anoBytes | TECHNOLOGY SOLUTIONS

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-ALIGNED · HOW-TO GUIDE

General educational guidance, not affiliated with or endorsed by WireGuard or NIST. Published 2026-08-07.

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 worth naming up front: 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 with no mixing, so anyone who can see both ends (the provider, or a well-resourced observer) can link you to it. Anonymity needs Tor or a multi-hop design.

It does not protect the endpoints

WireGuard secures data on the wire. A compromised laptop or server, and the applications on them, are outside its scope. Keep host hardening in place (separate SSH and host-firewall hardening).

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; pair it with separate DNS hardening.

USES

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, so for always-on use add a fail-closed firewall rule (a "kill switch") that blocks traffic when the tunnel is down.

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

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

MAP Distribution cheat-sheet

The WireGuard kernel module ships with 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	`sudo dnf install wireguard-tools`	Module in-tree.
RHEL 9 / 10	`sudo dnf install wireguard-tools`	In AppStream, no EPEL needed. Red Hat ships WireGuard as a Technology Preview, not covered by production support.
Ubuntu 26.04	`sudo apt install wireguard`	The `wireguard` package pulls in `wireguard-tools`.
CachyOS	`sudo pacman -S wireguard-tools`	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. 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).
- Basic comfort editing files and using the terminal.

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. Confirm it is there and that the kernel module is available:

```
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. Generate them with a tight umask so the private key is not world-readable:

```
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. For an optional extra symmetric layer shared between a server and one client, also generate a preshared key once (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`. This is how WireGuard knows which peer a packet belongs to (its authors call this "cryptokey routing"). 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
```

- `ListenPort = 51820` is set explicitly. WireGuard has no built-in default port (it picks a random one if you omit this); 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.

1. Enable IP forwarding

A gateway must be allowed to route packets between interfaces (this is 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
net.ipv6.conf.all.forwarding = 1
EOF
sudo sysctl --system
```

Add the IPv6 line 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. Static and DHCPv6 setups are unaffected.

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

Keep this in your host firewall rather than the tunnel config, so it lives in one place. On Fedora and RHEL, firewalld accepts this forwarded traffic by itself: wg0, unassigned, joins the default zone beside eth0, and a zone forwards traffic between its own interfaces by default:

```
# 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
```

On Ubuntu or CachyOS with ufw active, keep the forwarding and the NAT in ufw's own configuration, so one manager owns the rules. 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 (no ufw), add the NAT rule and a forward-accept rule to `/etc/nftables.conf`, then load it and enable the service so it survives a reboot (replace eth0 with your WAN interface and the subnet with your tunnel's):

```
# /etc/nftables.conf: 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 {
        # ... your existing rules / policy drop ...
        iifname "wg0" oifname "eth0" accept
        iifname "eth0" oifname "wg0" ct state established,related accept
    }
}
```

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

Masquerade alone is not enough on a hardened host

FORWARD CHAIN vs NAT CHAIN

A forwarded packet hits 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 packet is dropped before masquerade ever runs, so the tunnel handshakes but carries no traffic. The `iifname "wg0" oifname "eth0" accept` pair above is what lets it through on nftables, the `ufw route allow` rule is the same permission on ufw, and on firewalld the default zone's forwarding between its own interfaces covers it, as long as wg0 and eth0 share that zone.

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. Add `iifname "wg0" oifname "wg0" accept` on nftables or `sudo ufw route allow in on wg0 out on wg0` on ufw; firewalld's intra-zone forwarding already allows it. 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 an IPv6 range (`fd00::/64` on both ends), keep the IPv6 forwarding line above, and add IPv6 masquerade and forward-accept rules (`ip6 saddr fd00::/64 oifname "eth0" masquerade`, or a firewalld rich rule).

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 --add-port=51820/udp --permanent
sudo firewall-cmd --reload
```

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

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 vs full tunnel. 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>
Endpoint = server.example.com:51820
AllowedIPs = 0.0.0.0/0, ::/0
PersistentKeepalive = 25
```

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.
- `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 actually 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 out of the box through `systemd-resolved`. RHEL does not run systemd-resolved by default, so there the DNS line makes `wg-quick up` fail: configure the tunnel in NetworkManager (`nmcli`) instead, which handles the DNS itself and is how Red Hat's own WireGuard documentation does it, or drop the line and set the resolver another way. On CachyOS, if wg-quick reports `resolvconf: command not found`, install `systemd-resolvconf` (which wires it into systemd-resolved) or `openresolv`. Pair with separate DNS hardening.
- `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, which is why the manual says most users will not need it. Leave it off for a server that only receives.
- `Endpoint` is only on the client. The server learns each peer's current address from its incoming packets, which is what lets a client roam.

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 actually carrying your traffic (full tunnel):

```
curl https://ifconfig.me          # should print your SERVER's public IP, not your local one
resolvectl query example.com      # confirm DNS resolves through the tunnel
```

If ifconfig.me shows the server's IP, your traffic is exiting through the tunnel. Take the manual tunnel down so systemd can own it, then enable it to start at boot with the shipped unit:

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

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

Bringing the tunnel down first matters: `--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).

CRYPTO The cryptography, and a FIPS caveat

WireGuard uses one fixed set of modern primitives: Curve25519 for the key exchange (an ECDH handshake, the X25519 function, 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, the fix is a new version that every endpoint updates to.

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: SP 800-77 Rev 1 notes that none of WireGuard's algorithms are NIST-approved. If you need FIPS compliance, use an IPsec or TLS VPN with approved algorithms instead. Products marketed as "FIPS-validated WireGuard" achieve it by substituting FIPS-approved primitives, which is a different protocol from standard WireGuard.

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, SC-8(1)
The VPN server as a controlled network boundary	SC-7 (Boundary Protection)
Protected remote access to a private network	AC-17, AC-17(2)
Key pairs, preshared keys, and rotation	SC-12 (Cryptographic Key Establishment and Management)
Use of cryptographic protection	SC-13

NIST SP 800-77 Rev 1 (Guide to IPsec VPNs, 2020) discusses WireGuard directly in its Section 8.3, which is where the algorithm observation in the FIPS box comes from. SP 800-113 (Guide to SSL VPNs, 2008) predates WireGuard; its general guidance still applies.

HELP Troubleshooting

- **No handshake in wg show**
  The server's UDP port is not reachable, or a key or endpoint is wrong. Confirm the firewall allows 51820/udp (Step 5), the client's Endpoint is correct, and each side has the other side's public key (a common mistake is pasting your own).

- **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, and the forwarded traffic must be accepted: on ufw that is 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: `sudo ufw status verbose` , `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.

- **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.

- **DNS does not resolve, or leaks to the local network**
  Two causes. First, the DNS= line needs a resolvconf provider: Fedora and Ubuntu ship one through `systemd-resolved`; on RHEL configure the tunnel's DNS in NetworkManager instead (Step 6), and on CachyOS install `systemd-resolvconf` or `openresolv` if wg-quick reports `resolvconf: command not found`. Second, the resolver you named must actually answer: a public resolver such as 1.1.1.1 works over a full tunnel, but pointing DNS at the server's own tunnel address only works if you run a resolver on the server listening there. Confirm your lookups use the tunnel resolver.

- **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 where the path carries less (PPPoE, some mobile networks) the encapsulated packets are too big for it. The ICMP messages that would report that are often filtered, so large packets disappear 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 (keep saved keys elsewhere)
```

If you enabled forwarding and NAT only for this VPN, remove `/etc/sysctl.d/70-wireguard-routing.conf` (then `sudo sysctl --system`) and undo the masquerade and forward-accept rules from Step 4 (on ufw, the `route allow` rule and the before.rules block).

DONE What you achieved

- An encrypted, authenticated WireGuard tunnel between your own machines, using modern fixed cryptography with no downgrade path.
- 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, so lookups do not leak to the network you are on.
- Your traffic protected in transit, with trust moved from the local network and your ISP to whoever hosts your server.

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)

NIST SP 800-53 Rev 5 · SP 800-77 Rev 1 (Guide to IPsec VPNs) · SP 800-113 (Guide to SSL VPNs) · csrc.nist.gov