

DevSecOps cheat sheet

DevSecOps is DevOps with the security work built into the same pipeline that builds and releases the code. What to check at every stage, the tools that do it, and the frameworks to build on.

PIPELINE CONTROLS · TOOLING · SUPPLY CHAIN · NIST SSDF · OWASP · SLSA · CIS

General educational guidance, not affiliated with or endorsed by any project or vendor named. Try the commands in a test pipeline first. Framework references follow NIST SP 800-218 (SSDF) v1.1, OWASP ASVS 5.0, SLSA v1.2, CIS Controls v8.1, and the other NIST, OWASP and CIS documents named inline. CIS is the Center for Internet Security. Refer to cisecurity.org/controls for the current version of the CIS Controls. Published 2026-09-06.

WHERE SECURITY FITS THE PIPELINE

Every stage gets its own checks. The ones in the pipeline run on each change, the way tests do, so findings reach the developer while the change is still open and cheap to fix. That is what "shift left" means: each check moves to the earliest stage that can catch the problem.

```
plan -> code -> build -> test -> release -> operate
-----
1 plan      threat model; requirements from ASVS
2 code      review everything; secrets + SAST pre-commit
3 build     SCA + IaC + container scan; SBOM each build
4 test      DAST against a staging deploy
5 release   sign + provenance; verify before deploy
6 operate   monitor; patch what is exploited; feed back
```

SAST static analysis: reads source for vulnerable patterns before it ever runs. Semgrep. CodeQL is free on open-source code and needs a paid GitHub license for a private repo; its engine is not open source.

SCA software composition analysis: checks the packages you depend on against known-vulnerability databases. OSV-Scanner, Trivy, Gripe.

secrets scanning finds keys, tokens, and passwords in the repo and its history. gitleaks, TruffleHog.

IaC scanning infrastructure as code: reads Terraform, Kubernetes, and Dockerfiles for insecure settings before anything is provisioned. Trivy, Checkov.

DAST dynamic analysis: probes the running application from outside, the way an attacker would. ZAP.

SBOM software bill of materials: the machine-readable parts list of what an artifact contains. Syft.

provenance a signed record of how, where, and from what an artifact was built. Sigstore.

1 PLAN: THREAT MODEL THE DESIGN, THEN TAKE REQUIREMENTS FROM A STANDARD

Threat modeling asks four questions of each new design: what are we building, what can go wrong, what are we doing about it, and did we do enough? Write the answers down; they become test cases and review points downstream.

Take security requirements from OWASP ASVS 5.0 (the Application Security Verification Standard) rather than inventing your own. Level 1 is the starting set, Level 2 covers standard security practice, and Level 3 holds the advanced, high-assurance requirements. NIST's Secure Software Development Framework (SSDF, SP 800-218) covers what the organization has to set up around the code: requirements, roles, toolchains, criteria for security checks, and hardened development environments (its PO practices).

2 CODE: SECRETS AND SAST BEFORE THE PUSH

Run the fast checks as a pre-commit hook and again in CI: the hook answers the developer in seconds, and CI catches whoever skipped the hook.

```
# in CI: the working tree and the whole history
gitleaks git .      # scan the repo, history included
gitleaks dir .      # scan a plain directory
# in the hook: staged changes only, so it stays fast
gitleaks git --pre-commit --staged --redact
# SAST with the public registry rules; --config auto
# sends a hash of your project URL and usage metrics
# to Semgrep; a known URL is matchable from the hash
semgrep scan --config auto
# platform side: require review before merge, and turn on
# push protection so a secret is blocked at push time
# (on GitHub: free on public repos, paid on private)
```

Revoke and reissue any secret that leaked; deleting the commit does not undo the exposure. Code from an AI assistant enters through the same review and the same scanners as anyone else's change. Assistants also invent package names that do not exist, and attackers register those names ahead of time (slopsquatting), so check any suggested dependency against the registry before you install it.

3A BUILD: KNOW WHAT IS IN THE BUILD

```
osv-scanner scan -r . # lockfiles vs the OSV database
trivy fs .           # deps + secrets across the tree
grype sbom:./sbom.json # rescan this build's SBOM
                     # (section 5 writes it)
```

Commit lockfiles and pin versions so a build cannot silently pull something new; let a bot (Renovate, Dependabot) open update pull requests so patching stays small and routine. Set a cooldown so a version has to be a few days old before it installs: npm's min-release-age, Renovate's minimumReleaseAge. Most malicious releases are pulled within hours of publication. Transitive dependencies, the packages your dependencies pull in, are yours to patch too. Keep each build's SBOM in a store that re-checks it as new advisories are published (Dependency-Track).

The ChainDrop worm (August 2026) republished 400+ npm packages as malicious patch releases that ran on install. A committed lockfile holds the versions you chose if you install with npm ci, and npm 12 blocks dependency install scripts by default. A lockfile also pins a bad version once you have it, so bump it when the fix lands. Some of it ran from editor configuration the worm wrote into the repo, which no install flag stops. The worm collected npm, cloud, and repository credentials, so rotate every token a compromised install could reach.

When the next backdoor like xz-utils (CVE-2024-3094) is disclosed, the SBOM store answers "is it in a build we released, and where?" in minutes.

3B BUILD: SCAN CONTAINERS AND IaC

```
trivy image <image> # OS and app packages, secrets
trivy config .      # Terraform, Kubernetes, Dockerfiles
checkov -d .        # IaC checks, policy as code
```

Build images small: a non-root user (CIS Docker Benchmark v1.8.0, 4.1), a minimal base pinned by digest, and multi-stage builds so compilers stay out of the final layer.

In Kubernetes, block bad workloads as the cluster admits them. Pod Security Admission is built in: set a namespace to the baseline or restricted level and it rejects privileged pods with no extra tool. Add policy as code (Kyverno or OPA Gatekeeper) for what that does not cover, such as accepting only signed images. Audit nodes against the CIS Kubernetes Benchmark with kube-bench.

4 TEST: PROBE THE RUNNING APP

```
docker run -t ghcr.io/zaproxy/zaproxy:stable \
  zap-baseline.py -t https://staging.example.com
```

The baseline scan spiders the site briefly and reports what it can see without attacking: missing headers, cookie flags, error pages that reveal internals. It attacks nothing, so it can run on every staging deploy. The active scan (zap-full-scan.py) does attack: run it with permission and a maintenance window. An unauthenticated scan covers only what an anonymous visitor can reach.

5 RELEASE: SBOM, SIGNATURE, PROVENANCE

Generate the SBOM in the build, sign the artifact by digest with the pipeline's own identity, and publish provenance; at deploy, verify all of it mechanically.

```
syft <image> -o cyclonedx-json > sbom.json
# keyless artifact signing needs id-token: write on the
# job, and -y to answer the prompt that would stall CI.
# It writes the digest, the signing identity and the
# time to the public Rekor transparency log
cosign sign -y <image>@<digest>
# provenance.json is the predicate your build writes:
# source repo, commit, builder, build parameters
cosign attest -y --type slsaprovenance1 \
  --predicate provenance.json <image>@<digest>
cosign verify <image>@<digest> \
  --certificate-identity <the signing workflow> \
  --certificate-oidc-issuer <the CI provider>
cosign verify-attestation --type slsaprovenance1 \
  <image>@<digest> <the same two flags>
# on GitHub, actions/attest signs the provenance for
# you. Give the job id-token: write, attestations:
# write and artifact-metadata: write, then read it back
# with gh attestation verify: it goes to GitHub's own
# store rather than the registry. Private repos need
# GitHub Enterprise Cloud
# identity, on GitHub Actions: the workflow URL + ref
# https://github.com/acme/shop/.github/workflows/
# release.yml@refs/tags/v1.2
# issuer: https://token.actions.githubusercontent.com
```

SLSA v1.2 (Supply-chain Levels for Software Artifacts) grades the build: L1 means provenance exists, L2 that a hosted platform generated and signed it, and L3 that the build platform resists tampering. Its Source Track grades the repository on its own four levels, up to two-party review. Publishing a package rather than an image follows the same shape: the registry's own trusted publishing signs from the workflow and records provenance. Provenance shows which commit a build came from. It does not show the commit was authorized: some ChainDrop releases went out through the projects' own workflows with valid provenance.

CycloneDX (ECMA-424) and SPDX (ISO/IEC 5962, which covers SPDX 2.2.1; 3.0 is still a draft at ISO) are the two main SBOM formats; CISA's 2026 Minimum Elements (July 2026) set what a complete one carries.

6 OPERATE: PATCH WHAT IS EXPLOITED FIRST

The SBOM store re-flags released builds as advisories are published; runtime detection (Falco) watches the workload. Sorting by severity alone leaves more findings than a team can work through, so order them:

- Fix first what is being exploited (CISA's Known Exploited Vulnerabilities catalog, KEV).
- Then what is likely next (EPSS, the Exploit Prediction Scoring System, estimates the chance each CVE is exploited in the next 30 days).
- Then the rest by severity and reachability.

When a vulnerability is present but cannot be exploited in your build, say so in a VEX statement (vulnerability exploitability exchange) so scanners stop re-opening it.

Publish a security.txt (RFC 9116) and a disclosure policy so reporters reach you. SSDF's RV practices cover what happens after a report arrives: confirm it, fix it, and find the root cause.

HARDEN THE PIPELINE ITSELF

CI runs code from many sources with credentials to your cloud, registry, and repos. The OWASP Top 10 CI/CD Security Risks and the CIS GitHub Benchmark v1.2.0 cover pipeline security in depth; start here:

no stored cloud keys jobs fetch short-lived cloud credentials over OIDC (OpenID Connect) federation, so no long-lived key exists to steal. Scope the cloud role to your own repository and ref, or any repository on the platform can assume it.

pin actions to a commit SHA an attacker re-pointed the tj-actions version tags at malicious code (CVE-2025-30066), and jobs using it dumped CI secrets into build logs.

least-privilege tokens set the job token's default to read-only, then grant write scopes per job; GitHub organizations created before 2 February 2023 still default to read/write.

protected branches review plus passing checks to merge; no force-push, no direct pushes to main.

ephemeral runners each job starts from a clean machine; never echo a secret, since masking catches only the exact value.

runtime secrets a secrets manager (OpenBao, your cloud's own) injects them at deploy; the repo and CI config hold references, never values.

THE FRAMEWORKS, AND WHAT EACH ONE IS FOR

FRAMEWORK	USE IT FOR
NIST SP 800-218 (SSDF) v1.1	SDLC practices: prepare, protect, produce, respond
OWASP ASVS 5.0	security requirements, three levels
OWASP Top 10:2025	top web app risks; supply chain enters at A03
OWASP Top 10 CI/CD Security Risks	the ten CICD-SEC pipeline risks
OWASP Cheat Sheet Series	secure coding how-tos, per topic
SLSA v1.2	build and source integrity levels
CIS Controls v8.1, Control 16	prioritized app security safeguards
CIS Benchmarks	hardening baselines: GitHub, Docker, Kubernetes
NIST SP 800-204D	supply chain security in CI/CD

FRAMEWORK	USE IT FOR
OWASP SAMM / DSOMM	maturity models: software assurance, DevSecOps
OWASP Top 10 for LLM Applications 2026	risks when the app embeds an LLM

ADOPT THE CHECKS IN THIS ORDER, HIGHEST RISK FIRST

- a turn on the platform's features: push protection (paid on private repos), dependency alerts and updates, branch protection
- b secrets scanning + SCA in CI, report-only; fix the criticals they surface
- c make both blocking for NEW findings; baseline the backlog and work through it on a schedule
- d add SAST, IaC, and image scanning; tune the rules until most findings are real
- e SBOM generation, signing, provenance in the release job; verify at deploy
- f DAST on staging; then measure: time to remediate by severity, share of builds with SBOM + provenance

Block builds only on new, confirmed findings, and schedule the backlog: the team switches off a gate that fails every build on findings nobody has reviewed.

REGULATORY DATES

The EU Cyber Resilience Act (CRA) covers products with digital elements made available on the EU market in the course of a commercial activity, paid or free; free and open-source software its maintainer does not monetize falls outside.

Article 14 puts the reporting duty on manufacturers from 2026-09-11: actively exploited vulnerabilities and severe incidents go to the designated CSIRT (incident response team) and ENISA.

The steward rule is a separate hook and can reach an organization even where the software itself falls outside: a foundation or company that gives sustained support to open-source built for commercial use is an open-source software steward (Article 24), monetized or not. A steward reports an actively exploited vulnerability where it helps develop the product, and a severe incident where that incident reaches the systems it provides for the development. Article 24 sits outside the articles the CRA applies early, so the steward's duty starts 2027-12-11, when the Regulation applies in full.

Whether the CRA reaches a particular product, and whether an organization is a steward, are legal questions; take advice before deciding either. In the US, OMB M-26-05 (January 2026) ended the government-wide attestation mandate; agencies may still ask for the attestation form or an SBOM where their own risk assessment calls for it.