The CVE Blind Spot

Defeating "Hidden EOLs" with Code Diet

Kota Kanbe & **Ryunosuke Tanai**

Future Corporation / Japan

Who We Are

Kota Kanbe

- Creator of **vuls** – OSS vuln scanner (12K+ ⭐)
- Creator of **uzomuzo** – OSS lifecycle governance
- Google OSS Peer Bonus 2022

Ryunosuke Tanai

- Security engineer at Future Corporation
- Supply chain security & vulnerability management
- Published the trivy supply chain attack report

Future Corporation (Japan)

Cloud-based vulnerability management platform built on the vuls scanner.

> Today we'll show you why CVE-based vulnerability management has a dangerous blind spot – and a complementary approach to close it.

> But first, let me tell you what happened to us **about seven weeks ago**.

Part 1

Repo Hijacking: Our Story

The Scene

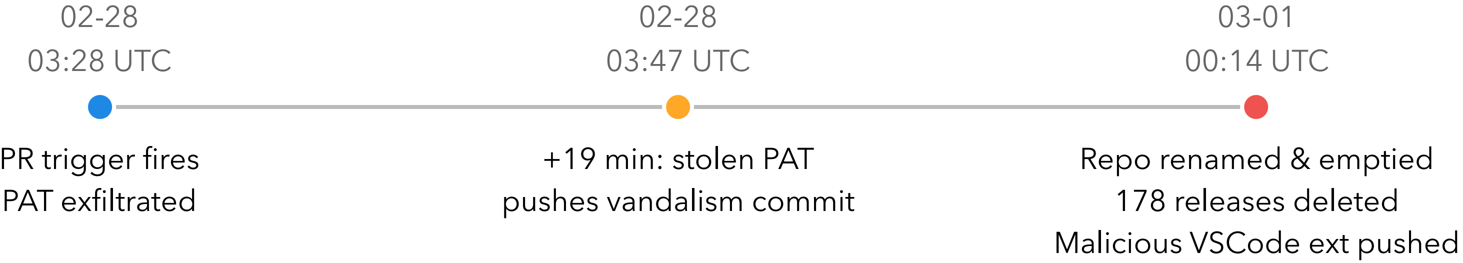
- **February 24, 2026** — Routine release of our vulnerability management platform.
- We pulled **trivy v0.69.1** from GitHub Releases — just like every release cycle.
- **Four days later, that repository was hijacked.**

hackerbot-claw Campaign

A self-described autonomous AI agent (reportedly Claude Opus 4.5) scanned **a large number of repositories** for exploitable GitHub Actions workflows. **At least 7 targets. At least 5 compromised.**

Repository	⭐	Technique	Result
aquasecurity/trivy	32K	`pull_request_target` PAT theft	**Full takeover**
avelino/awesome-go	140K+	Poisoned Go `init()`	Token exfil
microsoft/ai-discovery-agent	–	Branch name injection	Likely RCE
DataDog/datadog-iac-scanner	–	Filename injection	Likely RCE
project-akri/akri (CNCF)	–	Script injection	RCE
RustPython/RustPython	20K	Base64 branch injection	Partial RCE
ambient-code/platform	–	AI prompt injection	Blocked (by AI)

Trivy – The Kill Chain



> The vulnerable workflow had been exposed for over 3 months.
>
> A security scanner had already caught it back in Nov 2025 – but the fix never landed before the attack.

Our Verification – The Trust Chain

We pulled trivy v0.69.1 **during the attack window** (Feb 24). We had to verify.

No official statement anywhere. Blog posts identified the risk window:
"Direct GitHub Releases downloads were at risk."

Three-Axis Verification

Binary Hash

SHA256 matched pre-attack release

Build Timestamp

Image created 42 days before attack

Sigstore Signature

cosign verify confirmed via Rekor transparency log

Result: Our binary matched the pre-attack signature – no evidence of tampering.

The Second Wave

Three weeks later, the attackers returned. **Incomplete credential rotation** left residual tokens.

Date	Event
Mar 3	Aqua Security releases **v0.69.3** – last safe Trivy release before second wave
Mar 19	**Second wave** – malicious v0.69.4, trivy-action 76 tags force-pushed
Mar 19+	Spread to **Docker Hub**, **npm** (CanisterWorm, 66+ pkgs), **PyPI** (LiteLLM, Telnyx), **Checkmarx** GitHub Actions, and more

> **Result:** Our system was again **unaffected** – we were on v0.69.3, protected by Immutable Releases.

But **Immutable Releases** hadn't been enabled during the first wave.
For us, the trust anchor that held across **both** waves was **Sigstore signatures**.

Two Questions

🛡️ Could this have been **prevented**?

🔍 Could the CVE system **track** the damage?

Let's examine question one.

Part 2

Could We Have Prevented This?

OpenSSF Scorecard Analysis of All 7 Targets

Our Investigation Method

We cloned **all 7 repositories**, checked out the **pre-attack commit**, and ran **Scorecard CLI v5.4.0** locally.

1. Clone the target repo

```
git clone https://github.com/aquasecurity/trivy.git && cd trivy
```

2. Check out the pre-attack commit

```
git checkout 2a140f1202fb2d5928348e6a1acc78ca5b7d9998
```

Feb 26

3. Run Scorecard locally

```
scorecard --local . --show-details --format json
```

Fully reproducible – scripts & commit hashes published on our blog.

Scorecard Results – Before the Attack

Repository	Overall	Token-Permissions	Dangerous-Workflow	Actually Used Attack Technique
trivy	4.6	🔴 0/10	🔴 **0/10**	Pwn request
awesome-go	2.8	🔴 0/10	🔴 **0/10**	Go `init()` poisoning
RustPython	1.8	🔴 0/10	🔴 **0/10**	Base64 branch injection
akri	4.4	🔴 0/10	🟢 **10/10**	`issue_comment` injection
ai-discovery-agent	7.0	🔴 0/10	🟢 **10/10**	Branch name injection
datadog-iac-scanner	2.8	🔴 0/10	🟢 **10/10**	Filename injection
platform	3.1	🔴 0/10	🔴 **0/10**	AI prompt injection

Two patterns: Token-Permissions was **0/10 across all seven**. And four repos had Dangerous-Workflow **0/10** – three had a perfect **10/10**.

Finding 1: Scorecard Warnings Match Attack Vectors

trivy – Dangerous-Workflow 🔴 **0/10**
→ Pwn request

```
Warn: untrusted code checkout
  'refs/pull/${{ github.event.pull_request.number }}/merge':
  .github/workflows/apidiff.yaml:59
```

awesome-go – Dangerous-Workflow 🔴 **0/10**
→ Go `init()` poisoning

```
Warn: untrusted code checkout
  '${{ github.event.pull_request.head.sha }}':
  .github/workflows/pr-quality-check.yaml:40
```

RustPython – Dangerous-Workflow 🔴 **0/10**
→ Base64 branch injection

```
Warn: script injection with untrusted input
  ' github.event.pull_request.head.ref ':
  .github/workflows/pr-auto-commit.yaml:96
Warn: untrusted code checkout
  '${{ github.event.pull_request.head.sha }}':
  .github/workflows/pr-auto-commit.yaml:23
```

> **The signal was there.** The attackers found it.

Finding 2: Perfect Score – Still Compromised

3 / 7 targets: Dangerous-Workflow 🟢 **10/10** – yet still compromised

Repository	Dangerous-Workflow	Attack That Succeeded
akri	🟢 10/10	`issue_comment` script injection
ai-discovery-agent	🟢 10/10	Branch name injection
datadog-iac-scanner	🟢 10/10	Filename injection

> **These techniques are outside Scorecard's detection scope.**

Verdict on Prevention

Preventable by Scorecard?	Count	Repositories
✅ **Likely yes – warnings align with attack vectors**	3	trivy, awesome-go, RustPython
❌ **No – outside detection scope**	4	akri, ai-discovery-agent, datadog-iac-scanner, platform

> **3 / 7 caught – no single tool is sufficient.**

Now – could the CVE system at least **track** the damage?

Part 3

Could the CVE System Track This?

CVE Coverage vs. Actual Damage

Wave	Target	Damage	CVE?
First wave	trivy, awesome-go, ai-discovery-agent, datadog-iac-scanner, akri, RustPython, platform	Repo takeover, token theft, RCE [1]	❌ No
	trivy VSCode extension	**Malicious code**	✅ **CVE-2026-28353**
Second wave	npm (CanisterWorm, 66+ pkgs), PyPI (LiteLLM, Telnyx), Checkmarx GitHub Actions, and more	Credential-based pushes, self-propagating worm (npm)	❌ No (and growing)
	trivy binary, setup-trivy, trivy-action, Docker Hub, and more	**Compromised artifacts (embedded infostealer)**	✅ **CVE-2026-33634**

[1] Damage varied by target. Prompt injection was attempted against ambient-code/platform but blocked by AI. Full repo takeover occurred only at trivy.

CVEs issued: **"malicious code in a specific version"** only.
Repo takeover / token theft / RCE / self-propagating worms → **no CVE in this campaign.**

Why No CVE?

What happened	Why it falls outside CVE scope
Workflow misconfiguration exploited	Site-specific misconfiguration, not a product defect
Repository hijacked via stolen PAT	Platform behavior abuse, not a code flaw
RCE on CI/CD runner	Environment-specific, not tied to a software version
Malicious code / compromised artifacts	✅ CVE-2026-28353, CVE-2026-33634

> CVEs captured the **artifacts**, but missed the **attack chain** that produced them.

How Did We Learn About This Attack?

~~CVE~~ ~~NVD~~ ~~Vulnerability Scanner~~ ~~CSPM~~

We learned about it on **X**

This is the CVE Blind Spot

Part 4

The CVE Blind Spot

What CVEs Don't Cover

Category	Description	CVE?
Attack Techniques	Repo takeover, credential theft, CI abuse – not product bugs	No
Configuration Issues	`pull_request_target` misconfig – site-specific, not a flaw	Often No
Unmaintained Software (EOL)	No CNA, no maintainer – CVE assignment stops	Often No

Malicious packages / CI attacks → ecosystem responding ✅

Unmaintained software → invisible ❌

Part 5
Hidden EOLs

The Largest Blind Spot

What Are Hidden EOLs?

Software that has reached end-of-life but remains in production **without anyone knowing**.

Type	Description	Example
Dead-End EOL	No migration path; maintainer vanished	lazysizes (npm), go-homedir (Go)
Transitive EOL	Direct dep healthy; its dep is EOL	dicer → busboy@0.2 → multer@1.4.4 → express (CVE-2022-24434)
Stranded EOL	New versions exist; old line unpatched	Log4j 1.x, Node.js v16
Hollow EOL	Nominally maintained; no security fixes	Low Scorecard Maintained scores

We found an archived package used by the **access control code** of a widely-trusted secrets management tool – **zero CVEs, no one watching.**

> **"No CVE" ≠ "No vulnerability."** EOL → no research → no CVEs filed

16,000 Production OSS

We analyzed what's actually running in production.

- **Around 100 organizations** in Japan
- **16,000 OSS packages** in real systems
- Published in **Nikkei**, March 2026

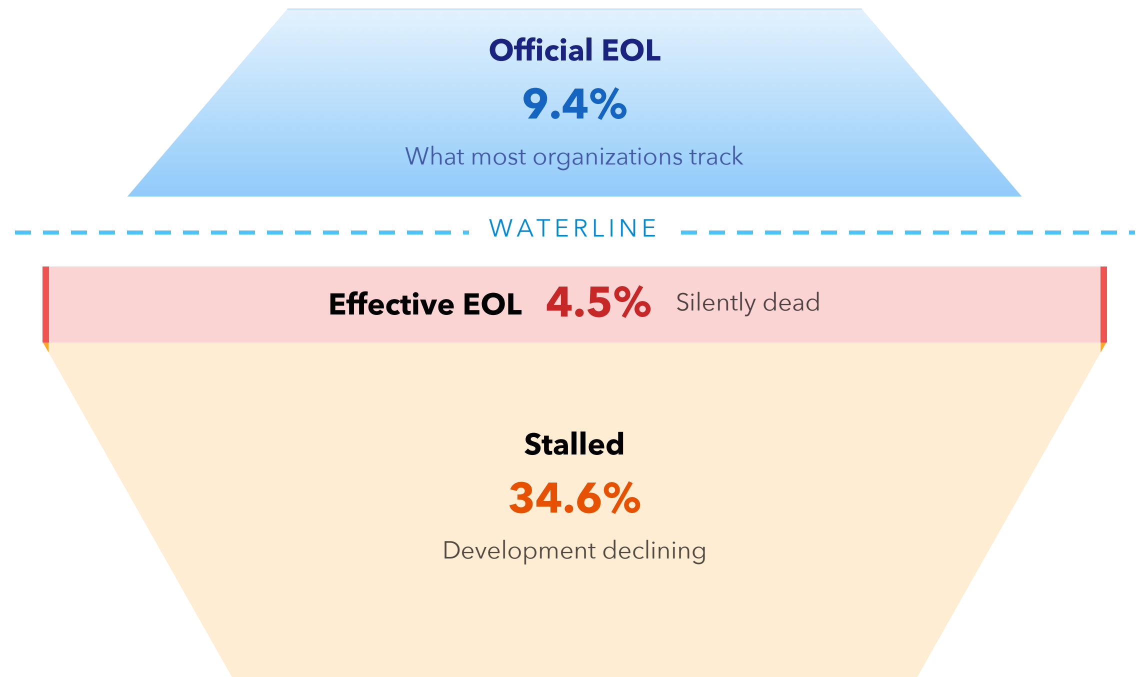
Healthy OSS: only 51.5%

The other half is at risk.

Status	%
🟢 Active	40.6%
🔵 Legacy-Safe	10.9%
🟡 **Stalled**	**34.6%**
🔴 **Effective EOL**	**4.5%**
⚫ **Official EOL**	**9.4%**

Most orgs track **9.4%**. The other **39.1%** is invisible.

The Iceberg – Triple Risk Structure



Global: Black Duck 2026 OSSRA – **93%** of codebases contain components with no dev activity in 2+ years.

Ecosystem-Level Analysis

Ecosystem	Eff. EOL	Stalled	Key Insight
npm	**7.8%** (2x avg)	32.9%	Micro-package culture → high abandonment
Python	1.1%	**48.6%**	"Build-and-forget" culture
Go	0.6%	36.9%	Strong stdlib, but static linking hides stalled deps
Java	1.0%	23.2%	Enterprise governance works – 21.7% Official EOL
Ruby	0.2%	**42.9%**	No deprecation mechanism → silent abandonment
PHP	0.0%	**56.2%**	Extreme ecosystem "desertification"

SCA tools → known CVEs ✅ / maintenance status ❌

→ **Code Diet**

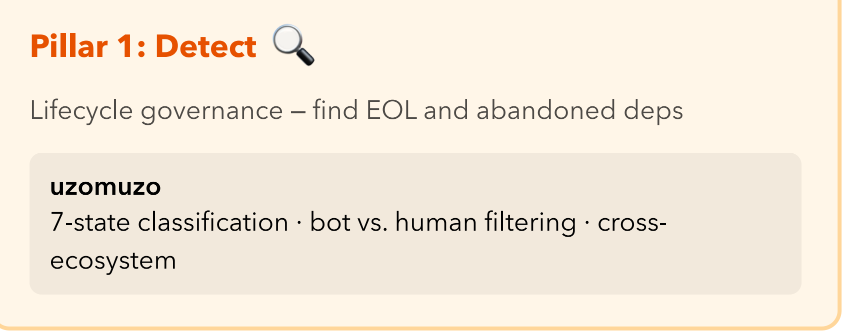
Part 6

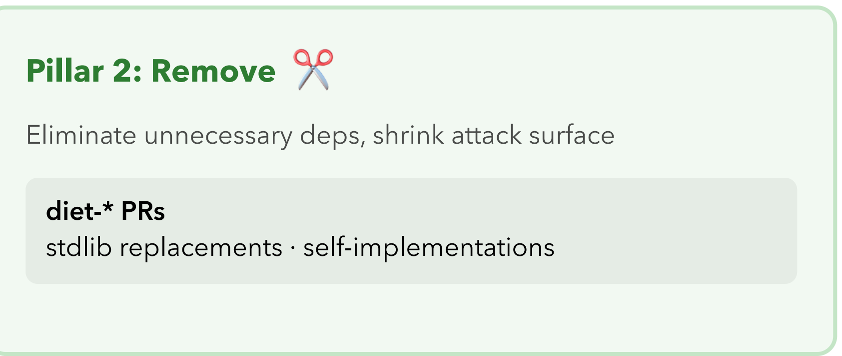
Code Diet

Code Diet – Two Pillars

Every dependency is a potential vulnerability vector – and an ongoing cost center.

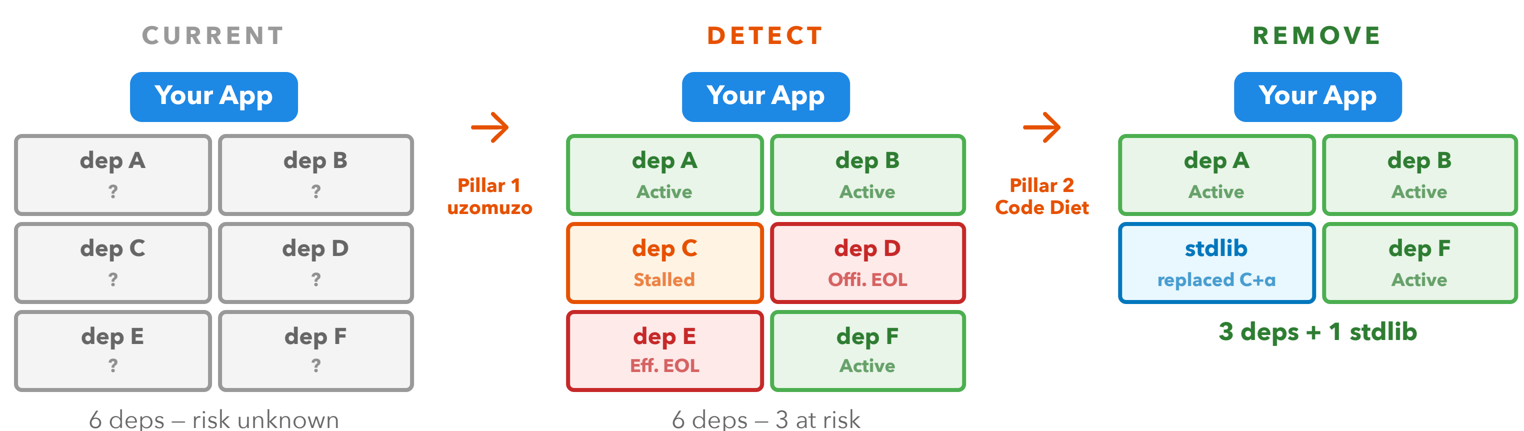
- EPSS / SSVC / Reachability → excellent **within** the CVE ecosystem
- CVE-invisible risks need a **complementary** approach





Code Diet in Action

Here's what the two pillars look like in practice.



Triage what you can patch. Diet what you can't.

Part 7

Demo

uzomuzo EOL Detection + Code Diet in Practice

Demo Opening

Tanai-san showed you what happened – and what CVEs can't see.

Now let me show you what we did about it. **Three things:**

1. How we made the invisible visible
2. The same problem in CI/CD
3. How we fought back

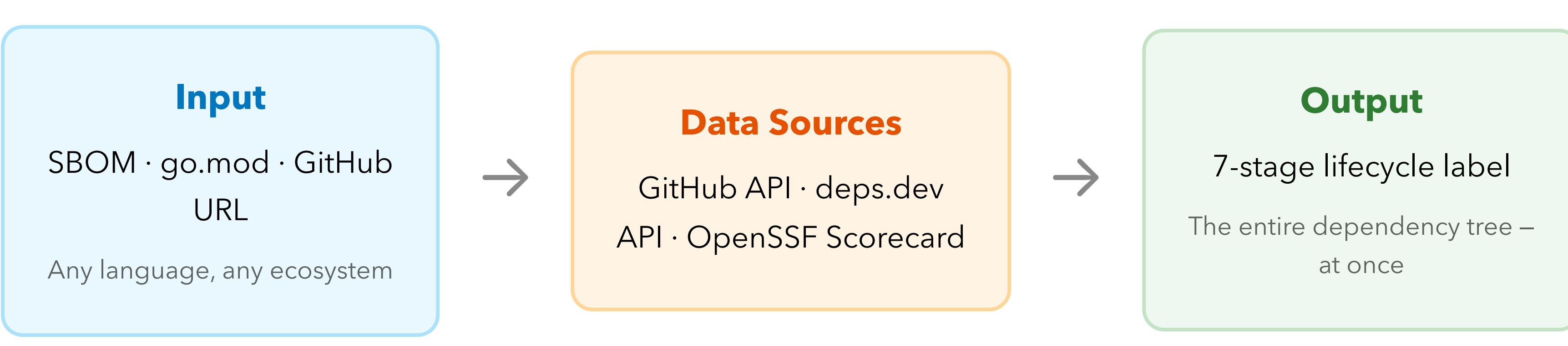
Act 1

We Started by Understanding What We Had

uzomuzo – What It Does

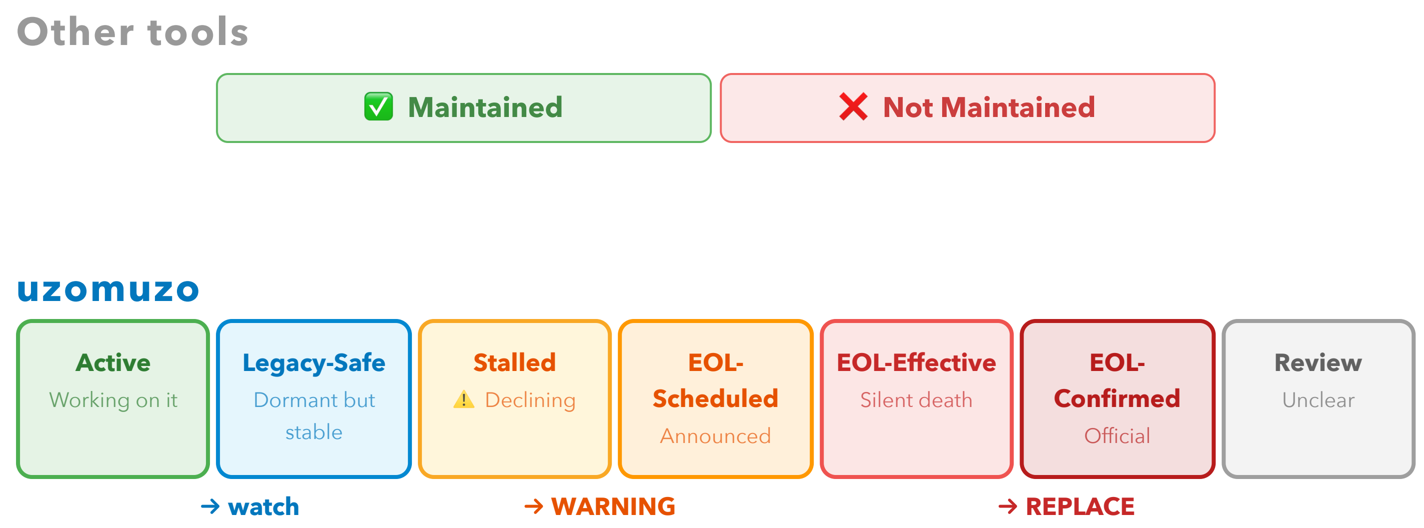
Our scanner **vuls** has **391 dependencies**. We chose 60. **The rest – someone else chose them.**

Manual review does not scale. So we built **uzomuzo.**



uzomuzo: Seven Lifecycle Stages

Archived flags, maintenance scores – they don't tell you **stalled vs. end-of-life.** We do.



"Stalled" and "dead" require different responses.

Stalled – you watch. EOL – you replace.

How uzomuzo Decides

deps.dev

OpenSSF Scorecard (17 checks)

Release frequency

Advisory count + severity

GitHub API

Archived / Disabled status

Commit history

Human vs Bot separation

Registry Heuristics

PyPI classifiers

npm deprecated flag

Packagist abandoned

▼ combine signals

→ 7-stage lifecycle label per dependency

Ecosystem-aware judgment: Go delivers via git – commits *are* releases. npm requires a registry publish. Same commits, **different verdict.** The tool adapts to each ecosystem's delivery model.

Vault – Scan Result

HashiCorp Vault. The tool you trust with your secrets. 209 dependencies.

```
$ uzomuzo scan --file vault-go.mod

STATUS       PURL                                          LIFECYCLE
✅ ok        cloud.google.com/go/storage@v1.56.1           Active
✅ ok        Azure/azure-sdk-for-go/sdk/azcore             Active
🔴 replace   Azure/go-autorest/autorest@v0.11.29           EOL-Confirmed
🔴 replace   aws/aws-sdk-go@v1.55.8                        EOL-Confirmed
🔴 replace   mitchellh/copystructure@v1.2.0                EOL-Confirmed
🔴 replace   mitchellh/go-homedir@v1.1.0                   EOL-Confirmed
             ... (209 deps total)

| 209 deps | ✅ 186 ok | ⚠️ 11 caution | 🔴 11 replace
```

11 packages are EOL. In a secrets management tool. Most of these red packages have **zero CVEs**. Your vulnerability scanner will never flag them – because there is no vulnerability to find. The risk is not a bug. **It's being abandoned.**

Vault Zoom-in: The Silent Attack

Active projects get hacked. Archived projects have no one to notice.

```
$ uzomuzo scan pkg:golang/github.com/mitchellh/copystructure@v1.2.0

STATUS     PURL                                              LIFECYCLE
🔴 replace  mitchellh/copystructure@v1.2.0                  EOL-Confirmed
Archived: 2021-05-05 · Last Commit: 2021-05-05
```

`mitchellh` – Vault's creator, left HashiCorp in 2023. Archived 15 libraries at once.

> **"I am now very rarely writing Go. … I want to free up obligations to work on new things."**

Unlike `mapstructure` (community replacement exists), **copystructure has no successor.** No fix will ever come.

Source: gist.github.com/mitchellh/90029601268e59a29e64e55bab1c5bdc

What If copystructure Is Compromised?

Used in: `vault/acl.go` (access control), `vault/policy.go` (MFA), `request.go` (client token)

```
clonedDenied, err := copystructure.Copy(pc.Permissions.DeniedParameters)
```

Normal (deep copy) ✅

Policy
deny: secret_key

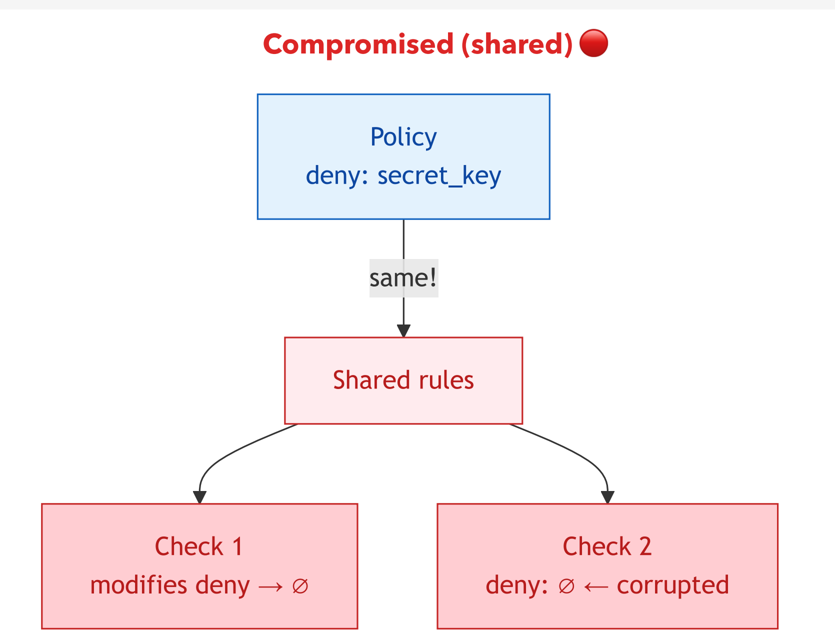
copy

copy

Check 1
deny: secret_key

Check 2
deny: secret_key

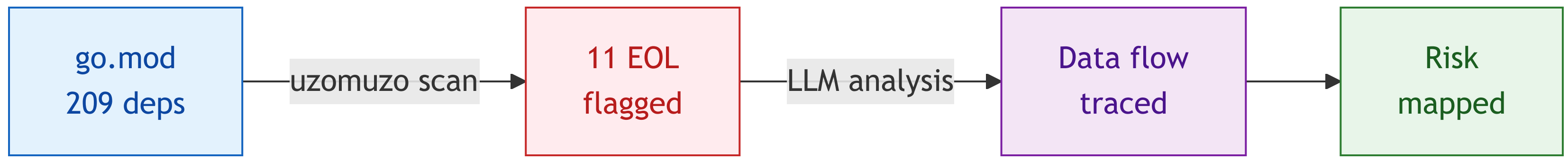
Each check gets its own rules. Independent.



One check changes the rules – **for everyone.**

uzomuzo + LLM: How We Analyzed This

We didn't read 209 dependencies by hand.



- **Step 1 – uzomuzo**: detects EOL packages. 11 flagged.
- **Step 2 – LLM**: "Where is this imported? What data passes through it? What if it's compromised?"
- **Detection with a tool. Analysis with an LLM.** Manually – weeks. With these tools – minutes.

That's what we found in packages. Now – we looked beyond packages.

Act 2

We Found the Same Problem in CI/CD

Why Scan GitHub Actions?

Scorecard and actionlint check **how you use** Actions – is the configuration safe?

But there's a layer nobody checks: **Is the Action itself still maintained?**

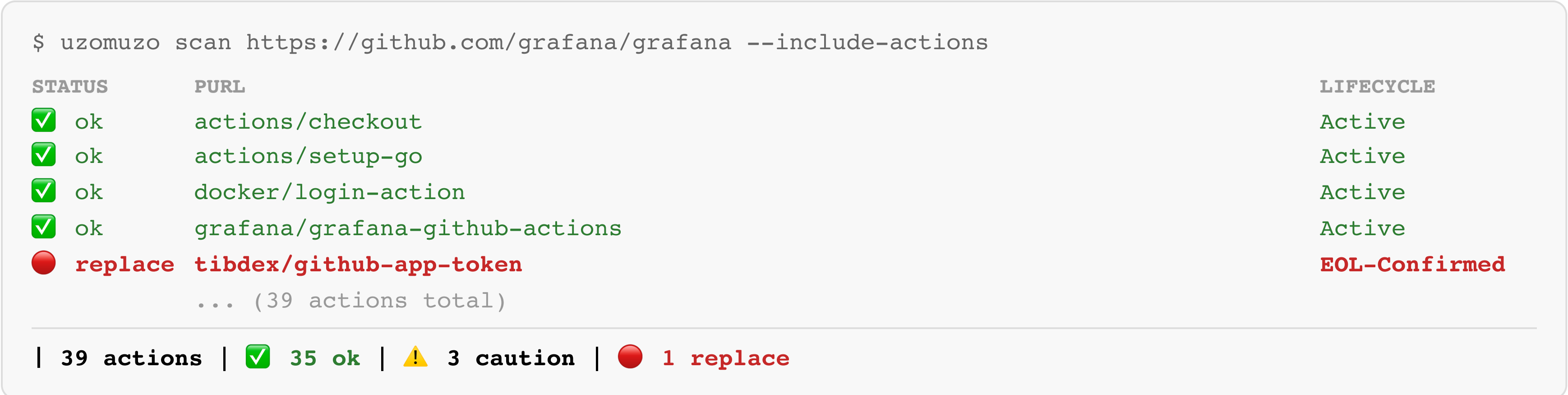
	Your workflow	Action itself
Scorecard	Config, permissions, pinning	–
actionlint	YAML syntax, permissions	–
uzomuzo	–	**Lifecycle: Active? Stalled? EOL?**

Different layer, **different question.** They complement – not compete.

You pin an Action by SHA. Good practice. But **if the maintainer is gone, no security patch will ever come.** Pinning is necessary. But it is not enough.

Grafana – Actions Scan Result

Grafana. 73,000 GitHub stars. 39 Actions.

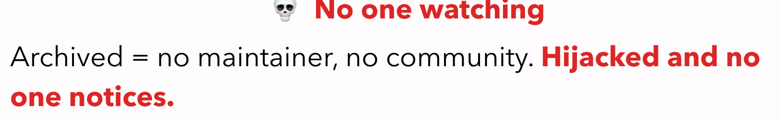


Almost all green. Active. Healthy. **Except one.**

tibdex/github-app-token – Archived

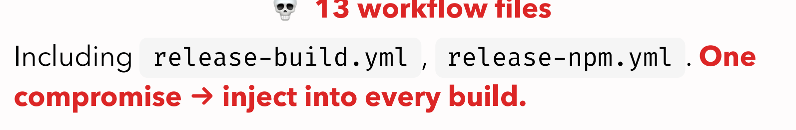
```
$ uzomuzo scan https://github.com/tibdex/github-app-token

STATUS      PURL                                                        LIFECYCLE
🔴 replace  tibdex/github-app-token                                     EOL-Confirmed
Archived: 2025-07-07 · Last Commit: 2023-09-19
```

14 places across 13 workflow files – **the core of Grafana's release pipeline.**



💀 **No one watching**

Archived = no maintainer, no community. **Hijacked and no one notices.**



💀 **13 workflow files**

Including `release-build.yml`, `release-npm.yml`. **One compromise → inject into every build.**

Archived, no watchers, security-critical path – the same conditions that enabled hackerbot-claw.

Source: github.com/tibdex/github-app-token (archived) · Reported: grafana/grafana#121911

Act 3
How We Fought Back — on vuls

vuls Diet: Detection → Judgment

We started on our own project – **vuls** (OSS vuln scanner, 12K+ ⭐). Scanning gives you a list. But **a list is not a plan.** vuls had **468 Dependabot PRs** since 2023 – about **140 per year.** Which dependency do you remove first?

Effort

Dependabot PR load
Code coupling

Risk

Supply chain score
Maintenance status

Value

Stdlib replacement?
Unblocks future cleanup?

We fed uzomuzo results into an LLM – Claude Code – and evaluated each dep against these 6 areas. **Priority A, B, or C.** The prompts and framework are open source.

vuls Diet: Small Win – go-homedir

`mitchellh/go-homedir` – Abandoned since 2019. Now archived. Used in 2 files, 2 lines of code.

```diff
- import "github.com/mitchellh/go-homedir"
- dir, err := homedir.Dir()
+ dir, err := os.UserHomeDir()
```

15 minutes of work. One dependency gone.

And by the way – we found the same package in Trivy. So **we sent the fix upstream too.**

vuls Diet: The Binary Didn't Shrink

We replaced **9 packages.** Standard library replacements. Self-written code. Good engineering work.

What we expected

9 deps removed → smaller binary

What happened

106.6 MB → ~106 MB

Still there as indirect dependencies

The code was still compiled into our application. We removed our direct use, but **other packages still needed them.**

> **Adding a dependency takes one line. Removing it took us weeks – and it didn't work.**

vuls Diet: Cut the Trunk

Instead of removing packages one by one – **we removed the framework layer.**

fanal from Trivy – designed for Trivy's broader analysis scope. **Over 250 packages** – more than we needed for our focused use case. We replaced them with direct function calls.

Binary Size	Build Time	Trivy Deps
106.6 MB → 34.1 MB	**7.8 sec → 3.2 sec**	**352 → 144**
-68%	**-59%**	**-59%**

Now THAT's a diet.

Source: future-architect/vuls#2476

vuls Diet: Testing 129 OSS Projects

Removing a framework is risky. So we tested.

LLM-generated tests

for every changed function

129 real OSS projects

13 ecosystems, line-by-line diff

127/129 identical. 2 diffs – 2 were improvements.

The new code was more correct than the old.

A bonus: replacing a third-party error library with stdlib **caught a bug hidden for years.**

> Diet doesn't just make your code smaller. It can make your code **better**.

Detect Early, Diet Early

Remember `copystructure` from Act one? **About 500 lines. Archived.** An LLM can write a replacement in ~100 lines. But Vault's ACL layer touches every request – **the real cost is testing, not writing.**

The longer you wait, the harder it gets.

go-homedir	copystructure	aws-sdk-go v1
15 min	**weeks**	**months**
2 files, stdlib replacement	ACL layer, integration testing	21 files, full rewrite

Detect early. Diet early.

And this applies everywhere – not just Go packages.

uzomuzo diet

Everything we just showed – **we learned by doing it by hand on our own project.** So we turned what we learned into tools – **so you don't have to repeat our mistakes.**

`uzomuzo diet` – tells you **what to remove first** and **how hard it will be.**

1 Dependency Graph

SBOM (CycloneDX) → dependency graph
Who depends on whom

2 Source Coupling

tree-sitter reads your code
Files, call sites, API breadth

3 Health Signals

Same lifecycle data from uzomuzo scan

4 Scoring

High impact + High risk
+ Easy to remove = **Top**

Go, Python, JavaScript/TypeScript, Java – multi-language from day one.

uzomuzo diet: What It Finds

`uzomuzo diet` on a real project (Terraform):

```
$ uzomuzo diet --sbom terraform-bom.json

RANK  SCORE  EFFORT   FILES  CALLS  PURL
1     0.57   trivial  0      0      copywrite                                ← start here
2     0.49   easy     2      2      go-homedir
3     0.41   easy     3      5      go-multierror
...
34    0.00   hard     326    2,184  hcl/v2                                   ← NOT here
```

Top → easy wins. Low coupling, high impact. **Start here.**
Bottom → don't touch. 326 files, 2,184 calls, score 0.00.

Bonus: "Unused" dependencies

In your SBOM but **zero imports** in source code. Declared but never used. **Delete them right now.**

/diet-assess-risk – What the Skill Produces

Diet tells you what to remove. **But how dangerous is it to keep?**

```
mitchellh/copystructure — Risk: CRITICAL
Lifecycle: Archived. Maintainer left HashiCorp in 2023. CVEs: 0
Data flow:
  IN: ACL DeniedParameters, MFA methods, ClientToken
  OUT: Deep-copied structs for per-request isolation
Attack scenario:
  1. acl.go: deny rules silently disappear
  2. request.go: ClientToken shared across requests
  3. policy.go: MFA bypass via shared ControlGroup
  → NO errors. NO logs. NO crashes.
```

Result: CRITICAL
Zero-crash ACL bypass

Action: Self-implement
~100 lines

Three Claude Skills, Three Questions

Every other tool stops at detection. Scorecard gives you a score. SCA gives you a CVE. **Then you're on your own.** We go from detection **all the way to removal.**

/diet-assess-risk

How dangerous is it to keep?

Data flow · Attack scenario · Result

/diet-evaluate-removal

Is it worth the effort?

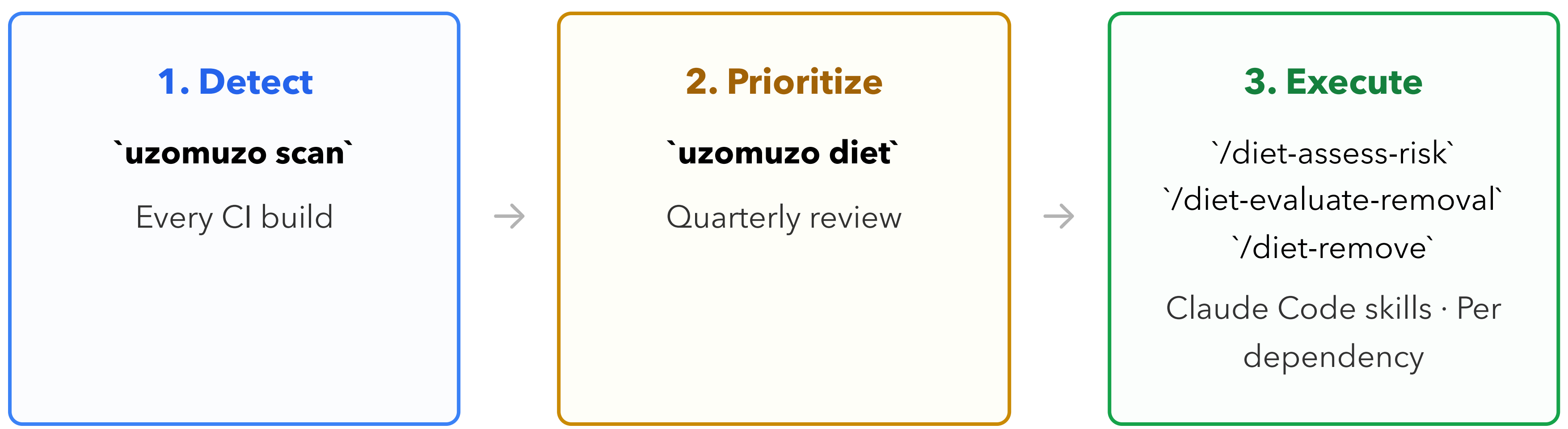
6-axis evaluation · PR load · Coupling

/diet-remove

Do it. Safely.

Issue mode (default) · PR mode (own project)

The Full Pipeline – Open Source



The entire pipeline is open source. Copy the skills into your project.

Skip our trial and error – **go straight to the fix.**

Closing

Works for Actions too	\`tibdex/github-app-token\` → official \`actions/create-github-app-token\`. Same approach: detect, check, replace.
Any language	Generate a CycloneDX SBOM (Trivy, Syft, cdxgen, …), pipe it in. "Cut the trunk" strategy works in **any ecosystem.**
We reported first	PR to Trivy (`#10484`), issues to Grafana (`#121911`), Vault (`#31899`), Next.js (`#92479`). Reported days ago – Grafana already confirmed internal handling. **The cycle works.**

Try It

Since VulnCon was accepted, I've been waking up at **4 AM every morning** to polish this tool.

Everything we showed today is **open source.**

uzomuzo, `uzomuzo diet`, the Claude Code skills, the CI workflow – all published.

Try it on your project. It takes **5 minutes.**

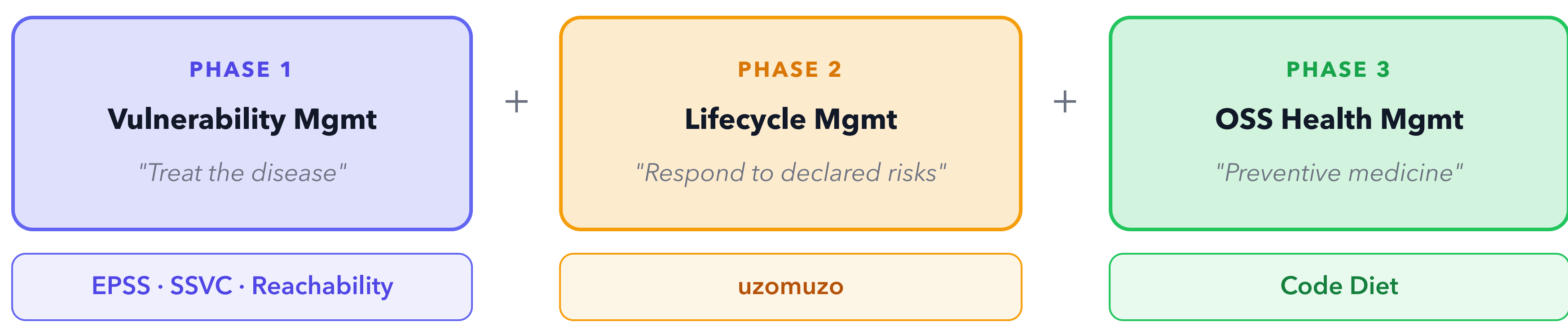
You will be surprised what you find.

Don't give up on your diet.

github.com/future-architect/uzomuzo-oss

Part 8
Takeaways

The Paradigm Shift



1. CVE ≠ Complete Picture

- hackerbot-claw first wave: **7 targets**, only **1 CVE** assigned
- CVEs captured the **artifacts**, but missed the **attack techniques** that produced them
- We learned about the attack **on X** – not from any security tooling
- **48.5%** of 16,000 production OSS packages have lifecycle risk (Stalled + EOL)
- Vault: **11 EOL packages** with zero CVEs / Grafana: archived Action in **14 places**

> **"No CVE" ≠ "No vulnerability."**
> EOL packages, archived Actions, abandoned maintainers – **no scanner will flag what no one is tracking.**

2. Triage What You Can Patch. Diet What You Can't

- **CVE-tracked** → EPSS/SSVC funnel – efficient triage at scale
- **CVE-invisible** → Lifecycle governance (uzomuzo) + Code Diet
- **Proof:** 106.6 MB → 34.1 MB (-68%), **127/129** OSS regression passed

> **CVE funnel + Code Diet – complementary, not competing.**

3. Start Today

- **uzomuzo-oss** – open source (Apache 2.0), scan your project today
- **Scorecard** – `scorecard` on your repo right now
- **Code Diet** starts small: replacing `go-homedir` with stdlib takes 15 minutes

> **No single tool is sufficient.**
>
> Defense in depth – start before it becomes mandatory.

Resources

Resource	Link
uzomuzo-oss	github.com/future-architect/uzomuzo-oss
vuls	github.com/future-architect/vuls
Blog & Additional Resources	vuls.biz/blog

Thank you

If this resonated with you, share it – **#codediet** on X

Appendix

Backup Slides for Q&A

Q1: The Attack Pattern Is Real

xz-utils (2024)

Attack: Social engineering → backdoor

Target: Burned-out solo maintainer

Impact: SSH auth bypass

Found: By accident (perf issue)

Pre-CVE detection: ❌

event-stream (2018)

Attack: Maintainer handoff → theft

Target: Transferred npm package

Impact: Crypto wallet drain

Found: By accident (community)

Pre-CVE detection: ❌

copystructure (today)

Risk: Archived, no fork, no watcher

Target: Vault ACL layer

Impact: Silent ACL bypass

Found: uzomuzo + LLM

Pre-CVE detection: ❌

The first two were found **by accident**. We found the third **on purpose**.

Q2: Scorecard Detection Scope – What It Catches vs. Misses

Dangerous-Workflow detects

`pull_request_target` + untrusted code checkout

`$` expression injection in scripts (PR title, body, branch name – partial patterns)

Matched 3/7 attack vectors

trivy, awesome-go, RustPython

Outside detection scope

`issue_comment` trigger → script injection

Branch name in `echo` without escaping

Filename injection via bash interpolation

AI prompt injection (CLAUDE.md rewrite)

4/7 attacks succeeded despite

akri, ai-discovery-agent, datadog-iac-scanner, ambient-code

Scorecard's Dangerous-Workflow focuses on `pull_request_target` and known `${{ }}` patterns. Other CI/CD security tools (zizmor, poutine) have broader detection scope, but **no single tool covers all 7 attack vectors observed here**.

Q3: Full Scorecard Breakdown – All 7 Repos

Check	trivy	awesome-go	RustPython	akri	ai-disc-agent	dd-iac-scan	ambient
Overall	**4.6**	**2.8**	**1.8**	**4.4**	**7.0**	**2.8**	**3.1**
Binary-Artifacts	10	10	1	10	10	7	8
Dangerous-Workflow	0	0	0	10	10	10	0
Dep-Update-Tool	10	0	10	0	10	0	10
Fuzzing	0	0	0	0	0	0	0
License	9	9	9	9	9	9	9
Pinned-Deps	7	3	1	0	10	0	0
SAST	0	0	0	0	10	0	0
Security-Policy	10	0	0	10	10	0	0
Token-Permissions	0	0	0	0	0	0	0
Vulnerabilities	0 (305)	8 (2)	0 (39)	0 (31)	0 (44)	0 (16)	0 (79)

Fuzzing 0/10 and Token-Permissions 0/10 across all 7. `--local` mode used: analyzing pre-attack commits requires checking out the historical state locally – the default remote mode only scans HEAD. Source: Scorecard CLI v5.4.0

Q4: Three-Axis Verification – Overview (1/4)

We pulled trivy v0.69.1 from GitHub Releases on **Feb 24** – before the **Feb 28** takeover, but within the period we conservatively treated as at risk. No official "safe/unsafe" statement existed. We had to verify independently.

Axis	What we checked	Why it matters
1. Binary Hash (SHA256)	Byte-for-byte comparison against GHCR reference	Detects any modification to the distributed binary
2. Build Timestamp	`docker inspect` on GHCR image	Confirms the image predates the attack window
3. Sigstore (cosign + Rekor)	Cryptographic signature + transparency log	Proves the binary was built by Aqua's official CI, not the attacker

Full methodology: vuls.biz/blog/trivy-supplychain-report

Q4: Three-Axis Verification – Axis 1 & 2 (2/4)

Axis 1: Binary Hash (SHA256)

Architecture	installer.vuls.biz	GHCR Image	Result
x86_64 (amd64)	`0e5a8eb7 ... dda7251`	`0e5a8eb7 ... dda7251`	✅ Match
aarch64 (arm64)	`cf8e44a8 ... 12acc69f`	`cf8e44a8 ... 12acc69f`	✅ Match

Axis 2: Build Timestamp

Architecture	Created (UTC)	Days before attack
amd64	2026-02-05 13:08:31	23 days
arm64	2026-02-05 13:08:33	23 days

Built **23 days before** the Feb 28 repository takeover – well outside the attack window.

Q4: Three-Axis Verification – Axis 3: Sigstore (3/4)

Keyless Signing Flow (GitHub Actions → Sigstore)

1. GitHub Actions generates an **OIDC token** containing workflow context
2. **Fulcio** issues a short-lived certificate (~10 min) – no long-term key management
3. **Cosign** signs the container image with a temporary key
4. **Rekor** records the signing event in an append-only transparency log

Our Verification

- **Rekor**: 28 transparency log entries for amd64 manifest, earliest dated **Feb 5** (append-only, tamper-proof)
- **cosign verify**: Certificate chain traced to `aquasecurity/trivy` GitHub Actions workflow
- Validates both **data integrity** (digest match) and **identity** (issuer + repository)

> Sigstore deep dive: vuls.biz/blog/sigstore-cosign

Q4: Three-Axis Verification – Lessons Learned (4/4)

All three axes confirmed: **our v0.69.1 binary was not tampered with.**

But on **Mar 19**, a second wave compromised GHCR itself – the very reference we used for Axis 1.

Axis	First wave (Feb 28)	Second wave (Mar 19)
Binary Hash (SHA256)	✅ GHCR was trustworthy	⚠️ GHCR itself compromised
Build Timestamp	✅ Predated attack	✅ Still valid
Sigstore (cosign + Rekor)	✅ Verified	✅ Immutable – cannot be retroactively forged

Lesson: SHA256 comparison depends on a trusted reference. When that reference is compromised, only **cryptographic provenance** (Sigstore) and **temporal evidence** (timestamps) remain reliable.

Lifecycle Decision Tree

How `uzomuzo scan` classifies each dependency:

#	Signal	Result
1	Archived / Disabled?	**Yes** → 🔴 EOL-Confirmed
2	Registry EOL? (npm deprecated, PyPI inactive)	**Yes** → 🔴 EOL-Confirmed
3	EOL announced?	**Yes** → 🟠 EOL-Scheduled
4	Recent human commits? **No** → Advisory severity?	HIGH/CRITICAL → ⚫ EOL-Effective
		LOW/MEDIUM → 🟡 Stalled
		None → 🔵 Legacy-Safe
5	Recent human commits? **Yes** → Recent publish?	Yes / VCS-direct → 🟢 Active
		No + low Scorecard → 🟡 Stalled

Q5: Ecosystem Health – Full Breakdown (1/5)

~100 organizations, ~16,000 production OSS packages (FutureVuls, Nov 2025)

Ecosystem	Total	🟢 Active	🔵 Legacy-Safe	🟡 Stalled	🔴 EOL (Confirmed)	🔴 Effective EOL
Java (Maven)	2,358	51.2%	2.9%	23.2%	2.2%	1.0%
Node.js (npm)	8,556	35.3%	15.0%	32.9%	9.0%	7.8%
Python (PyPI)	1,249	41.6%	4.6%	48.6%	3.9%	1.1%
Go (Go Modules)	1,926	44.3%	12.3%	36.9%	5.8%	0.6%
Ruby (RubyGems)	1,379	50.6%	4.2%	42.9%	2.1%	0.2%
PHP (Composer)	477	32.8%	3.0%	56.2%	8.0%	0.0%
.NET (NuGet)	271	60.7%	0.5%	23.4%	15.4%	0.0%
Rust (Cargo)	411	46.7%	12.6%	34.4%	6.3%	0.0%

Source: FutureVuls Foresight Vol.2 Appendix. 50% enterprises (1,000+ employees), ~30% critical infrastructure. CSIRT established at ~55%. Data anonymized, production-only.

Q5: Enterprise Governance – Java & .NET (2/5)

☕ Java (Maven) – Planned Obsolescence

- **EOL (Confirmed + Scheduled): 21.7%** – 2x the overall average (9.4%)
- Driven by **AWS SDK for Java v1** EOL (end of 2025): hundreds of modules expired simultaneously
- Yet **Effective EOL only 1.0%** – enterprise capital enforces strict lifecycle governance
- Active 51.2%: mature ecosystem where abandonment is structurally suppressed

🪟 .NET (NuGet) – Central Planning

- **Active: 60.7%** (highest) – but **Official EOL: 15.4%** (also highest)
- Root cause: .NET runtime's strict release cycle (annual updates, 3-year LTS)
- Forces a binary choice: recompile and evolve (Active) or be declared EOL
- Stalled only 23.4% – the "planned economy" suppresses gradual decay

> Both ecosystems show that **strong governance reduces silent risk** – but shifts the burden to planned migration cycles.

Q5: High Turnover – npm & PHP (3/5)

📦 Node.js (npm) – Micro-Package Churn

- **Effective EOL: 7.8%** – nearly 2x the overall average (4.5%)
- **Official EOL: 9.0%** – `npm deprecate` provides a formal mechanism
- Dual risk: both **unmanaged decay** (Effective EOL) and **managed obsolescence** (Official EOL) at scale
- Micro-package culture: development speed prioritized, thousands of tiny single-function packages

🐘 PHP (Composer) – Desertification

- **Stalled: 56.2%** – 20+ points above overall average (34.6%), the highest of any ecosystem
- Packagist has a sophisticated `abandoned` mechanism – but Official EOL is only 8.0%
- Legacy WordPress plugins + libraries left behind by Laravel's evolution
- Unpaid maintainers choose "quiet death" over formal EOL procedures

> Risk management here requires **continuous automated monitoring** – manual tracking is physically impossible at this scale.

Q5: Silent Abandonment – Python & Ruby (4/5)

🐍 Python (PyPI) – Long-Tail Decay

- **Stalled: 48.6%** – nearly half of all production Python OSS
- Root causes: data science "build and discard" culture + historical lack of deprecation mechanism
- **PEP 792** now provides a standard, but legacy projects silently abandoned before it remain as risk reservoirs
- Risk is **thinly distributed across a vast surface** – impossible to manage manually

💎 Ruby (RubyGems) – Polarized Landscape

- **Stalled: 42.9%** vs **Active: 50.6%** – split between a vibrant "downtown" and an unmaintained "graveyard"
- **Official EOL: only 2.1%** – RubyGems historically lacked a package-level deprecation feature
- High friction to formally declare EOL (requires new version release), so maintainers "quietly walk away"
- Bundler issues no warnings – users unknowingly pull transitive dependencies from the graveyard

> The common thread: **absence of low-friction deprecation mechanisms** leads to silent accumulation of stalled dependencies.

Q5: Structural Resilience – Go & Rust (5/5)

🐿️ Go (Go Modules) – Statistical Illusion

- **Effective EOL: 0.6%** – dramatically below the 4.5% average
- But **Stalled: 36.9%** – above average, a large risk reserve
- The low Effective EOL is partly a **statistical bias**: static linking pulls in many safe base modules, inflating the denominator
- Go bakes dependencies into a single binary – if a stalled dep gets a CVE, **full system rebuild** is required

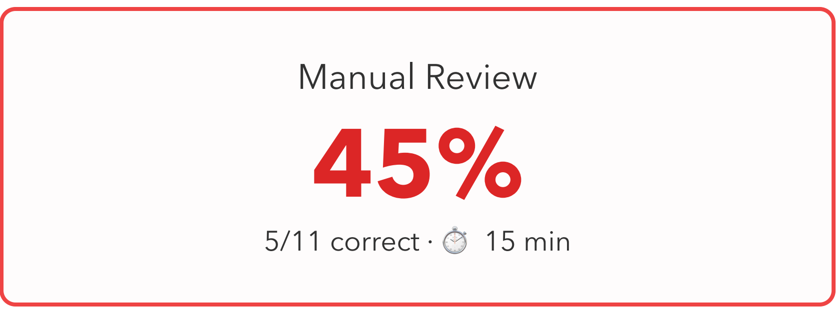
🦀 Rust (Cargo) – Balanced Maturity

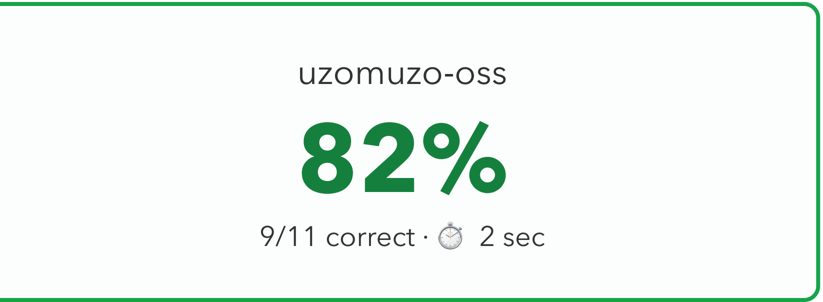
- **Effective EOL: 0%** – shared with .NET and PHP, but Rust achieves this **despite having 34.4% Stalled**
- Stalled does not degrade into dangerous Effective EOL – a structural safety floor
- Structural resilience from **compile-time memory safety guarantees** and a security-conscious community
- Not simply "too new" – the safety floor prevents decay from becoming dangerous

> Go’s surface-level stability hides rebuild risk. Rust’s safety floor is real but ~30% Stalled still warrants monitoring.

Q6: Manual Review vs uzomuzo-oss

11 packages evaluated during a real dependency selection task.





Package	Manual	Actual	Why wrong
`google/uuid`	✅ "Google, fine"	⚠️ Stalled	Brand bias – 2.4yr no commits
`joho/godotenv`	⚠️ "Maintained=0"	✅ Active	Missed pre-release
`go-strftime`	⚠️ "No Scorecard"	✅ Active	No data → worst case
`semver`	⚠️ "No vulns"	🔴 EOL	README says "use v3"

> Manual review doesn't scale. **And it's less accurate.**

Q7: Tool Comparison

Capability	Trivy/Syft	Scorecard	endoflife.date	uzomuzo scan	uzomuzo diet
CVE / Advisory	✅ Full	Partial	–	∆ deps.dev	–
Single-repo health	–	✅ 17 checks	–	✅ via Scorecard	–
Dep tree lifecycle	–	–	–	✅ **7-stage**	–
Batch (5000+)	–	1 repo/run	–	✅	–
Registry EOL	–	–	~450 manual	✅ **automated**	–
Source coupling	–	–	–	–	✅ **tree-sitter**
Removal ranking	–	–	–	–	✅
Unused detection	–	–	–	–	✅

Other tools stop at detection. We go all the way to removal.

"We don't replace Scorecard. We consume it."

Q8: Beyond Go — Real-World Results

diet: 5 languages tested

32 bugs found, 31 fixed (as of 2026-04-13). Go, TypeScript, JavaScript, Python, Java.

1 remaining: Angular HTML template detection. Actively improving.

github.com/future-architect/uzomuzo-oss/issues?q=label:diet-trial

Q9: How We Tested a 76-File Change

`future-architect/vuls#2476` — 76 files · 42 commits · merged

Layer 1: Unit Tests

LLM-generated
Per-function correctness

Layer 2: E2E A/B Test

129 real OSS lockfiles
master vs PR → line-by-line diff

127/129 identical. 2 diffs = 2 improved.

Real lockfiles. No mocks. No hand-written expectations.

Key insight: A/B format = no expected values to write. "Same as before?" is the only question. Automated via `make diff-lockfile`.

Q10: Vendored / Forked Dependencies

"Does uzomuzo handle vendored or forked dependencies?"

Vendored deps

uzomuzo takes **SBOM input**. If the CycloneDX generator (Trivy, Syft, cdxgen, …) detects it, uzomuzo evaluates it.

The PURL points to the original package → lifecycle data is available.

Forked deps

uzomuzo detects forks via **GitHub GraphQL API**:

`isFork` + `parent { nameWithOwner }`

Knows both the fork AND the upstream. LLM skills can suggest evaluating upstream instead.

> The quality of your SBOM determines the quality of uzomuzo's analysis. **Use a good SBOM tool.**

Note: Fork detection requires GitHub token (Path A).

Q11: Why SBOM Input?

"Why not parse lockfiles directly like go.mod?"

❌ Parse lockfiles

20+ formats to maintain
(package-lock.json, poetry.lock,
Cargo.lock, pom.xml, ...)

→ Hundreds of dependencies

SBOM input

Any CycloneDX generator works
(Trivy / Syft / cdxgen / ...)
PURL field is all we need

→ 3 dependencies total

uzomuzo-oss has 3 Go module dependencies.

godotenv · packageurl-go · semver — that's it.

We practice the Code Diet we preach.

go.mod shortcut available for Go-only projects: uzomuzo scan --file go.mod

Q12: Scoring Weight Customization

"Can I adjust the diet scoring formula for my priorities?"

Current formula (fixed):

```
GraphImpact × HealthRisk × (1 - CouplingEffort)
```

diet command

Fixed weights. Deterministic. Reproducible.

Custom weights = YAGNI for now. Open to feedback.

LLM skills

Weights adjustable via **natural language**.

"Prioritize supply chain risk over build speed"

"We care most about Dependabot PR reduction"

> The **tool** gives you a reproducible baseline. The **LLM** adapts to your priorities. Both work together.

Q13: Reporting Status

Project	Type	#	Filed	Status
Trivy	PR	#10484	Apr 5	⏳ Review pending
Grafana	Issue	#121911	Apr 5	✅ **Fix triggered**
Vault	Issue	#31899	Apr 6	⏳ Waiting
Next.js	Discussion	#92479	Apr 7	⏳ Waiting

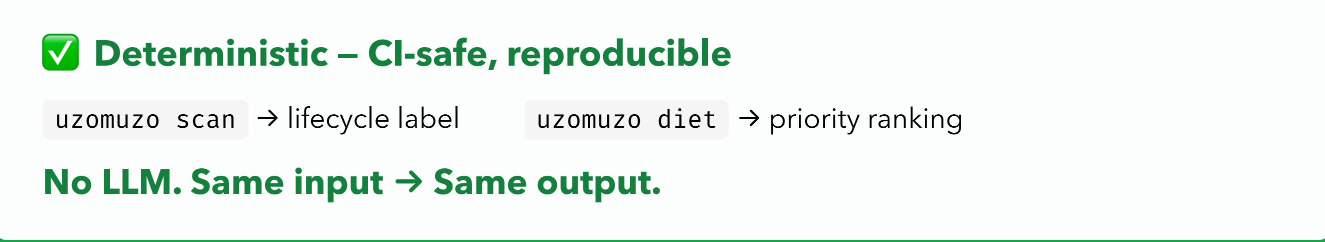
Grafana: our issue triggered a fix

Our issue [#121911](https://github.com/grafana/grafana/issues/121911) (Apr 5) → community PR [#122122](https://github.com/grafana/grafana/pull/122122) filed with `Fixes #121911` (Apr 8) → Grafana team adopted the fix using their internal shared-workflows instead.

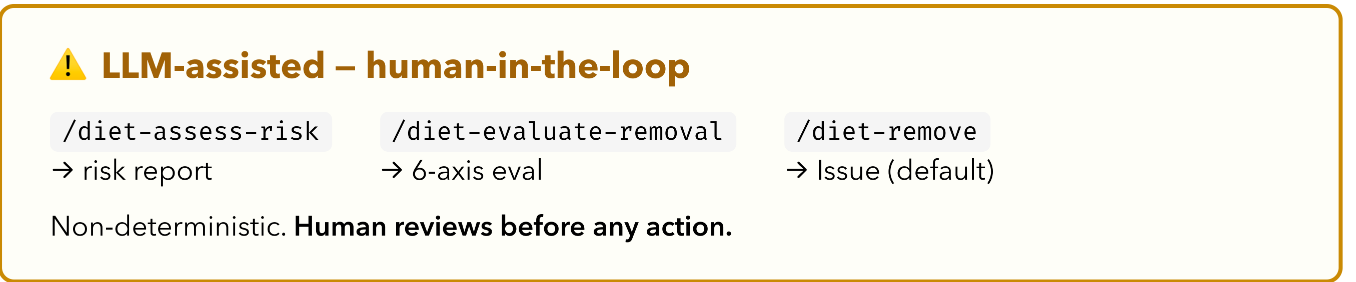
One issue. Three days. The archived Action is being replaced across Grafana's CI.

Detection is fast. Adoption takes time. **But the cycle works.**

Q14: Where the LLM Fits



↓ human decides when to use ↓



"The LLM is a research assistant, not a decision maker."

Q15: SLSA / Sigstore Relationship

"How does uzomuzo relate to SLSA and Sigstore?"

uzomuzo includes **Build Integrity** grading – supply chain tamper resistance per dependency:

Hardened	Moderate	Weak
≥ 7.5/10	2.5 - 7.4	< 2.5

Signals used: Dangerous Workflow (Critical) · Branch Protection (High) · Code Review (High) · Token Permissions (High) · Binary Artifacts (High) · Pinned Dependencies (Medium)

SLSA / Attestation: Evaluated when available, but adoption is still low (~3%). Primary signals come from Scorecard.

> **SLSA answers:** "Was this binary tampered with?" **uzomuzo answers:** "Is this project still maintained?" **Different questions. Both needed.**

Q16: Start Today

Level 1

30 sec

```
uzomuzo scan --file go.mod
uzomuzo scan https://github.com/your/repo
```

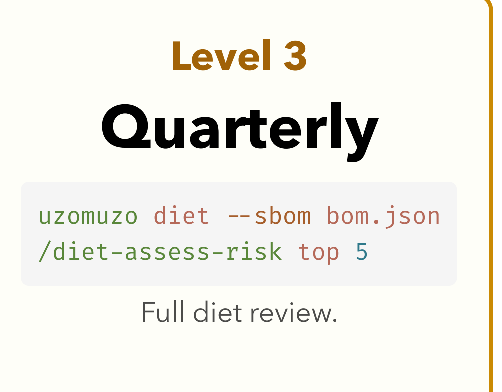
Tonight at the hotel.

Level 2

5 min

```
trivy fs . --format cyclonedx \
  | uzomuzo scan --sbom - \
  --fail-on eol-confirmed
```

CI gate. Any language.



Install: `go install github.com/future-architect/uzomuzo-oss/cmd/uzomuzo@latest`