



DevSecOps toolchain security checklist

A practical companion to our guide on DevSecOps toolchain security. Use it to check CI/CD controls, code and dependency security, cloud configuration, API testing, and runtime protection across your pipeline.

Work through each section with your security and engineering teams. Tick items when the control is actually enforced in your pipeline, not when a policy says it should be.



CI/CD Controls

- ☐ **Require MFA on every pipeline access point**

Confirm multi factor authentication is enforced for anyone with access to the CI/CD system, not just production. The pipeline is the highest value target in the toolchain, since it has privileged access to source code, credentials, deployment environments, and artifact registries.

- ☐ **Scope RBAC to what each role actually needs**

Confirm role based access control limits each user and service account to only the permissions their role requires, rather than granting broad pipeline access by default.

- ☐ **Manage secrets through a vault, not environment variables**

Confirm credentials used by the pipeline are stored in a dedicated secrets vault rather than pipeline environment variables, which are frequently logged, readable by anyone with pipeline access, and not rotated automatically.

- ☐ **Log every pipeline action for audit**

Confirm comprehensive audit logging captures every action taken in the pipeline, so activity can be reconstructed if something goes wrong.

- ☐ **Sign artifacts and verify provenance**

Confirm artifacts are signed and provenance is verified at each promotion step, so what reaches production is confirmed to be what was actually built, not something substituted in transit.

Code security

- ☐ **Integrate SAST into the CI/CD pipeline**

Confirm static analysis runs automatically in the pipeline, not just as an occasional manual scan, so insecure coding patterns are caught before they reach later, more expensive stages.

☐ **Enable SAST feedback inside the IDE and pull requests**

Confirm developers get static analysis findings directly inside the IDE and as pull request checks, so issues are caught at the point of writing code rather than after it has already merged.

☐ **Tune SAST rules to control false positives**

Confirm scanning rules are tuned for your codebase and findings are integrated into issue tracking, since a high false positive rate is what causes engineers to stop trusting and acting on scanner output.

☐ **Document and enforce secure coding standards**

Confirm secure coding standards exist and are actually referenced during code review, not just published and forgotten.

Dependency security

☐ **Run SCA against every build**

Confirm software composition analysis checks every dependency version against known vulnerabilities as part of the build process, and that builds can be blocked on high risk findings.

☐ **Generate and maintain an SBOM**

Confirm a software bill of materials is generated for each build and kept current, so the exact composition of what shipped is always known.

☐ **Automate dependency updates and ongoing monitoring**

Confirm dependency updates are automated where possible, and that vulnerability monitoring continues after deployment, not just at build time, so newly disclosed CVEs in dependencies already in production get caught.

Cloud security

- ☐ **Scan infrastructure as code before deployment**

Confirm Terraform, CloudFormation, Kubernetes manifests, and other templates are scanned for misconfigurations such as exposed ports, overly permissive IAM roles, and unencrypted resources before they are provisioned.

- ☐ **Enforce policy as code across environments**

Confirm infrastructure policy is enforced consistently through policy as code, so the same rules apply whether infrastructure is provisioned through GitOps or directly through the pipeline.

- ☐ **Monitor cloud posture and IAM continuously**

Confirm a CSPM tool monitors deployed environments for misconfiguration and policy drift after deployment, and that IAM roles are validated against least privilege on an ongoing basis, not just checked once at provisioning.

API security

- ☐ **Run DAST against deployed applications, not just source code**

Confirm dynamic testing runs against the application in staging or pre production, validating that vulnerabilities identified elsewhere are actually exploitable in a real environment.

- ☐ **Test authenticated flows, not just unauthenticated surfaces**

Confirm DAST scanning covers behavior behind login, including authorization checks and multi step workflows, since these are the flaw classes most commonly exploited in production

- ☐ **Cover REST, GraphQL, and SOAP API endpoints**

Confirm API scanning extends to every protocol your application uses, not just the ones with public documentation, and includes schema validation.

☐ **Map API testing coverage to the OWASP API Top 10**

Confirm API security testing is checked against the OWASP API Top 10, since APIs left untested are one of the gaps attackers most actively look for.

Runtime security

☐ **Enforce WAF protection in production**

Confirm a web application firewall is active in production to reduce the window of exploitability for known vulnerability classes while fixes are being developed and deployed.

☐ **Feed security logs into a SIEM**

Confirm security relevant logs, metrics, and traces are collected and fed into a SIEM where correlation rules can surface anomalies that individual data points would not show on their own.

☐ **Maintain incident response playbooks**

Confirm incident response playbooks exist and are current, so the team has a defined process to follow when monitoring surfaces rather than improvising during an active incident.

☐ **Track dwell time as a security metric**

Confirm the time between compromise and detection is tracked, since it is one of the strongest predictors of breach cost and shows whether observability investments are actually working.

Treat the toolchain like production

Pipelines carry the same access, credentials, and blast radius as the systems they deploy to, which is why they deserve equivalent controls. Building a secure toolchain is not a one time project: it means choosing tools that integrate rather than sprawl, building governance that produces actionable findings instead of alert volume, and treating retesting as a standard part of remediation rather than an optional follow up.