GUIDE

Secrets Management Guide for Web3 Teams

Storing, distributing, rotating, and detecting the secrets behind your stack

Prepared for	Web3 teams	Classification	Public
Prepared by	Oak Security	Date	2026-06-17

Oak Security GmbH

1. What This Guide Is For

This guide is for engineers and operators who hold the credentials a Web3 company runs on: API keys, RPC and node credentials, cloud and KMS keys, deploy keys, CI tokens, and the environment configuration that ties them together. It treats secrets management as a discipline that cuts across every system you run: how secrets are stored, how they reach the people and machines that need them, how they are rotated, and how you detect them when they leak.

This is deliberately not a wallet-key guide. Signing-key custody, multisig ceremonies, and treasury operations are covered in the [Multisig Treasury Operations Setup Guide](/resources/multisig-treasury-operations-setup-guide). Source control and the release pipeline are covered in the [GitHub Organization and CI/CD Hardening Guide](/resources/github-org-cicd-hardening-guide). Cloud and KMS configuration is covered in the [Cloud Account and KMS Hardening Guide](/resources/cloud-account-kms-hardening-guide). This guide is the layer underneath all three: the secrets themselves, regardless of which system holds them.

The reason this layer needs its own discipline is that teams consistently underestimate how many secrets they hold. Asked how many keys they manage, most teams name one: the treasury multisig threshold. The real answer for a working company is twenty or more, and most of them never touch a wallet. The team that loses money is the one that never wrote the list down.

> **NOTE**
>
> **Baseline rule:** no secret in a repository, ever. Every secret has a named owner, a storage tier, and a rotation cadence. Prefer short-lived credentials over static ones, scope every token to the minimum it needs, and scan for leaks on every push.

2. The Secret Inventory

The estate is wider than the wallet keys. Before you can protect secrets, you have to enumerate them. The table below is the working inventory most teams should hold. Each row needs a named owner (a person, not a team), a storage location, and a rotation cadence recorded against it. A secret with none of those is not under management. It is in the wild.

Secret type	Examples	Where it belongs	Handling
Third-party API keys	Infura, Alchemy, exchange APIs, data providers, analytics	Secrets manager; injected at runtime	Scope to the narrowest permission set. Rotate on cadence. Never in repos.
Signing keys (off-chain infra)	Validator signing keys, relayer keys, oracle signers, fee-payer	HSM or MPC, never a file on the box	A remote signer sits between the process and the key, with its own audit log. See the treasury guide for on-chain custody.
RPC / node credentials	Authenticated RPC gateways, node admin tokens, peer auth	Secrets manager; private network only	JSON-RPC defaults to loopback. Authenticated gateway in front of any public RPC.
Cloud / KMS keys	Cloud root, KMS key references, IAM access keys, storage bucket credentials	Cloud KMS plus short-lived assumed roles	Prefer STS/Workload Identity over static keys. See the cloud/KMS guide.
Deploy keys	GitHub deploy keys, registry push tokens, npm/PyPI/Cargo publish tokens	Secrets manager or OS keychain; CI via OIDC	Single-repo scope, read-only unless write is required. Revoke on offboarding the same day.
Environment config	`.env` files, app config holding DB credentials, webhook secrets	Secrets manager; fetched at runtime	Never committed. Never baked into images. `.env` files are gitignored and local-only.

CI tokens	Actions secrets, deploy credentials, registry login	OIDC-issued short-lived tokens	Replace stored long-lived secrets with federated identity wherever the target supports it.
Auth material	TOTP seeds, SSO signing certs, TLS keys, MFA backup codes, SSH keys	OS keychain, HSM, or physical storage	These issue or authenticate everything else. Treat them as the highest tier.

The recurring accident across all of these is the same one: the secret ends up in source control. The `.env` committed by mistake, the test fixture with real values, the `private_key.txt` on a desktop, the API key pasted into a comment. Section 4 and Section 7 address this directly, because it is the single failure that shows up in nearly every post-mortem.

3. Secret Storage: Vaults and What They Buy You

Every secret that is not a hardware-backed credential lives in a secrets manager. The options that recur in Web3 stacks: HashiCorp Vault, 1Password Service Accounts, Doppler, AWS Secrets Manager, GCP Secret Manager. Applications fetch their secrets at runtime. Secrets are never baked into container images and never committed.

A vault buys you four things a `.env` file does not: a central audit trail of who read what and when, access control per secret rather than per file, programmatic rotation, and a single place to revoke. The audit trail is the control. A team that cannot answer "who has access to this credential right now?" with a definitive list does not have management, only optimism.

> **NOTE**
>
> **Adopt a vault before you have a leak, not after.** The point at which a startup wires every service to read from a manager is much cheaper than the point at which it untangles secrets scattered across repos, CI configs, and laptops after an incident. If you are past a handful of services or a handful of people, you are past the point where `.env` files on disk are defensible.

For secrets that genuinely must live in Git (sealed Kubernetes secrets, encrypted Terraform state), use KMS-backed encryption that decrypts only at deploy time. SOPS (Mozilla's Secrets OPerationS) and age (modern file encryption with KMS-backed keys) both do this. The plaintext never sits in the repo:

bash # Encrypt a secrets file with SOPS against a KMS key. # The repo only ever holds the ciphertext. sops --encrypt --kms arn:aws:kms:eu-central-1:111122223333:key/abcd \ secrets.yaml > secrets.enc.yaml # At deploy time, the pipeline (with the KMS role) decrypts in memory. sops --decrypt secrets.enc.yaml | kubectl apply -f -

4. Local Development and .env Hygiene

Local development is where secrets leak most casually. A developer copies a production key into `.env` to debug "just this once," commits the file by reflex, and the key is now in the Git history forever.

Two rules close most of the gap. First, never put production secrets in a local `.env`. Local development uses test keys, test accounts, and test funds that are strictly separate from production. The phrase "just use the production key to debug, it's only the once" is the prelude to a post-mortem. Second, make committing a `.env` mechanically impossible.

```
gitignore # .gitignore: keep every flavour of local secret out of the repo .env .env.* !.env.example *.key *.pem secrets.*.yaml !secrets.enc.yaml
```

Commit a `.env.example` with the variable names and dummy values so the shape of the config is documented without any real secret. Note the negation patterns above: `.env.example` and SOPS-encrypted files are explicitly re-included, because those are safe to track.

> **NOTE**
>
> **A secret in Git history is leaked even after you delete the file.** Removing a committed secret in a later commit does not remove it from history. The fix is to rotate the secret, not to rewrite history and hope nobody cloned the repo. Treat any secret that ever touched a repo as burned.

Access tokens belong in the OS keychain (the Keychain on macOS, Credential Manager on Windows, libsecret on Linux), not in a `.env` file. SSH keys carry a strong passphrase, with the passphrase stored in the OS keychain rather than typed each time. Never copy private keys between machines and never sync them through cloud storage or chat.

5. Distributing Secrets to People and to CI/CD

A secret is only useful if it reaches the people and machines that need it. Distribution is where scope and lifetime get decided, and where the difference between a contained incident and a company-wide one is set.

Distributing to People

Use the vault's own sharing, not Slack, email, or a shared document. 1Password Service Accounts, Vault policies, and Doppler all grant per-secret access to named identities with an audit trail. When you onboard someone, grant the secrets their role needs through a group or policy, not by pasting values. When they leave, you revoke the group membership and every secret goes with it.

The discipline that pays off is least privilege even for privileged roles. A developer who needs read access to one service's config does not need the database master credential. A token that can read one repo cannot publish to npm; a token that can publish to npm cannot read the database. Every person and every machine identity is scoped to what its job requires and nothing more.

Distributing to CI/CD: Prefer OIDC Over Long-Lived Secrets

Stored CI secrets are long-lived bearer credentials. Every workflow that can read them, and every dependency that runs inside such a workflow, can exfiltrate them. The fix is to stop storing long-lived cloud credentials in CI at all.

Use OIDC federation: the CI job presents a signed identity token, exchanges it for a short-lived cloud credential scoped to the specific repository, branch, and environment, and that credential disappears when the job ends. There is no static key in the pipeline to steal.

yaml # GitHub Actions: assume an AWS role via OIDC. No stored AWS keys. permissions: id-token: write # allow the job to request the OIDC token contents: read steps: - uses: aws-actions/configure-aws-credentials@<pinned-sha> with: role-to-assume: arn:aws:iam::111122223333:role/deploy-prod aws-region: eu-central-1 # the role's trust policy restricts which repo/branch/env may assume it

The same model exists for GCP (Workload Identity Federation) and Azure. Where a static secret is genuinely unavoidable (a third-party API with no federation support), store it in the vault, scope it as tightly as the provider allows, give it the shortest expiry the workflow tolerates, and rotate it on cadence. A credential that lives longer than a single deploy is too long-lived for production. Details on pinning actions, the default `GITHUB_TOKEN`, and dangerous workflow patterns are in the [CI/CD hardening guide](/resources/github-org-cicd-hardening-guide).

6. Rotation

Rotation is routine. Compromise is exceptional. Both need a runbook with a named owner, and both need to have been rehearsed before they happen under pressure.

Rotation on Cadence

Different secrets warrant different cryptoperiods. A relayer key that signs every block, a code-signing key that signs a monthly release, and a third-party API key have different exposure and tolerate different lifetimes. NIST SP 800-57 Part 1 Rev. 5 is the framework for reasoning about this: not a single cadence, but a structured way to decide how long any given key should live before replacement.

Secret class	Suggested cadence	Note
CI / deploy tokens	Per-job (OIDC) or 30 days max	Short expiry is the whole defence. A 30-day token cannot be used on day 31.
Third-party API keys	90 days	Automate where the provider supports key versioning.
Cloud static keys (where unavoidable)	90 days, or eliminate via STS	Prefer assumed roles so there is nothing to rotate.
TLS keys	Periodically, regardless of patch state	The Heartbleed lesson: a long-lived TLS key may already have been harvested by a zero-day.
Signing-infra keys (relayer, oracle)	Per the key's blast-radius tier	High-frequency keys rotate more often than low-frequency ones.

Rotation on Departure

Rotate on every personnel change, not on a schedule that happens to be convenient. Leavers lose access within 48 hours, not "soon," not "next sprint." Wire this to the offboarding checklist so it actually happens.

The Ledger Connect Kit incident (December 2023, roughly $600,000) is the canonical case. A former Ledger employee who had left months earlier still held npm publish rights on `@ledgerhq/connect-kit`. The attacker phished that dormant account and pushed three malicious versions that propagated a wallet drainer through every dApp loading the package via an unpinned CDN URL. The credential should have been revoked on the day the employee left. Role changes trigger the same rule: a developer who becomes a signer, or a signer who leaves the set, triggers rotation of every credential the change affected.

> **NOTE**
>
> **If in doubt, swap it out.** Compromise-suspected is compromise-assumed. The cost of rotating a key is bounded. The cost of being wrong about whether it is still safe is not. A "maybe" is treated as a "yes" for rotation.

7. Detecting Leaked Secrets

Prevention fails eventually. Detection is what bounds the damage when it does.

Secret Scanning

Run secret scanning in CI on every push, and run it on the whole history, not just the diff. Tools: GitHub secret scanning with push protection (blocks the commit before it lands), Gitleaks, TruffleHog, detect-secrets. Push protection is the highest-value control because it stops the leak at the source rather than detecting it after.

> bash # Pre-commit hook or CI step: fail the build if a secret is detected. gitleaks protect --staged --redact --verbose # Scan full history when onboarding a repo into scanning: gitleaks detect --source . --redact

Scanning is necessarily best-effort. It catches known patterns (AWS keys, tokens with recognisable prefixes, high-entropy strings). It will miss a custom-format secret. Combine it with `.gitignore` discipline (Section 4) so secrets do not reach the staging area in the first place.

Canary Tokens

Plant credentials that have no legitimate use and exist only to fire an alert when touched: a fake AWS key in a tempting location, a honeytoken in a config file, a canary URL in documentation. Services like Canarytokens generate these. If a canary in your repo or build environment is ever used, you have an intruder, with no false-positive ambiguity. Canaries are cheap and they catch the attacker who is already inside, after the scanner failed.

What To Do When One Leaks

The order matters. Cut off access before assessing damage, not after.

1. Rotate the leaked secret immediately. Do not wait to confirm it was exploited. Assume it was.
2. Revoke the old credential at the provider so the leaked value stops working.
3. Review the audit trail for use of the credential between leak and rotation: unexpected API calls, logins, deploys, outbound transfers.
4. If the secret reached a public repo or a package, treat anything it could access as exposed and rotate that too. Blast radius is transitive.
5. Record the incident: what leaked, how, when it was caught, what was rotated. The record is what stops the same leak recurring.

8. The Supply-Chain Angle

A secret does not have to be committed to leak. A compromised dependency can read every environment variable in the process and exfiltrate it. This is the failure mode that turns a tightly-managed `.env` into a liability the moment a single transitive package turns malicious.

A typical dApp pulls 500 to 1,500 transitive packages at build time, and every one of them executes: `npm install` runs `postinstall` scripts, `pip install` runs `setup.py`, Cargo runs build scripts. The Solana web3.js incident (December 2024, roughly $184,000) is the model. A phished maintainer account published `@solana/web3.js` versions 1.95.6 and 1.95.7 with a backdoor that exfiltrated private keys through an obscure HTTP header (`CloudFlare-Ipcountry`) chosen to evade traffic monitoring. Anyone who had pinned `1.95.5` was untouched; anyone running `^1.95.0` who rebuilt during the window pulled the backdoor.

The secrets-management defences against this class:

- **Minimise what is in scope.** Build steps and runtime processes should hold only the secrets they actually need. A build that does not deploy does not need deploy credentials in its environment.
- **Short-lived over static.** A backdoor that exfiltrates an OIDC-issued token gets a credential that expires in minutes. A backdoor that exfiltrates a static cloud key gets one that works until someone notices.
- **Pin everything.** Lockfiles with integrity hashes, SRI on CDN script tags, container digests rather than tags. "Latest" is the version the attacker ships next.
- **Egress awareness.** A build environment that can only reach the registries and endpoints it needs cannot quietly POST your environment to an attacker's server.

Pinning, signed artefacts (Sigstore, Cosign, npm provenance), and SLSA build provenance are covered in depth in the [CI/CD hardening guide](/resources/github-org-cicd-hardening-guide). The point here is narrower: your secrets are only as protected as the code that runs alongside them.

9. The Gold-Standard Workflow

The Oak end-state for secrets management:

- **The inventory is written down.** Every secret has a named owner, a storage tier, a rotation cadence, and a blast-radius note. The list is reviewed regularly, and the keys teams forget (infrastructure and developer credentials) are on it.
- **No secret in a repository, ever.** Not in `.env.example`, not in tests, not in comments. Secret scanning with push protection runs on every push. Leaked secrets are rotated within the hour.
- **Vaults hold everything not hardware-backed.** Applications fetch at runtime. Nothing is baked into images. Secrets that must live in Git are SOPS/age-encrypted against KMS.
- **Short-lived credentials by default.** OIDC from CI into scoped, ephemeral cloud roles. STS/Workload Identity over static keys. Any credential outliving a single deploy is justified explicitly or eliminated.
- **Tokens scoped narrowly.** Fine-grained over broad. One capability per token. Read where read suffices.
- **Rotation is routine and tested.** On cadence per NIST 800-57, and on every personnel change within 48 hours. The first rotation is not performed under incident pressure.
- **Detection is layered.** Scanning catches the careless commit; canary tokens catch the intruder the scanner missed; the audit trail bounds the damage when both fail.

10. Checklist

Check	Done
The full secret inventory is written down, each entry with a named owner, storage tier, and rotation cadence.	
No secrets in any repository; `.gitignore` covers `.env`, `*.key`, `*.pem`, and secrets files.	
Secret scanning with push protection runs in CI on every push, and full history has been scanned.	
A secrets manager holds every non-hardware-backed secret; apps fetch at runtime.	
Secrets that live in Git are encrypted with SOPS/age against a KMS key.	
Local development uses test keys and test funds, never production secrets.	
CI uses OIDC into short-lived scoped cloud roles; no long-lived cloud keys stored in CI.	
Every token is scoped to the minimum permission and the shortest expiry the workflow tolerates.	
Rotation cadences are set per secret class and actually executed.	
Offboarding revokes every credential the same day; publish rights are removed on departure.	
Canary tokens are planted in repos and build environments, wired to alerts.	
A leaked-secret runbook exists (rotate, revoke, review audit trail, record) and has been rehearsed.	
Build and runtime processes hold only the secrets they need; egress from build environments is constrained.	

11. References

- HashiCorp Vault: https://developer.hashicorp.com/vault/docs
- SOPS (Secrets OPerationS): https://github.com/getsops/sops
- age (file encryption): https://github.com/FiloSottile/age
- GitHub OIDC for cloud authentication: https://docs.github.com/en/actions/security-for-github-actions/security-hardening-your-deployments/about-security-hardening-with-openid-connect
- GitHub secret scanning and push protection: https://docs.github.com/en/code-security/secret-scanning/about-secret-scanning
- Gitleaks: https://github.com/gitleaks/gitleaks
- TruffleHog: https://github.com/trufflesecurity/trufflehog
- Canarytokens: https://canarytokens.org/
- NIST SP 800-57 Part 1 Rev. 5 (key management): https://csrc.nist.gov/pubs/sp/800/57/pt1/r5/final
- Companion: GitHub Organization and CI/CD Hardening Guide: /resources/github-org-cicd-hardening-guide
- Companion: Cloud Account and KMS Hardening Guide: /resources/cloud-account-kms-hardening-guide
- Companion: Multisig Treasury Operations Setup Guide: /resources/multisig-treasury-operations-setup-guide