Linux Server Security Course

Flavio Copes

Updated August 3, 2026

Contents

Secure access **2**
- Threat-model the server . . . 2
- Use personal admin accounts . . . 3
- Use SSH keys safely . . . 4
- Harden SSH without locking yourself out . . . 5
- Check your understanding: secure access . . . 6

Network exposure **6**
- Inventory listening services . . . 6
- Configure a default-deny firewall . . . 7
- Terminate TLS deliberately . . . 8
- Keep private services private . . . 9
- Check your understanding: network exposure . . . 10

System hardening **10**
- Apply security updates . . . 10
- Disable unneeded services . . . 11
- Restrict files and processes . . . 12
- Store server secrets outside code . . . 13
- Check your understanding: system hardening . . . 14

Observe and back up **14**
- Read authentication and service logs . . . 14
- Notice unexpected change . . . 15
- Design actionable alerts . . . 16
- Back up and restore . . . 17
- Check your understanding: observe and back up . . . 18

Respond and rebuild **18**
- Recognize a possible compromise . . . 18
- Contain the host . . . 19
- Rebuild from known-good inputs . . . 20
- Complete the server security review . . . 21
- Check your understanding: respond and rebuild . . . 21

Harden and operate an Ubuntu server with controlled SSH access, limited exposure, patched services, useful evidence, tested backups, and a rebuild plan.

What you'll learn: Secure a disposable Ubuntu VPS, verify its access and exposure, monitor important changes, restore it, and rehearse compromise response.

Secure access

Threat-model the host and control accounts, sudo, SSH keys, and recovery.

Threat-model the server

Identify exposed services, valuable data, privileged accounts, provider controls, and recovery paths before changing Ubuntu security settings.

A server is not secured by one hardening script. Start with what this machine runs and what would happen if it failed.

Threat modeling sounds like an enterprise ritual. For a single Ubuntu VPS it is one page of notes. You write down what the server holds, who can reach it, and which paths an attacker could take. Every control you add later should trace back to that page.

Inventory the machine

List public ports, applications, databases, credentials, backups, DNS, cloud-provider access, and people with administrative rights. Concrete beats complete:

```
host: blog-vps (Ubuntu 24.04 LTS at Hetzner)
public: 22/tcp ssh, 80/tcp + 443/tcp nginx
runs: nginx, node blog app, postgresql (local only)
data: posts, newsletter subscriber emails, TLS private key
admins: flavio (ssh key), provider console (password + MFA)
recovery: provider console, nightly off-site backup
```

Ten minutes of writing, and you already know which firewall rules make sense, which accounts need review, and which credentials are worth stealing.

Name the attack paths

Separate internet threats from a compromised application or a stolen provider account. Each path meets different controls:

```
internet -> sshd, nginx          firewall, patched packages, ssh keys
web app bug -> shell as app     service user, file permissions
stolen provider login -> root   console MFA, provider audit log
```

A public blog VPS and a private database host need different controls. A stolen provider account can also bypass a carefully hardened SSH service.

Rank each path by reachable assets and recovery cost. This keeps the exercise practical: provider MFA may matter more than another SSH tweak when the console can replace the server.

The mistake to avoid

The common failure is modeling only the SSH path. You spend an evening on it while the provider password is reused from an old forum account. You can recognize this by reading your own page: ten SSH mitigations and not one line about the console means the model is lopsided. Fix the cheap, high-value gaps first.

Draw the lab VPS, provider console, DNS, backups, public ports, and administrator paths. Remove one assumed control, such as SSH or provider MFA, and record the remaining exposure and recovery

path.

Use personal admin accounts

Give each administrator an attributable account, use sudo for privileged work, and remove access when responsibilities change.

Shared accounts hide who changed the system. Give each administrator their own login.

When three people share `ubuntu`, a sudo event cannot identify who changed Nginx. Personal accounts add lifecycle work, but let you remove one person without rotating everyone's access.

Create a personal account

Create the account and add it to the `sudo` group:

```
sudo adduser dana
sudo usermod -aG sudo dana
```

Have Dana log in and confirm privilege escalation works:

```
sudo whoami
# root
```

Use an unprivileged account for normal work and `sudo` only for the command that needs it. Every escalation lands in the authentication log with a username attached, so a `sudo systemctl restart nginx` at 02:14 points to a person, not a mystery.

Review who has access

Review group membership and sudo rules on a schedule, not only when something breaks:

```
getent group sudo
# sudo:x:27:dana,marco
sudo ls /etc/sudoers.d/
```

Edit sudo rules only through `visudo`, which validates the syntax before saving. A typo in `/etc/sudoers` can lock every administrator out of root at once:

```
sudo visudo -f /etc/sudoers.d/deploy
```

Keep service accounts non-interactive unless a documented task requires a shell. An application user with a login shell and a password is an extra door nobody watches.

Remove a person cleanly

When responsibilities change, lock the account and expire it instead of deleting it right away, so file ownership and audit history stay readable:

```
sudo usermod -L marco
sudo chage -E 0 marco
```

Then remove their key from `authorized_keys` and their entry from any sudoers file. Deleting the account first is the common mistake: files owned by a vanished UID show up as raw numbers, and you lose the trail you may need later.

Keep one independently protected recovery identity outside normal daily use. Review its access too, because an emergency account that never expires can become the least visible administrator.

Save a list of login shells, groups, and effective sudo permissions for each administrator. Disable one test account and prove its SSH key and sudo path both fail while another administrator still has recovery access.

Use SSH keys safely

Generate a modern key locally, protect the private key, install only the public key, and remove stale authorized keys.

Your SSH private key stays on your machine. The server receives the public key.

The key pair replaces the password. The server keeps `~/.ssh/authorized_keys`, a list of public keys allowed to log in. Anyone can hold the public half. Only the private half, on your laptop, can answer the login challenge.

Generate a modern key

Use a supported modern key type. Ed25519 is the current default choice:

```
ssh-keygen -t ed25519 -C "flavio-macbook-2026"
```

The `-C` comment labels the key, so a year from now you know which device it belongs to. Set a passphrase where practical, and use an agent or secure hardware for frequent use so you type it once per session:

```
ssh-add ~/.ssh/id_ed25519
```

Install it on the server

```
ssh-copy-id dana@203.0.113.10
```

This appends the public key to `authorized_keys` with the right permissions. Verify host fingerprints on first connection: the client shows the server's key fingerprint, and you confirm it against a value you obtained some other way.

Host verification protects the other direction. Record the expected server fingerprint through a trusted channel so a convincing login prompt cannot silently collect credentials or commands.

One key per device

A key copied across ten servers turns one stolen laptop into ten entry points. A dedicated key costs more management but allows narrow revocation and clearer attribution.

Label authorized keys and remove them when a device or person no longer needs access. The comment field makes the cleanup a one-line edit:

```
grep flavio-macbook /home/dana/.ssh/authorized_keys
```

The mistake to watch for: copying the private key onto the server ”to hop to the next machine”. Never move the private file. If you need to reach a second host through the first, use `ProxyJump` in your SSH config, which keeps the key on your laptop.

Create a dedicated lab key and record its fingerprint beside the authorized entry. Remove that public key, then prove the old private key fails while the documented recovery key still works.

Harden SSH without locking yourself out

Test key access, keep a recovery channel, disable unnecessary authentication paths, and validate configuration before reloading SSH.

SSH changes can lock you out. Keep the current session open and prove a second connection before closing it.

The order of operations matters more than the settings themselves. Confirm key login works first. Only then remove the weaker paths.

Put the change in a drop-in file

On Ubuntu, sshd reads `/etc/ssh/sshd_config` plus every fragment in `/etc/ssh/sshd_config.d/`. Keep your hardening in its own file:

```
# /etc/ssh/sshd_config.d/90-hardening.conf
PermitRootLogin no
PasswordAuthentication no
KbdInteractiveAuthentication no
```

After key login works, disable direct root login and password authentication when the environment supports it. Restrict users where useful with `AllowUsers dana marco`, and keep provider-console recovery ready.

Validate before reloading

Validate the configuration before reload and never experiment first on the only production path:

```
sudo sshd -t
```

No output means the syntax is valid. Then check what the daemon will actually enforce. Test the effective daemon configuration, not only the edited file — included configuration fragments can override a setting and make a clean-looking file provide false confidence:

```
sudo sshd -T | grep -Ei 'permitrootlogin|passwordauthentication'
# permitrootlogin no
# passwordauthentication no
```

Now reload, and open a second session from another terminal before closing the first:

```
sudo systemctl reload ssh
```

If the second login works, the change is safe. If it fails, your original session is still open and you can revert.

Prove the lockdown

Force a password attempt and confirm the server refuses it:

```
ssh -o PreferredAuthentications=password -o PubkeyAuthentication=no dana@203.0.113.10
# dana@203.0.113.10: Permission denied (publickey).
```

Disabling passwords before testing key login can leave the server unreachable. Keeping an open session helps, but a provider console is the independent recovery path. Confirm you can actually log into that console before you need it at 3am.

Validate the SSH configuration and open a second key-only session before reloading. Test password and root login failures, then prove the provider recovery path remains available.

Check your understanding: secure access

Check server assets, accounts, sudo, SSH keys, root login, and recovery access.

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

Take this interactive quiz online

Network exposure

Inventory listeners, configure the firewall, terminate TLS, and keep private services private.

Inventory listening services

Find which processes listen on which interfaces and decide whether each port should be public, private, or local-only.

A firewall rule is easier to reason about when we know what is actually listening. Start from the process and interface.

Use `ss` to inspect TCP and UDP listeners, then map each socket to its service and owner:

```
sudo ss -lntup
```

```
Netid State  Local Address:Port  Process
tcp   LISTEN 0.0.0.0:22          users:(("sshd",pid=712,fd=3))
tcp   LISTEN 0.0.0.0:5432        users:(("postgres",pid=1103,fd=6))
tcp   LISTEN 127.0.0.1:3000      users:(("node",pid=1288,fd=19))
```

Read the local address column carefully. `127.0.0.1:3000` is loopback only: reachable by processes on this host, invisible from the network. `0.0.0.0:5432` listens on every interface the server has.

PostgreSQL on `0.0.0.0:5432` is reachable through every server interface even if the app uses localhost. A firewall may hide it today, but binding it narrowly removes another exposure path.

Fix the binding at the source

A database used only by the local application should bind to loopback or a private network. For PostgreSQL that is one line in `/etc/postgresql/16/main/postgresql.conf`:

```
listen_addresses = 'localhost'
```

Restart the service and run `sudo ss -lntup` again — the entry should now read `127.0.0.1:5432`. Remove services the server does not need at all instead of rebinding them.

Verify from outside

The listener table is the inside view. Confirm the outside view from another machine:

```
nc -vz 203.0.113.10 5432
# nc: connect to 203.0.113.10 port 5432 (tcp) failed: Connection refused
```

Capture UDP listeners and container-published ports too. A service may not appear in the expected application configuration while Docker or a socket-activated unit still exposes it. A `-p 8080:8080` in a compose file publishes the port on all interfaces, and it will not show up in any config file under `/etc`.

Save listener output with process, owner, address, port, and purpose. Stop or rebind one unnecessary listener, then verify locally and from an external host that only intended paths work.

Configure a default-deny firewall

Allow only required inbound services and stage firewall changes so SSH recovery remains possible.

The firewall should express the server's exposure map. Deny unsolicited inbound traffic, then allow the small set of public services.

On Ubuntu, UFW provides a clear interface over the host firewall. The safe sequence is: set the default policy, allow SSH, then enable.

Set policy and allow SSH first

Allow the tested SSH port before enabling the firewall. In the other order, `ufw enable` drops your own connection path:

```
sudo ufw default deny incoming
sudo ufw default allow outgoing
sudo ufw allow OpenSSH
sudo ufw enable
```

`OpenSSH` is an application profile that maps to `22/tcp`. If your sshd listens on another port, allow that port explicitly instead.

Then add HTTP and HTTPS as needed:

```
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp
```

Verify the ruleset

```
sudo ufw status verbose
```

```
Status: active
Default: deny (incoming), allow (outgoing), disabled (routed)

To                         Action      From
--                         ------      ----
22/tcp (OpenSSH)           ALLOW IN    Anywhere
80/tcp                     ALLOW IN    Anywhere
443/tcp                    ALLOW IN    Anywhere
```

The `deny (incoming)` default is the line that matters. A default deny policy turns a forgotten new listener into a local problem instead of a public service. The tradeoff is operational: one missing SSH rule can cut off administration.

Confirm from another machine that an allowed port answers and everything else does not:

```
nc -vz 203.0.113.10 443   # succeeds
```

```
nc -vz 203.0.113.10 5432  # times out
```

A firewall limits reachability but does not patch or authenticate the service behind it. Treat the firewall as a second boundary, not as permission to bind every service publicly. Keep databases on private interfaces and verify the rules from outside the host, where an attacker would meet them.

Record the expected public ports, enable matching rules, and scan from another machine. Prove an allowed web port responds, a closed test port fails, and the tested SSH session survives a firewall reload.

Terminate TLS deliberately

Use a maintained reverse proxy or platform endpoint for HTTPS and keep upstream connections, redirects, certificates, and headers explicit.

HTTPS protects traffic to the TLS endpoint. Know exactly where decryption happens and what path follows it.

On a single VPS the usual answer is: Nginx terminates TLS on port 443 and forwards plain HTTP to the application on loopback. That is a fine design — as long as you chose it, and can state it.

One host, one clear boundary

```
server {
    listen 443 ssl;
    server_name blog.flaviocopes.com;

    ssl_certificate     /etc/letsencrypt/live/blog.flaviocopes.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/blog.flaviocopes.com/privkey.pem;

    location / {
        proxy_pass http://127.0.0.1:3000;
        proxy_set_header X-Forwarded-Proto https;
        proxy_set_header X-Forwarded-For $remote_addr;
    }
}
```

Bind the application to loopback when only the reverse proxy should reach it. Nginx may terminate TLS while forwarding plain HTTP to an app on loopback. That is reasonable on one host, but the same hop across a private network needs its own protection decision.

Redirect HTTP deliberately instead of leaving port 80 serving content:

```
server {
    listen 80;
    server_name blog.flaviocopes.com;
    return 301 https://$host$request_uri;
}
```

Keep renewal automated and proven

Let's Encrypt certificates expire after 90 days, so automate certificate renewal and rehearse it:

```
sudo certbot renew --dry-run
```

```
# Congratulations, all simulated renewals succeeded
```

Test certificate expiry and renewal alerts. The classic failure is silent: renewal broke months ago, nothing complained, and you learn about it from a browser error on launch day. An external monitor that checks days-to-expiry catches this while it is still boring.

Guard the identity headers

Pass trusted proxy headers through a controlled boundary. Proxy identity headers need the same boundary. Remove caller-supplied copies at the edge, then add trusted values so the application never confuses internet input with proxy evidence. In the snippet above, `proxy_set_header` overwrites whatever `X-Forwarded-Proto` a client tried to smuggle in.

Record the protocol, certificate owner, and trust boundary for every browser-to-database hop. Force HTTP, an expired test certificate, and a direct app-port request, then prove each fails or redirects as designed.

Keep private services private

Restrict databases, dashboards, metrics, queues, and control panels to local or private access with their own authentication.

Operational interfaces often reveal more power and data than the public application. Do not expose them merely because they have a login screen.

A database login page is still an exposed parser and authentication service. Every packet that reaches it exercises code that was never meant to face the open internet, before any password is checked.

Choose a private path

Use loopback, private networking, VPN access, provider firewalls, or an authenticated tunnel. The lightest option: bind the service to loopback and reach it over SSH when you need it:

```
ssh -L 5432:localhost:5432 dana@203.0.113.10
```

While that tunnel is open, `psql -h localhost -p 5432` on your laptop talks to the server's PostgreSQL, wrapped inside your authenticated SSH session. Nothing new is exposed.

VPN or tunnel access adds operational dependency, but removes routine internet traffic from the interface.

Keep authentication on

Keep service authentication enabled even on private paths. The private network is one control and the password is another. You want both, because a mistake in either one should not be enough to open the data.

Check both firewall layers

Review cloud firewall and host firewall together so one layer does not create a false sense of safety. Test both network layers independently. The provider firewall limits paths before packets reach the host, while the host firewall protects against mistakes or traffic arriving through another interface.

From an external machine, prove the private port really is unreachable:

```
nc -vz 203.0.113.10 5432
# nc: connect to 203.0.113.10 port 5432 (tcp) failed: Connection refused
```

The recurring mistake: a dashboard exposed ”for a quick demo” with `ufw allow 8080`, never removed. You recognize it by re-running the listener and firewall inventory on a schedule — an allow rule with no matching entry in your exposure map is a finding, not background noise.

Administer the lab database through loopback, a VPN, or an authenticated tunnel. Scan its public address and prove the database port is closed while local authentication still rejects a wrong password.

Check your understanding: network exposure

Check listening services, firewalls, TLS, reverse proxies, databases, and administrative interfaces.

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

Take this interactive quiz online

System hardening

Patch the system, remove services, restrict processes, and protect configuration.

Apply security updates

Track Ubuntu support, install security fixes promptly, plan reboots, and test application health after changes.

Known vulnerabilities become easier to exploit after public disclosure. Keep the operating system inside a supported update path.

An Ubuntu LTS release receives standard security fixes for five years. A server on an unsupported release gets nothing, no matter how carefully you run the commands below.

Review and apply pending fixes

Refresh package metadata, review pending updates, apply security fixes:

```
sudo apt update
apt list --upgradable
sudo apt upgrade
```

Then check whether a reboot is required:

```
cat /var/run/reboot-required
# *** System restart required ***
```

A kernel fix may install successfully but remain inactive until reboot. Record the running kernel and package version separately. That evidence prevents a successful package command from being mistaken for a completed fix when the vulnerable code remains loaded:

```
uname -r
# 6.8.0-45-generic   <- still the old kernel until you reboot
```

Automate the security channel

Ubuntu ships `unattended-upgrades` to install security fixes on their own:

```
sudo apt install unattended-upgrades
sudo dpkg-reconfigure -plow unattended-upgrades
```

Answering yes writes `/etc/apt/apt.conf.d/20auto-upgrades`, which turns on daily security updates. Automate updates only with monitoring and a recovery plan. Automatic updates shorten exposure, while staged updates make application regressions easier to catch.

My advice for a single-purpose VPS: enable unattended security updates for the OS, and move application dependencies through your normal deploy process where you can test them.

Snapshotting is useful, but it does not replace a tested application rollback and backup.

Check health after every update

An update that breaks the application is still an outage. After upgrading and rebooting, hit the health endpoint and watch the service logs before you walk away:

```
curl -s https://blog.flaviocopes.com/health
# ok
```

Record pending packages, installed versions, reboot state, and service checks before and after updating. Break one health check or use a disposable failed service, then prove monitoring detects the incomplete maintenance.

Disable unneeded services

Remove or stop packages, daemons, timers, and sockets the server does not need to reduce code, privileges, and exposed behavior.

Every running service adds code and configuration you must trust. If the product does not need it, remove the obligation.

Find what runs

Review enabled systemd units, installed server packages, scheduled jobs, and container workloads:

```
systemctl list-units --type=service --state=running
systemctl list-unit-files --state=enabled
```

Check timers and sockets as well as services. A stopped unit can start again when its socket receives traffic or a scheduled timer fires after the review:

```
systemctl list-timers
systemctl list-sockets
```

Confirm ownership before disabling anything. "I don't recognize it" is not the same as "nothing needs it". Read what the unit does with `systemctl cat exim4`, find which package installed a file with `dpkg -S`, and check whether the application or a backup job depends on it.

Stop, disable, or remove

A default image may enable a mail daemon the application never uses. Stopping it removes the listener now, while disabling or removing it prevents the next reboot from restoring exposure:

```
sudo systemctl disable --now exim4
```

If nothing on the server needs the package at all, go one step further and `sudo apt purge exim4`. A removed package is one less thing to patch, configure, and audit for the life of the server.

Verify the exposure is gone

Recheck listeners afterward and document services the application depends on:

```
sudo ss -lntup
```

The mail port should be gone from the list. Reboot the lab machine once and check again — the reboot is what catches a socket or timer quietly reactivating the unit you thought was dead.

Save the enabled-unit and listener evidence for one unneeded service. Disable it, reboot the lab host, and prove the port stays closed while the application's health checks still pass.

Restrict files and processes

Use dedicated service users, narrow filesystem permissions, controlled working directories, and systemd sandboxing where the application supports it.

A web application rarely needs access to the whole server. Give its process a small part of the filesystem and no interactive login.

A dedicated service user

Run each service as a dedicated unprivileged user:

```
sudo adduser --system --group --home /srv/blog blog
```

The `--system` flag creates an account with no password and a non-login shell. It exists to own a process, not to be logged into.

Narrow the filesystem

Make code read-only to the process and grant writes only to explicit data directories:

```
sudo chown -R root:blog /srv/blog/app
sudo chmod -R 750 /srv/blog/app
sudo chown blog:blog /srv/blog/uploads
```

Now the `blog` user can read and execute the application but cannot modify it. A compromised web process that can write its own executable can make persistence easy. Read-only code and one writable upload directory reduce damage, but overly strict rules can break legitimate updates.

Let systemd enforce it

The service unit can lock this in at the process level:

```
[Service]
User=blog
NoNewPrivileges=true
ProtectSystem=strict
ProtectHome=true
PrivateTmp=true
ReadWritePaths=/srv/blog/uploads
```

`ProtectSystem=strict` mounts the entire filesystem read-only for this process, and `ReadWritePaths` opens exact exceptions. Add systemd protections gradually, test them, and avoid giving application users sudo access.

Apply sandboxing one capability at a time and watch service logs. The goal is verified confinement, not a long directive list that operators disable after an unexplained outage. When a directive breaks the app, the journal usually shows a permission error naming the path — that is your cue to add a `ReadWritePaths` entry, not to remove the protection.

Prove the confinement

```
sudo -u blog touch /srv/blog/app/server.js
# touch: cannot touch '/srv/blog/app/server.js': Permission denied
sudo -u blog touch /srv/blog/uploads/probe.txt
# succeeds
```

Record the service user and every required read or write path. Attempt writes to code, configuration, and the approved data directory, then prove only the intended data write succeeds.

Store server secrets outside code

Deliver credentials to the service through protected configuration, limit readers, rotate values, and keep secrets out of commands and logs.

A secret committed with the application travels through every clone, build, and backup of that repository.

Keep credentials out of the code, out of the shell history, and out of process listings. Deliver them to the service through protected configuration instead.

A root-controlled environment file

Use the deployment platform or a root-controlled configuration file with narrow permissions:

```
sudo install -m 640 -o root -g blog /dev/null /etc/blog/env
sudoedit /etc/blog/env
```

```
# /etc/blog/env
DATABASE_URL=postgres://blog:9sK2mfA81vTq@localhost/blog
```

Root owns the file, the `blog` group can read it, and mode `640` keeps everyone else out. Using `sudoedit` matters: typing the value into an `echo` command would leave it in your shell history. The systemd unit then loads it:

```
[Service]
EnvironmentFile=/etc/blog/env
```

Avoid command-line arguments that appear in process listings. Anything passed as an argument is visible in `ps` output to every user on the machine for as long as the process runs.

Verify the boundary

```
sudo -u www-data cat /etc/blog/env
# cat: /etc/blog/env: Permission denied
```

Separate production and development values and document rotation without printing the value. A rotation note should say where the credential lives and who ran the change, never what it was.

Rotation is part of the move

Moving a password from Git into a world-readable environment file does not solve exposure. The old value also remains valid until you rotate it. Check backups and deployment logs for older copies. Rotation ends the credential's authority, while removing residual plaintext reduces future accidental disclosure and investigation noise.

Move one credential to root-controlled configuration and record its mode, owner, and readers without printing it. Prove the service starts, an unrelated user cannot read it, and the rotated old value no longer authenticates.

Check your understanding: system hardening

Check updates, services, permissions, secrets, process isolation, and secure configuration.

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

Take this interactive quiz online

Observe and back up

Read logs, detect drift, create alerts, and prove backups through restores.

Read authentication and service logs

Use the system journal and authentication records to investigate access, failures, restarts, privilege use, and application behavior.

Logs show what the server observed. Learn the normal records before you need them during an incident.

On Ubuntu, the systemd journal collects everything, and the classic text files still exist under `/var/log`. Both views are useful.

Query the journal

Use `journalctl` by unit, time, priority, and boot:

```
sudo journalctl -u ssh --since today
sudo journalctl -p err -b
sudo journalctl -u nginx --since "14:00" --until "15:00"
```

The first shows today's SSH activity. The second shows every error-priority message since the last boot — a fast way to spot a crashing service. The third narrows one unit to a suspicious hour.

Know the lines that matter

Authentication events also land in `/var/log/auth.log`:

```
sudo grep sshd /var/log/auth.log | tail -3
# Failed password for invalid user admin from 198.51.100.7 port 51022 ssh2
```

```
# Failed password for invalid user admin from 198.51.100.7 port 51040 ssh2
# Accepted publickey for dana from 203.0.113.44 port 55910 ssh2: ED25519 SHA256:kq2R...
```

Review SSH and sudo events, service restarts, kernel messages, and application failures. A successful SSH login after many failures matters more than either event alone. Sudo entries in the same file record the user, the working directory, and the exact command that ran.

Get copies off the host

Send important logs off the server so a compromise cannot erase the only copy. Local logs help investigation, but an attacker with root may edit or erase them. A remote syslog target or a log-shipping agent turns the journal into evidence the host cannot rewrite after the fact.

Synchronize clocks and preserve stable host identifiers. Without consistent time and source information, responders cannot reliably connect provider, firewall, SSH, and application events. Check that `timedatectl` reports `System clock synchronized: yes` on every machine involved.

Generate a success, failure, sudo command, and service restart with known timestamps. Find each event locally and remotely, then stop log forwarding and prove the missing stream creates an alert.

Notice unexpected change

Establish a baseline for users, services, listeners, packages, scheduled jobs, files, and resource use so suspicious drift becomes visible.

You cannot recognize unusual server behavior without a picture of normal behavior. Keep the baseline small enough to review.

Capture the baseline

Watch new accounts, authorized keys, sudo rules, enabled units, listeners, packages, cron jobs, application files, CPU, disk, and outbound traffic. Save the stable parts as plain text you can diff later:

```
{
  getent passwd | awk -F: '$3 >= 1000'
  sudo sha256sum /home/*/.ssh/authorized_keys
  sudo ls /etc/sudoers.d/
  systemctl list-unit-files --state=enabled
  sudo ss -lntu
  ls /etc/cron.d/ /etc/cron.daily/
} > baseline-2026-08-03.txt
```

Store the file off the server, next to your threat-model notes. A baseline stored on the host can be edited by the same attacker it is supposed to expose.

Compare on a schedule

Weeks later, run the same capture into a new file and diff the two:

```
diff baseline-2026-08-03.txt baseline-2026-09-01.txt
# > 99-deploy-tmp
```

One new file in `/etc/sudoers.d/` you cannot explain — that single line is exactly the needle this habit exists to find.

A new cron job may be a deployment change or attacker persistence. A reviewed baseline provides the evidence needed to distinguish expected work from drift. Configuration management and deployment manifests make unexpected differences easier to see: when the server is built from a script, the script itself is the baseline.

Keep the noise out

Exclude expected volatile data such as temporary files and counters. A smaller, deliberate baseline makes a changed key or unit visible instead of burying it in harmless differences.

The failure mode is a diff of 400 lines of routine package bumps every week. You stop reading it, and that is how the one real change slips through. Drop version numbers from noisy sections, or split packages into their own report reviewed less often.

Save a small baseline of users, keys, sudo rules, listeners, units, packages, and scheduled jobs. Add one harmless unauthorized change and prove the comparison names it without flooding the report with normal log growth.

Design actionable alerts

Alert on meaningful access, privilege, availability, and integrity signals with an owner and an immediate investigation path.

An alert nobody investigates is noise. Start with events that could cause real harm.

Examples include a new sudo user, an unexpected listening port, repeated SSH failures followed by success, a disabled firewall, full disk, or changed deployment files. Each of those is rare, meaningful, and has an obvious first response. That combination is what makes an event worth waking someone for.

Define the whole alert, not just the trigger

An alert definition needs more than a threshold. Write down who receives it and what they do first:

```
signal:    new file in /etc/sudoers.d/
threshold: any occurrence
owner:     flavio
evidence:  baseline diff + auth.log for the same hour
first act: match against the deployment log; if absent, treat as incident
```

Set thresholds from normal traffic and test notification delivery. Alerting on every failed SSH login creates noise. Alerting when failures are followed by success highlights a plausible account attack but can still catch a forgetful administrator.

Deliver through an independent path

Send alerts through a path independent from the host when possible. A full disk or compromised server may break the same local mail command that was supposed to report it. An external uptime monitor, a provider metric alarm, or a push service running elsewhere keeps talking when the host cannot speak for itself.

Fire it on purpose

An alert you have never triggered is a hope, not a control. Create the sudoers file in the lab, watch the notification arrive, then check the quiet case too:

```
sudo journalctl -u ssh --since -1h | grep -cE 'Failed password'
# 3   <- below threshold, no alert expected
```

The long-term failure mode is fatigue. When a channel fills with ignorable messages, the one that matters gets dismissed with the rest. My rule: an alert that fired three times without causing any action gets tuned or deleted.

Define signal, threshold, owner, evidence link, and first action for three alerts. Trigger one real alert and one near-threshold case, then prove notification delivery and suppression both behave as documented.

Back up and restore

Protect data and configuration with separate, encrypted, versioned backups and prove recovery on a clean system.

A backup job is only a promise. A restore test is evidence.

Decide what must survive

Back up application data, required configuration, and recovery instructions without blindly copying live secrets. For a typical VPS that means the database, uploaded files, and the handful of config files you wrote by hand — not a raw copy of the whole disk:

```
sudo -u postgres pg_dump blog > /backup/blog-2026-08-03.sql
tar czf /backup/blog-files-2026-08-03.tar.gz \
  /srv/blog/uploads /etc/blog /etc/nginx/sites-available
```

Keep copies out of reach

Keep copies separate from the server and protect deletion. A backup stored under the production root credential can be deleted during the same compromise. Independent access and version retention improve survival, but require a separate recovery procedure.

Ship the archives somewhere the server can write but not delete: object storage with versioning enabled, or a backup host that pulls from the server instead of receiving pushes.

Restore is the real test

Restore into a clean environment, verify integrity, and record recovery time and missing steps:

```
sudo -u postgres createdb blog_restore
sudo -u postgres psql blog_restore < blog-2026-08-03.sql
sudo -u postgres psql blog_restore -c "select count(*) from posts;"
#  count
# -------
#   1773
```

Verify application meaning after restoration, not only file presence. Row counts, checksums, a login, and one representative read provide stronger evidence that the recovered system is usable.

The classic failure looks like this: nightly dumps ran for a year, but the database password changed in March, and every archive since then contains an error message instead of data. Nobody noticed because nobody restored. You catch this by alerting on archive size drops and by rehearsing the restore on a schedule.

Restore application data and required configuration onto a clean host and record checksums, missing steps, and recovery time. Delete or corrupt one source file, then prove an older protected version restores correctly.

Check your understanding: observe and back up

Check logs, process and file changes, alerts, backups, restore tests, and protected evidence.

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

Take this interactive quiz online

Respond and rebuild

Investigate signals, contain the host, rotate credentials, and rebuild from trusted inputs.

Recognize a possible compromise

Treat unexpected accounts, keys, listeners, processes, outbound traffic, files, and privilege events as evidence to investigate, not proof to ignore or panic over.

One strange process does not prove compromise. It does justify preserving evidence and checking what changed.

The signals worth stopping for: an account you did not create, an authorized key you cannot place, a listener on an unexpected port, a process with a slightly misspelled system name, outbound traffic to an address nothing should call, or a sudo event at an hour nobody was working.

Record before you react

Record time, symptoms, recent deployments, access events, network connections, process trees, and relevant provider activity:

```
date -u
ps auxf > /tmp/evidence-ps.txt
sudo ss -tnp > /tmp/evidence-net.txt
last -a | head -20
sudo stat /home/dana/.ssh/authorized_keys
```

That `stat` matters more than it looks. The modification time on a key file tells you when access changed, which anchors the whole timeline you are about to build. Copy these captures to another machine right away.

Avoid installing many new tools on the suspect host. Every package install rewrites parts of the filesystem you are trying to read, and a compromised host may hand you tampered tools anyway.

Trust independent sources first

Collect evidence from independent systems first when practical. Provider audit logs and remote authentication records remain useful even if timestamps or files on the host were changed. Compare against known-good configuration and independent logs — the baseline file and the forwarded logs you set up earlier are the reference this moment was built for.

Do not destroy context

An unknown SSH key may come from automation, a former administrator, or an attacker. Deleting it immediately reduces access but can destroy context before you identify its origin and use. Capture first, then act.

The classic mistake is rebooting "to see if it clears up". A reboot wipes memory, running processes, and active connections — often the only volatile evidence — while leaving any persistence exactly where it was.

Build a timeline from key metadata, authentication logs, provider events, processes, and network connections. Test the checklist with a harmless planted key and prove it preserves evidence before containment begins.

Contain the host

Limit attacker access and outgoing harm using provider controls, credential revocation, isolation, and carefully recorded actions.

Containment protects other systems and users. Do it deliberately so you do not lose the only evidence or recovery path.

Once you believe the host is compromised, the job changes. You are no longer administering a server. You are limiting what it can do next.

Cut access from outside the host

Isolate the host from public traffic or detach it behind provider controls, revoke affected keys and tokens, and block known malicious paths. Prefer the provider's firewall and console over commands on the machine itself. Do not trust the compromised host to prove it is clean, and do not trust it to isolate itself either. Provider firewall controls can isolate it without trusting commands run inside the suspect host.

Revocation also happens off the host: the deploy token in CI, the database password in the secret store, the API keys the server held in its environment. Anything the machine could read, treat as stolen.

Choose what to preserve

Decide which evidence is time-sensitive before powering down. Memory and active connections disappear, while continued operation may allow more harm, so the incident lead must choose deliberately. Snapshot where policy allows and keep an action timeline:

```
14:02 UTC  alert: outbound connections to 185.220.101.34 from blog user
14:09 UTC  provider firewall: inbound blocked except admin IP (flavio)
14:15 UTC  disk snapshot taken (id: snap-8842107)
14:21 UTC  deploy token revoked, DATABASE_URL password rotated
14:30 UTC  ss/ps/last captures copied to external storage
```

Every line has a time and an actor. Memory is unreliable during an incident, and the timeline is what lets you reconstruct decisions afterward — including for the people the incident affects.

Disconnecting the VPS stops public abuse but may also cut off volatile evidence and customer traffic. There is no free option here, only a chosen one.

Write and rehearse the containment order using a disposable VPS and provider-side controls. Prove public traffic stops, affected credentials fail from another machine, and the action timeline and evidence copy remain available.

Rebuild from known-good inputs

Replace a compromised server from trusted images, reviewed configuration, patched code, rotated secrets, and verified data instead of cleaning it in place.

After privileged compromise, proving every hidden change is gone is difficult. Rebuilding is usually the clearer recovery path.

Removing the visible backdoor does not prove a root-compromised host is clean. A rebuild costs time, but creates a baseline you can reproduce and review.

What "known-good" means

Create a new host from a supported image, apply reviewed automation, deploy patched artifacts, rotate all reachable credentials, and restore verified data. Every input must come from somewhere the attacker could not touch:

```
image:    provider Ubuntu 24.04 LTS image, not a snapshot of the old host
config:   setup script from Git, reviewed after the incident
app:      artifact rebuilt in CI from patched source
secrets:  all new values - the old ones are burned
data:     backup checked for integrity and inspected for planted content
```

Do not copy executables or system configuration from the suspect filesystem. Restore only required data after validation, and keep the old host isolated for the agreed evidence-retention period.

Verify before exposing

Compare configuration and expose traffic gradually while monitoring. Diff the new host's package list against what the setup script is supposed to install:

```
dpkg -l | awk '/^ii/ {print $2}' > new-host-packages.txt
diff expected-packages.txt new-host-packages.txt
```

An empty diff means the machine matches its manifest. Then prove the old credentials really are dead:

```
ssh -i ~/.ssh/old_lab_key dana@203.0.113.10
# dana@203.0.113.10: Permission denied (publickey).
```

Point DNS at the new host, watch it under real traffic for a day, and keep the old machine isolated rather than deleted.

The tempting shortcut is restoring "just one file" from the old disk — a binary or a config that is annoying to regenerate. That single file can be the persistence you rebuilt to escape. If it cannot be rebuilt from source, treat it as data to inspect, not software to run.

Build a fresh host from the supported image, reviewed configuration, patched artifact, rotated secrets, and verified data. Compare it with the intended manifest, then prove an old credential and an unlisted package are absent.

Complete the server security review

Verify access, exposure, updates, privileges, secrets, observability, backups, recovery, and ownership as one repeatable operating checklist.

Server security is maintenance, not a one-time installation step. Make the review repeatable after every meaningful infrastructure change.

Everything in this course collapses into one operating loop: check access, check exposure, check updates, check evidence, and prove recovery still works.

A checklist with evidence attached

Reconcile accounts and keys, listeners and firewall rules, packages and services, file permissions, secret inventory, alerts, backup restores, and incident contacts. For each line, capture the command output that proves the state — not a checkbox:

```
access:    getent group sudo matches admin roster          2026-08-03  flavio
ssh:       sshd -T shows permitrootlogin no                2026-08-03  flavio
exposure:  ss -lntup matches map (22, 80, 443 only)        2026-08-03  flavio
updates:   apt list --upgradable empty, kernel current     2026-08-03  flavio
secrets:   /etc/blog/env root:blog 640, rotated in Q3      2026-08-03  flavio
alerts:    sudoers-change alert fired in test              2026-07-28  flavio
restore:   pg restore drill, 14 min, row counts match      2026-07-20  flavio
```

A checklist that only records ”done” becomes stale quickly. Evidence such as fingerprints, package versions, listener output, alert tests, and restore times makes the review repeatable.

Date every artifact and name its owner. A passing result has limited value after accounts, packages, firewall rules, or deployment architecture change without another review.

Exceptions are findings with deadlines

Record exceptions with an owner and deadline. ”Port 8080 open for the migration, close by Aug 15, owner marco” is a managed risk. The same open port with no note is drift, and drift is where the previous lessons' work quietly erodes.

Close the loop on the lab

Remove the lab server when the course is complete, then verify from the outside that it is really gone:

```
nc -vz 203.0.113.10 22
# nc: connect to 203.0.113.10 port 22 (tcp) failed: Operation timed out
```

A ”deleted” server that still answers means something is still running, still billed, and still attackable.

Run the final review and save non-secret evidence for access, exposure, updates, permissions, alerts, and recovery. Introduce one disposable mismatch, prove the review catches it, then remove the lab server and verify it no longer responds.

Check your understanding: respond and rebuild

Check compromise signals, containment, evidence, credential rotation, rebuilding, recovery, and post-incident work.

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

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/linux-server-security/.