browsing

Self-Hosted vs Commercial VPN

The remaining chapters in this course focus on **self-hosted VPNs** — running your own WireGuard or OpenVPN server. This is different from a commercial VPN service like ProtonVPN or Mullvad:

- **Self-hosted** — you run the server. Traffic exits from your home IP or a VPS you control. No third-party provider to trust. Ideal for accessing your own network remotely. Does not hide your IP from websites (they see your home IP).
- **Commercial VPN** — a company runs thousands of servers. Traffic exits from a shared IP pool in whichever country you choose. Useful for geo-unblocking and hiding your IP. You must trust the provider's no-logs claims.

> **This course covers both.** Chapters 3–6 cover setting up WireGuard and OpenVPN servers yourself. Chapter 7 covers using commercial VPN clients (ProtonVPN and Mullvad) on Linux, including CLI usage and DNS leak testing. Chapter 8 brings it together with troubleshooting and security hardening.

Next — Chapter 2: VPN Protocols Compared. WireGuard, OpenVPN, IKEv2/IPsec, and L2TP are not interchangeable — each makes different trade-offs between speed, security, and compatibility. Chapter 2 explains how each works and when to choose which.

Chapter 2 — VPN Protocols Compared

A VPN protocol defines how two endpoints authenticate each other, negotiate encryption keys, and wrap your traffic in a secure tunnel. The protocol choice affects speed, battery life, firewall traversal, and the size of the attack surface you're trusting. Four protocols dominate real-world deployments — this chapter explains what each one is, how it works, and when to choose it.

WireGuard

WireGuard

Recommended — new deployments

WireGuard was merged into the Linux kernel in 2020 and has quickly become the default recommendation for new VPN deployments. Its design philosophy is radical minimalism: the entire codebase is around **4,000 lines** compared to OpenVPN's ~100,000. Fewer lines means fewer places to hide vulnerabilities, and a codebase small enough for independent security researchers to audit in full. It uses a fixed, modern set of cryptographic primitives — there are no negotiable cipher suites, which eliminates a whole class of downgrade attacks.

WireGuard operates at the kernel level and uses UDP. Handshakes are silent by default (the server doesn't respond to probes unless it recognises the initiating key), which makes WireGuard servers effectively invisible to port scanners.

Speed	Security	Code size	Transport
Fastest	Excellent	~4,000 lines	UDP only

Strengths

- Fastest of the four — kernel-level performance
- Tiny, auditable codebase
- Excellent battery life on mobile (stateless reconnection)
- Simple config files — easy to understand and automate

Limitations

- UDP only — blocked by some restrictive firewalls
- Static IPs for peers (privacy consideration — IP is logged per session)
- No built-in obfuscation — identifiable as WireGuard traffic
- Relatively new — less deployed than OpenVPN in legacy environments

- Built into Linux kernel, Windows, macOS, iOS, Android
- Silent server — invisible to port scanners

OpenVPN

OpenVPN

Solid — flexible, widely supported

OpenVPN has been the gold standard for open-source VPNs since 2001. It runs in userspace (not the kernel), uses TLS for the control channel, and can tunnel over either **UDP or TCP**. Running on TCP port 443 makes OpenVPN traffic visually indistinguishable from HTTPS, which is why it can pass through almost any firewall — including those that block WireGuard's UDP. This firewall traversal capability is OpenVPN's primary advantage over WireGuard for challenging network environments.

The large codebase and flexible cipher negotiation that made OpenVPN powerful also make it harder to audit and configure correctly. Misconfiguration (weak ciphers, missing `--tls-auth`, self-signed certs without verification) is a real risk.

Speed	Security	Code size	Transport
Good	Good (if configured correctly)	~100,000 lines	UDP or TCP

Strengths

- TCP mode on port 443 — passes through almost any firewall
- Mature — 20+ years of real-world deployment and auditing
- Flexible cipher configuration
- Certificate-based auth — fine-grained per-client revocation
- Supported by nearly every router, OS, and commercial VPN

Limitations

- Slower than WireGuard — userspace overhead
- Complex setup (PKI, Easy-RSA, certificate management)
- Large attack surface — more code = more potential vulnerabilities
- TCP-over-TCP (when tunnelling TCP traffic) causes performance issues on lossy links

IKEv2 / IPsec

IKEv2 / IPsec

Good — native on iOS/macOS/Windows

IKEv2 (Internet Key Exchange version 2) is the key exchange protocol; IPsec handles the actual encryption. Together they form the VPN built natively into iOS, macOS, Windows, and Android — meaning no client software needs to be installed on mobile devices. IKEv2 has a **MOBIKE** extension that handles network changes (Wi-Fi → cellular) seamlessly, making it the traditional choice for mobile VPNs. Speed is competitive with OpenVPN and often better on mobile hardware that has IPsec acceleration.

Speed	Security	Mobile roaming	Transport
Fast (hardware accel)	Strong	Excellent (MOBIKE)	UDP 500/4500

Strengths

- Native OS support — no client app needed on iOS/macOS/Windows
- MOBIKE handles network changes without dropping the tunnel
- Fast reconnection after sleep/standby
- Hardware IPsec acceleration on many devices

Limitations

- Uses UDP 500/4500 — may be blocked on restrictive networks
- Complex server setup (StrongSwan / libreswan)
- IPsec has a complicated history of NSA involvement concerns
- Less commonly self-hosted than WireGuard or OpenVPN

L2TP / IPsec

L2TP / IPsec

Legacy — avoid for new setups

L2TP (Layer 2 Tunnelling Protocol) has no encryption of its own — it is always paired with IPsec to provide security. The combination was widely used in the 2000s and early 2010s and is still found in many older routers and NAS devices. It is the slowest of the four: traffic is encapsulated twice (L2TP inside IPsec), doubling overhead. It also uses fixed UDP port 1701 which is easily blocked, and the pre-shared key authentication most consumer setups use is considerably weaker than certificate-based alternatives. There is no good reason to set up L2TP for a new deployment.

Speed	Security	Overhead	Use case

Slowest	Adequate (if PSK is strong)	Double encapsulation	Legacy / NAS devices

Strengths

- Built into almost every OS and router
- Useful if your hardware only supports L2TP

Limitations

- Slowest — double encapsulation overhead
- Fixed port easily blocked by firewalls
- Weak pre-shared key auth in most consumer setups
- No forward secrecy with PSK mode
- No reason to choose this over WireGuard for a new setup

Side-by-Side Comparison

Feature	WireGuard	OpenVPN	IKEv2/IPsec	L2TP/IPsec
Speed	Fastest	Good	Fast	Slow
Code size	~4,000 lines	~100,000 lines	Complex	Complex
Transport	UDP only	UDP or TCP	UDP 500/4500	UDP 1701 (fixed)
Firewall traversal	Moderate	Excellent (TCP 443)	Moderate	Poor
Mobile roaming	Excellent	Good	Excellent (MOBIKE)	Adequate
Native OS support	All (kernel/built-in)	Client required	iOS/macOS/Win native	All (built-in)
Setup complexity	Simple	Moderate (PKI)	Complex	Simple (PSK)
Forward secrecy	Yes	Yes	Yes	No (PSK mode)
Recommended for	New deployments	Firewall bypass	iOS/macOS native	Legacy only

Encryption Primitives Used

WireGuard's fixed cryptography is one of its defining features — you can't accidentally configure a weak cipher. The others offer configurability, which is both a strength and a risk:

WireGuard — fixed, modern

Key exchange: Curve25519
Encryption: ChaCha20-Poly1305
Hashing: BLAKE2s
No cipher negotiation — one suite, always

OpenVPN — configurable

Control channel: TLS 1.2 / 1.3
Data cipher: AES-256-GCM (recommended)
HMAC: SHA-256 / SHA-512
Avoid: BF-CBC, DES, RC2 (legacy defaults)

IKEv2 / IPsec — configurable

Key exchange: Diffie-Hellman group 14+
Encryption: AES-256-GCM (recommended)
PRF/integrity: HMAC-SHA2-256
Avoid: 3DES, group 1/2 DH

L2TP / IPsec — typically PSK

Auth: Pre-shared key (consumer)
Encryption: AES-256 (if configured)
Risk: weak PSK + no forward secrecy
Avoid in new setups

Which Protocol Should You Choose?

New self-hosted VPN server

→ WireGuard

Fastest, simplest config, smallest attack surface. Chapters 3–4 cover the full setup.

Must work through restrictive firewalls

→ OpenVPN on TCP 443

HTTPS traffic on 443 is almost never blocked. Set `proto tcp` and `port 443` in the server config.

iOS or macOS devices, no app install

→ IKEv2/IPsec

Built into Apple devices natively. WireGuard also works well on iOS/macOS via the App Store app.

Existing router only supports L2TP

→ L2TP/IPsec with a strong PSK

Acceptable if you have no choice. Use a long random pre-shared key and consider upgrading the hardware.

Commercial VPN subscription

→ WireGuard (or OpenVPN)

Most quality providers now offer WireGuard. Fall back to OpenVPN if WireGuard is blocked on a given network.

Maximum compatibility across all devices

→ OpenVPN

20+ years of client support across every platform. Every commercial VPN, router, and NAS that supports VPN supports OpenVPN.

> **This course uses WireGuard first.** Chapters 3 and 4 cover building and connecting to a WireGuard server — the recommended starting point for any new self-hosted VPN. Chapters 5 and 6 then cover OpenVPN for situations where WireGuard won't work.

Next — Chapter 3: WireGuard Server. Install WireGuard on a Linux server, generate key pairs, write `wg0.conf`, bring the interface up, and verify a client can connect — all from scratch.

Chapter 3 — WireGuard Server

WireGuard is built into the Linux kernel since version 5.6, which means on any modern Debian or Ubuntu machine there is nothing to compile — just install the userspace tools and configure the interface. By the end of this chapter you'll have a working VPN server that clients can connect to securely.

What We're Building

The server gets the first address in the VPN subnet (`10.0.0.1`). Each client gets its own address (`10.0.0.2`, `10.0.0.3`, ...). Traffic between clients and the server travels through the encrypted WireGuard tunnel. Optionally you can route *all* client internet traffic through the server too — useful when on untrusted Wi-Fi.

Step 1 — Install WireGuard

server — install WireGuard tools

```
# Debian 11+ / Ubuntu 20.04+ — WireGuard is in the standard repos root@server:~#
apt update && apt install -y wireguard # confirm the kernel module is available
root@server:~# modprobe wireguard && echo "module loaded" module loaded # check
the installed tools root@server:~# wg --version wireguard-tools v1.0.20210914 -
https://git.zx2c4.com/wireguard-tools/
```

Step 2 — Generate Key Pairs

WireGuard uses Curve25519 public/private key pairs for authentication. Each peer (server and every client) needs its own key pair. The private key never leaves the machine it was generated on.

server — generate server key pair

```
# work in /etc/wireguard — lock down permissions immediately root@server:~# cd
/etc/wireguard root@server:/etc/wireguard# umask 077 # generate the server
private key root@server:/etc/wireguard# wg genkey > server_private.key # derive
the server public key from the private key root@server:/etc/wireguard# wg pubkey
< server_private.key > server_public.key # generate the first client's key pair
(do this for each client) root@server:/etc/wireguard# wg genkey >
client1_private.key root@server:/etc/wireguard# wg pubkey < client1_private.key >
client1_public.key # optional but recommended: preshared key for extra symmetric
encryption root@server:/etc/wireguard# wg genpsk > client1_psk.key # view the
keys — you'll paste these into config files root@server:/etc/wireguard# cat
server_private.key qH8X2vK9mN3pL7rT4wY1cB6sF5nJ0dA8eI2oU9gM= ← server private
(keep secret) root@server:/etc/wireguard# cat server_public.key
P3mK9vX2nL7rH8T4wY1cB6sF5nJ0dA8eI2oU9gQ= ← share with clients
```

Private keys must stay private. Never paste a private key into a chat message, email, or web form. The key files in `/etc/wireguard/` should be owned by root and readable only by root (`chmod 600`). Anyone with the server private key can impersonate your VPN server.

Step 3 — Write the Server Config (wg0.conf)

The server config has two sections: `[Interface]` defines this machine, and one `[Peer]` block per authorised client.

```
# /etc/wireguard/wg0.conf — server configuration

[Interface]
Address    = 10.0.0.1/24           # server's VPN IP
ListenPort = 51820                 # UDP port clients connect to
PrivateKey = qH8X2vK9mN3pL7rT...   # paste server_private.key here
```

```
# IP forwarding — needed if you want clients to route internet traffic through t
PostUp   = iptables -A FORWARD -i %i -j ACCEPT; \
           iptables -A FORWARD -o %i -j ACCEPT; \
           iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
PreDown  = iptables -D FORWARD -i %i -j ACCEPT; \
           iptables -D FORWARD -o %i -j ACCEPT; \
           iptables -t nat -D POSTROUTING -o eth0 -j MASQUERADE

# ── client 1 ──────────────────────────────────────────────
[Peer]
PublicKey    = X7nK2vP9mL3rH8T4...   # paste client1_public.key here
PresharedKey = R5mT9vX2nL7rH8P4...   # paste client1_psk.key (optional)
AllowedIPs   = 10.0.0.2/32           # only this IP is allowed from this peer

# add more [Peer] blocks for each additional client
# [Peer]
# PublicKey  = ...client2 public key...
# AllowedIPs = 10.0.0.3/32
```

> **Replace eth0 with your actual interface name.** On modern Debian/Ubuntu systems the external interface might be `ens3`, `enp0s3`, or `ens18`. Check with `ip link show` or `ip route | grep default`.

Step 4 — Enable IP Forwarding

If you want VPN clients to route their internet traffic through the server (full-tunnel mode), the server kernel needs IP forwarding enabled. Without this, the `PostUp` iptables rules won't route traffic onward:

server — enable IP forwarding

```
# enable immediately (until next reboot) root@server:~# sysctl -w
net.ipv4.ip_forward=1 # make it permanent across reboots root@server:~# echo
"net.ipv4.ip_forward=1" >> /etc/sysctl.conf root@server:~# sysctl -p
net.ipv4.ip_forward = 1 # verify root@server:~# cat /proc/sys/net/ipv4/ip_forward
1
```

Step 5 — Open the Firewall

server — UFW rule for WireGuard

```
root@server:~# ufw allow 51820/udp root@server:~# ufw status Status: active To
Action From -- ------ ---- 51820/udp ALLOW Anywhere
```

Step 6 — Start WireGuard and Enable on Boot

server — bring up the interface

```
# bring the wg0 interface up now root@server:~# wg-quick up wg0 [#] ip link add
wg0 type wireguard [#] wg setconf wg0 /dev/fd/63 [#] ip -4 address add
10.0.0.1/24 dev wg0 [#] ip link set mtu 1420 up dev wg0 [#] iptables -A FORWARD -
i wg0 -j ACCEPT ... # enable as a systemd service — starts automatically on boot
root@server:~# systemctl enable --now wg-quick@wg0 root@server:~# systemctl
status wg-quick@wg0 ● wg-quick@wg0.service - WireGuard via wg-quick(8) for wg0
Active: active (exited) since Thu 2026-06-18 10:22:11 BST # show current
WireGuard status and peer list root@server:~# wg show interface: wg0 public key:
P3mK9vX2nL7rH8T4wY1cB6sF5nJ0dA8eI2oU9gQ= private key: (hidden) listening port:
51820
```

Step 7 — Write the Client Config

Generate this on the server using the client's private key, then transfer it securely to the client device (QR code, encrypted file transfer, or physical access). Do **not** send it over plain email or chat.

```
# client1.conf — the config file given to the client device

[Interface]
Address    = 10.0.0.2/32           # this client's VPN IP
PrivateKey = Y2nL7rH8T4wX3vK9m...  # client1_private.key
DNS        = 10.0.0.1               # use the VPN server as DNS resolver (optior

[Peer]
PublicKey    = P3mK9vX2nL7rH8T4...  # server_public.key
```

```
PresharedKey = R5mT9vX2nL7rH8P4...   # client1_psk.key (if used)
Endpoint     = your.server.ip:51820 # server's public IP or hostname
AllowedIPs   = 0.0.0.0/0            # route ALL traffic through VPN (full tunnel
# AllowedIPs = 10.0.0.0/24          # OR: route only VPN subnet (split tunnel)
PersistentKeepalive = 25            # keep tunnel alive through NAT (send keepal
```

> **AllowedIPs controls what gets routed through the tunnel.** `0.0.0.0/0` is full-tunnel mode — all internet traffic goes through the VPN server. `10.0.0.0/24` is split-tunnel mode — only traffic destined for other VPN peers goes through the tunnel, and regular internet traffic continues directly. Chapter 8 covers split tunnelling in detail.

Generating a QR code for mobile clients

server — QR code for phone import

```
root@server:~# apt install -y qrencode root@server:~# qrencode -t ansiutf8 <
/etc/wireguard/client1.conf [QR code appears in terminal — scan with WireGuard
iOS/Android app]
```

Verifying the Connection

Once a client connects, `wg show` on the server reveals the handshake time and traffic counters — the definitive proof that the tunnel is working:

server — verify peer connection

```
root@server:~# wg show interface: wg0 public key:
P3mK9vX2nL7rH8T4wY1cB6sF5nJ0dA8eI2oU9gQ= private key: (hidden) listening port:
51820 peer: X7nK2vP9mL3rH8T4wY1cB6sF5nJ0dA8eI2oU9gR= preshared key: (hidden)
endpoint: 82.45.110.23:54321 allowed ips: 10.0.0.2/32 latest handshake: 14
seconds ago transfer: 1.44 MiB received, 248 KiB sent # ping the client from the
server — confirms two-way tunnel root@server:~# ping -c 3 10.0.0.2 PING 10.0.0.2:
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=14.2 ms # from the client — verify
all traffic routes through VPN (full tunnel) # your IP should now show as the VPN
server's public IP user@client:~$ curl ifconfig.me 203.0.113.42 ← should match
your server's public IP
```

Check	Command	Expected result
Interface up	`ip link show wg0`	state UP
Peer handshake	`wg show`	latest handshake: N seconds ago
Ping server from client	`ping 10.0.0.1`	replies received
Ping client from server	`ping 10.0.0.2`	replies received
Public IP changed (full tunnel)	`curl ifconfig.me`	shows server's public IP
Service running on boot	`systemctl is-enabled wg-quick@wg0`	enabled

Adding More Clients

server — add a second client without restarting

```
# generate keys for the new client root@server:/etc/wireguard# wg genkey > client2_private.key root@server:/etc/wireguard# wg pubkey < client2_private.key > client2_public.key # add the new peer to the RUNNING interface (no restart needed) root@server:~# wg set wg0 peer $(cat /etc/wireguard/client2_public.key) \ allowed-ips 10.0.0.3/32 # save the running config back to wg0.conf so it survives a reboot root@server:~# wg-quick save wg0
```

> **`wg set`** + **`wg-quick save`** lets you add peers to a live WireGuard interface without dropping existing connections. The `save` command writes the current in-memory state back to `wg0.conf`, keeping the file and the running interface in sync.

Quick Reference — wg Commands

```
# show all interface and peer status
wg show

# bring interface up / down
wg-quick up wg0
```

```
wg-quick down wg0

# reload config after editing wg0.conf (drops and re-adds peers)
wg-quick down wg0 && wg-quick up wg0

# add a peer to a running interface
wg set wg0 peer <PUBLIC_KEY> allowed-ips 10.0.0.X/32

# remove a peer from a running interface
wg set wg0 peer <PUBLIC_KEY> remove

# save running state back to config file
wg-quick save wg0

# generate a new private key
wg genkey

# derive public key from private key
wg pubkey < private.key

# generate a preshared key
wg genpsk
```

Next — Chapter 4: WireGuard Clients. The server is running — now connect to it. Chapter 4 covers WireGuard clients on Windows, macOS, iOS, and Android: installing the app, importing the config file (or scanning the QR code), enabling the kill switch, and verifying the tunnel from each platform.

Chapter 4 — WireGuard Clients

The server is running. Now connect to it. WireGuard has official client apps for Windows, macOS, iOS, and Android — all free, all using the same config file format you wrote in Chapter 3. This chapter walks through installing each one, importing a config, enabling the kill switch, and verifying the tunnel is working.

The Client Config — Quick Recap

Every client needs its own `.conf` file generated on the server in Chapter 3. It looks like this — make sure you have it ready before starting any of the platform sections below:

```
[Interface]
Address    = 10.0.0.2/32           # this client's VPN IP
PrivateKey = Y2nL7rH8T4wX3vK9m...  # client's private key
DNS        = 10.0.0.1               # optional — use server as DNS

[Peer]
PublicKey           = P3mK9vX2nL7rH8T4...  # server public key
Endpoint            = your.server.ip:51820
AllowedIPs          = 0.0.0.0/0            # full tunnel
PersistentKeepalive = 25
```

Transfer the config file securely. It contains the client's private key. Use a QR code displayed on-screen (Chapter 3), copy it over SSH/SCP, or use an encrypted password manager. Never send it over plain email, WhatsApp, or Slack.

Windows

Windows 10 / 11

Official WireGuard app — wireguard.com/install · also available via winget

1. **Install the app.** Download from `wireguard.com/install` or run `winget install WireGuard.WireGuard` in PowerShell. The installer adds a WireGuard system service and a tray icon.
2. **Import the config.** Open WireGuard → click the arrow next to *Add Tunnel* → *Import tunnel(s) from file* → select your `client1.conf` file. The tunnel appears in the list with its name taken from the filename.
3. **Activate.** Click the tunnel name → click *Activate*. The status changes to **Active** and a green indicator appears in the system tray.
4. **Enable the kill switch** (recommended). Click *Edit* on the tunnel → tick **Block all traffic when tunnel is not active (kill switch)**. This prevents any traffic leaving your machine if the VPN drops.

Command-line alternative (no GUI)

Windows PowerShell — WireGuard CLI

```
# copy config to the WireGuard directory PS> Copy-Item client1.conf "C:\Program Files\WireGuard\Data\Configurations\" # install and start the tunnel as a Windows service PS> & "C:\Program Files\WireGuard\wireguard.exe" /installtunnelservice "C:\Program Files\WireGuard\Data\Configurations\client1.conf" # start / stop via sc (service control) PS> sc.exe start WireGuardTunnel$client1 PS> sc.exe stop WireGuardTunnel$client1
```

macOS

macOS 12 Monterey and later

Mac App Store · also available via Homebrew

1. **Install.** Search *WireGuard* in the Mac App Store, or run `brew install wireguard-tools` for the command-line version (no GUI, same as Linux).
2. **Import.** Open WireGuard → click the **+** button → *Import tunnel(s) from file* → select your `.conf` file. macOS will ask for permission to add VPN configurations — click *Allow*.
3. **Activate.** Toggle the tunnel on. Status shows in the menu bar icon.
4. **On-Demand (optional).** Click *Edit* → enable **On-Demand** → set rules, e.g. activate automatically when *Wi-Fi* is connected but not when on a trusted SSID. Useful for always-on protection on untrusted networks.

Homebrew / command-line (wg-quick)

macOS terminal — wg-quick

```
# install wireguard-tools (includes wg-quick) philip@mac:~$ brew install
wireguard-tools # copy config and bring tunnel up philip@mac:~$ sudo cp
client1.conf /etc/wireguard/wg0.conf philip@mac:~$ sudo wg-quick up wg0
philip@mac:~$ sudo wg-quick down wg0
```

iOS

iPhone / iPad — iOS 15+

App Store — free · deepest integration via QR code import

1. **Install WireGuard** from the App Store.
2. **Import via QR code** (easiest). On the server, run `qrencode -t ansiutf8 < /etc/wireguard/client1.conf` → hold your phone up to the screen → tap *Allow* to add the VPN profile.
3. **Or import a file.** Transfer the `.conf` file to your phone via AirDrop, iCloud Drive, or the Files app → open it → iOS offers to import it into WireGuard.
4. **Activate.** Tap the toggle next to the tunnel name. iOS adds a VPN indicator to the status bar.
5. **On-Demand** (optional). Tap the tunnel info icon → *On-Demand Activation* → choose Wi-Fi, cellular, or both. Useful for automatic activation on all networks except your home Wi-Fi.

> **Kill switch on iOS:** iOS has no explicit kill switch toggle in the WireGuard app, but when *On-Demand* is enabled iOS re-establishes the tunnel automatically if it drops — effectively achieving the same result. For a true kill switch, enable On-Demand on all interfaces with no exclusions.

Android

Android 7+ (API 24+)

Google Play Store · also F-Droid for open-source build

1. **Install WireGuard** from the Play Store (or F-Droid).
2. **Import via QR code.** Tap the **+** button → *Scan from QR code* → scan the code generated by `qrencode` on the server. Name the tunnel and tap *Create Tunnel*.
3. **Or import a file.** Tap **+** → *Import from file or archive* → navigate to the `.conf` file on the device.
4. **Activate.** Toggle the tunnel on. Android will prompt you to approve the VPN connection on first use.

5 **Enable the kill switch.** Go to Android *Settings → Network → VPN* → tap the gear icon next to the WireGuard VPN → enable **Always-on VPN** and **Block connections without VPN**. This is Android's system-level kill switch — more reliable than any in-app toggle.

> **Android kill switch is system-level.** Unlike the in-app toggles on Windows and iOS, Android's "Block connections without VPN" is enforced by the OS network stack — no traffic leaves the device on any interface if the VPN is down, regardless of which app is trying to send it.

Kill Switch Summary — Platform Comparison

Windows	macOS	iOS	Android
Built-in	Limited	Via On-Demand	System-level
Tick "Block all traffic when tunnel is not active" in the tunnel Edit dialog	No kill switch in the GUI app. Use On-Demand rules or a firewall rule via `pf`	Enable On-Demand on all interfaces with no exclusions — iOS reconnects automatically	Settings → Network → VPN → Always-on VPN + Block connections without VPN

Verifying the Connection on Any Platform

Test	How to run it	Expected result
Public IP changed	Visit `ifconfig.me` or `ipinfo.io`	Shows your VPN server's IP, not your real IP
Ping VPN server	`ping 10.0.0.1`	Replies with low latency
DNS not leaking	Visit `dnsleaktest.com` → Extended test	Only your VPN server or its upstream DNS visible
WebRTC leak	Visit `browserleaks.com/webrtc`	No local IP exposed (or only VPN IP visible)
Server sees handshake	`wg show`	Peer shows "latest handshake: N seconds ago"

Common Client Problems

Tunnel activates but no internet

IP forwarding not enabled on the server, or the iptables MASQUERADE rule is missing/wrong interface name.

Fix: On the server: `cat /proc/sys/net/ipv4/ip_forward` should return 1. Check `iptables -t nat -L` for the POSTROUTING rule. Verify the interface name in PostUp matches your actual external interface.

Handshake never completes

UDP port 51820 is blocked by a firewall between client and server, or the Endpoint IP/port in the client config is wrong.

Fix: Confirm the server's public IP in `Endpoint`. Test UDP reachability: `nc -u your.server.ip 51820`. Check `ufw status` on the server. Try a different network (mobile data vs Wi-Fi) to isolate the blocking firewall.

DNS leaking despite DNS setting

The `DNS = 10.0.0.1` line sets the resolver but the OS may still use its default DNS for non-tunnelled queries, or `AllowedIPs` doesn't include the DNS server's IP.

Fix: Ensure `AllowedIPs = 0.0.0.0/0` (full tunnel). On Windows, check that the WireGuard interface's DNS is listed first in network adapter settings. Run `dnsleaktest.com` to confirm.

Tunnel drops on mobile when screen locks

Mobile OS aggressively kills background network connections to save battery.

Fix: Ensure `PersistentKeepalive = 25` is set in the client config — this sends a UDP keepalive every 25 seconds to prevent NAT tables from timing out. Enable On-Demand on iOS for automatic reconnection.

Wrong key — handshake rejected

The public key in the server's `[Peer]` block doesn't match the client's actual private key, or keys were copy-pasted with extra whitespace.

Fix: On the client, run `wg show` and note the public key shown. Compare it to what's in the server's `wg0.conf` `[Peer]` block. Regenerate and re-import the config if they don't match.

Can reach VPN server but not LAN devices

You're in split-tunnel mode (`AllowedIPs = 10.0.0.0/24`) and LAN devices have a different subnet, or the server isn't routing to the LAN.

Fix: Add the LAN subnet to `AllowedIPs` on the client, e.g. `10.0.0.0/24, 192.168.0.0/24`. Ensure the server has a route to the LAN (it does if it's on the LAN; it won't if it's a remote VPS).

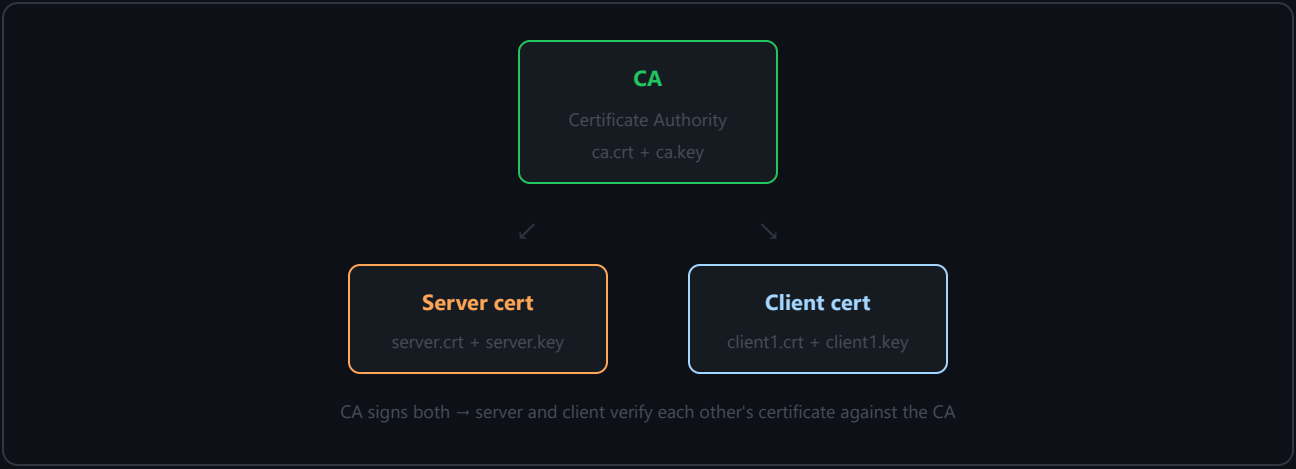
Next — Chapter 5: OpenVPN Server. WireGuard is the right choice for most new deployments, but some networks block UDP entirely. Chapter 5 covers installing OpenVPN on Linux, building a PKI with Easy-RSA, writing the server config, and issuing client certificates — giving you a VPN that can tunnel over TCP port 443 and pass through almost any firewall.

Chapter 5 — OpenVPN Server

WireGuard's one weakness is that it is UDP-only. Some corporate networks, hotel captive portals, and restrictive ISPs block all UDP traffic except DNS. OpenVPN's killer feature is the ability to run over **TCP port 443** — identical to HTTPS traffic — which passes through virtually every firewall on the planet. This chapter builds a complete OpenVPN server from scratch.

How OpenVPN Authentication Works — PKI

Unlike WireGuard's simple key pairs, OpenVPN uses a **Public Key Infrastructure (PKI)** — a Certificate Authority that signs certificates for both the server and each client. The server and clients trust each other because they both hold certificates signed by the same CA.



Easy-RSA is the tool bundled with OpenVPN for managing this PKI. It handles the CA creation, certificate signing, and revocation list — without needing a deep understanding of raw `openssl` commands.

Step 1 — Install OpenVPN and Easy-RSA

```
root@server:~# apt update && apt install -y openvpn easy-rsa # confirm versions
root@server:~# openvpn --version | head -1 OpenVPN 2.6.3 x86_64-pc-linux-gnu
root@server:~# /usr/share/easy-rsa/easyrsa --version EasyRSA Version: 3.1.7
```

Step 2 — Build the PKI and Certificate Authority

server — create PKI and CA

```
# create a working directory for the PKI root@server:~# make-cadir ~/openvpn-pki
&& cd ~/openvpn-pki # initialise the PKI structure root@server:~/openvpn-pki#
./easyrsa init-pki init-pki complete; you may now create a CA or requests. #
build the CA — you'll be asked for a CA name (e.g. "HomeVPN")
root@server:~/openvpn-pki# ./easyrsa build-ca nopass CA creation complete. Your
new CA certificate file is at: pki/ca.crt
```

> **nopass** skips the CA key passphrase — convenient for automated setups but means anyone who gets `ca.key` can sign arbitrary certificates. For a production setup, omit `nopass` and store the CA passphrase securely in a password manager.

Step 3 — Generate the Server Certificate

server — server certificate + DH params + TLS auth key

```
# generate and sign the server certificate (name it "server")
root@server:~/openvpn-pki# ./easyrsa gen-req server nopass root@server:~/openvpn-
pki# ./easyrsa sign-req server server Certificate created at:
pki/issued/server.crt # generate Diffie-Hellman parameters (takes a minute)
root@server:~/openvpn-pki# ./easyrsa gen-dh DH parameters of size 2048 created at
pki/dh.pem # generate a TLS authentication key — extra HMAC layer against
DoS/port scanning root@server:~/openvpn-pki# openvpn --genkey secret pki/ta.key #
copy everything the server needs to /etc/openvpn/server/ root@server:~/openvpn-
pki# cp pki/ca.crt pki/issued/server.crt pki/private/server.key \ pki/dh.pem
pki/ta.key /etc/openvpn/server/
```

Step 4 — Generate a Client Certificate

server — client certificate (repeat for each client)

```
# generate client key and certificate request root@server:~/openvpn-pki#
./easyrsa gen-req client1 nopass # sign it with the CA root@server:~/openvpn-pki#
./easyrsa sign-req client client1 Certificate created at: pki/issued/client1.crt
# the files needed for the client .ovpn profile: # pki/ca.crt — CA certificate
(verify the server) # pki/issued/client1.crt — client certificate #
pki/private/client1.key — client private key (KEEP SECRET) # pki/ta.key — TLS
auth key
```

Step 5 — Write the Server Config

```
# /etc/openvpn/server/server.conf

# ── Network ──────────────────────────────────────────────
port    1194          # change to 443 for firewall bypass (requires proto tcp)
proto   udp           # change to tcp for port 443 mode
dev     tun

# ── Certificates ─────────────────────────────────────────
ca      /etc/openvpn/server/ca.crt
cert    /etc/openvpn/server/server.crt
key     /etc/openvpn/server/server.key
dh      /etc/openvpn/server/dh.pem
tls-auth /etc/openvpn/server/ta.key 0   # 0 = server side

# ── VPN subnet ───────────────────────────────────────────
server 10.8.0.0 255.255.255.0       # assigns 10.8.0.1 to server, clients get 10
ifconfig-pool-persist /var/log/openvpn/ipp.txt

# ── Routing — push settings to clients ──────────────────
push "redirect-gateway def1 bypass-dhcp"  # full tunnel — all traffic via VPN
push "dhcp-option DNS 8.8.8.8"            # push Google DNS (or use 10.8.0.1)
push "dhcp-option DNS 8.8.4.4"

# ── Security ─────────────────────────────────────────────
cipher       AES-256-GCM
auth         SHA256
tls-version-min 1.2
tls-cipher   TLS-ECDHE-RSA-WITH-AES-256-GCM-SHA384
key-direction 0
```

```
# ── Hardening ──────────────────────────────────────────
user  nobody
group nogroup
persist-key
persist-tun

# ── Performance & reliability ──────────────────────────
keepalive 10 120
compress  lz4-v2
push "compress lz4-v2"
max-clients 10

# ── Logging ────────────────────────────────────────────
status      /var/log/openvpn/status.log
log-append  /var/log/openvpn/openvpn.log
verb        3
```

> **Firewall bypass mode:** to run on TCP 443, change `port 1194` → `port 443` and `proto udp` → `proto tcp`. Note that TCP-over-TCP (tunnelling TCP application traffic inside a TCP tunnel) causes performance issues on lossy connections — prefer UDP 1194 when firewalls allow it.

Step 6 — Enable IP Forwarding and Start OpenVPN

server — firewall, IP forwarding, start service

```
# IP forwarding (same as WireGuard — skip if already done) root@server:~# echo
"net.ipv4.ip_forward=1" >> /etc/sysctl.conf && sysctl -p # iptables NAT rule —
replace eth0 with your external interface root@server:~# iptables -t nat -A
POSTROUTING -s 10.8.0.0/24 -o eth0 -j MASQUERADE # make the iptables rule
persistent across reboots root@server:~# apt install -y iptables-persistent &&
netfilter-persistent save # open the firewall root@server:~# ufw allow 1194/udp #
or for TCP 443 mode: root@server:~# ufw allow 443/tcp # create log directory
root@server:~# mkdir -p /var/log/openvpn # start and enable OpenVPN
root@server:~# systemctl enable --now openvpn-server@server root@server:~#
systemctl status openvpn-server@server ● openvpn-server@server.service - OpenVPN
service for server Active: active (running) since Thu 2026-06-18 11:05:44 BST
```

Step 7 — Create the Client .ovpn Profile

The cleanest approach is an **inline** `.ovpn` file — all certificates and keys embedded in the single file, so the client only needs one file to import:

```
# client1.ovpn — all-in-one inline profile

client
dev tun
proto udp                          # match server (udp or tcp)
remote your.server.ip 1194         # server public IP and port
resolv-retry infinite
nobind
persist-key
persist-tun

remote-cert-tls server             # verify server cert has server flag
cipher  AES-256-GCM
auth    SHA256
key-direction 1                    # 1 = client side (server uses 0)
verb    3

<ca>
-----BEGIN CERTIFICATE-----
...paste contents of ca.crt here...
-----END CERTIFICATE-----
</ca>

<cert>
-----BEGIN CERTIFICATE-----
...paste contents of client1.crt here...
-----END CERTIFICATE-----
</cert>

<key>
-----BEGIN PRIVATE KEY-----
...paste contents of client1.key here...
-----END PRIVATE KEY-----
</key>

<tls-auth>
```

```
-----BEGIN OpenVPN Static key V1-----
...paste contents of ta.key here...
-----END OpenVPN Static key V1-----
</tls-auth>
```

Automating .ovpn generation with a script

```
#!/bin/bash
# generate-client.sh — usage: ./generate-client.sh client1

CLIENT=$1
PKI=~/openvpn-pki
SERVER_IP=your.server.ip

cat > ~/${CLIENT}.ovpn <<EOF
client
dev tun
proto udp
remote ${SERVER_IP} 1194
resolv-retry infinite
nobind
persist-key
persist-tun
remote-cert-tls server
cipher AES-256-GCM
auth SHA256
key-direction 1
verb 3
<ca>
$(cat ${PKI}/pki/ca.crt)
</ca>
<cert>
$(cat ${PKI}/pki/issued/${CLIENT}.crt)
</cert>
<key>
$(cat ${PKI}/pki/private/${CLIENT}.key)
</key>
<tls-auth>
$(cat ${PKI}/pki/ta.key)
</tls-auth>
EOF
echo "Profile written to ~/${CLIENT}.ovpn"
```

PKI File Reference

File	Goes to	Purpose
`pki/ca.crt`	Server + Client	CA certificate — both sides verify each other's cert against this
`pki/issued/server.crt`	Server only	Server's identity certificate, signed by the CA
`pki/private/server.key`	Server only — never share	Server's private key
`pki/dh.pem`	Server only	Diffie-Hellman parameters for key exchange
`pki/ta.key`	Server + Client	TLS-auth HMAC key — shared secret, direction 0 (server) / 1 (client)
`pki/issued/client1.crt`	Client only	Client's identity certificate, embedded in the .ovpn profile
`pki/private/client1.key`	Client only — never share	Client's private key, embedded in the .ovpn profile

Revoking a Client Certificate

One advantage of PKI over WireGuard's key model: you can revoke a specific client's access without affecting any other client or regenerating the CA:

server — revoke a client certificate

```
# revoke client1's certificate root@server:~/openvpn-pki# ./easyrsa revoke client1 # regenerate the Certificate Revocation List root@server:~/openvpn-pki# ./easyrsa gen-crl root@server:~/openvpn-pki# cp pki/crl.pem /etc/openvpn/server/ # add this line to server.conf if not already present, then restart root@server:~# echo "crl-verify /etc/openvpn/server/crl.pem" >> /etc/openvpn/server/server.conf root@server:~# systemctl restart openvpn-server@server # client1 can no longer connect — their certificate is now on the revocation list
```

> **WireGuard vs OpenVPN for revocation:** in WireGuard you revoke access by removing the peer's `[Peer]` block from `wg0.conf` and running `wg set wg0 peer <key> remove`.

> OpenVPN's CRL approach is more formal but serves the same purpose. Neither is inherently better — it depends on whether you prefer file-based or PKI-based access management.

Next — Chapter 6: OpenVPN Clients. The server is running and the `.ovpn` profile is ready. Chapter 6 covers connecting from Linux (`openvpn --config` and NetworkManager), the Windows GUI client, and the anatomy of the `.ovpn` file so you can customise it confidently.

Chapter 6 — OpenVPN Clients

The server is running and the `.ovpn` profile is ready. This chapter covers connecting from three environments: Linux via the command line and NetworkManager, Windows via the official GUI client, and a close look at the `.ovpn` file itself so you can customise it with confidence.

Linux — Command Line

Linux — openvpn CLI

Works on any distro · runs in foreground or as a systemd service

linux client — connect with openvpn

```
# install the client philip@laptop:~$ sudo apt install -y openvpn # connect — runs in foreground, Ctrl+C to disconnect philip@laptop:~$ sudo openvpn --config client1.ovpn ... Initialization Sequence Completed ← tunnel is up when you see this # connect in background (daemon mode) philip@laptop:~$ sudo openvpn --config client1.ovpn --daemon --log /var/log/openvpn-client.log # verify the tunnel interface exists philip@laptop:~$ ip addr show tun0 4: tun0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 inet 10.8.0.2/24 brd 10.8.0.255 scope global tun0 # confirm all traffic routes through the VPN philip@laptop:~$ curl ifconfig.me 203.0.113.42 ← your server's public IP # disconnect (daemon mode) philip@laptop:~$ sudo pkill openvpn
```

Running as a systemd service (always-on client)

linux client — systemd service

```
# copy the profile to /etc/openvpn/client/ philip@laptop:~$ sudo cp client1.ovpn /etc/openvpn/client/client1.conf # the service name is derived from the filename: client1.conf → openvpn-client@client1 philip@laptop:~$ sudo systemctl enable --now openvpn-client@client1 philip@laptop:~$ sudo systemctl status openvpn-client@client1 ● openvpn-client@client1.service - OpenVPN tunnel for client1
```

```
Active: active (running) # view live connection log philip@laptop:~$ sudo
journalctl -u openvpn-client@client1 -f
```

NetworkManager GUI integration

linux — import .ovpn into NetworkManager

```
# install the NetworkManager OpenVPN plugin philip@laptop:~$ sudo apt install -y
network-manager-openvpn-gnome # GNOME: Settings → Network → VPN → + → Import from
file → select client1.ovpn # XFCE/KDE: use nm-connection-editor or the network
applet # or import via nmcli (no GUI needed) philip@laptop:~$ nmcli connection
import type openvpn file client1.ovpn Connection 'client1' (uuid) successfully
added. # connect / disconnect via nmcli philip@laptop:~$ nmcli connection up
client1 philip@laptop:~$ nmcli connection down client1
```

> **nmcli vs openvpn --config:** NetworkManager is better for laptops that roam between networks — it handles reconnection automatically and integrates with the desktop's network switcher. The raw `openvpn --config` approach is better for servers or scripts where you want explicit control.

Windows — OpenVPN GUI Client

Windows 10 / 11 — OpenVPN Connect

openvpn.net/client · also available via winget

1. **Install OpenVPN Connect** from `openvpn.net/client` or run `winget install OpenVPNTechnologies.OpenVPNConnect`. This is the official client — not the older "OpenVPN GUI" which is community-maintained and less polished.
2. **Import the profile.** Open OpenVPN Connect → click *Upload File* → select your `client1.ovpn`. The profile appears in the list.
3. **Connect.** Click the toggle next to the profile name. Windows may prompt for admin approval on first connect — this is normal (OpenVPN needs to add a route to the routing table).
4. **Verify.** The tray icon turns green. Open a browser and check `ifconfig.me` — it should show your server's IP.

> **Alternative — OpenVPN GUI (community):** the older `openvpn.net/community-downloads` package installs a lighter tray-icon app. Copy `client1.ovpn` to `C:\Program Files\OpenVPN\config\`, right-click the tray icon → *Connect*. Useful if you prefer a simpler interface or need to run multiple profiles simultaneously.

Windows CLI — for scripts and automation

Windows PowerShell — openvpn CLI

```
# run openvpn from PowerShell (as Administrator) PS> & "C:\Program Files\OpenVPN\bin\openvpn.exe" --config client1.ovpn # check the VPN interface appeared PS> Get-NetAdapter | Where-Object { $_.InterfaceDescription -like "*TAP*" -or $_.InterfaceDescription -like "*Wintun*" }
```

The .ovpn File — Full Anatomy

Understanding every directive in the `.ovpn` file lets you adapt it for split tunnelling, different ports, or stricter security without guessing.

Directive	What it does
`client`	Declares this is a client config (implies `pull` — accept pushed settings from server)
`dev tun`	**tun** = layer-3 tunnel (IP routing). Use `tap` only for layer-2 bridging (rare)
`proto udp`	Transport protocol. Change to `tcp` for port 443 firewall bypass
`remote 203.0.113.42 1194`	Server IP (or hostname) and port. Can repeat for multiple servers — client tries each in turn
`resolv-retry infinite`	Keep trying to resolve the hostname if DNS fails on startup — important for dynamic IP servers
`nobind`	Don't bind to a specific local port — use any available ephemeral port

`persist-key` `persist-tun`	Keep the key and tun device across restarts (required when running as non-root after dropping privileges)
`remote-cert-tls server`	**Important security setting.** Verifies the server cert has the server flag — prevents a client cert from being used to impersonate the server
`cipher AES-256-GCM`	Data channel cipher. Must match (or be negotiable with) the server. AES-256-GCM is the recommended modern choice
`auth SHA256`	HMAC digest for packet authentication. SHA256 is the minimum; SHA512 is stronger
`key-direction 1`	TLS-auth key direction. Server uses 0, client uses 1 — they use different halves of the ta.key
`verb 3`	Log verbosity. 3 = normal. Increase to 6 for debugging connection issues; 0 for silent
`<ca>...</ca>`	Inline CA certificate — used to verify the server's identity
`<cert>...</cert>`	Inline client certificate — proves this client's identity to the server
`<key>...</key>`	Inline client private key — **keep this file secret**
`<tls-auth>...</tls-auth>`	Inline TLS-auth HMAC key — shared secret between all clients and the server

Common customisations

```
# Split tunnel — only route VPN subnet through tunnel, NOT all internet traffic
# Replace the "redirect-gateway" pushed by the server with a specific route
route-nopull                          # ignore all routes pushed by server
route 10.8.0.0 255.255.255.0          # add only the VPN subnet route manually

# Connect to a different server if the first is unreachable
remote server1.example.com 1194
remote server2.example.com 1194     # tried if server1 fails
remote-random                         # pick a random server from the list

# Reconnect automatically on failure
```

```
connect-retry 5                    # retry every 5 seconds
connect-retry-max 10               # give up after 10 attempts

# Use a specific DNS server on the client (overrides server-pushed DNS)
dhcp-option DNS 1.1.1.1

# Increase log verbosity for debugging
verb 6
```

Verifying the OpenVPN Connection

linux client — connection checks

```
# tunnel interface should be up philip@laptop:~$ ip addr show tun0 inet
10.8.0.2/24 scope global tun0 # default route should go through tun0 (full
tunnel) philip@laptop:~$ ip route | grep default default via 10.8.0.1 dev tun0 #
public IP should be server's IP philip@laptop:~$ curl -s ifconfig.me 203.0.113.42
# check the server log to confirm this client connected root@server:~# grep
"client1" /var/log/openvpn/openvpn.log | tail -5 client1/82.45.110.23:54123
MULTI: Learn: 10.8.0.2 -> client1/82.45.110.23:54123 client1/82.45.110.23:54123
Initialization Sequence Completed
```

Common Connection Problems

TLS handshake failed

Certificate mismatch, wrong `key-direction`, or `remote-cert-tls` failing because the server cert wasn't signed with the server flag.

Fix: Check client has `key-direction 1` (server has 0). Confirm server cert was generated with `sign-req server` not `sign-req client`. Set `verb 6` for detailed TLS error output.

AUTH_FAILED — certificate revoked

This client's certificate is on the server's CRL (Certificate Revocation List).

Fix: Issue a new certificate with `easyrsa gen-req` + `sign-req client` and distribute a new `.ovpn` profile.

Connects but no internet (full tunnel)

Server's iptables MASQUERADE rule is missing or targets the wrong interface, or IP forwarding is disabled.

Cannot connect on corporate Wi-Fi

UDP port 1194 is blocked. The whole point of TCP 443 mode.

Fix: Change `proto udp` → `proto tcp` and `remote ... 1194` → `remote ... 443` in the

Fix: On server: `sysctl net.ipv4.ip_forward` should be 1. Check `iptables -t nat -L POSTROUTING` for the MASQUERADE rule. Confirm the interface name matches.

`.ovpn` file. Update the server config and UFW rule to match.

Next — Chapter 7: Commercial VPN Clients. Self-hosted VPNs give you full control but your public IP is still your home or VPS IP. Chapter 7 covers installing ProtonVPN and Mullvad on Linux via their CLI clients, using them from the terminal, and running DNS leak tests to verify they're working correctly.

Chapter 7 — Commercial VPN Clients on Linux

The previous four chapters covered self-hosted VPNs — you own the server, you control the traffic, and your public IP is your own server's IP. Commercial VPN services flip that model: a company runs thousands of servers worldwide, and you connect to whichever country you need. Your traffic exits from a shared IP pool, making individual attribution harder.

This chapter covers installing and using two of the most privacy-focused commercial providers — **ProtonVPN** and **Mullvad** — from the Linux command line, and how to verify that neither is leaking your real DNS queries or IP address.

ProtonVPN vs Mullvad — Key Differences

ProtonVPN

Swiss jurisdiction · open-source · free tier available

Built by the team behind ProtonMail. Strong transparency record — published independent audits and open-sourced all clients. The free tier is genuinely usable (no data cap, 3 countries) making it the best free option in the market. Paid plans add Secure Core (routing through privacy-friendly countries before exiting), NetShield (DNS-based ad/malware blocking), and P2P servers.

- **Jurisdiction:** Switzerland (not EU/US)
- **Logging:** No-logs, independently audited
- **Protocols:** WireGuard, OpenVPN, Stealth (obfuscated)
- **Free tier:** Yes — 3 countries, unlimited data
- **Linux CLI:** Official `proton-vpn-gtk-app` + headless mode

Mullvad

Swedish jurisdiction · account-number only · flat €5/month

Mullvad takes anonymity further than almost any other provider: accounts are random numbers with no email required, you can pay in cash by post, and they have resisted and publicised law enforcement attempts to hand over user data. The flat €5/month pricing (no annual discounts, no tiers) keeps things simple. The Linux app is a first-class CLI tool, not an afterthought.

- **Jurisdiction:** Sweden
- **Logging:** No-logs, independently audited
- **Protocols:** WireGuard, OpenVPN
- **Free tier:** No — €5/month flat
- **Linux CLI:** Official `mullvad` CLI — excellent
- **Accepts:** Cash, Monero, Bitcoin, cards

Accepts: Cash, crypto, cards

> **The trust question doesn't go away.** A commercial VPN moves trust from your ISP to the VPN provider. "No-logs" claims can only be partially verified through audits — you are ultimately trusting the provider's honesty and jurisdiction. If true anonymity is the goal, a VPN alone is not sufficient regardless of provider.

ProtonVPN on Linux

Install

debian/ubuntu — install ProtonVPN

```
# install dependencies philip@laptop:~$ sudo apt install -y wget gnupg # add ProtonVPN repository philip@laptop:~$ wget -q -O- https://repo.protonvpn.com/debian/public_key.asc | \ sudo gpg --dearmor -o /usr/share/keyrings/protonvpn.gpg philip@laptop:~$ echo "deb [signed-by=/usr/share/keyrings/protonvpn.gpg] \ https://repo.protonvpn.com/debian stable main" | \ sudo tee /etc/apt/sources.list.d/protonvpn.list philip@laptop:~$ sudo apt update && sudo apt install -y proton-vpn-gtk-app
```

CLI usage (non-interactive / headless)

protonvpn — CLI commands

```
# login (stores credentials locally) philip@laptop:~$ protonvpn-cli login your@email.com # connect to the fastest available server philip@laptop:~$ protonvpn-cli connect --fastest Connecting to NL#47 via WireGuard... Connected! # connect to a specific country philip@laptop:~$ protonvpn-cli connect --cc CH # Switzerland philip@laptop:~$ protonvpn-cli connect --cc DE # Germany philip@laptop:~$ protonvpn-cli connect --cc US # United States # connect using a specific protocol philip@laptop:~$ protonvpn-cli connect --fastest --protocol wireguard philip@laptop:~$ protonvpn-cli connect --fastest --protocol openvpn-tcp # show connection status philip@laptop:~$ protonvpn-cli status Connected to NL#47 Protocol: WireGuard Time: 00:04:31 IP: 185.159.157.47 Country: Netherlands Server load: 34% # disconnect philip@laptop:~$ protonvpn-cli disconnect # enable kill
```

```
switch (blocks internet if VPN drops) philip@laptop:~$ protonvpn-cli killswitch --on philip@laptop:~$ protonvpn-cli killswitch --off
```

Mullvad on Linux

Install

debian/ubuntu — install Mullvad

```
# add Mullvad repository philip@laptop:~$ curl -fsSL https://repository.mullvad.net/deb/mullvad-keyring.asc | \ sudo gpg --dearmor -o /usr/share/keyrings/mullvad-keyring.gpg philip@laptop:~$ echo "deb [signed-by=/usr/share/keyrings/mullvad-keyring.gpg arch=$( dpkg --print-architecture )] \ https://repository.mullvad.net/deb/stable $(lsb_release -cs) main" | \ sudo tee /etc/apt/sources.list.d/mullvad.list philip@laptop:~$ sudo apt update && sudo apt install -y mullvad-vpn
```

CLI usage

mullvad — CLI commands

```
# login with your account number (no email required) philip@laptop:~$ mullvad account login 1234567890123456 Logged in to account 1234567890123456 # check account status and expiry philip@laptop:~$ mullvad account get Account: 1234567890123456 Expires: 2026-12-18 00:00:00 UTC # connect (auto-selects fastest server) philip@laptop:~$ mullvad connect Connecting... # show current status philip@laptop:~$ mullvad status Connected to Amsterdam, Netherlands (nl-ams-wg-201) via WireGuard # set preferred country / city philip@laptop:~$ mullvad relay set location ch # Switzerland philip@laptop:~$ mullvad relay set location se got # Sweden, Gothenburg philip@laptop:~$ mullvad relay set location us dal # US, Dallas # list available countries philip@laptop:~$ mullvad relay list | grep -E "^\w" | head -20 # set protocol philip@laptop:~$ mullvad relay set tunnel-protocol wireguard philip@laptop:~$ mullvad relay set tunnel-protocol openvpn # kill switch — always on by default in Mullvad philip@laptop:~$ mullvad lockdown-mode set on # block all traffic when disconnected philip@laptop:~$ mullvad lockdown-mode set off # disconnect philip@laptop:~$ mullvad disconnect
```

> **Mullvad's lockdown mode** is stricter than a standard kill switch — it blocks all internet traffic whenever the VPN is not connected, not just when the tunnel drops unexpectedly. This means

> even if you manually disconnect, nothing gets through until you reconnect. Enable it if you want a guarantee that unencrypted traffic is impossible on this machine.

DNS Leak Testing — Why It Matters

A DNS leak occurs when your device sends DNS queries outside the VPN tunnel — to your ISP's resolver or your router — despite being connected to a VPN. This reveals your browsing activity to your ISP even while the VPN is active, defeating the purpose of using one.

DNS leaks happen because:

- The OS prefers the system DNS resolver over the VPN-pushed one
- Some browsers use their own DNS-over-HTTPS resolver (bypassing the OS)
- Split-tunnel mode may not route DNS queries through the tunnel
- IPv6 DNS resolvers leak when the VPN only tunnels IPv4

What a clean result looks like

dnsleaktest.com — Extended Test results		✓ No leak detected
DNS Server IP	**ISP / Owner**	**Country**
185.159.157.1	Mullvad	Netherlands
185.159.157.2	Mullvad	Netherlands

What a leaking result looks like

dnsleaktest.com — Extended Test results		✗ Leak detected — your ISP can see your queries
DNS Server IP	**ISP / Owner**	**Country**
185.159.157.1	Mullvad	Netherlands
82.132.240.1	BT (your ISP)	United Kingdom ← LEAK

DNS leak test from the terminal

linux — check DNS resolution path

```
# see which DNS server is currently being used philip@laptop:~$ resolvectl status
| grep "Current DNS" Current DNS Server: 185.159.157.1 ← should be VPN provider's
DNS # quick check — query a hostname and see which server answered
philip@laptop:~$ dig +short TXT whoami.ds.akahelp.net "ns" "185.159.157.1" ← the
resolver that answered # check for IPv6 DNS leak (if IPv6 is active)
philip@laptop:~$ dig -6 +short myip.opendns.com @2606:4700:4700::1111 # confirm
no ISP DNS servers in resolv.conf philip@laptop:~$ cat /etc/resolv.conf
nameserver 185.159.157.1 ← VPN DNS only — no leak nameserver 185.159.157.2
```

Full Verification Checklist

Public IP check

Confirm your IP shows as the VPN server's IP, not your real IP

`ipinfo.io · ifconfig.me · ipify.org`

DNS leak test — standard

Run the standard test: all DNS servers shown should belong to the VPN provider

`dnsleaktest.com → Standard Test`

DNS leak test — extended

Runs more queries to catch intermittent leaks. Run this if the standard test passes but you're still suspicious

`dnsleaktest.com → Extended Test`

WebRTC leak test

Browsers can expose your real local IP via WebRTC even through a VPN. Check that no local IPs are revealed

`browserleaks.com/webrtc · ipleak.net`

IPv6 leak test

If your VPN only tunnels IPv4, IPv6 traffic goes unencrypted. Mullvad and ProtonVPN block IPv6 by default when connected

`ipleak.net · ipv6leak.com`

Kill switch test

Enable kill switch, connect to VPN, then manually disconnect. Attempt to load a webpage — it should fail if the kill switch is working

`curl ifconfig.me (should time out or be refused)`

Browser DNS-over-HTTPS can bypass VPN DNS. Chrome and Firefox have built-in DoH resolvers that bypass the OS DNS settings entirely. If your VPN reports no DNS leak but the browser behaves differently, check: Chrome → Settings → Privacy and security → Security → Use secure DNS — disable or point it at your VPN provider's DoH endpoint. Firefox: about:preferences#privacy → DNS over HTTPS.

Next — Chapter 8: Troubleshooting and Security Hardening. The final chapter covers DNS leaks in depth, kill switch implementation, MTU issues (the silent performance killer), logging and audit, and a practical checklist for verifying that your VPN setup — self-hosted or commercial — is actually doing what you think it is.

Chapter 8 — Troubleshooting and Security Hardening

A VPN that appears to be working may still be leaking. This final chapter covers the four most common failure modes — DNS leaks, IPv6 leaks, kill switch gaps, and MTU fragmentation — along with logging, server hardening, and a final checklist that applies to both self-hosted and commercial VPN setups.

DNS Leaks — Finding and Fixing Them

Chapter 7 introduced DNS leaks conceptually. Here we go deeper into the specific causes and their fixes on a self-hosted setup.

Cause 1 — systemd-resolved ignoring VPN DNS

diagnosing systemd-resolved DNS routing

```
# see which DNS is being used per interface philip@laptop:~$ resolvectl status
Global DNS Servers: 82.132.240.1 ← ISP DNS still global! Link 4 (tun0) DNS
Servers: 10.8.0.1 ← VPN DNS only on tun0 # fix — set VPN interface DNS as the
global default philip@laptop:~$ resolvectl dns tun0 10.8.0.1 philip@laptop:~$
resolvectl domain tun0 "~." # "~." means "use this interface for ALL domains" —
makes tun0 the default resolver # permanent fix for OpenVPN — add to server.conf
# push "dhcp-option DNS 10.8.0.1" (already in Chapter 5 config) # and add to
client.ovpn: # script-security 2 # up /etc/openvpn/update-resolv-conf # down
/etc/openvpn/update-resolv-conf
```

Cause 2 — browser DNS-over-HTTPS bypassing VPN

```
# Chrome — disable DoH (or point it at VPN provider's DoH endpoint)
# chrome://settings/security → Use secure DNS → turn off OR set custom
# Mullvad's DoH: https://dns.mullvad.net/dns-query
# ProtonVPN's DoH: https://dns.protonvpn.com/dns-query

# Firefox — about:preferences#privacy → DNS over HTTPS
# Set to "Off" or enter provider's DoH URL in the custom field
```

Cause 3 — IPv6 leaking outside the tunnel

disable IPv6 to prevent leaks (WireGuard / OpenVPN)

```
# check if IPv6 is active philip@laptop:~$ ip -6 addr show | grep -v "::1" inet6
2a00:23c7:a48:8400::1/128 scope global ← IPv6 active, may leak # disable IPv6 for
the session philip@laptop:~$ sudo sysctl -w net.ipv6.conf.all.disable_ipv6=1
philip@laptop:~$ sudo sysctl -w net.ipv6.conf.default.disable_ipv6=1 # make
permanent philip@laptop:~$ echo "net.ipv6.conf.all.disable_ipv6 = 1" | sudo tee -
a /etc/sysctl.conf philip@laptop:~$ echo "net.ipv6.conf.default.disable_ipv6 = 1"
| sudo tee -a /etc/sysctl.conf # WireGuard alternative — tunnel IPv6 too (add to
client config) # AllowedIPs = 0.0.0.0/0, ::/0 ← routes both IPv4 and IPv6 through
VPN
```

Kill Switch — Implementing It Yourself

Commercial clients have kill switches built in. For a self-hosted WireGuard or OpenVPN setup, you implement the kill switch yourself with iptables or nftables — blocking all non-VPN traffic at the firewall level.

WireGuard kill switch via iptables (add to wg0.conf)

```
# /etc/wireguard/wg0.conf — PostUp/PreDown kill switch
# Replace eth0 with your actual external interface

PostUp  = iptables -I OUTPUT ! -o %i -m mark ! --mark $(wg show %i fwmark) \
          -m addrtype ! --dst-type LOCAL -j REJECT; \
          ip6tables -I OUTPUT ! -o %i -m mark ! --mark $(wg show %i fwmark) \
          -m addrtype ! --dst-type LOCAL -j REJECT

PreDown = iptables -D OUTPUT ! -o %i -m mark ! --mark $(wg show %i fwmark) \
          -m addrtype ! --dst-type LOCAL -j REJECT; \
          ip6tables -D OUTPUT ! -o %i -m mark ! --mark $(wg show %i fwmark) \
          -m addrtype ! --dst-type LOCAL -j REJECT
```

> **How this works:** WireGuard marks its own packets with a firewall mark. The iptables rule rejects any OUTPUT packet that is *not* on the WireGuard interface and does not carry WireGuard's mark — meaning all non-VPN traffic is dropped. The rule is added when the tunnel

> comes up (`PostUp`) and removed when it goes down (`PreDown`), so you regain internet access when you manually disconnect.

OpenVPN kill switch via UFW

openvpn kill switch with ufw

```
# allow traffic only on the VPN interface and to/from the VPN server itself
philip@laptop:~$ sudo ufw default deny outgoing philip@laptop:~$ sudo ufw default
deny incoming # allow the VPN server endpoint (replace with your server IP and
port) philip@laptop:~$ sudo ufw allow out to 203.0.113.42 port 1194 proto udp #
allow all traffic on the VPN tunnel interface philip@laptop:~$ sudo ufw allow out
on tun0 philip@laptop:~$ sudo ufw allow in on tun0 # allow DNS on the VPN
interface only philip@laptop:~$ sudo ufw allow out on tun0 to any port 53
philip@laptop:~$ sudo ufw enable # result: no internet without VPN — only the VPN
server and tun0 traffic allowed
```

MTU Issues — The Silent Performance Killer

MTU (Maximum Transmission Unit) fragmentation is one of the most common and least obvious VPN problems. When VPN encapsulation adds overhead to each packet, packets that were exactly at the network's MTU limit become too large and must be fragmented — or dropped entirely if the **Don't Fragment** bit is set.

```
Standard Ethernet frame: 1500 bytes MTU
WireGuard adds overhead: 60 bytes (headers + encryption)
Effective WireGuard payload: 1420 bytes (wg-quick sets this automatically)

OpenVPN UDP overhead: ~50–80 bytes depending on cipher
OpenVPN TCP overhead: even higher — TCP headers on top
Symptom of wrong MTU: large sites work, large downloads stall, SSH hangs on banner
```

finding and fixing the right MTU

```
# test the largest packet size that doesn't fragment (do this with VPN connected)
# Linux — ping with Don't Fragment bit, decrease size until it works
philip@laptop:~$ ping -M do -s 1400 8.8.8.8 PING 8.8.8.8: 1400 data bytes ← if
this works, try larger philip@laptop:~$ ping -M do -s 1450 8.8.8.8 ping: local
```

```
error: message too long, mtu=1420 ← 1420 is the limit # WireGuard — set MTU explicitly in wg0.conf [Interface] if auto-detection fails # MTU = 1420 # OpenVPN — add to server.conf and client.ovpn # tun-mtu 1400 # mssfix 1360 # quick fix on a running interface philip@laptop:~$ sudo ip link set dev tun0 mtu 1400 # verify current MTU philip@laptop:~$ ip link show tun0 | grep mtu 4: tun0: <...> mtu 1400 ...
```

Reading VPN Logs

key log lines to recognise

```
── WireGuard ──────────────────────────────────────── root@server:~# wg show wg0 peer: X7nK2vP9... latest handshake: 14 seconds ago ← connected latest handshake: (none) ← never connected / key mismatch transfer: 1.44 MiB received, 248 KiB sent ← traffic flowing ── OpenVPN server log (/var/log/openvpn/openvpn.log) ────────── Initialization Sequence Completed ← server started OK client1/.../54123 MULTI: Learn: 10.8.0.6 ← client connected, got IP TLS Error: TLS key negotiation failed ← cert/key mismatch or wrong direction AUTH_FAILED ← revoked cert or wrong credentials read UDPv4: Connection refused ← nothing listening on that port ── OpenVPN client log (journalctl or --log file) ──────────── Initialization Sequence Completed ← tunnel up SIGTERM received, sending exit notification ← clean disconnect Connection reset, restarting ← tunnel dropped, reconnecting NOTE: --mute triggered ← too many repeated messages suppressed # increase OpenVPN verbosity for TLS debugging philip@laptop:~$ sudo openvpn --config client1.ovpn --verb 6 2>&1 | grep -i tls
```

Common Problems — Quick Reference

Large file downloads stall or hang

MTU fragmentation — large packets are dropped because the VPN overhead pushes them over the physical MTU limit.

Fix: Test with `ping -M do -s 1400 8.8.8.8` and decrease until it works. Set `MTU = 1420` in WireGuard or `tun-mtu 1400` + `mssfix 1360` in OpenVPN.

VPN connected but DNS still leaks

systemd-resolved is using the system-wide DNS rather than the VPN interface's DNS, or the browser has DoH enabled.

Fix: Run `resolvectl domain tun0 "~."` to make the VPN interface the default resolver. Disable browser DoH or point it at the VPN provider's DoH endpoint.

Tunnel drops every ~2 minutes on mobile

IPv6 sites work but show real location

NAT table on the router or mobile carrier times out idle UDP sessions after 120 seconds.

Fix: Set `PersistentKeepalive = 25` in the WireGuard client config. For OpenVPN add `keepalive 10 60` to the server config.

VPN is only tunnelling IPv4. IPv6 traffic bypasses the tunnel entirely.

Fix: Add `::/0` to WireGuard `AllowedIPs`, or disable IPv6 on the client: `sysctl -w net.ipv6.conf.all.disable_ipv6=1`.

WireGuard handshake times out from one network

UDP port 51820 is blocked by a corporate firewall or ISP.

Fix: Change `ListenPort` to `53` (DNS) or `443` on the server, update UFW and the client's `Endpoint` port. Or switch to OpenVPN on TCP 443.

OpenVPN TLS handshake failed

`key-direction` mismatch (client must be 1, server must be 0), or `remote-cert-tls server` failing because the server cert was signed as a client cert.

Fix: Verify `key-direction 1` in the client `.ovpn`. Confirm the server cert was signed with `sign-req server`, not `sign-req client`. Use `verb 6` for detailed TLS output.

Server Security Hardening Checklist

- [ ] **Change the default port.** WireGuard default 51820 and OpenVPN default 1194 are well-known. Using a non-standard port eliminates automated scan noise. Common alternatives: 443, 53, 4433.
- [ ] **Use key-based SSH only on the VPN server.** Disable password authentication: `PasswordAuthentication no` in `/etc/ssh/sshd_config`. The VPN server is a high-value target — it must not be brute-forceable.
- [ ] **Run fail2ban on SSH.** Even with key-only auth, fail2ban stops log noise and blocks systematic scanning.
- [ ] **Keep WireGuard and OpenVPN updated.** Both have had security patches. Run `apt upgrade wireguard openvpn` regularly and set up unattended-upgrades for security packages.
- [ ] **Restrict firewall to VPN port only.** The server should have SSH (on a non-standard port if possible) and the VPN port open — nothing else. `ufw default deny incoming` with only those two ports allowed.
- [ ] **Rotate client credentials periodically.** WireGuard: regenerate the client key pair and update the server's `[Peer]` block. OpenVPN: issue a new certificate and revoke the old one via Easy-RSA CRL.

- [ ] **Don't log more than necessary.** If your goal is privacy, minimise what the server records. For WireGuard there is no built-in logging. For OpenVPN, set `verb 0` or `verb 1` in production and rotate logs with logrotate.

- [ ] **Use a preshared key in WireGuard.** The `PresharedKey` directive adds a layer of symmetric encryption on top of Curve25519 — useful post-quantum insurance. Generate one per peer with `wg genpsk`.

- [ ] **Verify your public IP after every connection.** Make it a habit: connect, run `curl ifconfig.me`, confirm it shows the VPN server's IP. Takes five seconds and catches misconfiguration immediately.

- [ ] **Run the full DNS + WebRTC + IPv6 leak test suite periodically** — especially after OS updates or VPN client upgrades. A system update can reset DNS settings or re-enable IPv6 without warning.

Course Complete — VPN

From understanding what a VPN actually does and doesn't protect, through building WireGuard and OpenVPN servers from scratch, connecting every type of client, using commercial providers intelligently, and verifying that nothing is leaking — you now have a complete, practical understanding of VPN technology from both sides of the tunnel.

8	3	4	10
Chapters	Protocols covered	Platforms (Win/Mac/iOS/Android)	Hardening checklist items