

Secrets, Keys and Auth

Chris Timperley · Wednesday, September 9, 2026

Administrivia

- HW1 grading is in progress
- HW2 is now released
  - Checkpoint A is due September 13th
  - Checkpoint B is due September 20th
- **Goal: build a fast CI pipeline that is able to catch developer mistakes**
  - Bonus points for speed and ability to spot particularly tricky mistakes

🎯 Learning Goals

- Explain why secrets need lifecycle controls that ordinary settings do not
- Distinguish principals, credentials, authentication, authorization, and permissions
- Explain how credential scope and lifetime limit blast radius after a leak
- Compare different storage and delivery methods and their tradeoffs
- Diagnose a credential leak and recommend controls before, during, and after exposure, including complete rotation

What did we talk about last week?



INTEGRATE

Change in small batches

Frequent merges make failures easier to locate.



CHECK

Automate useful evidence

Run credible checks after each change.



PROTECT

Keep the main branch working

Repair a failed main branch immediately.



IMPROVE

Keep feedback trustworthy

Reduce slow, flaky, and noisy checks.

⚠ Your CI pipeline might expose secrets!

```yaml
name: Deploy Tiny Notes
on: push

jobs:
  deploy:
    runs-on: ubuntu-latest
    env:
      DEPLOY_TOKEN: "DEMO_ONLY_tn-deploy-7F3K"
    steps:
      - uses: actions/checkout@v4
      - run: ./deploy.sh
```

Credential Exposure

What makes a leaked secret dangerous?

Copying a secret can copy access or authority



KEY CONCEPT

Secret

A value whose copy gives its holder access or authority.

A password grants account access · an Azure SAS URL grants stored-data access · a private key authorizes transfers

The value's effect determines whether it is a secret

Secrets

Account password
Authenticates whoever presents it

Private signing key
Authorizes new signatures

Signed storage URL
Grants the access encoded in the URL

Non-secrets

Account username
Identifies an account but doesn't grant access

Public verification key
Checks signatures and is meant to be shared

Ordinary storage URL
Names a resource but doesn't grant access

Secrets must be hidden, replaceable, and tracked



CONFIDENTIALITY

Hidden

Fewer people and systems can copy it.



RECOVERY

Replaceable

Issue a replacement and revoke the leaked credential.



LIFECYCLE

Tracked

Find every consumer and copy during response.

An ordinary setting does not carry the same access when copied.

First, ask which systems accept the credential



Then, ask what the credential permits



Reachable systems and allowed actions define the credential's scope.

Next, ask how long the credential works



ACCESS

Which systems accept it?

PERMISSIONS

What actions are allowed?

LIFETIME

When does it stop working?

NEXT QUESTION

Finally, ask where copies may remain



ACCESS
Which systems accept it?

PERMISSIONS
What actions are allowed?

LIFETIME
When does it stop working?

COPIES
Where else could it remain?

Blast radius depends on access, permissions, lifetime, and copies.

Copies can outlive the place where the leak began

Removing the first copy may leave the same value in:

- source history and forks
- logs and build artifacts
- caches and container images
- backups

Someone may recover the credential from any remaining copy!

```
build.log
08:41:22 DEPLOY_TOKEN=...7F3K

artifact/env.txt
DEPLOY_TOKEN=...7F3K

image/config.json
"DEPLOY_TOKEN=...7F3K"
```

Deleting leaked text does not revoke the credential

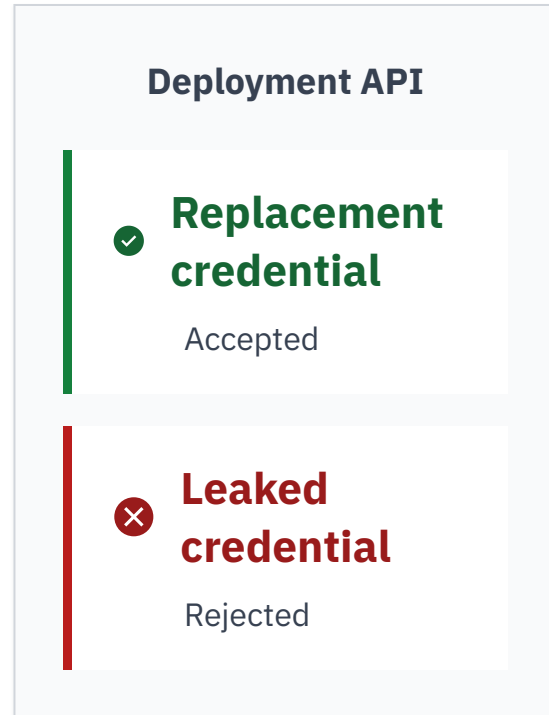
Repository cleanup can make a leaked value harder to find. It cannot remove copies that someone already made or stop the credential from working.

Contain the exposure

Revoke the leaked credential promptly, even if consumers must stop while you replace it.

Complete rotation requires both outcomes

- Every intended consumer works with the replacement.
- The leaked credential is rejected.



Identities, Credentials, and Permissions

Who is acting, what proves it, and what may they do?

One request raises three different questions



WHO IS ACTING?

Principal

The CI deployment job



WHAT IS THE PROOF?

Credential

A deployment token



WHAT MAY IT DO?

Permission

Update the staging service

People and workloads can both be principals

A **workload** is software that performs work (e.g., CI job, running service, AI agent).

Example	Classification	Why
Maya	Human principal	She can make requests as a person
CI deployment job	Workload principal	It makes deployment requests
Tiny Notes service	Workload principal	It makes requests while running
API key	Credential, not a principal	It is proof, not an actor

Give each workload its own credential, limited to the resources and actions it needs.

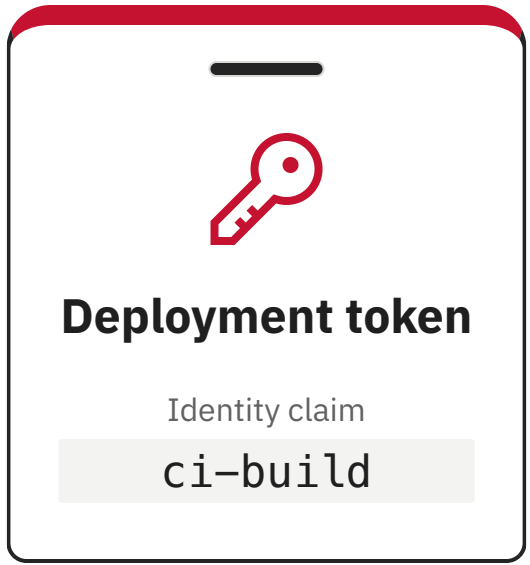
A credential is proof, not the actor

Systems may accept different forms of proof:

- a password
- an access token
- a certificate, together with proof of control of its corresponding key

A signing key can create signed credentials that other systems accept.

With a **bearer token**, possession is enough: someone with a stolen copy can present it as the owner.



How public-key cryptography works

Valid proof does not grant every permission



Authentication ✓

The token is valid. Identity accepted: ci-build.

Authorization ✗

ci-build cannot delete the production database.

A valid credential does not permit every action.

Credential Scope and Lifetime

How much can a stolen credential do, and for how long?

Stolen credentials exposed personal data



U.S. Department of Justice · July 22, 2022

> **used stolen credentials to access a private source code repository and obtain a private access key.**
>
> — U.S. DEPARTMENT OF JUSTICE, 2022

≈57 million
user records copied

Attackers used the recovered key to access the data.

Limit the possible damage before a credential leaks

SCOPE
Allowed resources and actions

+

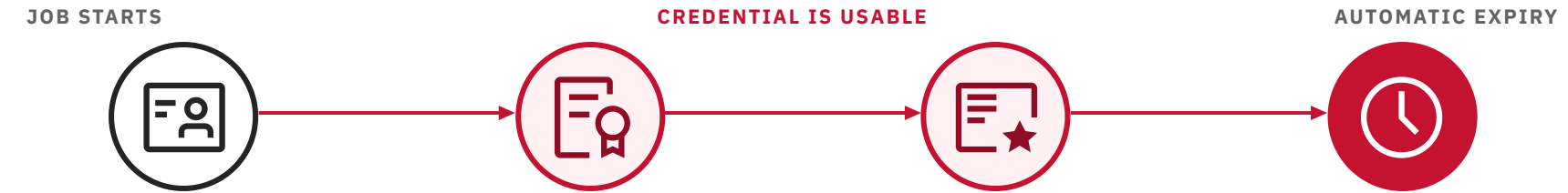
LIFETIME
Time available

→

BLAST RADIUS
Potential harm

Least privilege: allow only the access the job requires.

A platform can issue a credential for each run



Prove identity

The platform identifies the running job

Check

The issuer **verifies** the job and its allowed role

Issue

The issuer returns a **temporary** credential

Expire

The credential **stops working** automatically

No person has to create a new credential for each run.

Scope limits reach; lifetime limits time

SCOPE ↓ · LIFETIME →	LONG LIFETIME	SHORT LIFETIME
BROAD SCOPE	*Persistent, widespread access*	**A · Temporary administrator token** Every Tiny Notes system · 10 minutes
NARROW SCOPE	**B · Read-only token for one repository** One repository · no expiration	**Design target** Narrow scope · short-lived

OIDC lets the cloud verify a job's identity

OpenID Connect provides a standard way to convey identity using a signed token:

1. **GitHub issues an identity token** describing the running job.
2. **The cloud checks the token's signature** and whether the job meets its access rules.
3. **If the checks pass, the cloud issues a separate, temporary credential.**

The cloud decides who gets a role and what it permits

- A **role** is a named set of permissions
- The cloud's **Identity and Access Management (IAM)** policies control who can obtain it and what it allows

Cloud Policy	Example
Trust policy: who may obtain this role?	Accept verified job identity from the Tiny Notes repository in the approved production release context.
Permission policy: what may the role do?	Read the approved release artifact and update the production API service.

GitHub Actions can deploy without a stored cloud key

The Tiny Notes deployment job exchanges its OIDC identity token for temporary cloud access.



Actions job

The approved Tiny Notes workflow runs

Identity proof

GitHub **signs** details about this job

Cloud check

The cloud **matches** the expected repository and environment

Temporary access

The cloud issues the deployment credential

GitHub Docs | Configure OpenID Connect for a cloud provider

⚠ A compromised job can request fresh credentials

Expiration limits use of a stolen copy. An attacker still running code in the job may obtain another credential.



Protect the job

Require review for changes to deployment code and workflow

Check its identity

Accept verified proof only for the expected repository and release context

Limit issued access

Allow only the resources and actions needed to deploy

**Protect the job and who can obtain credentials.
Narrow permissions limit what a compromised job can do.**

⚠ Secret settings control delivery, not what code does afterward

BEFORE INJECTION

Platform controls delivery

- Repository or environment scope selects eligible jobs
- A review gate can delay access

SECRET INJECTED →

AFTER INJECTION

Job code controls use

- The process can read the credential
- The process can send it anywhere the credential is accepted

This job tests changes and can deploy a release

```yaml
on: [pull_request, workflow_dispatch]
permissions:
  contents: read
  id-token: write
jobs:
  test-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: example/cloud-login@v1
        with: {role: deployer}
      - run: npm ci
      - run: npm test
      - run: ./ci/upload-coverage.sh
      - if: github.event_name == 'workflow_dispatch'
        run: ./ci/deploy.sh production/api
```

1. **Run for a pull request or a manual trigger (workflow_dispatch).**
2. **Read repository contents and request an OIDC identity token.**
3. **Exchange the identity token for temporary cloud credentials for the deployer role, available to later steps.**
4. **Install dependencies and run tests. upload-coverage.sh downloads and runs a third-party coverage uploader.**
5. **Only the deployment step requires a manual trigger.**

Where does this job expose deployment access?

```yaml
on: [pull_request, workflow_dispatch]
permissions:
  contents: read
  id-token: write
jobs:
  test-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: example/cloud-login@v1
        with: {role: deployer}
      - run: npm ci
      - run: npm test
      - run: ./ci/upload-coverage.sh
      - if: github.event_name == 'workflow_dispatch'
        run: ./ci/deploy.sh production/api
```

Deployer role: Accepts Tiny Notes repository jobs; grants 10-minute access to read artifacts and update services across Tiny Notes.

Must support: Deploy an approved release to `production/api`.

In pairs:

1. Which steps have cloud access? Which need it?
2. Which jobs should receive credentials?
3. What resources and actions should be allowed?

An attacker now controls the deployment job

The attacker can run code in the deployment job and obtain its credentials.

- **What do your proposed permissions still prevent?**
- What can the attacker still do to the production API?

Each control limits a different part of the attack

Change	What it limits
Separate tests and coverage upload from deployment	Which code can obtain deployment access
Allow only the approved production deployment job to obtain credentials	Which jobs can obtain deployment access
Read approved artifacts; update only the production API	What a compromised deployment job can reach

A compromised deployment job can misuse permission to update the production API.

Secret Storage and Delivery

How should a workload receive a secret?

```
// db.js — TODO: remove before committing
const db = new Client({
  host: 'db.internal',
  user: 'tiny_notes',
  password: 'DEMO_ONLY_tn-db-7mQ!',
})
```

⚠️ Repositories can expose secrets through more than source files

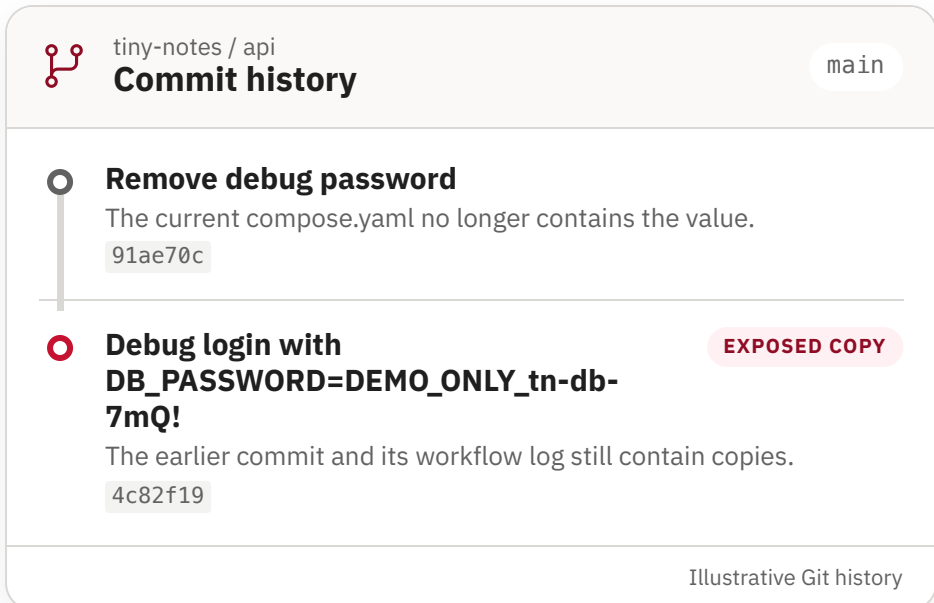
```
# compose.yaml
+ DB_PASSWORD=DEMO_ONLY_tn-db-7mQ!
```

```
tiny-notes — zsh

$ git commit -am "debug login with DB_PASSWORD=DEMO_ONLY_tn-db-7mQ!"
[main 4c82f19] debug login with DB_PASSWORD=DEMO_ONLY_tn-db-7mQ!

$ gh run view 8421 --log
workflow debug: DB_PASSWORD=DEMO_ONLY_tn-db-7mQ!
```

A cleanup commit does not erase the exposing commit



Copies remain
History, clones, forks, and stored workflow logs may still hold the value.

Respond first!
Disable or replace the credential before considering history cleanup.

Better: Ignore `.env` before Git can copy it

1. WRITE THE HOST FILE

```
# .env — plaintext on this host
DB_PASSWORD=DEMO_ONLY_tn-db-7mQ!
```

2. IGNORE IT AND RESTRICT ACCESS

```
tiny-notes — zsh

$ echo .env >> .gitignore
$ chmod 600 .env
$ git status --short --ignored
!! .env
```

`!!` means that Git is ignoring the file.

3. INJECT IT AT STARTUP

```
# compose.yaml
environment:
  DB_PASSWORD: ${DB_PASSWORD}
```

⚠ Runtime injection still creates plaintext copies



Host .env
Plaintext source on the host

Compose
Interpolates the value

Process
Receives a **plaintext** environment variable

Connection
The database client uses it

Child processes, diagnostics, and crash reports may create more copies.

Secret storage must outlive an application instance

- A deployment can replace an old container with a new one
- A failed server can be replaced by another server
- The new instance still needs access to the secret



SECRET STORAGE

Keeps the secret

Keeps the current credential after Tiny Notes is replaced.



TINY NOTES

Uses a copy

Retrieves the current credential each time it starts.

Durable secret storage follows two common patterns



VERSIONED WITH CONFIGURATION

Encrypted file

SOPS stores ciphertext in Git; age or a cloud KMS controls who can decrypt it.



CENTRALIZED BEHIND AN API

Managed secret store

Google Secret Manager, AWS Secrets Manager, Azure Key Vault, or Vault controls retrieval and lifecycle.

Both patterns protect a durable source.
Decryption or retrieval creates plaintext elsewhere.

Encrypted configuration still needs protected key access

```
tiny-notes — zsh

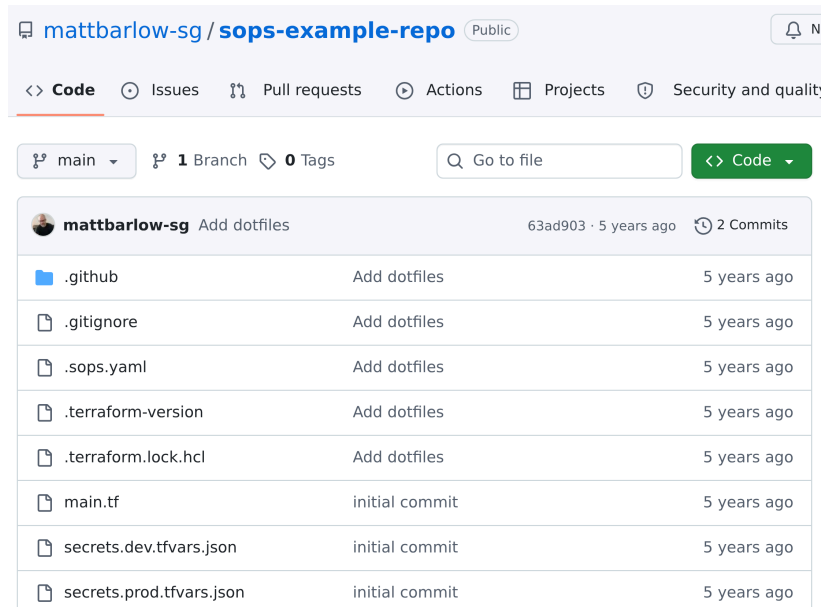
$ SOPS_AGE_KEY_FILE=~/.config/sops/age/keys.txt
  sops decrypt secrets.yaml
database:
  host: db.internal
  password: DEMO_ONLY_tn-db-7mQ!
```

age key file
A private key stored on the developer machine or deployment host. **Keep it out of Git.**

Cloud KMS
A cloud service stores the decryption key and checks which users or workloads may use it.

Use SOPS when configuration should travel with a release in Git.
Always keep the decryption key outside Git!

Explore encrypted configuration in Git



github.com/mattbarlow-sg/sops-example-repo ↗

A managed secret store controls retrieval

DURABLE SOURCE

Managed secret store

Stores the current secret and returns it through an API.

tiny-notes.js

```
const result = await secrets.send(
  new GetSecretValueCommand({
    SecretId: "tiny-notes/database",
  }),
)
```

Limit reads

Permissions can give Tiny Notes access to one named secret.

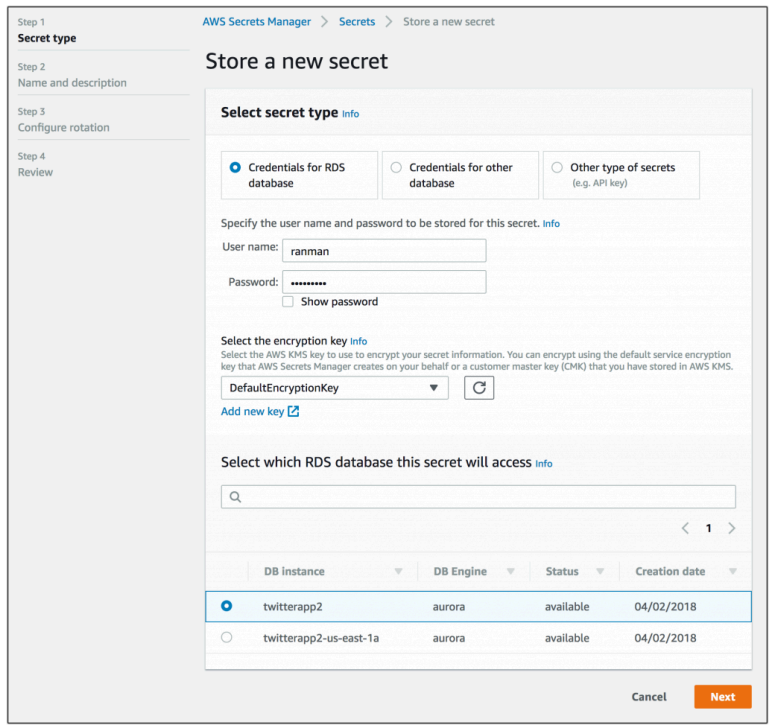
Record requests

The service logs which identity requested a value.

Keep versions

The team can add a replacement value and control when workloads use it.

Store a database credential

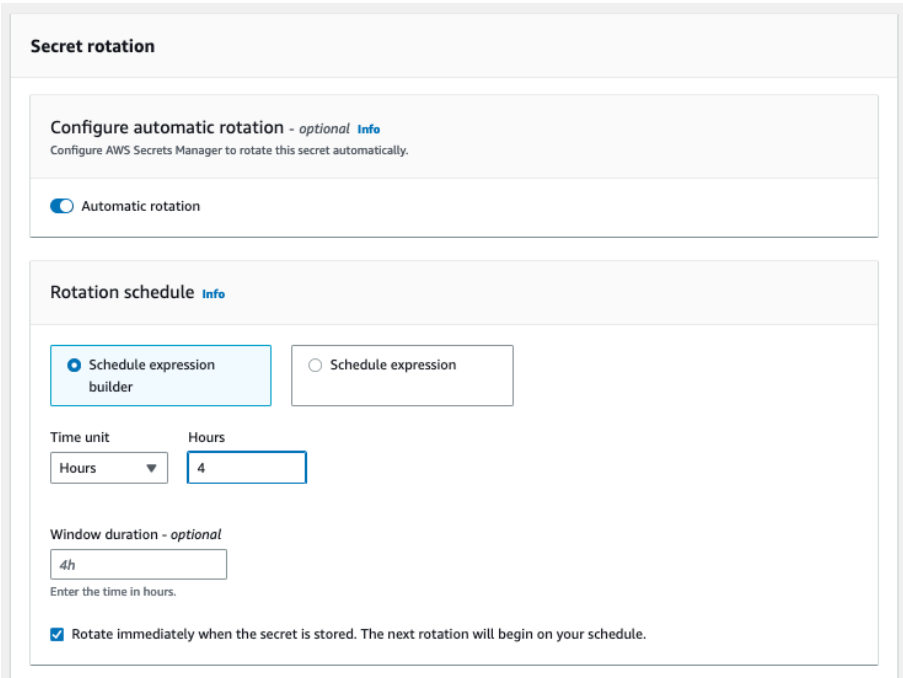


- Enter the database username and password.
- Choose the key that encrypts the stored value.
- Select the database these credentials access.

AWS console example, 2018.

Source: AWS walkthrough ↗

Configure when the stored credential rotates



- Enable automatic rotation.
- Set its frequency and allowed time window.
- The application must still adopt the new value.

AWS console example, 2022.

Source: AWS rotation walkthrough ↗

Retrieval passes control to the workload

MANAGED STORE

Store controls

- Verify which workload is asking
- Check permission for this secret
- Record the request and selected version

API RETURNS SECRET

WORKLOAD AND DELIVERY PATH

Workload controls

- Plaintext in memory, files, or environment variables
- Extra copies in logs, debuggers, or crash reports
- Use of old copies until rotation or revocation

The first request needs a credential from outside the secret manager

The team can provision another shared secret, or the platform can supply short-lived proof of the workload's identity.

Workload identity removes the bootstrap secret

A **workload identity** is an identity assigned to an application or service. The platform supplies credentials that let it prove that identity.

- No stored access key is needed for the first request to the secret manager.
- The retrieved database password still needs rotation.



EXAMPLE

Tiny Notes

Running application

↓

Platform-issued temporary credentials

↓

Secret manager

Checks identity and permission

Retrieval still ends at an application interface

API

DIRECT RETRIEVAL

Application

The application fetches the secret and holds it in memory.

MOUNTED FILE

Helper

A helper retrieves the secret and writes it to a local file.

PLATFORM SECRET

Controller

A controller copies the value into the platform's secret object.

When a secret changes, the delivery path must update the copy and the application must reload it.

Runtime constraints shape secret delivery

Deployment constraint	Likely fit and accepted tradeoff
Platform supports workload identity	Request short-lived credentials; depend on platform identity and policy.
Older application accepts only a file or environment variable	Use a compatible adapter; accept a plaintext runtime copy.
Workload must operate offline	Use a durable encrypted file; protect and deliver the separate decryption key.
Team needs centralized access records and rotation	Use a managed store; accept that retrieved copies remain outside the store.
Simple local development	Use a protected, ignored `.env`; accept host plaintext and manual rotation.

The password changed. What must the app do?

APPLICATION STARTUP

```
const db = new Client({
  host: 'db.internal',
  user: 'tiny_notes',
  password: process.env.DB_PASSWORD,
})
await db.connect()
```

The deployment supplies `DB_PASSWORD` from the secret manager. This app reads it only at startup.

The password changes in the secret manager. **What must happen before this app uses it?**

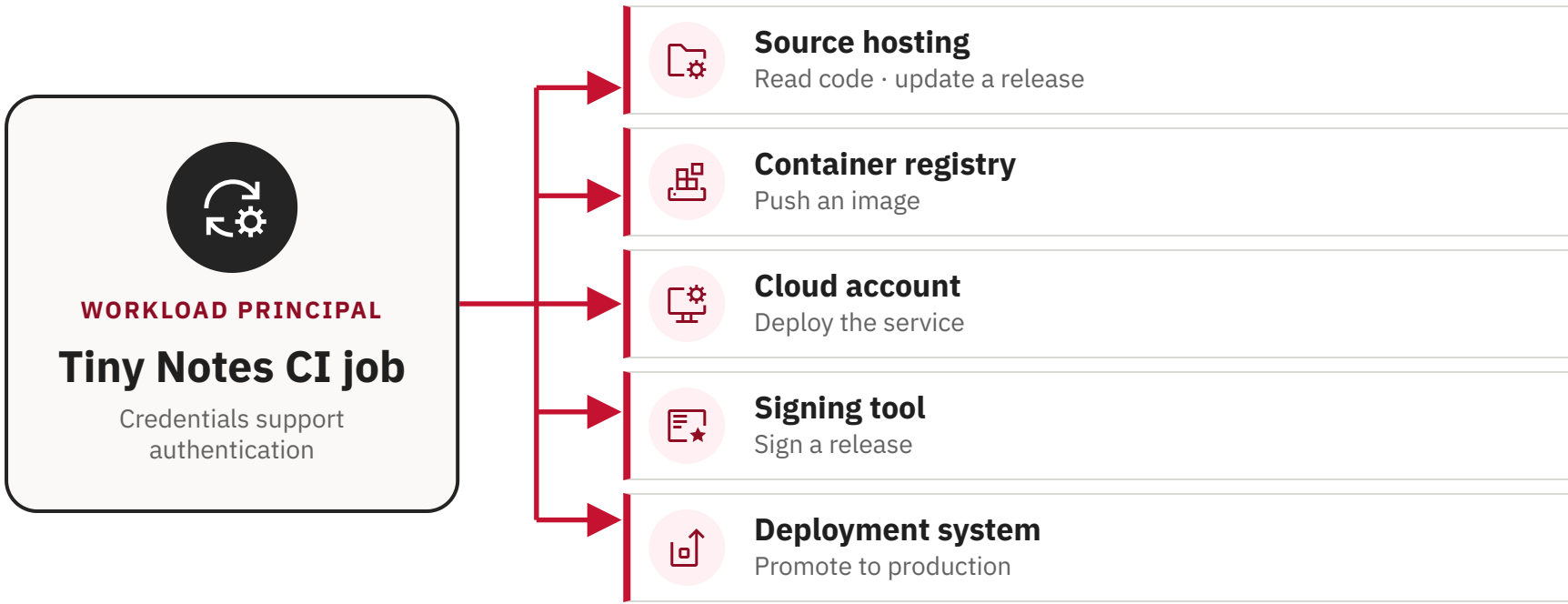
Update the delivered value, then restart or replace the app with the new environment.

How would restarting affect availability?

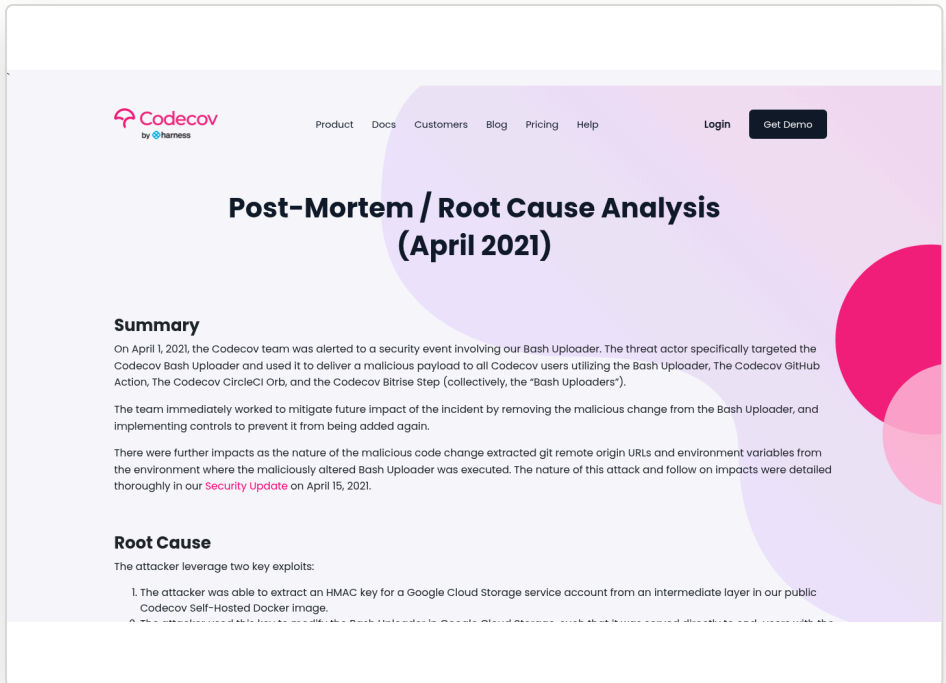
CI/CD Credentials

Why do pipeline secrets spread beyond one system?

🚨 One pipeline principal can reach several important systems



An exposed key let attackers replace Codecov's uploader



Codecov · April 2021 postmortem

1. A public Docker image layer exposed a Google Cloud Storage account key.
2. The key let the attacker replace the Bash Uploader stored by Codecov.
3. Customer CI jobs downloaded and ran the altered uploader.

Read the Codecov postmortem

The altered uploader copied each job's environment variables

Example job environment

```
CI_TOKEN=••••
CLOUD_KEY=••••
PACKAGE_TOKEN=••••
```



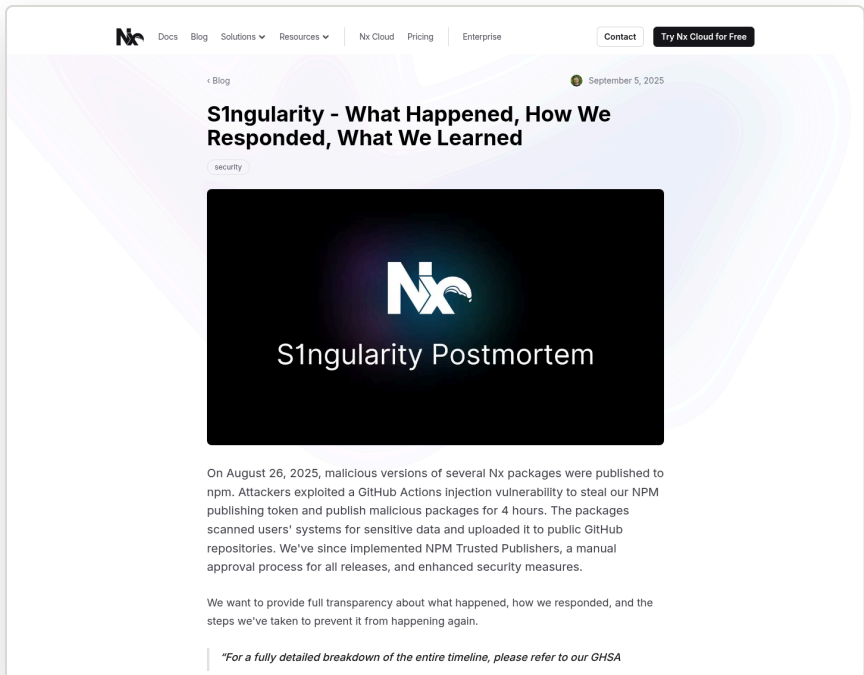
copied out

Attacker-controlled server

The malicious code extracted environment variables and Git remote information from each environment where it ran.

A secret available to the uploader could leave with the uploader.

Untrusted PR data opened Nx's publishing path



Nx · S1ngularity postmortem · September 5, 2025

1. A privileged workflow inserted an untrusted PR title into a shell command.
2. Command injection gained a write-capable repository token, letting attackers change a CI script and trigger the publish workflow.
3. The script stole the npm publishing token; attackers used it to publish malicious packages outside the release pipeline.

Read the Nx postmortem

Finished jobs can leave usable credential copies behind

Copy channel	What can persist
Logs	A command prints a credential
Artifacts or test reports	A collected file contains the value
Caches	A credential-bearing file is restored later
Container layers	An earlier build layer keeps injected data
Forks	A committed secret remains in copied history
Downstream systems	An uploader or API receives another copy

The job can end while a copied credential still works.

Code is trusted only after the exact commit passes required review and protected-branch controls.

A pull request remains untrusted while its author can still change what a job executes.

Better: Run PR checks without publishing access

```yaml
jobs:
  pull-request-checks:
    permissions:
      contents: read
    steps:
      - uses: actions/checkout@v4
      - run: npm ci && npm test
      - uses: vendor/report@a4f3e1d2c5b6a7980e1f2a3b4c5d6e7f8091a2b3

  ...
```

1. The job's repository token has read-only access.
2. PR code runs here without a publishing credential.
3. A full commit SHA fixes which version of the action runs.

**No release environment. No publishing credential.
Read-only repository access.**

Only the protected publish job receives the credential

```yaml
jobs:
  ...
  publish:
    needs: pull-request-checks
    if: >-
      github.event_name == 'push' &&
      github.ref == 'refs/heads/main'
    environment: release
    permissions:
      contents: read
    steps:
      - uses: actions/checkout@v4
      - run: npm ci
      - run: npm publish
        env:
          PUBLISH_CREDENTIAL: ${{ secrets.PUBLISH_CREDENTIAL }}
```

1. **Checks must pass in this workflow run.**
2. **Publish only on a push to main.**
3. **The release environment requires approval and stores the publishing secret.**
4. **This step receives the publishing credential.**

Beware: Earlier code in this job can alter the publish step. Injecting the secret into one step does not isolate it from a compromised job.

A credential must change. Can we replace it everywhere without breaking production?

Secret Lifecycles and Inventories

How do you rotate a secret completely and safely?

Rotation replaces a credential and retires the old one

Routine rotation

- Limits the credential's lifetime
- Tests the replacement process before an incident
- **replace → migrate → verify → retire**

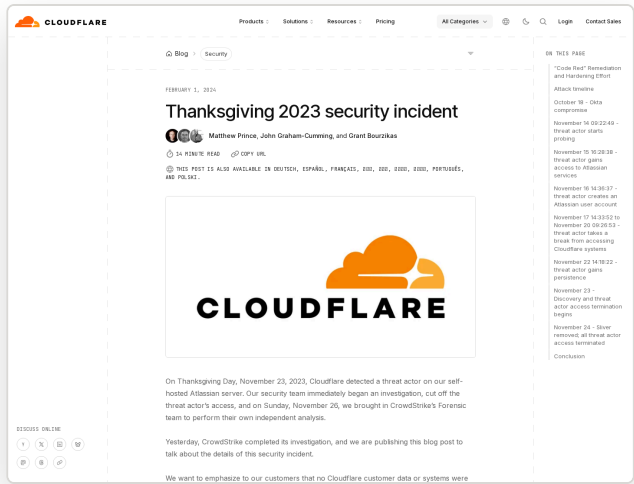
Emergency rotation

- Contains a suspected exposure
- May revoke first, disrupting consumers until they migrate

Emergency: **contain exposure first**, then restore and verify consumers.

Four credentials believed unused still granted access

1. **October 2023:** An Okta compromise exposed Cloudflare credentials.
2. **Rotation:** Thousands were rotated, but one service token and three service accounts stayed active because they were believed to be unused.
3. **November 2023:** An attacker used them to access Atlassian services and source code management.



Read Cloudflare's incident report

A credential remains a risk until its revocation is verified.

You cannot rotate credentials you cannot list

Inventory field	Question it answers
Owner	Who coordinates the change and verifies it?
Consumers	Which services and jobs use it to authenticate?
Copies	Where is the value stored or delivered?
Permissions	What can its principal do?
Expiration	When should the credential stop working?

A written inventory helps the team find dependencies that memory may miss.

Record which services and jobs use each secret

AWS Secrets Manager



- **Tags:** record owner and service metadata.
- **CloudTrail:** records Secrets Manager API activity.

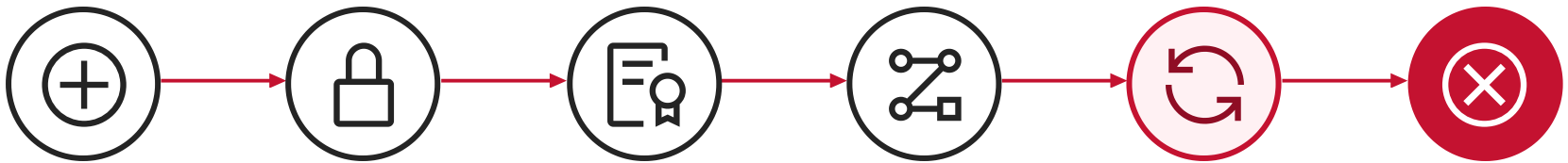
Tags · API logging

HashiCorp Vault

- **Custom metadata:** record inventory fields for KV v2 secrets.
- **Audit logs:** record Vault requests and responses when enabled.

Metadata · Audit logging

Secret management covers the whole life of a credential



Create
Issue the credential

Store
Protect its source

Control access
Limit who can retrieve it

Deliver
Reach the intended consumer

Rotate
Replace the value

Revoke
Reject the old credential

Track the credential through every stage: who owns it, who receives it, where copies remain, and when the old value is rejected.

Routine rotation moves consumers before retiring old access

	1 · Create	2 · Update	3 · Verify	4 · Disable
Replacement	Issued	Used by some	Used by every consumer	Active
Old	Accepted	Still accepted	No consumer uses it	Rejected

An exposed credential may need immediate revocation, even if service is disrupted.

⚠ Some credentials can create or reveal others

Signing key



What it can do
Create new signed tokens.

Response
Make validators reject tokens signed by the compromised key.

Integration token



What it can do
Reveal records containing other credentials.

Response
Find and rotate every exposed credential.

Replacing the first credential may leave created tokens or exposed credentials usable.

Rotation is complete only when the old credential no longer works

The replacement works

- Every inventoried consumer succeeds with it
- Logs show consumers have moved to it

The old credential fails

- Direct checks show the accepting service rejects it
- Logs show no accepted use after retirement

For Tiny Notes:

Update stored password → Reload every application instance → **Disable old password in database**

Credential Leak Diagnosis and Response

Which controls matter before, during, and after a leak?

Preparation shortens the dangerous window

BEFORE EXPOSURE



LEAK DISCOVERED



OLD ACCESS BLOCKED



Prepare

Narrow scope · short lifetime · inventory

Contain

Alerts and rehearsal **speed revocation**

Verify

Complete rotation · checks · monitoring

Prevention lowers the chance of a leak; preparation shortens how long it remains useful.

Secret scanners find possible credentials—but not all

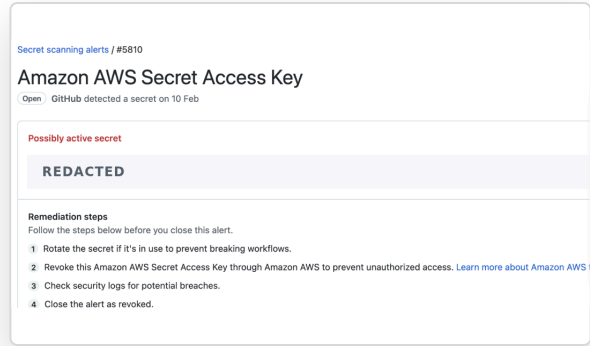
Secret scanning asks: was a credential exposed?

Tools such as Gitleaks or TruffleHog look for token patterns or unusually random strings. They can miss secrets or flag harmless text.

Access monitoring asks: how was it used?

Service logs can show requests and actions made with the credential. Missing log evidence does not prove it was unused.

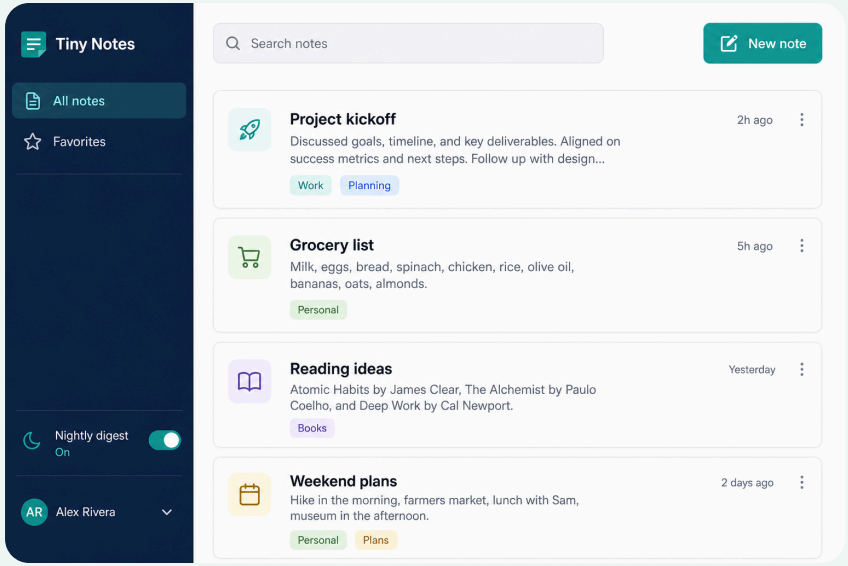
If a valid credential was public, treat it as copied—even without confirmed use.




```
$ gitleaks git --config .gitleaks.toml --verbose …
Finding: AWS_ACCESS_KEY_ID=REDACTED
RuleID: aws-access-token
File: notes/retired-debug.env
Commit: 0d45b82a…

INF 3 commits scanned
WRN leaks found: 2
```

Can Tiny Notes declare the incident contained?



After a morning spent responding to a leaked deployment token, a teammate asks:

> **“Can we close this incident and get back to the release?**

Go through the **Tiny Notes incident evidence handout** in a small group.

Would your checks catch this regression?

- A teammate suggests ignoring the `tests/` **directory** to silence the fixture finding
- A later commit adds a real deployment credential **beside that fixture**
- **Revise your checks:** reject the new credential while accepting the harmless fixture
- Would your original test pair have caught the broad exception?

Working production is not proof of containment

Decision	Evidence needed
Revoke the old token	Deployment service rejects it; rollback disruption is addressed
Update deployment and rollback	Both use the replacement; missing token stops the script
Check the scanner	Real credential beside the harmless fixture is blocked; fixture is allowed

Preserve: commit IDs, exposure times, alerts, and available logs.

Still unknown: whether someone used the token before 11:30.

🚀 Today, we learned how to ...

- Explain why secrets need lifecycle controls that ordinary settings do not
- Distinguish principals, credentials, authentication, authorization, and permissions
- Explain how credential scope and lifetime limit blast radius after a leak
- Compare different storage and delivery methods and their tradeoffs
- Diagnose a credential leak and recommend controls before, during, and after exposure, including complete rotation