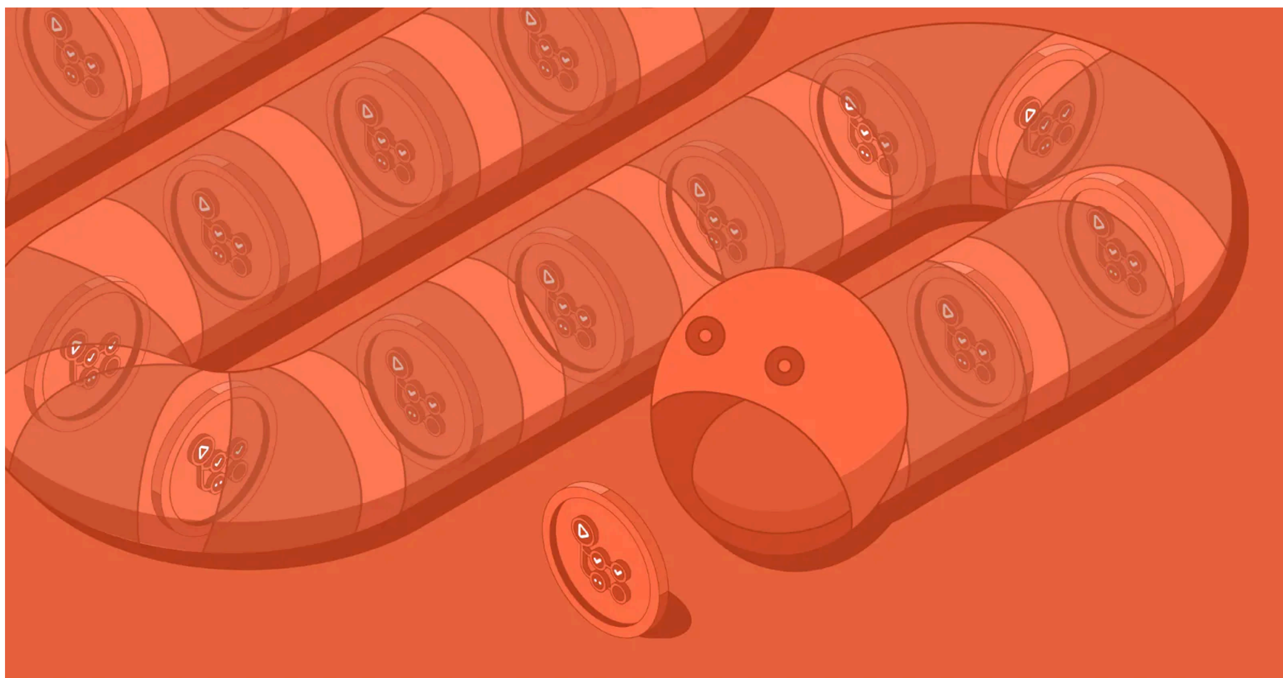
The GitHub Actions Worm: Compromising GitHub Repositories Through the Actions Dependency Tree

By By Asi Greenholts

Published: 2023-09-14 · Archived: 2026-04-02 12:02:10 UTC

Learn how a novel attack vector in GitHub Actions allows attackers to distribute malware across repositories using a technique that exploits the actions dependency tree and puts countless open-source projects and internal repositories at risk. Get an in-depth look at the attack vectors, technical details and a real-world demo in this blog post highlighting our latest research.



As the premier platform for hosting open-source projects, GitHub's popularity has boosted the popularity of its CI/CD platform — GitHub Actions. This popularity, however, extends beyond the DevOps community to attract hackers eager to exploit the platform's expanding attack surface.

Initial Attack Vectors

In recent supply chain attacks, we see attackers repeatedly executing the same scheme, ultimately compromising a repository or a software library and infecting it with a malicious payload targeting its direct dependents. The payload tries to steal secrets or create a reverse shell, whether running in pipelines or production environments.

To achieve control or write permissions on a repository, attackers draw from a range of established techniques:

1. Repojacking

When a repository moves to a new owner (organization or user), or the owner changes its name, GitHub automatically redirects requests sent to the old repository name to the new owner and repository. But if the owner is deleted, an attacker can register the previous name on GitHub and create a repository containing malicious code. Use of the previous repository name disables GitHub's automatic redirection, and consumers consuming this project unknowingly consume the malicious repository.

To protect against repojacking, GitHub employs a security mechanism that disallows the registration of previous repository names with 100 clones in the week before renaming or deleting the owner's account. The 100-clone security measure, though, often proves inadequate for repositories hosting actions. When a GitHub Actions workflow uses an action, it downloads a zip of the repository via the GitHub API, bypassing the clone count. In other words, the downloaded zip file doesn't contribute to the repository's tally of clones.

2. NPM Package Maintainer Email Hijacking

Actions written in JavaScript usually involve dependencies maintained by developers who typically use email addresses to sign into NPM. Should attackers acquire a maintainer's email domain, and the maintainer's NPM account lack two-factor authentication (2FA), the attackers could reset the password. This would allow the attackers to create a malicious package version that, when used by the action, will enable the attackers to execute malicious code within GitHub workflows as the action runs.

3. GitHub Actions Command Injection

Workflows receive context about the triggering event, including information like issue title, pull request title, and commit message. Since attackers can control some of these fields, developers should treat them as untrusted input. Workflows that use the untrusted input in bash commands — when using bash's command substitution, for example — can be vulnerable to command injection.

Other attack methods used to compromise repositories include:

1. Dependency confusion
2. Public-PPE
3. Compromise a maintainer's access token or credentials
4. Create a pull request with hidden malicious code in hopes that project maintainers will merge it

Regardless of the initial attack vector, once attackers gain a foothold in an action's repository, they set out to create a worm. The worm will allow the attackers to directly infect with malware any GitHub Actions workflow consuming this action. But how can the attackers extend their reach and infect more repositories?

How Actions Depend on Actions

After creating the worm, the next step involves finding a path for it to spread. For a GitHub Actions worm, the path travels from action to action, which could involve any of the three types of actions — JavaScript, Docker, or Composite. Additionally, actions can depend on actions in one of two ways. The first way uses composite actions, which combine multiple workflow steps within one action. All action types, though, use an action.yml file to define the action's inputs, outputs and main entrypoint.

```yaml
name: 'Nice checkout'
description: 'Greet someone before checkout'
inputs:
  who-to-greet:  # id of input
    description: 'Who to greet'
    default: 'World'
outputs:
  random-number:
    description: "Random number"
    value: ${{ steps.random-number-generator.outputs.random-number }}
runs:
  using: "composite"
  steps:
    - run: echo Hello ${{ inputs.who-to-greet }}.
      shell: bash
    - uses: actions/checkout@v3
    - id: random-number-generator
      run: echo "random-number=$(echo $RANDOM)" >> $GITHUB_OUTPUT
      shell: bash
```

Figure 1: Action.yml file, the metadata file of actions

In figure 1, a sample action.yml file instructs a composite action to run actions/checkout, a dependency of the composite action.

The second way actions can depend on other actions is through the action's CI/CD pipeline. It's common to see actions that use GitHub Actions workflows to build and test their code.

```yaml
on:
  workflow_dispatch:
  push:
    branches:
      - main
  pull_request:

jobs:
  check-links:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v3

      - name: Setup node
        uses: actions/setup-node@v3
        with:
          node-version: 16.13.x
          cache: npm

      - name: Install
        run: npm ci

      # Creates file "$/files.json", among others
      - name: Gather files changed
        uses: trilom/file-changes-action@a6ca26c14274c33b15e6499323aac178af06ad4b
        with:
          fileOutput: 'json'
```

Figure 2: An example workflow file used as a CI of an action

We can see that the workflow file uses the trilom/file-changes-action action during its run, which makes it an implicit dependency of the action.

Note that this dependency action isn’t used as part of the action but only in the action’s workflow, a component process in the proper flow of the CI process.

These two ways for actions to depend on other actions form a tree of dependencies that interconnect the actions in the [GitHub Marketplace](https://github.com/marketplace).

We can now use this knowledge to parse action.yml files that define actions and CI workflows of actions. In this, we can identify actions dependant on other actions and create the GitHub Actions dependency tree over a Neo4j graph:

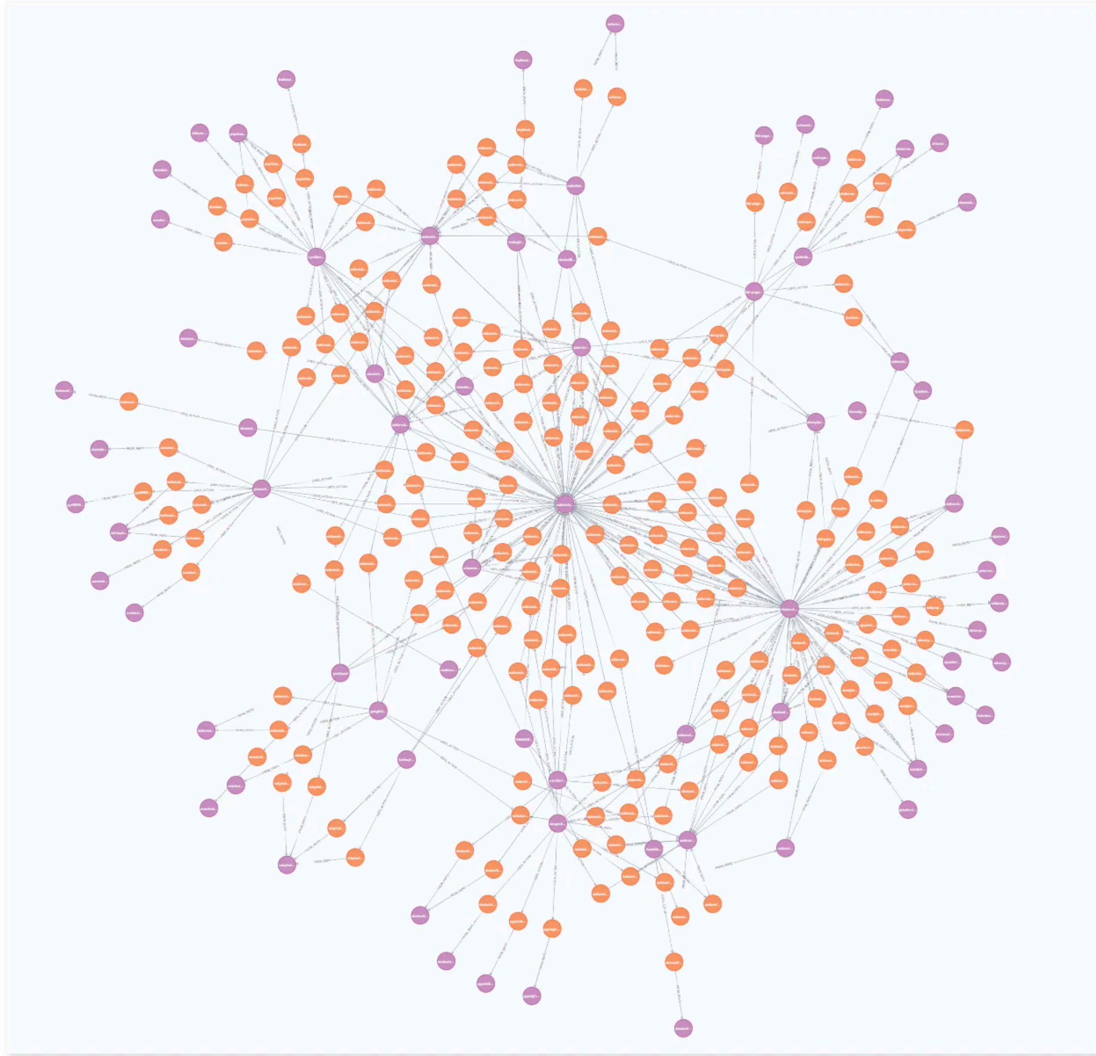


Figure 3: GitHub Actions dependency tree over a Neo4j graph referred to as "The Flower"

A purple node represents a repository. An orange node represents an instance of an action used in another action's workflow or an action directly referenced in an action configuration file (action.yml).

In figure 3, we see several repos (purple) with a workflow. Each workflow uses an action (orange), and the action is hosted in another repo (purple). The action stored in this repository might have its own workflow, which uses another action (orange), and so on.

How Compromised Actions Infect Dependency Actions

We demonstrated how actions can be dependent on other actions. Let's now explore how attackers can abuse these dependencies to spread their worm.

Dumping Secrets from a Runner's Memory

To secure secrets in GitHub Actions, you can use its encrypted secrets feature, which allows you to define secrets in the organization, repository or environment settings.

When a job starts, the GitHub Actions runner receives all the secrets used in the job. Because the runner receives the secrets when the job starts, we can [dump the runner's memory to reveal all secrets defined in the job]() even before they're used. This means that no matter when we achieve code execution by compromising an action used in the job, we can read from memory all secrets referenced in the job.

Also noteworthy, jobs can use a secret called GITHUB_TOKEN — uniquely generated for each workflow run — to allow jobs to authenticate and use GitHub's API against the repository. Is the GITHUB_TOKEN as accessible as other secrets? We'll soon find out.

```
on:
  workflow_dispatch:

jobs:
  first:
    runs-on: ubuntu-latest
    steps:
      - run: echo ${{ secrets.FIRST_SECRET }}
  second:
    runs-on: ubuntu-latest
    steps:
      - run: |
          sudo apt-get install -y gdb
          sudo gcore -o k.dump "$(ps ax | grep 'Runner.Listener' | head -n 1 | awk '{ print $1 }')"
          grep -Eao '"[^"]+":\{"value":"[^"]*","issecret":true\}' k.dump* | base64
      - run: echo ${{ secrets.SECOND_SECRET }}
```

Figure 4: Workflow file that dumps the memory of the runner

The workflow seen in figure 4 contains two jobs. Each job runs on a different runner and uses a different secret. In the second job, we see that the second step dumps the runner's memory to retrieve its secrets.


```
“second_secret”:{“value”:“imasecret”,”issecret”:true}
“system.github.token”:{“value”:“ghs_uTzaHn7ntsarT3RUE7dsGx3Qa4689V2Jaoq0”,”issecret”:true}
```


Figure 5: Decoded memory dump

In the decoded base64 that the second job prints, you'll notice two interesting details:

1. We don't see the FIRST_SECRET secret because it's used in a different job and runs on a different machine. Note the possibility on self-hosted runners for two jobs to run on the same runner if the runner isn't ephemeral.
2. We see the GITHUB_TOKEN secret, although we didn't reference it in the workflow file. This token, in other words, can be accessed from any job in the workflow, even if the workflow doesn't reference it.

Incidentally, we often see a GitHub personal access token (PAT) stored as a secret and used by steps in the workflow to perform tasks against the repository.

Overriding Action's Code

To understand how we can infect an action's repository, we need to understand how actions are used.

The common format for calling an action follows {owner}/{repo}@{ref}. The "ref" key has three forms:

1. Reference a commit hash.

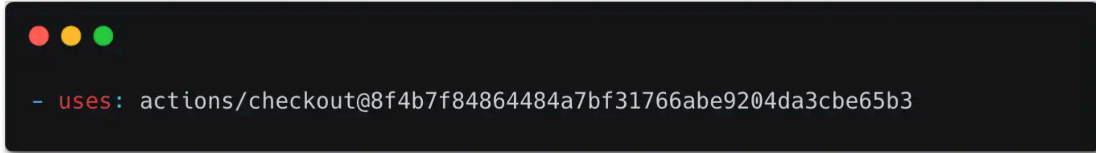


Figure 6: Calling an action using a commit hash

2. Reference a branch.

Figure 7: Calling an action using branch name

3. Reference a tag.

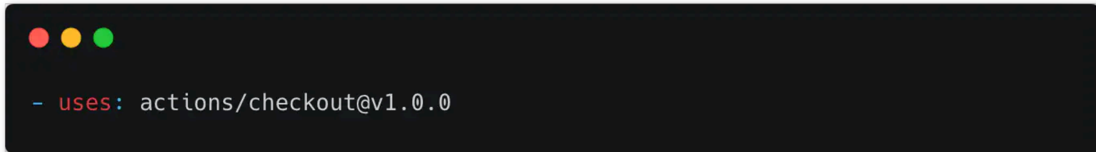


Figure 8: Calling an action using a tag

We can use the secrets exfiltrated in the flow to infect the repository with malicious code. Overwriting a commit while keeping its hash the same isn't possible, so we can't abuse a commit hash reference. We still have two options:

1. **Infecting by pushing code to a branch**: We can use the GITHUB_TOKEN or PAT used in a job to push malicious code to a branch if the token is granted with the *contents:write* permission.
2. **Infecting by creating a tag**: We can use the GITHUB_TOKEN or PAT used in a job to create a malicious tag or override an existing tag to infect dependent repositories referencing the action by a tag.

Successfully pushing code also depends on the GITHUB_TOKEN permissions, branch protection rules and protected tags configured in the repository.

You can calculate the GITHUB_TOKEN permissions of a workflow, even before running it. Until recently, all GitHub Actions workflows had default read and write permissions against their repositories. GitHub changed the

default configuration in February 2023 to read permissions on the contents, packages and metadata scopes. This means that most repositories now have default write permissions.

Besides permissions granted in the repository setting of the GITHUB_TOKEN, permissions can be overwritten by configuring them inside the workflow file.

```
name: Feature request triage bot

on:
  schedule:
    # Run at 14:00 every day
    - cron: '0 14 * * *'

# Declare default permissions as read only.
permissions:
  contents: read

jobs:
  feature_triage:
    if: github.repository == 'angular/angular'
    runs-on: ubuntu-latest
    steps:
      - uses: angular/dev-infra/github-actions/feature-request@0109d498b0f6aae418ed4924a5e5c65695f0ac61
        with:
          angular-robot-key: ${{ secrets.ANGULAR_ROBOT_PRIVATE_KEY }}
```

Figure 9: Workflow file limiting the contents permission of the GITHUB_TOKEN

While a real worm doesn't need to know the permissions — it simply tries to infect any repository it encounters — we only created a static analysis as part of our research and wanted to discover the permissions granted to the GITHUB_TOKEN in each workflow. To do that, we examined the workflow run log.

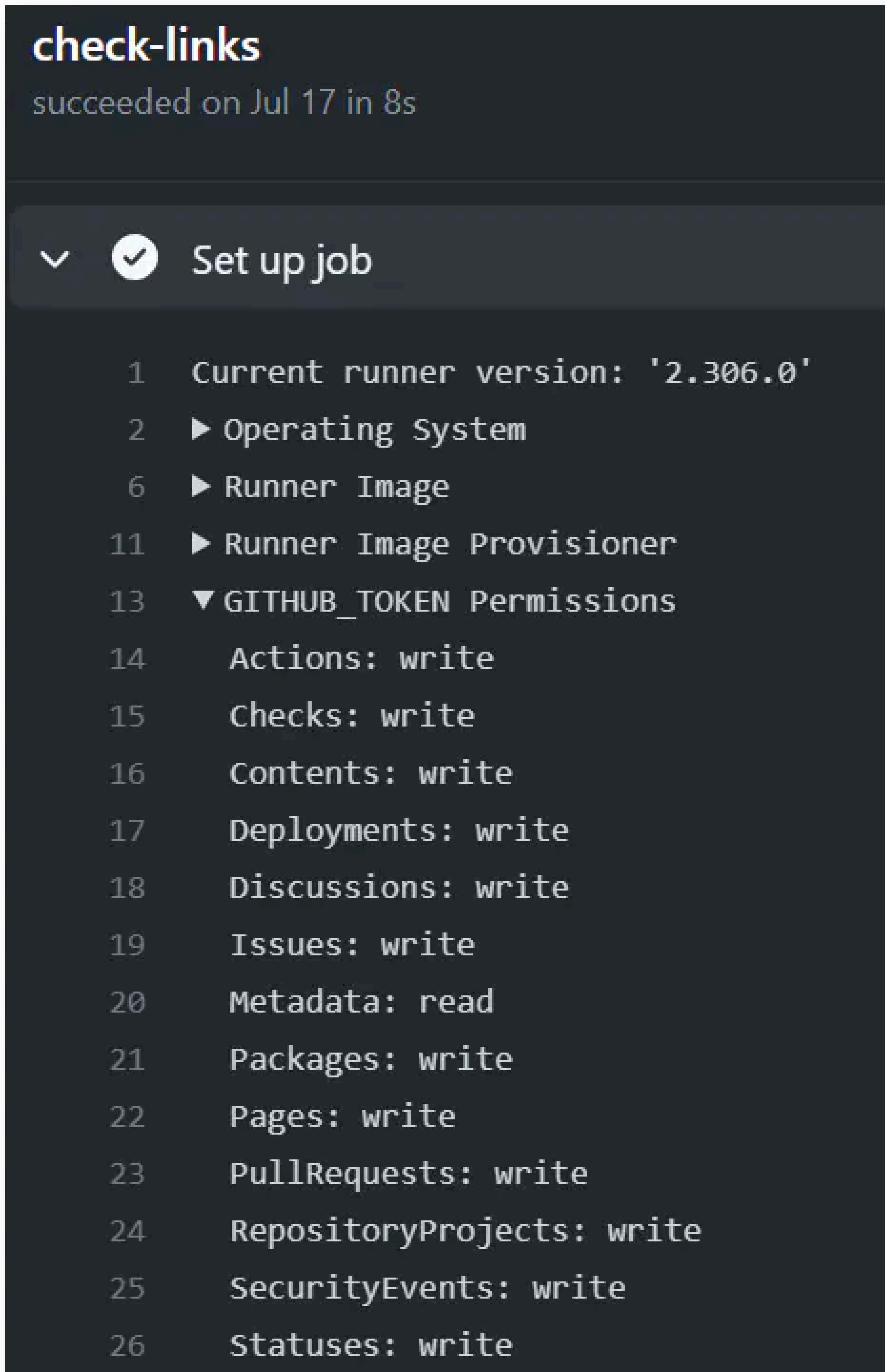


Figure 10: Sample workflow run

We can see the write permission on the contents scope — and many other granted permissions — and use them to push code to the repository.

Sounds Like a Worm, Right?

Let’s pause and look at what we have so far.

1. We know how we can find our initial attack vector to compromise the first action's repository.
2. We also know how we can leverage the compromised action's repository to infect a dependent repository's code — by pushing malware to one of its branches or overwriting an existing tag.

Imagine what would happen if we did this recursively. We have the potential to create a GitHub Actions worm across the Actions dependency tree.

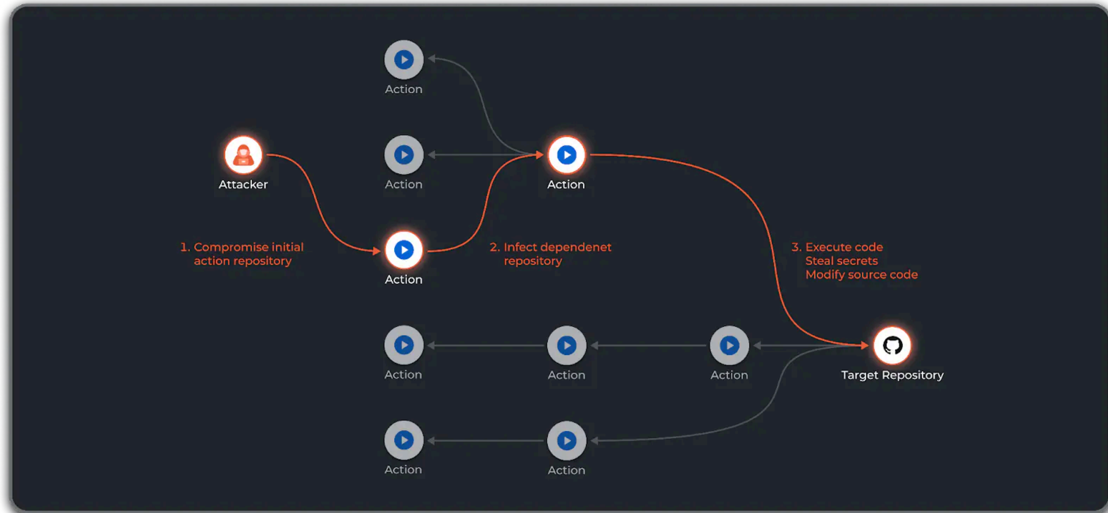


Figure 10: Flow of a GitHub Actions worm

Finding Attack Graphs at Scale

Now that we have all the pieces of the puzzle, we can start scanning targets for dependencies vulnerable to exploitation.

First, we gathered two sets of targets:

- 4,700 repositories that use GitHub Actions out of GitHub's top 10,000 repositories by star count.
- 7,200 repositories that use GitHub Actions out of 32,000 repositories of companies that have a bug bounty program.

To create a Neo4j graph, we then automated a process that accomplished the following for each target:

1. Clone the repository and create a node for it.
2. Check if any of the potential initial attack vectors apply to this repository.
3. Parse the actions' workflows and action.yml files to find all used actions and enrich nodes with metadata, such as secret names accessible to the action, version in use and the permission of the GITHUB_TOKEN on the repository contents scope.
4. Recursively undertake the above steps for each discovered action.

Figure 11 depicts an attack graph of repositories we can't disclose. In this graph, you can see two viable initial attack vectors to execute on two repositories. These two repositories are actually the same repository but moved

from one org to another. By attacking the initial repository, we can directly infect 18 actions that depend on it. From infecting these 18 actions, we can infect 72 of the target repositories.

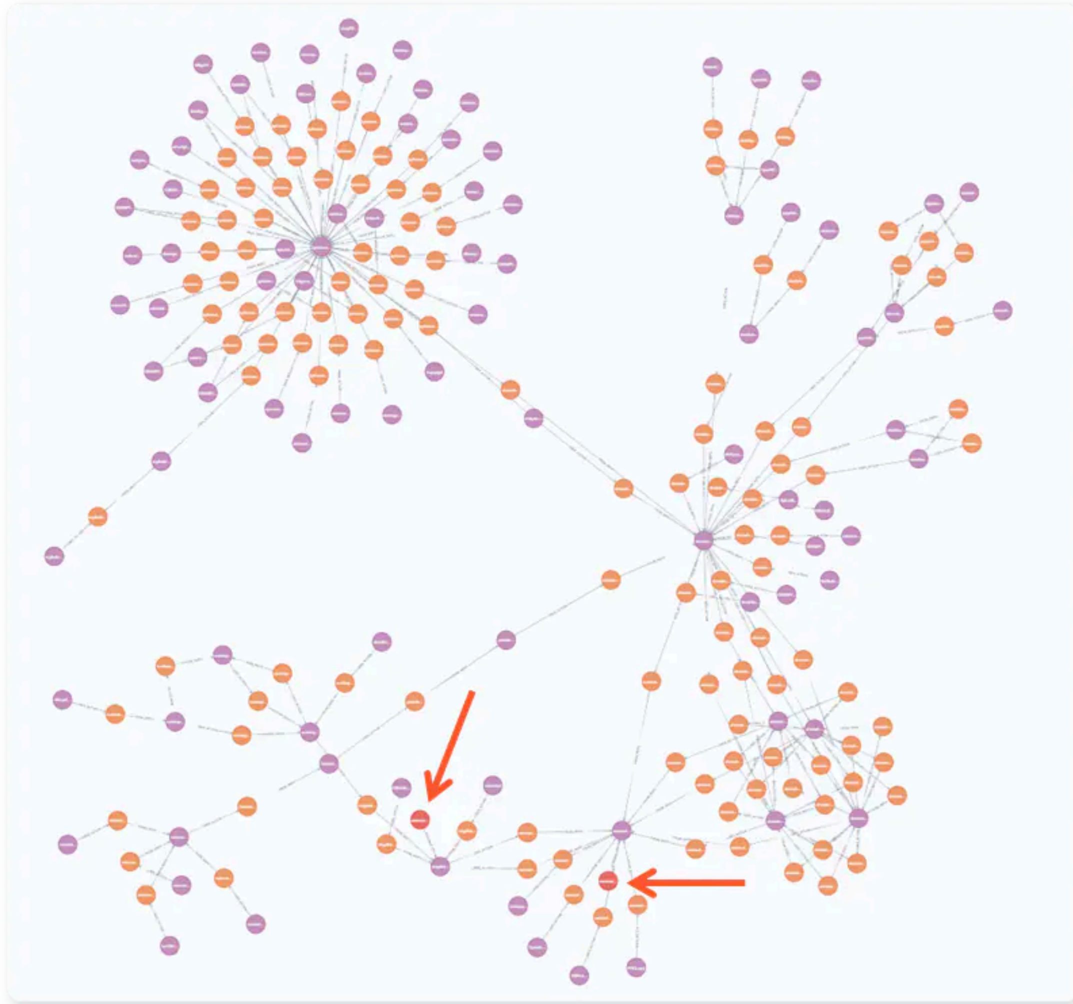


Figure 11: Purple/repository, orange/action usage, red/initial attack vector

The scale of this infection chain is larger than presented in the graph. We'll explain why through a practical, real-world example.

Public Disclosure

We reported the issues we found to all vulnerable projects we could contact. Hangfire and Veracode allowed us to publicly disclose their cases.

In the figure 12 attack graph, you can see the HangfireIO/Hangfire (8.4k ⭐) public GitHub repository at the bottom-left side. This repository used two actions in one of its workflow files: veracode/veracode-pipeline-scan-results-to-sarif and papeloto/action-zip. The veracode action also used papeloto/action-zip in its workflow file.

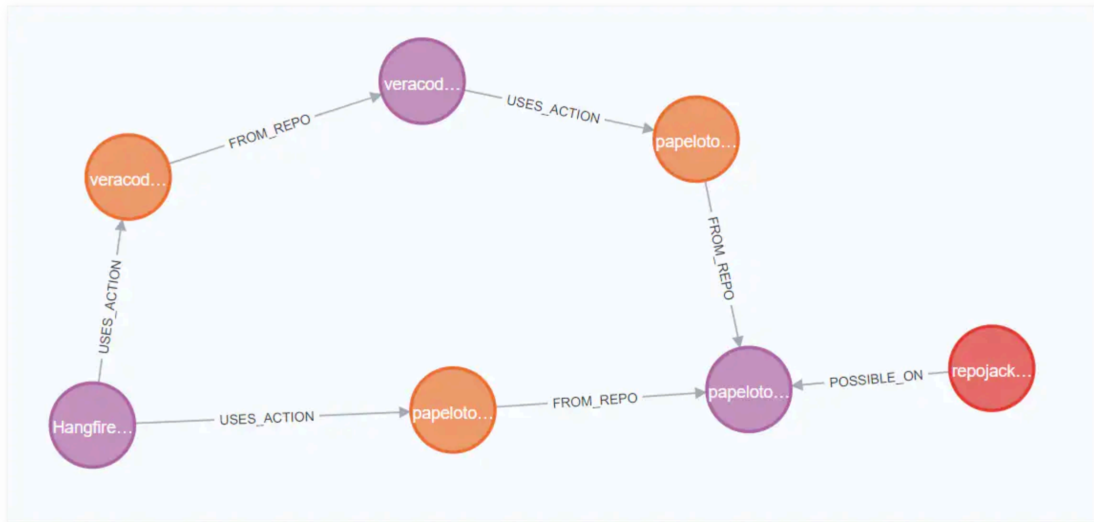


Figure 12: Attack graph

The papeloto/action-zip action was moved from its original repository to the **vimtor** organization, and the **papeloto** organization was available for registration, making the action-zip repository vulnerable to repojacking. Our team registered the organization to prevent malicious actors from exploiting this vulnerability.

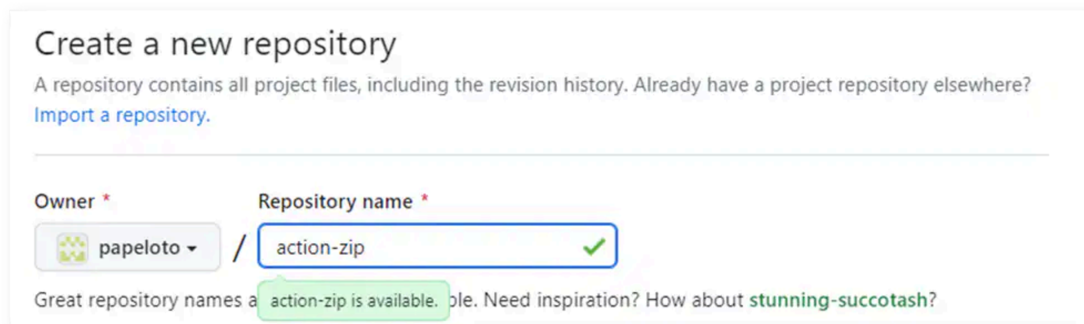


Figure 13: Ownership of the papeloto organization and the ability to create the action-zip repository

By repojacking this repository, we successfully attacked the HangfireIO/Hangfire repository directly. Infecting the veracode/veracode-pipeline-scan-results-to-sarif repository achieved the same ends.

The potential scale of this vulnerability is massive. We can attack the veracode repository's 1,600 dependents, represented in figure 14 by orange circles. The action-zip action has about 600 dependents that will execute our malicious code. But that's not all.

The Hangfire repository deploys a NuGet package that has 9,400 daily downloads we can attack. Additionally, the dependents have dependents, and the NuGet package consumers likely have their dependents, so the infection chain continues *ad infinitum*. That's still not all, though.

We're talking only about the public repositories we analyzed in the GitHub public ecosystem. A real worm would run on a vast number of private repositories — and impose an immediate impact. If the worm encounters a private

repository granting minimal read-only permissions to the GITHUB_TOKEN, it could steal source code. If the repository's code can be modified, the attack escalates. *Disastrous* best describes the scope of this attack scenario.

Now take all of that and imagine multiple possible attack graphs like the one in figure 14.

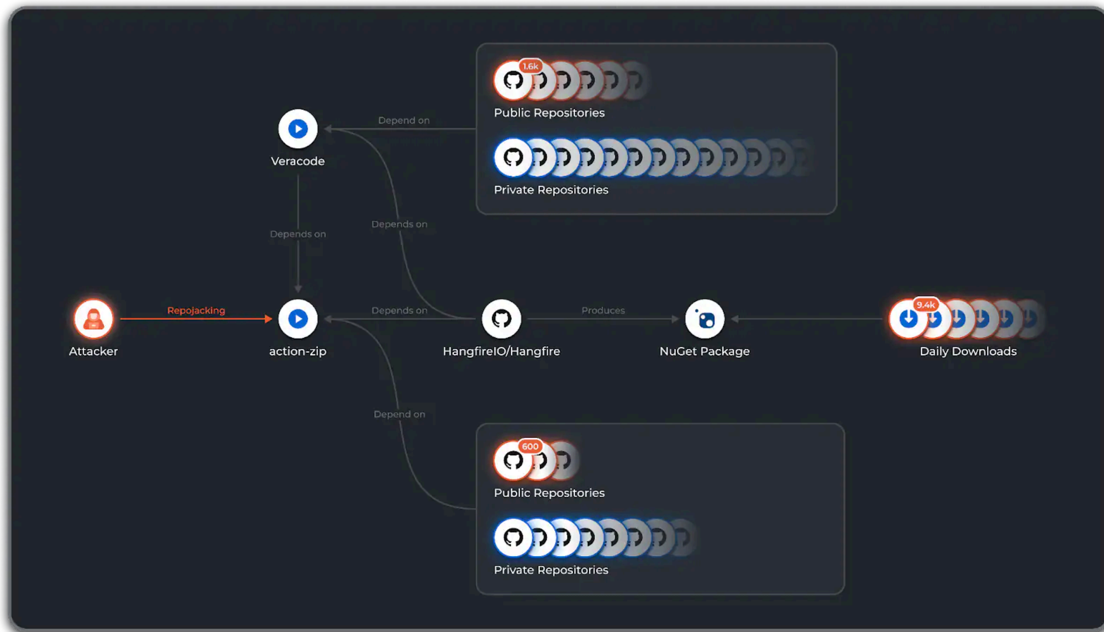


Figure 14: Impact of a worm spread

Veracode and Hangfire acted on the findings we reported:

- Veracode fixed the issue by replacing the vulnerable action with custom bash commands.
- Hangfire decided to completely delete all workflow files.

The Worm Demo

We created a closed demo environment to show how a GitHub Actions worm takes advantage of the methods used to spread malware to any infectable repository across the GitHub Actions dependency tree.

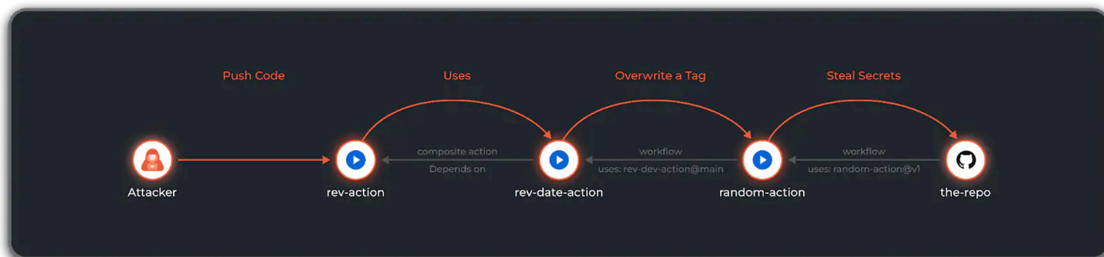


Figure 15: Illustration of the demo attack flow

Our demo includes four repositories:

- A repository named the-repo is our primary target, and we want to steal its secrets. The repo has a workflow that uses the random-action action with the v1 tag reference.

- random-action has a workflow that uses the rev-date-action action using the main branch reference.
- rev-date-action depends on the rev-action action by using it directly through its action.yml file.

rev-action is the weakest link of the chain. The attacker had previously compromised the repository to gain write access using an initial attack vector.

Video: The infection action chain demonstrated

As seen in the video showing the infection chain, the attacker infects the rev-action action by committing code to its action.yml file, which downloads and executes the worm.

1. rev-date-action is a composite action that directly depends on *rev-action*, configured through its action.yml file. It's immediately infected via the rev-action action without the worm performing any action.
2. Next, we see the execution of the CI workflow of the random-action action. This workflow uses the rev-date-action action, and it also grants the GITHUB_TOKEN write permissions on the contents scope against the repository. When rev-date-action runs, it uses the GITHUB_TOKEN and tries to infect the random-action repository by pushing code to the main branch. This attempt fails, because the repository has a branch protection rule that protects the main branch. Next, the worm tries to overwrite the latest created tag. This attempt succeeds, and it successfully overwrites the tag with the malicious code that installs the worm.
3. the-repo, our target repository, contains the VERY_SECRET secret. When its CI workflow runs, it uses the infected v1 tag of the random-action action, which makes the workflow leak its credentials, including the VERY_SECRET credential.

Know Your Pipeline Dependencies

In software development, the concept of dependencies is well-understood. When you build a software application, you rely on a variety of software components — libraries, frameworks, tools, etc. Development teams can track and manage these dependencies using a software bill of materials (SBOM).

The same concept applies to pipelines. A pipeline is a set of steps used to automate a task, such as building, testing and deploying software. Pipelines can also have dependencies in the form of other pipelines, tools and services.

As in software development, the dependencies of pipelines can pose security risks. If one of the dependencies up the dependency chain is compromised, it could affect the entire pipeline. This is why teams need to track and manage the dependencies of pipelines as carefully as they would track and manage the dependencies of software applications.

How to Protect Your Workflows and Assets

Multiple security controls can prevent or raise the difficulty of successfully attacking repositories using the worm. In order of effectiveness, controls include:

1. Set GITHUB_TOKEN and PAT contents permission to the minimum required — with special attention to reducing write permissions against the repository — to prevent infection by the worm. Consider using GitHub's actions-permissions project to reduce workflow permissions. In general, implement strict PBAC

(Pipeline-Based Access Controls) to make sure the workflow is granted with the least privileges and access it needs to fulfill its purpose.
2. Configure branch and tag protection to further prevent infection and protect the codebase.
3. Monitor and limit outbound network connections from workflow runners to prevent the download of malicious code into pipelines and prevent malware from reporting to C2 servers.
4. Pin actions using a commit hash to reduce the risk of using a maliciously modified action.

Learn More

The CI/CD attack surface has changed considerably in recent years, making it challenging to know where to get started with CI/CD security. If you're looking for practical help, check out the Top 10 CI/CD Security Risks: The Technical Guide.

And if you're now thinking about Prisma Cloud, take it for a free 30-day test drive and discover the advantage.

Source: https://www.paloaltonetworks.com/blog/cloud-security/github-actions-worm-dependencies/