

Set up a self-hosted WireGuard VPN on Linux

An encrypted tunnel between machines you control, for secure remote access to your own network or a full-tunnel VPN off untrusted Wi-Fi. Covers firewalld, ufw, and nftables on Fedora, RHEL, Ubuntu, and Arch-based systems.

WIREGUARD · WG-QUICK · SELF-HOSTED VPN · NIST 800-53 MAPPED · HOW-TO GUIDE

General educational guidance, not affiliated with or endorsed by WireGuard or NIST. Published 2026-09-10.

7 steps, install to a verified tunnel

server first, then routing and firewall, then the client

4 distributions covered

Fedora 44 · RHEL 9 and 10 · Ubuntu 26.04 · CachyOS

WireGuard wraps your traffic in an encrypted UDP tunnel between two machines you control. People use it two ways: to reach their own devices and home network securely from anywhere, and to route a laptop's entire connection through a server they run so the local network and ISP cannot see it. This guide covers both. The second use has a limit: a VPN you host yourself changes who can see your traffic. It does not make you anonymous.

LIMITS What this does, and what it does not do

It encrypts and authenticates traffic in transit

Between your device and your server, nobody on the path (the coffee-shop Wi-Fi, the ISP) can read or alter your traffic. They see only encrypted UDP going to your server's address.

It moves trust to your server's host

For a full-tunnel setup, the tunnel ends at your server (often a rented VPS), which decrypts your traffic and forwards it to the internet. That host, and its provider, can see the destinations you connect to and any traffic that is not itself encrypted (content inside HTTPS stays protected by TLS).

It changes your apparent source IP

Your traffic appears to come from the server, but this is a single hop, and nothing mixes your traffic with anyone else's, so anyone who can see both ends (the provider, or a well-resourced observer) can link you to it. To be anonymous you need Tor or a multi-hop design.

It does not protect the endpoints

WireGuard protects data while it travels between the two machines. It does nothing for a compromised laptop or server, or for the applications running on them. Harden those machines separately, covering SSH access and the host firewall.

DNS must go through the tunnel or it leaks

If your device keeps using the local network's resolver, your lookups still reveal every site you visit to that network. Step 6 sets an in-tunnel resolver for a full tunnel. Encrypting those lookups and validating the answers is a separate job.

Two ways to use this guide

Remote access

Reach a home server, a lab, or a private subnet from anywhere, without exposing those services to the public internet. The tunnel carries only traffic bound for that subnet (a "split tunnel").

Full-tunnel VPN

Send all of a device's traffic through your server. Use this to protect a laptop on public Wi-Fi, or to present a stable exit IP. It needs routing and NAT on the server (Step 4). A full tunnel protects traffic only while it is up. If it drops, the device falls back to the local link in the clear. For always-on use, add a fail-closed firewall rule that blocks any traffic that is not going through the tunnel (a "kill switch"). The next box has the rule.

The kill switch for an always-on full tunnel

CLIENT [INTERFACE] BLOCK

wg-quick's manual documents the rule that does this. It rejects any packet leaving the client that is neither encrypted for the tunnel nor addressed to the machine itself, so traffic stops when the tunnel drops rather than going out in the clear. It belongs in the tunnel config rather than the host firewall, because it has to appear and disappear with the interface. Add both settings to the client's [Interface] block in Step 6, on a client whose AllowedIPs is `0.0.0.0/0, ::/0`:

```
# Each setting below is ONE line in the file. The indented part is where
# that line wraps on this page, not a second setting.
PostUp = iptables -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
```

Before you rely on it:

- **Install iptables first.** The rule needs the `iptables` command, which `wireguard-tools` does not pull in on Fedora and which Arch's `base` does not carry. Run `iptables -V` on the client and install the package if it is missing, because wg-quick stops at the first failed command and takes the whole tunnel with it.
- **Cover IPv6 as well.** These two lines cover IPv4. On a dual-stack client, add the same pair with `ip6tables` in place of `iptables`, which the manual also suggests, or IPv6 traffic leaves on the local link when the tunnel drops. DHCP still gets out either way, because most DHCP clients use a socket type that skips the firewall, so the rule never sees those packets.
- **Test it at the console, never over SSH.** With the tunnel up, `sudo iptables -S OUTPUT` lists it as a single rule. To watch it hold, simulate a drop that skips the teardown, since `wg-quick down` runs the PreDown line and removes the rule on its way out: run `sudo ip link set wg0 down`, confirm `curl https://ifconfig.me` fails rather than answering over the local link, then restore the tunnel with `sudo wg-quick down wg0 && sudo wg-quick up wg0`. Bringing the link back up on its own leaves the machine with no working connection until wg-quick runs again, because taking it down removes the default route wg-quick put in table 51820 and the kernel does not put it back. The rule also rejects the replies to an inbound SSH session, so dropping the interface over SSH ends that session and blocks the reconnect until you reach a console. A reboot does not clear the rule once `wg-quick@wg0` is enabled at boot (Step 7), because the unit reinstalls it.

- It blocks your local network the whole time it is in place, including while the tunnel is up and carrying traffic. wg-quick leaves the route to your local subnet alone, so traffic to a printer or the router's admin page goes out on the local link, carries no tunnel mark, and the rule rejects it. To keep that traffic, add `PostUp = iptables -I OUTPUT -d 192.168.1.0/24 -j ACCEPT` after the reject line above, so it lands ahead of the reject in the chain, and a matching `PreDown` with `-D` to remove it. On a dual-stack client, add the same pair with `ip6tables` and your local IPv6 prefix. That opens a path around the tunnel for everything on that subnet.

The setup is the same up to Step 3. The difference is the AllowedIPs value on the client (Step 6) and whether the server does NAT (Step 4).

(a) Remote access = split tunnel: only AllowedIPs enter the tunnel

```
+-----+      AllowedIPs =      +-----+
| client wg0 |    10.10.10.0/24,    | server wg0 |
| 10.10.10.2 |    192.168.50.0/24   | 10.10.10.1 |
|           | == encrypted UDP ==> | :51820/udp |
+-----+      +-----+
|           |
| everything else: direct on the normal link
|
v
internet    (local Wi-Fi / ISP sees it as usual)
```

(b) Full tunnel: AllowedIPs = 0.0.0.0/0, ::/0 sends all traffic

```
+-----+      AllowedIPs =      +-----+
| client wg0 |    0.0.0.0/0, ::/0   | server wg0 |
| 10.10.10.2 |    all traffic       | 10.10.10.1 |
| all traffic| == encrypted UDP ==> | :51820/udp | ==> internet
+-----+      +-----+
server: NAT/masquerade + forward accept + ip_forward=1 (Step 4)
local Wi-Fi / ISP sees only encrypted UDP to :51820
```

MAP Distribution cheat-sheet

The WireGuard kernel module has been part of the Linux kernel since 5.6, so on current systems you install only the `wireguard-tools` userspace package. The tools give you `wg` and `wg-quick`.

DISTRIBUTION	INSTALL COMMAND	NOTES
Fedora 44	<code>sudo dnf install wireguard-tools</code>	Module included in the kernel.
RHEL 9 / 10	<code>sudo dnf install wireguard-tools</code>	In the AppStream repository, so you do not need EPEL. Red Hat ships WireGuard as a Technology Preview: no production SLA, and Red Hat does not recommend it for production.
Ubuntu 26.04	<code>sudo apt install wireguard-tools</code>	In main. The <code>wireguard</code> metapackage in universe pulls in the same thing.
CachyOS	<code>sudo pacman -S wireguard-tools</code>	From the Arch extra repository.

PREP Prerequisites

- A server with a reachable address. A VPS with a public IP is the common case. A home server works too if you can forward a UDP port to it, which needs a public address rather than a connection behind carrier-grade NAT. This is the machine peers connect to.
- Root or sudo on both the server and each client.
- A client device running Linux (this guide's client steps use `wg-quick` ; WireGuard also has apps for phones, Windows, and macOS that read the same config).
- Comfort editing config files from the terminal, and enough familiarity with your host firewall to add a rule to it. Steps 4 and 5 give separate instructions for `firewalld`, `ufw`, and a hand-written `nftables` ruleset.

Throughout, replace placeholders: `server.example.com` is your server's public address, `eth0` is the server's internet-facing interface (find it with `ip route get 1.1.1.1`), and the `10.x.x.x` tunnel addresses are examples you can keep or change.

01 Install WireGuard

Install `wireguard-tools` on the server and on each client, using the command from the cheat-sheet. Check that the tools installed and that the kernel module loads:

```
wg --version
sudo modprobe wireguard && echo "module OK"
```

On current kernels the module loads on demand when you bring up a tunnel, so a clean `modprobe` with no error is all you need.

02 Generate keys

Each machine (the server and every client) gets its own key pair. The private key never leaves the machine that owns it; you share only the public key. Run these as root in `/etc/wireguard` , so no copy of the private key lands in a home directory, and set a tight `umask` first so the file is not world-readable. Start a root shell rather than prefixing each line with `sudo`, because `umask` applies to the shell that runs the next command:

```
sudo -i
cd /etc/wireguard
umask 077
wg genkey | tee privatekey | wg pubkey > publickey
```

`privatekey` now holds this machine's private key, `publickey` its public key. Do this once per machine. If you generate the pair somewhere else, move `privatekey` into `/etc/wireguard` once you have copied its value into the config. A preshared key adds a second, symmetric layer on top of the Curve25519 key exchange. WireGuard's whitepaper gives the reason: it guards against an attacker who records your encrypted traffic now and decrypts it years later, once quantum computers can break Curve25519. Generate one per server-and-client pair if your traffic is worth protecting that far ahead (with `umask 077` still set, since this file is a secret too), and put the same value on both ends:

```
wg genpsk > presharedkey
```

Add it as a `PresharedKey =` line inside the matching [Peer] block on each end (the server's [Peer] for that client, and the client's [Peer] for the server). Keep private keys secret and never reuse one key pair across devices. Treat the private key like an SSH private key (NIST SC-12, key management).

03 Configure the server

Create `/etc/wireguard/wg0.conf` on the server. The [Interface] block is the server itself; each [Peer] block is one client. On the server, a peer's AllowedIPs is that client's address inside the tunnel, as a `/32` . AllowedIPs decides both where a

packet goes and which sources a peer may use, a pairing its authors call "cryptokey routing". Sending, WireGuard compares a packet's destination against each peer's list to decide which peer gets it. Receiving, it drops any packet whose source address falls outside the sending peer's list, so one client cannot send traffic claiming to be another. Keep each peer's list as narrow as the traffic it sends.

A remote-access server config, where clients reach the 10.10.10.0/24 tunnel:

```
[Interface]
Address = 10.10.10.1/24
ListenPort = 51820
PrivateKey = <server-private-key>

[Peer]
# Laptop
PublicKey = <client-public-key>
AllowedIPs = 10.10.10.2/32
```

- Set `ListenPort = 51820` explicitly: WireGuard has no built-in default port and picks a random one if you omit the line. 51820/udp is the conventional choice, used in WireGuard's own examples.
- `Address` is the server's tunnel IP. `Address`, and the client's DNS line later, are features of `wg-quick`, the tool this guide uses to bring tunnels up.

Add one [Peer] block per client, each with a unique AllowedIPs address. Then lock down the file (it holds a private key):

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

For a full-tunnel VPN gateway, use the same file and continue to Step 4, which adds routing and NAT so client traffic can reach the internet through the server.

04

Turn on routing and NAT (full tunnel, or reaching a network behind the server)

Skip this step if remote clients only need to reach services on the server itself (its 10.10.10.1 tunnel address). You need this step to reach a subnet behind the server (the 192.168.50.0/24 LAN in diagram (a)), to send full-tunnel traffic to the internet, or to route between two peers. Each of those makes the server forward packets it did not originate. Part 1 below applies to every server that needs this step. In part 2, follow only the block for your firewall: firewalld on Fedora and RHEL, ufw on Ubuntu and CachyOS, or a hand-written nftables ruleset. The notes after the blocks cover a hardened host, a LAN behind the server, traffic between two clients, and IPv6, whichever firewall you run.

1. Enable IP forwarding

The kernel will not route packets between interfaces until you allow it (the same sysctl mechanism as kernel network hardening; a VPN gateway is one of the legitimate reasons to set `ip_forward=1`, alongside routers and container hosts):

```
sudo tee /etc/sysctl.d/70-wireguard-routing.conf >/dev/null <<'EOF'
net.ipv4.ip_forward = 1
EOF
sudo sysctl --system
```

Add `net.ipv6.conf.all.forwarding = 1` to that file only if you route IPv6 to clients, and check how the server gets its own IPv6 address first. Enabling IPv6 forwarding makes the kernel ignore router advertisements, so a server whose address comes from them (SLAAC, common on a VPS) loses its own IPv6 connectivity unless you also set `net.ipv6.conf.eth0.accept_ra = 2` for the internet-facing interface. Put the `accept_ra` line above the forwarding line in the same file, so both land in one `sysctl --system` run: when forwarding goes to 1 the kernel drops every default route it learned from a router advertisement, sparing only an interface whose `accept_ra` is already 2. The address itself stays, so check `ip -6 route` rather than `ip -6 addr` if the server loses IPv6 here. Static and DHCPv6 setups are unaffected.

If the server is hardened to a CIS benchmark

CIS UBUNTU LINUX 26.04 LTS BENCHMARK v1.0.0, SECTION 3.3

A hardened build can carry settings that undo what you just wrote, and the benchmark treats them differently.

- **ip_forward is exempt.** Recommendation 3.3.1.1 asks for `net.ipv4.ip_forward = 0` at Level 1 Server, so most hardened builds carry it, and 3.3.1.2 asks for `net.ipv4.conf.all.forwarding = 0`, which is the same kernel value under another name, so a build carrying either line turns forwarding off. The benchmark exempts the case you are in: its section 3.3.1 preamble says those recommendations do not apply to a system configured as a network router, and a VPN gateway that forwards packets is one. There is no deviation to record.
- **accept_ra is a recorded exception.** The benchmark exempts a router from recommendations 3.3.2.1 to 3.3.2.4 only, says 3.3.2.5 through 3.3.2.8 apply to every system, and 3.3.2.7 asks for `net.ipv6.conf.all.accept_ra = 0`. Its check reads that `all` value alone, so the interface-level setting above still passes the audit. Record the interface-level exception anyway on a host you harden to it.
- **On Ubuntu, check ufw's own sysctl file.** ufw reads `/etc/ufw/sysctl.conf` when it starts, and that file's own header says it overrides both `/etc/sysctl.conf` and `/etc/sysctl.d`. A stock install leaves its forwarding line commented out, so the file you just wrote wins. Where a hardening pass has written `net.ipv4.ip_forward = 0` into it, ufw puts forwarding back to 0 on every boot; set `net.ipv4.ip_forward=1` there instead. Files in `/etc/sysctl.d` load in name order and the last value wins, so a hardening drop-in numbered above 70 also beats your file, and `sudo sysctl --system` reads `/etc/sysctl.conf` after every drop-in, so a value there beats them all. Current Ubuntu no longer ships `/etc/sysctl.conf`, so only an upgraded or hardened server is likely to have one.

2. Masquerade the tunnel subnet, and accept the forwarded traffic

Keep this in your host firewall rather than the tunnel config, so all the firewall rules stay in one configuration. On Fedora and RHEL, firewalld can accept this forwarded traffic without a new rule, on one condition: `eth0` has to be in the default zone by name. Intra-zone forwarding, on by default since firewalld 1.0, accepts a packet that leaves an interface you added to the zone, and a packet arriving on an interface with no zone, as `wg0` is, falls through to the default zone. Run `sudo firewall-cmd --info-zone=$(firewall-cmd --get-default-zone)`: `eth0` should appear on the `interfaces:` line and the zone should show `forward: yes`. `wg0` will not appear there, and does not need to. If `eth0` is missing, add it with `--add-interface=eth0`, or forwarding fails the way the box below describes:

```
# Fedora and RHEL (firewalld): masquerade plus open the port. --permanent persists.
sudo firewall-cmd --permanent --add-masquerade
sudo firewall-cmd --permanent --add-port=51820/udp
sudo firewall-cmd --reload
```

`--add-masquerade` masquerades every IPv4 packet the zone forwards, including traffic that has nothing to do with the tunnel. On a server that also routes containers or a second network, scope it by source: `sudo firewall-cmd --permanent --add-rich-rule='rule family=ipv4 source address=10.10.10.0/24 masquerade'`. Rich rules cannot match an interface, so that form still covers traffic leaving `wg0`; the paragraph on client-to-client traffic below says what to do where that is a problem.

On Ubuntu or CachyOS with ufw active, keep the forward rule and the NAT in ufw's own configuration, so one tool manages both. ufw's default forward policy is deny, so allow the tunnel's forwarded traffic, and add the masquerade to `/etc/ufw/before.rules`:

```
# Allow the tunnel's forwarded traffic (ufw blocks forwarding by default)
sudo ufw route allow in on wg0 out on eth0
```

```
# /etc/ufw/before.rules: add this *nat block at the end, after the *filter section's COMMIT
*nat
:POSTROUTING ACCEPT [0:0]
-A POSTROUTING -s 10.10.10.0/24 -o eth0 -j MASQUERADE
COMMIT

# then reload so both take effect
sudo ufw reload
```

On a host whose firewall is a hand-written nftables ruleset, with neither ufw nor firewalld managing it, add the NAT rule and a forward-accept rule to the file your distribution's nftables service reads: `/etc/nftables.conf` on Ubuntu and CachyOS, `/etc/sysconfig/nftables.conf` on Fedora and RHEL. `systemctl cat nftables` shows which one, on its `ExecStart` line. Then load that file and enable the service so the rules survive a reboot (replace eth0 with your WAN interface and the subnet with your tunnel's):

```
# In that file: NAT so replies find their way back...
table inet nat {
    chain postrouting {
        type nat hook postrouting priority srcnat; policy accept;
        ip saddr 10.10.10.0/24 oifname "eth0" masquerade
    }
}

# ...AND accept the forwarded traffic in the filter table's forward
# chain, or a policy-drop chain discards the packet first (see below):
table inet filter {
    chain forward {
        type filter hook forward priority filter; policy drop;
        # ... your existing rules ...
        iifname "wg0" oifname "eth0" accept
        iifname "eth0" oifname "wg0" ct state established,related accept
    }
}
```

```
sudo nft -f /etc/sysconfig/nftables.conf # or /etc/nftables.conf
sudo systemctl enable --now nftables    # otherwise the rules are lost on reboot
```

Keep the `type filter hook forward` line. A chain declared without it is a plain chain that no packet ever enters, so the two rules under it would never match. Where your ruleset already has a forward chain, leave its existing type line and policy as they are and add only the two `iifname` rules. Your ruleset file should start with `flush ruleset`, or loading it a second time adds the rules again. That line clears every table in the kernel, firewalld's and ufw's included, so turn off whichever one is running before you load the file and keep one manager: `sudo systemctl disable --now firewalld` on Fedora and RHEL, `sudo systemctl disable --now ufw` on Ubuntu and CachyOS.

A hardened host must also accept the forwarded traffic

FORWARD CHAIN vs NAT CHAIN

A forwarded packet reaches the filter `forward` chain before the NAT `postrouting` chain, and it must pass the forward chain of every firewall on the host; masquerade only rewrites the source address and never admits a packet through a filter. If a forward chain sits at `policy drop` (a hardened host's does, and so does ufw by default), the kernel drops the packet before masquerade ever runs, so the tunnel handshakes but carries no traffic. On nftables, the `iifname "wg0" oifname "eth0" accept` pair above is what lets it through, and the `ufw route allow` rule grants the same permission on ufw. On firewalld the default zone's forwarding covers it as long as eth0 is in that zone by name. wg0 does not need to be, because the accept is keyed on the interface the packet leaves by (Step 4).

The traditional rule is `iptables -t nat -A POSTROUTING -s 10.10.10.0/24 -o eth0 -j MASQUERADE` plus `iptables -A FORWARD -i wg0 -o eth0 -j ACCEPT` and `iptables -A FORWARD -i eth0 -o wg0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT`, but bare iptables rules are not saved across reboots on their own (use `iptables-persistent`, or prefer ufw's files, firewalld, or the nftables file above).

A return path for a subnet behind the server. To reach a LAN behind the server (not the internet), replies to 10.10.10.0/24 must find their way back: either masquerade the tunnel subnet onto the LAN interface (`ip saddr 10.10.10.0/24 oifname "<lan-if>" masquerade`), or add a static route for 10.10.10.0/24 via the server on each LAN host.

Traffic between two clients. It arrives and leaves on wg0, so the rules above, which pair wg0 with eth0, do not cover it:

- nftables: add `iifname "wg0" oifname "wg0" accept`.
- ufw: `sudo ufw route allow in on wg0 out on wg0`.
- firewalld: put wg0 in the default zone first (`sudo firewall-cmd --permanent --zone=$(firewall-cmd --get-default-zone) --add-interface=wg0` , then reload). Intra-zone forwarding accepts a packet only when it leaves an interface you added to the zone by name, so it does not cover wg0 while wg0 is unbound. Both firewalld masquerade forms above also rewrite this traffic, because their rule matches every interface except `lo` and wg0's traffic reaches the default zone either way, so each client sees the other's as coming from 10.10.10.1. Exclude the tunnel subnet as a destination to stop that: `sudo firewall-cmd --permanent --add-rich-rule='rule family=ipv4 source address=10.10.10.0/24 destination not address=10.10.10.0/24 masquerade'`. The nftables and ufw rules scope on the outgoing interface, so they leave it alone.
- On every firewall, each client's AllowedIPs also has to include the other's tunnel address.

IPv4 and IPv6. These commands masquerade IPv4 only. A full-tunnel client using `AllowedIPs = 0.0.0.0/0, ::/0` therefore routes its IPv6 into an IPv4-only tunnel, where it is dropped: that stops IPv6 from leaking around the VPN, but it does not carry IPv6 through it. To pass IPv6 as well:

- Give the tunnel a unique local IPv6 range of its own on both ends. RFC 4193 section 3.2.1 has you generate the 40 bits after `fd` at random, which `openssl rand -hex 5` supplies. Use the prefix it gives you rather than the `fd00::` that every example prints, since an all-zero range collides with the next network that copied the same one.
- Add each client's IPv6 address as a `/128` beside its `/32` in the server's [Peer] block.
- Add the IPv6 forwarding line from part 1 above.
- Add IPv6 masquerade and forward-accept rules: `ip6 saddr fdxx:xxxx:xxxx::/64 oifname "eth0" masquerade`, or a firewalld rich rule.

Where the provider routes a /64 or larger to your server, take the tunnel's addresses from that prefix instead and skip the IPv6 masquerade.

05 Open the firewall port

WireGuard listens on UDP, so open its port on the server:

```
# Fedora and RHEL (firewalld) -- skip if you already opened it in Step 4
sudo firewall-cmd --permanent --add-port=51820/udp
sudo firewall-cmd --reload
```

```
# Ubuntu and CachyOS (ufw); Step 4 added the forwarding rule a gateway also needs
sudo ufw allow 51820/udp
```

An open WireGuard port answers nothing it cannot authenticate: it stores no state before authentication and sends no reply to an unauthenticated packet. The whitepaper calls this "silence is a virtue". What a scanner makes of that depends on how your firewall answers its other ports. ufw drops every unopened port, so 51820/udp looks like the rest of them. firewalld's default zone rejects instead, so every other UDP port returns an ICMP rejection and 51820/udp is the only one that stays quiet, which is itself a signal.

Reaching a service on the server itself takes one more opening. Opening 51820/udp lets the tunnel form and leaves everything behind it closed. The tunnel interface is filtered like any other, so a web app or a database listening on 10.10.10.1 stays blocked until you allow it, and the opening should be scoped to the tunnel rather than opening the port to the internet:

```
# Fedora and RHEL (firewalld): HTTPS from the tunnel subnet only
RULE='rule family=ipv4 source address=10.10.10.0/24 service name=https accept'
sudo firewall-cmd --permanent --add-rich-rule="$RULE"
sudo firewall-cmd --reload

# Ubuntu and CachyOS (ufw): HTTPS arriving on the tunnel only
sudo ufw allow in on wg0 to any port 443 proto tcp
```

The two rules differ in what they match. The ufw rule matches the interface, so only a packet that arrived on the tunnel can use it. The rich rule matches the source address inside a zone that also holds eth0, so a packet arriving from the internet with a spoofed source address in 10.10.10.0/24 matches it as well, and reverse path filtering does not catch that on a stock host, which runs in loose mode with a route to that subnet via wg0. An attacker never sees the replies, which makes this hard to use against a TCP service and much easier against a UDP one. So on a server with one internet-facing interface, turn on strict reverse path filtering in the drop-in file from Step 4 before you open a UDP port to peers. The kernel takes the higher of `net.ipv4.conf.all.rp_filter` and the interface's own value, and the distributions ship different defaults:

- Ubuntu: add both `net.ipv4.conf.eth0.rp_filter = 1` and `net.ipv4.conf.all.rp_filter = 1`. The procs package sets the `all` value to 2 in `/usr/lib/sysctl.d/55-network-security.conf`, so the second line is always needed there.
- Fedora, RHEL and CachyOS: add `net.ipv4.conf.eth0.rp_filter = 1`. They leave the `all` value at 0 and set each interface to 2 instead, so the interface line is enough. Confirm with `sysctl net.ipv4.conf.all.rp_filter`, and add the `all` line as well where it returns 2.

Strict mode drops a reply that would leave by a different interface from the one the request arrived on, so check any second uplink or policy route on the server first. Matching on the source subnet also leaves wg0 unbound, falling through to the default zone, and Step 4 relies on that for forwarding; giving wg0 a zone of its own would break that. Repeat the rule for each port peers need, using `port port=8443 protocol=tcp` in place of `service name=https` where the service has no firewalld name.

If SSH to this server is your only way in, confirm SSH still works before you rely on the tunnel.

06 Configure the client

Create `/etc/wireguard/wg0.conf` on the client. Here the [Peer] is the server, and AllowedIPs means "what to send into the tunnel," the setting that decides split tunnel or full. A full-tunnel client (all traffic through the server):

```
[Interface]
PrivateKey = <client-private-key>
Address = 10.10.10.2/24
DNS = 1.1.1.1

[Peer]
PublicKey = <server-public-key>
# PresharedKey = <preshared-key>
Endpoint = server.example.com:51820
AllowedIPs = 0.0.0.0/0, ::/0
PersistentKeepalive = 25
```

Uncomment the `PresharedKey` line only if you generated one in Step 2, and put the same value on the server's [Peer] block for this client.

Split-tunnel client (only the private subnet through the tunnel, everything else direct): change AllowedIPs to the target networks, for example `AllowedIPs = 10.10.10.0/24, 192.168.50.0/24`, and usually drop the DNS line. What the notable lines do:

- `AllowedIPs = 0.0.0.0/0, ::/0` captures all IPv4 and IPv6 traffic. A narrower value gives a split tunnel. For an always-on full tunnel, add the kill switch's `PostUp` and `PreDown` lines to this [Interface] block as well; the box near the start of the guide has them.
- `DNS = 1.1.1.1` sets the resolver the client uses while the tunnel is up. Because a full tunnel captures all traffic, the query travels inside the tunnel and out through the server, so the local network never sees your lookups. Use any resolver you trust: a public one like `1.1.1.1` or `9.9.9.9`, or one you run yourself (pointing DNS at the server's tunnel address only works if a resolver is listening there, which is extra setup). wg-quick applies the setting through `resolvconf`, the small helper that wires a DNS server into the system resolver. Fedora and Ubuntu provide it through `systemd-resolved` with no extra work; RHEL and CachyOS each need a step of their own, which Troubleshooting gives.
- `PersistentKeepalive = 25` sends a keepalive every 25 seconds, the value WireGuard's own examples use. It keeps the NAT mapping at the client's side open so the server can still reach a quiet client; a client that only initiates traffic works without it, so the manual says most users will not need it. Leave it off for a server that only receives.
- Only the client needs an `Endpoint` here. It is an optional per-peer setting on either side; the server learns each peer's current address from its incoming packets, so a client can roam. WireGuard resolves that name once, when the tunnel comes up, so if the server's address changes, as a home connection's usually does, the client keeps sending to the old one until you bring the tunnel down and up. `wireguard-tools` ships a `reresolve-dns` example for running that check from a timer: `/usr/share/doc/wireguard-tools/contrib/reresolve-dns/` on Fedora and RHEL, `/usr/share/doc/wireguard-tools/examples/reresolve-dns/` on Ubuntu, and `/usr/share/wireguard-tools/examples/reresolve-dns/` on CachyOS.

Lock the file down (`sudo chmod 600 /etc/wireguard/wg0.conf`). Finally, tell the server about this client: add a [Peer] block to the server's wg0.conf with the client's public key and its /32 tunnel address, then bring the server tunnel up (Step 7).

07 Start it, verify, and make it persistent

Bring the tunnel up on both ends by the interface name (`wg0` maps to `/etc/wireguard/wg0.conf`):

```
sudo wg-quick up wg0
```

Check the tunnel status and that a handshake happened:

```
sudo wg show
```

A line reading **latest handshake:** with a recent time means the two ends authenticated and exchanged keys. If it never appears, see Troubleshooting.

Verify it is carrying your traffic (full tunnel):

```
curl https://ifconfig.me      # should print your SERVER's public IP, not your local one
resolvectl status wg0        # the DNS server here should be the one you set
```

resolvectl is a systemd-resolved tool, and it reports the tunnel's own setting whether or not the system reads it, so run **resolvectl status** with no interface as well and check it says **resolv.conf mode: stub**. On a client using openresolv instead, read the nameserver out of **/etc/resolv.conf** and confirm it answers with **dig +short example.com**. The first command tells ifconfig.me your server's address; to keep that between you and the server, compare against what the server itself reports with **ip -4 addr**. On a home server behind a router that address is private, so read the WAN address off the router instead.

If ifconfig.me shows the server's IP, your traffic is exiting through the tunnel. A server that never enables the unit drops every peer at its next reboot, so do this on both machines. Take the manual tunnel down so systemd can own it, then enable it to start at boot with the unit file the package installs:

```
sudo wg-quick down wg0
```

```
sudo systemctl enable --now wg-quick@wg0
```

Bring the tunnel down first, because **--now** re-runs **wg-quick up**, which fails if wg0 is already up and leaves the service in a failed state. To take the tunnel down later: **sudo wg-quick down wg0** (and **sudo systemctl disable --now wg-quick@wg0** to stop it starting at boot).

KEYS Replace a key, remove a peer

WireGuard keys never expire, so replacing one is a job you do by hand, and a lost laptop is what usually makes it urgent. Generate a new pair on the client (Step 2), put the new public key in the server's matching [Peer] block, and put the new private key in the client's [Interface]. To take a peer away for good, delete its whole [Peer] block from the server's config. Either way, apply the change on the server without dropping the other tunnels:

```
sudo bash -c 'wg syncconf wg0 <(wg-quick strip wg0)'
```

wg-quick strip prints the config with the wg-quick-only settings removed and **wg syncconf** applies only what differs from the running interface, which is the pairing wg-quick's own manual shows. It runs inside **bash -c** because **sudo** closes the file descriptor that **<()** opens. Once **wg-quick@wg0** owns the tunnel (Step 7), **sudo systemctl reload wg-quick@wg0** runs the same pair from the unit file. A peer whose block is gone can no longer authenticate. That removes a lost device's access. On the client, bring the tunnel down and up (**sudo wg-quick down wg0 && sudo wg-quick up wg0**) so it picks up the new private key. Replace a preshared key the same way, on both ends in the same sitting, since a mismatched one stops the handshake. If the server itself is compromised, its private key is too. Generate a new server pair, update the **PublicKey** in every client's [Peer] block, and replace any preshared keys in the same sitting.

CRYPTO The cryptography, and why it fails FIPS

WireGuard uses one fixed set of modern primitives: Curve25519 for the key exchange (the X25519 ECDH function, inside a handshake built on the Noise protocol framework), ChaCha20-Poly1305 for authenticated encryption, and BLAKE2s for hashing. WireGuard has no cipher or protocol agility by design, so there is no handshake where a downgrade could be

forced; if a primitive were ever broken, WireGuard would release a new version and every endpoint would have to update to it.

WireGuard cannot be used where FIPS 140-validated cryptography is mandated

RELEVANT TO MANY US FEDERAL AND REGULATED SYSTEMS

Two of its primitives, ChaCha20-Poly1305 and BLAKE2s, are not among the FIPS-approved algorithms, and its X25519 key agreement is not an approved key-establishment scheme. (Ed25519 signatures were approved in FIPS 186-5, but WireGuard does not use those; it uses X25519 for key agreement, a different and non-approved use.) NIST's own VPN guide says the same of the algorithms it lists: SP 800-77 Rev 1 names Curve25519 and ChaCha20-Poly1305 among them and notes that none is NIST-approved. It does not discuss BLAKE2s. If you need FIPS compliance, use an IPsec or TLS VPN with approved algorithms instead. RHEL enforces this in the operating system: Red Hat's own WireGuard documentation says WireGuard works only while FIPS mode is off, so turning FIPS mode on takes the tunnel with it. On RHEL 10, FIPS mode is set during installation and cannot be switched on afterwards, so that choice is made before the tunnel exists. Where a product is marketed as FIPS-validated WireGuard, read what the validation covers before you rely on it. Standard WireGuard's algorithms are fixed and none of them is approved, so a FIPS claim means something has changed; ask which module is validated and which algorithms it runs.

NIST NIST alignment

There is no CIS Benchmark specific to WireGuard. A self-hosted VPN maps to these NIST SP 800-53 Rev 5 controls:

WHAT IT PROVIDES	NIST SP 800-53 REV 5
Encrypted, authenticated tunnel (confidentiality and integrity in transit)	SC-8 (Transmission Confidentiality and Integrity), SC-8(1)
The VPN server as a controlled network boundary	SC-7 (Boundary Protection)
Protected remote access to a private network	AC-17 (Remote Access), AC-17(2)
Key pairs, preshared keys, and rotation	SC-12 (Cryptographic Key Establishment and Management)
Use of cryptographic protection	SC-13 (Cryptographic Protection)
Split tunnel or full tunnel, and what the split allows	SC-7(7) (Split Tunneling for Remote Devices)

Running WireGuard is one part of meeting these controls rather than the whole of them: the key management and access review that SC-12 and AC-17 ask for are the organization's to write down. WireGuard authenticates a device key. There is no user directory behind it and no second factor, and the keys never expire, so a copied client config keeps working until someone deletes that peer from the server. Where a baseline pairs AC-17 with IA-2(1) and IA-2(2), the multi-factor sign-in has to sit in front of the services the tunnel reaches. SC-7(7) asks an organization to prevent split tunneling for remote devices unless the split is securely provisioned, so the split-tunnel option above and the local-subnet carve-out in the kill-switch box are both choices you have to justify and record. WireGuard ships nothing for AC-17(1), monitoring and control. `wg show` reports each peer's public key, its current endpoint address, its byte counters and the time of its last handshake, and it prints them as they stand at that moment while keeping no history, so poll `wg show all dump` into wherever you keep logs if you need the record.

NIST SP 800-77 Rev 1 (Guide to IPsec VPNs, 2020) discusses WireGuard directly in its Section 8.3, and the algorithm observation in the FIPS box comes from there. That section dates from 2020 and predates some of WireGuard's current documentation, so read it for the algorithm point rather than the feature list.

- **wg-quick up fails partway on the client**

If the error names an `ip -6` command, IPv6 is switched off on the host, which a hardened build often does at boot. wg-quick stops at the first failed command and its own cleanup deletes the interface on the way out, so there is nothing left to take down: `wg-quick down wg0` answers `'wg0' is not a WireGuard interface`. Take `::/0` out of AllowedIPs and bring the tunnel up again.

- **No handshake in wg show**

The server's UDP port is not reachable, or a key or endpoint is wrong. Work through these:

- The firewall allows 51820/udp (Step 5).
- The client's Endpoint is correct.
- Each side has the other side's public key. A common mistake is pasting your own.
- On a cloud server, the provider's security group or network ACL filters the packet before the host firewall sees it, so 51820/udp has to be open in both.
- On a home server, the router still forwards 51820/udp to it and the connection carries a public address. An ISP that puts you behind carrier-grade NAT leaves nothing to forward, and the server has to move somewhere reachable.
- The clock on both machines. WireGuard's replay protection stamps each handshake with the sending machine's wall-clock time, and the receiver keeps the highest stamp it has seen from that peer and discards anything at or below it, so a client whose clock ran ahead and was then corrected cannot connect until its clock passes the stamp already recorded. Running NTP on both ends avoids it.

- **Handshake works but no internet (full tunnel)**

Routing or NAT is missing on the server. Recheck Step 4:

- `ip_forward` must be 1: `sysctl net.ipv4.ip_forward`.
- Masquerade must be active for the tunnel subnet.
- The forwarded traffic must be accepted: on ufw, the `ufw route allow in on wg0 out on eth0` rule and the before.rules masquerade; on nftables, the filter forward chain must accept wg0 to eth0. Check the forward chain as well as NAT with `sudo ufw status verbose` or `sudo nft list chain inet filter forward` (or `iptables -L FORWARD -v`). A drop forward policy with no accept rule blocks all forwarded traffic even when `ip_forward=1` and masquerade look correct.

If `ip_forward` reads 1 now but goes back to 0 after a reboot, something is setting it later:

- Check `/etc/ufw/sysctl.conf` and any file in `/etc/sysctl.d` whose name sorts after 70.
- Search those files for `net.ipv4.conf.all.forwarding` as well as `ip_forward`: both names write the same kernel value, so a file carrying only the first still switches your setting off.
- On a server upgraded from an older Ubuntu, check whether `/etc/sysctl.d/99-sysctl.conf` still points at `/etc/sysctl.conf`: that symlink is what applies `/etc/sysctl.conf` at boot, and current Ubuntu no longer ships it.
- If it reads 0 straight after you ran `sudo sysctl --system`, look at `/etc/sysctl.conf` first, since that command reads it last.

- **Handshake works but a subnet behind the server is unreachable**

You skipped Step 4. Set `ip_forward=1` and give the tunnel subnet a return path: masquerade it onto the LAN interface, or add a route to 10.10.10.0/24 on the LAN hosts. Check too that the client's AllowedIPs includes that subnet (Step 6), which produces the same symptom when it is missing.

- **Handshake works but a service on the server does not answer**

The host firewall is blocking it. Opening 51820/udp lets the tunnel form and leaves everything behind it closed, so allow each port you want peers to reach (end of Step 5). Check too that the service is listening on the tunnel address rather than only on 127.0.0.1: run `sudo ss -ltnup | grep ':443'` for the port you want and read the address it is bound to. A service bound to 127.0.0.1 cannot answer on the tunnel; one bound to 10.10.10.1, or to every address, can.

- **wg-quick reports resolvconf: command not found**

The `DNS=` line needs a `resolvconf` provider, and `wg-quick` stops at the first failed command, so a missing one makes `wg-quick up` fail. What each distribution needs:

- Fedora and Ubuntu provide one through `systemd-resolved`. Nothing to add.
- CachyOS: install `systemd-resolvconf` (which wires it into `systemd-resolved`) or `openresolv`.
- RHEL 10: RHEL installs no provider, so add one. Install `systemd-resolved`, point the system at its stub resolver, and start it: `sudo dnf install systemd-resolved`, then `sudo ln -sf /run/systemd/resolve/stub-resolv.conf /etc/resolv.conf`, then `sudo systemctl enable --now systemd-resolved` and `sudo systemctl restart NetworkManager`. Red Hat has supported the service since 10.0. The symlink is what hands DNS to it: without one, `NetworkManager` carries on writing `/etc/resolv.conf` itself, so `wg-quick up` stops failing while your lookups still go to the local network's resolver.
- RHEL 9: the service is an unsupported Technology Preview, so configure the tunnel in `NetworkManager` instead, the way Red Hat's own WireGuard documentation does, and set the resolver yourself with `nmcli connection modify <name> ipv4.dns 1.1.1.1`, since that chapter covers the tunnel and not the DNS. On a full tunnel add `ipv4.dns-priority -42` as well: a negative priority makes `NetworkManager` use only the lowest-priority connection's servers, so the tunnel's resolver wins while it is up.

- **DNS does not resolve, or leaks to the local network**

Find out which resolver the system is using. `resolvectl status wg0` does not tell you: it reports the tunnel's own setting whether or not anything reads it. Run `resolvectl status` with no interface, and look for `resolv.conf mode: stub`. Anything else means `/etc/resolv.conf` is a plain file listing other servers and the tunnel's setting is being ignored, which is the RHEL case above. The resolver you named also has to answer: a public resolver such as 1.1.1.1 works over a full tunnel, while pointing DNS at the server's own tunnel address works only if you run a resolver on the server listening there.

- **Handshake and ping work, but large transfers stall**

Usually the tunnel MTU. `wg-quick` sets it from the endpoint address or the default route, which suits most links, but on a path with a smaller MTU (PPPoE, some mobile networks) the encapsulated packets are too big for it. The router that drops an oversized packet is supposed to send back an ICMP message saying so, and firewalls often filter those, so large packets vanish while small ones get through. Set a lower value on the client's [Interface], for example `MTU = 1380`, then take the tunnel down and up and re-test with a large download.

- **Tunnel drops when idle**

Add `PersistentKeepalive = 25` on the client (Step 6); a NAT device timed out the mapping.

- **ifconfig.me still shows your local IP**

`AllowedIPs` on the client is not `0.0.0.0/0, ::/0`, so traffic is not being routed into the tunnel.

ROLLBACK

How to roll back

```
sudo systemctl disable --now wg-quick@wg0 # stop and unset boot start
sudo wg-quick down wg0                    # if still up
sudo rm /etc/wireguard/wg0.conf           # remove the config
```

Step 2 wrote `privatekey`, `publickey` and, if you generated one, `presharedkey` into `/etc/wireguard`. Remove those as well, or the private key stays on the host after the tunnel is gone:

```
sudo rm -f /etc/wireguard/privatekey /etc/wireguard/publickey \
        /etc/wireguard/presharedkey
```

On a rented VPS, deleting a file does not erase it from the provider's storage, so treat a key that has lived there as one to replace rather than reuse.

If you enabled forwarding and NAT only for this VPN, remove `/etc/sysctl.d/70-wireguard-routing.conf` and set the values back by hand, since re-running `sudo sysctl --system` only re-applies what the remaining files say:

```
sudo sysctl -w net.ipv4.ip_forward=0
sudo sysctl -w net.ipv6.conf.all.forwarding=0 # only if you set it
```

On a full-tunnel client, `wg-quick` also sets `net.ipv4.conf.all.src_valid_mark=1` and does not undo it; `sudo sysctl -w net.ipv4.conf.all.src_valid_mark=0` returns it. Skip the forwarding lines above if something else on the host still needs forwarding, such as a container runtime. Then undo the masquerade and forward-accept rules from Step 4 (on `ufw`, the `route allow` rule and the `before.rules` block). Close the port from Step 5 as well (`sudo firewall-cmd --permanent --remove-port=51820/udp` or `sudo ufw delete allow 51820/udp`), along with any per-service rule you added for peers.

DONE

What you achieved

- An encrypted, authenticated WireGuard tunnel between your own machines, using one fixed set of modern algorithms, so nothing can negotiate it down to something weaker.
- Either secure remote access to a private subnet, or a full-tunnel VPN that takes a device's traffic off the local network and presents your server's exit IP.
- In-tunnel DNS on a full tunnel, so lookups do not leak to the network you are on. A split tunnel keeps using the local resolver.
- On a full tunnel, your traffic protected while it travels, so the local network and your ISP cannot read it. Whoever hosts your server can. A split tunnel protects only what AllowedIPs sends into it.

REF

References

WireGuard and its protocol page · wireguard.com · wireguard.com/protocol

WireGuard whitepaper (Donenfeld, NDSS 2017) · wireguard.com/papers/wireguard.pdf

`wg(8)` and `wg-quick(8)` manuals · man7.org/linux/man-pages

Ubuntu Server: WireGuard VPN · ubuntu.com/server/docs · Red Hat: Setting up a WireGuard VPN (RHEL 9 and 10) · docs.redhat.com

`ufw` route rules and `before.rules` NAT · manpages.ubuntu.com (`ufw`, `ufw-framework`)

CIS Ubuntu Linux 26.04 LTS Benchmark v1.0.0, recommendations 3.3.1.1 (`ip_forward`) and 3.3.2.7 (`accept_ra`) · cisecurity.org

NIST SP 800-53 Rev 5 · SP 800-77 Rev 1 (Guide to IPsec VPNs) · csrc.nist.gov

anobytes: the guides the Limits box points at · anobytes.com/resources#secure-ssh-access-linux · `#host-firewall-hardening` · `#secure-dns-quad9-dot-dnssec`

"WireGuard" is a registered trademark of Jason A. Donenfeld.