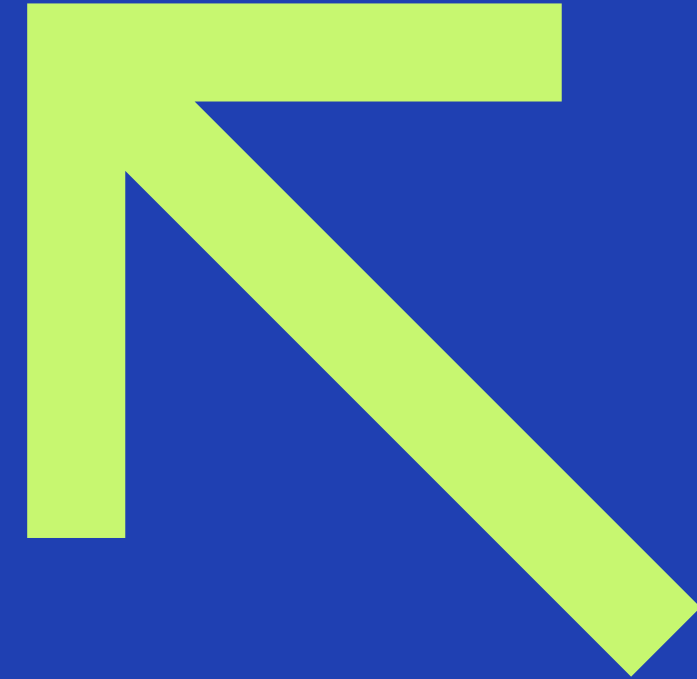


One Committed Secret and the GitOps Framework That Changed my Workflow



MATT PARKER

Senior IT Engineer
PrizePicks
github.com/maparker/shikomi



AGENDA

1. The story. One committed secret, and the pipeline it built.
2. The mental model. What GitOps actually means for a Jamf script.
3. The guardrails. Two gates, deconstructed — from a bad script to a blocked commit to a shipped fix.
4. The tool. Shikomi, from an empty file to a deployed script.
5. Getting there from here. Bridging to Jamf Pro, and where to start Monday.



The Story

One committed secret, and the pipeline it built.



A Tuesday Morning, Like any Other

I was writing a Jamf script that needed to talk to a third-party API.

I hardcoded the API token. Just for testing.
I'd swap it out before committing.

I forgot.

Three Days Later

A security scanner flagged the pattern in our repository.

By the time anyone told me, the credential was already rotated.

I didn't catch it. Tooling did, after the fact.

Why Didn't Anything Stop Me?

The token was obvious.

The pattern was a known one.

The tooling existed.

IT JUST WASN'T WIRED UP FOR SHELL SCRIPTS.

Scripts Feel Small. Until They Aren't.

What we treat like real code

APPLICATION REPOS

- ▶ Pull request reviews
- ▶ CI gates
- ▶ Versioned releases
- ▶ Secret scanning

What we treat like scratch work

JAMF SCRIPTS

- ▶ Direct paste into the console
- ▶ "I'll test it on one machine"
- ▶ "Whatever's in Jamf right now"
- ▶ "It's just internal"

A script that runs on every Mac in the fleet is not scratch work.

By the End of This Session

1. Identify the security risks in traditional Jamf Pro script management, including credential exposure, version drift and ungoverned code promotion.
2. Describe how a GitOps pipeline prevents scripting mistakes at two validation gates: local pre-commit hooks and pull request-level CI checks.
3. Evaluate how the Shikomi framework automates secret scanning, linting and versioning so admins can focus on solving problems rather than policing their own workflow.

The Mental Model

What GitOps actually means for a Jamf script.



Two ways to manage a Jamf script

The Manual Way

- ▶ Write it locally (or in the Jamf console)
- ▶ Paste into the script editor
- ▶ Save and hope
- ▶ A colleague edits it next week
- ▶ Nobody knows what changed, when, or why
- ▶ The "current version" lives in Jamf Pro

The Repository Way

- ▶ Write it in your editor
- ▶ Commit to a branch
- ▶ Open a pull request
- ▶ Automated checks run
- ▶ Reviewer approves
- ▶ The repository is the canonical version — Jamf Pro is downstream

SAME SCRIPT. SAME ADMIN. DIFFERENT BLAST RADIUS.

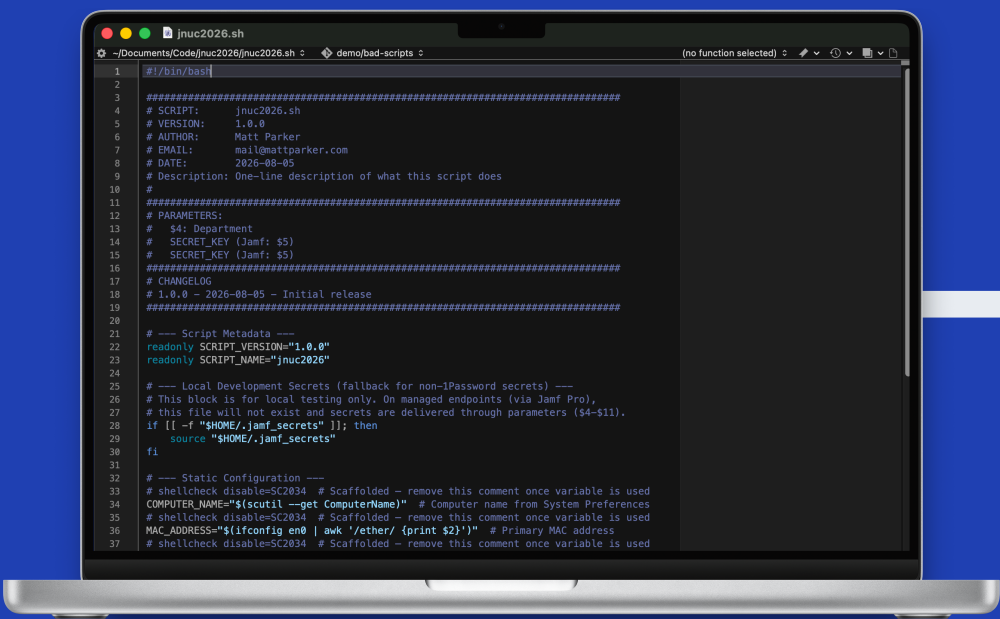
GitOps in One Sentence

The repository is the source of truth. Everything that runs on the fleet got there by being merged.

With GitOps, a repo gives us the opportunity to add checks at two points: the commit and the pull request.

Where Guardrails Live

YOUR LAPTOP



Gate 1

Pre-commit

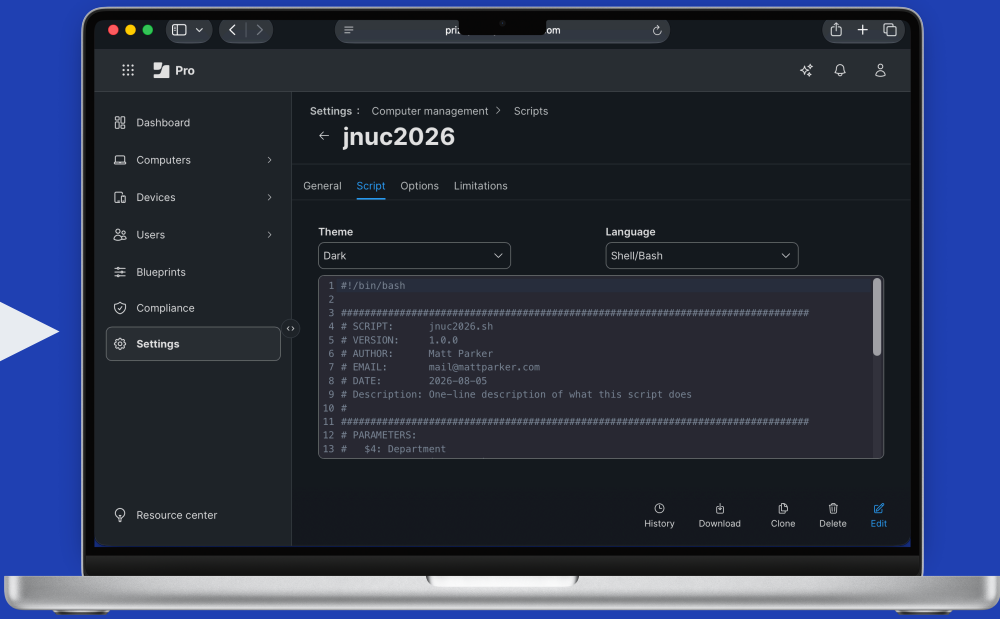
PULL REQUEST



Gate 2

GitHub Actions

JAMF PRO



ANYTHING THAT GETS TO JAMF PRO HAS CLEARED BOTH GATES.

These two gates aren't generic best practice, they map to two specific things. A secret ending up in code. Shell being uniquely easy to get subtly wrong. That's not abstract; that's the opening story, and it's about to happen to the next script you see.

The Guardrails

Two gates, deconstructed: from a bad script to a blocked commit to a shipped fix.

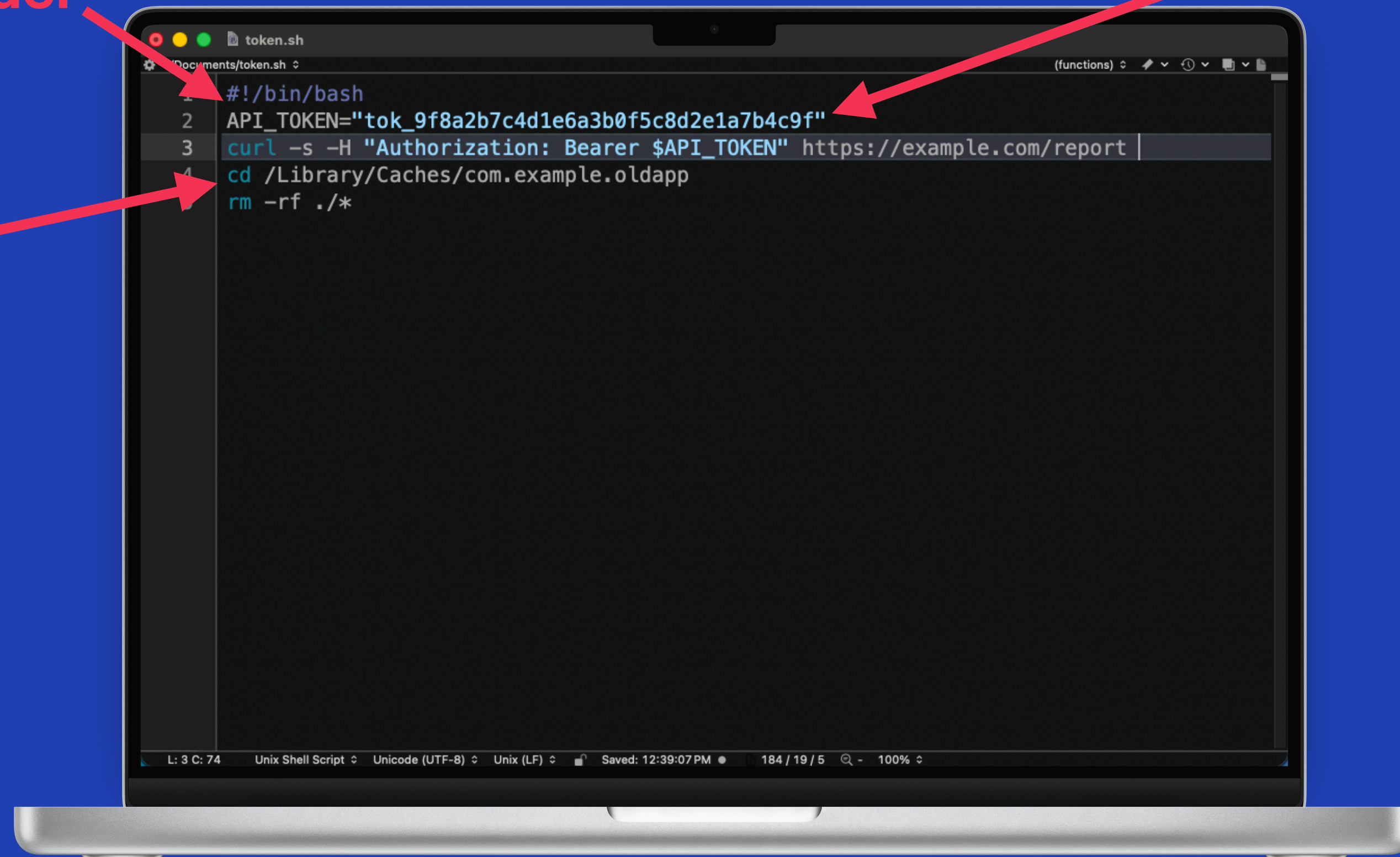


Meet the Specimen

No version header

A hardcoded token

A cd with no failure check.



```
1 #!/bin/bash
2 API_TOKEN="tok_9f8a2b7c4d1e6a3b0f5c8d2e1a7b4c9f"
3 curl -s -H "Authorization: Bearer $API_TOKEN" https://example.com/report
4 cd /Library/Caches/com.example.oldapp
rm -rf ./*
```

This is the script we trace through both gates.

The Secret Never Leaves the Machine

Secret detected

Redacted and
never output

```
~/Documents/Code/jnuc2026 main*
> git commit -m "Add API integration"
Detect hardcoded secrets..... Failed
- hook id: betterleaks
- exit code: 1

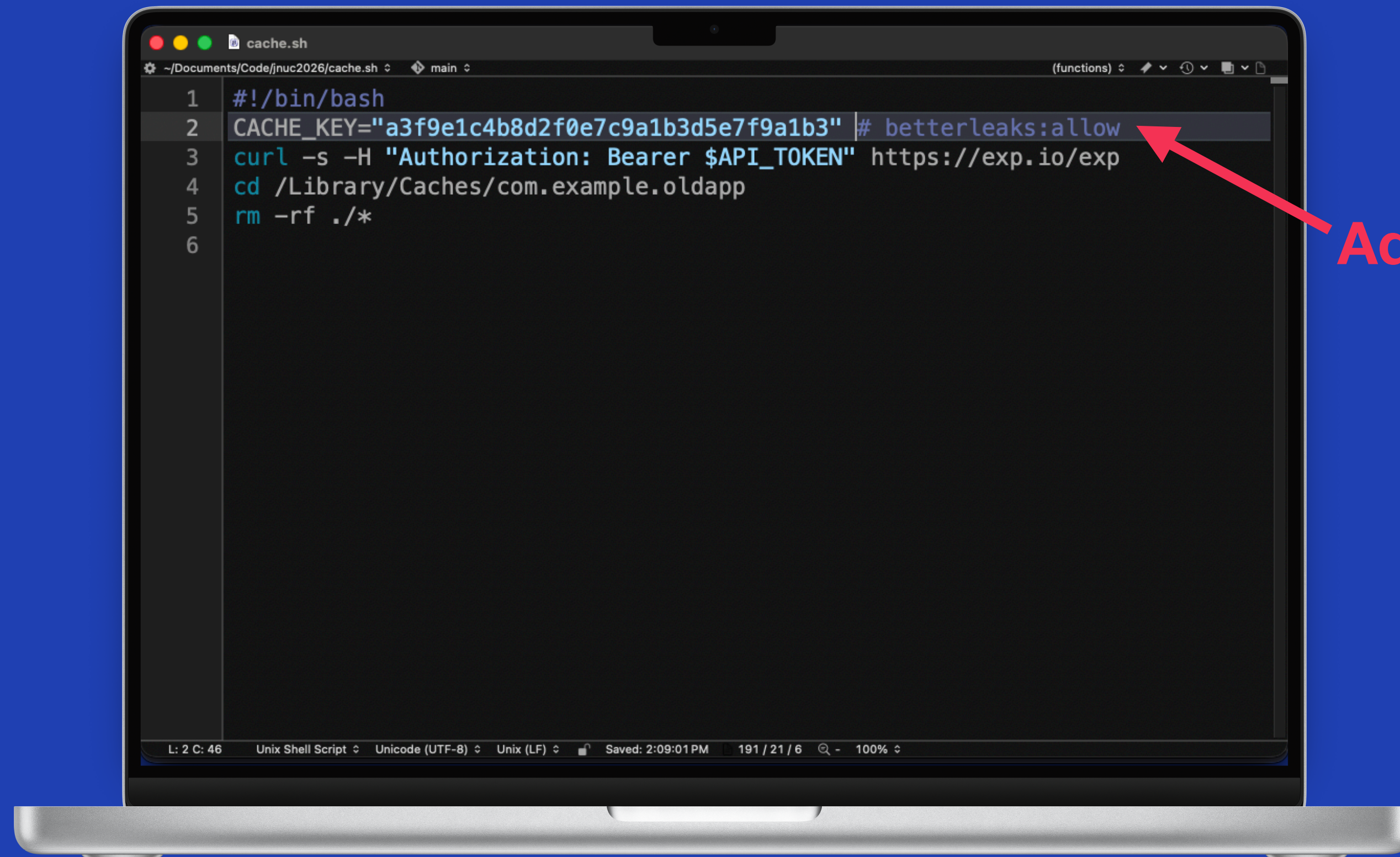
+ o
  ▼
  betterleaks dev

  generic-api-key—o
  2 | API_TOKEN="REDACTED"
    | ^^^^^^^^
  attributes:
    git.message .....
    git.sha .....
    path ..... token.sh
    resource ..... git.patch_content

1:41PM INF scanned ~175 bytes (175 bytes) in 102ms
1:41PM WRN leaks found: 1
```

No commit. No push. No 3 a.m. rotation.

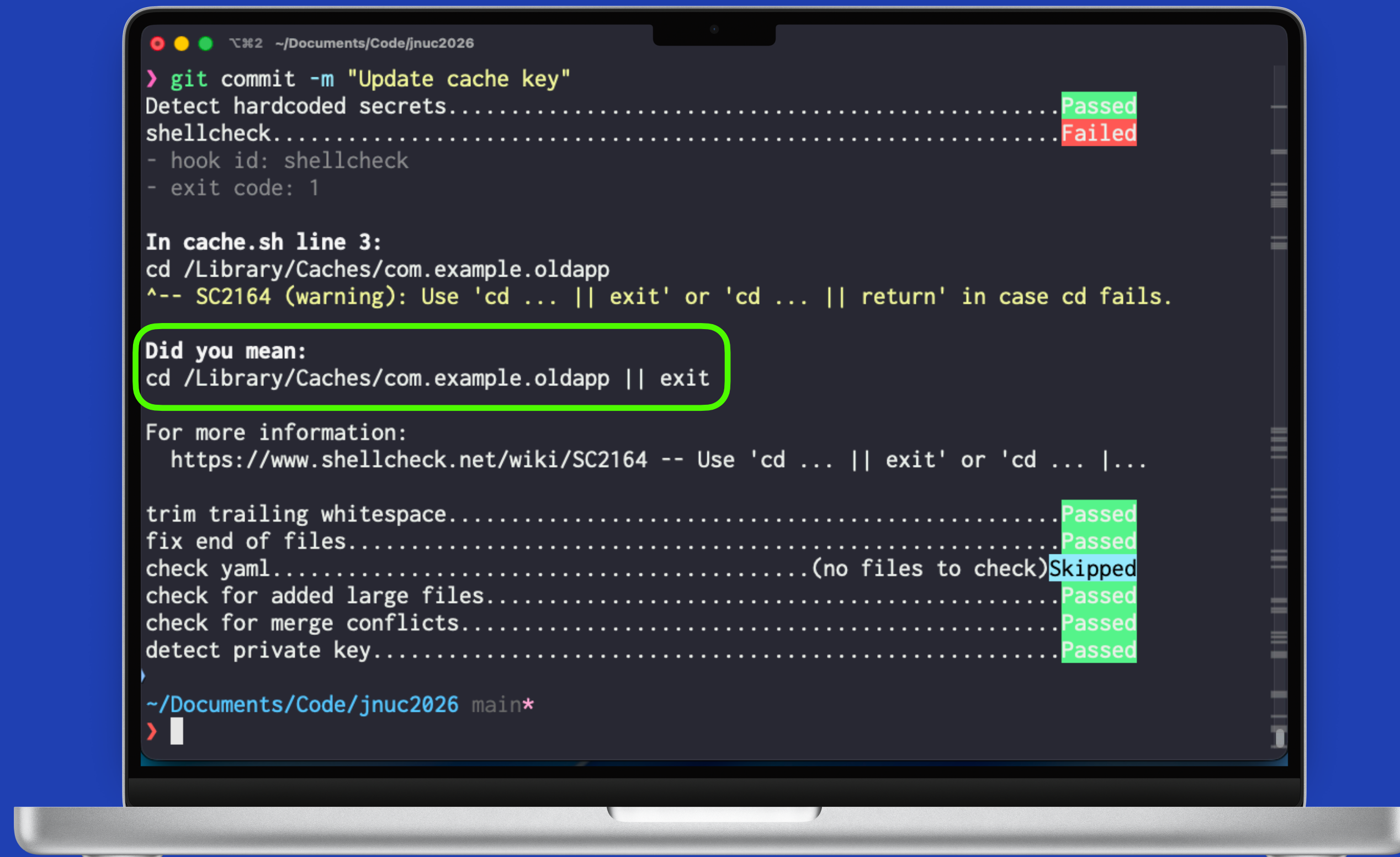
What Happens When It's Not Actually a Secret



Add to Allow List

You allowlist the fingerprint of this one line.
You do not disable the hook.

The Bug Stays on the Machine Too



```
~%2 ~/Documents/Code/jnuc2026
> git commit -m "Update cache key"
Detect hardcoded secrets.....Passed
shellcheck.....Failed
- hook id: shellcheck
- exit code: 1

In cache.sh line 3:
cd /Library/Caches/com.example.oldapp
^-- SC2164 (warning): Use 'cd ... || exit' or 'cd ... || return' in case cd fails.

Did you mean:
cd /Library/Caches/com.example.oldapp || exit

For more information:
https://www.shellcheck.net/wiki/SC2164 -- Use 'cd ... || exit' or 'cd ... ||...'

trim trailing whitespace.....Passed
fix end of files.....Passed
check yaml.....(no files to check)Skipped
check for added large files.....Passed
check for merge conflicts.....Passed
detect private key.....Passed

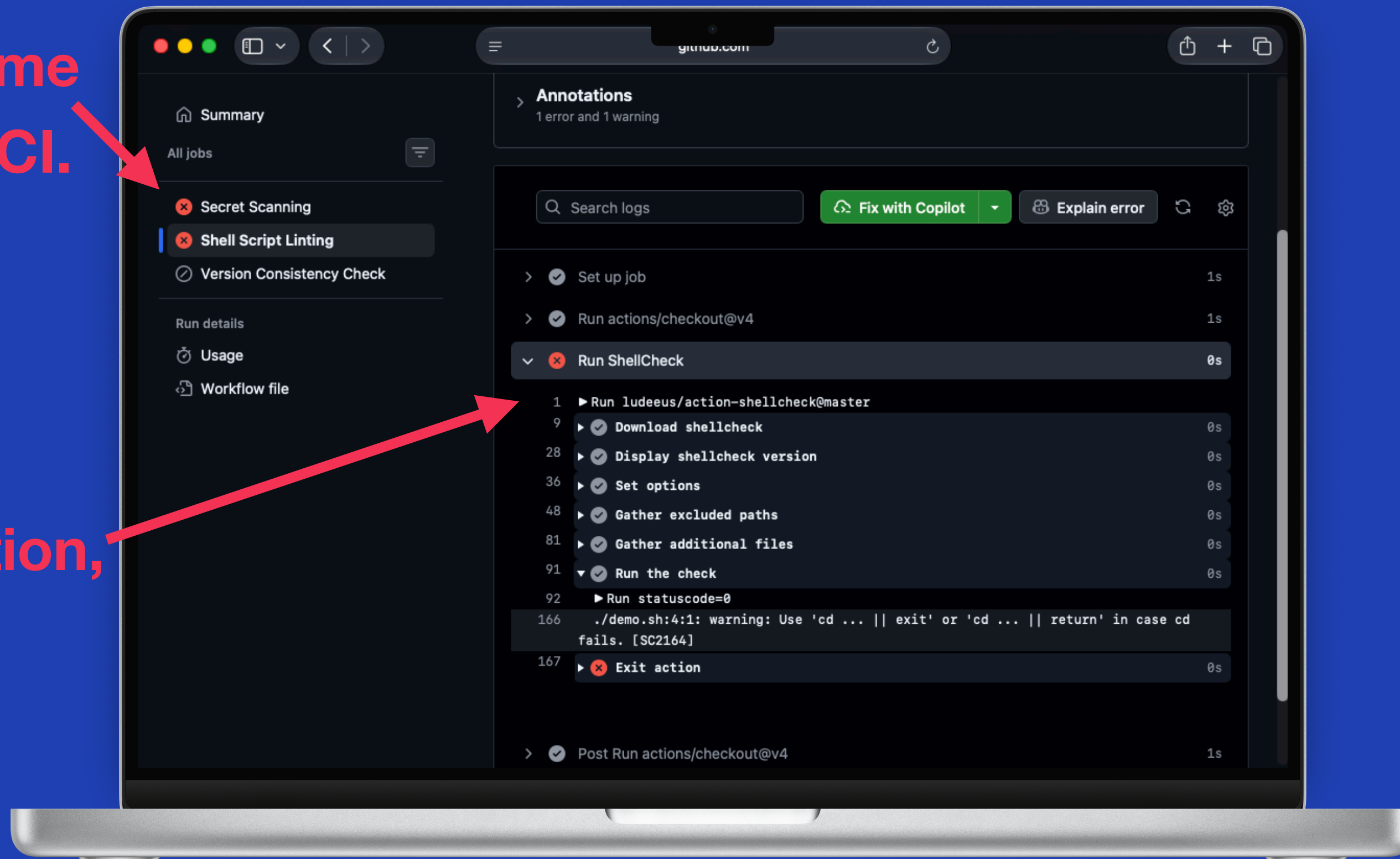
~/Documents/Code/jnuc2026 main*
>
```

ShellCheck doesn't just say "bad."
It tells you the fix.

The Safety Net for Anything That Slips Past Gate 1

Same script, same checks, now in CI.

No inline annotation, just a failed job.

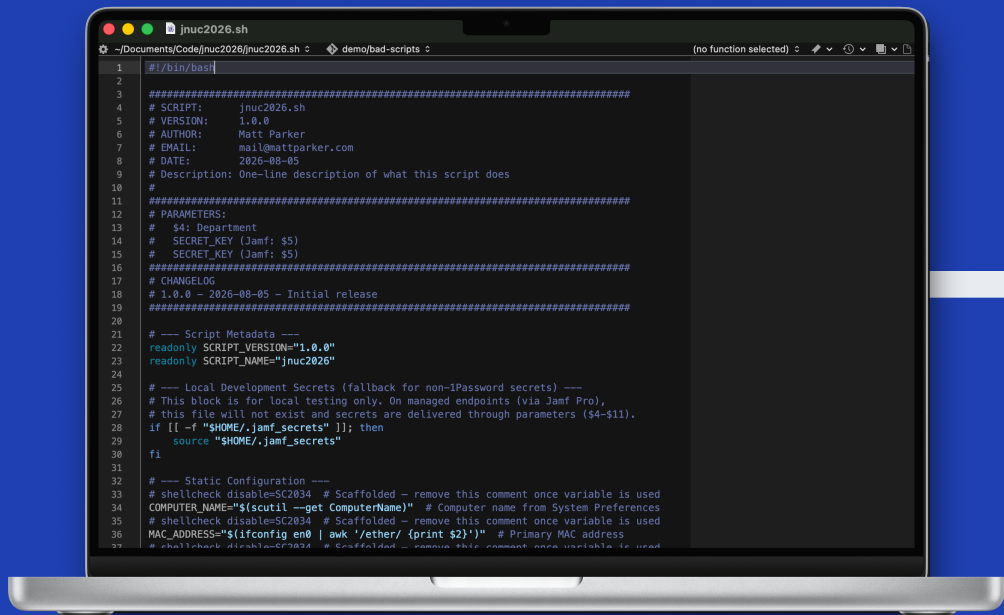


Local hooks are a convenience.

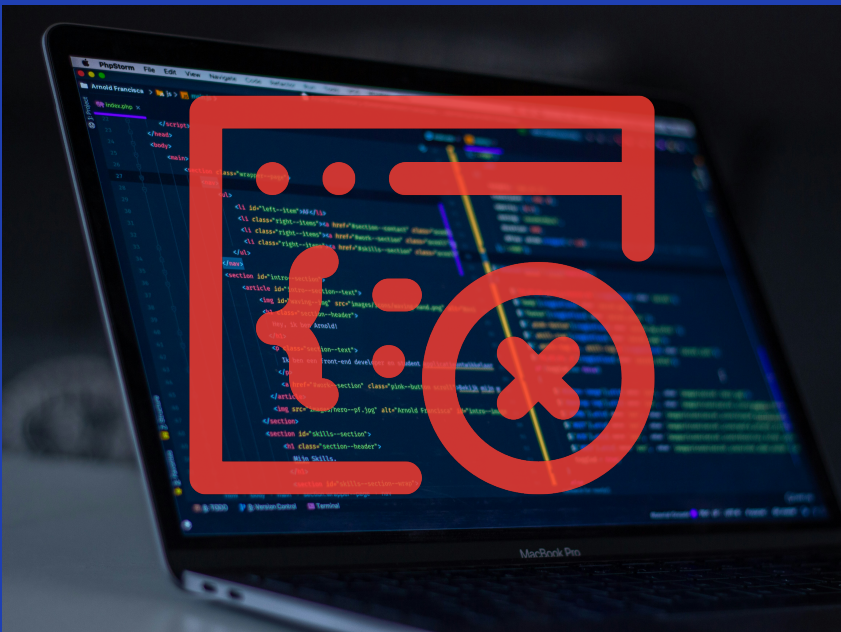
CI is the contract.

Full Circle

WRITTEN LOCALLY



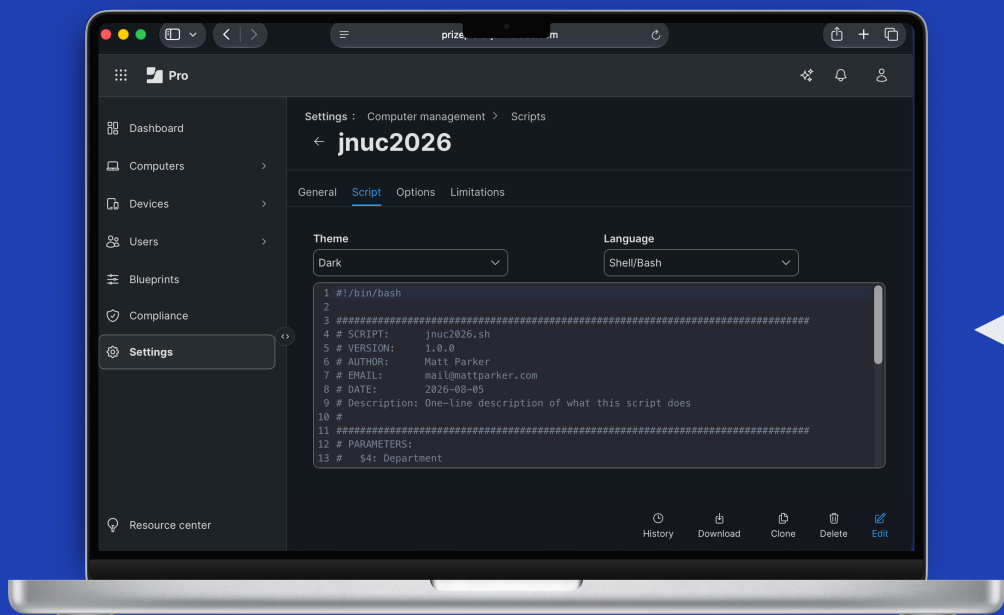
GATE 1: BLOCKED



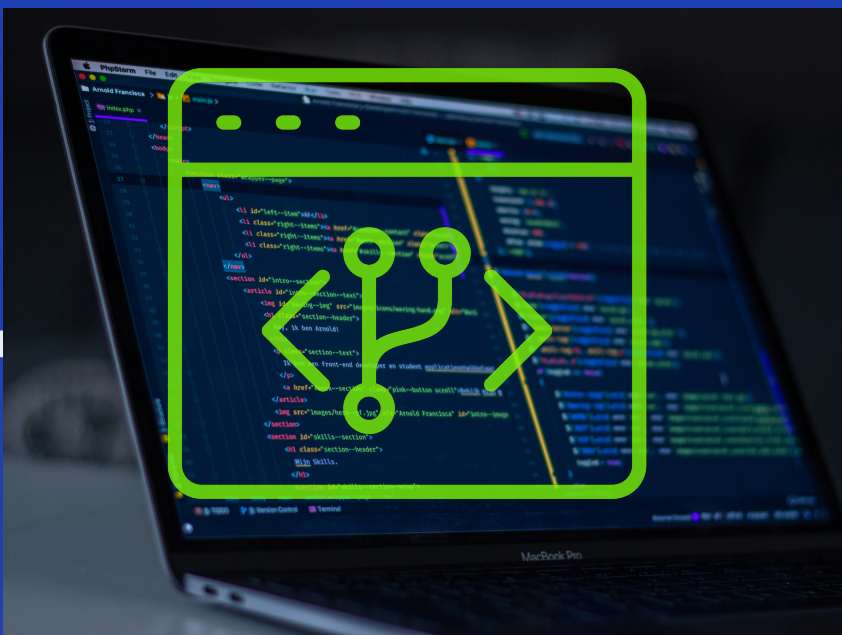
FIXED



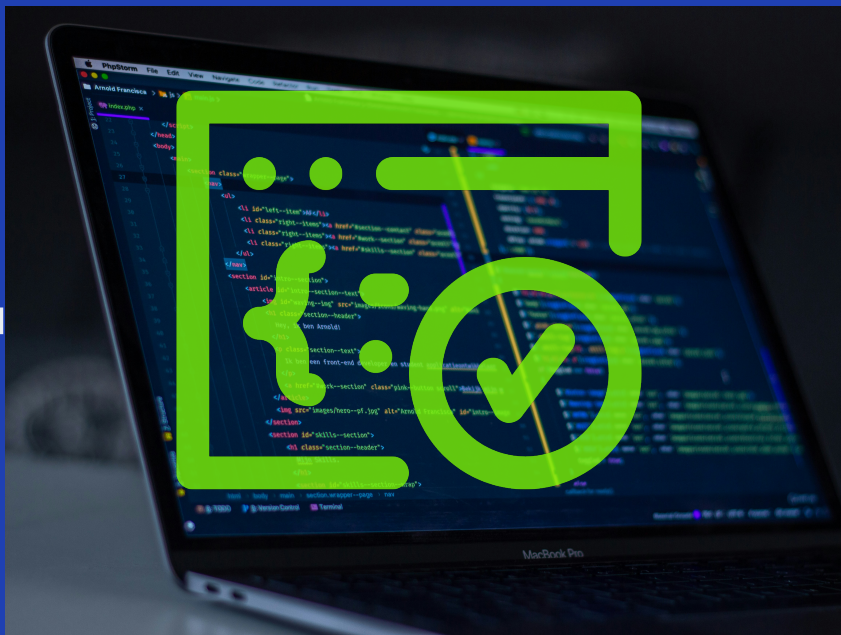
JAMF PRO



MERGED



GATE 2: PASSES



What It Takes To Wire This Up Yourself

```
# .pre-commit-config.yaml
repos:
  - repo: https://github.com/betterleaks/betterleaks
    rev: v1.7.3
    hooks:
      - id: betterleaks
  - repo: local
    hooks:
      - id: shellcheck
        entry: shellcheck
        args: [--severity=warning]
```

Plus a matching GitHub Actions workflow for the same two checks in CI. Pin the right versions. Get the severity flags right. Do it once, then remember to do it again for the next repo.

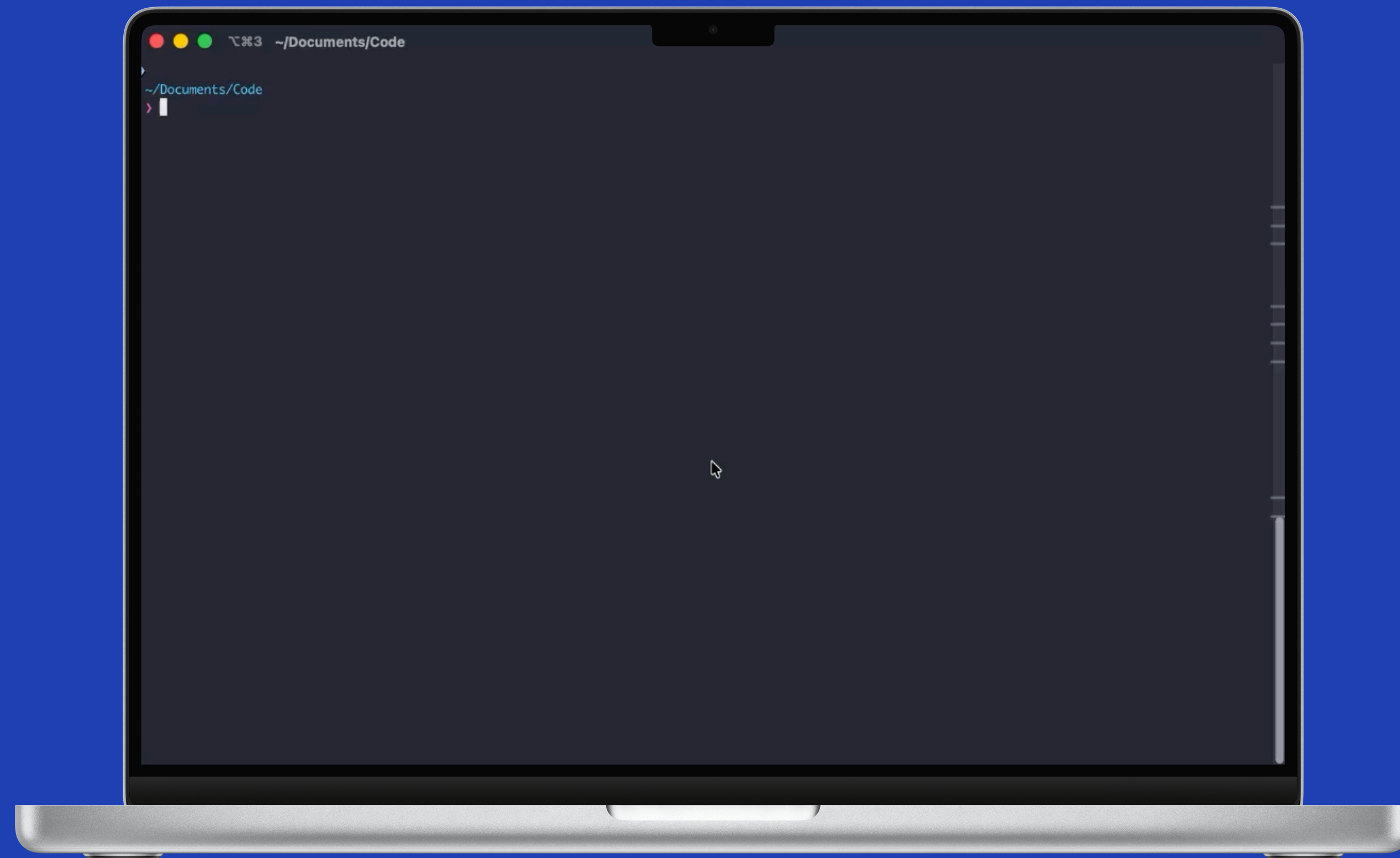
You don't have to write any of this yourself.

Shikomi 仕込み

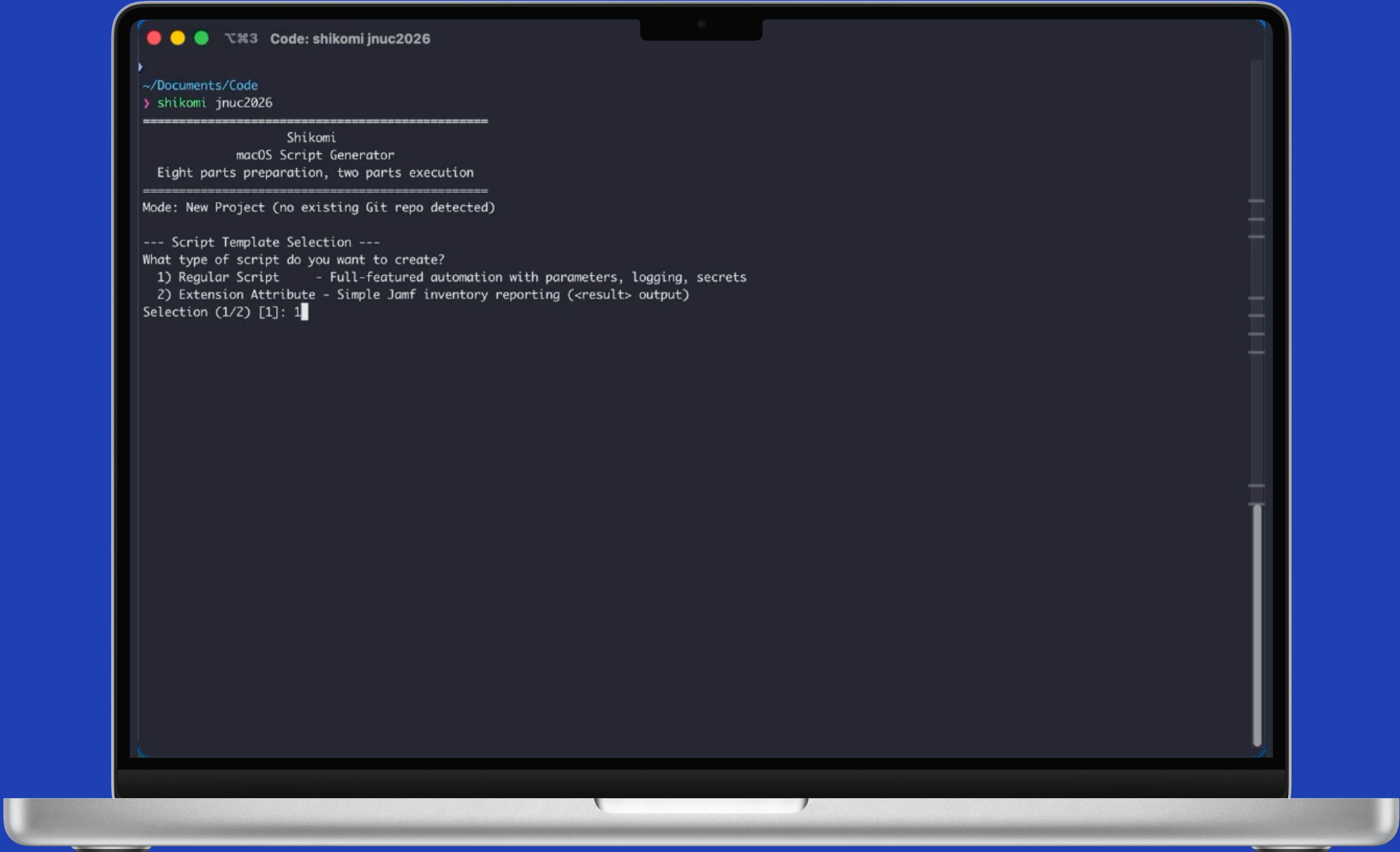
Secret scanning. Linting. Semantic versioning.
Wired together from the first commit.



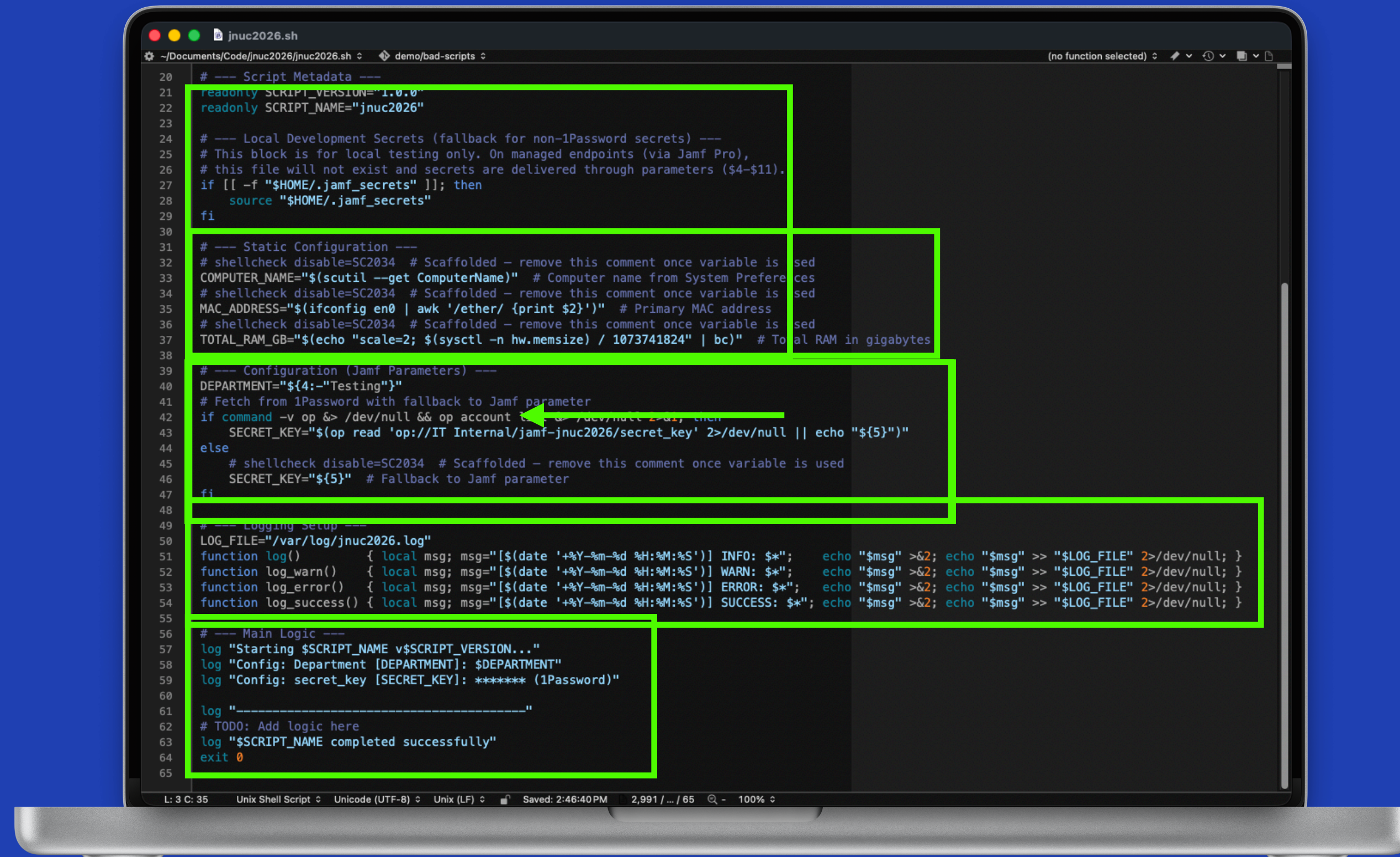
Choosing a Starting Point



Five Decisions, Made Once



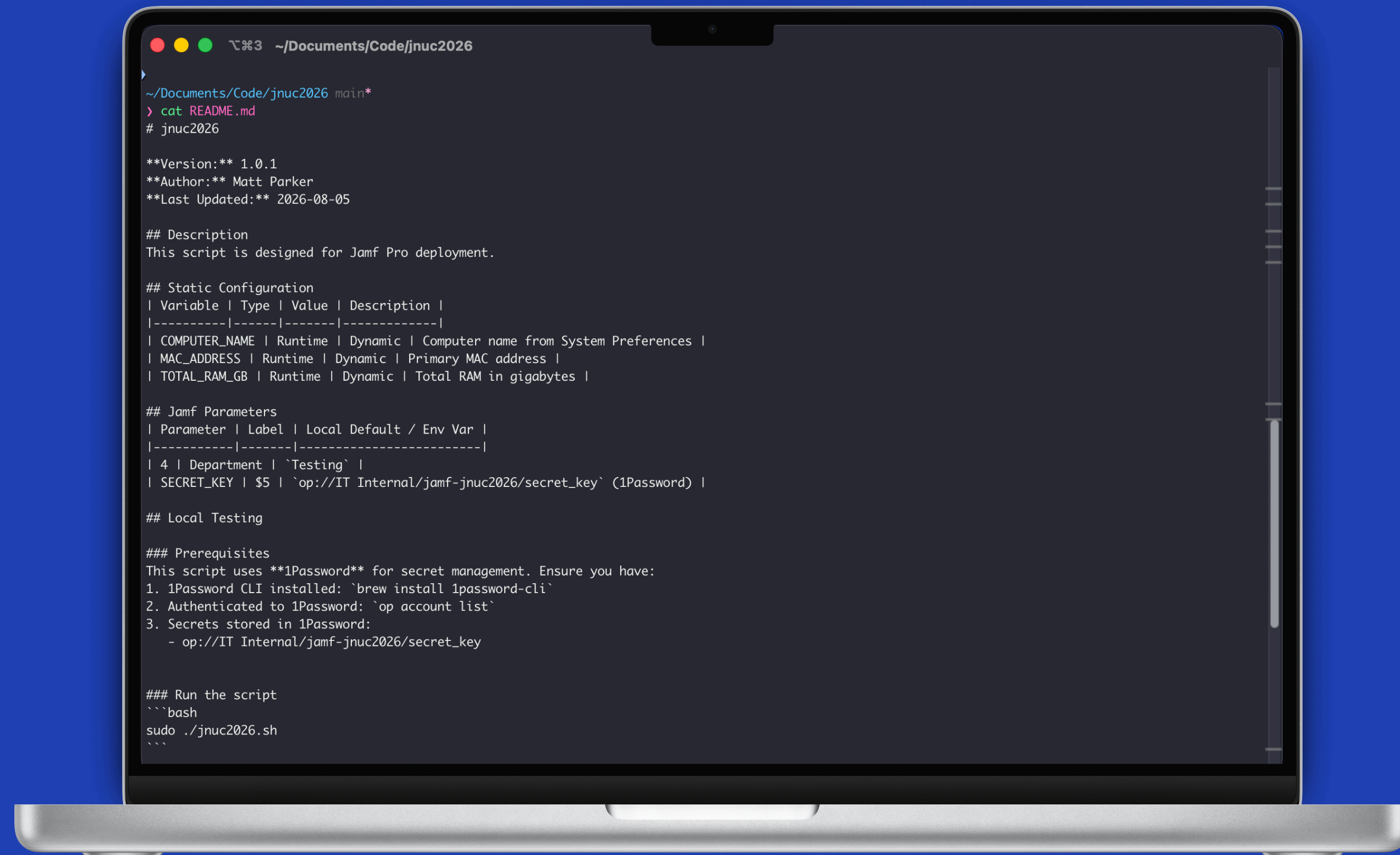
What's Actually in the File



```
20 # --- Script Metadata ---
21 readonly SCRIPT_VERSION="1.0.0"
22 readonly SCRIPT_NAME="jnuc2026"
23
24 # --- Local Development Secrets (fallback for non-1Password secrets) ---
25 # This block is for local testing only. On managed endpoints (via Jamf Pro),
26 # this file will not exist and secrets are delivered through parameters ($4-$11).
27 if [[ -f "$HOME/.jamf_secrets" ]]; then
28     source "$HOME/.jamf_secrets"
29 fi
30
31 # --- Static Configuration ---
32 # shellcheck disable=SC2034 # Scaffolded - remove this comment once variable is set
33 COMPUTER_NAME="$(scutil --get ComputerName)" # Computer name from System Preferences
34 # shellcheck disable=SC2034 # Scaffolded - remove this comment once variable is set
35 MAC_ADDRESS="$(ifconfig en0 | awk '/ether/ {print $2}')" # Primary MAC address
36 # shellcheck disable=SC2034 # Scaffolded - remove this comment once variable is set
37 TOTAL_RAM_GB="$(echo "scale=2; $(sysctl -n hw.memsize) / 1073741824" | bc)" # Total RAM in gigabytes
38
39 # --- Configuration (Jamf Parameters) ---
40 DEPARTMENT="${4:-"Testing"}"
41 # Fetch from 1Password with fallback to Jamf parameter
42 if command -v op && /dev/null && op account < /dev/null 2>/dev/null; then
43     SECRET_KEY="$(op read 'op://IT Internal/jamf-jnuc2026/secret_key' 2>/dev/null || echo "${5}")"
44 else
45     # shellcheck disable=SC2034 # Scaffolded - remove this comment once variable is used
46     SECRET_KEY="${5}" # Fallback to Jamf parameter
47 fi
48
49 # --- Logging Setup ---
50 LOG_FILE="/var/log/jnuc2026.log"
51 function log() { local msg; msg="[$(date '+%Y-%m-%d %H:%M:%S')] INFO: $*"; echo "$msg" >&2; echo "$msg" >> "$LOG_FILE" 2>/dev/null; }
52 function log_warn() { local msg; msg="[$(date '+%Y-%m-%d %H:%M:%S')] WARN: $*"; echo "$msg" >&2; echo "$msg" >> "$LOG_FILE" 2>/dev/null; }
53 function log_error() { local msg; msg="[$(date '+%Y-%m-%d %H:%M:%S')] ERROR: $*"; echo "$msg" >&2; echo "$msg" >> "$LOG_FILE" 2>/dev/null; }
54 function log_success() { local msg; msg="[$(date '+%Y-%m-%d %H:%M:%S')] SUCCESS: $*"; echo "$msg" >&2; echo "$msg" >> "$LOG_FILE" 2>/dev/null; }
55
56 # --- Main Logic ---
57 log "Starting $SCRIPT_NAME v$SCRIPT_VERSION..."
58 log "Config: Department [DEPARTMENT]: $DEPARTMENT"
59 log "Config: secret_key [SECRET_KEY]: ***** (1Password)"
60
61 log "-----"
62 # TODO: Add logic here
63 log "$SCRIPT_NAME completed successfully"
64 exit 0
65
```

Every one of these six things existed before a single line of business logic was written.

Versioning That Stays in Sync



Updates the script header, the README and the CHANGELOG in the same command, and the munkipkg build-info file, if the script ships inside a package. patch/minor/major map directly to semantic versioning's x . y . z.

Getting There From Here

Bridging to Jamf Pro, and where to start Monday.



What Happens After "Merge"

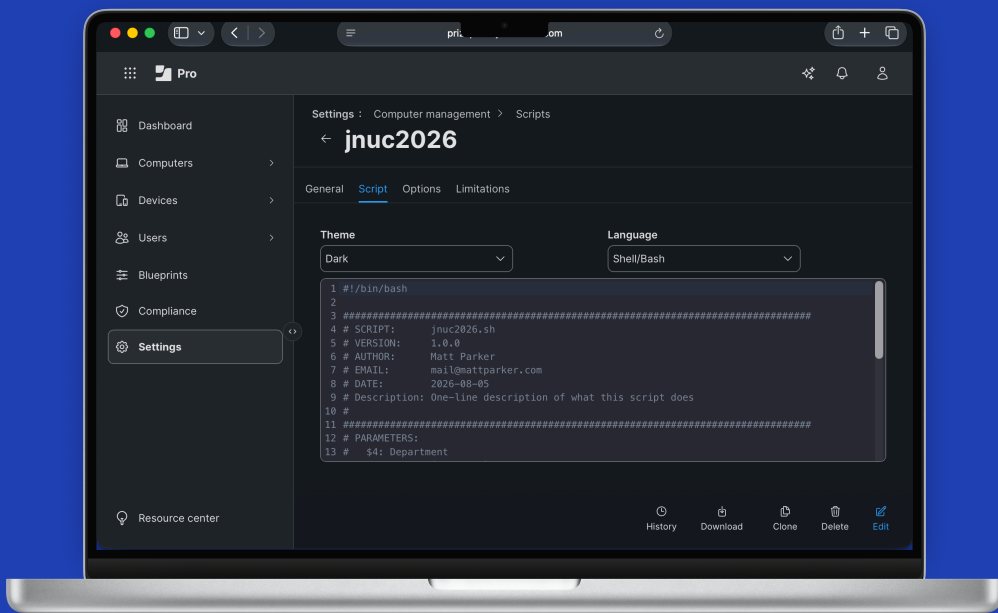
PR MERGED
TO MAIN



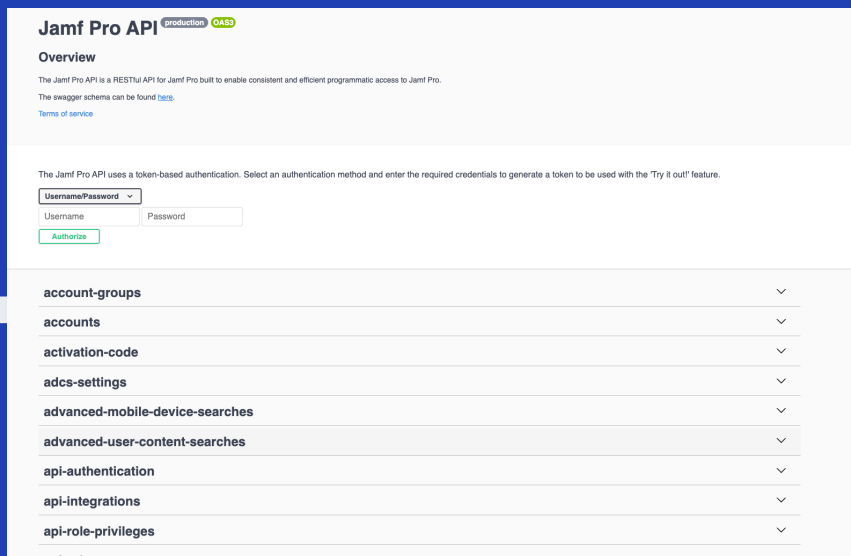
GITHUB RUNNER
ACTION



JAMF PRO



JAMF PRO API



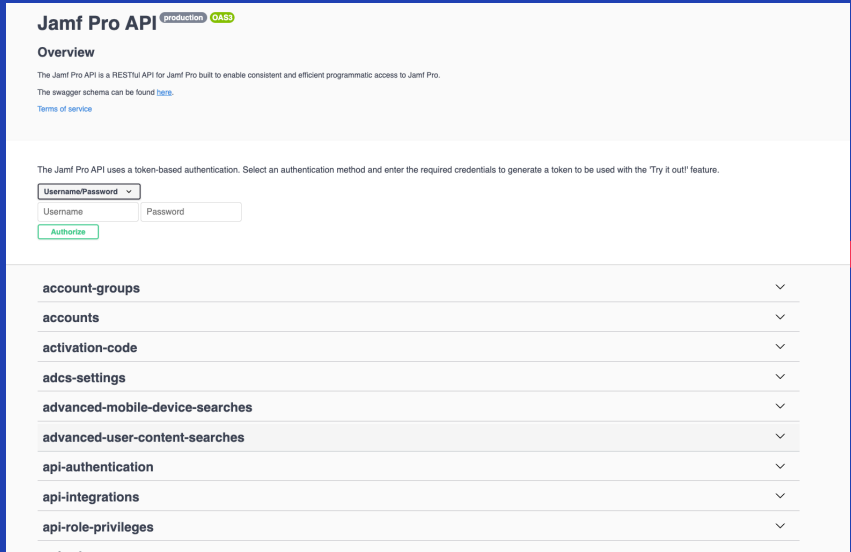
Any policy that already references this script by ID sees the new content automatically; there's no separate "update the policy" step. The script's identity in Jamf doesn't change, only its contents.

Does a Bad Deploy Ever Reach the Fleet?

GITHUB RUNNER
ACTION



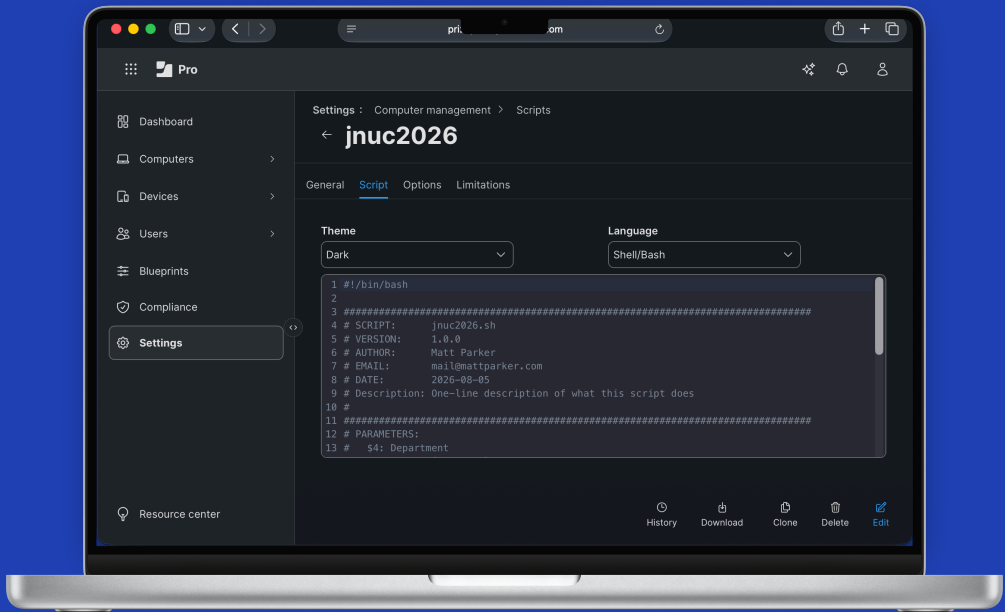
JAMF PRO API CALL



FAILS

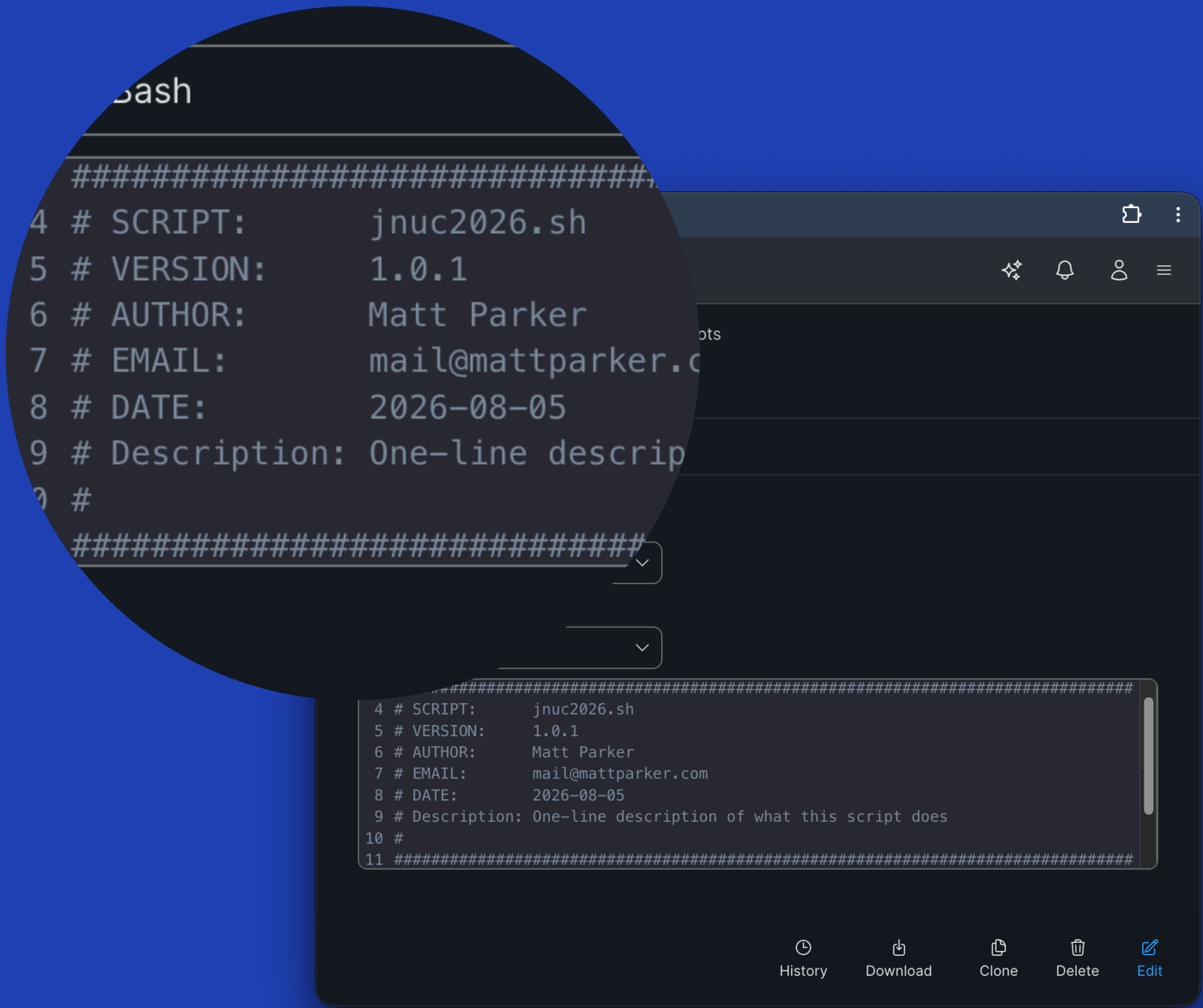
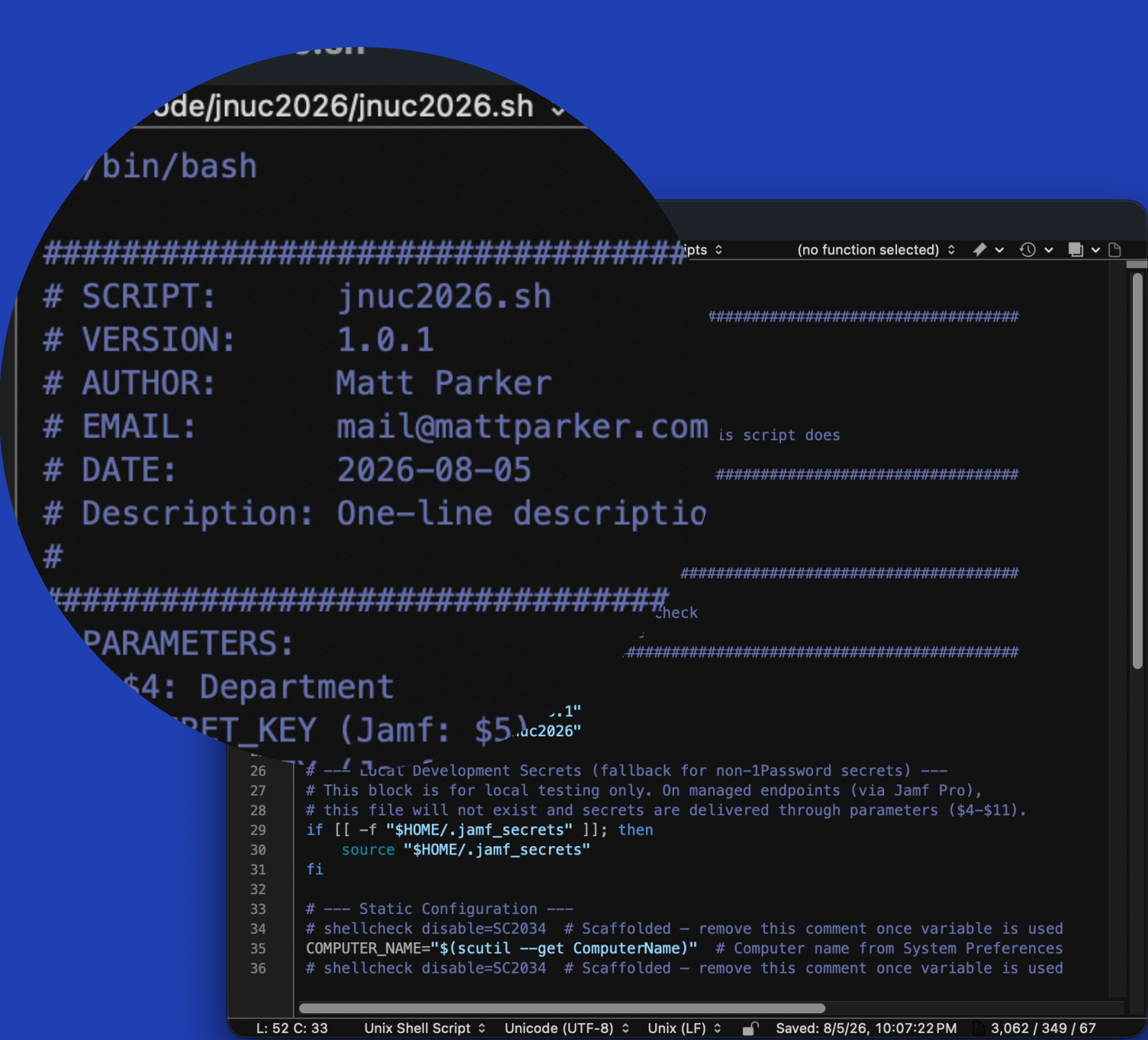


JAMF PRO



A failed deploy means nothing happened. It doesn't mean something broke.

How Do You Know It Actually Deployed?



Drift detection in 30 seconds; no console archaeology required.

You're Not Betting on a Future Feature

As native integrations between Jamf Pro and source control continue to evolve, teams already working this way will be positioned to take advantage of tighter connections as they arrive.

One Script. Not a Hundred.

1. Pick one new script you haven't written yet.
2. Run it through the full workflow: Shikomi scaffold, pre-commit, PR, CI, deploy.
3. Let that one script prove the process before you touch a single legacy script.

Don't migrate. Adopt.

What Governed Script Management Actually Means

Version Control

The repository is the source of truth. Every change is reviewed. Every version is recoverable.

Secret Scanning

Credentials don't enter the repo. Local hooks catch them early. CI catches what hooks miss.

Automated Linting

Subtle bugs get found before deploy. The same rules apply to every team member. Quality stops being a personality trait.

Three pillars. Three tools. One framework that wires them together.

Take It With You

- ▶ **Repository** github.com/maparker/shikomi
- ▶ **Documentation** github.com/maparker/shikomi/issues
- ▶ **License** MIT. Fork it, modify it, ship it.

Tooling Caught It. After the Fact.

The next time, I want it to catch it before the fact.

QUESTIONS?

- ▶ **Repository** github.com/maparker/shikomi
- ▶ **Documentation** github.com/maparker/shikomi/issues
- ▶ **Slack** @mattparker
- ▶ **Hallway**