

# PropTracker API Security

## security-posture.yml — Complete Walkthrough

GitHub Actions | GHAS | AI-Powered Vulnerability Detection

This workflow is your **always-on security camera** for the PropTracker API. It runs every 15 minutes with no human involvement and performs four actions: reads what GitHub already knows, probes the live API for runtime vulnerabilities, uploads findings to the Security tab, and creates GitHub Issues so a human can approve AI-generated fixes.

## Section 1 -- When Does the Workflow Run?

| Trigger               | When                   | Why  |
|-----------------------|------------------------|--|
| schedule */15 * * * * | Every 15 minutes, 24/7 | Continuous monitoring -- catches issues even with no code activity |
| push -> main          | Every code push        | Instant regression check -- new commit? Checked immediately        |
| pull_request          | Every PR opened        | Gate: blocks merge if new vulnerabilities are introduced           |
| workflow_dispatch     | Manual, on demand      | Incident review or customer demo -- run a full audit right now     |

**Manual inputs (workflow\_dispatch):** When triggered by hand you can choose the target environment (staging / production / local) and whether failing vulnerabilities should block the check (fail\_on\_high). This makes the same workflow usable as both a live gate and a one-off audit tool.

## Section 2 -- Permissions (Principle of Least Privilege)

Every GitHub Actions workflow runs with a scoped **GITHUB\_TOKEN**. By default it can only read the repository. We grant exactly three extra permissions -- nothing more:

| Permission      | Level | Used For   |
|-----------------|-------|--|
| security-events | write | Upload SARIF results to Security > Code Scanning tab       |
| issues          | write | Create GitHub Issues for new vulnerability findings        |
| contents        | read  | Check out source code (needed for SARIF file-path mapping) |

## Section 3 -- The 9 Steps Explained

| <b>Step 1</b>          | <b>Checkout Repository</b>   | <b>Infrastructure</b> |
|------------------------|--|-----------------------|
| <b>What it does:</b>   | Downloads the repository source code into the GitHub Actions runner.   |                       |
| <b>Why it matters:</b> | The SARIF upload step (Step 6) needs actual file paths from the source tree so GitHub can link each finding directly to the correct file and line in the code editor. Without checkout, SARIF results would be orphaned with no source context.  |                       |
| <b>Key code:</b>       | <code>uses: actions/checkout@v4</code>   |                       |
| <b>Step 2</b>          | <b>Read GHAS Code Scanning Alerts</b>  | <b>GHAS Static</b>    |
| <b>What it does:</b>   | Calls the GitHub API: GET /repos/{owner}/{repo}/code-scanning/alerts. Returns every open alert found by CodeQL -- GitHub's semantic static code analyser.  |                       |
| <b>Why it matters:</b> | Before running our own probes we want to know what GitHub already knows. This prevents duplicate findings and lets us build a combined picture: static analysis + runtime probes in one unified report. CodeQL finds: SQL injection in source code, insecure deserialization, path traversal, hardcoded credentials. |                       |
| <b>Key code:</b>       | <code>gh api /repos/\$REPO/code-scanning/alerts?state=open</code>  |                       |
| <b>Step 3</b>          | <b>Read Secret Scanning Alerts</b>   | <b>GHAS Secrets</b>   |
| <b>What it does:</b>   | Calls GET /repos/{owner}/{repo}/secret-scanning/alerts. GitHub continuously monitors every git push for patterns matching known secret formats: JWT secrets, API keys, AWS credentials, database connection strings, and 200+ other types.   |                       |
| <b>Why it matters:</b> | A leaked JWT_SECRET lets an attacker forge tokens and authenticate as any user. A leaked DATABASE_URL gives direct database access. Secret scanning catches these within seconds of a push -- faster than any code review. push_protection_bypassed shows if a developer received a warning and pushed anyway.       |                       |
| <b>Key code:</b>       | <code>gh api /repos/\$REPO/secret-scanning/alerts?state=open</code>  |                       |

| <b>Step 4</b>          | <b>Read Dependabot Alerts</b>   | <b>Supply Chain</b> |
|------------------------|---|---------------------|
| <b>What it does:</b>   | Calls GET /repos/{owner}/{repo}/dependabot/alerts. Checks every npm/pip/Maven package against the GitHub Advisory Database -- a continuously updated list of CVEs.  |                     |
| <b>Why it matters:</b> | 80% of modern application code is open-source dependencies. A vulnerability in an npm package you use is just as dangerous as one you wrote yourself. Example: CVE-2021-44228 (Log4Shell) -- every org using log4j was vulnerable the day it was published. Dependabot tells you the CVE ID, the vulnerable version range, and the patched version to upgrade to. |                     |
| <b>Key code:</b>       | <code>gh api /repos/\$REPO/dependabot/alerts?state=open</code>  |                     |
| <b>Step 5</b>          | <b>Live Runtime API Probes (The Unique Step)</b>  | <b>Runtime</b>      |
| <b>What it does:</b>   | Makes REAL HTTP requests to the running API to verify vulnerabilities exist at runtime -- not just in code. Four probes: BOLA (can Bob read Alice's job?), Broken Auth (is an expired JWT accepted?), Rate Limit (do 20 rapid requests trigger a 429?), SQL Injection (does the search endpoint return extra rows with OR 1=1 payload?).                          |                     |
| <b>Why it matters:</b> | Static analysis reads code but cannot know about runtime configuration -- environment variables, middleware loading order, framework behaviour, or infrastructure settings. Runtime probes catch what static analysis misses. If ignoreExpiration:true is set as an env var, CodeQL cannot see it -- but our probe will catch it immediately.                     |                     |
| <b>Key code:</b>       | <code>urllib.request.urlopen(req) # Pure Python -- no curl needed in CI</code>  |                     |
| <b>Step 6</b>          | <b>Generate SARIF File</b>  | <b>SARIF</b>        |
| <b>What it does:</b>   | Converts runtime probe findings into SARIF 2.1.0 format -- an open JSON standard for representing security tool output. Each finding maps to: rule ID, severity level, affected file path, and line number.   |                     |
| <b>Why it matters:</b> | Without SARIF, probe results only exist in the Actions log. With SARIF, GitHub processes results and makes them appear in Security > Code Scanning with clickable file links, PR annotations, and org-wide aggregation in GitHub Security Overview. This turns our custom scripts into first-class GHAS tools.  |                     |
| <b>Key code:</b>       | <code>sarif = {"version": "2.1.0", "runs": [{"tool": {...}, "results": [...]}]}</code>  |                     |

| Step<br>7       | Upload SARIF to GitHub Security Tab   | GHAS Upload |
|-----------------|---|-------------|
| What it does:   | Uses the official <code>github/codeql-action/upload-sarif@v3</code> action to send the SARIF file to GitHub. The <code>if: always()</code> condition ensures upload happens even when vulnerabilities are found -- we WANT them in the Security tab.  |             |
| Why it matters: | The action handles gzip compression, base64 encoding, chunked upload for large files, retry on failure, and correctly sets the commit SHA so findings are pinned to the right version of the code. The <code>category: runtime-api-probe</code> parameter groups these findings separately from CodeQL results so engineers can filter by source.               |             |
| Key code:       | <pre>uses: github/codeql-action/upload-sarif@v3 with: sarif_file: /tmp/results.sarif category: runtime-api-probe</pre>  |             |
| Step<br>8       | Create GitHub Issues for New Findings   | Issues      |
| What it does:   | For every runtime finding, checks if a GitHub Issue with that vulnerability ID is already open. If not, creates one. The issue includes full evidence, the AI-recommended fix, and a table showing three response options: Apply Fix, Accept Risk, or Investigate.  |             |
| Why it matters: | Deduplication prevents 1,344 duplicate issues after 2 weeks of 15-minute runs. The issue stays open until the fix is merged. Adding the <code>apply-fix</code> label triggers a separate workflow ( <code>apply-fix.yml</code> ) that automatically opens a Copilot-generated PR -- closing the loop from detection to remediation without ever leaving GitHub. |             |
| Key code:       | <pre>gh issue create --label<br/>'security,ai-recommendation,vuln:VULN-1-BOLA,critical'</pre>   |             |
| Step<br>9       | Enforce: Fail on Critical/High Findings   | Gate        |
| What it does:   | If the workflow runs on a pull request and <code>fail_on_high</code> is true (the default), the workflow exits with code 1 -- failing the required check. GitHub branch protection prevents merging until all required checks pass.   |             |
| Why it matters: | Without enforcement, security findings are informational only -- engineers can ignore them. By making this a required PR check, no vulnerable code can be merged without an explicit admin override. This is the 'security as code' principle: policy enforced automatically, not by manual review.   |             |
| Key code:       | <pre>if [ "\$COUNT" -gt "0" ]; then exit 1; fi # Blocks the PR merge</pre>  |             |

## Section 4 -- End-to-End Data Flow

Here is what happens from the initial trigger all the way to a merged fix:

| # | Event                                     | Actor                  | Output                              |
|---|---|------------------------|-------------------------------------|
| 1 | Cron fires (every 15 min)                 | GitHub Scheduler       | Workflow starts                     |
| 2 | Read GHAS APIs                            | GitHub Actions         | JSON alert lists saved to /tmp/     |
| 3 | HTTP probes hit live API                  | Python urllib          | runtime_findings.json               |
| 4 | Python converts findings to SARIF         | Python                 | results.sarif                       |
| 5 | SARIF uploaded                            | CodeQL Action          | Findings appear in Security tab     |
| 6 | New finding -> GitHub Issue created       | gh CLI                 | Issue with apply-fix label option   |
| 7 | Engineer adds apply-fix label             | Human                  | apply-fix.yml triggered             |
| 8 | Copilot generates patch -> PR opened      | apply-fix.yml          | PR with fix ready to review         |
| 9 | <b>PR merged -&gt; vulnerability gone</b> | <b>Human + Copilot</b> | <b>Next run: 0 findings (clean)</b> |

## Section 5 -- GitHub Actions + GHAS vs Kong Real-Time Monitoring

| Capability                 | Kong API Gateway   | GitHub Actions + GHAS              |
|----------------------------|--------------------|------------------------------------|
| Runtime request inspection | Every request      | <b>Every 15 min probes</b>         |
| Static code analysis       | Not supported      | <b>CodeQL -- semantic analysis</b> |
| Secret detection in git    | Not supported      | <b>Secret Scanning</b>             |
| Dependency CVE tracking    | Not supported      | <b>Dependabot</b>                  |
| AI-generated fix PRs       | Not supported      | <b>Copilot + apply-fix.yml</b>     |
| Cost                       | \$\$\$ Licence fee | <b>Included in GHAS</b>            |
| Enforcement on PRs         | Post-deploy only   | <b>Blocks merge before deploy</b>  |
| Unified Security tab view  | Separate dashboard | <b>GitHub Security Overview</b>    |

**Key differentiator:** Kong stops attacks at the gateway *after* your API is already deployed and under attack. GitHub Actions + GHAS catches vulnerabilities **before deployment** -- in the code, in the PR, in the dependency manifest. The runtime probes bridge the gap by testing the live API on a schedule, giving you coverage at every layer of the stack with no additional licence cost.

## Section 6 -- Key YAML Concepts in This File

\$GITHUB\_STEP\_SUMMARY

A special environment variable pointing to a file on the runner. Anything written to this file is rendered as markdown in the Actions run summary page -- the rich dashboard you see when you click on a workflow run. Supports tables, code blocks, and Mermaid charts.

#### **id: code\_scanning -> steps.code\_scanning.outputs.\***

Giving a step an id: field allows later steps to reference its outputs. We use echo 'key=value' >> \$GITHUB\_OUTPUT to pass alert counts between steps without needing temporary files.

#### **if: always()**

Normally, if a step fails, subsequent steps are skipped. always() overrides this -- the SARIF upload runs even when vulnerabilities are found, because we want them in the Security tab regardless of whether the overall workflow passes or fails.

#### **python3 - <<'PYEOF' ... PYEOF**

A shell heredoc pattern for running multi-line Python scripts inline in a YAML run: block. Single-quoted 'PYEOF' prevents bash from expanding shell variables inside the Python code -- essential when the Python script itself uses variable notation.

#### **SARIF 2.1.0**

Static Analysis Results Interchange Format -- an open JSON standard. GitHub's Security tab natively understands SARIF, turning any custom script output into first-class security findings with file links, PR annotations, and org-wide aggregation in the GitHub Security Overview dashboard.

#### **gh api ...**

The GitHub CLI makes authenticated API calls using the GITHUB\_TOKEN automatically -- no separate credential management. The --jq flag applies a jq filter to shape the JSON response directly in the shell command, avoiding the need for a separate parsing step.

#### **category: runtime-api-probe**

When uploading SARIF, the category field groups findings by their source tool. This lets engineers filter the Security tab: view only CodeQL results, only runtime probe results, or all findings together. Without a category, uploads overwrite each other.