SETUP GUIDE

GitHub Organization and CI/CD Hardening Guide for Web3 Teams

Locking down source control, secrets, and the release pipeline

Prepared for	Web3 team members	Classification	Internal
Prepared by	Oak Security	Date	2026-06-17

Oak Security GmbH

1. What This Guide Is For

This guide is for engineering leads, repository maintainers, and release owners on a Web3 team. Source control and CI are not a side concern. They are a direct path to production and to the published artifacts that users trust: smart contracts, frontends, SDKs, CLI tools, and container images. An attacker who controls a merge, a workflow run, or a release does not need to find a bug in your code. They write their own and ship it through your name.

The threat is not hypothetical. Long-lived tokens leaked in CI logs, malicious dependency updates, and compromised third-party actions have all been used to exfiltrate secrets and push backdoored builds. The supply chain is the soft target because most teams harden the application and leave the pipeline that builds it wide open.

> **NOTE**
>
> **Baseline rule:** enforce SSO plus phishing-resistant 2FA, protect branches and reviews, use least-privilege short-lived tokens, keep no long-lived secrets in CI, and sign your releases.

This guide is written around GitHub because that is what most teams use, but the principles are portable. SSO enforcement, branch protection, OIDC federation, action pinning, and release signing all have equivalents on GitLab, Bitbucket, and self-hosted forges. Read the GitHub specifics as concrete examples of a general model.

2. Organization and Access Model

Fix the access model before you tune individual settings. Most pipeline compromises start with an over-privileged account, a stale collaborator, or a default permission nobody reviewed.

Roles and Ownership

GitHub organizations have a small set of roles that matter: organization owners, members, and per-repository roles (read, triage, write, maintain, admin). Owners can change anything, including security settings, billing, and member access. Treat owner as the most dangerous role in your entire stack, because it can rewrite the rules that protect everything else.

Keep the number of owners minimal. Two or three is usually right for a small team. Every additional owner is another phishing target whose compromise is game over.

Access Building Blocks

Building Block	What To Do
SSO / SAML	Enforce SSO for the organization. Require members to authenticate through your identity provider so that disabling one IdP account cuts off access everywhere.
Teams	Grant repository access through teams, not to individuals. Map teams to function (frontend, contracts, infra) so access is auditable and easy to revoke.
Base permissions	Set the organization base permission to None or Read. Do not leave it at Write, which silently grants every member push access to every repository.
Outside collaborators	Use outside collaborators for contractors and auditors. Scope them to specific repositories, time-box the access, and remove them when the engagement ends.
Audit log	Treat the organization audit log as a primary signal. It records permission changes, secret access, OAuth grants, and SSO events. Stream it to

	your SIEM or logging stack if you have one.

Least Privilege and Review

Access granted is access forgotten. Run a periodic access review (at least quarterly) covering owners, team membership, outside collaborators, machine accounts, and installed apps. Remove anything without a current justification. Tie deprovisioning to your offboarding checklist so that a departing engineer loses source-control access the same day, not weeks later.

3. Quick Setup Checklist

These are the settings every organization should have before it ships anything users depend on.

Organization Security Settings

Setting	What To Do	Where To Check
SSO enforcement	Require SAML SSO for all members and outside collaborators.	Organization Settings > Authentication security
2FA requirement	Require two-factor authentication for everyone in the organization.	Organization Settings > Authentication security
Base permissions	Set base permission to None or Read.	Organization Settings > Member privileges
Repository creation	Restrict who can create public repositories and who can change repository visibility.	Organization Settings > Member privileges
Audit log streaming	Enable audit log review or streaming to your logging stack.	Organization Settings > Logs > Audit log

Two-Factor and Tokens

Area	What To Do
Phishing-resistant 2FA	Require security keys or passkeys for owners and release owners. TOTP is acceptable for general members but is phishable.
Personal access tokens	Prefer fine-grained tokens with explicit repository scope and short expiry. Avoid classic tokens.
GITHUB_TOKEN permissions	Set the default workflow token to read-only at the organization level and grant write per job only where needed.

Branches and Secrets

Area	What To Do
Branch protection	Protect the default branch with required reviews, required status checks, and no force-push.
CODEOWNERS	Require review from code owners on sensitive paths (contracts, CI config, release scripts).
Secrets	Move cloud and registry credentials to OIDC where possible. Keep no long-lived secrets in CI that you can avoid.
Third-party apps	Restrict OAuth app and GitHub App access. Require owner approval for new installs.

4. Authentication and Token Hygiene

Credentials are the attack surface. Everything below reduces the number of long-lived, broadly-scoped, phishable secrets that can reach your code or your pipeline.

Human Authentication

- Require 2FA for the whole organization, not just owners.
- For owners and release owners, require phishing-resistant methods: WebAuthn security keys or passkeys. TOTP and SMS can be relayed by a real-time phishing proxy.
- Enforce SSO so that identity-provider deprovisioning revokes GitHub access in one place.
- Review registered 2FA methods and recovery options. A weak recovery path (SMS, email) undermines a strong primary factor.

SSH vs HTTPS

Both are fine for normal use. The risk is not the protocol, it is the key. Use a dedicated work SSH key, protect it with a passphrase, and prefer hardware-backed keys for access to sensitive repositories. Do not sync private keys through cloud storage or chat.

Personal Access Tokens

Token Type	Use For	Caveat
Fine-grained PAT	Scripted access scoped to specific repositories and specific permissions, with a short expiry.	The right default. Still a bearer credential: store it in a secret manager, never in code or shell history.
Classic PAT	Legacy integrations that do not support fine-grained tokens yet.	Broad scopes (`repo`, `workflow`) grant access to every repository the user can see. Avoid. Migrate off these.
Deploy key	Single-repository machine access (read, or write if	Scoped to one repository, which limits blast radius, but a leaked

	required).	write deploy key still lets an attacker push. Set read-only unless write is needed.

Set the shortest expiry the workflow tolerates. A token that expires in 30 days is a token an attacker cannot use in 31.

The Default GITHUB_TOKEN

GitHub Actions injects a `GITHUB_TOKEN` into every workflow run. Set its default permission to read-only at the organization or repository level, then grant `permissions:` per workflow or per job only where a write is genuinely needed (for example, publishing a release or pushing a tag). A workflow that only builds and tests does not need write access to your repository, and granting it anyway hands free privilege to any injected code.

Installed Apps and Offboarding

- Review installed OAuth apps and GitHub Apps. Each one holds a token against your organization. Remove anything unused or unrecognized.
- Restrict who can authorize new OAuth apps and require owner approval for GitHub App installs.
- Wire deprovisioning to offboarding. SCIM with your identity provider automates removal; if you do not use SCIM, removal must be a manual offboarding step that actually happens the same day.

5. Branch, Review, and Commit Integrity

The default branch is what gets built and shipped. If an attacker can write to it directly, or merge without review, nothing downstream matters.

Branch Protection and Rulesets

Protect the default branch (and any release branches) with a ruleset that:

- requires pull requests before merging, with at least one approving review;
- dismisses stale approvals when new commits are pushed;
- requires review from code owners on sensitive paths;
- requires status checks to pass before merge;
- requires branches to be up to date before merge;
- blocks force-pushes;
- blocks direct pushes to the protected branch, including by admins.

Rulesets apply across multiple repositories and branches from one definition, which is easier to audit than per-repository protection. Use them for organization-wide policy.

CODEOWNERS and Required Review

Add a `CODEOWNERS` file and require code-owner review on the paths where a malicious change does the most damage: smart contract source, CI workflow definitions, release and deploy scripts, dependency manifests, and the `CODEOWNERS` file itself. The reviewer requirement is only as strong as the paths it covers, so cover the dangerous ones explicitly.

Required Status Checks

Require the checks that actually gate quality and security: tests, linters, dependency review, and any security scanning. A required check that can be skipped is decoration. Make sure the check cannot be satisfied by a workflow the pull request author controls without review.

Commit and Tag Integrity

Control	What To Do
Signed commits	Require signed, verified commits on protected branches. This binds commits to a known key and makes spoofed author identities visible.
Linear history	Require linear history to keep the branch auditable and avoid merge-commit obfuscation.
Tag protection	Protect release tags with rules so that tags cannot be created or moved by unauthorized users. A moved tag can point a release at attacker code.
Release protection	Restrict who can publish releases. Treat publishing as a privileged action, separate from merge access.

Author identity in git is self-asserted. Without signing, "committed by" a trusted maintainer means nothing. Verified signatures plus protected tags are what make the history trustworthy.

6. CI/CD and Supply-Chain Security

CI runs code on every push and pull request, often with access to secrets. Treat the runner as hostile territory and minimize what it can reach.

Secrets vs OIDC Federation

Stored Actions secrets are long-lived bearer credentials. Every workflow that can read them, and every action that runs inside such a workflow, can exfiltrate them. Replace them with short-lived credentials wherever the target supports it.

Use OIDC federation to exchange a workflow's signed identity token for short-lived cloud credentials (AWS, GCP, Azure, and any provider that trusts GitHub's OIDC issuer). The workflow never holds a static cloud key, and you scope the trust policy to specific repositories, branches, and environments. This removes the most valuable secret from CI entirely.

Pinning Third-Party Actions

Pin every third-party action to a full commit SHA, not a tag or branch:

- a tag like `@v4` is mutable: the maintainer (or an attacker who compromises them) can move it to point at new code;
- a full 40-character commit SHA is immutable: you run exactly the code you reviewed.

Pin first-party and well-known actions by SHA too where the risk justifies it. Combine pinning with a tool like Dependabot to surface updates, so pinning does not mean running stale, vulnerable actions forever. Review the diff before bumping a pin.

Dangerous Workflow Patterns

Pattern	Why It Is Dangerous
`pull_request_target` + checkout of PR head + secrets	`pull_request_target` runs with repository secrets and write-capable token, in the context of the base repository, but checking out the untrusted PR head executes attacker-controlled code with those secrets in scope. This is a well-

	known exfiltration path. Do not check out untrusted code in a privileged trigger.
Untrusted input in run steps	Interpolating PR titles, branch names, or issue bodies directly into `run:` shell steps allows script injection. Pass untrusted values through environment variables and quote them.
Self-hosted runners on public repos	A public repository's CI can be triggered by anyone's pull request. A self-hosted runner gives that arbitrary code a foothold on your infrastructure, with persistence between jobs. Never attach self-hosted runners to public repositories.
Secrets echoed into logs	Logs are visible to anyone with read access and are retained. Never print secrets, and be aware that masking is best-effort, not a guarantee.

Runner Isolation

If you must use self-hosted runners, isolate them: ephemeral runners that are destroyed after each job, no standing credentials, no network access beyond what the job needs, and never on public repositories. A persistent self-hosted runner is a shared machine that every workflow can modify for the next workflow.

Dependencies and Provenance

- Pin dependencies (lockfiles, exact versions) and review dependency updates rather than auto-merging them. A malicious version bump is a common supply-chain entry point.
- Sign release artifacts and generate build provenance. SLSA provenance attests to how and where an artifact was built, so consumers can verify it came from your pipeline and not a side channel.
- When publishing to a package registry (npm, crates.io, container registries), require 2FA for publishing accounts and publish with provenance where the registry supports it. npm supports provenance attestation generated from a trusted CI build.

7. The Recommended Setup Flow

Step 1: Enforce SSO and 2FA

Turn on SAML SSO for the organization and require it for all members and outside collaborators. Require 2FA organization-wide. For owners and release owners, require phishing-resistant security keys or passkeys. Verify that no member is grandfathered out of the requirement.

Step 2: Set Base Permissions and Teams

Set the organization base permission to None or Read. Create teams mapped to function and grant repository access through them. Move any individual grants onto teams. Restrict repository creation and visibility changes to roles that should have them.

Step 3: Configure Branch Rulesets

Create a ruleset for the default branch (and release branches) that requires pull requests, approving reviews, code-owner review on sensitive paths, passing status checks, linear history, and signed commits, and that blocks force-pushes and direct pushes including for admins. Apply it across repositories rather than configuring each one by hand.

Step 4: Audit Tokens and Third-Party Apps

Inventory personal access tokens, deploy keys, OAuth apps, and GitHub Apps. Revoke unused and over-scoped credentials. Migrate classic PATs to fine-grained tokens with short expiry. Set the default `GITHUB_TOKEN` to read-only and require owner approval for new app installs.

Step 5: Move Secrets to OIDC and Pin Actions

Replace static cloud credentials in CI with OIDC federation, scoping trust policies to specific repositories and environments. Pin every third-party action to a full commit SHA. Set up automated update surfacing so pins stay current. Review workflows for `pull_request_target` plus untrusted checkout and remove that pattern.

Step 6: Enable Signed Commits and Signed Releases

Require verified signed commits on protected branches. Protect release tags. Sign release artifacts and generate SLSA provenance. For package publishing, require 2FA on publishing accounts and enable provenance attestation.

Step 7: Monitor the Audit Log and Set Alerts

Review or stream the organization audit log. Alert on high-signal events: new owners, changed branch protection or rulesets, new OAuth or GitHub App installs, new deploy keys, new self-hosted runners, and secret access. The earlier you see an anomalous permission change, the smaller the incident.

8. Extra Rules for Critical-Access Users

Some accounts can turn a single compromise into a company-level incident: organization owners and release owners. Apply the stricter profile below to them.

Control	Requirement
Security-key-only auth	Owners and release owners authenticate with WebAuthn security keys or passkeys. No TOTP or SMS as the primary factor for these accounts.
Minimal owners	Keep organization owners to the smallest workable number (two to three). Every owner can disable every protection.
Protected environments	Configure deployment environments with required reviewers and branch restrictions, so a production deploy requires explicit human approval from someone other than the author.
Deployment approvals	Gate production and release deployments behind manual approval. Separate "can merge" from "can ship."
Break-glass account	Maintain a documented, monitored break-glass owner account with credentials in secure offline storage, used only when normal access is unavailable. Alert on any use.
Separation of duties	The person who authors a release should not be the only person who approves and publishes it where the artifact is high-value.
Token discipline	No long-lived broadly-scoped PATs. Use OIDC and short-lived, environment-scoped credentials for privileged automation.
Suspicious activity	Rotate credentials and review the audit log before resuming privileged work after any suspected compromise.

9. Things To Avoid

These are common and each one has been used in real supply-chain attacks.

- Long-lived personal access tokens in CI, especially classic PATs with broad scopes.
- Unpinned third-party actions referenced by mutable tags or branches.
- `pull_request_target` (or other privileged triggers) that check out untrusted PR code with secrets in scope.
- Self-hosted runners attached to public repositories.
- Secrets echoed into build logs, or relying on log masking as protection.
- A broad organization base permission (Write) that grants every member push access to every repository.
- Unreviewed OAuth apps and GitHub Apps holding tokens against your organization.
- An unprotected default branch that allows direct pushes, force-pushes, or merges without review.
- Auto-merging dependency updates without reviewing the diff.
- Treating a tag-based release as immutable when the tag can be moved.

10. If Something Suspicious Happens

Treat suspected compromise of a token, a runner, or the pipeline as urgent. The goal is to cut off access and rotate everything reachable before assessing damage, not after.

Immediate Actions

1. Revoke the suspected token, deploy key, or app credential immediately. Do not wait to confirm it was the vector.
2. Rotate all secrets the affected workflow or account could read. Assume exposure of anything in scope.
3. Invalidate sessions for the affected accounts and force re-authentication.
4. Suspend or restrict the affected account or runner so it cannot act while you investigate.

Review the Audit Trail

Area	Action
Audit log	Review recent permission changes, secret access, new owners, new collaborators, and SSO events.
Actions runs	Review recent workflow runs for unexpected jobs, modified workflows, and exfiltration steps (outbound network calls, encoded payloads).
Collaborators	Look for new or unexpected outside collaborators and team additions.
Webhooks	Look for new or modified webhooks that could be leaking events or payloads.
Deploy keys	Look for new deploy keys and remove any you cannot account for.
Self-hosted runners	Look for new or modified runners. Quarantine and rebuild any that may be compromised.

Contain the Supply Chain

- Re-pin or quarantine any third-party action involved in the incident. If a dependency or action is compromised upstream, pin to a known-good SHA or remove it.
- Rotate any package-registry publishing credentials and review recent published versions for tampering.
- If a release may have shipped compromised, treat the published artifact as suspect: yank or deprecate it, publish a clean signed build, and notify consumers.

The general lesson is unforgiving: a single compromised third-party action has, in real incidents, leaked secrets across many repositories at once, because everyone referenced it by a mutable tag and ran whatever the maintainer's account pushed. SHA-pinning, least-privilege tokens, and keeping no long-lived secrets in CI are what limit the blast radius when (not if) an upstream component is compromised.

11. Monthly Self-Check

Once a month, run this short review across the organization.

Check	Done
SSO is enforced for all members and outside collaborators.	
2FA is required organization-wide; owners and release owners use security keys or passkeys.	
Organization base permission is None or Read.	
Owner count is minimal and every owner is accounted for.	
The default branch is protected: required reviews, status checks, no force-push, no direct push.	
CODEOWNERS covers contracts, CI config, and release scripts.	
Signed commits and tag protection are enforced on protected branches.	
Personal access tokens have been reviewed; no broad classic PATs in CI.	
Default GITHUB_TOKEN is read-only, with write granted per job only where needed.	
Cloud credentials in CI use OIDC, not long-lived stored secrets.	
Third-party actions are pinned to full commit SHAs.	
No self-hosted runners are attached to public repositories.	
OAuth apps, GitHub Apps, and deploy keys have been reviewed.	
Release artifacts are signed and publish with provenance; publishing accounts require 2FA.	

The audit log has been reviewed and alerts on high-signal events are working.	
Stale collaborators, team members, and tokens have been removed.	

12. References

- GitHub organization security best practices: https://docs.github.com/en/organizations/keeping-your-organization-secure/managing-security-settings-for-your-organization/best-practices-for-organizational-security
- GitHub account security hardening: https://docs.github.com/en/authentication/keeping-your-account-and-data-secure
- GitHub branch protection rules and rulesets: https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/managing-rulesets/about-rulesets
- GitHub Actions security hardening: https://docs.github.com/en/actions/security-for-github-actions/security-guides/security-hardening-for-github-actions
- OpenID Connect in GitHub Actions: https://docs.github.com/en/actions/security-for-github-actions/security-hardening-your-deployments/about-security-hardening-with-openid-connect
- GitHub Actions self-hosted runner security: https://docs.github.com/en/actions/how-tos/manage-runners/self-hosted-runners/manage-access
- OpenSSF Scorecard: https://github.com/ossf/scorecard
- SLSA (Supply-chain Levels for Software Artifacts): https://slsa.dev/
- npm package provenance: https://docs.npmjs.com/generating-provenance-statements
- npm two-factor authentication: https://docs.npmjs.com/configuring-two-factor-authentication