GitLab CI/CD Hardening Field Guide

Fast audit and hardening guidance for engineers taking over an existing GitLab deployment

Author: Ivan Piskunov | Release date: 01 April 2026 | Version 1.0

This guide is an independent technical compilation based on current GitLab documentation, field-tested review patterns, and the author's practical experience. It is written for engineers who need to assess and harden an existing GitLab environment under time pressure.

Built for real-world handoff scenarios

The document assumes a messy but normal starting point: inherited GitLab groups, mixed runner trust levels, token sprawl, half-documented production paths, and uneven security ownership. The goal is not abstract theory. The goal is to find blast radius, reduce it fast, and leave behind clear operating controls.

Disclaimer, scope, safe use, and reuse

This handbook is compiled from many sources, including official GitLab materials, practical engineering experience, operational review patterns, and the author's editorial judgment. It is not an official GitLab publication, not legal advice, and not a substitute for change-management discipline, staging, or vendor support.

Use the procedures in this guide carefully. Misapplied configuration changes can cause broken pipelines, deployment freezes, access loss, or production incidents. The author provides the material as-is and assumes no responsibility for direct or indirect damage caused by unsafe, incorrect, or out-of-context use.

This guide is for education, defensive operations, review work, and legitimate hardening only. It does not endorse unethical conduct, unauthorized access, or offensive activity outside an explicitly approved scope.

How to use this guide

- If you just joined a company, start with Sections 5, 6, and 7. They cover first-hour triage, the five-day audit, and the two-week hardening plan.
- If you are documenting current posture for leadership, Sections 4, 14, and 15 give you impact language, evidence expectations, and remediation phrasing.
- If you are actively configuring GitLab, Sections 8 through 13 provide step-by-step menu paths, what to click, what to enable, and what to verify afterward.
- If you are refreshing concepts before an interview or review panel, use the glossary and Q&A appendices.

> **Tier reality check**
> Some GitLab controls are tier- or offering-specific. Keep the control objective even when the exact feature is unavailable. If you do not have a built-in control, replace it with process, API automation, or compensating governance.

Table of contents

1. Executive summary

GitLab is not just source control. In most companies it also acts as a privileged code-execution system, a release gate, a token broker, an artifact transport, and, in many cases, a route into production infrastructure. That is why an existing GitLab deployment deserves the same scrutiny as any other high-value administrative platform.

When you inherit a messy GitLab setup, do not start with scanner cosmetics or dashboard cleanup. Start by mapping trust: who can change pipeline logic, who can approve or bypass change control, where jobs actually run, where secrets land, and who can push a deployment into production. Until those answers are clear, everything else is second-order work.

Priority	What to answer fast	Why it matters
P0	Who are the instance, group, and project admins?	This defines administrative blast radius and break-glass risk.
P0	Who can merge into default and release branches?	Weak merge governance turns repo access into pipeline control.
P0	Who can deploy to production environments?	Code control and deploy control are different attack surfaces.
P0	What runners execute sensitive jobs?	This is where token theft and host compromise usually happen.
P1	What token and secret model is in use?	Static, broad credentials create persistence and abuse windows.
P1	What audit evidence exists and where is it exported?	Without evidence, you cannot defend or investigate quickly.

Senior framing

Treat GitLab as a privileged change system. Your audit order should be identity, change control, execution surface, secrets, deployment governance, and finally telemetry.

2. GitLab CI/CD architecture

GitLab CI/CD is easiest to reason about if you split it into three zones. **The first is the human and administrative layer:** administrators, owners, maintainers, groups, projects, protected branch rules, merge approvals, variables, environments, and policy projects. **The second is the execution layer:** runners, containers, host operating systems, caches, package registries, and network egress. The third is the target and evidence layer: production systems, cloud APIs, cluster endpoints, deployment targets, artifacts, and audit streams.

In self-managed environments, also remember the core application components. GitLab reference architectures describe GitLab as a multi-component system that includes webservice or Rails nodes, Sidekiq, Gitaly, PostgreSQL, Redis, and object storage. From a hardening standpoint, this means you are protecting both application features and the platform that serves them.

2.1 Component roles that matter during a security review

Web/API and GitLab Shell/Workhorse are the human-facing control surface. This is where users authenticate, where maintainers change policy, and where pipeline definitions are submitted. If this layer is sloppy, attackers do not need an exotic exploit - they can often just *change the rules of the game* through normal features.

Gitaly and Praefect are the repository truth layer. They hold refs, commits, and repository operations. From a hardening perspective, they matter because a repo is not just source code; it is also CI definitions, IaC, release manifests, and the policy-bearing files that control what runners execute.

Sidekiq, PostgreSQL, Redis, object storage, registry, and packages are the state and evidence plane. Queues, approvals, artifacts, uploads, packages, and cached build outputs live here. If this layer is unavailable, corrupted, or weakly protected, your delivery process becomes both brittle and hard to investigate.

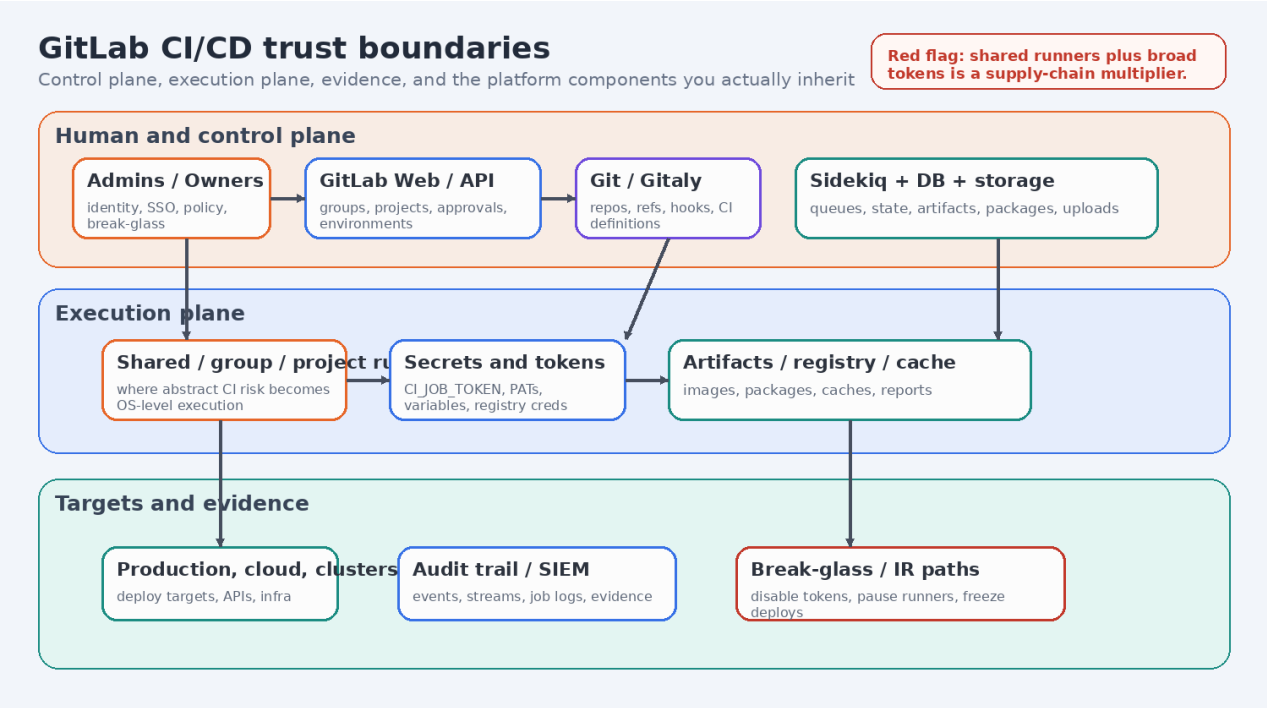
Runners are where abstract pipeline risk turns into real host, network, and credential risk. Treat every runner as a remote code execution service that developers indirectly control through repository content.

2.2 GitLab.com versus self-managed: what changes from a security angle

GitLab.com and GitLab Dedicated customers do not get the same instance-admin plane that self-managed administrators do. That means some instance-wide knobs, such as the full Admin area, instance variables, runner registration controls, and instance audit streaming, are only part of the customer story on self-managed and Dedicated estates. On GitLab.com, the customer-owned control plane starts mainly at the top-level group, project, and runner choices.

Self-managed gives you more control and more liability. You own patch timing, secrets-file protection, backup handling, instance settings, runner fleet design, and the blast radius of every integration. A mature self-managed setup can be tighter than SaaS, but a neglected one can also be far messier.

GitLab.com removes some platform plumbing, but not the big customer mistakes. Weak branch rules, weak MR approvals, broad deployment permissions, poorly scoped variables, unsafe self-managed runners, and overpowered maintainers are still very much your problem in SaaS.



Visual atlas 1 - GitLab trust boundaries and the main surfaces you inherit on day one

Component or plane	Security question	Typical failure mode
Web/API and groups/projects	Who can administer and change policy?	Too many admins, weak 2FA, stale owners, soft default settings.
Repo and merge control	Who can change pipeline logic or release-critical files?	Direct pushes, missing approvals, missing CODEOWNERS.
Runners	Where does code really run?	Shell executor, reused hosts, privileged Docker, mixed trust tiers.
Tokens and variables	How do jobs authenticate downstream?	PAT sprawl, unprotected variables, unclear ownership, no expiry.
Environments and deploy gates	Who can ship to prod?	Unprotected environments, missing deployment approvals.
Audit and evidence	Can you reconstruct what happened?	No streaming, weak retention, evidence stuck in UI only.

3. Trust boundaries and main attack surfaces

Most GitLab breaches do not start with a Hollywood-grade exploit. They start with permissive defaults, stale identities, weak branch rules, or an execution surface that was left too open because it was convenient. **The core trust boundaries are simple:** source control, pipeline execution, secret access, deployment authorization, and evidence generation.

Surface	What attackers want	Primary control
Source control	To land malicious CI logic, release config, or infra changes	Protected branches, MR approvals, CODEOWNERS.
Execution plane	To execute code on a runner and steal tokens or reach internal services	Runner isolation, ephemeral execution, egress control.
Registry and artifacts	To poison software outputs or steal packages and reports	Artifact access limits, provenance, access scoping.
Production deploy path	To ship an unauthorized change or bypass approvals	Protected environments, deployment approvals, manual gating.
Audit trail	To make the story hard to reconstruct	Audit event streaming, retained logs, ownership and alerts.

Practical rule

Never evaluate a control by itself. Evaluate the chain. Protected branches without real approvals, or approvals without protected environments, still leave a weak path into production.

4. Common compromise patterns and business impact

A useful review does not stop at technical findings. You must translate them into business impact that leadership and service owners understand. That means showing how a configuration mistake maps to outage risk, software supply-chain risk, credential exposure, lateral movement, fraud exposure, or an investigation blind spot.

4.1 Ten attack patterns that show up again and again in GitLab estates

1. Direct push to a protected path. If a release or default branch still allows direct pushes, an attacker or rushed insider can land malicious CI, IaC, or release logic without meaningful review. The business impact is simple: unauthorized change lands fast and looks like normal engineering work.

2. Merge approval rubber-stamping. Teams sometimes enable approvals but still let the author self-approve, let committers approve after changing the MR, or leave CODEOWNERS missing for .gitlab-ci.yml and infra files. That creates a paper gate, not a real one.

3. Compromised maintainer token or browser session. A leaked PAT, stale project access token, or hijacked maintainer session can be enough to alter pipelines, variables, branch rules, or deploy jobs. The technical finding quickly becomes a business problem because one identity now owns both change control and release motion.

4. Shared runner lateral movement. On a persistent shared runner, one hostile job can steal tokens, inspect leftover workspaces, or abuse local caches and Docker state. In practice this turns one weak repository into a launchpad against better protected projects.

5. Shell executor host takeover. GitLab explicitly warns that shell executors create high risk on the runner host and network. If shell is attached to mixed-trust workloads, a bad job can become a host compromise, not just a pipeline issue.

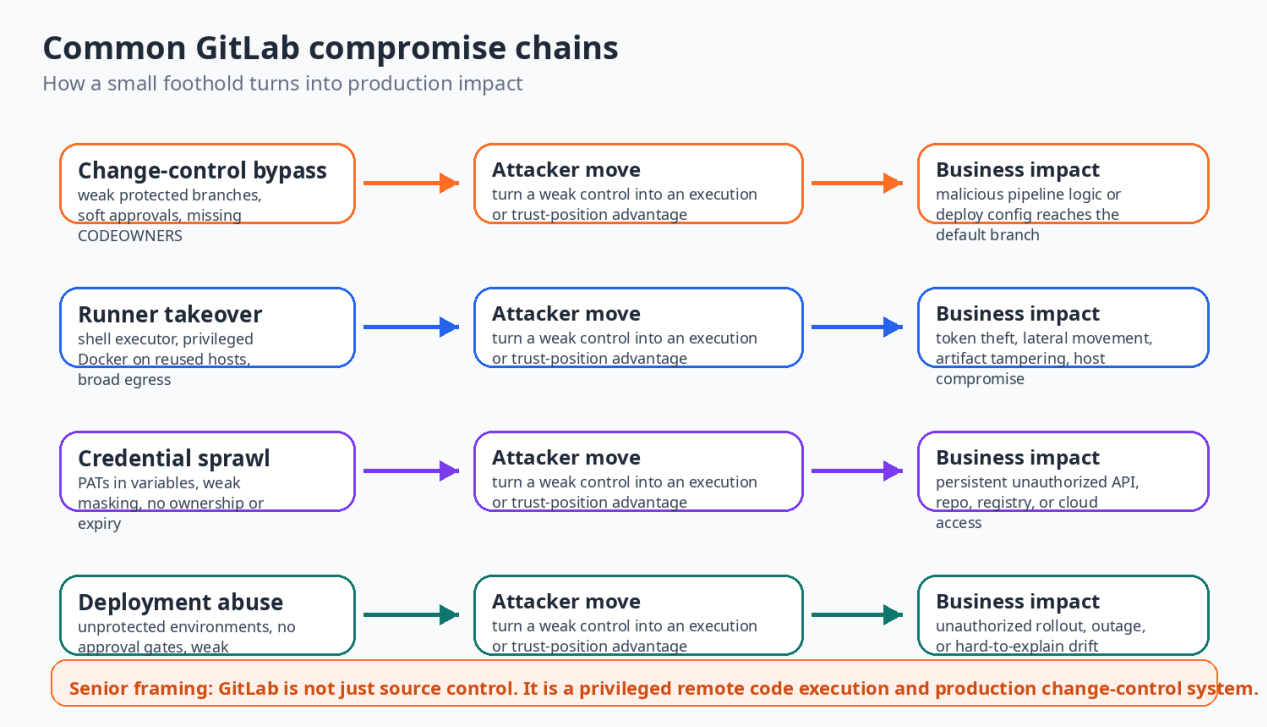
6. Variable and secret exfiltration. Secrets leak through echo statements, debug output, artifacts, service-container logs, or by simply running env/printenv in the wrong place. The point is not whether masking exists; the point is whether the job author can still coerce the platform into revealing or misusing the credential.

7. Cross-project job token abuse. A CI_JOB_TOKEN is ephemeral, but it is still powerful while the job runs. If allowlists are loose or public/internal project features are too open, one project can become the staging point for reaching another.

8. Dependency-chain poisoning inside the pipeline. Unsafe includes, unpinned base images, unverified packages, and blind trust in upstream artifacts let an attacker modify what the pipeline consumes rather than what the repo stores. This is one reason CI incidents often turn into supply-chain incidents.

9. Deployment gate bypass. If production is not a protected environment, or if the same broad maintainer population can both change the pipeline and approve the deploy, the supposed release gate is cosmetic. A code compromise becomes a production compromise with almost no friction.

10. No durable evidence path. When audit streams are absent, retention is weak, and teams do not know where deploy evidence or policy evidence lives, incident response slows down badly. The gap is not only detective; it also blocks accountability and regulator-facing evidence later.



Visual atlas 2 - common compromise chains in a GitLab-driven delivery system

Finding pattern	Likely exploit path	Business impact
Direct pushes allowed on release branches	Attacker or rushed insider changes CI or deploy logic without review	Unauthorized deployment, production instability, hard-to-explain hotfixes.
Shared persistent runner with broad network reach	Malicious job steals tokens or pivots to internal services	Credential theft, infra compromise, wider incident scope.
PATs stored as variables	Token lands in logs, artifacts, or reused runner workspace	Persistent repo/API access after the initial foothold.
Unprotected production environment	Anyone who can run the job can deploy	Unauthorized or rushed production change, audit friction.
No audit export or weak retention	Key state changes live only in the UI or disappear too soon	Slow incident response, poor forensics, weak accountability.

5. First 60 minutes in a new company

The first hour is not for perfection. It is for orientation. You are trying to answer five questions fast: who has power, what is protected, where code executes, where credentials live, and whether you can reconstruct decisions later.

5.1 Walk the first hour like an operator, not a tourist

Your first pass is not a line-by-line audit. It is a **trust map**. You are trying to find where one bad identity, one bad merge, or one bad runner can cascade into prod or into a broad credential leak. The menu paths below are the ones worth opening even before you start collecting screenshots.

5.2 Identity and admin map

Self-managed path: Admin -> Settings -> General -> Sign-in restrictions and the broader **Admin** area. Also enumerate top-level group owners and critical project owners. On GitLab.com, start from the top-level group because customers do not get the instance Admin area.

What can be wrong: no enforced 2FA, no Admin Mode, too many instance admins, break-glass accounts used for normal work, stale owners who kept power after role changes, and SSO configured in a way that still leaves risky password fallback open without a documented emergency path.

How to interpret it: if one human session can both change security policy and push urgent repo changes, your control plane is too concentrated. *If the same people are admins everywhere, your blast radius is already telling you the answer.*

Fast cleanup: freeze new admin grants, name the real break-glass identities, enable **Enforce two-factor authentication**, enable **Enforce two-factor authentication for administrators**, and switch on **Enable Admin Mode** where that control exists.

5.3 Branch rules, merge approvals, and CODEOWNERS

Project paths: Settings -> Repository -> Branch rules, **Settings -> Merge requests -> Merge request approvals**, and the repository tree for **CODEOWNERS**.

What can be wrong: default or release branches still allow pushes, **Allowed to push and merge** is not set to an explicit deny like **No one**, required approvals are too low, authors can approve their own changes, committers can still approve after modifying the MR, or there is no CODEOWNERS coverage for .gitlab-ci.yml, deploy manifests, helm charts, Terraform, or release automation.

How to interpret it: weak repo governance means a repo write can become a runner execution and deployment event. In GitLab terms, that is often the shortest path from a repo compromise to an environment compromise.

Fast cleanup: protect the default and release branches, set **Allowed to push and merge = No one** where appropriate, require approvals, turn on **Prevent approval by merge request creator**, use **Prevent**

approvals by users who add commits when available, and make CODEOWNERS mandatory on the files that change pipeline behavior.

5.4 Protected environments and deploy approvals

Project paths: Settings -> CI/CD -> Protected environments and **Operate -> Environments** for a reality check on what is actually deployed.

What can be wrong: production is not protected at all; the environment exists but has no narrow deploy list; **Allowed to deploy** still points at a wide role like all Developers for a crown-jewel target; deployment approvals are missing; or the people who author the pipeline are also the only approvers.

How to interpret it: if branch protection is decent but prod is still open, the attacker simply shifts from source-control bypass to deployment-path abuse. This is why branch controls are necessary but never sufficient.

Fast cleanup: create or edit the **production** protected environment, keep **Allowed to deploy** narrow, add deployment approvals, and separate deployers from approvers when possible. Then confirm in **Operate -> Environments** that the gate behaves the way the policy says it should.

5.5 Runners, executors, and trust tiers

Paths: Admin -> CI/CD -> Runners for instance runners, **Build -> Runners** at group scope, and **Project -> Settings -> CI/CD -> Runners** inside critical projects.

What can be wrong: shell executors attached to mixed-trust workloads, privileged Docker used on reusable hosts, runners that accept untagged jobs, instance runners exposed to sensitive projects by default, no protected runners for protected branches, weak maintenance notes, and old authentication tokens that were never rotated.

How to interpret it: the runner list is not a convenience inventory - it is your host compromise shortlist. The GitLab docs are blunt here: shell executors are high risk, and non-ephemeral shared runners multiply token theft and lateral movement risk.

Fast cleanup: pause obviously dangerous runners, tag runners by trust tier, mark sensitive runners as **Protected**, stop broad untagged execution, and review who can still create runners. In instance settings, clear legacy runner-registration options you do not need.

5.6 Variables, job tokens, and registry access

Paths: Project -> Settings -> CI/CD -> Variables, **Project -> Settings -> CI/CD -> Job token permissions**, and for self-managed estates also **Admin -> Settings -> CI/CD -> Job token permissions**.

What can be wrong: secrets stored as plain visible variables, variables not **Masked** or **Masked and hidden**, no **Protect variable** on production-bound credentials, broad environment scope, PATs used where a **CI_JOB_TOKEN** or project token would do, and job-token allowlists disabled or widened without a compelling cross-project need.

How to interpret it: variable misuse usually means one compromised job can spill far more credential power than the pipeline really needed. Remember that GitLab itself says variables are convenient but less secure than a dedicated secrets manager.

Fast cleanup: move crown-jewel secrets to an external secrets manager if you have one; otherwise set sensitive variables to **Masked and hidden**, use **Protect variable**, tighten environment scope, and re-enable a tight job-token allowlist. On self-managed, consider instance enforcement of job-token allowlists.

5.7 Audit events, streams, and whether you can investigate later

Paths: Admin -> Monitoring -> Audit events -> Streams for instance scope on self-managed, and **Group -> Secure -> Audit events -> Streams** for top-level group streaming.

What can be wrong: no streaming destination at all, a destination that exists but is inactive, no TLS confidence in the endpoint, no one validating the **verification token**, no event-type filters where they would help, or a stream that exists on paper but is not actually monitored by a SIEM or alerting path.

How to interpret it: an audit page with nothing exported is a future incident-response tax. Also remember that GitLab documents that streamed events can arrive more than once, so the downstream pipeline must deduplicate by event id instead of treating duplicates as noise or as separate incidents.

Fast cleanup: add or repair the streaming destination, confirm the endpoint is trusted and encrypted, expand the stream entry and verify the token/header setup, then test whether rule changes, owner changes, and deploy-control changes show up where responders can actually query them.

Time slice	What to click and inspect	What to record
0-10 min	Admin or group owner views; list top-level groups and critical projects; identify admins and owners	Admin map, owner map, sensitive projects.
10-20 min	Project > Settings > Repository > Branch rules; Project > Settings > Merge requests	Default branch posture, approvals, CODEOWNERS coverage, direct-push gaps.
20-30 min	Project > Settings > CI/CD > Variables, Job token permissions, Protected environments	Token model, variable exposure, deploy controls, prod gates.
30-45 min	Runner inventory in Admin area or project/group runner settings; inspect executor types and tags	Runner trust tiers, shell usage, privileged Docker, reuse patterns.
45-60 min	Admin > Monitoring > Audit events > Streams or group audit configuration	Evidence path, retention, export gaps, IR blockers.

- Do not leave the first hour without an admin and owner list.
- Do not leave the first hour without knowing whether production is a protected environment.
- Do not leave the first hour without knowing whether any shell executors are attached to shared or sensitive workloads.

6. Five-day audit plan

A five-day audit should produce a high-confidence risk picture, not a pile of screenshots. Each day needs a focused question, a deliverable, and a list of owners who can validate or challenge your assumptions.

6.1 What each audit day should look like in practice

Use the five-day plan as a **disciplined operator schedule**, not as a box-ticking exercise. Each day below assumes you are building evidence, validating with owners, and ending with a short written delta: what you confirmed, what remains unproven, what needs fast containment.

Day 1 - identity, admin access, and scope map

Start by enumerating the instance, top-level groups, critical projects, and business-critical delivery paths. In self-managed, go to **Admin**, then review **Settings -> General -> Sign-in restrictions** and the relevant ownership lists. In group and project scope, record owners, maintainers, service owners, SSO assumptions, and any named break-glass path.

By the end of Day 1 you should know **who can change policy**, **who can approve or bypass change control**, and **which repos or environments are crown-jewel paths**. If you cannot answer those three questions, stay on identity work instead of rushing into scanners.

Day 2 - source control and deploy gates

Open critical repos and walk **Settings -> Repository -> Branch rules**, **Settings -> Merge requests -> Merge request approvals**, and **Settings -> CI/CD -> Protected environments**. Then cross-check real deployment history under **Operate -> Environments** so you do not mistake a nice setting page for an actually used gate.

Document bypass paths plainly: direct push, low or cosmetic approvals, missing CODEOWNERS on CI or infra files, prod deploy rights too broad, or no approval layer on protected environments. End Day 2 with a concise **change-control findings list** and a sketch of the shortest path from repo write to prod.

Day 3 - runners, network reach, and execution isolation

Go through **Admin -> CI/CD -> Runners** or the group/project runner pages. Identify which runners are instance, group, or project scoped; which accept untagged jobs; which are protected; which use shell or privileged Docker; and which are static versus ephemeral.

Talk to the platform owner the same day. A runner review without a network review is incomplete. Ask what those runner hosts can reach: cloud metadata services, kube APIs, package registries, artifact stores, internal GitLab services, and production control planes. Your Day 3 deliverable is a **runner trust-tier map** with obvious containment actions.

Day 4 - secrets, tokens, and policy enforcement

Review **Settings -> CI/CD -> Variables** at project, group, and where relevant instance scope. Check visibility, protection, environment scoping, and whether teams are storing long-lived human tokens for automation. Then inspect **Job token permissions** and any cross-project patterns that depend on allowlists.

If the organization uses scan execution or pipeline execution policies, confirm actual runtime evidence rather than trusting that a policy YAML file exists. Day 4 should end with a **credential reduction plan** and a **policy reality check**: what is enforced, what is aspirational, and what is bypassable today.

Day 5 - evidence chain, reporting, and priority decisions

Use the last day to verify your evidence path: audit events, audit streams, runner logs, deploy history, and token ownership records. Open **Admin -> Monitoring -> Audit events -> Streams** or **Secure -> Audit events -> Streams** at top-level group scope, confirm the destination is active, and test whether important changes are actually queryable downstream.

Finish with three outputs: **a short leadership summary**, **an engineering finding tracker**, and **a two-week hardening queue**. The best Day 5 readout is blunt: here is where one bad MR or one bad runner becomes an outage, a prod compromise, or a secrets spill.

Day	Focus	Detailed steps	Deliverable
1	Scope and identity	Map instances, top-level groups, critical projects, admins, owners, SSO posture, 2FA posture, break-glass accounts, and naming conventions.	Context map and privileged access summary.
2	Source and merge control	Review branch rules, merge request approvals, committer/self-approval settings, CODEOWNERS, release process, and emergency paths.	Change-control findings and bypass map.
3	Execution and secrets	Review runners, executor choices, reuse, privilege, egress, variables, job token permissions, PATs, project access tokens, deploy tokens, and secret-manager coverage.	Runner and credential risk register.
4	Deployment governance and evidence	Review protected environments, deployment approvals, manual jobs, artifact access, audit events, and log exports.	Production-protection and evidence summary.
5	Business impact and remediation sequencing	Prioritize findings by blast radius, ease of abuse, and service criticality; assign owners and dates.	Executive summary plus hardening roadmap.

What a good five-day deliverable looks like

Top findings with owners, a trust-boundary map, confirmed production-protection posture, token and runner hygiene gaps, and a short list of 48-hour quick wins versus structural 2-to-6-week fixes.

7. Two-week hardening sprint

The first two weeks should focus on closing the easiest abuse paths. The sequence below starts with identity, source and deploy control, runners, and tokens before it moves into policy tuning and reporting.

7.1 Turn the sprint table into a real operator runbook

The sprint is about **closing the shortest abuse paths first**. That usually means identity, deploy control, runner isolation, and token blast radius before you spend energy on fine-grained polish. The steps below assume you already completed the first-hour triage and have owners available to validate changes.

Days 1-2 - identity and break-glass discipline

In self-managed, go to **Admin -> Settings -> General -> Sign-in restrictions**. Enable **Enforce two-factor authentication**, enable **Enforce two-factor authentication for administrators**, and set the grace period only if the organization truly needs a short landing window. Then enable **Enable Admin Mode** so administrators are not always operating in a permanently elevated session.

At the same time, shrink the human blast radius. Remove stale owners, document which identities are real break-glass accounts, and make emergency access noisy and reviewable. **Do not keep "temporary" owners around for later.** In GitLab they have a habit of becoming permanent power centers.

Days 3-4 - branch and deploy gates

For the most sensitive repositories, open **Settings -> Repository -> Branch rules** and make the push posture explicit. On high-value branches, that often means **Allowed to push and merge = No one** with changes flowing only through MRs. Then open **Settings -> Merge requests -> Merge request approvals** and require real approvals, not ceremonial ones.

Next, open **Settings -> CI/CD -> Protected environments** and protect production and other high-value targets. Keep **Allowed to deploy** narrow, add deployment approvals, and verify in **Operate -> Environments** that the deploy path now behaves like a gated release lane rather than a convenience shortcut.

Days 5-7 - runner isolation by trust tier

Use **Admin -> CI/CD -> Runners** to separate runners by purpose and trust. Stop using shell executors for mixed-trust work. Where privileged Docker is unavoidable, keep it on isolated and preferably ephemeral hosts. Add runner tags that reflect trust boundaries rather than team nicknames, and mark sensitive runners as **Protected** so they only serve protected refs.

Review who can still register runners. In **Admin -> Settings -> CI/CD -> Runners** clear legacy registration options you do not need, prefer the newer runner creation workflow, and rotate runner authentication tokens on a schedule. The sprint goal is not "clean runner naming"; it is *making one bad job much less able to damage the next host or project*.

Days 8-10 - secret hygiene and token reduction

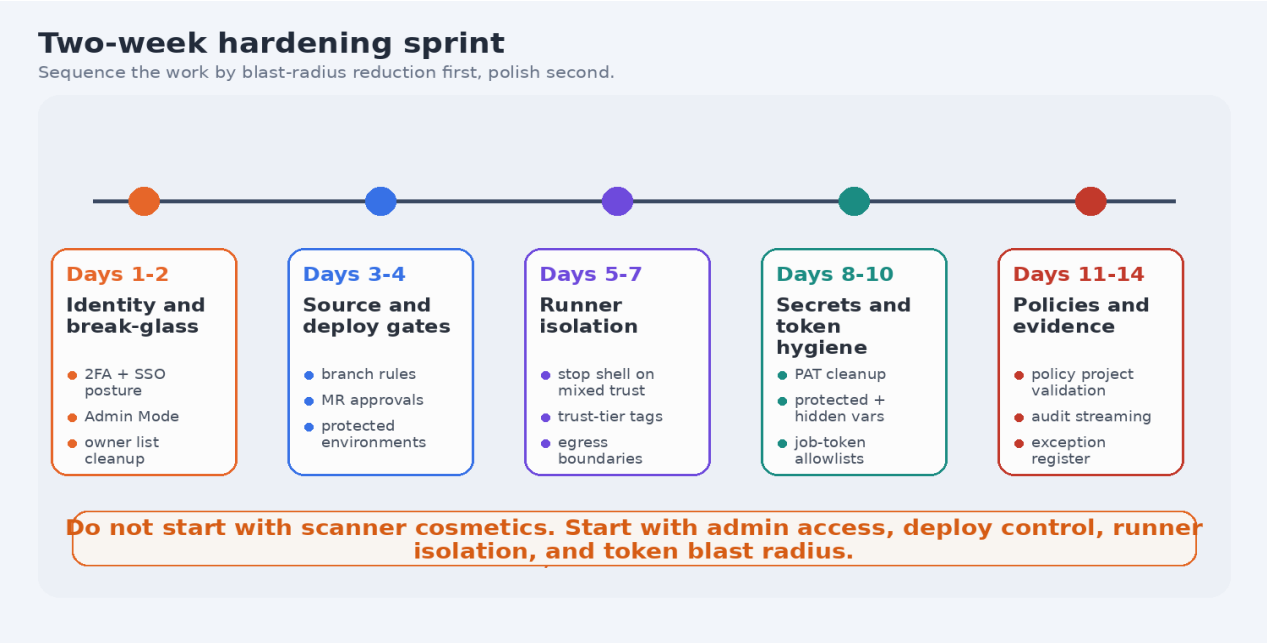
Walk project, group, and instance variables. Turn sensitive values into **Masked and hidden** where possible, set **Protect variable**, and reduce environment scope. Remove human PATs from normal automation if a job token, project token, deploy token, or group token can solve the same problem with less standing privilege.

Then tighten **Job token permissions**. Keep the allowlist small, use cross-project access only when there is a documented need, and consider instance-wide enforcement on self-managed estates. This part of the sprint pays off quickly because token cleanup often reduces both stealth and persistence for an attacker.

Days 11-14 - policies, logging, and the exception register

Use the last block to validate that critical scanners or pipeline-enforced jobs really run, and that the evidence survives after the pipeline finishes. Review the policy project, confirm anti-bypass behavior, and make sure exceptions are named, time-bound, and owner-assigned instead of living as tribal knowledge.

Finally, go back to **Audit events -> Streams** and confirm that your new control changes are visible where responders can query them. A hardening sprint without an exception register and an evidence path usually degrades back into ad hoc tuning within a quarter.



Visual atlas 4 - a realistic two-week hardening sprint for an existing GitLab deployment

Sprint block	Priority actions	Expected outcome
Days 1-2	Enforce 2FA, review SSO posture, enable Admin Mode where relevant, identify break-glass accounts, revoke obviously stale admin paths.	Administrative blast radius is smaller and explicit.
Days 3-4	Fix branch rules, require merge approvals, wire CODEOWNERS to sensitive paths, protect production environments.	Default path to main and prod now runs through real control points.
Days 5-7	Separate runner trust tiers, retire shell for untrusted work, reduce privileged reuse, constrain egress where possible.	Pipeline execution surface is materially harder to abuse.
Days 8-10	Clean up PATs, protect or hide variables, review job token permissions, move crown-jewel secrets to a dedicated secrets manager.	Credential posture moves from ad hoc to intentional.
Days 11-14	Enable policy enforcement where available, validate scanner execution evidence, turn on audit event streaming, document exceptions.	Controls are auditable and survivable after the initial cleanup.

8. Identity, admin access, and sign-in controls

This section is written as a hands-on playbook. Exact menu names can vary slightly by offering and tier, but the control objectives stay the same.

8.1 Sign-in restrictions and two-factor authentication

Menu path for self-managed instances: Admin > Settings > General > Sign-in restrictions.

- Enable Enforce two-factor authentication.
- Enable Enforce two-factor authentication for administrators.
- If the organization relies on SSO, review Allow password and passkey authentication for the web interface. Disable alternate web sign-in only when you understand the break-glass implications.

What to verify afterward: sign-in policy applies to expected users, emergency access is documented, and administrators are not silently exempt.

8.2 Admin Mode

Menu path: Admin > Settings > General > Sign-in restrictions > Enable Admin Mode.

- Enable Admin Mode so administrator sessions are not permanently privileged.
- Document which admin operations truly require elevated mode.
- Important caveat: Admin Mode does not protect Git client access, and Admin Mode sessions time out after six hours.

8.3 Ownership and break-glass hygiene

- Build a current list of instance admins, top-level group owners, critical project owners, and service owners.
- Keep break-glass accounts minimal, named, documented, and monitored.
- Kill stale owners before debating lower-value hygiene items.

> **What good looks like**
> Every privileged path has a named owner, 2FA is enforced, SSO posture is intentional, and emergency accounts exist only for real emergency use.

9. Source control and merge governance

Source control is where most GitLab security stories begin. If a user can land malicious or unsafe CI logic, infra-as-code, or release configuration on the wrong branch, the rest of the platform often follows.

9.1 Protect the default and release branches

Menu path: Project > Settings > Repository > Branch rules.

- Create or review a branch rule for the default branch and any release branches.
- Set Allowed to merge appropriately, usually Maintainers only for high-sensitivity branches.
- Set Allowed to push and merge to No one, not blank. An unconfigured value is not the same as an explicit deny.
- Enable Require approval from code owners on sensitive branches.

9.2 Merge request approvals

Menu path: Project > Settings > Merge requests > Merge request approvals.

- Set required approvals for high-sensitivity repositories.
- Enable Prevent approval by merge request creator.
- Enable Prevent approvals by users who add commits when possible.
- Reset approvals on push for change-sensitive repositories unless you have a justified exception.

9.3 CODEOWNERS for high-risk paths

Use CODEOWNERS for .gitlab-ci.yml, deployment manifests, infra directories, release workflows, secrets-handling code, and anything that materially changes blast radius.

Control	Menu or file	What to set
Branch protection	Settings > Repository > Branch rules	Explicit deny for direct push on protected release paths.
Approvals	Settings > Merge requests	Minimum approvals plus anti-self-approval controls.
Path ownership	CODEOWNERS file	Sensitive paths tied to named human owners or groups.

10. Deployment governance and production protection

Protected branches govern who can change code. Protected environments govern who can actually deploy. Well-run teams use both because source control and production control are separate attack surfaces.

10.1 Protect production environments

Menu path: Project > Settings > CI/CD > Protected environments.

- Create a protected environment entry for production and any equivalent high-risk environment.
- Restrict Allowed to deploy to the smallest practical group or role set.
- Use deployment-only access patterns where engineers need deploy rights but do not need full code ownership.

10.2 Require deployment approvals

Menu path: Project > Settings > CI/CD > Protected environments > Approval options or environment approval rules.

- Require deployment approvals for production-grade environments.
- Decide whether the pipeline triggerer can self-approve. In most environments, leave self-approval off.
- Remember that approval does not automatically execute the job. The job must still be run manually.

Check	Why it matters	Bad sign
Protected environment exists	Separates deploy authority from source authority.	Anyone who can run the job can deploy.
Approvals required	Creates a second control point before production.	Prod deploy is one click by the same actor who built the change.
Approval history visible	Supports investigations and compliance evidence.	No one can later explain who approved what.

11. Runners: where pipeline risk becomes host risk

If there is one GitLab topic to over-index on, it is runners. GitLab itself warns about high security risk with the shell executor and about reused privileged environments. The runner is where repository-controlled logic becomes real code execution.

11.1 Inventory runner types and trust tiers

Where to click: inspect the Admin area runner inventory if you have instance access, plus group- and project-level runner settings for critical repos.

- Separate shared, group, and project runners by trust tier.
- Mark which runners touch production credentials, signing keys, or deployment networks.
- Identify executor type: shell, Docker, Kubernetes, or custom.

11.2 High-risk posture to eliminate first

- Shell executor for untrusted or mixed-trust workloads.
- Privileged Docker on reused hosts when jobs from different trust domains share the same machine.
- Broad outbound network reach from build runners into internal APIs, databases, or cluster control planes.
- Persistent workspaces or caches that allow cross-job residue.

Red flag	Why it is dangerous	Immediate move
Shell executor	Jobs run with host-level user permissions and can threaten other projects on the same server.	Limit to trusted, tightly controlled workloads or retire it.
Reused privileged Docker	Breaks isolation and increases token or host-theft risk.	Prefer non-privileged execution and ephemeral workers.
Mixed trust on same runner fleet	Low-trust code can reach secrets intended for high-trust jobs.	Split runners by project sensitivity and tags.
Open egress everywhere	A compromised job can pivot to internal targets.	Use network boundaries and private endpoints intentionally.

12. Secrets, variables, tokens, and registry access

GitLab provides many ways to authenticate jobs and automation. That flexibility is useful, but it also creates sprawl if no one owns the decision tree. Your goal is to prefer short-lived, narrow-scope, non-human credentials wherever possible.

12.1 Variables

Menu path: Project, Group, or Admin > Settings > CI/CD > Variables.

- For new variables, prefer protected scope for anything that should never appear in non-protected branch pipelines.
- Use Masked or Masked and hidden where appropriate, but remember masking is not a magic shield against bad pipeline design or service-container logs.
- For crown-jewel secrets, prefer a dedicated secrets manager over plain CI/CD variables.

12.2 Job tokens and access tokens

Menu path: Project > Settings > CI/CD > Job token permissions.

- Keep the CI job token allowlist tight. Do not disable it casually.
- Avoid using human PATs for normal automation if a CI_JOB_TOKEN, project access token, deploy token, or group token can do the job with less risk.
- Review expirations, owners, scopes, and business purpose. Kill orphaned tokens quickly.

Credential type	Best use case	Main risk
CI_JOB_TOKEN	Job-scoped access during pipeline runtime	Leaked token is still dangerous while the job runs; runner hygiene matters.
Project access token	Automation tied to one project	Can become a long-lived backdoor if no owner or expiry exists.
Deploy token	Narrow package, container, or repo access	Still needs inventory, expiry, and scope discipline.
Personal access token	Last resort for human API usage or legacy automation	Inherits user power and becomes the first place to look for bad hygiene.

Simple rule

If the credential still works after the person leaves, or if no one can explain why the credential exists, assume it is technical debt until proven otherwise.

13. Common findings and remediation language

The table below is written in the language that works well in real remediation trackers: finding, why it matters, fast containment, and structural fix.

Finding	Why it matters	48-hour action	Structural fix
Shell executor used by mixed-trust projects	One malicious job can threaten the host and neighboring workloads.	Freeze new use and isolate the affected runner immediately.	Replace with ephemeral or stricter isolated runner tiers.
Direct pushes still possible on release branch	Review can be bypassed on a path that changes production behavior.	Set Allowed to push and merge to No one.	Standardize branch-rule baselines at project or group level.
PAT sprawl with weak ownership	Long-lived credentials create persistence and vague accountability.	Inventory and revoke orphaned tokens.	Move automation to non-human tokens with expiry and ownership.
Production is not a protected environment	Anyone who can run the job can deploy.	Protect the environment and restrict deployers.	Add approval rules and documented emergency path.
Audit events not exported	Security-relevant state changes are hard to investigate later.	Turn on streaming to a trusted destination.	Add retention, alerting, and evidence review ownership.

14. Snippets library

The snippets below are not drop-in universal answers. They are starting points you can adapt for your own platform standards and tier constraints.

Snippet: secure deployment skeleton

```yaml
stages:
  - verify
  - build
  - deploy

workflow:
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH

include:
  - component: $CI_SERVER_FQDN/security-group/pipeline-components/secret-detection@1.2.0

verify_pipeline_logic:
  stage: verify
  image: alpine:3.20
  script:
    - test -f .gitlab-ci.yml

build_app:
  stage: build
  image: registry.example.com/platform/build:2026-03
  script:
    - make build

deploy_prod:
  stage: deploy
  environment:
    name: production
    action: start
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
  script:
    - ./deploy.sh
  when: manual
```

Snippet - secure deployment skeleton with production environment separation

Snippet: runner config.toml posture

```
concurrent = 4
check_interval = 0

[[runners]]
  name = "prod-isolated-docker"
  url = "https://gitlab.example.com/"
  executor = "docker"
  [runners.docker]
    image = "alpine:3.20"
    privileged = false
    disable_cache = true
    pull_policy = ["if-not-present"]
    shm_size = 0
```

Snippet - runner posture emphasizing non-privileged Docker and reduced reuse risk

Snippet: CODEOWNERS for sensitive paths

```
# CODEOWNERS
/.gitlab-ci.yml              @platform-security @release-engineering
/infra/**                    @platform-security @sre-team
/charts/production/**        @sre-team @payments-owners
/app/config/**               @appsec @service-owners
```

Snippet - CODEOWNERS for CI, infra, and production-bound paths

Snippet: pipeline execution policy

```yaml
pipeline_execution_policy:
  - name: enforce-baseline-security-jobs
    description: Ensure critical checks always run
    enabled: true
    pipeline_config_strategy: inject_policy
    content:
      include:
        - project: platform/security-policy-project
          file: .gitlab/pipeline-execution/baseline.yml
    skip_ci:
      allowed: false
    variables_override:
      allowed: false
```

Snippet - baseline pipeline execution policy with anti-bypass intent

15. Training lab: CI/CD Goat and the OWASP Top 10 CI/CD risks

For hands-on practice, a useful training environment is CI/CD Goat, a deliberately vulnerable CI/CD lab from the official repository maintained by cider-security-research. It is effectively a WebGoat-style training lab for CI/CD security, with multiple challenges that let engineers practice against a realistic delivery environment.

The project describes eleven challenges and includes GitLab plus a GitLab Runner in the lab topology, alongside other CI/CD components. That makes it valuable as a safe learning appendix for engineers who want to understand how misconfigurations become attack paths in practice.

15.1 Reading the OWASP Top 10 through a GitLab lens

The OWASP list is useful because it describes recurring CI/CD risk classes, not just product-specific bugs. CI/CD Goat is helpful for the same reason: it lets engineers practice on a safe, intentionally broken environment. *It is not a perfect clone of a production GitLab estate, but the mental models transfer well.*

CICD-SEC-1: Insufficient Flow Control Mechanisms In GitLab this usually shows up as weak protected branches, low-value MR approvals, missing CODEOWNERS on CI and infra paths, or no deployment approvals on production. It is still common because teams optimize for speed first and only later discover they left no hard gate between a repository write and a release event.

CICD-SEC-2: Inadequate Identity and Access Management Think stale owners, too many maintainers, weak SSO hygiene, no enforced 2FA, and emergency accounts that quietly became daily-driver accounts. In GitLab reviews this risk is everywhere because access often accumulates faster than it is revoked.

CICD-SEC-3: Dependency Chain Abuse In GitLab pipelines this includes blind trust in upstream includes, unpinned base images, unsafe package pulls, and third-party build logic. It maps cleanly to the reality that many GitLab jobs assemble software from far more than the local repository.

CICD-SEC-4: Poisoned Pipeline Execution (PPE) This is the classic “the pipeline ran exactly what the attacker wanted” problem. In GitLab, poisoned execution often begins with a malicious .gitlab-ci.yml change, a dangerous include, or abuse of a runner that already has the right network and credential reach.

CICD-SEC-5: Insufficient PBAC (Pipeline-Based Access Controls) GitLab makes this visible in who can trigger, approve, or deploy pipelines and environments. When the same role can author the pipeline, run it, approve it, and deploy it, PBAC is usually too weak no matter how pretty the project dashboard looks.

CICD-SEC-6: Insufficient Credential Hygiene This maps directly to CI/CD variables, job tokens, PATs, deploy tokens, registry credentials, kube credentials, and cloud keys. It remains one of the most

frequent findings because secrets are convenient to add and socially hard to remove once pipelines depend on them.

CICD-SEC-7: Insecure System Configuration For GitLab this covers runner executor choice, privileged mode, reused hosts, broad instance runners, weak runner-registration controls, open deploy permissions, and sloppy feature visibility on public or internal projects. This category is common precisely because it often starts life as a convenience choice.

CICD-SEC-8: Ungoverned Usage of Third-Party Services GitLab estates often rely on external registries, secret stores, SaaS scanners, cloud deploy APIs, chatops bots, and package mirrors. If ownership, token scope, and trust assumptions are not documented, those integrations quietly become supply-chain extensions of your CI system.

CICD-SEC-9: Improper Artifact Integrity Validation Artifacts, images, packages, and caches are not harmless by-products; they are part of the delivery chain. In GitLab, weak artifact trust shows up when teams promote artifacts or images without enough provenance checking, signing, or environment separation.

CICD-SEC-10: Insufficient Logging and Visibility If owner changes, branch-rule changes, token creation, deploy approvals, and pipeline policy actions are not visible in durable logs or streams, responders are forced to reconstruct the incident from fragments. In practice this is still common because logging is often treated as a compliance afterthought instead of a runtime safety control.

Practical use: use CI/CD Goat to build detection muscle and attacker empathy, then map each lesson back into your own GitLab controls: branch rules, approvals, protected environments, runner isolation, token hygiene, artifact trust, and durable evidence.

OWASP CI/CD risk	Why it matters in GitLab	What to check in your environment
Insufficient Flow Control Mechanisms	Weak gating lets risky changes flow too easily.	Branch rules, approvals, protected environments, manual prod jobs.
Inadequate Identity and Access Management	Too much power sits with the wrong roles or stale accounts.	Admins, owners, SSO, 2FA, token owners, break-glass.
Dependency Chain Abuse	Build inputs can be poisoned or drift silently.	Registry trust, package controls, provenance, pinned inputs.
Poisoned Pipeline Execution	Malicious pipeline logic or runner compromise turns CI into an attack vehicle.	Runner isolation, pipeline review, executor posture.
Insufficient PBAC	Pipeline-based access controls are too soft or absent.	Protected variables, job token permissions, environment scoping.
Insufficient Credential Hygiene	Long-lived credentials persist after the initial foothold.	PAT cleanup, hidden variables, expiry, secret-manager coverage.
Insecure System	Default or inconsistent settings	Sign-in restrictions, Admin Mode, branch

Configuration	create easy wins for attackers.	baselines, artifact access.
Ungoverned Usage of 3rd Party Services	External actions and registries widen trust without clear ownership.	External includes, registries, cloud roles, SaaS integrations.
Improper Artifact Integrity Validation	You build one thing and ship another.	Artifact provenance, protected tags, controlled registry paths.
Insufficient Logging and Visibility	You cannot explain what changed quickly enough.	Audit event streaming, log retention, policy-source evidence.

16. Appendix A: GitLab glossary and field slang

Term	Plain meaning	How operators say it
Blast radius	How much damage one bad action can cause.	"I want to shrink the runner blast radius first."
Break-glass	Emergency access path.	"Document break-glass, but make it noisy."
Footgun	A feature that is easy to misuse and hurt yourself with.	"Unprotected prod is a footgun."
Snowflake runner	A special runner that no one understands well enough.	"That runner is a snowflake and it scares me."
Toxic default	A default that is convenient but too weak for serious use.	"Blank push permissions are a toxic default in the wrong repo."
Policy project	A project that centrally stores security policies.	"Treat the policy project like production control logic."
Prod gate	The control point right before deployment to production.	"The prod gate is missing a second approver."
Token sprawl	Too many credentials with unclear purpose or owner.	"We have obvious PAT sprawl."

17. Appendix C: Official references

Use these links as the official baseline for menu paths, feature behavior, and vendor-specific nuance. This guide translates them into a practical review and hardening workflow, but you should still verify your exact version and tier before making production changes.

- GitLab sign-in restrictions and Admin Mode - https://docs.gitlab.com/administration/settings/sign_in_restrictions/
- GitLab reference architectures - https://docs.gitlab.com/administration/reference_architectures/
- GitLab protected branches - https://docs.gitlab.com/user/project/repository/branches/protected/
- GitLab CODEOWNERS - https://docs.gitlab.com/user/project/codeowners/
- GitLab merge request approvals - https://docs.gitlab.com/user/project/merge_requests/approvals/
- GitLab protected environments - https://docs.gitlab.com/ee/ci/environments/protected_environments.html
- GitLab deployment approvals - https://docs.gitlab.com/ci/environments/deployment_approvals/
- GitLab security for self-managed runners - https://docs.gitlab.com/runner/security/
- GitLab shell executor security - https://docs.gitlab.com/runner/executors/shell/
- GitLab pipeline security - https://docs.gitlab.com/ci/pipeline_security/
- GitLab CI/CD variables - https://docs.gitlab.com/ci/variables/
- GitLab CI/CD job token - https://docs.gitlab.com/ci/jobs/ci_job_token/
- GitLab token overview - https://docs.gitlab.com/security/tokens/
- GitLab scan execution policies - https://docs.gitlab.com/user/application_security/policies/scan_execution_policies/
- GitLab pipeline execution policies - https://docs.gitlab.com/user/application_security/policies/pipeline_execution_policies/
- GitLab audit event streaming for instances - https://docs.gitlab.com/administration/compliance/audit_event_streaming/
- GitLab audit event streaming for top-level groups - https://docs.gitlab.com/ee/user/compliance/audit_event_streaming.html
- CI/CD Goat repository - https://github.com/cider-security-research/cicd-goat
- OWASP Top 10 CI/CD Security Risks - https://owasp.org/www-project-top-10-ci-cd-security-risks/
- GitLab patch release example (security fixes ship in patch releases) - https://about.gitlab.com/releases/2026/03/11/patch-release-gitlab-18-9-2-released/

Selected release-track notes for security-focused readers

Release	Security-relevant change or focus area	Why you care operationally
17.4	Read-access sharing for linked pipeline execution policy files and cleanup of removed SAST analyzer debt.	Makes centrally enforced pipeline policies easier to operate and reduces stale vulnerability noise.
17.11	Protected container tags and job-source metadata for CI jobs.	Hardens release image control and improves artifact and policy provenance.
18.0	Security scanners can run in merge-request pipelines with AST control.	Moves security signal closer to review time and improves change gating.
18.1	Compromised-password detection for native credentials on GitLab.com and variable-precedence controls in pipeline execution policies.	Improves credential hygiene and makes policy-driven variable handling more precise.
18.4	Advanced SAST performance gains and tighter artifact-download controls.	Reduces scan friction and limits who can pull sensitive job outputs.
18.5	Compliance and security policy groups, policy-bypass exceptions with audit trails, diff-based scans, and custom Advanced SAST logic.	Strengthens centralized governance while preserving accountable emergency paths.
18.6	Modernized security dashboard experience on GitLab.com beta.	Improves security visibility, but do not confuse UI improvements with core control maturity.
18.9.x patch line	Patch releases continue to ship important bug and security fixes twice monthly, with urgent upgrade guidance for self-managed users.	For self-managed environments, patch hygiene remains a primary security control, not routine housekeeping.

EOF

GitLab CI/CD Hardening Field Guide

Fast audit and hardening guidance for engineers taking over an existing GitLab deployment

Author: Ivan Piskunov | Release date: 01 April 2026 | Version 1.0