Securing Your Web Server

A practical 8-chapter course for home server administrators

CONTENTS

osztromok.com

Chapter 1 — HTTPS with Let's Encrypt

HTTPS encrypts the connection between a visitor and your server. Without it, anyone on the same network can read usernames, passwords, cookies, and session tokens in plain text. Browsers mark plain HTTP sites as "Not Secure." Search engines rank HTTPS sites higher. Let's Encrypt provides free, trusted certificates — and with the DNS-01 challenge, you can get one even when port 80 is blocked by your ISP.

> **What this chapter covers:** What TLS actually does and why it matters beyond the padlock icon. How certificate authorities and the trust chain work. Let's Encrypt overview and the ACME protocol. Why HTTP-01 challenge fails behind Virgin Media and how DNS-01 solves it. Getting a certificate using certbot with the Cloudflare DNS plugin. Configuring Apache for HTTPS. Automatic renewal via systemd timer. Cloudflare Origin Certificates as an alternative. Choosing the right Cloudflare SSL/TLS mode. Testing with openssl and SSL Labs.

Why HTTPS Matters

The padlock icon is the visible part. What's actually happening underneath is more important:

- **Encryption:** all data exchanged between browser and server is encrypted. On plain HTTP, a coffee shop Wi-Fi operator, your ISP, or anyone running a packet sniffer can read login credentials and session cookies verbatim.
- **Authentication:** TLS proves the server is who it claims to be. Without it, a man-in-the-middle attack can serve a fake version of your site to users — they'd have no way to tell.
- **Integrity:** TLS detects tampering. ISPs are known to inject ads into plain HTTP pages. TLS prevents this.
- **HTTP/2 and HTTP/3:** both require HTTPS. Without it you're stuck on HTTP/1.1 — significantly slower for sites with many assets.
- **Browser warnings:** Chrome and Firefox show "Not Secure" in the address bar for HTTP pages with forms. Visitors see this and leave.
- **Cloudflare Tunnel already gives you HTTPS for visitors** — but understanding TLS and having your own certificate enables Full (Strict) SSL mode and works outside the tunnel for SSH, direct connections, and local network access.

> **You already have HTTPS for visitors via Cloudflare Tunnel.** Cloudflare handles the TLS for the visitor-to-Cloudflare leg. This chapter teaches you how to extend that to the full end-to-end path — and gives you a real certificate for when you need one (local network, Full Strict SSL mode, future direct connections).

How TLS Works (the short version)

```
TLS in 4 steps:

1. Client says hello, lists supported cipher suites
2. Server sends its certificate (signed by a CA the browser trusts)
3. Browser verifies: cert matches the domain, signed by trusted CA, not expired
4. Both sides derive a shared symmetric key — all further traffic is encrypted with it

Certificate trust chain:

Let's Encrypt Root CA (pre-installed in your browser/OS)
  └── Let's Encrypt Intermediate CA (signed by root)
        └── Your cert for osztromok.com (signed by intermediate)

Browser checks: is the chain valid? Is the leaf cert for the domain I requested?
Is it within the validity period? If all yes → padlock shown, connection encrypted.
```

The key concept is the **trust chain**. Your certificate is trusted because it's signed by Let's Encrypt's intermediate CA, which is trusted because it's signed by a root CA, which is pre-installed in operating systems and browsers. Self-signed certificates break this chain — the browser has no pre-installed trust for them and shows a scary warning instead of a padlock.

Let's Encrypt certificates are **Domain Validation (DV)** — they prove you control the domain, nothing more. That's sufficient for encryption. Extended Validation (EV) certs that show the company name in the address bar are largely obsolete in modern browsers.

Let's Encrypt and the ACME Protocol

Let's Encrypt is a non-profit Certificate Authority that issues free DV certificates. The process is automated via the ACME protocol — you run `certbot`, it talks to Let's Encrypt's servers, proves you control the domain, and receives a signed certificate. No forms, no waiting, no credit card.

- **Certificate validity:** 90 days. Short by design — forces automation and limits damage if a cert is compromised. Auto-renewal handles this transparently.
- **Rate limits:** 5 certificates per domain per week. For a personal site this is never a problem.
- **Wildcard certificates:** supported — `*.osztromok.com` covers all subdomains. Requires DNS-01 challenge.
- **Cost:** free, forever. Funded by Mozilla, Google, EFF, and others.

The Challenge Problem — Why HTTP-01 Doesn't Work Here

To prove you control a domain, Let's Encrypt uses a "challenge." You have two main options — and one of them is blocked by Virgin Media.

HTTP-01 Challenge — NOT available

Let's Encrypt places a file at `http://osztromok.com/.well-known/acme-challenge/TOKEN` and then fetches it from outside to verify it's really there.

Why it fails here:

- Requires port 80 to be reachable from the internet
- Virgin Media Hub 3.0 blocks all inbound connections on port 80
- Cloudflare Tunnel could theoretically help, but certbot and tunnel timing is unreliable
- Bottom line: if you can't receive an inbound HTTP request, HTTP-01 is unavailable

DNS-01 Challenge — This is what we use

Let's Encrypt gives you a token and tells you to place it as a TXT record in your DNS: `_acme-challenge.osztromok.com TXT "TOKEN"`. It then queries DNS to verify the token.

Why it works here:

- No inbound HTTP required — purely DNS-based
- Works regardless of ISP port blocking
- DNS is on Cloudflare → certbot-dns-cloudflare plugin creates/deletes the TXT record automatically
- Supports wildcard certificates (HTTP-01 cannot)
- Entire process is automated once configured

Installing certbot and the Cloudflare Plugin

```
# Install certbot and the Cloudflare DNS plugin
$ sudo apt update
$ sudo apt install certbot python3-certbot-dns-cloudflare -y

# Verify both are installed
$ certbot --version
certbot 2.x.x
$ python3 -c "import certbot_dns_cloudflare; print('plugin OK')"
plugin OK
```

Create a Cloudflare API token for DNS editing

In the Cloudflare dashboard → **My Profile → API Tokens → Create Token**. Use the **Edit zone DNS** template:

- **Permissions:** Zone → DNS → Edit
- **Zone Resources:** Include → Specific zone → osztromok.com
- Copy the token — shown only once.

```
# Create a secure credentials file for certbot to read
$ sudo mkdir -p /etc/letsencrypt
$ sudo nano /etc/letsencrypt/cloudflare.ini
```

```
# /etc/letsencrypt/cloudflare.ini
# Cloudflare API credentials for certbot DNS-01 challenge
```

```
dns_cloudflare_api_token = your-cloudflare-api-token-here
```

```
# Lock down the file — it contains a credential
$ sudo chmod 600 /etc/letsencrypt/cloudflare.ini
$ sudo chown root:root /etc/letsencrypt/cloudflare.ini
```

Getting the Certificate — Complete Walkthrough

FULL WALKTHROUGH · CERTBOT DNS-01 VIA CLOUDFLARE

Obtain a Let's Encrypt certificate for osztromok.com and www.osztromok.com using DNS-01 challenge — no open ports required.

1 **Request the certificate.** The `--dns-cloudflare-propagation-seconds 60` flag gives DNS time to propagate the TXT record before Let's Encrypt checks it.

```
$ sudo certbot certonly \
  --dns-cloudflare \
  --dns-cloudflare-credentials /etc/letsencrypt/cloudflare.ini \
  --dns-cloudflare-propagation-seconds 60 \
  -d osztromok.com \
  -d www.osztromok.com \
  --email emubantam@gmail.com \
  --agree-tos \
  --no-eff-email
Saving debug log to /var/log/letsencrypt/letsencrypt.log
Requesting a certificate for osztromok.com and www.osztromok.com
Waiting 60 seconds for DNS changes to propagate
Successfully received certificate.
Certificate is saved at: /etc/letsencrypt/live/osztromok.com/fullchain.pem
Key is saved at:         /etc/letsencrypt/live/osztromok.com/privkey.pem
This certificate expires on 2026-09-14.
These files will be updated when the certificate renews.
```

2 **Check what certbot created.**

```
$ sudo ls -la /etc/letsencrypt/live/osztromok.com/
cert.pem       ← your certificate (leaf only)
chain.pem      ← intermediate CA certificate
fullchain.pem  ← cert + chain combined (use this in Apache)
privkey.pem    ← your private key (keep secret, never share)

$ sudo certbot certificates
Found the following certs:
  Certificate Name: osztromok.com
    Domains: osztromok.com www.osztromok.com
    Expiry Date: 2026-09-14 (VALID: 89 days)
    Certificate Path: /etc/letsencrypt/live/osztromok.com/fullchain.pem
    Private Key Path: /etc/letsencrypt/live/osztromok.com/privkey.pem
```

3 **Enable mod_ssl in Apache.**

```
$ sudo a2enmod ssl
$ sudo a2enmod headers   # needed for HSTS headers in Chapter 2
$ sudo apache2ctl configtest && sudo systemctl restart apache2
# restart (not reload) after enabling new modules
```

4 **Add an HTTPS VirtualHost to the Apache config.** Open `/etc/apache2/sites-available/osztromok.com.conf` and add the `*:443` block below the existing `*:80` block:

```
<VirtualHost *:443>
    ServerName    osztromok.com
    ServerAlias   www.osztromok.com
    DocumentRoot  /var/www/osztromok.com/public_html

    <Directory /var/www/osztromok.com/public_html>
        Options       FollowSymLinks
        AllowOverride All
        Require       all granted
    </Directory>

    SSLEngine             on
    SSLCertificateFile    /etc/letsencrypt/live/osztromok.com/fullchain.pem
    SSLCertificateKeyFile /etc/letsencrypt/live/osztromok.com/privkey.pem

    ErrorLog  ${APACHE_LOG_DIR}/osztromok_error.log
    CustomLog ${APACHE_LOG_DIR}/osztromok_access.log combined
</VirtualHost>
```

5 **Validate and reload.**

```
$ sudo apache2ctl configtest
Syntax OK
$ sudo systemctl reload apache2
```

6 **Test locally with openssl.**

```
$ echo | openssl s_client -connect localhost:443 -servername osztromok.com 2>/dev/null | grep -E "subject|is
subject=CN=osztromok.com
issuer=C=US, O=Let's Encrypt, CN=R10
Verify return code: 0 (ok)
# Verify return code 0 = valid cert, trusted chain
```

If using the Cloudflare Tunnel, also switch Cloudflare SSL/TLS mode to **Full (Strict)** now that Apache has a valid cert (see SSL below).

The certificate files in /etc/letsencrypt/live/ are symlinks to the latest version in /etc/letsencrypt/archive/. When certbot renews, it updates the archive and the symlinks automatically. Apache reads the symlinks, so it picks up the renewed cert on the next reload — no config changes needed.

Wildcard Certificates

A wildcard certificate (`*.osztromok.com`) covers all first-level subdomains with a single cert. Every new subdomain you add is automatically covered — no re-issuing needed. DNS-01 is the only challenge type that supports wildcards.

```
# Request a wildcard cert (covers *.osztromok.com AND osztromok.com as a SAN)
$ sudo certbot certonly \
  --dns-cloudflare \
  --dns-cloudflare-credentials /etc/letsencrypt/cloudflare.ini \
  --dns-cloudflare-propagation-seconds 60 \
  -d osztromok.com \
  -d "*.osztromok.com" \
  --email emubantam@gmail.com \
  --agree-tos \
  --no-eff-email
# Note: *.osztromok.com does NOT cover the apex (osztromok.com) by itself.
# Always include -d osztromok.com alongside -d "*.osztromok.com"
```

> **Wildcards only cover one level.** `*.osztromok.com` covers `blog.osztromok.com` but not `www.blog.osztromok.com`. For deeper subdomains, request them explicitly. In practice, for a personal site, a wildcard cert for `*.osztromok.com` covers every subdomain you'll ever create.

Automatic Renewal

Let's Encrypt certificates expire after 90 days. Certbot installs a systemd timer automatically when it's installed on Debian/Ubuntu — you just need to verify it's working.

```
# Check that the certbot renewal timer is active
$ sudo systemctl status certbot.timer
● certbot.timer - Run certbot twice daily
     Active: active (waiting)
    Trigger: Mon 2026-06-15 12:00:00 BST; 14h left

# Test that renewal would work (--dry-run doesn't actually issue a cert)
$ sudo certbot renew --dry-run
Simulating renewal of an existing certificate for osztromok.com
Congratulations, all simulated renewals succeeded:
  /etc/letsencrypt/live/osztromok.com/fullchain.pem (success)

# View renewal configuration certbot created
$ sudo cat /etc/letsencrypt/renewal/osztromok.com.conf
[renewalparams]
```

```
authenticator = dns-cloudflare
dns_cloudflare_credentials = /etc/letsencrypt/cloudflare.ini
dns_cloudflare_propagation_seconds = 60
```

Reload Apache after renewal

Certbot renews the cert files but doesn't reload Apache automatically. Apache continues serving the old cert from memory until it's reloaded. Add a deploy hook to reload Apache whenever a cert is renewed:

```
$ sudo nano /etc/letsencrypt/renewal-hooks/deploy/reload-apache.sh
```

```
#!/bin/bash
# /etc/letsencrypt/renewal-hooks/deploy/reload-apache.sh
# Runs automatically after every successful cert renewal
systemctl reload apache2
```

```
$ sudo chmod +x /etc/letsencrypt/renewal-hooks/deploy/reload-apache.sh
# Test it runs without error
$ sudo /etc/letsencrypt/renewal-hooks/deploy/reload-apache.sh
# If no output and exit code 0, it's working
```

> **Certbot renews certs when they're within 30 days of expiry.** With a 90-day cert and checks twice daily, you'll get renewal attempts starting at day 60. If both DNS and Apache are working, renewal is completely hands-off. You can monitor with `sudo certbot certificates` anytime to check the current expiry date.

Cloudflare Origin Certificates — The Alternative

Cloudflare offers their own free certificates called **Origin Certificates**. Unlike Let's Encrypt, these are issued by Cloudflare's own CA — they're only trusted when traffic goes through Cloudflare's proxy. Browsers connecting directly to your server's IP would see an "untrusted" warning.

FEATURE	LET'S ENCRYPT	CLOUDFLARE ORIGIN CERT
Validity period	90 days (auto-renewed)	Up to 15 years (set once, forget it)
Trusted by browsers	Yes — universally	Only via Cloudflare proxy
Works for direct connections	Yes	No — not browser-trusted
Works for local network HTTPS	Yes	No
Requires renewal automation	Yes (every 90 days)	No (15-year validity)

FEATURE	LET'S ENCRYPT	CLOUDFLARE ORIGIN CERT
Enables Full (Strict) SSL mode	Yes	Yes
Works with Cloudflare Tunnel	Yes	Yes

Creating a Cloudflare Origin Certificate

Go to Cloudflare dashboard → SSL/TLS → Origin Server → Create Certificate. Choose RSA key, validity 15 years, and add the hostnames (`osztromok.com`, `*.osztromok.com`). Cloudflare shows you the cert and key — copy them to your server.

```
# Save the cert and key Cloudflare gives you to your server
$ sudo nano /etc/ssl/cloudflare/osztromok.com.pem    # paste the certificate here
$ sudo nano /etc/ssl/cloudflare/osztromok.com.key    # paste the private key here
$ sudo chmod 600 /etc/ssl/cloudflare/osztromok.com.key
# Then point Apache SSLCertificateFile and SSLCertificateKeyFile at these paths
```

> **Recommendation:** use **Let's Encrypt** if you want flexibility (local network HTTPS, future direct connections, full browser trust regardless of Cloudflare). Use **Cloudflare Origin Cert** if you're 100% committed to routing everything through Cloudflare and want to avoid renewal automation entirely. For a learning setup, Let's Encrypt is the better choice.

Cloudflare SSL/TLS Modes — Which to Choose

Go to Cloudflare dashboard → SSL/TLS → Overview. This setting controls how Cloudflare communicates with your origin server (the leg *behind* Cloudflare, not the visitor-to-Cloudflare leg which is always HTTPS).

Off	HTTP only. Visitors see no padlock. **Never use this.**
Flexible	**Visitor → Cloudflare:** HTTPS (padlock shown). **Cloudflare → your server:** plain HTTP. No certificate required on your server. This is the default for tunnel setups. **The tunnel connection itself is encrypted regardless**, so this is acceptable — but it's not end-to-end encryption. Mixed content issues can arise.
Full	Both legs use HTTPS. Your server needs a certificate — even a self-signed one. Cloudflare doesn't verify the cert is valid or matches the domain. Better than Flexible but still doesn't guarantee authenticity of the cert.
Full (Strict)	**Recommended once you have a cert.** Both legs use HTTPS and Cloudflare verifies your certificate is valid and issued by a trusted CA (or Cloudflare Origin CA). End-to-end encryption with authenticity. Requires a real Let's Encrypt or Cloudflare Origin cert on your Apache server.

```
Flexible (before this chapter):
```

```
Visitor ──HTTPS──▶ Cloudflare ──HTTP──▶ cloudflared ──HTTP──▶ Apache
(encrypted)        (proxied)    (plain)   (tunnel, but            (no cert)
                                          internally plain)

Full (Strict) (after this chapter):

Visitor ──HTTPS──▶ Cloudflare ──HTTPS──▶ cloudflared ──HTTPS──▶ Apache
(encrypted)        (proxied)    (encrypted) (tunnel, TLS          (with LE cert)
                                           end-to-end)
```

> **Don't switch to Full (Strict) before the cert is installed.** If you enable Full (Strict) and Apache doesn't have a valid certificate, Cloudflare will refuse to connect and visitors will see a 526 error. Always install and verify the cert first (using the openssl test command above), then change the Cloudflare SSL mode.

Testing Your Certificate

```
# Test the certificate from the server itself
$ echo | openssl s_client -connect localhost:443 -servername osztromok.com 2>/dev/null \
  | openssl x509 -noout -dates -subject -issuer
notBefore=Jun 14 21:00:00 2026 GMT
notAfter=Sep 14 21:00:00 2026 GMT
subject=CN=osztromok.com
issuer=C=US, O=Let's Encrypt, CN=R10

# Check expiry in human-readable days
$ echo | openssl s_client -connect localhost:443 -servername osztromok.com 2>/dev/null \
  | openssl x509 -noout -enddate
notAfter=Sep 14 21:00:00 2026 GMT

# Verify the full chain is correct (no "unable to verify" errors)
$ curl -vI https://localhost --resolve localhost:443:127.0.0.1 2>&1 | grep -E "SSL|TLS|cert|i
* TLSv1.3 (OUT), TLS handshake
* Server certificate:
*   subject: CN=osztromok.com
*   issuer: C=US; O=Let's Encrypt; CN=R10
*   SSL certificate verify ok.
```

> **SSL Labs test:** once your domain is publicly accessible (via Cloudflare Tunnel + Full Strict mode), visit `ssllabs.com/ssltest` and enter `osztromok.com`. SSL Labs gives a grade (A, A+, B) and flags any configuration weaknesses — weak cipher suites, missing HSTS, certificate chain issues. Aim for an A or A+ rating.

Troubleshooting

certbot: "DNS problem: NXDOMAIN looking up TXT for _acme-challenge.osztromok.com"

The TXT record wasn't visible in DNS when Let's Encrypt checked. Causes: (1) DNS not yet on Cloudflare — the Cloudflare plugin creates TXT records in Cloudflare, but if your nameservers still point to IONOS those records won't be seen. Verify with `dig NS osztromok.com +short` — should return Cloudflare nameservers. (2) Propagation delay — increase `--dns-cloudflare-propagation-seconds` to 120. (3) Wrong API token scope — check the token has Zone → DNS → Edit permission for osztromok.com.

certbot: "Error: Could not read credentials from /etc/letsencrypt/cloudflare.ini"

Permissions on the credentials file are wrong. Run: `sudo chmod 600 /etc/letsencrypt/cloudflare.ini && sudo chown root:root /etc/letsencrypt/cloudflare.ini`. The file must be readable only by root — certbot refuses to use a credentials file that's world-readable as a security measure.

Apache: "SSLCertificateFile: file '/etc/letsencrypt/live/...' does not exist or is empty"

The path in the Apache config doesn't match where certbot saved the cert. Run `sudo certbot certificates` to see the exact paths, then update `SSLCertificateFile` and `SSLCertificateKeyFile` in the vhost config accordingly. Also check that the directory is accessible: `sudo ls /etc/letsencrypt/live/osztromok.com/`.

Visitors see 526 (Cloudflare: Invalid SSL) after switching to Full (Strict)

Cloudflare can reach your Apache but the certificate isn't valid or isn't trusted. Check: (1) is mod_ssl enabled? `sudo apache2ctl -M | grep ssl` — should show `ssl_module`. (2) Did Apache restart (not just reload) after enabling mod_ssl? (3) Run the openssl test: `echo | openssl s_client -connect localhost:443 -servername osztromok.com 2>/dev/null | grep Verify` — should say `Verify return code: 0 (ok)`. (4) Temporarily switch Cloudflare back to Flexible to restore access, then diagnose.

Renewal dry-run fails: "Challenge failed for domain ..."

The renewal config at `/etc/letsencrypt/renewal/osztromok.com.conf` stores the original certbot options. If the Cloudflare API token changed or expired, update it in `/etc/letsencrypt/cloudflare.ini` and run `sudo certbot renew --dry-run` again. Also check that the API token hasn't been deleted in the Cloudflare dashboard (My Profile → API Tokens).

Quick Reference — Chapter 1

COMMAND	PURPOSE
`sudo certbot certonly --dns-cloudflare ...`	Obtain a cert using Cloudflare DNS-01 challenge — no open ports needed
`sudo certbot certificates`	List all certs managed

COMMAND	PURPOSE
	by certbot with their expiry dates
`sudo certbot renew --dry-run`	Test renewal without actually issuing a new cert — run this monthly
`sudo systemctl status certbot.timer`	Verify the auto-renewal timer is active and see when it next fires
`sudo a2enmod ssl headers`	Enable SSL and headers modules in Apache (restart required after)
`sudo apache2ctl -M | grep ssl`	Confirm ssl_module is loaded in the running Apache
`echo | openssl s_client -connect localhost:443 -servername osztromok.com 2>/dev/null | grep Verify`	Quick cert validity check — "Verify return code: 0 (ok)" = good

FILE / PATH	PURPOSE
`/etc/letsencrypt/live/osztromok.com/fullchain.pem`	Use in Apache SSLCertificateFile — cert + intermediate chain
`/etc/letsencrypt/live/osztromok.com/privkey.pem`	Use in Apache SSLCertificateKeyFile — never share or commit this
`/etc/letsencrypt/cloudflare.ini`	Cloudflare API token for certbot DNS-01 — chmod 600, owner root
`/etc/letsencrypt/renewal/osztromok.com.conf`	Renewal configuration — stores the original certbot options
`/etc/letsencrypt/renewal-hooks/deploy/`	Scripts here run after every successful renewal — use for Apache reload
`/var/log/letsencrypt/letsencrypt.log`	Full certbot log — first place to look when something goes wrong

Chapter 2 — Enforcing HTTPS

Installing a certificate (Chapter 1) makes HTTPS available — but it doesn't stop users arriving over plain HTTP. A visitor who types `osztromok.com` without the `https://`, follows an old bookmark, or clicks a link from an older site will make their first request over HTTP. That first request is unencrypted, and could include cookies or credentials. Enforcing HTTPS means redirecting HTTP traffic to HTTPS before any content is served — and then using HSTS to tell browsers to never attempt HTTP again.

> **What this chapter covers:** Why a certificate alone is not enough. Three Apache redirect methods and when to use each. Cloudflare's "Always Use HTTPS" edge setting. The redirect loop trap with Cloudflare Flexible SSL mode — and how to avoid it. HSTS header parameters (max-age, includeSubDomains, preload). Configuring HSTS in Apache with mod_headers. The HSTS preload list — commitment and consequences. Testing redirects and headers with curl. Troubleshooting ERR_TOO_MANY_REDIRECTS and missing HSTS headers.

Why a Certificate Alone Is Not Enough

```
Without enforcement — both paths exist:

Visitor types http://osztromok.com
     |
     ▼ HTTP request — unencrypted, credentials visible to ISP/network
Apache serves the page over HTTP   ← dangerous

Visitor types https://osztromok.com
     |
     ▼ HTTPS — TLS handshake, certificate verified, encrypted
Apache serves the page over HTTPS  ← safe

With enforcement:

Visitor types http://osztromok.com
     |
     ▼ 301 Permanent Redirect → https://osztromok.com
     |
```

```
        ▼ HTTPS — encrypted connection established
  Apache serves the page              ← always safe
```

The redirect is a **301 Permanent Redirect**, not a 302 Temporary. The permanent status tells browsers and search engines that HTTP is never the right choice for this domain. Search engines consolidate SEO signals to the HTTPS version, and browsers cache the redirect so repeat visitors skip the HTTP round-trip entirely.

Three Ways to Redirect HTTP to HTTPS in Apache

Method 1 — Redirect directive (simplest)

One-liner inside the *:80 VirtualHost. Clean, readable, no modules beyond what's already loaded.

- Uses mod_alias (built in)
- Redirects all URLs under /
- Preserves path and query string
- **Best for:** most cases — this is the recommended approach

Method 2 — mod_rewrite

More powerful — can add conditions (e.g. skip redirect for certbot challenge paths, or only redirect specific domains).

- Requires mod_rewrite (usually already enabled)
- Conditional redirects possible
- Handles X-Forwarded-Proto checks
- **Best for:** complex redirect logic or Cloudflare Flexible SSL mode

Method 3 — RedirectMatch

Like Redirect but uses regex — redirect only URLs matching a specific pattern, leave others alone.

- Regex-based URL matching
- Good for partial enforcement
- Less common in practice
- **Best for:** migrating specific paths to HTTPS before full enforcement

Cloudflare "Always Use HTTPS"

Handled at Cloudflare's edge — HTTP requests are redirected to HTTPS before they ever reach your server. Zero Apache config.

- Cloudflare dashboard → SSL/TLS → Edge Certificates
- Eliminates redirect loop risk entirely
- Works in Flexible, Full, or Full Strict mode
- **Best for:** simplest setup when all traffic goes through Cloudflare

Method 1 — Redirect directive

```
# /etc/apache2/sites-available/osztromok.com.conf
# The *:80 block now only redirects — no DocumentRoot needed

<VirtualHost *:80>
    ServerName   osztromok.com
    ServerAlias  www.osztromok.com
```

```
    # 301 permanent redirect — whole site, preserve path
    Redirect permanent / https://osztromok.com/
</VirtualHost>

<VirtualHost *:443>
    ServerName   osztromok.com
    ServerAlias  www.osztromok.com
    DocumentRoot /var/www/osztromok.com/public_html

    <Directory /var/www/osztromok.com/public_html>
        Options       FollowSymLinks
        AllowOverride All
        Require       all granted
    </Directory>

    SSLEngine             on
    SSLCertificateFile    /etc/letsencrypt/live/osztromok.com/fullchain.pem
    SSLCertificateKeyFile /etc/letsencrypt/live/osztromok.com/privkey.pem

    ErrorLog  ${APACHE_LOG_DIR}/osztromok_error.log
    CustomLog ${APACHE_LOG_DIR}/osztromok_access.log combined
</VirtualHost>
```

Method 2 — mod_rewrite (for Flexible SSL or conditional logic)

```
<VirtualHost *:80>
    ServerName  osztromok.com
    ServerAlias www.osztromok.com

    RewriteEngine On

    # Check if the original request (before Cloudflare) was HTTP.
    # X-Forwarded-Proto is set by Cloudflare to "https" when the
    # visitor used HTTPS — even when Cloudflare sends HTTP to Apache.
    RewriteCond %{HTTP:X-Forwarded-Proto} !https
    RewriteCond %{HTTPS} off
    RewriteRule ^ https://%{HTTP_HOST}%{REQUEST_URI} [L,R=301]
</VirtualHost>
```

> **Which method to use?** If you followed Chapter 1 and switched Cloudflare to Full (Strict) SSL mode, use Method 1 — Apache receives genuine HTTPS connections from Cloudflare, and the simple `Redirect permanent` works without any loop risk. If you're on Flexible SSL mode (Cloudflare sends HTTP to Apache), use Method 2 with the `X-Forwarded-Proto` check, or better yet — use Cloudflare's "Always Use HTTPS" edge setting instead.

Cloudflare "Always Use HTTPS"

This is the cleanest solution if all traffic goes through Cloudflare. The redirect happens at Cloudflare's edge — the HTTP request never touches your server. Enable it in the Cloudflare dashboard:

SSL/TLS → Edge Certificates → Always Use HTTPS → toggle On

When enabled, a visitor who requests `http://osztromok.com` gets a 301 redirect to `https://osztromok.com` from Cloudflare's servers directly. Your Apache config doesn't need a redirect at all — the *:80 VirtualHost can simply return a 404 or be left as-is.

> **Use either Cloudflare "Always Use HTTPS" OR an Apache redirect — not both.** Enabling both creates a situation where Cloudflare intercepts the redirect and tries again, Apache redirects again, and the cycle can confuse some clients. Pick one layer to own the redirect. Cloudflare edge is preferred; Apache redirect is the fallback for non-Cloudflare traffic.

The Redirect Loop Trap

This is the most common mistake when adding HTTPS redirects behind Cloudflare, and it produces the browser error `ERR_TOO_MANY_REDIRECTS`.

```
The loop — happens with Flexible SSL + Apache redirect:

Visitor → http://osztromok.com
     |
     ▼ Cloudflare (Flexible mode): forwards as HTTP to Apache
     |
Apache sees HTTP → redirects to https://osztromok.com (301)
     |
```

```
      ▼ Cloudflare receives the redirect, fetches https://osztromok.com
        but forwards it as HTTP to Apache (because Flexible mode)
      |
Apache sees HTTP → redirects to https://osztromok.com (301) again
      |
      ↻ Loop — browser gives up after 10 iterations: ERR_TOO_MANY_REDIRECTS

_____________________________________________________________________

Solutions (pick one):

A. Switch Cloudflare to Full (Strict) SSL mode
   → Cloudflare sends HTTPS to Apache → Apache sees HTTPS → no redirect
   (this is what Chapter 1 set up)

B. Use Cloudflare "Always Use HTTPS" instead of Apache redirect
   → HTTP is handled at Cloudflare edge → Apache never sees HTTP → no loop

C. Use mod_rewrite with X-Forwarded-Proto check
   → Apache only redirects when X-Forwarded-Proto is NOT https
   → If Cloudflare set it to "https", Apache passes through → no loop
```

HSTS — HTTP Strict Transport Security

A redirect upgrades the *current* HTTP request to HTTPS. HSTS goes further: it tells the browser to never attempt HTTP for this domain again — for a specified duration. The browser enforces HTTPS locally, before making any network request, so there's no HTTP exposure even on the first visit after the cached instruction is set.

```
How HSTS works:

First visit (HTTPS):
Browser → https://osztromok.com
Apache responds with content + header:
  Strict-Transport-Security: max-age=31536000; includeSubDomains

Browser caches: "osztromok.com requires HTTPS for 31536000 seconds (1 year)"

Any future visit within that year:
User types "osztromok.com" or clicks "http://osztromok.com"
```

```
  Browser converts to https:// INTERNALLY, before sending any request
  → No HTTP request ever leaves the browser for this domain
```

This eliminates the "HTTPS downgrade attack" window — the brief moment during the first HTTP request before the redirect fires, when an attacker on the same network could intercept or modify the traffic.

HSTS Header Parameters

`max-age=31536000`	**How long the browser enforces HTTPS** (in seconds). 31536000 = 1 year. Start with a shorter value (e.g. `max-age=86400` = 1 day) while testing, then increase to a year once you're confident HTTPS is solid. Once set to a long duration, reducing it requires waiting for all cached instructions to expire.
`includeSubDomains`	**Apply HSTS to all subdomains.** If set, the browser also enforces HTTPS for `blog.osztromok.com`, `api.osztromok.com`, etc. Only add this when *every* subdomain has a valid HTTPS certificate. If any subdomain has no cert, adding this will make that subdomain inaccessible in browsers that have cached the HSTS instruction.
`preload`	**Request inclusion in the browser preload list.** The HSTS preload list is hardcoded into Chrome, Firefox, Safari, and Edge — so even a *first-ever* visit to your domain is forced to HTTPS. Requires submitting to hstspreload.org. See the commitment note below before adding this.

Configuring HSTS in Apache

HSTS is set via the `Strict-Transport-Security` response header, added by mod_headers. It must be in the ***:443 VirtualHost only** — browsers ignore this header on plain HTTP responses (rightly so, since an attacker on HTTP could strip it).

```
# /etc/apache2/sites-available/osztromok.com.conf — complete with redirect + HS

<VirtualHost *:80>
    ServerName  osztromok.com
    ServerAlias www.osztromok.com
    # Redirect everything to HTTPS permanently
    Redirect    permanent / https://osztromok.com/
```

```
</VirtualHost>

<VirtualHost *:443>
    ServerName   osztromok.com
    ServerAlias  www.osztromok.com
    DocumentRoot /var/www/osztromok.com/public_html

    <Directory /var/www/osztromok.com/public_html>
        Options       FollowSymLinks
        AllowOverride All
        Require       all granted
    </Directory>

    SSLEngine             on
    SSLCertificateFile    /etc/letsencrypt/live/osztromok.com/fullchain.pem
    SSLCertificateKeyFile /etc/letsencrypt/live/osztromok.com/privkey.pem

    # ── HSTS — tell browsers to always use HTTPS ──────────────────
    # Start with max-age=86400 (1 day) while testing.
    # Increase to 31536000 (1 year) once HTTPS is confirmed stable.
    # Add includeSubDomains only when ALL subdomains have valid certs.
    Header always set Strict-Transport-Security "max-age=86400"

    ErrorLog  ${APACHE_LOG_DIR}/osztromok_error.log
    CustomLog ${APACHE_LOG_DIR}/osztromok_access.log combined
</VirtualHost>
```

```
# Verify mod_headers is loaded (enabled in Chapter 1)
$ sudo apache2ctl -M | grep headers
headers_module (shared)   ← must be present

# If not loaded:
$ sudo a2enmod headers && sudo systemctl restart apache2

# Validate config, then reload
$ sudo apache2ctl configtest && sudo systemctl reload apache2
```

Progressively hardening HSTS

Don't jump straight to a 1-year HSTS with preload — if anything breaks, you're locked in for a very long time. Work through this progression over weeks:

STAGE	HEADER VALUE	WHEN TO MOVE TO NEXT STAGE
Testing	`max-age=86400` (1 day)	HTTPS stable, no cert issues after 1 week
Moderate	`max-age=604800` (1 week)	All subdomains have certs; no issues after 2 weeks
Production	`max-age=31536000; includeSubDomains`	Confident everything will stay on HTTPS long-term
Preload-ready	`max-age=31536000; includeSubDomains; preload`	Submit to hstspreload.org — permanent commitment

HSTS Preload List

The HSTS preload list is a database maintained by Google (and used by Chrome, Firefox, Safari, Edge) that hardcodes domains as HTTPS-only directly in the browser binary. Even a brand new browser that has never visited your site will refuse to make HTTP requests to a preloaded domain.

Before submitting to the preload list — understand the commitment

Preloading is **effectively permanent**. Once your domain is in the list and shipped in browser releases:

- Removal takes 6–12 months minimum — browsers don't update daily, and cached binaries persist.
- Every subdomain of osztromok.com must serve HTTPS — no exceptions. Any subdomain without a valid cert becomes completely inaccessible in browsers that have cached the preload entry.
- You cannot run **anything** over plain HTTP on the domain — ever. Not even temporary test pages, admin panels, or internal tools.
- If your cert expires and renewal fails, your entire domain goes dark until it's fixed — browsers won't even show a "proceed anyway" option.

For a personal/learning site: production-level HSTS (max-age=31536000; includeSubDomains) is sufficient. Preload is for large-scale, mission-critical deployments where even first-visit HTTPS matters.

Complete Setup Walkthrough

FULL WALKTHROUGH · HTTPS ENFORCEMENT + HSTS

Enable Cloudflare "Always Use HTTPS", configure Apache redirect as a defence-in-depth fallback, and add HSTS — starting conservative and verifying at each step.

1. **Enable "Always Use HTTPS" in Cloudflare (primary enforcement).** Go to **SSL/TLS → Edge Certificates → Always Use HTTPS → toggle On**. This handles the redirect at Cloudflare's edge — the most reliable layer since Cloudflare processes all traffic before it reaches your server.

2. **Confirm Cloudflare SSL mode is Full (Strict) (from Chapter 1).** Go to **SSL/TLS → Overview → Full (Strict)**. This ensures that when Cloudflare contacts your Apache server (via the tunnel), it uses HTTPS and verifies your Let's Encrypt certificate — so the redirect in Apache won't loop.

3. **Update the Apache *:80 VirtualHost to redirect (defence in depth).** Edit `/etc/apache2/sites-available/osztromok.com.conf`. Replace the *:80 block content with:

```
    ServerName  osztromok.com
    ServerAlias www.osztromok.com
    Redirect    permanent / https://osztromok.com/
```

   This ensures any direct HTTP connection to Apache (bypassing Cloudflare) is still redirected.

4. **Add a conservative HSTS header to the *:443 VirtualHost.** Inside the `<VirtualHost *:443>` block, add:

```
Header always set Strict-Transport-Security "max-age=86400"
```

   Starting with 1 day — easy to recover from if anything breaks.

5. **Validate config and reload Apache.**

```
$ sudo apache2ctl configtest
Syntax OK
$ sudo systemctl reload apache2
```

6. **Test the redirect and verify the HSTS header.**

```
# Test redirect locally (bypassing Cloudflare)
$ curl -I -H "Host: osztromok.com" http://localhost
HTTP/1.1 301 Moved Permanently
Location: https://osztromok.com/

# Test HSTS header is present on HTTPS response
$ curl -sk -H "Host: osztromok.com" https://localhost -o /dev/null -D - | grep -i strict
Strict-Transport-Security: max-age=86400
```

```
# From outside — follow redirects and show final URL
$ curl -sIL http://osztromok.com | grep -E "HTTP|Location|Strict"
HTTP/1.1 301 Moved Permanently
Location: https://osztromok.com/
HTTP/2 200
strict-transport-security: max-age=86400
```

7 **After 1 week with no issues, increase max-age and add includeSubDomains.** Edit the HSTS header in the Apache config:

```
Header always set Strict-Transport-Security "max-age=31536000; includeSubDomains"
```

Only add `includeSubDomains` when all your subdomains have valid HTTPS certificates. Reload Apache after the change.

HSTS takes effect from the first HTTPS response the browser sees. Once a browser has cached it, even clearing browser history won't remove the HSTS policy — the user would need to go to `chrome://net-internals/#hsts` (Chrome) and delete the entry manually. This is by design.

X-Forwarded-Proto — Reading the Real Protocol

When traffic passes through a proxy (Cloudflare, a load balancer, another reverse proxy), the proxy adds an `X-Forwarded-Proto` header to tell the backend what protocol the original client used — even if the proxy changed it en route.

```
Cloudflare Flexible SSL mode — what Apache sees vs what the visitor did:

Visitor → HTTPS → Cloudflare → HTTP → Apache
Apache sees: request over HTTP
But: X-Forwarded-Proto: https  ← Cloudflare adds this

Visitor → HTTP → Cloudflare → HTTP → Apache
Apache sees: request over HTTP
And: X-Forwarded-Proto: http  ← no HTTPS involved

mod_rewrite can use this to redirect only genuine HTTP visits:
RewriteCond %{HTTP:X-Forwarded-Proto} =http → redirect to HTTPS
RewriteCond %{HTTP:X-Forwarded-Proto} =https → pass through (already HTTPS)
```

This is why the mod_rewrite method (Method 2) works correctly with Flexible SSL — it checks `X-Forwarded-Proto` before deciding whether to redirect. The simple `Redirect permanent` doesn't do this check, which is why it causes loops in Flexible mode.

> **Only trust X-Forwarded-Proto from sources you control.** Any client can add a fake `X-Forwarded-Proto: https` header to bypass your redirect if Apache accepts it from arbitrary sources. Behind Cloudflare, this is mitigated because Cloudflare strips or overwrites the header before forwarding. In general, only rely on proxy headers from known, trusted proxies.

Troubleshooting

ERR_TOO_MANY_REDIRECTS — browser can't load the page at all

Classic redirect loop. Check: (1) Is Cloudflare SSL mode set to Flexible? If so, Apache receives HTTP from Cloudflare, and a simple `Redirect permanent` sends it back to HTTPS — which Cloudflare again sends as HTTP to Apache. Fix: switch to Full (Strict) SSL mode (Chapter 1) or use mod_rewrite with `X-Forwarded-Proto` check. (2) Is "Always Use HTTPS" enabled in Cloudflare AND you have an Apache redirect? Disable one. (3) Test without Cloudflare: `curl -I -H "Host: osztromok.com" http://localhost` — if this also loops, the issue is in Apache alone (check for duplicate redirect rules).

HSTS header not appearing in the response

Three common causes: (1) mod_headers not loaded — check with `sudo apache2ctl -M | grep headers`. If missing, run `sudo a2enmod headers && sudo systemctl restart apache2`. (2) Header is in the *:80 block instead of *:443 — browsers ignore HSTS on HTTP responses. (3) Apache was reloaded (not restarted) after enabling a module — modules require restart, not just reload. Check: `curl -sk https://localhost -D - -o /dev/null | grep -i strict`.

Redirect works but drops the path — /blog/post-1 becomes just /

The `Redirect permanent /` directive preserves the path — `http://osztromok.com/blog/post-1` redirects to `https://osztromok.com/blog/post-1`. If the path is being dropped, check for a secondary redirect somewhere (Cloudflare Page Rules, a .htaccess RewriteRule, or a duplicate Redirect directive pointing at a specific path). Run `curl -sIL http://osztromok.com/blog/post-1` and examine each redirect in the chain.

Browser shows "Not Secure" on some pages even after enabling HTTPS

Mixed content — the page loads over HTTPS but references resources (images, scripts, stylesheets) via plain `http://` URLs. The page itself is encrypted but those resources are not, so the browser downgrades the security indicator. Fix: update all asset URLs to use `https://` or protocol-relative `//`. Apache can help with `mod_substitute` or a blanket Content-Security-Policy upgrade (covered in Chapter 6).

HSTS is set but a subdomain is now inaccessible

You added `includeSubDomains` before the subdomain had a valid HTTPS certificate. The browser is enforcing HTTPS on the subdomain and the cert is missing or invalid. Fix: immediately get a certificate for the subdomain (certbot with `-d subdomain.osztromok.com`) and configure Apache for HTTPS on that subdomain. To test the fix, clear HSTS for the domain: in Chrome, visit `chrome://net-internals/#hsts`, enter the subdomain under "Delete domain security policies," then retest.

Quick Reference — Chapter 2

COMMAND / CHECK	PURPOSE
`curl -I -H "Host: osztromok.com" http://localhost`	Test redirect locally — should return 301 with Location: https://
`curl -sIL http://osztromok.com | grep -E "HTTP|Location|Strict"`	Follow the full redirect chain and check for HSTS in the final response
`curl -sk https://localhost -D - -o /dev/null | grep -i strict`	Verify HSTS header is in the HTTPS response from Apache directly
`sudo apache2ctl -M | grep headers`	Confirm mod_headers is loaded (required for HSTS)
`sudo a2enmod headers && sudo systemctl restart apache2`	Enable mod_headers — restart required (not just reload)
`chrome://net-internals/#hsts`	Inspect or delete HSTS entries in Chrome — useful for testing

HSTS VALUE	MEANING	WHEN TO USE
`max-age=86400`	Enforce HTTPS for 1 day	Testing phase — easy to recover
`max-age=604800`	Enforce HTTPS for 1 week	Early production — gaining confidence
`max-age=31536000`	Enforce HTTPS for 1 year	Stable production — HTTPS confirmed solid
`; includeSubDomains`	Apply rule to all subdomains	Only when all subdomains have valid certs

HSTS VALUE	MEANING	WHEN TO USE
`; preload`	Request browser preload inclusion	After submitting to hstspreload.org — permanent

APPROACH	BEST FOR	REDIRECT LOOP RISK?
`Cloudflare "Always Use HTTPS"`	All traffic through Cloudflare	None — handled at edge before Apache
`Apache Redirect permanent (Method 1)`	Full (Strict) SSL mode	None — Apache receives genuine HTTPS
`mod_rewrite + X-Forwarded-Proto (Method 2)`	Flexible SSL mode or complex logic	None — checks the original protocol first
`Apache Redirect + Flexible SSL (wrong combo)`	Avoid	Yes — ERR_TOO_MANY_REDIRECTS

Chapter 3 — Firewall with UFW

A firewall is the first thing that meets any incoming network connection. It decides, before any application sees the traffic, whether that connection should be allowed at all. Even though Cloudflare Tunnel handles your web traffic without opening inbound ports, a server-side firewall protects against attacks from within the local network, mis-configured services, and future changes to your setup. UFW — the Uncomplicated Firewall — provides a clean interface to iptables that's simple enough to set up correctly in minutes.

> **What this chapter covers:** What a firewall does and why it matters even behind a Cloudflare Tunnel. UFW's default-deny-inbound policy. Checking what's listening before enabling. The safe setup order (SSH allow before enable — or you lock yourself out). Application profiles. Port-specific and source-specific rules. Rate limiting with `ufw limit`. Managing rules with numbered list, delete, and insert. UFW logging. IPv6 behaviour. The minimal ruleset for a Cloudflare Tunnel server. What to do if you lock yourself out.

Why a Firewall Matters Even with Cloudflare Tunnel

```
Without a firewall — all these attack surfaces exist:

Internet (blocked by ISP)        ──X──▶ Your server
Cloudflare Tunnel (outbound)     ──✓──▶ localhost:80  (controlled)
Local network (home devices)     ──?──▶ Your server:22, :80, :3306, :anything
Compromised home device          ──?──▶ Your server (all ports open)
Accidental service bind          ──?──▶ 0.0.0.0:8080 (any network)

With UFW (default deny inbound):

Internet                         ──X──▶ blocked at firewall
Cloudflare Tunnel (outbound)     ──✓──▶ outbound rules allow this
Local network                    ──?──▶ only port 22 (SSH) allowed
Compromised home device          ──X──▶ port 3306 (MySQL) denied — can't touch i
Accidental service on :8080      ──X──▶ denied — firewall catches it
```

The key insight: the Cloudflare Tunnel uses an *outbound* connection. UFW's default policy allows all outbound traffic, so the tunnel works perfectly with a strict inbound firewall. Web requests arrive from Cloudflare to `localhost:80` — the loopback interface, which is treated separately from network interfaces. Meanwhile, the firewall blocks anything unexpected trying to reach the server from other machines on your home network.

- **MySQL (port 3306)** is bound to `127.0.0.1` by default — good. But if it were ever misconfigured to `0.0.0.0`, UFW would block access from outside the server.
- **New services you install** often bind to all interfaces. UFW denies them until you explicitly allow them.
- **Defence in depth** — the ISP blocks inbound, the router has its own firewall, and UFW adds a third layer on the server itself. Any one layer failing alone doesn't expose the server.

UFW Default Policies

Incoming — DENY (default)

All inbound connections are dropped unless you explicitly add an allow rule. This is the "default-deny" principle — start from zero and only open what you need.

Effect: any port not listed in your allow rules is silently blocked. Attackers scanning for open ports get no response.

Outgoing — ALLOW (default)

All outbound connections from the server are allowed by default. This means your server can: reach package repositories, contact Cloudflare for the tunnel, make DNS queries, send emails, etc.

Effect: you don't need to add rules for outbound traffic. The tunnel just works.

> **UFW also has a FORWARD policy** (traffic being routed through the server to another destination). Default is DENY. Leave it that way unless you're setting up a router or VPN server.

Check What's Listening Before You Start

Before enabling UFW, know what services are running and which ports they use. This prevents accidentally blocking something you need, and helps you identify services that shouldn't be exposed.

```
# List all listening ports with the process that owns them
$ sudo ss -tlnp
State   Recv-Q  Send-Q  Local Address:Port  Peer Address:Port  Process
LISTEN  0       128     0.0.0.0:22          0.0.0.0:*          users:(("sshd",...
LISTEN  0       511     *:80                *:*                users:(("apache2",
LISTEN  0       70      127.0.0.1:3306      0.0.0.0:*          users:(("mysqld",.
LISTEN  0       128     [::]:22             [::]:*             users:(("sshd",...
LISTEN  0       511     [::]:80             [::]:*             users:(("apache2",

# Key observations from this output:
# - sshd on 0.0.0.0:22 and [::]:22 — IPv4 and IPv6, all interfaces
# - apache2 on *:80 — all interfaces (needs to accept tunnel connections from lo
# - mysqld on 127.0.0.1:3306 — loopback ONLY, already safe
# UFW rules to write: allow 22, optionally allow 80, deny everything else
```

> **Any service listening on `0.0.0.0` or `*` (not `127.0.0.1`) is reachable from the network.** Without UFW, that means reachable by anyone on your home network (or internet, if port forwarding exists). With UFW, they're blocked unless you add an allow rule. This is exactly the protection you want.

Safe Setup Order — SSH First, Then Enable

The cardinal rule: allow SSH before enabling UFW

If you enable UFW with its default-deny policy *without first allowing SSH*, your SSH session will be immediately cut off and you won't be able to reconnect. Recovery requires:

1. Physical access to the server (plug in a keyboard and monitor)
2. Or a remote console via your hosting provider (not applicable for a home server)
3. Log in locally, run `sudo ufw allow ssh && sudo ufw reload`

Always allow SSH first, then enable. Never reverse this order.

SETUP WALKTHROUGH · SAFE UFW ACTIVATION

Install UFW, add the SSH rule, then enable — in the correct order to avoid locking yourself out.

1 **Install UFW (already included on Ubuntu/Debian, but verify).**

```
$ sudo apt install ufw -y
$ sudo ufw status
Status: inactive   ← UFW installed but not yet running — this is correct at this stage
```

2 **Set default policies explicitly.**

```
$ sudo ufw default deny incoming
Default incoming policy changed to 'deny'
$ sudo ufw default allow outgoing
Default outgoing policy changed to 'allow'
```

3 **Allow SSH — do this BEFORE enabling.**

```
$ sudo ufw allow ssh
# "ssh" is a UFW application profile — equivalent to "allow 22/tcp"
Rules updated
Rules updated (v6)

# If your SSH runs on a non-standard port (e.g. 2222), use the port number:
$ sudo ufw allow 2222/tcp
```

Not sure which port SSH is on? Check: `sudo ss -tlnp | grep ssh`

4 **Enable UFW.**

```
$ sudo ufw enable
Command may disrupt existing ssh connections. Proceed with operation (y|n)? y
Firewall is now active and enabled on system startup
```

Your existing SSH session continues because the SSH allow rule was already in place.

5 **Verify status — confirm SSH is allowed and everything else is denied.**

```
$ sudo ufw status verbose
Status: active
Logging: on (low)
Default: deny (incoming), allow (outgoing), disabled (routed)
New profiles: skip

To                         Action      From
--                         ------      ----
22/tcp                     ALLOW IN    Anywhere
22/tcp (v6)                ALLOW IN    Anywhere (v6)
```

6 **Open a new SSH session from a different terminal to verify you're not locked out.**

```
$ ssh philip@webserver   # from another terminal/window
philip@webserver:~$   ← successful login — firewall is not blocking SSH
```

Only close the original session once this confirms SSH still works. If the new session fails, you still have the original session to diagnose and fix.

UFW rules persist across reboots automatically. Once enabled, the firewall will be active every time the server starts — no need to re-enable manually.

Which Ports to Open for Your Setup

The right ruleset depends on how traffic reaches your server. With Cloudflare Tunnel, web traffic arrives via `localhost` (the loopback interface) — not from the network — so ports 80 and 443 don't need to be open in UFW for the tunnel to work.

`22 / tcp`

SSH

Always open. Without this you can't administer the server remotely. Use `ufw allow ssh` or `ufw allow 22/tcp`.

`80 / tcp`

HTTP (web)

Not needed for tunnel. Cloudflare Tunnel delivers traffic to localhost — UFW doesn't see it. Only open if you need direct HTTP access from the local network.

`443 / tcp`

HTTPS (web)

Not needed for tunnel. Same as port 80 — tunnel bypasses UFW's network interface rules. Only open for direct HTTPS connections.

`3306 / tcp`

MySQL

Keep closed. MySQL should only be accessible from `127.0.0.1` (already configured by default). Never open this to the network.

`everything else`

All other ports

Denied by default. Any new service that accidentally binds to a public interface is blocked automatically until you explicitly allow it.

> **The minimal tunnel ruleset — just one rule:** `sudo ufw allow ssh`. That's genuinely all you need if all public-facing traffic goes through Cloudflare Tunnel. Everything else is denied. The tunnel (outbound) is unaffected. This is one of the cleanest possible firewall configurations.

UFW Rule Syntax

```
sudo ufw allow ssh ← allow SSH using the named application profile

sudo ufw allow 80/tcp ← allow HTTP on TCP only (not UDP)

sudo ufw allow in from 192.168.1.0/24 to any port 80
← allow HTTP from local network only

sudo ufw deny from 45.33.32.156 ← block a specific IP address entirely

sudo ufw limit ssh ← allow SSH but rate-limit: ban after 6 attempts in 30s

sudo ufw delete allow 80/tcp ← remove a rule
```

```
# Source-restricted rules — allow port 80 only from home network
$ sudo ufw allow from 192.168.1.0/24 to any port 80 proto tcp
# Replace 192.168.1.0/24 with your actual subnet (check: ip route | grep 'src')

# Find your local subnet
$ ip route | grep proto
192.168.1.0/24 dev eth0 proto kernel scope link src 192.168.1.100
# Subnet is 192.168.1.0/24

# Allow HTTP and HTTPS from local network only (useful for internal testing)
$ sudo ufw allow from 192.168.1.0/24 to any port 80 proto tcp
$ sudo ufw allow from 192.168.1.0/24 to any port 443 proto tcp
```

Application Profiles

UFW ships with named application profiles for common services — shortcuts that allow the right ports without needing to remember numbers.

```
# List all available application profiles
$ sudo ufw app list
Available applications:
  Apache
  Apache Full
  Apache Secure
  OpenSSH
```

```
# What does each Apache profile open?
$ sudo ufw app info 'Apache Full'
Profile: Apache Full
Title: Web Server (HTTP,HTTPS)
Description: Apache V2 is the next generation of the omnipresent Apache web serv
Ports: 80,443/tcp

$ sudo ufw app info 'OpenSSH'
Ports: 22/tcp

# The three Apache profiles:
# Apache        → port 80 only (HTTP)
# Apache Secure → port 443 only (HTTPS)
# Apache Full   → ports 80 and 443 (both)
```

> **For a Cloudflare Tunnel setup, avoid `ufw allow 'Apache Full'`.** Opening ports 80 and 443 to all sources is unnecessary — the tunnel delivers traffic via localhost. At most, allow those ports restricted to your local subnet. The only externally-visible benefit is that attackers can see your server has something running on 80/443, which is information you'd rather not give away.

Rate Limiting with `ufw limit`

`ufw limit` allows connections to a port but automatically blocks an IP that makes 6 or more connection attempts within 30 seconds. It's a quick defence against SSH brute-force attacks without the full weight of fail2ban (covered in Chapter 4).

```
# Replace "allow ssh" with "limit ssh" for rate-limited SSH
$ sudo ufw delete allow ssh      # remove the plain allow rule first
$ sudo ufw limit ssh              # add the rate-limited version

# Verify — should now show LIMIT IN instead of ALLOW IN
$ sudo ufw status
To             Action      From
--             ------      ----
22/tcp         LIMIT IN    Anywhere
22/tcp (v6)    LIMIT IN    Anywhere (v6)
```

> **ufw limit vs fail2ban:** `ufw limit` is simple — fixed threshold (6 attempts / 30 seconds), IPv4 and IPv6, built into UFW. fail2ban (Chapter 4) is more powerful — configurable thresholds, persistent bans, monitors log files, and covers Apache and other services too. Use both: limit for quick UFW-level protection now, fail2ban for deeper coverage later.

Managing Rules

View rules with numbers

```
$ sudo ufw status numbered
Status: active

     To                         Action      From
     --                         ------      ----
[ 1] 22/tcp                     LIMIT IN    Anywhere
[ 2] 80/tcp                     ALLOW IN    192.168.1.0/24
[ 3] 22/tcp (v6)                LIMIT IN    Anywhere (v6)
[ 4] 80/tcp (v6)                ALLOW IN    Anywhere (v6)
```

Delete a rule by number

```
# Delete rule number 4 (the IPv6 port 80 allow in the example above)
$ sudo ufw delete 4
Deleting:
 allow in on any port 80 proto tcp from Anywhere (v6)
Proceed with operation (y|n)? y
Rule deleted (v6)

# Or delete by specifying the rule itself
$ sudo ufw delete allow 80/tcp
```

Block a specific IP address

```
# Block all traffic from a specific IP (e.g. a persistent scanner you spotted ir
$ sudo ufw deny from 45.33.32.156
Rule added
```

```
# Block a whole subnet
$ sudo ufw deny from 45.33.32.0/24

# Deny rules should be inserted BEFORE allow rules to take effect
# Insert at position 1 (top of list) to guarantee it's checked first
$ sudo ufw insert 1 deny from 45.33.32.156
```

Reset UFW (nuclear option — removes all rules)

```
$ sudo ufw reset
Resetting all rules to installed defaults. This may disrupt existing ssh
connections. Proceed with operation (y|n)? y
# This disables UFW and removes all custom rules.
# Use when you want to start completely fresh.
# Remember to re-add SSH allow rule before re-enabling!
```

UFW Logging

```
# Enable logging (off by default in some installations)
$ sudo ufw logging on
# Logging levels: off / low / medium / high / full
# "low" logs blocked packets; "medium" also logs allowed. Start with low.
$ sudo ufw logging medium

# UFW logs go to /var/log/ufw.log (and also syslog)
$ sudo tail -f /var/log/ufw.log
Jun 15 03:22:14 webserver kernel: [UFW BLOCK] IN=eth0 OUT= MAC=... SRC=45.33.32.
# UFW BLOCK = connection was denied.
# SRC = attacking IP, DPT = destination port (3306 = MySQL scan — blocked)

Jun 15 03:25:01 webserver kernel: [UFW ALLOW] IN=eth0 OUT= MAC=... SRC=192.168.1
# UFW ALLOW = connection permitted (your SSH from another device on the LAN)
```

> **What to look for in UFW logs:** Repeated BLOCK entries on port 22 from diverse IPs = SSH brute force scan (this is normal and expected on any internet-connected server — fail2ban in Chapter 4 handles this automatically). Repeated BLOCK entries on port 3306 = MySQL scan

> (blocked correctly). Any UFW BLOCK on a port you didn't know was being probed is worth noting.

IPv6 Support

UFW handles IPv4 and IPv6 with the same rules by default — when you `allow ssh`, UFW automatically creates both an IPv4 and IPv6 rule. Verify this is enabled:

```
$ sudo cat /etc/default/ufw | grep IPV6
IPV6=yes   ← this should be yes

# If IPV6=no, edit the file and restart UFW
$ sudo nano /etc/default/ufw   # set IPV6=yes
$ sudo ufw disable && sudo ufw enable
```

With `IPV6=yes`, every rule you add automatically gets an IPv6 counterpart. When you run `sudo ufw status`, you'll see rules listed twice — once for IPv4 and once as `(v6)` for IPv6. This is correct behaviour.

The Complete Ruleset for This Setup

```
# ── Complete UFW setup for a Cloudflare Tunnel home server ──────────

# 1. Set default policies
$ sudo ufw default deny incoming
$ sudo ufw default allow outgoing

# 2. Allow SSH with rate limiting
$ sudo ufw limit ssh

# 3. Allow HTTP/HTTPS from local network only (for internal access/testing)
#    Skip this if you only ever access the site via the Cloudflare Tunnel
$ sudo ufw allow from 192.168.1.0/24 to any port 80 proto tcp
$ sudo ufw allow from 192.168.1.0/24 to any port 443 proto tcp

# 4. Enable the firewall
$ sudo ufw enable
```

```
# 5. Verify
$ sudo ufw status verbose
Status: active
Default: deny (incoming), allow (outgoing), disabled (routed)
To                         Action      From
--                         ------      ----
22/tcp                     LIMIT IN    Anywhere
80/tcp                     ALLOW IN    192.168.1.0/24
443/tcp                    ALLOW IN    192.168.1.0/24
22/tcp (v6)                LIMIT IN    Anywhere (v6)
```

> **Test the Cloudflare Tunnel still works after enabling UFW.** Visit `https://osztromok.com` from outside — if it loads, the tunnel is unaffected (as expected, since it uses outbound connections). Then try SSHing from another machine on the network — should succeed on port 22.

Troubleshooting

SSH dropped immediately after ufw enable — locked out

You enabled UFW without first allowing SSH. Recovery options: (1) Physical access — connect a keyboard and monitor to the server, log in locally, run `sudo ufw allow ssh && sudo ufw reload`. (2) If you have another active SSH session open in a different terminal, use it immediately — run the allow command before it times out. (3) If the server is running in a VM with a hypervisor console, use the console to log in and fix the rule. Going forward: **always allow SSH first, enable second.**

Cloudflare Tunnel stops working after enabling UFW

Unexpected — the tunnel uses outbound connections which UFW's default `allow outgoing` policy covers. Check: (1) Did you accidentally change the outgoing default to deny? Run `sudo ufw status verbose` and look for `Default: ... allow (outgoing)`. (2) Is cloudflared running? `sudo systemctl status cloudflared`. (3) The tunnel connects to Cloudflare's servers on port 443 (outbound) — this should always be allowed by the default outgoing policy.

Local network access to the website is blocked after enabling UFW

You haven't added the local network allow rule. Run: `ip route | grep proto kernel` to get your subnet (e.g. `192.168.1.0/24`), then: `sudo ufw allow from 192.168.1.0/24 to any port 80 proto tcp` and `sudo ufw allow from 192.168.1.0/24 to any port 443 proto tcp`. Verify with `sudo ufw status` that the rules were added, then reload: `sudo ufw reload`.

ufw status shows "inactive" even though I ran ufw enable

UFW requires a reboot to activate on some systems, or the enable command failed. Check: `sudo systemctl status ufw` — if it shows failed, there may be a conflict with iptables. Run `sudo ufw enable` again and look for error output. Also check: `sudo iptables -L` — if iptables rules already exist from another firewall tool (firewalld, nftables), they may conflict.

My IP got rate-limited by ufw limit and I can't SSH in

`ufw limit` bans your IP for 30 seconds after 6 connection attempts. Wait a minute and try again — the ban is temporary. If you keep triggering it, you may have an SSH client that retries aggressively. Switch to `ufw allow ssh` while debugging, then switch back to `ufw limit ssh` once the client behaviour is corrected. fail2ban (Chapter 4) has a whitelist mechanism for trusted IPs that UFW's built-in rate limiting lacks.

Quick Reference — Chapter 3

COMMAND	PURPOSE
`sudo ufw status verbose`	Show active rules, default policies, and logging level
`sudo ufw status numbered`	Show rules with numbers — needed for targeted deletion
`sudo ufw default deny incoming`	Block all inbound by default — set before enabling
`sudo ufw default allow outgoing`	Allow all outbound — required for tunnel, DNS, apt updates
`sudo ufw allow ssh`	Allow SSH — must run BEFORE ufw enable
`sudo ufw limit ssh`	Allow SSH with rate limiting (6 attempts / 30 seconds)
`sudo ufw allow from 192.168.1.0/24 to any port 80 proto tcp`	Allow port 80 from local network only
`sudo ufw deny from IP`	Block all traffic from a specific IP address
`sudo ufw insert 1 deny from IP`	Insert deny rule at top of list (evaluated first)

COMMAND	PURPOSE
`sudo ufw delete NUMBER`	Delete rule by number (from ufw status numbered)
`sudo ufw logging medium`	Enable medium logging — blocked and allowed packets
`sudo ufw reload`	Reload rules without disabling — safe to run while active
`sudo ufw reset`	Wipe all rules and disable — use with caution
`sudo ufw app list`	List available named application profiles

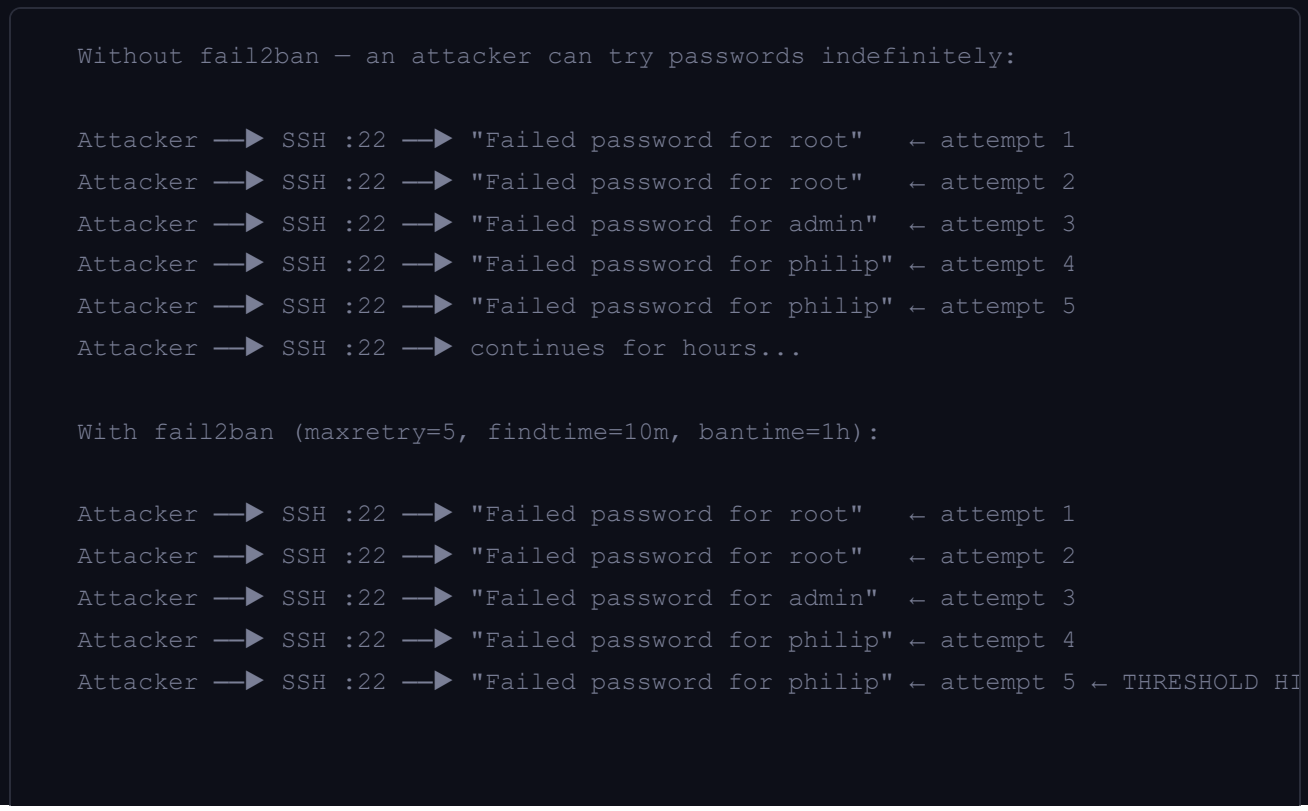
PORT / SERVICE	RULE FOR CLOUDFLARE TUNNEL SETUP
`22/tcp (SSH)`	ufw limit ssh — always open, rate-limited
`80/tcp (HTTP)`	Optional: allow from local subnet only — not needed for tunnel
`443/tcp (HTTPS)`	Optional: allow from local subnet only — not needed for tunnel
`3306/tcp (MySQL)`	Never open — keep bound to 127.0.0.1 only
`Everything else`	Denied by default — don't open unless you have a specific reason

Chapter 4 — fail2ban

Every server exposed to the internet is scanned continuously. Automated bots try common passwords against SSH around the clock, and web crawlers probe Apache for known vulnerabilities. UFW (Chapter 3) blocked the network ports you don't need. fail2ban handles the ports you do need open — specifically SSH — by watching logs and automatically banning IPs that show attack patterns. A failed SSH attempt is not a problem; six failed attempts from the same IP in thirty seconds is.

> **What this chapter covers:** How fail2ban works (jail → filter → action model). Installing fail2ban and creating `jail.local` (the safe override file). DEFAULT section settings — bantime, findtime, maxretry, ignoreip. SSH jail configuration. The Cloudflare Tunnel IP problem: why Apache jails will ban Cloudflare's IPs and break your site without `mod_remoteip`. Fixing that with mod_remoteip. Apache jails once logging is correct. Managing bans with `fail2ban-client`. Persistent increasing bans. Monitoring logs. Testing filters.

What fail2ban Does



```
fail2ban ──▶ iptables ──▶ DROP all from 45.33.32.156  ← IP banned for 1 hour
Attacker ──▶ SSH :22 ──▶ no response (connection dropped silently)
```

fail2ban doesn't change the password; it just makes a brute-force attack too slow to succeed. With a one-hour ban and only 5 attempts before triggering it, an attacker would need *years* to try a typical wordlist — and each ban resets their progress.

The Jail → Filter → Action Model

COMPONENT

Jail

A named config block that ties everything together. Each jail watches one service (sshd, apache, etc.). The jail says which log file to watch, which filter to apply, and which action to take. You enable or disable jails in `jail.local`.

COMPONENT

Filter

A file of regex patterns in `/etc/fail2ban/filter.d/` that match malicious log lines. When Apache logs "File does not exist: /etc/passwd", the apache-noscript filter recognises the pattern. Filters are already provided for most common services.

COMPONENT

Action

What happens when maxretry is reached. The default action adds an iptables DROP rule for the offending IP. A UFW action is also available. Actions can optionally send email alerts.

COMPONENT

Backend

How fail2ban watches the log file. `systemd` reads from the systemd journal (faster, works without log files). `auto` picks the best available method. Use `backend = systemd` for SSH on modern Ubuntu/Debian.

```
fail2ban reads the log ──▶ runs it through the filter (regex)
                       ──▶ counts matches per source IP
                       ──▶ when count hits maxretry within findtime:
                                ──▶ calls the action (add iptables rule)
                                ──▶ logs the ban to /var/log/fail2ban.log
                                ──▶ starts bantime countdown
                       ──▶ when bantime expires: removes the iptables rule
```

Installing fail2ban

```
$ sudo apt install fail2ban -y

# fail2ban starts automatically after install. Verify:
$ sudo systemctl status fail2ban
● fail2ban.service - Fail2Ban Service
     Loaded: loaded (/lib/systemd/system/fail2ban.service)
     Active: active (running)

# By default, fail2ban installs but enables NO jails (nothing is being watched y
$ sudo fail2ban-client status
Status
|- Number of jail:      0
`- Jail list:

# You need to create jail.local to enable jails
```

> **Never edit** `/etc/fail2ban/jail.conf` — it's overwritten by package upgrades. Create `/etc/fail2ban/jail.local` instead. Any setting in `jail.local` overrides the same setting in `jail.conf`. You only need to include the settings you want to change — you don't have to copy the entire file.

Creating jail.local — The DEFAULT Section

The `[DEFAULT]` section sets global values that apply to every jail unless overridden at the jail level. Get these right first — they're the most important settings.

```
$ sudo nano /etc/fail2ban/jail.local
```

```
# /etc/fail2ban/jail.local

[DEFAULT]

# ── Timing ──────────────────────────────────────────────────
bantime  = 1h
findtime = 10m
maxretry = 5
```

```
# bantime  — how long to ban the offending IP.
#            Default is 10 minutes — far too short for real attacks.
#            1h is a sensible minimum; 24h is better for SSH.
#
# findtime — the sliding window for counting failures.
#            5 failures within 10 minutes triggers a ban.
#
# maxretry — number of failures that triggers a ban.
#            5 is reasonable. SSH key auth (Chapter 5) means
#            legitimate users never fail here at all.

# ── Whitelist — IPs that are NEVER banned ──────────────────────
ignoreip = 127.0.0.1/8 ::1 192.168.1.0/24

# IMPORTANT: Add your home network subnet here.
# If you mistype your password 6 times while troubleshooting at
# 2am, you don't want to lock yourself out.
# Replace 192.168.1.0/24 with your actual subnet (check: ip route)
# You can also add individual IPs: ignoreip = 127.0.0.1/8 ::1 192.168.1.0/24 203

# ── Action ─────────────────────────────────────────────────────
banaction = iptables-multiport

# iptables-multiport is the default — blocks the IP across all ports
# (not just the port that triggered the ban).
# This is usually what you want: block the whole IP, not just SSH.

# ── Backend (how fail2ban reads logs) ──────────────────────────
backend = auto
```

bantime

`= 1h`

How long the ban lasts. Accepts: **s** (seconds), **m** (minutes), **h** (hours), **d** (days). Default is 10m — too short. Start at 1h; escalate to 24h once you're confident.

findtime

`= 10m`

The sliding window. Failures older than this don't count toward the maxretry threshold. 10m is reasonable — an attacker making 1 attempt per minute is clearly malicious.

maxretry

`= 5`

Number of failures before a ban. 5 is the default — allows for a few typos. With SSH key auth (Ch.5), legitimate logins never fail at all, so you could reduce this to 3.

ignoreip

`= 127.0.0.1/8 ::1 LAN`

Space-separated list of IPs and CIDRs that are never banned, regardless of failure count. Always include loopback (127.0.0.1/8 ::1) and your home subnet.

The SSH Jail

Add the SSH jail directly in `jail.local` below the DEFAULT section. SSH is the most important jail to have — your SSH port is the only port genuinely exposed to the internet (via the Dynamic DNS A record from Chapter 7 of the hosting course).

```
# Append to jail.local — below the [DEFAULT] section

[sshd]
enabled  = true
port     = ssh
logpath  = %(sshd_log)s
backend  = %(sshd_backend)s
maxretry = 5
bantime  = 24h

# enabled  — activates this jail. Without this, the jail is ignored.
#
# port     — "ssh" expands to port 22. If you moved SSH to another
#            port (e.g. 2222), use: port = 2222
#
# logpath  — %(sshd_log)s is a variable defined in paths-common.conf.
#            On Ubuntu/Debian it resolves to the systemd journal
#            or /var/log/auth.log depending on the system.
#
# backend  — %(sshd_backend)s uses systemd journal on modern systems.
#
# bantime  — override the global 1h: SSH is critical, use 24h.
#            A 24-hour ban makes most brute-force attempts futile.
```

```
# Save jail.local, then restart fail2ban to apply changes
$ sudo systemctl restart fail2ban

# Verify the sshd jail is now active
$ sudo fail2ban-client status
Status
|- Number of jail:      1
`- Jail list:           sshd

# Detailed jail status
```

```
$ sudo fail2ban-client status sshd
Status for the jail: sshd
|- Filter
|  |- Currently failed: 2
|  |- Total failed:     47
|  `- Journal matches:  _SYSTEMD_UNIT=sshd.service + _COMM=sshd
|- Actions
   |- Currently banned: 1
   |- Total banned:     8
   `- Banned IP list:   45.33.32.156

# "Currently failed: 2" — two IPs have recent failures but haven't hit maxretry
# "Total banned: 8"     — 8 bans since fail2ban started (previous ones expired)
# "Banned IP list: ..." — this IP is actively banned right now
```

The Cloudflare Tunnel IP Problem — Read This Before Adding Apache Jails

Important: Apache jails will break your site without mod_remoteip

When a visitor hits `osztromok.com`, the request travels: **visitor → Cloudflare → cloudflared (on your server) → localhost → Apache**.

Apache sees the connection arriving from `127.0.0.1` (the loopback address, because cloudflared connects locally). It logs `127.0.0.1` as the client IP in its access log.

If you enable an Apache fail2ban jail without fixing this, fail2ban reads those Apache logs, sees repeated requests from `127.0.0.1`, and bans it. `127.0.0.1` is the loopback address — banning it breaks the Cloudflare Tunnel, which takes down your **entire website**.

The fix is **mod_remoteip**: it tells Apache to read the real visitor IP from the `CF-Connecting-IP` header that Cloudflare inserts into every request, and use that as the logged client address. Then Apache logs real IPs, and fail2ban bans real attackers.

Without mod_remoteip

Apache logs: `127.0.0.1` for every visitor.

fail2ban sees: all requests from `127.0.0.1`.

With mod_remoteip

Apache logs: real visitor IPs (from `CF-Connecting-IP`).

After N failures: bans `127.0.0.1`.

Result: **Cloudflare Tunnel broken, site offline.**

fail2ban sees: requests from `45.33.32.156`, etc.

After N failures: bans that real attacker IP.

Result: **Attacker blocked, tunnel unaffected.**

Fixing Apache Logging with mod_remoteip

mod_remoteip replaces the `REMOTE_ADDR` (the IP Apache thinks the connection came from) with the IP from a trusted header. For Cloudflare Tunnel, the real visitor IP is in the `CF-Connecting-IP` header.

SETUP WALKTHROUGH · MOD_REMOTEIP FOR CLOUDFLARE TUNNEL

Enable mod_remoteip and configure Apache to read real visitor IPs from Cloudflare's header.

1 **Enable mod_remoteip.**

```
$ sudo a2enmod remoteip
Enabling module remoteip.
To activate the new configuration, you need to run:
  service apache2 restart
```

2 **Create a mod_remoteip config file.** This tells Apache to trust the loopback address (where cloudflared connects) and use the `CF-Connecting-IP` header for the real IP.

```
$ sudo nano /etc/apache2/conf-available/remoteip.conf
```

```
# /etc/apache2/conf-available/remoteip.conf

RemoteIPHeader CF-Connecting-IP
RemoteIPTrustedProxy 127.0.0.1
RemoteIPTrustedProxy ::1

# RemoteIPHeader    — which header contains the real client IP.
#                     Cloudflare injects "CF-Connecting-IP" with the
#                     original visitor's IP into every request.
# RemoteIPTrustedProxy — only trust the header when the connection
#                     comes from these addresses (the loopback, where
#                     cloudflared runs). An untrusted source could
#                     forge this header — this restriction prevents that.
```

3 **Enable the config and restart Apache.**

```
$ sudo a2enconf remoteip
$ sudo systemctl restart apache2
```

4 **Verify Apache now logs real IPs.** Visit your website from outside, then check the access log.

```
$ sudo tail -5 /var/log/apache2/access.log
82.45.123.67 - - [15/Jun/2026:10:22:14 +0000] "GET / HTTP/1.1" 200 4821
# 82.45.123.67 is your real public IP — not 127.0.0.1.
# mod_remoteip is working correctly.

# If you still see 127.0.0.1, check the CF-Connecting-IP header is present:
$ curl -I https://osztromok.com
# Cloudflare should be injecting CF-Connecting-IP for all proxied requests.
# If it's missing, check your Cloudflare SSL/TLS mode (Full Strict is correct).
```

Once mod_remoteip is active, all Apache logs — access, error, and virtual host logs — will show real visitor IPs. This also improves analytics tools that read these logs.

Apache Jails

With mod_remoteip logging real IPs, Apache jails are now safe to enable. Add these to `jail.local` after the sshd section.

```
# Append to /etc/fail2ban/jail.local

[apache-auth]
enabled  = true
port     = http,https
logpath  = %(apache_error_log)s
maxretry = 6
bantime  = 1h

# Watches Apache error log for authentication failures.
# Triggers on things like: "user philip: authentication failure"
# (when using password-protected directories with Apache's own auth)


[apache-badbots]
enabled  = true
port     = http,https
logpath  = %(apache_access_log)s
bantime  = 24h
maxretry = 2

# Watches access log for known malicious bots and scanners.
# The badbots filter has a list of bad User-Agent strings.
# maxretry=2 because these should never legitimately visit: ban on second hit.


[apache-noscript]
enabled  = true
port     = http,https
logpath  = %(apache_error_log)s
maxretry = 6
bantime  = 1h

# Watches for requests to non-existent scripts — classic exploit scanners.
# Examples: GET /cgi-bin/test.cgi, GET /phpMyAdmin/setup.php
# Pattern: "File does not exist: /var/www/..." for script file types


[apache-overflows]
enabled  = true
```

```
port     = http,https
logpath  = %(apache_error_log)s
maxretry = 2
bantime  = 24h

# Watches for request URI/header overflow attempts — buffer overflow probes.
# Very low maxretry (2) because legitimate traffic never looks like this.
```

```
# Apply the new jails
$ sudo systemctl restart fail2ban

# Verify all jails are now active
$ sudo fail2ban-client status
Status
|- Number of jail:      5
`- Jail list:           apache-auth, apache-badbots, apache-noscript, apache-ove
```

> **Cloudflare's own bot protection vs. fail2ban:** Cloudflare already blocks most bots before they reach your server. The Apache jails catch anything that gets through. Think of Cloudflare's firewall as the outer wall and fail2ban as the inner guard — both doing their jobs at different layers.

Managing Bans with fail2ban-client

```
# ── Checking status ──────────────────────────────────────

# Overall status — how many jails, which ones
$ sudo fail2ban-client status

# Status of a specific jail — shows currently banned IPs and counts
$ sudo fail2ban-client status sshd

# ── Banning and unbanning ────────────────────────────────

# Manually ban an IP in the sshd jail (useful if you spot an attacker in logs)
$ sudo fail2ban-client set sshd banip 203.0.113.45
1   # "1" = one IP banned — success
```

```
# Unban an IP — use this if you accidentally ban yourself
$ sudo fail2ban-client set sshd unbanip 192.168.1.5
1

# Unban across ALL jails at once (nuclear unban — use with care)
$ sudo fail2ban-client unban 192.168.1.5

# ── Reloading config ──────────────────────────────────────

# Reload jail.local without a full restart (preserves active bans)
$ sudo fail2ban-client reload

# Reload a specific jail only
$ sudo fail2ban-client reload sshd

# Full restart (clears all active bans — use sparingly)
$ sudo systemctl restart fail2ban
```

> **If you ban yourself and can't SSH in:** You need physical access to the server (or a local terminal session that's already open). From there, run `sudo fail2ban-client set sshd unbanip YOUR-IP`. Check your current external IP with `curl api.ipify.org` from another device. To avoid this: always keep your home subnet in `ignoreip`.

Persistent Increasing Bans

By default, bans expire after `bantime`. An attacker who returns after each ban is simply working at a slower pace. fail2ban's `bantime.increment` feature makes the punishment escalate with each repeat offence — 1 hour becomes 2, then 4, then 8, until it's effectively permanent.

```
# Add to the [DEFAULT] section of jail.local

[DEFAULT]
# ... existing settings ...
bantime  = 1h
findtime = 10m
maxretry = 5
```

```
ignoreip = 127.0.0.1/8 ::1 192.168.1.0/24

# ── Escalating bans for repeat offenders ──────────────────────
bantime.increment   = true
bantime.multiplier  = 2
bantime.maxtime     = 4w    # cap at 4 weeks (not truly permanent, but effective
bantime.overalljails = true  # count bans across all jails, not just the one tha

# How this works:
# 1st ban: 1h
# 2nd ban (same IP returns): 2h
# 3rd ban:  4h
# 4th ban:  8h
# ...continuing until 4w cap
#
# bantime.overalljails=true means an IP banned from sshd and then
# apache-badbots has its total repeat count considered — it doesn't
# get a "fresh start" in the second jail.
```

> **Check the database of repeat offenders:** fail2ban stores ban history in a SQLite database at `/var/lib/fail2ban/fail2ban.sqlite3`. The `fail2ban-client status sshd` command doesn't show this history, but persistent bans use it internally to decide the escalated ban duration.

Monitoring fail2ban Logs

```
# Watch the fail2ban log in real time
$ sudo tail -f /var/log/fail2ban.log
2026-06-15 03:22:14,891 fail2ban.filter         [INFO]  [sshd] Found 45.33.32.15
2026-06-15 03:22:31,204 fail2ban.filter         [INFO]  [sshd] Found 45.33.32.15
2026-06-15 03:22:44,891 fail2ban.filter         [INFO]  [sshd] Found 45.33.32.15
2026-06-15 03:22:57,012 fail2ban.filter         [INFO]  [sshd] Found 45.33.32.15
2026-06-15 03:23:08,445 fail2ban.filter         [INFO]  [sshd] Found 45.33.32.15
2026-06-15 03:23:08,571 fail2ban.actions        [NOTICE] [sshd] Ban 45.33.32.156

# "Found" = a log line matched the filter (failure detected) — counting toward m
# "Ban"   = maxretry hit — iptables rule added, IP is now blocked

# To see unban events too (bans expiring):
$ sudo grep "Unban\|Ban\|Found" /var/log/fail2ban.log | tail -20
```

```
# Count total bans per jail since fail2ban started:
$ sudo grep "Ban " /var/log/fail2ban.log | awk '{print $6}' | sort | uniq -c | s
     23 [sshd]
      4 [apache-badbots]
      1 [apache-noscript]
# sshd is the busiest — expected, SSH is the main attack surface
```

Testing Filters

If fail2ban isn't triggering bans when you expect it to, test the filter directly against the log file to see what it matches — and what it misses.

```
# Test the sshd filter against the auth log — shows what it would detect
$ sudo fail2ban-regex /var/log/auth.log /etc/fail2ban/filter.d/sshd.conf
Running tests
=============
Use    failregex filter file : sshd, basedir: /etc/fail2ban
Use         log file : /var/log/auth.log
Use         encoding : UTF-8

Results
=======

Failregex: 49 total
|- #) [# of hits] regular expression
|  1) [24] Failed ...
|  2) [15] Invalid user ...
|  3) [10] Connection closed ...

Ignoreregex: 0 total

Date template hits:
|- [# of hits] date format
|  [194] {Day}/{Month}/{Year}:{Hour}:{Minute}:{Second} ...

Lines: 194 lines, 0 ignored, 49 matched, 145 missed
# 49 matched = 49 bad attempts detected.
# If this shows 0 matched, the filter isn't reading the log correctly —
```

```
# check if the log path in jail.local matches where auth.log actually lives.

# Test the Apache badbots filter against access log:
$ sudo fail2ban-regex /var/log/apache2/access.log /etc/fail2ban/filter.d/apache-
```

Complete jail.local File

```
# /etc/fail2ban/jail.local — complete recommended config for this setup

[DEFAULT]
bantime             = 1h
findtime            = 10m
maxretry            = 5
ignoreip            = 127.0.0.1/8 ::1 192.168.1.0/24
banaction           = iptables-multiport
backend             = auto
bantime.increment   = true
bantime.multiplier  = 2
bantime.maxtime     = 4w
bantime.overalljails = true

[sshd]
enabled  = true
port     = ssh
logpath  = %(sshd_log)s
backend  = %(sshd_backend)s
maxretry = 5
bantime  = 24h

[apache-auth]
enabled  = true
port     = http,https
logpath  = %(apache_error_log)s
maxretry = 6
bantime  = 1h

[apache-badbots]
enabled  = true
port     = http,https
logpath  = %(apache_access_log)s
```

```
bantime  = 24h
maxretry = 2

[apache-noscript]
enabled  = true
port     = http,https
logpath  = %(apache_error_log)s
maxretry = 6
bantime  = 1h

[apache-overflows]
enabled  = true
port     = http,https
logpath  = %(apache_error_log)s
maxretry = 2
bantime  = 24h
```

> **Reminder:** The Apache jails require `mod_remoteip` to be configured first (see the mod_remoteip section above). Without it, the jails will ban `127.0.0.1` and take down the Cloudflare Tunnel.

Troubleshooting

fail2ban won't start — "ERROR: Failed to parse" or similar

Syntax error in `jail.local`. Check: `sudo systemctl status fail2ban` for the error line, or `sudo journalctl -u fail2ban -n 50`. Common causes: missing space before the `=` in a key=value pair, mismatched section headers (typo in `[sshd]`), or invalid time format (use `1h` not `1 hour`).

Jail is active but IPs are not being banned

Run `sudo fail2ban-regex /path/to/logfile /etc/fail2ban/filter.d/filtername.conf` to confirm the filter matches the log. If it shows "0 matched", either the log path is wrong (check with `sudo fail2ban-client get sshd logpath`), or the log format doesn't match the filter regex. Also check: is the log file actually being written to? `sudo tail /var/log/auth.log` — if it's empty, SSH may be logging to the systemd journal instead (set `backend = systemd`).

Apache jails are triggering on 127.0.0.1 — site goes down

mod_remoteip is not configured, or the Apache log still shows 127.0.0.1. Check: `sudo tail /var/log/apache2/access.log` — do you see real IPs or 127.0.0.1? If still 127.0.0.1: verify `a2enconf remoteip` was run, the config file is correct, and Apache was restarted (not just reloaded). Temporarily disable the Apache jails while fixing: `sudo fail2ban-client stop apache-badbots`.

I banned myself and can't SSH in

You need a way to run commands on the server without SSH: (1) physical access — keyboard and monitor directly on the server. (2) If you have another active SSH session open, use it immediately. From there: find your external IP on another device (`curl api.ipify.org`) and run `sudo fail2ban-client set sshd unbanip YOUR.IP.HERE` . To prevent this in future: ensure your home subnet is in `ignoreip` in `jail.local` .

fail2ban-client status shows jail but no bans — is it working?

Probably yes — if your SSH is well-configured (key auth only, sensible password) and the scanning bots haven't found you yet, there may just not be enough failures to trigger a ban. Check if there are recent "Found" entries (failures detected but not yet at maxretry threshold): `sudo grep Found /var/log/fail2ban.log | tail -20` . If you see "Found" entries, it's working — just no ban threshold reached yet. You can test by deliberately failing SSH login from another IP.

Getting "ERROR jail 'xxx' does not exist" from fail2ban-client

The jail name in the command doesn't match the section header in `jail.local` . Jail names are case-sensitive. Run `sudo fail2ban-client status` to see the exact names of active jails, then use those names in subsequent commands.

Quick Reference — Chapter 4

COMMAND	PURPOSE
`sudo fail2ban-client status`	List active jails and how many are running
`sudo fail2ban-client status sshd`	Detailed status for the sshd jail — banned IPs, total counts
`sudo fail2ban-client set sshd unbanip IP`	Manually unban an IP from a specific jail
`sudo fail2ban-client unban IP`	Unban an IP from all jails at once
`sudo fail2ban-client set sshd banip IP`	Manually ban an IP in a specific jail
`sudo fail2ban-client reload`	Reload jail.local config without restarting (preserves active bans)
`sudo systemctl restart fail2ban`	Full restart — clears active bans, apply new config
`sudo tail -f /var/log/fail2ban.log`	Watch fail2ban activity in real time
`sudo fail2ban-regex LOG FILTER`	Test how many log lines a filter would match
`sudo journalctl -u fail2ban -n 50`	Check fail2ban's own startup/error logs

SETTING	RECOMMENDED VALUE	NOTES
bantime (SSH)	24h	Override the global in [sshd] — SSH attacks are persistent, make bans count
bantime (Apache)	1h–24h	Shorter for auth jails; 24h for badbots/overflows
findtime	10m	Default is fine — aggressive scanners hit the threshold in under a minute
maxretry	5	Allows some typos; with SSH key auth, legitimate users never fail
ignoreip	LAN + loopback	127.0.0.1/8 ::1 and your home subnet — never skip this
bantime.increment	true	Escalating bans punish repeat offenders automatically

Chapter 5 — SSH Hardening

UFW rate-limits SSH. fail2ban bans IPs after repeated failures. Both are important — but they're protecting against attacks on a password. The cleanest solution is to eliminate the password entirely and switch to SSH key authentication. With keys in place, brute-force becomes physically impossible: there's nothing to guess. After that, a handful of sshd_config changes close the remaining gaps.

> **What this chapter covers:** Why SSH keys are stronger than passwords and how the authentication works. Generating an Ed25519 key pair from Windows (PowerShell and WSL). Copying the public key to the server. Testing key auth — and the critical rule of testing before disabling passwords. Hardening `sshd_config` via a drop-in file. Validating the config with `sshd -t` before reloading. SSH agent for passphrase management. Multiple machines and multiple keys. The `~/.ssh/config` shortcut file. What to do if you lose your private key. Tightening fail2ban now that passwords are gone.

Why SSH Keys Beat Passwords

```
Password authentication — the attacker's job:

Attacker tries: "admin123"    → wrong
Attacker tries: "password"    → wrong
Attacker tries: "philip2026"  → wrong
...
Attacker tries: "P@ssw0rd!"   → MATCH (if your password is guessable)

With 5 tries/30s limit: slow, but not impossible. A strong random
password takes years. A weak one can fall in hours.

SSH key authentication — no password exists to guess:

Attacker tries to guess the private key → 2^256 possibilities (Ed25519)
At 1 trillion guesses/second: would take longer than the age of the universe.
There is no feasible brute-force attack against a properly generated key.
```

SSH keys work by mathematics, not secrets transmitted over the network. You hold a private key (never shared). The server holds your public key (can be posted publicly — it's useless without the matching private key). During login, the server challenges you to prove possession of the private key by signing a random message. The signature is verified against the public key. No password is transmitted. No secret crosses the wire.

- **Unguessable:** a 256-bit Ed25519 key has more possible values than atoms in the observable universe.
- **Nothing to steal remotely:** phishing for a password that doesn't exist doesn't work.
- **Passphrase optional:** you can add a passphrase to the private key for a second layer — even if someone copies your key file, they still need the passphrase to use it.
- **Per-device keys:** each machine you connect from gets its own key pair. Revoking access from one device (lost laptop) means removing one public key from the server — no password change needed on every device.

Choosing a Key Type

Ed25519

RECOMMENDED

Modern elliptic-curve algorithm. **Fast, small key files, high security.** Supported by all modern SSH clients and servers. Use this unless you need to connect to very old systems.

RSA 4096

ACCEPTABLE

The classic choice. 4096-bit RSA is still secure, but keys are larger and operations slower than Ed25519. **Use only if you need compatibility with older OpenSSH (< 6.5).**

RSA 2048

AVOID

Was the standard a decade ago. Now considered marginal. If you have existing 2048-bit keys, they're probably fine for now — but generate Ed25519 for new setups.

DSA / ECDSA

AVOID

DSA is disabled in modern OpenSSH. ECDSA can be weak if the random number generator is compromised. Neither is recommended for new keys.

Step-by-Step SSH Hardening

SAFETY ORDER — DO NOT SKIP OR REORDER THESE STEPS

- [ ] **Step 1:** Generate a key pair on your Windows machine
- [ ] **Step 2:** Copy the public key to the server
- [ ] **Step 3:** Test key login in a *new* terminal — keep the original session open
- [ ] **Step 4:** Only after confirming key login works — edit sshd_config to disable passwords
- [ ] **Step 5:** Validate config syntax with `sudo sshd -t`
- [ ] **Step 6:** Reload sshd — and immediately test login in another terminal

SETUP WALKTHROUGH · GENERATING AND DEPLOYING AN SSH KEY PAIR

Generate a key on Windows, copy it to the Linux server, and confirm it works before touching the password settings.

1 **Generate the key pair on your Windows machine.** Open PowerShell (or WSL — both work).

POWERSHELL

```
PS> ssh-keygen -t ed25519 -C "philip-laptop"
Generating public/private ed25519 key pair.
Enter file in which to save the key (C:\Users\philip/.ssh/id_ed25519):
↵ press Enter to accept the default path
Enter passphrase (empty for no passphrase): choose a strong passphrase
Enter same passphrase again: repeat it
Your identification has been saved in C:\Users\philip/.ssh/id_ed25519
Your public key has been saved in C:\Users\philip/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:AbCdEfGhIjKlMnOpQrStUvWxYz1234567890ab philip-laptop
```

WSL / LINUX CLIENT

```
$ ssh-keygen -t ed25519 -C "philip-laptop"
# Default path: ~/.ssh/id_ed25519 (i.e. /home/philip/.ssh/id_ed25519)
```

The `-C` comment identifies which machine this key came from — helpful when you have multiple keys in `authorized_keys`. Use the passphrase: if the private key file is ever stolen, the passphrase prevents the thief from using it.

2 **Copy the public key to the server.** Two methods depending on your environment. POWERSHELL (NO SSH-

```
# Read your public key and append it to authorized_keys on the server
PS> type $env:USERPROFILE\.ssh\id_ed25519.pub | ssh philip@webserver "mkdir -p ~/.ssh && ch
philip@webserver's password: (enter your password — last time you'll need to)
```

WSL / LINUX / MACOS (SSH-COPY-ID AVAILABLE)

```
$ ssh-copy-id -i ~/.ssh/id_ed25519.pub philip@webserver
philip@webserver's password: (enter your password)
Number of key(s) added: 1
Now try logging into the machine, with:   "ssh 'philip@webserver'"
```

Both methods do the same thing: append your public key to `~/.ssh/authorized_keys` on the server,

3 **Verify the permissions on the server.** Wrong permissions are the most common reason key auth silently fails.

```
# On the server — check permissions
$ ls -la ~/.ssh/
drwx------  2 philip philip 4096 Jun 15 10:01 .              ← 700 ✓
-rw-------  1 philip philip  574 Jun 15 10:01 authorized_keys ← 600 ✓

# If permissions are wrong, fix them:
$ chmod 700 ~/.ssh
$ chmod 600 ~/.ssh/authorized_keys
$ chown -R philip:philip ~/.ssh
```

4 **Test key login in a NEW terminal — do not close the original session.**

```
# Open a brand new PowerShell or WSL window. Do not close the existing SSH session.
PS> ssh -i $env:USERPROFILE\.ssh\id_ed25519 philip@webserver
Enter passphrase for key '/home/philip/.ssh/id_ed25519': (your key passphrase)
philip@webserver:~$   ← successful key-based login

# If it still asks for a system password (not the key passphrase), key auth failed.
# Debug with verbose output:
PS> ssh -v -i $env:USERPROFILE\.ssh\id_ed25519 philip@webserver 2>&1 | Select-String "Auth"
# Look for lines like:
# "Trying public key..." → server tried the key
# "Authentications that can continue: publickey" → server accepts keys
# "Server accepts key" → your key was recognised
```

Only proceed to Step 5 once this login works. If it fails, diagnose before touching any config.

5 **Create the sshd hardening drop-in file.** Using a drop-in file in `/etc/ssh/sshd_config.d/` is cleaner than editing `sshd_config` directly — it survives package upgrades and is easy to review later.

```
$ sudo nano /etc/ssh/sshd_config.d/hardening.conf
```

Write the following content (explained in the next section):

```
PermitRootLogin          no
PasswordAuthentication   no
PubkeyAuthentication     yes
PermitEmptyPasswords     no
ChallengeResponseAuthentication no
KbdInteractiveAuthentication    no
X11Forwarding            no
AllowUsers               philip
MaxAuthTries             3
LoginGraceTime           30
ClientAliveInterval      300
ClientAliveCountMax      2
```

6 **Validate the config before restarting anything.**

```
$ sudo sshd -t
# No output = no errors. Any output here is a syntax problem.
# Fix it before continuing — a broken config will prevent sshd from starting.
```

7 **Reload sshd — keeping the existing session open as a safety net.**

```
$ sudo systemctl reload sshd
# reload applies the new config without disconnecting existing sessions.
# Your current SSH session remains active.
# Now immediately open another new terminal and test login:
```

In a third terminal window: `ssh philip@webserver` — it should connect using your key. If it connects, you're done. If it asks for a password, check the drop-in file for typos and re-check the `sshd -t` validation.

With PasswordAuthentication no, anyone trying to SSH with a password gets "Permission denied (publickey)" immediately — no password prompt, no brute-force opportunity. The fail2ban jail still runs but will rarely trigger since there are no passwords to fail.

Understanding the sshd_config Settings

`PermitRootLogin`	`no`	Prevents anyone logging in directly as root. The root account is a constant brute-force target. Admin access happens via a normal user + sudo. Even if root has no password, this blocks all root login attempts.
`PasswordAuthentication`	`no`	The main hardening step. Disables password-based SSH login entirely. Only key authentication works. Brute-force attacks

		become futile.
`PubkeyAuthentication`	`yes`	Explicitly enables key-based auth. This is the default, but stating it clearly makes the intent obvious and prevents future confusion.
`PermitEmptyPasswords`	`no`	Blocks login to accounts that have no password set (an empty password string). Should always be no.
`ChallengeResponseAuthentication`	`no`	Disables keyboard-interactive authentication (a PAM-backed mechanism that can allow passwords even when PasswordAuthentication is no on some configurations). Disable it explicitly.
`KbdInteractiveAuthentication`	`no`	Same as ChallengeResponseAuthentication on newer OpenSSH versions (the name changed). Include both for compatibility.
`X11Forwarding`	`no`	Disables X11 (graphical desktop) forwarding over SSH. A headless server doesn't need this, and it's a historical attack surface.
`AllowUsers`	`philip`	Whitelist of usernames allowed to SSH in. Any user not listed is denied — even if they have a valid key. Prevents attackers from accessing service accounts (www-data, daemon) via SSH even if they somehow get a key onto the server.
`MaxAuthTries`	`3`	Limits the number of authentication attempts per connection to 3. After 3 failures, the connection is dropped. This is per-connection, not per-IP — fail2ban handles the IP-level banning across connections.
`LoginGraceTime`	`30`	How long (in seconds) sshd waits for a successful login before closing the unauthenticated connection. Default is 120 seconds — 30 is plenty for a legitimate key auth.
`ClientAliveInterval`	`300`	Send a keepalive to the client every 300 seconds (5 minutes) of idle. Prevents stale connections from accumulating and consuming server resources.
`ClientAliveCountMax`	`2`	After 2 missed keepalive responses (2 × 300s = 10 minutes unresponsive), disconnect the session. Keeps the connection list clean.

> **If you have multiple users on the server**, add them all to `AllowUsers`: `AllowUsers philip alice` (space-separated). If you forget a user, they'll be locked out of SSH even with a valid key — which you'll discover at the worst possible time.

SSH File Permissions — Why They Matter

OpenSSH refuses to use keys if the files or directories have permissions that are too open. This is a security feature, not a bug — a world-readable private key file is a security problem. If key auth silently fails, wrong permissions are the first thing to check.

~/.ssh/

`chmod 700`

Only the owner can read, write, or list this directory. Group and others: no access.

~/.ssh/authorized_keys

`chmod 600`

Only the owner can read and write. Group and others: no access.

~/.ssh/id_ed25519

`chmod 600`

Private key — only the owner can read it. If anyone else can read it, the key is compromised.

~/.ssh/id_ed25519.pub

`chmod 644`

Public key — readable by anyone (it's public). 644 is fine; it contains no secrets.

SSH Agent — Passphrase Once Per Session

If you added a passphrase to your private key (you should have), you're prompted for it every time you use the key. The SSH agent solves this: it holds the decrypted key in memory for your session, so you only type the passphrase once.

WINDOWS — SSH-AGENT IS BUILT IN

```
# Enable the ssh-agent service (one-time setup, run as administrator)
PS> Set-Service ssh-agent -StartupType Automatic
PS> Start-Service ssh-agent

# Add your key to the agent (prompts for passphrase once)
PS> ssh-add $env:USERPROFILE\.ssh\id_ed25519
Enter passphrase for C:\Users\philip/.ssh/id_ed25519: ••••••••
Identity added: C:\Users\philip/.ssh/id_ed25519 (philip-laptop)

# Now ssh-ing to the server uses the key without asking for the passphrase
PS> ssh philip@webserver
philip@webserver:~$   ← no passphrase prompt

# List keys currently in the agent
PS> ssh-add -l
256 SHA256:AbCdEf... philip-laptop (ED25519)
```

WSL / LINUX — AGENT STARTS PER SESSION

```
# Start the agent and add the key
$ eval "$(ssh-agent -s)"
Agent pid 12345
$ ssh-add ~/.ssh/id_ed25519
Enter passphrase: ••••••••
Identity added: /home/philip/.ssh/id_ed25519

# Add to ~/.bashrc or ~/.zshrc to auto-start the agent and load the key:
# eval "$(ssh-agent -s)" && ssh-add ~/.ssh/id_ed25519 2>/dev/null
```

Multiple Machines and Multiple Keys

Each machine you connect from should have its own key pair — never copy private keys between machines. If a laptop is lost, you remove that machine's public key from `authorized_keys` on the server without affecting any other machine.

```
# On the server — authorized_keys with multiple keys (one per line)
$ cat ~/.ssh/authorized_keys
ssh-ed25519 AAAA...abc philip-laptop
ssh-ed25519 AAAA...def philip-desktop
ssh-ed25519 AAAA...ghi philip-work-laptop

# To revoke access from a specific machine: open authorized_keys
# and delete the relevant line. Reload sshd (not required — authorized_keys
# is read on each connection attempt).

# To add a new machine: generate a key on the new machine, then
# append its public key to authorized_keys on the server:
$ echo "ssh-ed25519 AAAA...xyz philip-new-machine" >> ~/.ssh/authorized_keys
$ chmod 600 ~/.ssh/authorized_keys   # permissions not changed by echo, but gooc
```

The ~/.ssh/config Shortcut File

Instead of typing `ssh -i ~/.ssh/id_ed25519 philip@192.168.1.100 -p 22` every time, define a host alias in `~/.ssh/config` (on your Windows machine or WSL client).

WINDOWS PATH: C:\USERS\PHILIP\.SSH\CONFIG | LINUX / WSL PATH: ~/.SSH/CONFIG

```
# ~/.ssh/config — SSH client configuration

Host webserver
    HostName         192.168.1.100          # or ssh.osztromok.com for external
    User             philip
    IdentityFile     ~/.ssh/id_ed25519
    Port             22
    ServerAliveInterval 60

# External access via the DDNS hostname
Host webserver-ext
    HostName         ssh.osztromok.com
    User             philip
    IdentityFile     ~/.ssh/id_ed25519
    Port             22
```

```
# After saving config, connect with just the alias
PS> ssh webserver          # local connection
PS> ssh webserver-ext      # external via DDNS

# The config file should not be world-readable
$ chmod 600 ~/.ssh/config    # on WSL/Linux
# On Windows, the permissions are managed by the file system — keep it in your u
```

What to Do If You Lose Your Private Key

> **If your private key is lost and you have no other keys in authorized_keys, and PasswordAuthentication is set to no — you are locked out of SSH.** Recovery requires physical access to the server (keyboard + monitor). This is why it's important to keep a backup of the private key (encrypted, in a secure location like a password manager) and to set up at least two machines (or a backup key) before disabling password auth.

```
# Recovery — if you have physical access to the server
# 1. Log in locally (console access)
# 2. Temporarily re-enable password auth:
$ sudo nano /etc/ssh/sshd_config.d/hardening.conf
# Change: PasswordAuthentication no → PasswordAuthentication yes
$ sudo systemctl reload sshd
# 3. SSH in with your password from another machine
# 4. Generate a new key pair on the client, copy the new public key to authorize
# 5. Re-disable password auth in the config
$ sudo systemctl reload sshd

# Prevention: keep an encrypted backup of the private key
# Options: password manager (Bitwarden, 1Password), encrypted USB drive,
# or print the key and store it physically (it's just text)
```

> **Best practice: two independent keys.** Before hardening, add a key from a second machine (or a backup key stored securely). If you ever lose the primary key, the second one gets you back in without physical access to the server.

Updating fail2ban After Disabling Passwords

With `PasswordAuthentication no`, the only SSH authentication failures that can happen are: wrong key (rare — usually misconfiguration), unsupported key type, or connecting from a machine with no key at all. With no passwords to brute-force, the SSH jail's job is lighter — but keep it running, since any failure is now more suspicious.

```
# In /etc/fail2ban/jail.local — tighten the SSH jail now that passwords are off

[sshd]
enabled  = true
port     = ssh
logpath  = %(sshd_log)s
backend  = %(sshd_backend)s
maxretry = 3      # reduced from 5 — legitimate key auth doesn't fail
bantime  = 24h
```

```
$ sudo fail2ban-client reload sshd
```

> **Check your own IP is in ignoreip** before reducing maxretry. If your SSH client has connection issues (wrong key path, agent not running) and you try several times quickly, you could ban yourself with maxretry=3.

Troubleshooting

Permission denied (publickey) — key auth failing

In order of likelihood: (1) Wrong permissions on `~/.ssh` (needs 700) or `~/.ssh/authorized_keys` (needs 600) on the server. Fix: `chmod 700 ~/.ssh && chmod 600 ~/.ssh/authorized_keys`. (2) Public key not actually in authorized_keys — check: `cat ~/.ssh/authorized_keys` on the server; it should contain your public key. (3) Wrong user — the authorized_keys file must be in the home directory of the user you're logging in as. (4) SELinux context wrong (rare on Ubuntu) — `restorecon -Rv ~/.ssh`. (5) Verbose debug from client: `ssh -vvv philip@webserver` — look for "Offering public key" then "Server accepts key" or the rejection reason.

sshd won't start after editing sshd_config / drop-in file

Config syntax error. Check: `sudo sshd -t` — it prints the exact line and problem. Fix the issue, validate again, then reload. If you reloaded before checking and sshd is now down: you can still fix the config via the console. `sudo systemctl status sshd` will show the error. This is why the `reload` command is safer than `restart` — existing connections stay alive even if the reload fails.

sshd keeps asking for password even after PasswordAuthentication no

Another config file is overriding your setting. On Ubuntu, `/etc/ssh/sshd_config` may have `Include /etc/ssh/sshd_config.d/*.conf` — check that the drop-in file is in the right directory and has a `.conf` extension. Also check: `sudo grep -r PasswordAuthentication /etc/ssh/` — if there are conflicting settings across multiple files, the last one wins (files are processed alphabetically in `sshd_config.d/`). Name your file `99-hardening.conf` to ensure it's last.

AllowUsers locked out my other user

`AllowUsers philip` — only `philip` can SSH. Any other user is denied even with a valid key. Fix: add missing users: `AllowUsers philip alice`. If you're locked out, recover via console access or from an existing open SSH session. Reload after fixing: `sudo systemctl reload sshd`.

ssh-add says "Could not open a connection to your authentication agent"

The SSH agent isn't running in the current shell. On Linux/WSL: run `eval "$(ssh-agent -s)"` then `ssh-add ~/.ssh/id_ed25519`. On Windows: ensure the ssh-agent service is running — open PowerShell as admin and run `Start-Service ssh-agent`. Check current status: `Get-Service ssh-agent`.

Connection times out immediately after hardening

Check UFW — did the port get blocked? `sudo ufw status`. Also check fail2ban: `sudo fail2ban-client status sshd` — your IP may have been banned if you had several failed attempts during setup. Unban with: `sudo fail2ban-client set sshd unbanip YOUR.IP`. If the connection times out (no response at all), it's UFW or a network issue. If you get "Connection refused", sshd isn't running.

Quick Reference — Chapter 5

COMMAND / FILE	PURPOSE
`ssh-keygen -t ed25519 -C "label"`	Generate an Ed25519 key pair — run on client machine
`ssh-copy-id -i ~/.ssh/id_ed25519.pub user@server`	Copy public key to server (WSL/Linux clients)
`type pub.key | ssh user@server "cat >> ~/.ssh/authorized_keys"`	Copy public key manually (Windows PowerShell)
`ssh -v philip@webserver`	Verbose login — shows which auth methods are tried and why they fail
`sudo sshd -t`	Validate sshd_config syntax — run before every reload
`sudo systemctl reload sshd`	Apply new config without disconnecting existing sessions
`/etc/ssh/sshd_config.d/hardening.conf`	Drop-in file for hardening settings — upgrade-safe location
`~/.ssh/authorized_keys`	Holds allowed public keys (one per line) — permissions must be 600
`~/.ssh/config`	Client-side aliases: Host, HostName, User, IdentityFile, Port

COMMAND / FILE	PURPOSE
`ssh-add ~/.ssh/id_ed25519`	Add key to SSH agent — passphrase once per session
`ssh-add -l`	List keys currently loaded in the agent

SSHD_CONFIG SETTING	RECOMMENDED	EFFECT
`PermitRootLogin`	**no**	Block all direct root SSH login
`PasswordAuthentication`	**no**	Force key-only auth — eliminates brute-force attack surface
`PubkeyAuthentication`	**yes**	Explicitly enable key auth
`AllowUsers`	**philip**	Whitelist — only named accounts can SSH regardless of key
`MaxAuthTries`	**3**	Disconnect after 3 failed attempts per connection
`LoginGraceTime`	**30**	Close unauthenticated connections after 30 seconds
`X11Forwarding`	**no**	No graphical forwarding — server doesn't need it
`ClientAliveInterval`	**300**	Keepalive every 5 minutes
`ClientAliveCountMax`	**2**	Disconnect after 10 minutes of silence

Chapter 6 — Apache Security Headers

Every HTTP response your server sends contains headers — metadata that tells the browser how to handle the response. Most of these headers are functional (Content-Type, Content-Length, Cache-Control). Security headers are a distinct category: they tell the browser what the server permits, so the browser can refuse anything outside those boundaries. They cost nothing to add and protect against a wide range of real attacks — from clickjacking to cross-site scripting.

> **What this chapter covers:** How security headers work and why they matter. Hiding server version information (ServerTokens, ServerSignature, X-Powered-By). X-Frame-Options against clickjacking. X-Content-Type-Options against MIME sniffing. Referrer-Policy for URL privacy. Permissions-Policy to lock down browser APIs. Content Security Policy — the most powerful header, with a safe roll-out approach using Report-Only mode. Assembling all headers into the VirtualHost config. Testing with curl and online tools. How Cloudflare interacts with these headers. Common mistakes (duplicate headers, CSP breakage).

How Security Headers Work

```
Without security headers — the browser makes its own decisions:

Server ──▶ 200 OK                    ← response
           Content-Type: text/html
           (no further instructions)
Browser: can I run this script? → sure, why not
Browser: can I show this in an iframe? → sure
Browser: should I follow this redirect to http://? → yes

With security headers — the server sets the rules:

Server ──▶ 200 OK
           Content-Type: text/html
           X-Frame-Options: DENY           ← browser refuses to render in iframes
           X-Content-Type-Options: nosniff ← browser trusts the declared Content-
           Content-Security-Policy: default-src 'self' ← only scripts from same o
           Referrer-Policy: no-referrer    ← no URL leakage on navigation
Browser: request to iframe this page → blocked (X-Frame-Options)
```

```
  Browser: script from cdn.evil.com → blocked (CSP)
  Browser: MIME-sniff this upload → blocked (nosniff)
```

Security headers are client-side enforcement enforced by the browser — they don't stop a determined attacker from making raw HTTP requests, but they protect your actual users' browsers from being used against them. They're most effective against injected content, data leakage, and social-engineering attacks.

> **mod_headers must be enabled.** If you followed Chapter 2 (HSTS), mod_headers is already active. Confirm: `apache2ctl -M | grep headers` — you should see `headers_module`. If not: `sudo a2enmod headers && sudo systemctl restart apache2`.

Hiding Server Information

By default, Apache tells the world exactly what version it's running. That version information is directly useful to attackers — they can look up which CVEs apply to that exact version and craft targeted exploits.

DEFAULT — INFORMATION LEAKAGE

```
Server: Apache/2.4.57 (Ubuntu) X-Powered-By: PHP/8.2.10 ← exact version + OS + PHP version ← attacker searches for CVEs targeting Apache 2.4.57 specifically
```

HARDENED — MINIMAL DISCLOSURE

```
Server: Apache ← generic name, no version ← X-Powered-By removed entirely ← attacker knows it's Apache (unavoidable) but not which version
```

```
# /etc/apache2/conf-available/security.conf (or directly in apache2.conf)
# These are server-level settings — not per-VirtualHost

# Edit the existing security.conf (already present on Ubuntu):
$ sudo nano /etc/apache2/conf-available/security.conf

# Find and set these two lines:
ServerTokens Prod
ServerSignature Off

# ServerTokens Prod  — "Server: Apache" only, no version or OS
```

```
# ServerTokens Full  — the default: "Apache/2.4.57 (Ubuntu)" — too much info
# ServerSignature Off — no server version appended to error pages (404, 500 etc.

# Enable the security.conf if not already enabled:
$ sudo a2enconf security
$ sudo systemctl reload apache2
```

```
# Remove the X-Powered-By header (PHP adds this automatically)
# Add to your VirtualHost config or a global conf file:

Header always unset X-Powered-By
Header always unset X-AspNet-Version

# "unset" removes the header if it exists — safe even if PHP isn't installed
```

The Security Headers — One by One

`X-Frame-Options:` `SAMEORIGIN`

Protects against: Clickjacking — an attacker embeds your site in a transparent iframe over their own page, then tricks your logged-in users into clicking their buttons (submitting your forms, deleting accounts, etc.).

DENY — never allow framing (recommended if you don't embed your own site in iframes).
SAMEORIGIN — only allow framing from the same domain (useful if you use iframes internally).

Being superseded by CSP's `frame-ancestors` directive, but still valuable for older browser compatibility.

`X-Content-Type-Options:` `nosniff`

Protects against: MIME-type confusion attacks. Without this, a browser may "sniff" the content of a response and decide it looks like JavaScript, even if the server declared it as `text/plain`.

Attack scenario: A user uploads an image file that contains JavaScript code. The server stores it and serves it as `image/jpeg`. Without nosniff, some browsers execute the JS. With nosniff, the browser trusts the declared Content-Type and treats it as an image.

This is a single fixed value — there are no other valid options for this header.

`Referrer-Policy:` `strict-origin-when-cross-origin`

Protects against: URL leakage. When a user clicks a link from your site to another, the browser sends a `Referer` header containing the full URL of the page they came from — including any query parameters, tokens, or user IDs in the URL.

strict-origin-when-cross-origin — sends the full URL only to same-origin destinations; sends just the origin (domain name, no path) to cross-origin HTTPS destinations; sends nothing to HTTP destinations. A sensible default for most sites.

Alternatives: `no-referrer` (nothing sent, most private), `same-origin` (nothing sent to other domains).

```
Permissions-Policy: camera=(), microphone=(), geolocation=()
```

Protects against: Injected content requesting browser permissions. If an XSS attack injects a script that requests camera access, this header tells the browser the site doesn't have permission to do that — the request is denied before the user even sees a prompt.

`camera=()` — deny camera access entirely (empty parentheses = no allowed origins).
`microphone=()` — deny microphone access.
`geolocation=()` — deny geolocation access.

For a web hosting/tutorial site like osztromok.com, none of these features are needed — deny them all. Add only what your site actually requires.

```
Strict-Transport-Security: max-age=86400
```

Covered in detail in Chapter 2. Brief reminder: this header tells browsers to always use HTTPS for your domain, even if the user types `http://`. Don't add it again if you already set it in the VirtualHost — duplicate headers cause both values to be sent.

How Clickjacking Works

ATTACK EXPLAINED — CLICKJACKING

An attacker creates `evil.com/prize.html` — a page that appears to offer a free prize with a large "Claim Now" button. Hidden behind that button (via CSS opacity: 0) is an iframe loading `osztromok.com/account/delete`. The iframe is perfectly positioned so the attacker's button overlaps your site's "Confirm Delete" button.

A logged-in visitor to evil.com clicks what looks like "Claim Now" — they actually click "Confirm Delete" on your site. The request goes through with their session cookie because the browser is genuinely loading your site, just invisibly.

X-Frame-Options: DENY prevents your site from loading inside any iframe at all — the attack fails at the first step.

Fix: `Header always set X-Frame-Options "SAMEORIGIN"` (or DENY if you never use iframes)

Content Security Policy (CSP)

CSP is the most powerful security header — and the most complex. It defines a whitelist of trusted sources for every type of resource your page can load: scripts, stylesheets, images, fonts, API calls. The browser blocks anything that doesn't match the policy. A properly configured CSP makes XSS (cross-site scripting) attacks far harder to exploit — even if an attacker manages to inject HTML, the injected scripts won't run.

> **A wrong CSP will break your site.** Too restrictive and your own CSS or JavaScript stops loading. A blank page with no errors is a typical symptom. Always deploy CSP in Report-Only mode first — the browser enforces nothing but logs every violation, letting you see what would break before it does.

CSP Directives

`default-src`

The fallback for all resource types not explicitly listed. Set this first; override specific types below. **Most restrictive setting:** `'self'`

`script-src`

Where JavaScript can be loaded from. The highest-value directive — controls XSS. **Avoid `'unsafe-inline'` — it defeats much of CSP's XSS protection.**

`style-src`

Where CSS can be loaded from. Inline `<style>` tags require `'unsafe-inline'` or a hash/nonce. Google Fonts requires adding `https://fonts.googleapis.com`.

`img-src`

Where images can be loaded from. `data:` allows base64-embedded images. Add CDN domains if you load images from third-party services.

`font-src`

Where web fonts can be loaded from. If using Google Fonts: add `https://fonts.gstatic.com`.

`connect-src`

Which URLs JavaScript can connect to (fetch, XHR, WebSocket). Relevant if your site makes API calls to external services.

`frame-ancestors`

The CSP replacement for X-Frame-Options. `frame-ancestors 'none'` = DENY. More powerful — X-Frame-Options only controls one level of nesting.

`form-action`

Where forms can submit to. `'self'` prevents injected forms from submitting to attacker-controlled servers (data exfiltration via form).

`base-uri`

Restricts the `<base>` element, which can redirect all relative URLs. `'self'` or `'none'` prevents base tag injection attacks.

CSP Source Values

`'self'` `'none'` `https://trusted.com` `https:` `'unsafe-inline'` `'unsafe-eval'` `data:` `*` `http:`

Green = safe and recommended. Amber = use only where necessary (weaken security). Red = avoid — these open significant holes.

Deploying CSP Safely — Report-Only First

- **Step 1 — Deploy in Report-Only mode**

  Use `Content-Security-Policy-Report-Only` instead of `Content-Security-Policy`. The browser evaluates the policy and logs violations to the browser console (and optionally to a `report-uri`) but **does not block anything**. Your site works exactly as before.

- **Step 2 — Browse your site and watch the console**

  Open DevTools → Console. Any violation appears as: **"Refused to load the script 'https://cdn.example.com/lib.js' because it violates the following Content Security Policy directive: script-src 'self'"**. Note every domain mentioned — these are sources you need to whitelist.

- **Step 3 — Expand the policy to cover legitimate sources**

  Add each violation's source to the appropriate directive. For example, if jQuery loads from `cdnjs.cloudflare.com`, add it to `script-src`. If Google Analytics appears, add `www.google-analytics.com` and `www.googletagmanager.com` to `script-src` and `connect-src`.

- **Step 4 — Switch from Report-Only to enforcing**

  Once the console is clean (no violations), change `Content-Security-Policy-Report-Only` to `Content-Security-Policy` in Apache. Reload Apache. Test every page of the site — especially any with forms, embedded maps, videos, or third-party widgets.

- **Step 5 — Keep refining over time**

  When you add new third-party embeds (YouTube video, Disqus comments, analytics), they'll generate CSP violations. That's the policy doing its job — you need to explicitly allow each new source. This makes you conscious of every external dependency.

Putting It All Together — Complete VirtualHost Config

All headers go in the `*:443` VirtualHost (HTTPS only). The HTTP VirtualHost only handles redirecting to HTTPS — no need for security headers on the redirect response.

```
# /etc/apache2/sites-available/osztromok.com.conf

# ── HTTP VirtualHost — redirect only ──────────────────────────
<VirtualHost *:80>
    ServerName  osztromok.com
    ServerAlias www.osztromok.com
    Redirect    permanent / https://osztromok.com/
```

```
</VirtualHost>

# ── HTTPS VirtualHost — all security headers go here ──────────
<VirtualHost *:443>
    ServerName   osztromok.com
    ServerAlias  www.osztromok.com
    DocumentRoot /var/www/osztromok.com/public_html

    <Directory /var/www/osztromok.com/public_html>
        Options       FollowSymLinks
        AllowOverride All
        Require       all granted
    </Directory>

    SSLEngine             on
    SSLCertificateFile    /etc/letsencrypt/live/osztromok.com/fullchain.pem
    SSLCertificateKeyFile /etc/letsencrypt/live/osztromok.com/privkey.pem

    # ── Security headers ──────────────────────────────────────

    # Remove server information headers
    Header always unset X-Powered-By
    Header always unset X-AspNet-Version

    # Clickjacking protection
    Header always set X-Frame-Options "SAMEORIGIN"

    # MIME sniffing protection
    Header always set X-Content-Type-Options "nosniff"

    # Referrer privacy
    Header always set Referrer-Policy "strict-origin-when-cross-origin"

    # Lock down browser APIs
    Header always set Permissions-Policy "camera=(), microphone=(), geolocation=

    # HSTS — already set in Chapter 2 (don't duplicate if already present)
    Header always set Strict-Transport-Security "max-age=86400"

    # CSP — start in Report-Only mode, switch to enforcing after testing
    Header always set Content-Security-Policy-Report-Only "default-src 'self'; s

    # ──────────────────────────────────────────────────────────
```

```
    ErrorLog  ${APACHE_LOG_DIR}/osztromok_error.log
    CustomLog ${APACHE_LOG_DIR}/osztromok_access.log combined
</VirtualHost>
```

```
# Validate the config, then reload
$ sudo apache2ctl configtest
Syntax OK
$ sudo systemctl reload apache2
```

> **The CSP above is intentionally permissive as a starting point** — `'unsafe-inline'` is included for script-src and style-src because osztromok.com's lesson pages use a lot of inline CSS and JavaScript. Once you've identified all your legitimate sources via Report-Only violations, you can tighten this by removing `'unsafe-inline'` and replacing inline scripts with nonces or hashes.

Switching CSP to Enforcing Mode

```
# After testing with Report-Only and fixing all violations:
# Change the header name from Report-Only to enforcing

# Before:
Header always set Content-Security-Policy-Report-Only "default-src 'self'; ..."

# After:
Header always set Content-Security-Policy "default-src 'self'; ..."

# An example of a tighter policy (after identifying all sources):
Header always set Content-Security-Policy "default-src 'self'; script-src 'self'
```

Testing Your Headers

VERIFICATION · CHECK HEADERS ARE BEING SENT

Three methods to confirm your headers are active.

1. **curl — quick server-side check.**

```
$ curl -I https://osztromok.com
HTTP/2 200
x-frame-options: SAMEORIGIN
x-content-type-options: nosniff
referrer-policy: strict-origin-when-cross-origin
permissions-policy: camera=(), microphone=(), geolocation=(), payment=()
content-security-policy-report-only: default-src 'self'; ...
strict-transport-security: max-age=86400
server: Apache   ← good — no version number

# If a header is missing: check Apache error log and verify the conf loaded
$ sudo grep -i "header" /var/log/apache2/error.log | tail -10
```

2 **Browser DevTools — check CSP violations while you browse.**
Open Chrome or Firefox → F12 → Console tab. Browse every page of your site. CSP violations appear in red:

```
Refused to load script 'https://cdn.jsdelivr.net/npm/...' because it violates
the following Content Security Policy directive: "script-src 'self'".
```

Each violation tells you exactly which domain you need to add to the policy.

3 **Online security header scanners — get a scored report.**
Two reliable options (no account needed):

• **securityheaders.com** — grades A+ through F, flags missing or misconfigured headers, shows what each one does.
• **observatory.mozilla.org** — Mozilla's scanner, checks headers, cookies, redirect chains, and HSTS.

Enter `osztromok.com` and aim for B+ or above. An A+ requires a strict CSP with no `'unsafe-inline'`.

Run the online scanners after every change. A score regression usually means a header was accidentally removed or a duplicate was introduced (two conflicting values for the same header cancel out or cause unexpected behaviour).

How Cloudflare Interacts With These Headers

```
# Headers Cloudflare adds on top of Apache's response (you'll see these in curl)
cf-ray: 8a4b3c2d1e0f-LHR            # Cloudflare request ID — ignore
cf-cache-status: DYNAMIC           # Cloudflare cache state — ignore
server: cloudflare                 # ← Cloudflare replaces the Server header!
```

```
# The Server: header — note Cloudflare replaces "Apache" with "cloudflare"
# in the curl output. This is fine — it's even better than ServerTokens Prod
# since it hides that you're running Apache at all.
```

- **Your custom security headers pass through unchanged** — X-Frame-Options, CSP, X-Content-Type-Options, etc. all arrive at the browser exactly as Apache sent them.
- **The Server: header is replaced by Cloudflare** — visitors see `Server: cloudflare`, not `Server: Apache`. This is actually better; ServerTokens Prod still matters for direct connections and internal tooling.
- **Cloudflare can also set security headers via Transform Rules** (in the Cloudflare dashboard). If Cloudflare sets a header that Apache also sets, the visitor receives the header twice. This can cause unexpected behaviour — check with `curl -I` and look for duplicate header names.
- **Cloudflare's Bot Fight Mode and WAF** add their own layer of protection before requests reach Apache, complementing your headers rather than replacing them.

> **Check for duplicate headers after enabling Cloudflare Transform Rules.** If you set X-Frame-Options in both Apache and a Cloudflare Transform Rule, the browser receives both values. RFC behaviour for duplicate response headers varies by browser — some take the first, some take the last, some treat it as an error. Use `curl -I https://osztromok.com | grep -i "x-frame"` — you should see the header exactly once.

Troubleshooting

Security headers are not appearing in curl -I output

In order of likelihood: (1) Apache config not reloaded after changes — run `sudo systemctl reload apache2`. (2) Syntax error prevented the config from loading — run `sudo apache2ctl configtest`; fix any errors. (3) Headers are in the wrong VirtualHost — if you put them in the HTTP (`*:80`) block but are testing HTTPS, they won't appear. Move them to the `*:443` block. (4) mod_headers not enabled — check: `apache2ctl -M | grep headers`; enable with `sudo a2enmod headers`.

CSS / JavaScript / fonts stopped loading after enabling CSP

Your CSP is blocking a legitimate resource. Open DevTools Console — violation messages show exactly which source is blocked. Add that source to the appropriate directive in the policy (e.g. if `fonts.googleapis.com` is blocked, add it to `style-src`). If many things break at once, temporarily add `'unsafe-inline'` to `script-src` and `style-src` to get the site working, then identify and whitelist specific sources before removing `'unsafe-inline'` again. If

you're in a hurry: switch back to `Content-Security-Policy-Report-Only` to stop blocking while you refine the policy.

Same header appears twice in curl output

Duplicate header — being set in two places. Common causes: (1) Set in both the VirtualHost config and in an `.htaccess` file — remove from one. (2) Set in both Apache and a Cloudflare Transform Rule — remove one. (3) Set in both the server-level `security.conf` and the VirtualHost — pick one location and remove the other. Use `sudo grep -r "X-Frame-Options" /etc/apache2/` to find every place a header is set.

ServerTokens Prod doesn't seem to change the Server: header

Via Cloudflare Tunnel, the `Server:` header is overwritten by Cloudflare — you'll always see `Server: cloudflare` when curling through the tunnel. `ServerTokens Prod` still works, but its effect isn't visible via Cloudflare. To verify it's working, test from localhost: `curl -I http://localhost` — the Server header should show `Apache` (not the version). Also check: is `security.conf` actually enabled? `apache2ctl -t -D DUMP_INCLUDES` shows which conf files are loaded.

Inline styles stop working with CSP enabled

`<style>` tags and `style=""` attributes are blocked unless `'unsafe-inline'` is in `style-src`. For a site with many lesson pages using inline styles (like osztromok.com), the practical choice is to keep `'unsafe-inline'` in `style-src` — it's less ideal from a CSP perspective but doesn't enable script injection. The security benefit of removing inline *style* is smaller than removing inline *script*. Focus your efforts on keeping `script-src` as tight as possible.

Quick Reference — Chapter 6

SETTING / HEADER	RECOMMENDED VALUE	PROTECTS AGAINST
`ServerTokens`	**Prod**	Version disclosure — only "Apache", no version number
`ServerSignature`	**Off**	Version in error page footers
`X-Powered-By`	**unset**	PHP / framework version disclosure
`X-Frame-Options`	**SAMEORIGIN**	Clickjacking via malicious iframes
`X-Content-Type-Options`	**nosniff**	MIME confusion / uploaded file execution
`Referrer-Policy`	**strict-origin-when-cross-origin**	URL leakage to third-party sites
`Permissions-Policy`	**camera=(), microphone=(), geolocation=()**	Browser API abuse via injected scripts

SETTING / HEADER	RECOMMENDED VALUE	PROTECTS AGAINST
`Content-Security-Policy`	**default-src 'self'; ...**	XSS, data injection, resource hijacking

COMMAND	PURPOSE
`curl -I https://osztromok.com`	Check all response headers — verify headers are present and not duplicated
`sudo apache2ctl configtest`	Validate Apache config syntax before reload
`apache2ctl -M | grep headers`	Confirm mod_headers is loaded
`sudo grep -r "Header" /etc/apache2/`	Find all places a header directive is set — hunt for duplicates
`Browser DevTools → Console`	See CSP violation messages while browsing (Report-Only or enforcing)

Chapter 7 — ClamAV & Malware Protection

The previous chapters secured the doors — locking down SSH, hardening the firewall, adding browser-enforced policies. This chapter deals with what happens if something slips through anyway. Malware on a web server most commonly arrives as a PHP web shell uploaded through a vulnerable application, as a compromised package, or injected into legitimate files during an attack. ClamAV is the Linux standard for scanning and catching these — and a few Apache configuration tricks can prevent uploaded files from being executed at all.

> **What this chapter covers:** What ClamAV does and where it fits in your defences. The three ClamAV packages (clamscan, freshclam, clamd daemon). Keeping virus definitions current. Running on-demand scans with the most useful flags — and why to quarantine rather than auto-delete. Scheduled scans via cron and systemd timer. Protecting upload directories by denying PHP execution in Apache. On-access scanning for new uploads. The incident response playbook when a scan finds something. Complementary tools: rkhunter (rootkits), AIDE (file integrity). Resource considerations on a home server.

What ClamAV Does — and Its Limits

```
ClamAV's role in the security stack:

Layer 1 — Network:   UFW (Chapter 3)     blocks unwanted connections
Layer 2 — Auth:      fail2ban (Ch.4)     bans brute-force sources
                     SSH keys (Ch.5)     eliminates password attacks
Layer 3 — Browser:   Security headers (Ch.6) limits what browser runs
Layer 4 — Files:     ClamAV (this chapter)   detects malicious files
                     Apache upload rules      prevents PHP execution
Layer 5 — Monitoring: logwatch, unattended-upgrades (Chapter 8)

ClamAV specifically watches for:
├── PHP web shells (uploaded .php files that give remote shell access)
├── Malicious JavaScript injected into HTML/PHP files
├── Trojan scripts hidden in archives or disguised as images
├── Known malware signatures in downloaded packages
└── Modifications to existing files (when combined with AIDE)
```

ClamAV is **signature-based** — it recognises malware it has seen before (or variants close enough to match a pattern). It won't catch genuinely novel malware until signatures are updated. For a web server, the most important threats it catches are well-known PHP web shells and common malicious scripts — exactly the kind of tool-based attacks that automated exploit kits deploy.

- **It catches:** known web shells, common trojans, malicious scripts, infected archives, most real-world automated attacks.
- **It doesn't catch:** zero-day exploits, custom malware written specifically for your server, compromised dependencies in your code (use `composer audit` / `npm audit` for those).
- **It's not real-time by default:** scanning is on-demand or scheduled. On-access scanning (daemon mode) can make it reactive, but at a resource cost.

The Three ClamAV Components

`clamav`

Scanner

Contains `clamscan` — the command-line scanner. Loads signatures into memory each run (slower, but no daemon needed). Good for scheduled scans and one-off checks.

`clamav-freshclam`

Definition updater

The `freshclam` service downloads new virus signatures from ClamAV's mirrors. Runs as a daemon that checks for updates multiple times daily. **Definitions are useless if they're stale.**

`clamav-daemon`

Background daemon

Runs `clamd` — keeps signatures loaded in memory between scans. `clamdscan` talks to it and is much faster than `clamscan` for repeated scans. Also enables on-access scanning.

`clamdscan`

Daemon client

The client that sends scan jobs to `clamd`. Same flags as `clamscan` but faster because signatures are pre-loaded. Use this for scheduled scans if clamd is running.

> **Home server resource note:** clamd loads the entire signature database into RAM — roughly 500 MB–1 GB. On a server with 2–4 GB of RAM also running Apache, MySQL, and cloudflared, that's a significant chunk. If RAM is tight, skip `clamav-daemon` and use `clamscan` directly for scheduled scans — it's slower but uses no persistent RAM.

Installing ClamAV

```
# Install ClamAV and the daemon
$ sudo apt install clamav clamav-daemon -y

# Stop freshclam service before running a manual update
# (freshclam service and manual freshclam can't run simultaneously)
$ sudo systemctl stop clamav-freshclam

# Download the latest virus definitions (takes a few minutes on first run)
$ sudo freshclam
ClamAV update process started at Sun Jun 15 12:00:00 2026
daily.cvd database is up-to-date (version: 27249, sigs: 2054253, ...)
main.cvd database is up-to-date (version: 62, sigs: 6647427, ...)
bytecode.cvd is up-to-date (version: 334, sigs: 91, ...)

# Restart the freshclam service to resume automatic updates
$ sudo systemctl start clamav-freshclam
$ sudo systemctl enable clamav-freshclam

# Start clamd (if you're using the daemon)
$ sudo systemctl start clamav-daemon
$ sudo systemctl enable clamav-daemon

# Verify everything is running
$ sudo systemctl status clamav-freshclam clamav-daemon
● clamav-freshclam.service — ClamAV virus database updater
     Active: active (running)
● clamav-daemon.service — Clam AntiVirus userspace daemon
     Active: active (running)

# Check signature database version and age
$ clamscan --version
ClamAV 1.0.3/27249/Sun Jun 15 09:00:01 2026
#                  ^^^^^ daily DB version / date — should be within 2-3 days of
```

Running Scans

Scanning the web root (most important scan to run regularly)

```
# Scan the web root recursively — show only infected files (-i flag)
$ sudo clamscan -r -i /var/www/osztromok.com/public_html/

----------- SCAN SUMMARY -----------
Known viruses:     9580123
Engine version:    1.0.3
Scanned files:     847
Infected files:    0   ← nothing found
Time:              12.234 sec (0 m 12 s)

# Scan entire /var/www (covers all virtual hosts)
$ sudo clamscan -r -i /var/www/

# Using the daemon for faster scanning (clamd must be running)
$ sudo clamdscan -i /var/www/osztromok.com/public_html/
```

Key flags

`-r / --recursive`

Scan directories recursively — descends into subdirectories. **Essential for scanning a web root.** Without it, only files in the top directory are scanned.

`-i / --infected`

Only print infected file names. Without this, every scanned file is printed — noisy and hard to read in large directories. **Always use this.**

`--move=DIR`

Move infected files to a quarantine directory instead of leaving them in place. Preferred over --remove — you can review what was found before permanent deletion.

`--remove`

Use with caution — permanently deletes infected files immediately with no review. Appropriate for automated scans of upload temp directories, not for scanning your web root.

`--exclude-dir=DIR`

Skip a directory. Always exclude `/proc`, `/sys`, `/dev` when scanning the whole system — these are virtual filesystems, not real files.

`--log=FILE`

Write scan results to a log file. Essential for scheduled scans so you can review results later without watching the terminal.

`--max-filesize=MB`

Skip files larger than this size (default 25 MB). Large video files don't need virus scanning — adjust if your site serves large files.

`-v / --verbose`

Show every file being scanned (useful for debugging why a specific file isn't being scanned). Don't use in scheduled scans — generates huge log files.

Reading scan output — infected file found

```
/var/www/osztromok.com/public_html/uploads/image_20260615.php: Php.Webshell.Generi

----------- SCAN SUMMARY -----------
Known viruses:    9580123
Engine version:   1.0.3
Scanned files:    1247
Infected files:   1
Time:             18.451 sec (0 m 18 s)

↑ Signature name "Php.Webshell.Generic-2" tells you:
  Php       = PHP malware
  Webshell  = remote command execution shell
  Generic-2 = a known generic web shell variant
The path shows it was uploaded to the uploads directory — classic attack.
```

> **Do not delete without investigating first.** When ClamAV finds something, the immediate question is not "how do I delete this?" — it's "how did it get there?" If you delete the file without finding the entry point, the attacker will just upload another. See the incident response playbook below.

Setting Up a Quarantine Directory

```
# Create a quarantine directory — outside the web root
$ sudo mkdir -p /var/quarantine
$ sudo chmod 700 /var/quarantine   # only root can access

# Run a scan that moves infected files to quarantine instead of leaving them
$ sudo clamscan -r -i --move=/var/quarantine /var/www/
/var/www/osztromok.com/public_html/uploads/shell.php: Php.Webshell.Generic-2 FOU
/var/www/osztromok.com/public_html/uploads/shell.php: moved to '/var/quarantine/

# Review what's in quarantine before deleting
$ sudo ls -la /var/quarantine/
$ sudo file /var/quarantine/shell.php   # confirm file type
$ sudo cat /var/quarantine/shell.php    # read the content — understand what it
```

```
# When satisfied — delete quarantined files
$ sudo rm -rf /var/quarantine/*
```

Scheduled Scans

Option A — Cron job (simple)

```
$ sudo crontab -e   # edit root's crontab
```

```
# Run at 3:00 AM daily — scan web root, log results, email on detection
0 3 * * * clamscan -r -i --move=/var/quarantine --log=/var/log/clamav/daily-scar

# Simpler version — just log, no email:
0 3 * * * clamscan -r -i --move=/var/quarantine --log=/var/log/clamav/daily-scar

# Weekly full-system scan (Sundays at 4 AM — longer running, skip virtual filesy
0 4 * * 0 clamscan -r -i --move=/var/quarantine --exclude-dir=^/proc --exclude-d
```

Option B — Systemd timer (cleaner, with logging)

```
# Create the scan script
$ sudo nano /usr/local/bin/clamav-daily-scan.sh
```

```
#!/bin/bash
# /usr/local/bin/clamav-daily-scan.sh
LOGFILE="/var/log/clamav/daily-scan-$(date +%Y-%m-%d).log"
QUARANTINE="/var/quarantine"
WEBROOT="/var/www"

echo "ClamAV scan started at $(date)" > "$LOGFILE"
clamscan -r -i --move="$QUARANTINE" --log="$LOGFILE" "$WEBROOT"
EXIT_CODE=$?

if [ $EXIT_CODE -eq 1 ]; then
    echo "ALERT: Infected files found — check $LOGFILE and $QUARANTINE" | \
        mail -s "ClamAV THREAT DETECTED on $(hostname)" emubantam@gmail.com
```

```
fi

# Exit codes: 0 = no threats, 1 = threats found, 2 = error
exit $EXIT_CODE
```

```
$ sudo chmod +x /usr/local/bin/clamav-daily-scan.sh

# Create the systemd service unit
$ sudo nano /etc/systemd/system/clamav-daily-scan.service
```

```
[Unit]
Description=ClamAV Daily Web Root Scan
After=clamav-daemon.service

[Service]
Type=oneshot
ExecStart=/usr/local/bin/clamav-daily-scan.sh
User=root
```

```
# Create the timer unit
$ sudo nano /etc/systemd/system/clamav-daily-scan.timer
```

```
[Unit]
Description=Run ClamAV daily web root scan at 3 AM

[Timer]
OnCalendar=*-*-* 03:00:00
RandomizedDelaySec=300
Persistent=true

[Install]
WantedBy=timers.target
```

```
$ sudo systemctl daemon-reload
$ sudo systemctl enable --now clamav-daily-scan.timer

# Verify the timer is scheduled
$ sudo systemctl list-timers clamav-daily-scan.timer
```

```
NEXT                         LEFT       LAST PASSED UNIT
Mon 2026-06-16 03:04:32 BST   14h left   -    -      clamav-daily-scan.timer

# Test the scan script immediately (without waiting for 3 AM)
$ sudo systemctl start clamav-daily-scan.service
$ sudo journalctl -u clamav-daily-scan.service --no-pager
```

Protecting Upload Directories

ClamAV detects malicious files *after* they've been uploaded. Apache configuration can stop them from being *executed* even if they slip through. The two defences are complementary — use both.

The most dangerous upload attack is a PHP web shell: an attacker finds a file upload form (contact form, avatar upload, attachment) and uploads a file named `shell.php` (or `shell.php.jpg` to bypass naive extension checks). If Apache executes it, the attacker gains full shell access to your server under the web server's user.

```
# In your VirtualHost config — deny PHP execution in the uploads directory
# /etc/apache2/sites-available/osztromok.com.conf

<VirtualHost *:443>
    # ... existing config ...

    # Uploads directory — allow file serving but block ALL script execution
    <Directory /var/www/osztromok.com/public_html/uploads>
        # Disable PHP engine for this directory
        php_admin_flag engine off

        # Belt AND braces — also block direct requests to PHP files
        <FilesMatch "\.php[0-9]?$">
            Require all denied
        </FilesMatch>

        # Block other executable extensions often used in attacks
        <FilesMatch "\.(phtml|phar|php3|php4|php5|phps|cgi|pl|py|rb|sh)$">
            Require all denied
        </FilesMatch>
```

```
        # Disable .htaccess overrides in this directory
        # (attackers can upload a .htaccess that re-enables PHP)
        AllowOverride None
    </Directory>
</VirtualHost>
```

```
$ sudo apache2ctl configtest && sudo systemctl reload apache2

# Test that PHP is blocked in the uploads directory
$ echo "<?php echo 'shell'; ?>" | sudo tee /var/www/osztromok.com/public_html/up
$ curl https://osztromok.com/uploads/test.php
403 Forbidden  ← PHP execution blocked — correct
$ sudo rm /var/www/osztromok.com/public_html/uploads/test.php
```

> **`AllowOverride None` in the uploads directory is critical.** Without it, an attacker can upload a `.htaccess` file that re-enables PHP execution with `AddType application/x-httpd-php .jpg` — turning every uploaded image into an executable PHP file. AllowOverride None prevents Apache from reading any .htaccess files in that directory.

On-Access Scanning for Upload Directories

On-access scanning tells clamd to automatically scan files when they appear or are modified in a specific directory — catching malicious uploads the moment they land, before any web request can trigger them.

```
# Edit clamd configuration to enable on-access scanning
$ sudo nano /etc/clamav/clamd.conf
```

```
# Add these lines to /etc/clamav/clamd.conf
OnAccessIncludePath /var/www/osztromok.com/public_html/uploads
OnAccessExcludeUname clamav      # don't scan files created by clamav itself
OnAccessPrevention yes           # block access to infected files (not just log)
OnAccessMaxFileSize 20M          # skip files larger than 20 MB
```

```
# On-access scanning requires clamd to run as root (check User= in clamd.conf)
$ grep "^User" /etc/clamav/clamd.conf
User clamav   ← change to root for on-access to work

$ sudo nano /etc/clamav/clamd.conf
# Change: User clamav → User root

$ sudo systemctl restart clamav-daemon

# Verify on-access is active in the logs
$ sudo journalctl -u clamav-daemon -n 20 | grep -i "access"
clamd[1234]: On-access module loaded successfully.
clamd[1234]: Protecting directory: /var/www/osztromok.com/public_html/uploads
```

On-access scanning adds CPU overhead — every write to the watched directory triggers a scan. For a busy upload directory on a home server, this can be noticeable. If performance degrades, disable on-access and rely on scheduled scans instead. On-access also requires kernel support for fanotify, present in all modern Ubuntu/Debian kernels.

When a Scan Finds Something — Incident Response

Phase 1 **Contain — isolate the threat before investigating**

Move the infected file to quarantine (if not done automatically by the scan): `sudo mv /path/to/threat.php /var/quarantine/`. Do not delete it yet — you need to examine it. If the site is actively serving malicious content, consider temporarily taking the vhost offline: `sudo a2dissite osztromok.com.conf && sudo systemctl reload apache2`.

Phase 2 **Investigate — how did it get there?**

Check Apache access logs for requests to the infected file: `sudo grep "threat.php" /var/log/apache2/access.log`. Find when it was created: `sudo stat /var/quarantine/threat.php` — use the mtime (modification time). Check Apache logs around that time for POST requests (uploads): `sudo grep "POST" /var/log/apache2/access.log | grep "Jun 15"`. Look at the file contents: `sudo cat /var/quarantine/threat.php` — this tells you what the attacker intended and sometimes reveals the upload method. Check for other suspicious files uploaded at the same time: `sudo find /var/www/ -newer /var/quarantine/threat.php -type f | head -20`.

Phase 3 **Close the entry point**

Identify the upload mechanism the attacker used and close it. Common entry points: unpatched CMS plugin (run `sudo -u www-data wp core update && wp plugin update --all`), weak file type validation in custom upload code, world-writable directories, compromised credentials. Add PHP execution blocking to the upload directory (see the Apache config above) if not already in place. Add the uploading IP to fail2ban: `sudo fail2ban-client set apache-auth banip ATTACKER-IP`.

Phase 4 **Assess damage — was anything exfiltrated or modified?**

Check if the web shell was ever executed (not just uploaded): `sudo grep "threat.php" /var/log/apache2/access.log` — a GET request to it means it was accessed. If it was executed, check for: new user accounts (`cat /etc/passwd | grep -v nologin`), new cron jobs (`sudo crontab -l -u www-data`), modified system files (see AIDE below), outbound connections (`sudo ss -tnp | grep www-data`). Run ClamAV again on the whole system: `sudo clamscan -r -i /`.

Phase 5 **Clean up and recover**

Delete the quarantined file: `sudo rm /var/quarantine/threat.php`. Restore any modified legitimate files from git (if your web content is version-controlled) or from backup. Re-enable the site if it was taken offline: `sudo a2ensite osztromok.com.conf && sudo systemctl reload apache2`. Update all software: `sudo apt upgrade && sudo apt autoremove`. Run one final clean scan to confirm all threats are gone.

Complementary Tools

ClamAV catches known malware by signature. These tools cover different aspects of the threat landscape:

rkhunter

ROOTKIT / BACKDOOR SCANNER

Checks for rootkits — malware that hides from the OS itself. Looks for suspicious files in system directories, backdoored system binaries, and known rootkit signatures. **Install:** `sudo apt install rkhunter`. **Run:** `sudo rkhunter --check`.

AIDE

FILE INTEGRITY MONITOR

Advanced Intrusion Detection Environment. Takes a cryptographic snapshot of your filesystem (hashes, permissions, ownership). Later runs compare the current state to the baseline — any changed system file is flagged. **Install:** `sudo apt install aide`. **Init:** `sudo aideinit`. **Check:** `sudo aide --check`.

chkrootkit

ROOTKIT SCANNER

Simpler alternative to rkhunter. Checks for signs of rootkits in running processes, network interfaces, and filesystem. Good as a second opinion alongside rkhunter. **Install:** `sudo apt install chkrootkit`. **Run:** `sudo chkrootkit`.

lynis

SYSTEM SECURITY AUDIT

Comprehensive system hardening audit — checks hundreds of configuration settings across SSH,

Apache, filesystem permissions, logging, and more. Produces a scored report with actionable suggestions. **Install:** `sudo apt install lynis`. **Run:** `sudo lynis audit system`.

```
# Quick setup for rkhunter — run after installing
$ sudo apt install rkhunter -y
$ sudo rkhunter --update          # update rkhunter database
$ sudo rkhunter --propupd         # baseline current system (run after trusted s
$ sudo rkhunter --check --skip-keypress
[ Rootkits ]
Checking for known rootkit files and directories    [ None found ]
Checking for Anonoxymoron Rootkit                  [ Not found ]
  ... (many checks) ...
[ Application checks ]
Checking Apache2 configuration...                  [ OK ]
System checks summary: 0 warnings

# Quick AIDE setup
$ sudo apt install aide -y
$ sudo aideinit              # creates the initial database baseline (takes a fe
$ sudo cp /var/lib/aide/aide.db.new /var/lib/aide/aide.db
$ sudo aide --check          # compare current filesystem against baseline
AIDE found differences between database and filesystem!
Changed: /var/log/apache2/access.log  ← expected: log files change constantly
# Exclude log directories from AIDE monitoring in /etc/aide/aide.conf
```

Troubleshooting

freshclam fails — "ERROR: /var/log/clamav/freshclam.log: Permission denied"

The freshclam service is already running and holds the log file, or the log file permissions are wrong. Solution: stop the service before running freshclam manually — `sudo systemctl stop clamav-freshclam`, then `sudo freshclam`, then `sudo systemctl start clamav-freshclam`. Never run `sudo freshclam` while the service is running — they'll conflict trying to write to the same log file.

clamd fails to start — "ERROR: Can't open/parse the config file /etc/clamav/clamd.conf"

Syntax error in `clamd.conf`. Check: `sudo clamd --config-file=/etc/clamav/clamd.conf --check-config`. Common causes: typo in a directive name, a value missing where one is expected, or an

`OnAccessIncludePath` pointing to a non-existent directory. Also check ownership: `ls -la /etc/clamav/clamd.conf` — it should be owned by `clamav:clamav`.

clamscan is very slow — taking 10+ minutes for the web root

`clamscan` loads all signatures into memory on each run — that's ~150 MB of database reading before scanning starts. Switch to `clamdscan` if the daemon is running (signatures stay loaded in RAM, subsequent scans start immediately). Alternatively, accept the slowness and schedule scans during off-peak hours (3–4 AM). Also check: is `/var/www` on a network mount? Network filesystem scans are much slower than local disk.

ClamAV flagging a legitimate file as infected (false positive)

False positives happen — ClamAV's signatures are broad patterns that sometimes match benign files. Confirm it's a false positive: look at the file content and the signature name. Report it at **https://www.clamav.net/reports/fp**. Whitelist the specific file with `--exclude=FILENAME` or by adding a whitelist to the ClamAV database. Never blindly auto-delete files without a review step — this is why the `--move` flag (quarantine) is safer than `--remove`.

clamav-daemon not starting — "ERROR: SelfCheck: Database modification time has changed"

The signature database was updated by freshclam while clamd was running. clamd detected the change and logged an error, or may have stopped. This is normal — clamd should automatically reload the database. If it's stopped: `sudo systemctl restart clamav-daemon`. To prevent this: configure freshclam to signal clamd when updating (`NotifyClamd /etc/clamav/clamd.conf` in `freshclam.conf`).

Quick Reference — Chapter 7

COMMAND	PURPOSE
`sudo clamscan -r -i /var/www/`	Scan web root recursively, show only infected files
`sudo clamdscan -i /var/www/`	Same scan via daemon — much faster (daemon must be running)
`sudo clamscan -r -i --move=/var/quarantine /var/www/`	Scan and quarantine infected files — preferred over --remove
`sudo freshclam`	Manually update virus definitions (stop clamav-freshclam service first)
`clamscan --version`	Check current signature database version and date

COMMAND	PURPOSE
`sudo systemctl status clamav-freshclam clamav-daemon`	Check both ClamAV services are running
`sudo rkhunter --check --skip-keypress`	Scan for rootkits and backdoors
`sudo aide --check`	Compare current filesystem against the AIDE baseline
`sudo lynis audit system`	Full system security audit with scored report
`sudo find /var/www/ -name "*.php" -newer /var/www/index.php`	Find PHP files added after a reference date (incident investigation)

TOOL	WHAT IT CATCHES	WHEN TO RUN
`ClamAV`	Known malware, web shells, trojans in files	**Daily scheduled scan of web root**
`rkhunter`	Rootkits, hidden processes, backdoored binaries	**Weekly, and after any suspicious activity**
`AIDE`	Any file that has changed since the baseline	**Daily or after software updates**
`lynis`	Configuration weaknesses across the whole system	**Monthly, and after major config changes**

Chapter 8 — Monitoring & Maintenance

Security is not a state you reach — it's a process you maintain. Chapters 1 through 7 built the defences: encrypted connections, firewall rules, hardened SSH, security headers, malware scanning. This chapter closes the loop: making sure those defences stay current, that you're notified when something unusual happens, and that routine maintenance doesn't get missed. A server you're not watching is a server you don't control.

> **What this chapter covers:** Why continuous monitoring matters after hardening. logwatch for daily human-readable log digests. unattended-upgrades for automatic security patches. Setting up msmtp as an email relay so the server can send alerts. Apache log analysis — top IPs, scanning patterns, suspicious requests. Auth log monitoring — successful logins and sudo usage. Disk and certificate expiry alerts. Cloudflare analytics as a complement to server logs. Regular maintenance checklist (weekly → monthly → quarterly). Full course security stack summary.

The Monitoring Gap

```
What you've built across 8 chapters — without monitoring:

[Attacker] ──▶ UFW: blocked                    ← Ch.3 — working
[Attacker] ──▶ SSH brute force: fail2ban ban    ← Ch.4 — working
[Attacker] ──▶ SSH key auth: rejected           ← Ch.5 — working
[Attacker] ──▶ Uploads web shell                ← ClamAV should catch it
[ClamAV]   ──▶ finds shell.php at 3 AM          ← Ch.7 scan found it
[No alert] ──▶ nobody sees it for 3 days        ← THE MONITORING GAP

With monitoring:

[ClamAV]   ──▶ finds shell.php at 3 AM          ← Ch.7 scan
[Script]   ──▶ emails emubantam@gmail.com        ← you know within hours
[logwatch] ──▶ morning email: "847 POST requests to /uploads at 2:59 AM" ← context
[You]      ──▶ investigate → find entry point → patch → done
```

Every tool in this course generates logs. The question is whether anyone reads them. Manually checking logs daily is unrealistic. The solution is to let the tools summarise the logs and surface anomalies — only alerting you when something needs attention.

Email Alerts — Setting Up msmtp

logwatch, unattended-upgrades, ClamAV scan scripts, and certificate renewal all want to send emails. A home server typically doesn't have a full mail server — msmtp relays outgoing mail through an existing account (Gmail works well).

SETUP WALKTHROUGH · MSMTP EMAIL RELAY VIA GMAIL

Configure the server to send alert emails through your Gmail account.

1 **Install msmtp and mailutils.**

```
$ sudo apt install msmtp msmtp-mta mailutils -y
# msmtp       — the SMTP relay
# msmtp-mta   — makes msmtp the system sendmail (used by cron, logwatch, etc.)
# mailutils   — provides the "mail" command for testing
```

2 **Create a Gmail App Password.** Gmail requires an App Password (not your regular password) for SMTP access from non-Google apps.

Go to: **myaccount.google.com → Security → 2-Step Verification → App passwords**. Create a new app password for "Mail / Linux computer". Copy the 16-character code — you'll only see it once.

3 **Create the msmtp config file.**

```
$ sudo nano /etc/msmtprc
```

```
# /etc/msmtprc — system-wide msmtp config
defaults
auth           on
tls            on
tls_trust_file /etc/ssl/certs/ca-certificates.crt
logfile        /var/log/msmtp.log

account        gmail
host           smtp.gmail.com
port           587
from           emubantam@gmail.com
user           emubantam@gmail.com
password       abcd efgh ijkl mnop   ← your 16-char App Password (spaces OK)

account default : gmail
```

```
# Secure the config — it contains your App Password
$ sudo chmod 600 /etc/msmtprc
$ sudo chown root:root /etc/msmtprc
```

4 **Test email sending.**

```
$ echo "Test from webserver" | mail -s "msmtp test" emubantam@gmail.com
# Check your inbox — it should arrive within a minute.
# If it doesn't, check the log:
$ sudo tail /var/log/msmtp.log
host=smtp.gmail.com tls=on auth=on from=emubantam@gmail.com recipients=emubantam@gmail.com mailsize=... smtpstatus=2
```

msmtp-mta creates a symlink so that any program calling `/usr/sbin/sendmail` (what logwatch, cron, and unattended-upgrades all use) goes through msmtp. Once msmtp is configured, all the other tools work without any further changes.

logwatch — Daily Log Digest

logwatch parses your system logs overnight and emails you a human-readable summary — SSH activity, Apache traffic, package updates, disk events, fail2ban bans, and more. It turns a stack of machine-readable log files into a five-minute morning read.

```
$ sudo apt install logwatch -y

# Run logwatch immediately to see what it produces (uses yesterday's logs by default)
$ sudo logwatch --detail Med --range Yesterday --format text --output stdout | head -80
 ################### Logwatch 7.8 (01/21/24) ####################
        Processing Initiated: Mon Jun 15 08:00:00 2026
        Date Range Processed: yesterday ( 2026-Jun-14 )
                              Period is day.
 ################################################################

 ---------------------- Connections (secure-log) Begin ------------------------

 Unmatched Entries
  sshd[1234]: Accepted publickey for philip from 192.168.1.5 port 54321 : 1 Time(s)

 ----------------------- Connections (secure-log) End -------------------------

 ---------------------- SSHD Begin ------------------------
 Users logging in through sshd:
    philip:
       192.168.1.5 (home-laptop): 3 times
 ...
```

```
# Configure logwatch to email you daily
$ sudo nano /etc/logwatch/conf/logwatch.conf
```

```
# /etc/logwatch/conf/logwatch.conf
# Only include settings you want to override from the defaults

MailTo = emubantam@gmail.com
MailFrom = webserver-logwatch@osztromok.com
Detail = Med          # Low / Med / High — Med is a good balance
Range = Yesterday     # report on yesterday's logs
Format = text         # text or html

# logwatch runs nightly via cron (/etc/cron.daily/00logwatch)
# No further setup needed — it runs automatically at ~6 AM
```

```
# Send a test report to your email (today's logs)
$ sudo logwatch --detail Med --range Today --mailto emubantam@gmail.com
```

```
# Check your inbox — arrives within a minute via msmtp
```

> **What to scan for in logwatch reports:** SSH logins from unexpected IPs (anything not 192.168.1.x is worth noting). Large numbers of failed auth attempts that fail2ban didn't catch. Apache: unusually high 404 counts. Package manager: packages that failed to update (can indicate a signing key problem). Disk: filesystem nearing capacity.

Automatic Security Updates

Most compromises exploit known vulnerabilities — ones with published patches. Keeping security updates current is one of the highest-value maintenance tasks. `unattended-upgrades` applies security patches automatically, without waiting for you to log in and run `apt upgrade`.

```
$ sudo apt install unattended-upgrades -y

# Enable automatic updates (creates the trigger files)
$ sudo dpkg-reconfigure --priority=low unattended-upgrades
# Select "Yes" when prompted

# The two config files that control this:
# /etc/apt/apt.conf.d/20auto-upgrades — controls WHEN to run
# /etc/apt/apt.conf.d/50unattended-upgrades — controls WHAT to install
```

```
# Verify 20auto-upgrades has the right settings
$ cat /etc/apt/apt.conf.d/20auto-upgrades
APT::Periodic::Update-Package-Lists "1";   ← run apt update daily
APT::Periodic::Unattended-Upgrade "1";     ← apply upgrades daily
```

```
# Edit 50unattended-upgrades for email and cleanup settings
$ sudo nano /etc/apt/apt.conf.d/50unattended-upgrades
```

```
# Key settings to configure in 50unattended-upgrades:

// Email report when updates are applied (or fail)
Unattended-Upgrade::Mail "emubantam@gmail.com";
Unattended-Upgrade::MailReport "on-change";   // only email if something changed

// Remove packages that are no longer needed after upgrades
Unattended-Upgrade::Remove-Unused-Dependencies "true";

// Automatic reboot after security kernel updates — set to false for a home server
// (you want to control when the server reboots)
Unattended-Upgrade::Automatic-Reboot "false";
```

```
// Log to syslog (logwatch will include this in its daily report)
Unattended-Upgrade::SyslogEnable "true";
```

```
# Test the configuration — dry run (shows what would be upgraded, installs nothing)
$ sudo unattended-upgrade --dry-run --debug
Initial blacklisted packages:
Initial whitelisted packages:
Starting unattended upgrades script
Allowed origins are: ...
No packages found that can be upgraded unattended and no pending auto-removals

# Check what the last run actually did
$ sudo cat /var/log/unattended-upgrades/unattended-upgrades.log | tail -20
```

> **Automatic reboots:** Some kernel security updates require a reboot to take effect. With `Automatic-Reboot "false"`, the new kernel is installed but the old one keeps running. You'll see a "System restart required" message when you SSH in. Plan a manual reboot during off-peak hours when you see this message — `sudo shutdown -r +5 "Rebooting for kernel update"`.

Apache Log Analysis

Apache's access and error logs contain a continuous record of everything hitting your server. Knowing what's normal helps you spot what isn't.

Access log patterns to watch for

```
# ── Top requesting IPs (spot scanners and heavy users) ──────────
$ sudo awk '{print $1}' /var/log/apache2/access.log | sort | uniq -c | sort -rn | head -15
   1847 162.158.32.1     ← Cloudflare IP — expected (your tunnel)
    924 162.158.32.2     ← Another Cloudflare IP — expected
     83 192.168.1.5      ← your laptop — expected
     47 45.33.32.156     ← external IP with 47 requests — worth investigating

# ── Most requested 404 paths (reveals what scanners are looking for) ──
$ sudo grep " 404 " /var/log/apache2/access.log | awk '{print $7}' | sort | uniq -c | sort -rn | he
     43 /wp-admin/setup-config.php    ← WordPress scanner (you're not running WP)
     31 /phpMyAdmin/index.php         ← phpMyAdmin scanner
     28 /.env                         ← looking for exposed .env files
     19 /admin/login                  ← generic admin panel scanner
      3 /favicon.ico                  ← normal — browsers auto-request this

# ── Suspicious POST requests ────────────────────────────────────
$ sudo grep '"POST' /var/log/apache2/access.log | awk '{print $7}' | sort | uniq -c | sort -rn | he
    847 /uploads/image.php    ← POST to a PHP file in uploads — VERY suspicious
      4 /contact.php          ← form submissions — expected

# ── Status code distribution (spot unusual error spikes) ────────
```

```
$ sudo awk '{print $9}' /var/log/apache2/access.log | sort | uniq -c | sort -rn
  12847 200    ← normal responses
    431 404    ← not found — expected some
    124 301    ← redirects — expected
      8 500    ← server errors — investigate if high
```

NORMAL **Cloudflare IPs in access log**

All public web traffic comes through Cloudflare's IPs — expected. With mod_remoteip (Chapter 4), real visitor IPs are logged in the `%h` field. Without it, you'd only see Cloudflare IPs.

INVESTIGATE **Dozens of 404s for /wp-admin, /phpMyAdmin, /.env from one IP**

Automated vulnerability scanner. Annoying but mostly harmless — your site doesn't have these paths. Consider adding the IP to fail2ban: `sudo fail2ban-client set apache-noscript banip IP`. fail2ban's `apache-badbots` jail should catch persistent scanners automatically.

ALERT **POST requests to files in /uploads directory**

Attempted web shell execution. Investigate immediately — find the request in detail, check if the uploaded file exists, run ClamAV scan on the uploads directory. The Apache config (Chapter 7) should return 403 for these — if they're getting through, the upload protection isn't working.

ALERT **Spike in 500 errors coinciding with unusual requests**

Could indicate a successful injection causing PHP/Apache errors. Check the Apache error log: `sudo grep " 500 " /var/log/apache2/error.log | tail -20`. Look for PHP errors, path traversal attempts, or unusual file paths in the error messages.

INVESTIGATE **Sudden large traffic spike from a region you don't expect**

Could be: legitimate traffic (your site was shared somewhere), DDoS attempt (Cloudflare handles this), or a scraped content farm. Check Cloudflare Analytics for the source breakdown. If DDoS, enable Cloudflare "Under Attack" mode temporarily.

Auth log monitoring

```
# All successful SSH logins (who, from where, when)
$ sudo grep "Accepted" /var/log/auth.log
Jun 15 09:14:22 webserver sshd[2341]: Accepted publickey for philip from 192.168.1.5 port 51234 ssh
Jun 15 22:03:11 webserver sshd[3891]: Accepted publickey for philip from 203.0.113.45 port 49123 ss
# ↑ Second login is from an external IP — not your home network.
#   Is this expected (you SSHing in from work/phone)?
#   If not, treat as a security incident.

# All sudo commands run (who ran what)
$ sudo grep "COMMAND" /var/log/auth.log | tail -20
Jun 15 10:22:01 webserver sudo:   philip : TTY=pts/0 ; PWD=/var/www ; USER=root ; COMMAND=/bin/nano

# New user accounts created (should be rare)
$ sudo grep "new user" /var/log/auth.log
# Any output here is worth investigating

# Failed login attempts (fail2ban handles these, but useful for the count)
$ sudo grep "Failed password" /var/log/auth.log | wc -l
847   ← 847 failed attempts since the log was last rotated — normal background noise
```

Disk Space and Certificate Expiry Alerts

Disk space alert script

```
# A web root disk filling up means Apache logs stop being written
# (a security blind spot). Alert yourself before it happens.
$ sudo nano /usr/local/bin/disk-alert.sh
```

```
#!/bin/bash
# /usr/local/bin/disk-alert.sh
# Email alert if any filesystem exceeds 80% usage
THRESHOLD=80
EMAIL="emubantam@gmail.com"

df -H | grep -vE '^Filesystem|tmpfs|cdrom|udev' | awk '{print $5 " " $1 " " $6}' | while read usage
    pct="${usage%%%}"
    if [ "$pct" -ge "$THRESHOLD" ]; then
        echo "Disk alert on $(hostname): $fs ($mount) is at $usage" | \
            mail -s "DISK SPACE ALERT: $mount at $usage" "$EMAIL"
    fi
done
```

```
$ sudo chmod +x /usr/local/bin/disk-alert.sh

# Add to root's crontab — check daily at 8 AM
$ sudo crontab -e
```

```
0 8 * * * /usr/local/bin/disk-alert.sh
```

Certificate expiry check

```
# Check current certificate expiry dates
$ sudo certbot certificates
Found the following certs:
  Certificate Name: osztromok.com
    Domains: osztromok.com www.osztromok.com
    Expiry Date: 2026-09-12 09:00:00+00:00 (VALID: 89 days)
    Certificate Path: /etc/letsencrypt/live/osztromok.com/fullchain.pem

# The certbot renewal timer handles auto-renewal (Chapter 1)
# But verify the timer is running:
$ sudo systemctl status certbot.timer
Active: active (waiting)
$ sudo systemctl list-timers certbot.timer
NEXT                        ACTIVATES
Mon 2026-06-16 09:00:00     certbot.service
```

```
# Check certificate from outside (verifies the live cert, not just what certbot says)
$ echo | openssl s_client -connect osztromok.com:443 -servername osztromok.com 2>/dev/null | openss
notBefore=Jun 14 09:00:00 2026 GMT
notAfter=Sep 12 09:00:00 2026 GMT  ← 89 days from now — healthy

# Monthly cron to warn you if the cert renews less than 30 days before expiry
# (catches scenarios where auto-renewal silently fails)
$ sudo crontab -e
```

```
0 9 1 * * certbot certificates 2>&1 | grep -i "VALID:" | grep -v "VALID: [3-9][0-9] days\|VALID: [0
```

Cloudflare Analytics

Cloudflare sees all traffic before it reaches your server — its analytics give you a perspective that Apache logs can't: traffic that was blocked before it even hit the tunnel, geographic breakdowns, and threat intelligence.

- **Analytics → Traffic:** total requests, cached vs uncached, unique visitors, top countries. Baseline this — a sudden spike from an unexpected country may indicate a targeted scan.
- **Security → Events:** every request Cloudflare blocked — WAF rules triggered, bot fight mode blocks, rate limiting hits. These are attacks that never reached Apache.
- **DNS → Analytics:** DNS query volume — useful for detecting DNS-based probing or amplification if you've had a public IP leak.
- **Speed → Performance:** cache hit rate — higher is better; frequently requested static assets (CSS, images) should cache at Cloudflare and never reach Apache.

> **Cross-reference Cloudflare Security Events with your Apache logs.** If Cloudflare shows 500 blocked requests from a specific IP and your Apache logs show the same IP made 3 requests that got through — those 3 succeeded despite Cloudflare's block, which means they came via a direct path (not the tunnel). Investigate how they bypassed Cloudflare.

Regular Maintenance Checklist

DAILY Handled automatically — just read the emails

- **logwatch report** — scan the morning email. Look for: unexpected SSH logins, POST requests to unusual paths, high 404 counts, packages that failed to update.
- **ClamAV scan result** — the scan script emails you only on detection. No email = nothing found.
- **unattended-upgrades** — emails you when security packages are updated. If you see a kernel update, plan a reboot.

WEEKLY 5–10 minute check

- **fail2ban stats:** `sudo fail2ban-client status sshd` — are bans happening? Any single IP banned repeatedly (persistent attacker)?

- **Disk space:** `df -h` — confirm nothing is nearing 80%. Web root, /var/log, and /var/lib/mysql are the common culprits.
- **Running services:** `sudo systemctl status apache2 clamav-freshclam clamav-daemon fail2ban cloudflared` — all should be active (running).
- **New files in web root:** `sudo find /var/www/ -newer /var/www/osztromok.com/index.html -type f | head -10` — any unexpected new files?

MONTHLY **20–30 minute review**

- **Full manual apt upgrade:** `sudo apt update && sudo apt upgrade` — applies non-security updates that unattended-upgrades skips. Review packages before confirming.
- **Certificate expiry:** `sudo certbot certificates` — should show 60+ days remaining. If under 30, investigate why auto-renewal hasn't run.
- **rkhunter scan:** `sudo rkhunter --check --skip-keypress` — check for rootkits and backdoors. Update first: `sudo rkhunter --update`.
- **lynis audit:** `sudo lynis audit system` — check for configuration drift. Score should stay stable or improve over time.
- **Review authorized_keys:** `cat ~/.ssh/authorized_keys` — do you recognise every key? Remove any from machines you no longer use.
- **Cloudflare Security Events:** review blocked threats in Cloudflare dashboard — anything targeting specific paths warrants adding an Apache deny rule.

AFTER UPDATES / CHANGES **Verify nothing broke**

- **After Apache config changes:** `sudo apache2ctl configtest` then `sudo systemctl reload apache2`, then curl your site.
- **After kernel update reboot:** check all services are running — Apache, fail2ban, clamav-daemon, cloudflared.
- **After adding new PHP application:** run ClamAV scan on the new files, check Apache logs for unusual requests within the first 24 hours.
- **After rkhunter propupd:** only run `rkhunter --propupd` after intentional system changes (package upgrades) — it updates the baseline, so running it after a compromise hides the evidence.

Troubleshooting

logwatch is not sending email / emails go to spam

Test msmtp first: `echo "test" | mail -s "test" emubantam@gmail.com` — if this also fails, the problem is msmtp, not logwatch. Check `sudo tail /var/log/msmtp.log` for SMTP errors. If emails arrive but go to spam: the `From:` address doesn't match the sending domain — set `MailFrom` in logwatch.conf to your Gmail address, not a custom domain. Gmail's spam filter is more lenient with mail from the same account.

unattended-upgrades is not running — no log entries

Check the trigger: `cat /etc/apt/apt.conf.d/20auto-upgrades` — both `Update-Package-Lists` and `Unattended-Upgrade` should be `"1"`. Check the service: `sudo systemctl status apt-daily-upgrade.timer` and `sudo systemctl status apt-daily.timer` — both should be active. Check the log: `sudo cat /var/log/unattended-upgrades/unattended-upgrades.log` — even "no upgrades available" is a valid log entry that confirms it ran.

logwatch report shows SSH logins from an IP I don't recognise

Don't panic — first rule out legitimate sources: is it a VPN exit node you use? A mobile data IP? Use `whois IPADDRESS` or an IP lookup tool to identify the organisation. Check when the login occurred and what was done after: `sudo grep "IPADDRESS" /var/log/auth.log` for the full authentication record, then `sudo grep "COMMAND" /var/log/auth.log` for any sudo usage around that time. If it's genuinely unexpected and key-based auth succeeded (not just an attempt), change your authorized_keys immediately and investigate which machine's private key may be compromised.

Disk alert fires but df shows plenty of space

The disk-alert script checks all mounted filesystems. The alert may be for a small filesystem you're not thinking of — check: `df -H` — look at `/boot` (old kernels accumulate here), `/var/log` (log rotation may be misconfigured), or Docker's overlay filesystem if Docker is installed. Clean old kernels: `sudo apt autoremove`. Set up log rotation properly: `sudo logrotate --force /etc/logrotate.d/apache2`.

Course Summary — Your Complete Security Stack

Here is everything you've built across all eight chapters — a layered security posture that turns a default Apache install into a hardened, monitored web server.

Ch. 1	**HTTPS / Let's Encrypt**	DNS-01 challenge via Cloudflare plugin bypasses Virgin Media's port block. 90-day certificates with automatic renewal via systemd timer. **Full (Strict)** SSL mode in Cloudflare for end-to-end encryption through the tunnel.
Ch. 2	**Enforcing HTTPS**	HTTP redirects to HTTPS in Apache. HSTS header with escalating max-age. mod_rewrite with X-Forwarded-Proto header for Cloudflare compatibility — avoiding the redirect loop trap.
Ch. 3	**UFW Firewall**	Default-deny inbound. Only SSH open to the network — web traffic arrives via the Cloudflare Tunnel (outbound, not subject to inbound rules). **ufw limit ssh** for rate limiting. UFW logging enabled.
Ch. 4	**fail2ban**	SSH jail with 24h bans, Apache jails with mod_remoteip for real visitor IPs, escalating bans via bantime.increment. Home subnet whitelisted in ignoreip. Covers the gap between UFW's rate limit and persistent attackers.
Ch. 5	**SSH Hardening**	Ed25519 key pair per device. PasswordAuthentication no — brute-force is mathematically infeasible. PermitRootLogin no. AllowUsers whitelist. Drop-in file in sshd_config.d/ for clean, upgrade-safe configuration.
Ch. 6	**Security Headers**	ServerTokens Prod hides version. X-Frame-Options against clickjacking. X-Content-Type-Options against MIME confusion. Referrer-Policy for URL privacy. Content Security Policy in Report-Only mode, ready to enforce.
Ch. 7	**ClamAV & Malware**	Daily scheduled scan of the web root with quarantine for finds. Apache upload directories: PHP execution blocked, AllowOverride None prevents .htaccess bypass. Supplemented by rkhunter (rootkits) and AIDE (file integrity baseline).
Ch. 8	**Monitoring**	logwatch daily digest by email. unattended-upgrades for automatic security patches. msmtp relay for all server alerts. Apache and auth log analysis patterns. Disk and certificate expiry alerts. Regular maintenance schedule.

Securing Your Web Server — Complete

osztromok.com now runs behind eight layers of active defence. Traffic is encrypted end-to-end. The network firewall admits nothing it shouldn't. Brute-force attacks hit a wall at fail2ban. SSH accepts no passwords at all. The browser is told exactly what the server permits. Malware is scanned nightly. And every anomaly generates an alert before it becomes a problem.

Security is a practice, not a destination. The maintenance checklist in this chapter is the ongoing contract: read the daily emails, apply the monthly checks, and the stack stays sharp.

Quick Reference — Chapter 8

COMMAND	PURPOSE
`sudo logwatch --detail Med --range Today --mailto EMAIL`	Run logwatch immediately and email the report
`sudo unattended-upgrade --dry-run --debug`	Test unattended-upgrades without installing anything
`cat /var/log/unattended-upgrades/unattended-upgrades.log`	Review what was automatically updated and when
`echo "test" | mail -s "test" EMAIL`	Test msmtp email relay is working
`sudo awk '{print $1}' /var/log/apache2/access.log | sort | uniq -c | sort -rn | head -15`	Top requesting IPs — spot scanners and unusual sources
`sudo grep " 404 " /var/log/apache2/access.log | awk '{print $7}' | sort | uniq -c | sort -rn | head -15`	Most common 404 paths — reveals what scanners are probing for
`sudo grep "Accepted" /var/log/auth.log`	All successful SSH logins — verify all are expected
`sudo grep "COMMAND" /var/log/auth.log`	All sudo commands run — audit admin activity

COMMAND	PURPOSE
`sudo certbot certificates`	Check certificate expiry dates and renewal status
`df -h`	Disk space — watch /var/log and /var/www for growth
`sudo systemctl status apache2 fail2ban clamav-daemon cloudflared`	Quick service health check — all should show active (running)