Engineer

Hardening

removing what an attacker could use

What it is

Hardening is the work of configuring a system so that it exposes and permits only what its job needs: fewer running services, fewer listening ports, fewer privileges for each process, and stricter defaults on everything that stays. The software is the same before and after. What changes is how much of it an intruder can reach and how far a single foothold carries. You will also see it called system hardening, server hardening, OS hardening or SSH hardening, depending on which layer is being tightened.

A typical hardening pass on a Linux server touches several layers at once. Remote login accepts keys and refuses passwords. The root account cannot log in directly. Packages nobody uses are removed rather than merely stopped. Each long-running service gets its own unprivileged user, a read-only view of the file system, and a list of the system calls it may make. The firewall admits the handful of ports the host exists to serve. None of these steps is exotic; the discipline is in doing all of them, on every host, and keeping them true.

It helps to place hardening next to the practices it is often confused with. Patching fixes a specific flaw in code once a fix exists. Monitoring and intrusion detection notice an attack in progress. A firewall decides who may reach a port. Hardening decides what the program behind that port is allowed to do once it is reached, and whether the port should be open at all. Published baselines such as the CIS Benchmarks turn this into checklists per operating system and product, which is useful as a floor and misleading as a finish line.

Use the concept in a decision

Put “What is hardening?” to work in a real decision. We can help relate the concept to the system you are building or buying. Accounts, passwords and permissions arranged so people hold exactly the access their job needs, and leavers lose it that day.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Wrote 1,000 RBAC rules to secure Linux servers — 1,000 access rules written to close a security gap

Wrote 1,000 RBAC rules to secure Linux servers Get in touch →

Why it matters

The real decision is where to spend a limited security budget, and hardening is unusual in that it pays off against vulnerabilities that will only be found next year. A patch protects against one known vulnerability. A hardened service restricts what an exploited process can do. For example, denying writes to executable memory, raw sockets and access to home directories is designed to limit the impact of a remote code execution bug. Patching closes holes; hardening lowers what any hole is worth. A serious operation needs both, and hardening is what buys time in the days between a disclosure and the fix landing.

The cost is mostly engineering time and breakage risk rather than licences. Every restriction is a bet that the service never needs the thing being taken away, and some of those bets lose: a backup job that runs once a night, a log rotation that sends a signal, a plugin that shells out. That is why hardening done by hand on a live server tends to stall. The person doing it gets burned once by a restriction that broke something at three in the morning and stops tightening. Done as code, with the restrictions reviewed, versioned and applied identically on every rebuild, the same work becomes cheap to repeat and safe to roll back.

For a buyer, the practical questions are when to do it and how to know it was done. The answer to the first is at build time, as part of the automation that creates the host, because a checklist applied once to a running machine drifts within months. The answer to the second is evidence from the running system rather than from the configuration repository. Auditors increasingly ask for both: a documented baseline and proof that production matches it.

What it looks like in practice

On this company's own production host, SSH is the only interactive way in, and its configuration is written by automation. In September 2026 an access report read the daemon's resolved settings back and found two of the intended directives overridden: the operating system's defaults loaded ahead of the company's drop-in file, and for those settings the first value wins. A filename prefix that sorts earlier fixed the order. The lasting change was three-stage validation of 24 SSH directives on every converge: each candidate must parse before it replaces

the live file, the configuration as assembled must pass a check once installed, and then the settings the daemon actually resolved are read back and asserted against all 24. Two further directives were left out on purpose because the daemon no longer implements them.

The same host runs 13 unit files installed by its own automation, and they had been running with the service manager's default privileges. Each unit template received a privilege ceiling, a system-call filter, restricted address families and memory that may be writable or executable but never both. Measured with the supervisor's own scoring, the sandbox work moved a dead-man's-switch service from 9.6 to 1.5, the internet-facing git forge from 8.3 to 1.5, and the two backup units from 9.6 to 2.3 and 2.5. While writing them, one directive turned out to be misspelled in three templates; the supervisor logs an unrecognised key and starts the unit regardless. Every unit now passes through the supervisor's own verifier during the converge.

A hardening review in August 2026 asked whether the SSH brute-force protections worked, and read the running system to find out. All three protections had been inert since the host was built: the ban jail watched port 22 while the daemon listened on 1986, a firewall rate limit sat below a broader rule that matched first, and the ban action called a packet-filter binary the package system only recommends, which was absent. Two rules came out of that review and still govern the repository: a security control gets an assertion, not a comment, and a firewall is verified by the position of its rules, not by their presence.

Where it goes wrong

The setting is in the file and the service ignores it. Many daemons read several files and keep the first or the last value they meet for each key. A hardened directive in a drop-in can lose to a vendor default that loads earlier. Ask the program what it will actually do, for SSH with `sshd -T`, and compare that output with the intended values on every run.

A typo turns a control into a comment. Service managers and many daemons log an unknown key and carry on. The unit starts, the dashboard is green, and the restriction is absent. Run the parser's own verifier as part of deployment so that an unrecognised directive stops the rollout.

```
# What the SSH daemon will do once every Include is resolved
sshd -T | grep -E '^(passwordauthentication|permitrootlogin) '
# How much a unit's sandbox still exposes, scored 0.0 to 10.0
systemd-analyze security --no-pager example.service
# Unknown directives and other unit-file errors
systemd-analyze verify /etc/systemd/system/example.service
```

A protection guards the wrong door. A port moves and the ban rule keeps watching the old one. A rate limit is added below a broad allow rule that already matched. The control reports healthy because it is running, which says nothing about whether it ever fires. Test each control by its effect: trip it on purpose from outside, and confirm the packet is dropped.

The sandbox breaks the service at night. A system-call filter or a read-only path breaks the rare code path, such as the weekly export or the backup hook, and the failure appears hours after the change, at runtime rather than at parse time. Tighten in stages, exercise every scheduled job after each stage, and watch the journal for processes killed by the filter.

The benchmark score becomes the goal. A host can pass every line of a generic checklist while its one internet-facing application runs as root with write access to its own code. Start from what is reachable from the network and what each reachable process could touch, then use the benchmark to catch what that review missed.

Further reading

- Hardening, in the NIST glossary — the definition NIST uses, with the publication it is drawn from.
- sshd_config(5) — every SSH server directive, and the rule that the first value obtained for a keyword is the one used.
- systemd.exec(5) — the sandboxing directives for services: system-call filters, address families, write-or-execute memory.
- systemd-analyze(1) — the `security` verb's exposure score and the `verify` verb that catches unknown directives.
- CIS Benchmarks — consensus configuration baselines for operating systems, cloud platforms and server software.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is hardening?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/hardening/