Characterizing the Security of Github CI Workflows

Igibek Koishybayev
North Carolina State University

Aleksandr Nahapetyan
North Carolina State University

Raima Zachariah
Independent Researcher

Siddharth Muralee
Purdue University

Bradley Reaves
North Carolina State University

Alexandros Kapravelos
North Carolina State University

Aravind Machiry
Purdue University

Abstract

Continuous integration and deployment (CI/CD) has revolutionized software development and maintenance. Commercial CI/CD platforms provide services for specifying and running CI/CD actions. However, they present a security risk in their own right, given their privileged access to secrets, infrastructure, and ability to fetch and execute arbitrary code.

In this paper, we study the security of the newly popular GitHub CI platform. We first identify four fundamental security properties that must hold for any CI/CD system: *Admittance Control*, *Execution Control*, *Code Control*, and *Access to Secrets*. We then examine if GitHub CI enforces these properties in comparison with the other five popular CI/CD platforms. We perform a comprehensive analysis of 447,238 workflows spanning 213,854 GitHub repositories. We made several disturbing observations. Our analysis shows that 99.8% of workflows are overprivileged and have read-write access (instead of read-only) to the repository. In addition, 23.7% of workflows are triggerable by a pull_request and use code from the underlying repository. An attacker can exploit these workflows and execute arbitrary code as part of the workflow. Due to the modular nature of workflows, we find that 99.7% of repositories in our dataset execute some externally developed plugin, called "Actions"[1], for various purposes. We found that 97% of repositories execute at least one Action that does not originate with a verified creator, and 18% of repositories in our dataset execute at least one Action with missing security updates. These represent potential attack vectors that can be used to compromise the execution of workflows, consequently leading to supply chain attacks. This work highlights the systemic risks inherent in CI/CD platforms like GitHub CI; we also present our own Github action, GWChecker, which functions as an early warning system for bad practices that violate the identified security properties.

[1]In the rest of the paper, we use "plugins" to refer to Actions in GitHub CI

1 Introduction

Continuous Integration and Delivery [41], commonly referred to as CI/CD, are software development practices that involve automating integration, testing, and delivery of software in a consistent, regular and automated manner. CI/CD pipelines, in addition to increasing efficiency, also reduce costs for the organization [40]. Consequently, the adoption of CI/CD pipelines is increasing rapidly [7, 38]. There exist several CI/CD services (TravisCI [19], CircleCI [4], Gitlab CI [17], and more) that enable developers to set up their CI/CD pipeline quickly. Developers need to provide specific configuration parameters about how the software is built and tested. Furthermore, developers use these CI/CD services to automatically deploy the software to corresponding code repositories such as Python Package Index (PIP) [15] or Debian repository [6].

The ease of CI/CD adoption, thanks to third-party services, has its trade-offs. Now developers need to trust third-party CI/CD services to secure the code, artifacts, and secrets from supply-chain attacks [59]. These attacks could have devastative effects, as demonstrated by the recent SolarWinds [18] attack. It is essential to ensure that CI/CD pipelines are correctly configured and do not have any security vulnerabilities. Unfortunately, developers are known to misconfigure their CI/CD pipelines [61, 62]. The CI/CD infrastructure itself could have security vulnerabilities [51] jeopardizing the security of all the repositories using the corresponding infrastructure.

In late 2018, GitHub introduced a new CI/CD infrastructure called *GitHub CI*[2], which enables developers to create CI/CD pipelines called GitHub Workflows[3], which enables developers to define their pipelines by specifying a sequence of steps in a `YAML` file. The workflows are tightly integrated with the GitHub ecosystem and their execution can be controlled through various events such as `pull` or `push`. The workflows can also use *Actions*, which are modules written by other users

[2]GitHub's CI/CD product is called GitHub Actions. However, to avoid any confusion with actions (the external modules that can be used in workflows), we use GitHub CI instead.

[3]In the rest of the paper we use "workflows" to refer to GitHub Workflows

and available as public repositories on GitHub. These Actions are similar to libraries in software development and encompass commonly used tasks such as building a `cmake` project (Section 2.2). Furthermore, for each repository, GitHub provides helpful free resources (compute and storage) [1] to run Workflows. In addition to the features mentioned above, there are many other advantages of GitHub CI in comparison with other CI/CD services [11]. Consequently, since its introduction, GitHub CI has gained tremendous popularity, and developers are rapidly moving their CI/CD pipelines to GitHub CI [45].

Even large, security-aware organizations such as NSA [50], Bootstrap [60], Microsoft [48], and LLVM Project [46] have also started using workflows for their CI/CD.

Given its popularity and adoption, it is crucial to ensure that GitHub CI ecosystem is secure. The tight integration between workflows and GitHub ecosystem, in addition to enabling developers to streamline their CI/CD pipeline, unfortunately, also introduces new attack vectors, especially those related to supply chain attacks. For instance, an attacker can create a pull request and make a misconfigured workflow to perform a deployment based on the attacker's code. A more realistic example would be the recent backdoor introducing commit [14] in `PHP` which might have triggered a deployment workflow, thereby publishing the backdoored interpreter to official repositories. We identified[4] that you can execute arbitrary code using that pull request trigger, which was actively used to perform crypto-mining attacks. Recently, GitHub fixed this issue [12]. Despite these growing attacks, unfortunately, there is no work in understanding and analyzing the security risks associated with GitHub CI.

In this paper, we perform the first thorough security analysis of GitHub CI ecosystem and answer these research questions:

RQ1: What are the security properties (SPs) that need to hold to have a secure CI/CD? (Section 3.1)

RQ2: How does GitHub CI compare to other public CI/CD platforms according to SPs? (Sections 3.2 and 3.3)

RQ3: How does usage behavior of workflows affect GitHub CI SPs? (Section 5)

In order to answer these questions, we started by understanding the GitHub CI execution mechanisms and formulating the required security properties and corresponding necessary conditions. We further referred to the available documentation and reverse-engineered the workflow execution environment. Our analysis resulted in the identification of various attack vectors and security flaws in GitHub workflow execution. The details of the possible attack vectors are accompanied by Proofs-of-Concept (PoC), demonstrating the feasibility and impact of exploiting the attack vector.

Second, we perform a comprehensive evaluation of 447,238 GitHub Workflows spanning 213,854 repositories. We identified various exciting observations regarding workflows' usages and the developers' common flaws in their workflow design.

[4]This was also simultaneously discovered by another researcher [13]

We observed that 96.7% of analyzed repositories depend on third-party code *i.e.,* third-party actions or docker containers, where *38,315 of them depend on third-party actions with at least one active security vulnerability*. Furthermore, *in 146,803 of workflows an attacker can execute arbitrary code as part of the workflow by just raising a pull request*. All our findings have been reported and acknowledged by the GitHub security team and repository owners of the workflows missing a security property.

We conclude by suggesting various defense-in-depth mechanisms to secure GitHub Workflows (Section 5).

This paper makes the following contributions:

- We identify the necessary security properties for CI/CD platforms that must hold to protect infrastructure from software supply chain attacks. (Section 3.1)
- Analysis of the five most popular CI/CD platforms against the four identified security properties. (Section 3.3)
- In-depth analysis of security risks of GitHub CI. We build an extended list of attack scenarios against repositories that use untrustful or vulnerable third-party actions hosted on Github (Section 3.3)
- Extensive analysis of public repositories that use GitHub CI. We found that 18% of repositories in our dataset use vulnerable third-party actions, and less than 2% of all repositories follow the security guidelines provided by Github regarding commit hash references (Section 5)

2 GitHub CI Overview

GitHub CI[5] is a continuous integration (CI) and continuous development (CD) framework built into GitHub that was introduced in 2018. It can be enabled on a GitHub repository (private or public) through *Settings* $\rightarrow$ *Actions* in the repository webpage. GitHub CI enables developers to create *Workflows*. Each workflow describes a set of tasks that needed to be performed as part of its execution. Individual repositories may contain multiple workflows configured to automate part of the development process, *e.g.,* greeting new collaborators, testing, or deploying.

2.1 Workflow Configuration Syntax

A GitHub Workflow is described in `YAML` format by creating a file under the *.github/workflows* directory of the repository. Below is an explanation of the workflow syntax used by the sample workflow in Listing 1.

Execution Triggers: A workflow has one or more execution triggers (**`on`**) that specify *when* or *which events* on the repository should trigger the execution of the workflow. Our example workflow (Listing 1) will be executed when either a *push* or *pull event* occurs on the main branch and *every day*

[5]Github's CI/CD product is called GitHub Actions. To avoid any confusion with actions (the external modules of workflows), we use GitHub CI instead.

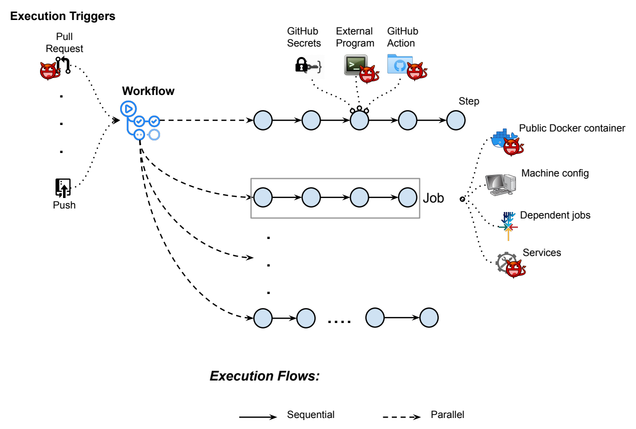


Figure 1: GitHub workflow architecture. When workflow is triggered by execution triggers, it will start the execution of one or more jobs in separate VMs. Each job consists of steps that are executed in the defined order. Steps can be a shell commands, third-party actions, external programs, docker containers

at 5 am UTC as specified by the cron timestamp. Workflows can also be triggered manually or through webhooks [9].

Jobs: Figure 1 shows the execution flow of a workflow, which is a collection of one or more jobs (`jobs`) that run in isolation on newly spawned virtual machines. Jobs can be made explicitly dependent (`needs`) on other jobs, wherein the dependent job(s) will be executed first before the current job. The workflow in Listing 1 has two jobs: `build` and `test`. The `test` job depends on `build` job, so the `build` job runs first. Note that GitHub does not allow cyclic dependencies, and thus any workflow having cyclic dependencies will not be executed.

Machine Configuration: Jobs need to specify the required machine configuration (`runs-on`) on which the job can be executed. In Listing 1, both jobs need to be run on a `ubuntu-latest` machine. GitHub provides labels for various well-maintained machine configurations [2], with the latest software packages. The developers can also use the label of a self-hosted machine with the custom configuration [3]. In this case, however, it is the responsibility of the repository owners to maintain the self-hosted machines, including installing the latest security patches to avoid security breaches.

Steps: Each job is a sequence of one or more steps (`steps`). The steps of a job are executed *sequentially* in the order specified in the `YAML` file. For instance, in Listing 1, the `build` job contains four steps and are executed in the order ①, ②, ③, and, ④. A step can be a sequence of run commands (*e.g.,* ②, and ⑤), where the provided commands (specified with tag `run`) will be executed using the default shell of the machine. For instance, step ② in Listing 1(*i.e.,* `sudo ./build.sh`) will be executed as a shell command on `ubuntu`. Note that the developer needs to make sure that all the files (*i.e.,* `build.sh`) needed to execute the command are available in the system path. In this case, `build.sh` is part of the repository, and it is checked out using a GitHub action (①), we will explore this next.

```
name: MyWorkflow    ← Workflow Name

on: ← Execution Triggers
  # Workflow triggers on push
  # and pull requests to the main branch
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
  # Also, workflow gets executed every day at 5 am UTC
  schedule:
    - cron: "0 5 * * *"

jobs: ← All Jobs in the Workflow

  build: ← Job (Name: Build)

    runs-on: ubuntu-latest ← Job's Machine configuration

    steps: ← All Steps in the Job
    # The following steps are executed sequentially

    # Check out the current repository
    # on default branch
    - name: Checkout the repository ①
      uses: actions/checkout@v2

    # Execute the given command using shell
    - name: Build Project  ②
      run: sudo ./build.sh

    # Execute action defined in the current repository
    - name: Local Action ③
      uses: ../path/to/action@v2
      with:
        apikey: ${{secrets.API_KEY}} 🔒

    # Perform static analysis of all source files
    # in the repository using an action from
    # its public GitHub  URL.
    - uses: microsoft/devskim-action@45bc8e9 ④
      with:
        directory-to-scan: .
        output-directory: scanneroutput

  test: ← Job (Name: Test)

    needs: build ← Dependent jobs
    runs-on: ubuntu-latest

    steps:
    - name: Test Project  ⑤
      run: sudo ./test.sh
```

Listing 1: Example of the workflow configuration file. The workflow contains two jobs (build, test), and uses two third-party actions.

2.2 GitHub Actions

To support modularity and code reuse, GitHub CI workflow can references externally defined modules, called **actions**, as a step inside the job. Listing 1 shows examples of reference to a GitHub Action (*e.g.,* ①, ③ and ④), with the field `uses`. Actions encapsulate commonly used tasks such as building a `cmake` project, deploying a Python package to `PyPI` repository, etc. For instance, the action `actions/checkout@v2` (①) in Listing 1 performs `gitcheckout` of the default branch of the current repository into the current directory.

A developer can write custom actions in their workflow or share the action with the GitHub community by making the corresponding repository public. To publish an action to the GitHub Marketplace, the developer only needs to set up 2FA on their account. There is no reviewing process for the actions published in the Marketplace. An action is created by creating `action.yml` or `action.yaml`, a `YAML` file which defines the inputs, outputs, and main entry point for the action. An action encapsulates code that performs the specified task on the given input and produces the desired output. For example, the action `microsoft/devskim-action` performs static analysis on all the source files in a given directory as shown in Listing 1. An action can be written in any language or could be a pre-built binary. However, GitHub provides additional support for actions written in JavaScript or encapsulated using Docker containers.

A step can use a local action (defined in the current repository) or an external public action using the corresponding repository path. An action is specified as `<path>@<reference>`. Here, `<path>` is a relative file path in case of local action or URL relative to `github.com` in the case of external action. `<reference>` is a commit reference, which can be a tag, a branch name, or a commit hash. For instance, the steps ③ of Listing 1 use a local action with path `../path/to/action` relative to the location of the workflow file. Similarly, ② use an external action with path `microsoft/devskim-action` (*i.e.,* repository path `github.com/microsoft/devskim-action`). Note, that in steps ① and ③ the action is referenced using a commit tag *i.e.,* `v2`, whereas in step ④, the action is referenced using a commit hash (`45bc8e9`).

GitHub Secrets: An action can require an input that could be a secret, such as an `APIKEY` or password for a PyPI repository. To pass the sensitive information to *individual steps* without revealing them in plain text, GitHub provides support for *Secrets* [8, 16]. Repository owners can define secrets as key-value pairs, where the key is the name for the secret and should be unique for a repository and the value contains the corresponding sensitive information. Workflows can use a secret by using `${{secrets.<key>}}`. During workflow execution GitHub runner will replace `${{secrets.<key>}}` with the value of the `<key>` secret. In Listing 1, as indicated by 🔒, `API_KEY` (a secret) is passed to the local action using Github Secrets. It is expected that GitHub will only pass the provided secret to the specified action.

3 GitHub CI Security Analysis

Figure 1 also has marked (with a devil icon) externally controlled entities which are the points through which a workflow execution could be affected by an external or malicious user, who need not be the owner of the repository. For instance, an attacker can trigger the execution of the workflow in Listing 1 by creating a pull request. Similarly, if the action `microsoft/devskim-action` has a vulnerability, then it can be used to gain complete control of the workflow execution. This is because, as mentioned before, each step runs with admin privileges.

In this section, we (1) define generic security properties that can be applied to any CI/CD pipeline, (2) compare differences in features that are relevant to the security of pipeline between GitHub CI and other CI/CD platforms (3) discuss how these features affect the security properties of GitHub CI and other CI/CD platforms.

3.1 Security Properties

A CI/CD infrastructure is primarily meant to perform continuous integration tasks such as testing and/or deploying the tested code. Consequently, CI/CD infrastructure should have at least the following capabilities w.r.t to the underlying source code repository *i.e.,* ability to read the contents of the repository and write to the deployments. These are, in fact, the only capabilities that are needed for CI/CD infrastructure to be able to achieve the "majority" of its goals. Because from a security perspective, according to the principle of least privilege [54], CI/CD infrastructure should *not* have write access to the code repository *i.e.,* it should *not* be able to perform any code changes[6]. However, during the security analysis of GitHub CI we observed that by default all workflows have write access to the repository code (See Table 1) even though workflows is triggered by *less* important events such as issue, comment etc, which violates the principle of least privilege. In other words, by default, any code running as part of the workflow in GitHub CI has write access to the repository code.

Considering the least privilege principle we want to ensure that only authorized users are able to perform the following tasks in the context of CI/CD pipeline[7]:

- *Admittance Control (AC):* Only people with the right permissions must be able to add, delete, or modify workflows to the repository. Otherwise, an attacker can add a workflow to hijack the resources of the CI/CD pipeline, delete/modify existing workflows to disrupt the automation.
- *Execution Control (EC):* To configure events that trigger the execution of workflows. Here, the intuition is that a workflow could be performing writes or deployments. The ability to change triggers for such deployment workflows could allow users to deploy from arbitrary and untested commits resulting in unstable and potentially buggy deployments.
- *Code Control (CC):* To control *which* code runs as part of a workflow. For instance, code (binaries, scripts) that runs in the CI/CD should not behave unpredictably and be consistent from one run to another. After initial configuration, the pipeline must be immutable and perform the task with predictable results.

[6]May be with the exception of some files related to testing

[7]We define authorized users as the organization members, owners or outside collaborators with write permission [27] to the repository

- *Access to Secrets (AS):* To avoid misuse of secrets, it is important to ensure that a secret can be accessed by only those steps to which it is explicitly passed. We want to ensure that these secrets are handled properly by using when explicitly specified in a workflow.

We apply these security properties to other CI/CD platforms as well, to compare them to GitHub CI in future sections.

3.2 GitHub CI vs Others: Features

The biggest difference of GitHub CI from other CI/CD platforms is its wide permission of the pipeline (Table 1) and plugin system that has higher privileges (Table 2).

Permissions. In GitHub CI, by default, all workflows have write permissions to the entire repository as shown in Table 1. Thus, any vulnerable or malicious code in a workflow can directly affect the repository (including code). On the other hand, Gitlab CI does not provide write permission to the (internal) repository code for the pipeline by default. To be able to write to the repository from a pipeline, developers must configure the deployment keys for the repository, and pass the key to the pipeline.

CI/CD Platforms	Permissions	
	Code read	Code write
TravisCI	●✓	◑✗
CircleCI	●✓	◑✗
Jenkins	●✓	●✗
Gitlab CI external	●✓	◑✗
Gitlab CI internal	●✓	○✓
GitHub CI	●✓	●✗

Table 1: The table shows the default read-write permission to the code by different CI/CD platforms. Open circle (○) - means no. Filled circle (●) - means yes. Half circle (◑) - means lowest permission possible but are restricted by options provided by external VCS. Note that Gitlab CI has two rows for external projects and internal projects. The markings ✗ and ✓ on top of the circles indicate over-privileged and expected privileges respectively.

Another interesting finding was that external CI/CD platforms such as TravisCI, CircleCI, and Gitlab-CI follow the principle of least privilege and request only the required permissions. For example for Bitbucket [57] and Gitlab [28] VCS TravisCI [29] and CircleCI [26] request read-only permission to the code. However, in the case of GitHub, external CI/CD platforms can only request repo scoped token, which grants full access to the private and public repositories of the user. In other words, if someone is using external CI/CD platforms with GitHub he/she is exposed to more security risk compared to when CI/CD platform is used with other VCS providers.

CI/CD Platforms	Plugins			
	First-party	Third-party	Mutable	Review
TravisCI	●✓	◑✗	○✓	○✗
CircleCI	●✓	●✗	○✓	○✗
Jenkins	○✓	●✗	○✓	○✗
Gitlab CI external	●✓	○✓	○✓	○✗
Gitlab CI internal	●✓	○✓	○✓	○✗
GitHub CI	●✓	●✗	●✗	○✗

Table 2: Different CI/CD platforms plugin support. Here mutable means referenced (installed) plugin can change without changing its reference. Open circle (○) - means no. Filled circle (●) - means yes. Half circle (◑) - means the capability of plugins are limited by what is available by API. The markings ✗ and ✓ on top of the circles indicate whether the corresponding support is bad or good for security respectively.

Plugins. GitHub CI plugin system significantly differs from other CI/CD platform's plugin systems because of the ability to include a plugin into the workflow by just referencing the repository. CircleCI also provides the ability to reference a plugin (called Orbs) using a repository. However, the main difference between them is that plugins in CircleCI are immutable after referencing, while GitHub CI plugin references are highly mutable. We define a mutable plugin as one that can change its behavior from one run to another. Mutable plugins are a threat to the overall security of the CI/CD pipeline since developers can not verify and pinpoint the execution of the pipeline. Thus, making the results of the CI/CD pipeline runs potentially unpredictable.

Referencing plugins. As described in Section 2, a workflow can reference a plugin by using branch name, tag, or commit hash. We consider tag and branch name to be *dynamic references* because they can change over time *i.e.,* developer of the action repository can modify the tag to point to a different commit. Thus, the action referenced by *dynamic references* can potentially change the runtime behavior of the corresponding workflow.

On the other hand, the commit hash is a *static or immutable reference* as it does not change with time. Here, the action's code is fixed and remains the same over all the executions of the workflow. However, during our security evaluation, we found that under certain conditions, an adversary can change the behavior of a commit hash referenced action (Section 5).

Even though GitHub CI advises everybody to use commit hash to reference actions, we show in Section 5 that only a handful of people follow the advice.

Despite the similarity of the concept of plugin for CircleCI, and GitHub CI, CircleCI employs semantic versioning for the plugin (called orbs) and makes sure that references to the certain version (*i.e.,* 1.2.3) of the plugin return exactly the same code always, *i.e.,* immutable. In CircleCI, developers can publish volatile (mutable) plugins for easy development.

However, CircleCI automatically deletes the mutable reference after 90 days. Other CI/CD platforms that support third-party plugins are (1) limited to providing only functionality that can be achieved with CI/CD platform API endpoint (TravisCI), or (2) require server administrator privileges to modify the plugins (Jenkins).

Gitlab CI only supports first-party plugins covering almost all essential CI/CD pipeline functionality, including deployment, testing, and maintenance. Their philosophy is that developers must not trust third-party plugins for essential functionalities. Gitlab CI does not provide the third-party plugin system citing quality and security degradation [23].

Plugin Review. Unfortunately, as seen in Table 2, *none of the CI/CD platforms that support plugins have a review mechanism* in place to check the quality of the third-party plugins. We believe having some level review process for the plugins will significantly improve the quality and security of the overall pipeline.

In summary, GitHub CI's by default wide permission (write to the code) combined with its plugin system warrants a thorough analysis of the security properties of GitHub CI. In addition to the *mutability*, third-party actions run with `sudo` privileges in GitHub CI (as discussed in Section 2) making the security analysis even more important.

3.3 GitHub CI vs Others: Security Properties

In this section, we analyze the necessary CI/CD security properties of GitHub CI in comparison with other platforms. Table 3 shows the summary of this comparison. We present a detailed analysis of each security property in the following subsections.

3.3.1 Admittance Control

Here, we want to ensure that only authorized users should be able to admit (add, delete, or modify) a workflow into the repository.

Importance. Verifying who is introducing new workflows or modifying the existing ones is crucial in securing pipelines. This is needed because, by admitting new workflows into CI/CD pipeline malicious users can exploit the pipeline to set up a botnet, perform cryptomining or "eat up" resources of the organization.

In all of the tested CI/CD platforms, configuration files reside together with the code in the VCS. Thus, only authorized users can admit a new workflow into the pipeline through access to the repository (**C1**).

Restriction on adding new workflows through CI/CD runs (C2). However, since GitHub CI (and Jenkins) provides write permission to the workflow by default, if the workflow is compromised during the execution of the pipeline, *e.g.,* through vulnerable and/or malicious third-party action, an attacker can introduce a new workflow into the pipeline using workflow's wide permissions. We developed a proof-of-concept action that introduces a new workflow to the pipeline [20]. This is a classic example of the confused deputy problem when unauthorized users elevate their privileges by using an intermediate application with higher privilege.

Executing workflows from a PR after merging (C3). Another way of introducing a new workflow is through a pull request (PR). We noticed that a pull request that adds a new workflow could be executed as part of the repository *before the pull request is merged* into the repository. Consequently, users who can raise a pull request can run arbitrary workflows as part of the repository. To exploit this, an attacker first forks the target repository into the attacker-controlled account. Next, the attacker modifies the local repository by adding a new workflow (say `attackwf`) with `pull_request` being one of the execution triggers. The steps of `attackwf` execute arbitrary code needed by the attacker. Finally, a pull request will be raised from the local repository to the target repository. This causes the attacker added workflow *i.e.,* `attackwf` to be executed as part of the target repository. This behavior is available in GitHub CI, Gitlab CI, CircleCI, and Jenkins. Recently, to prevent malicious usage of the feature from hijacking the resources GitHub CI and Gitlab CI disables the execution of newly created workflow if the request comes from a first-time contributor. This was in response to crypto mining campaigns [12] that were discovered by another researcher [13]. The summary of our analysis is shown in the first row of Table 3.

3.3.2 Execution Control

As mentioned in Section 2, we also want *which* events to trigger a workflow execution to be controlled by authorized (*i.e.,* with write access) users.

Importance. This is required as the workflows can be used for automated deployment, which should be allowed only for users with write access.

Except for GitHub CI and Gitlab-CI, all other CI/CD platforms do not store the trigger events in their configuration files. This prevents modification of the configuration file by contributors. Developers can change the triggers only by using the dashboard of the corresponding platform (**C4**). Also, except GitHub CI, all platforms have a limited number of triggers, such as push and pull-request events. GitHub CI has a plethora of events that can be used to trigger the workflow, such as issue creation, comment, etc. We noticed that a workflow executes with write permissions even if triggered by a "low" important event such as a comment on an issue.

Restriction on modifying execution triggers through CI/CD runs (C5). As explained in Section 2, triggers for a workflow are specified in the corresponding `YAML` file using **`on`** tag, which can only be modified by the users with write access. However, similar to **C2** (Section 3.3.1), as workflows

		TravisCI	CircleCI	Jenkins	Gitlab CI extrernal	Gitlab CI internal	Github CI
Admittance Control	(C1) Contributor can add workflow	●	●	●	●	●	●
	(C2) CI/CD run can NOT add new workflow	●	●	○	●	●	○^{W}
	(C3) Executes workflow from PR only after merge	●	◑	○	●	◑	◑^{W}
Execution Control	(C4) Contributors can modify the triggers	●	●	●	●	●	●
	(C5) CI/CD run can NOT modify the triggers	●	●	○	●	●	○^{W}
Code Control	(C6) CI/CD run can NOT modify the code	●	●	○	●	●	○^{W}
	(C7) CI/CD run is deterministic based on config	●	●	●	●	●	○^{W}
Access to Secrets	(C8) Masked	●	●	◑	●	●	●
	(C9) Accessible only to explicitly authorized steps	○	○	◑	●	●	○
	(C10) Restricted from pull requests	●	◑	◑	●	◑	◑^{W}

Table 3: Comparison of five different CI/CD platforms in four security properties (AC, EC, CC, AS). Open circle (○) - means no. Filled circle (●) - means yes. Half circle (◑) - means developers can configure the feature in config file or by using plugin. The shades of red indicate conditions violating the security property in corresponding platforms. The marker *W* indicates whether the condition is workflow or configuration dependent.

execute with write permissions, a malicious and/or vulnerable action in a workflow can change the trigger of an workflow.

3.3.3 Code Control

As mentioned earlier, Code Control is a security property that controls the code that runs as part of the CI/CD pipeline, such as binaries, plugins, and other things.

Importance. Any code that runs as part of a workflow must be trusted. Running untrusted code could have devastating effects. For instance, an untrusted command or action can tamper with the environment, *e.g.,* by changing the default registry used by package managers such as `npm`, thereby affecting all the steps that install packages from `npm`.

The plugin system of CI/CD platforms contributes most to the CC security property.

Restriction on changing code through CI/CD runs (C6). The plugins are part of the config (*i.e.,* workflow) file of CI/CD platforms. In platforms Jenkins and GitHub CI, CI/CD runs with write permissions (Section 3.3.1). Hence, a malicious and/or vulnerable action in a workflow can modify the code executed as part of the workflow by changing the plugin name or can introduce a new workflow with the required plugin.

Code control without modifying config (C7). As discussed in Section 3.2 GitHub CI has the most controversial plugin system. The third-party plugins in GitHub CI are mutable, which makes the GitHub's workflow runs unpredictable compared to other platforms as shown in Table 3. Also, as mentioned in Section 2, in GitHub CI every step executes with admin privileges. Consequently, any code that runs as part of a step has complete control of the underlying machine.

Given that these actions run with admin privileges, it is important to ensure that the code within these actions can be trusted. We consider the local actions to be trusted as they are part of the current repository. However, the developers need to be careful regarding the external actions.

Trusted vs. Untrusted action creators. As described in Section 2.2, anyone can create an action by creating `action.yml` file in their public repository. Some of the actions creators (*e.g.,* Microsoft, CheckMark) are verified, which means GitHub trusts these creators, and we consider that the actions developed by them as trusted. However, while using the actions from unverified creators, it is better to use static reference (*i.e.,* commit hash) for the action as it restricts the creator from changing the code. In summary, workflows should always try to use actions from verified creators, and actions from unverified creators should always be statically referred. This is also what GitHub suggests in their official documentation [16]. However, as we show in Section 5, developers seldom practice this, making their workflows wide open to be influenced by unverified developers.

Vulnerabilities in actions (C7). Finally, irrespective of the type of creators, the actions themselves could have security vulnerabilities making the workflow vulnerable and consequently leading to supply-chain attacks. As we show in Section 5, this is rampant, and many workflows are using actions with known vulnerabilities.

3.3.4 Access to Secrets

As discussed before Access to Secrets security property is about how CI/CD handles sensitive information.

Importance Developers may need to pass sensitive information (*e.g.,* `API_KEY`) to steps of a workflow to perform certain authorized tasks such as deployment to a `PIP` repository. As shown by the previous work [52, 64], the sensitive information should not be hardcoded in the repository files as everyone can read them. If CI/CD pipeline mishandles the sensitive information and malicious actor access the secrets, she can compromise other systems as well, *e.g.,* code registry.

Masking (C8). To ensure the safety of secrets, first, CI/CD should ensure that secrets are never visible outside the execution environment of the workflow or pipeline. One common way where developers are known to leak sensitive information is through logs. So, CI/CD platform should ensure that the secrets are scrubbed or masked in the execution logs. Except for Jenkins, all CI/CD platforms mask the secrets in the build log by default. To mask the secrets in Jenkins, developers need to install a third-party plugin, which complicates the initial secure configuration of the Jenkins pipeline.

Available to only authorized steps (C9). Second, even during the CI/CD pipeline execution, a secret should be visible/accessible to only those steps or plugins requiring the secret as specified in the workflow(or config) file. For instance, in Listing 1, we want *only* the step (3) to access the secret represented by `${{secrets.API_KEY}}`. Only GitLab implements this correctly by preventing access to secrets by steps unless explicitly granted in the config file.

Unfortunately, in GitHub CI, we found that during a workflow execution, all secrets specified in the workflow are decrypted and placed in a file under folder `/home/runner/_work`. Consequently, all steps in a workflow can access decrypted secrets *even* when they are not passed explicitly. For instance, in Listing 1, all sets can access the secret represented by `${{secrets.API_KEY}}`. To better demonstrate this, we developed a proof-of-concept action [20], which, when used in a workflow will dump the decrypted content of all the secrets mentioned in the workflow.

Hidden from pull requests (C10). Finally, access to secrets in a pipeline should also be regulated by how the pipeline execution is triggered. Specifically, secrets should not be accessible if the pipeline execution is triggered by no-privilege events, especially a pull request. A malicious user creates a pull request by modifying a script in the repository referenced by a pipeline. If the pull request triggers the pipeline, it will be using the script in the pull request. If the pipeline has access to secrets, the modified script can leak the secrets, *e.g.,* to a remote server.

By default, none of the CI/CD platforms share the secrets with the pipeline triggered by pull requests. However, you can allow sharing the secrets with the pull-request pipeline in the settings of the project for Gitlab CI, GitHub CI and CircleCI. Furthermore, GitHub CI also shares the secrets with pull requests by default if the pull request is internal, meaning it was raised within the project and not from a forked repository. Also, developers can pass the secrets to all pull-requests in GitHub CI by configuring the workflow to trigger on `pull_request_target`. We show how prevalent is the usage of `pull_request_target` in Section 5.

3.3.5 Summary

Table 3 summarizes our analysis of the desired security properties in GitHub CI along with other CI/CD platforms. We focus on GitHub CI, and as shown in Table 3, none of the security properties *always hold* in GitHub CI. The red markings in sub rows of security properties mark the conditions under which the corresponding security property will be violated. The conditions marked with *w* are workflow dependent, *i.e.,* their violation depends on configuration and contents of a workflow. For instance, the code execution property can be violated if the workflow uses a vulnerable action. On the other hand, the availability of secrets to all steps of a workflow is a platform-wide condition. In the following sections, we present a large-scale analysis of GitHub workflows and show that most of the workflows violate at least one of the conditions.

4 Data Collection and Methodology

In this section, we present our methodology for collecting the repositories' names with GitHub Workflow and third-party actions names.

Repositories with workflows. We use GHArchives to collect the list of repositories that use workflows [35]. We were unable to do a universal crawl using GitHub API because of the recently imposed restrictions (Section 6). GHArchive collects repository information by recording events, *i.e.,* push, pull_request, and more, posted on GitHub's events API endpoint. However, it does not track workflow events and cannot directly track repositories using GitHub CI. However, we noticed that the same github_bot user is responsible for all events resulting from GitHub CI workflows. Therefore, we selected all repositories containing events created by the github_bot user. We queried GHArchive for github_bot generated events from 2019 to July 2021 and used these to extract the names of the corresponding repositories using GitHub CI. We acknowledge that this dataset does not contain all the repositories that use GitHub CI. However, the dataset includes a list of most interesting use cases where workflows interact with the repository or Github APIs themselves. Almost 40% of all repositories we collected had at least one star, and 72.8% of all repositories were active in 2021.

After retrieving all repository names, we used GitHub's REST API to verify that these repositories are not forks of an existing repository and contain at least one workflow under *.github/workflow* directory (See Section 2). In total, we filtered 213,854 (65.5%) out of the initial 326,410 repository names retrieved from GHArchive.

Next, we extracted all workflow `YAML` files from the remaining repositories located under *.github/workflows* directory (Section 2) using GitHub's REST API [10]. We use these workflow files to analyze GitHub CI's usage patterns further. To do that, we parse all the workflow files and store the result as a JSON file in MongoDB. We will share our dataset with the research community upon publication. We discuss the analysis results of repositories' workflows in Section 5.1.

Actions repositories. As mentioned in Section 3.2, GitHub CI allows workflows to use third-party modules, called actions (see **`uses`** keyword in Listing 1). We collected actions by parsing the collected workflow files and filtering based on the **`uses`** keyword. We ignore local actions, *i.e.,*, actions that are part of the workflow repository itself.

In summary, we extract all the external actions used in these workflows and clone their repositories to analyze them for vulnerabilities (Section 5.2). We ended up with 11,438 unique actions.

5 GitHub CI Measurement Results

This section presents the analysis results of collected workflows and third-party actions used in these workflows. We first present our comprehensive analysis of the workflow files and their configuration. Our goal is to present the common usage patterns of workflows and how these violate the desired security conditions (Section 3.3) and result in critical vulnerabilities [5]. Second, we perform a similar comprehensive analysis of third-party actions' (*i.e.,* plugins) to check whether workflows safely use them and whether the actions themselves contain any security vulnerabilities.

5.1 Workflows Analysis

Overall we collected workflow files from 213K repositories, which contain a total of 447K workflows, with an average of 2.2 workflows per repository.

5.1.1 Workflows Permissions

As discussed previously, workflows in GitHub CI by default have wide permissions which grant write access to the repository. Consequently, as discussed in Section 3.3, an attacker can use a vulnerability in the workflow to modify the underlying repository and violate the desired conditions, C2, C5 and, C6. Developers can change the default permissions of a workflow by adding the **`permissions`** field into the workflow YAML file. However, *only 0.2% of all workflows* (**900**/447K) use the **`permissions`** field to configure the permissions of the workflows. Furthermore, even in these 900 workflows, only 62% of them set the permission to the desired read-only.

Recommendation: Following the least privilege principle and restricting the default permission to read-only will protect repository code from unauthorized changes.

5.1.2 Workflow Triggers

The workflows can be triggered in various ways as we discussed in Section 2. Table 4 shows the most popular ways of triggering (*i.e.,* firing) workflows in the analyzed repositories.

Trigger events	**Repositories (%)**	**Workflows (%)**
`push`	179,503 (83.9%)	279,337 (62.5%)
`pull_request`	94,962 (44.4%)	146,803 (32.8%)
`cron`	51,544 (24.1%)	70,719 (15.8%)
`manual`	45,134 (21.1%)	83,616 (18.7%)
`pull_request_target`	7,485 (3.5%)	8,874 (1.9%)

Table 4: Number of repositories with at least one workflow triggered on `push`, `pull_request`, `pull_request_target`, `manual`, and `cron` events. Note that percentages do not sum up to 100% because a repository can contain multiple workflows with each of these configured to be triggered by multiple events.

Pull request and cryptomining. An interesting and rather potentially dangerous way of triggering a workflow is by creating the `pull_request` or `pull_request_target` from a forked repository because the workflow will be executed using the code in the forked repository. Furthermore, workflows triggered by `pull_request_target` will have access to all the configured secrets. An attacker can exploit this by raising a pull request and consequently executing arbitrary code in GitHub CI environment as part of the workflow.

Until recently [12] it was possible to create a new workflow as part of a pull request which will automatically start running even if the pull request was not merged or validated by any means. An attacker used this *feature* to perform cryptomining on GitHub CI resources by raising a pull request with the workflow in Listing 2, which spawns a lot of runners performing mining. Here, the attacker raises a pull request containing the workflow in Listing 2.

Even though GitHub now disables the execution of new workflows created by first-time contributors without the manual approval from a repository owner with write access, we believe it is still possible to perform cryptomining. For example, attackers can gain the trust of the repository owners by first raising a valid pull request and later submitting the cryptomining code. Also, the attacker does not need to create a new workflow to perform cryptomining. They can perform cryptomining by updating the part of the codebase used in workflow, *e.g.,* unit test code. We found that 105K workflows in 66K repositories use scripts that are part of the repository's codebase. In other words, 30.9% of all repositories contain at least one workflow that uses scripts that are part of the codebase. An attacker can use these workflows to execute arbitrary code by raising a pull request with a modified repository where the referenced scripts contain the target code. Note that these numbers are lower bound numbers, and the real number of workflows that use the repository codebase *must* be higher.

In addition to cryptomining, attackers can harm the organization by continuously raising PR and finishing all the GitHub CI resources available for free to the organization, *i.e.,* performing DoS attacks.

Recommendation: Removing the ability to run newly created workflows from PRs until they are merged into the original repository will eliminate the possibility of using workflow resources in a malicious context. This will improve the Admittance Control security property.

Pull requests and self-hosted runners. As we describe in Section 2, GitHub allows developers to run the workflows in personal machines that do not belong to GitHub. However, this

```yaml
name: Cryptomining workflow
on: [pull_request]
  jobs:
   build:
     name: Fetch
     runs-on: ubuntu-latest
     container: ubuntu:20.10
     strategy:
       fail-fast: false
       matrix:
         runner: [0,1,2,3,4,5,6,7,8,9,10,...,19]
     steps:
       - run: ./obfuscated_cryptomining.sh
```

Listing 2: Example of cryptomining workflow that spawns multiple runners on each pull_request

comes with security implications if the machines are not returned to a clean state after the execution of the workflows [37]. Also, by executing workflows with Third-Party Actions (TPAs) you risk your machine being fully compromised. Even if workflow does not contain a TPA but allows it to be triggered by pull_request event, anyone with pull requests permission can compromise the self-hosted machine by introducing malicious code into the codebase. Therefore, GitHub strongly discourages the usage of self-hosted machines as runners for public repositories, which contain workflows that can be triggered by pull requests [16]. Despite this, we found cases of 565 public repositories that run on self-hosted machines. More than half (51.7%), or **292** out of 565 of these repositories are triggered by a pull request event. Note that since GitHub allows developers to define custom labels [36] to reference a self-hosted machine, all the numbers mentioned earlier are lower bound. In reality, the number of self-hosted runners could be greater than the reported number.

Also, we looked into what kind of repositories are using self-hosted machines to run workflows triggered by pull requests. On average 292 repositories have 744 stars and 133 forks. Approximately 20% (53 out of 292) repositories that run on self-hosted machines and are risking potentially being compromised have more than 100 stars. One of these repositories is `kubernetes/minikube` which has more than 20k stars and is popular among developers.

Key finding: There are at least 292 repositories that contain at least one workflow executed in a self-hosted machine that is triggered by a pull request event. This is 51.7% of all public repositories that run in a self-hosted machine.

5.1.3 Workflows Secrets

Securely storing and passing secrets in workflows is critical. As discussed in Section 2, GitHub has a mechanism to store and pass secrets to the workflows.

Our dataset contains 245K cases where secrets are passed using GitHub's mechanism, *i.e.,* `${secrets.foo}` pattern. Almost half (49.7%) of the repositories in our dataset pass secrets at least once. Repository secrets were passed directly to 4.5K external actions developed by third-parties. Among these 4.5K actions, only 359 were created by a verified creator, which accounts for less than 10% of all actions that have direct access to secrets.

In addition to actions that have *direct* access to secrets, some actions can have *indirect* access to them, *i.e.,* without developers passing secrets to the action. If secrets are passed to the workflow, GitHub CI will create files under `/home/runner/worker/_temp` directory that will contain all the secrets passed to the job during the workflow execution. Therefore, other TPAs that are part of the workflow jobs will be able to access the secrets by reading the contents of the directory. There are 5.7K actions with indirect access to the secrets, against 4.5K actions that have direct access to the secrets. Of these 5.7K actions with indirect access, only 53 are from verified creators. We disclosed the issue of indirect secret access by TPAs to GitHub. They responded that this is intentional behavior right now, and plan to restrict the access in future releases of GitHub CI.

Key finding: Developers pass the secrets without considering the TPAs' origin, *i.e.,* trustworthiness. More TPAs can indirectly access the secrets.

Recommendation: To prevent unauthorized access to secrets, GitHub should add the ability to pass the secrets only during the execution of the step that requires it or disable read access to the directory where actions store secrets.

Secrets passed as plain-text. As discussed in Section 2, developers must use a pattern like `${secrets.FOO}` to pass secrets to workflows. We use this knowledge to our advantage by querying our dataset for different usage patterns of secrets. First, we find the repositories that pass secrets to workflows and analyze which keys they use to pass secrets, *e.g.,* if the repository is passing the secret with the `${secrets.API_KEY}` pattern, the key will be `API_KEY`. Second, from the list of all keys, we deduce a reduced list of keywords most likely to be associated with the secrets, *e.g.,* token, password. We use this list of keywords to detect the repositories passing the secrets in plaintext. We do this by querying our database to find when a repository passes some value to the external action with a key that contains one of the keywords. The query result returned 2,240 possible candidates for secret leaks. After filtering out false positives, we have 333 possible leaks.

We raised the issue to the 333 repositories about possible secret leaks. From all of the reports, only 11 confirmed accidental leakage of the secrets and fixed the issue. While most of the issues were not answered, there are some interesting responses from the repository's maintainers regarding the intentional leakage of the secrets. Specifically, the repositories that use Chromatic, a tool to automate UI feedback gathering, are intentionally leaking `projectToken` value. Chromatic documentation [24] explains that this is the only way to allow forked repositories to run workflows with Chromatic. Further investigation revealed that due to security reasons GitHub CI by default does not share the secrets with workflows if a forked version of the repository triggers the workflow. Thus, repositories that want to allow forked repositories to execute workflows with secrets must either pass the secret in plain text inside the workflow configuration file or change the repository's settings to allow passing the secrets to the forked version of the repository. However, since updating the settings of the repository will pass *all* the secrets to the forked repository, developers choose to pass only a limited number of secrets in plain text in the workflow file.

Key finding: Some developers pass the secrets in plain text to allow forked versions of the repository to run the workflows.

Recommendation: Adding the ability to pass the only limited number of secrets to the forked repository can improve security.

5.2 Actions analysis

Over-privileged and mutable TPAs contribute most to the security risks of the Github CI workflows (Section 3). All security properties (AC, EC, CC, AS) can be compromised if the workflow depends on a vulnerable or malicious TPA. For example, the `wayou/turn-issues-to-posts-action` action, which is used to convert issues into posts, is vulnerable to shell injection attacks [42]. Suppose any repository depends on `wayou/turn-issues-to-posts-action` in its workflows. In that case, an attacker can run any code inside the CI/CD pipeline by just crafting the malicious issue. And since the workflow by default gives the write permission to the code even though the workflow is executed only on issue events, an attacker can modify the code.

Actions' statistics. As discussed in Section 4, we collected the action names through the workflow files that use the keyword **`uses`** to reference TPAs. The key **`uses`** were part of workflow files 1,623,413 times in 99.7% of all repositories in our dataset. From all 1.6M times when external actions were referenced, 0.75% were referencing local actions, actions that are part of the repository. During further investigation, we noticed that it is possible to directly reference the Docker image by using the key **`uses`**. There are 6,082 (0.3%) out of 1.6M cases when workflows are referencing the Docker image directly, as such `uses:docker://docker.io/hello-world`. Even though docker usage is only a tiny portion of overall usage, we believe that this behavior will introduce significant challenges in the future for the analysis of Github Actions.

The rest of the 1.6M references are indeed references to 1st-party actions in 62% and 3rd-party actions in 38% of the cases. In general, 213,209 (99.7%) of repositories in our dataset references at least one TPA. Overall there are 11,438 unique TPAs and 19,033 if we consider different versions.

Verified vs Unverified Actions. GitHub has a separate category in the marketplace for actions created by verified organizations, such as Azure, Docker, and Google, called verified creators. As of 15th November 2021, GitHub had only 75 organizations as verified creators who published the actions in the marketplace. In our dataset, only 335 out of 11,438 actions used by repositories are created by verified creators[8].

During analysis, we found, that people more often reference TPAs from non-verified creators than from verified creators, which is counter-intuitive. From the top 20 actions that are used as part of the workflow, if we ignore first-party actions managed by GitHub, there is only one TPA maintained by verified creators as opposed to nine from non-verified creators (Appendix B).

Key finding: The majority of the TPAs used are from non-verified creators. Verified creators maintain only 3% of all used actions.

Recommendation: Adding some level of automated review process for the TPAs can contribute to the defense in depth against malicious TPAs.

Third-Party Actions' References. Code Control property emphasizes the importance of knowing what code runs as part of the workflow (Section 3). One way of controlling what code runs is to audit TPA's code and make it immutable. Therefore, it is important to know exactly how workflows are referencing TPAs. For example, if referenced TPAs are mutable, it is impossible to control the code they are running. As discussed previously, there are three ways to reference the TPAs: (1) tag name, (2) branch name, (3) commit hash.

GitHub documentation suggests using commit hash to reference a TPA unless you trust the organization. Because any other way of referencing a TPA, such as tags, and branch names are mutable, which means actions code can be updated anytime (*e.g.,* injecting backdoor), if someone takes over the repository and/or if the organization has malicious intent. Therefore, we analyze how many of all TPA references follow Github documentation recommendations. Unfortunately, as you can see from Table 5 less than 1% of all TPAs are referenced using their commit hash. Of all 213,209 (99.7%) repositories that use TPAs, only 1.7% (*i.e.,* 3,248 repositories) use TPAs by referencing them with commit hashes. Even worse, commit hash references do not guarantee immutability of the TPAs' code if the referenced action uses a mutable reference to another dependency, *e.g.,* other actions. We developed a PoC [21] and reported the issue to GitHub.

Key finding: In general, developers do not reference TPAs by using commit hash, despite security risks associated with other ways of referencing actions.

[8]Note that the number is different from what is available in the marketplace because not all actions of the verified creator might be published in the marketplace

Reference types	References (non-verified)	Distribution in % (non-verified)
Tag name	474,166 (410,054)	78.8% (68.2%)
Branch name	120,633 (109,400)	20% (18.1%)
Commit hash	6,539 (5,687)	1% (0.9%)
Total	601,338 (525,141)	100% (88.2%)

Table 5: Distribution of different ways to reference all 3rd-party actions and specifically 3rd-party actions from non-verified creators. Despite the recommendation by GitHub to use commit hash references, only 0.1% of references use commit hash. Note: these numbers are for 3rd-party actions only, *i.e.,* does not include 1st-party actions

Another interesting finding is that there are a significant number of TPAs that are referenced by branch name (Table 5). This behavior not only introduces security risks but also may break the execution of the pipeline if maintainers of the actions push code with bugs into the actions' repository. For example, at one point *lowlighter/metric* introduced an infinite loop into the code that they fixed in 9b574376 commit.

The developers' behavior of not referencing the actions with commit hash shows that the developers tend to choose convenience over security.

Recommendation: Introduction of semver versioning scheme in TPAs references as in CircleCI Orbs [30] will lessen the dependency on highly mutable references such as branch names. Also, it will give the flexibility to update the actions if a vulnerability is detected without manual effort, as in the case of referencing using commit hash.

Actions Vulnerability Analysis. Since dependence on TPAs may present some security risks (Section 3), it is important to analyze the external actions for security vulnerabilities. For example, if a repository uses a vulnerable action, malicious actors can compromise the execution flow of the workflow by posting comments on an issue [43, 44] or by controlling git's tag value [32]. We perform the vulnerability analysis of actions by detecting the commit that potentially fixes some security vulnerability.

There are existing tools that can detect automatically vulnerable commit messages for large projects [65]. However, since actions are relatively small projects with the majority of them containing less than a hundred commits, we decided to use a simple regex matching-based tool, `git-vuln-finder` [25], which looks for security-related keywords in commit messages. As a result, `git-vuln-finder` returns a list of *potentially* vulnerability-fixing commits for each repository. After running the tool on all 11K cloned actions' repositories, the tool returned 5.4K potentially vulnerability fixing commits. After the manual verification step, we ended up with 659 actions, which accounts for 5% of all actions we cloned, that have vulnerability-fixing commits in their commit logs.

We then construct the set of vulnerable tags, *i.e.,* the set of tags that points to the commit which comes before the vulnerability-fixing commit. For example, for cake-build/cake-action the tag `v1.3.0` is pointing to a commit `ada1055`, which comes before the vulnerable fixing commit 985efc8. Therefore, we will add `v1.3.0` to a set of vulnerable tags for cake-build/cake-action action. We use this set to detect workflows that use a vulnerable actions version.

Vulnerability severity	Actions	Repositories
High-severity	26	582
Medium-severity	56	28,870
Low-severity	577	10,922

Table 6: Vulnerable first and third parties actions and number of repositories that reference vulnerable versions

Table 6 shows a number of repositories that are referencing a vulnerable version of the action, *i.e.,* action version that is missing vulnerability fixing commits. A group of graduate students who have experience in security categorized all vulnerability fixing commits according to severity (high, medium, low). From Table 6 one can see that majority of vulnerability fixing commits were low severity fixes, such as updating vulnerable npm dependencies.

Reference to one first-party action, `actions/checkout` version `v1` and prior contributes most to the number of repositories that reference medium-severity in Table 6. All versions prior to `v2` are passing GitHub authorization token into the command line in plaintext, which can lead to a leak of GitHub's token unintentionally.

The above results show that developers may depend on vulnerable actions in their repositories' workflows. This could lead to several serious outcomes if not tackled on time, and not come up with ways to inform developers about the security risks of using outdated/vulnerable actions.

Key finding: 38,315 or 17.9% of all repositories use at least one potentially vulnerable TPA in their workflows due to not upgrading the version

Due to limitations discussed in Section 6 we tried to notify only 582 repositories that are depending on the actions' version with high-severity vulnerability from Table 6. Overall, we successfully created issues for 542 repositories. We could not create issues for 40 because of (1) rename or removal of the repository; (2) issue creation is disabled; (3) repository is archived, meaning it is read-only. For rest of the repositories that depend on actions with medium and low severity vulnerabilities, we notified the GitHub itself directly.

Recommendation: Using reduced privileges by default for all actions can prevent attackers to use vulnerable actions as a trampoline during the attack. For example, workflows that are triggered only on events related to issues can have permissions limited to only read/write access to issues.

6 Discussions & Limitations

Data Collection Limitations. There are two main limitations in our data collection. The first limitation being our dataset does not contain all repositories with workflows. As discussed in Section 4, the dataset does not include any repositories with workflows that do not update the repository using workflow(s) or interact with GitHub's APIs. Initially, we tried to use "GitHub Activity Data" on BigQuery to collect *all* the repository names that use GitHub CI. However, since GitHub CI was introduced only in Fall 2018 and BigQuery's default dataset was not updated since 2017, we decided to use GHArchive data stored in Google's Big Query. There is also GitHub's REST API that can be used to crawl GitHub for repository with workflows. However, GitHub's REST API contains a significant rate limitation that limits crawling to 5000 requests/hour, and for each query returns only the first 1000 results *non-deterministically*.[9] In addition to this, GitHub community standards (*i.e.,* policy) forbids *bad* crawling behavior. These limitations were reasons to select GHArchive over GitHub's APIs or anything that uses those APIs in the backend, like Github advance search. Despite the limitations mentioned above, our dataset contains the most exciting cases when repositories use GitHub CI as we focus on cases where actions modifying the repository is an expected side-effect.

The second limitation is that our data does not include all third-party actions hosted in GitHub. However, we argue that even though the dataset does not contain all third-party actions, it contains actions that are used by other repositories in the wild, *i.e.,* which are of the most interest.

Actions' Vulnerability Analysis Limitations. The actions vulnerability analysis may not present accurate numbers regarding the actions vulnerability because of the methodology we employed. While we take measures to decrease the number of false positives by performing manual analysis, we still used basic regex matching of the commit message to detect vulnerability-fixing commits in the action's repository (Section 4). However, we argue that it is not an easy task to detect the vulnerability-fixing commits and, on its own, requires separate research. Also, the goal of the actions' vulnerability analysis was to raise awareness of the security risks that come with referencing third-party actions and providing them with broad permissions. Even if the third-party actions are not malicious, they may be vulnerable, and, thus they can be exploited by malicious actors.

Vulnerable Workflows Disclosure Limitation. We notified only repositories that depend on vulnerable actions with high severity because of GitHub's strict policies regarding automatic content creation. If the developer opens a lot of issues automatically, GitHub will block the account and hide all opened issues. Therefore, we decided to create issues directly only for repositories that depend on actions' version with high-severity vulnerability and notify GitHub directly through the support system about all other issues.

[9]Querying for identical keyword can return different results

Secret Analysis Limitations. Compared to previous works, our analysis of leaked secrets has several limitations. The first limitation is the scale. We analyze only leaks for 190K repositories, while recent papers analyzed a significantly more number [47]. Another limitation is that we did not perform a longitudinal study of the workflows. Thus, we may miss the old leaks of the secrets. Third, we were looking at the leaks only in workflows. Therefore the numbers are lower than in other studies.

7 Related Work

CI/CD Analysis There has been a considerable amount of research in analyzing CI/CD frameworks. Several works [31, 39, 49, 53, 55, 58] looked into the DevSecOps culture, trying to understand their challenges and trade-offs. Gruhn *et al.* [37] are among the first to analyze public continuous integration services and identify that isolation is important in executing various tasks of a CI/CD pipeline. Later, Bass *et al.* [22] looked into the security of Jenkins and proposed the division of Jenkins into *smaller* parts for easy configuration.

Configuration Smells. Configuration smells or problems due to improper configuration, especially in the context of CI/CD pipelines, had been studied extensively [33, 34, 51, 52, 64] and many works try to fix them automatically [61, 62]. Most of these works focus on Travis CI, where the bad patterns are identified manually [33, 34, 52] or through developer surveys [51, 64]. Recently, Vassallo *et al.* [61] proposed a log analysis technique that tries to identify configuration smells through log analysis. Subsequently, they also proposed a tool called CD-Linter [62] that automatically fixes the issues in the `YAML` configuration file.

Unlike previous works, in this paper, we focus on GitHub CI, by first systematically identifying required security properties than trying to find the violations in the workflow configuration file. Our work complements the existing work by extending research to GitHub CI.

Secrets Leakage Detection. Detecting secret leakage in public repositories has been well-studied [47, 56, 63]. Notable recent work is from Meli *et al.* [47], they use entropy-based techniques to find that over 100,000 repositories leak secrets for a set of pre-defined APIs. In our work, we use a much simpler technique based on keywords and independent of APIs. Unlike the previous work, we found significantly fewer leaked secrets as we focus only on workflow files, and most of the developers were well aware of GitHub's secret mechanism.

Identifying Security Fixes. There has been some work [65, 66] in detecting security fixes from commit messages, and most works are based on machine learning techniques. These techniques depend on the availability of a large dataset of commits fixing security bugs, which is hard to get, especially for action repositories, as they are relatively new (< 2 years) and have fewer commits. In our work, to handle this, we use `git-vuln-finder`, which doesn't require a large dataset.

Finally, identifying security fixes is not the paper's main contribution, and any work that doesn't depend on a large dataset can be used to detect security fixes in action repositories.

8 Conclusion

This paper defined four security properties that must hold to secure CI/CD platforms from supply-chain attacks. Then, we compared the newly introduced GitHub CI with five other public CI/CD platforms following security properties defined earlier. Additionally, we investigated how developers use workflows in GitHub CI and their effect on the security properties. Based on the GitHub CI usage by developers, we proposed recommendations on how to improve the security.

Also, we listed four security improvements that can be implemented as part of GitHub CI to protect from security weaknesses. As part of security improvements, we implemented an automated tool called GWChecker which developers can use to detect security weaknesses in their workflow configuration and can also automatically notify them by creating issues if security risks are detected.

We hope that our work will be the first of many kinds of research on GitHub CI.

Acknowledgments

We thank the anonymous reviewers and our shepherd Marcela Melara for their constructive comments and suggestions on how to improve this paper. This work was supported by the National Science Foundation (NSF) under grants CNS-2047260 and CNS-2055554, Office of Naval Research (ONR) under grant N00014-21-1-2159 and by the Defense Advanced Research Projects Agency (DARPA) under the grant N6600120C4031.

References

[1] About GitHub-hosted runners. https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners.

[2] About GitHub-hosted runners. https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners.

[3] About self-hosted runners. https://docs.github.com/en/actions/hosting-your-own-runners/about-self-hosted-runners.

[4] Continuous Integration and Delivery - CircleCI. https://circleci.com/.

[5] CWE-732: Incorrect Permission Assignment for Critical Resource. https://cwe.mitre.org/data/definitions/732.html.

[6] Debian package repository. https://www.debian.org/distrib/packages.

[7] DevOps Trends in 2021. https://www.bmc.com/blogs/devops-trends/.

[8] Encrypted secrets. https://docs.github.com/en/actions/reference/encrypted-secrets.

[9] Events that trigger workflows. https://docs.github.com/en/actions/reference/events-that-trigger-workflows#webhook-events.

[10] Get repository content. https://docs.github.com/en/rest/reference/repos#get-repository-content.

[11] Github Actions? Making the Right Choice for You. https://blog.bitsrc.io/github-actions-or-jenkins-making-the-right-choice-for-you-9ac774684c8.

[12] GitHub Actions update: Helping maintainers combat bad actors. https://github.blog/2021-04-22-github-actions-update-helping-maintainers-combat-bad-actors/.

[13] GitHub investigating crypto-mining campaign abusing its server infrastructure. https://therecord.media/github-investigating-crypto-mining-campaign-abusing-its-server-infrastructure/.

[14] Hackers backdoor PHP source code after breaching internal git server. https://arstechnica.com/gadgets/2021/03/hackers-backdoor-php-source-code-after-breaching-internal-git-server/.

[15] Python package index. https://pypi.org/.

[16] Security hardening for GitHub Actions. https://docs.github.com/en/actions/learn-github-actions/security-hardening-for-github-actions.

[17] Set up Automated CI Systems with GitLab. https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/.

[18] Solarwinds supply chain attacks. https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html.

[19] Travis CI - Test and Deploy Your Code with Confidence. https://travis-ci.org/.

[20] Anonymous. Proof-Of-Concept Actions. https://kapravelos.com/projects/githubactions/poc-action-7369/.

[21] Anonymous. Reference POC. https://kapravelos.com/projects/githubactions/poc-action-7369/mutable/commit-referenced/action.yml.

[22] L. Bass, R. Holz, P. Rimba, A. B. Tran, and L. Zhu. Securing a deployment pipeline. In *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, 2015.

[23] Chrissie Buchanan. The problem with plugins. https://about.gitlab.com/blog/2019/09/27/plugin-instability/, September 2019. Last Accessed: 01-26-2022.

[24] Chromatic. Automate chromatic with github actions. https://www.chromatic.com/docs/github-actions#run-chromatic-on-external-forks-of-open-source-projects.

[25] cve search. git-vuln-finder. https://github.com/cve-search/git-vuln-finder.

[26] CircleCI Docs. Permissions overview. https://circleci.com/docs/2.0/gh-bb-integration#permissions-overview. Last Accessed: 01-26-2022.

[27] GitHub Docs. Repository roles for an organization. https://docs.github.com/en/organizations/managing-access-to-your-organizations-repositories/repository-roles-for-an-organization. Last AccessedL 01-28-2022.

[28] Gitlab Docs. Personal access token scopes. https://docs.gitlab.com/ee/user/profile/personal_access_tokens.html#personal-access-token-scopes. Last Accessed: 01-26-2022.

[29] Travis CI Docs. Travis CI's use of Bitbucket API Scopes. https://docs.travis-ci.com/user/bb-oauth-scopes/. Last Accessed: 01-26-2022.

[30] CircleCI Docx. Orbs Concept. https://circleci.com/docs/2.0/using-orbs/#semantic-versioning. Last Accessed: 01-26-2022.

[31] T. F. Düllmann, C. Paule, and A. v. Hoorn. Exploiting devops practices for dependable and secure continuous delivery pipelines. In *2018 IEEE/ACM 4th International Workshop on Rapid Continuous Software Engineering (RCoSE)*, 2018.

[32] Eric Cornelissen. Shell-injection through Action input. https://github.com/ericcornelissen/git-tag-annotation-action/security/advisories/GHSA-hgx2-4pp9-357g, 2020.

[33] W. Felidré, L. Furtado, D. A. d. Costa, B. Cartaxo, and G. Pinto. Continuous integration theater. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019.

[34] K. Gallaba and S. McIntosh. Use and Misuse of Continuous Integration Features: An Empirical Study of Projects That (Mis)Use Travis CI. *IEEE Transactions on Software Engineering*, 2020.

[35] GHarchive. GHarchive's open public dataset. https://www.gharchive.org/#bigquery.

[36] Github. Using labels with self-hosted runners. https://docs.github.com/en/actions/hosting-your-own-runners/using-labels-with-self-hosted-runners.

[37] Volker Gruhn, Christoph Hannebauer, and Christian John. Security of public continuous integration services. In *Proceedings of the 9th International Symposium on Open Collaboration*, WikiSym '13. Association for Computing Machinery, 2013.

[38] Michael Hilton, Nicholas Nelson, Danny Dig, Timothy Tunnell, Darko Marinov, et al. Continuous integration (CI) needs and wishes for developers of proprietary code. 2016.

[39] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. Trade-offs in continuous integration: Assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017. Association for Computing Machinery, 2017.

[40] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 426–437. IEEE, 2016.

[41] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[42] Jaroslav Lobacevski. GHSL-2020-235: Arbitrary command injection in wayou/turn-issues-to-posts-action. https://securitylab.github.com/advisories/GHSL-2020-235-wayou-turn-issues-to-posts-action/, 2021.

[43] Jaroslav Lobačevski. gajira-comment GitHub action vulnerable to arbitrary code execution. https://github.com/atlassian/gajira-comment/security/advisories/GHSA-hj6w-pm28-h8hf, 2020.

[44] Jaroslav Lobačevski. gajira-create github action vulnerable to arbitrary code execution. https://github.com/atlassian/gajira-comment/security/advisories/GHSA-hj6w-pm28-h8hf, 2020.

[45] Timothy Kinsman, Mairieli Wessel, Marco A Gerosa, and Christoph Treude. How do software developers use github actions to automate their workflows? *arXiv preprint arXiv:2103.12224*, 2021.

[46] LLVM Foundation. LLVM-Project. https://github.com/llvm/llvm-project.

[47] Michael Meli, Matthew R McNiece, and Bradley Reaves. How bad can it git? characterizing secret leakage in public github repositories. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2019.

[48] Microsoft. Visual Studio Code. https://github.com/microsoft/vscode.

[49] Håvard Myrbakken and Ricardo Colomo-Palacios. Devsecops: A multivocal literature review. In *Software Process Improvement and Capability Determination*. Springer International Publishing, 2017.

[50] National Security Agency. Datawave. https://github.com/NationalSecurityAgency/datawave.

[51] C. Paule, T. F. Düllmann, and A. Van Hoorn. Vulnerabilities in continuous delivery pipelines? a case study. In *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2019.

[52] A. Rahman, C. Parnin, and L. Williams. The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[53] Mary Sánchez-Gordón and Ricardo Colomo-Palacios. Security as culture: A systematic literature review of devsecops. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20. Association for Computing Machinery, 2020.

[54] Fred B Schneider. Least privilege and more [computer security]. *IEEE Security & Privacy*, 1(5):55–59, 2003.

[55] M. Shahin, M. Ali Babar, and L. Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 2017.

[56] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani. Detecting and mitigating secret-key leaks in source code repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015.

[57] Atlassian Support. Use OAuth on Bitbucket Cloud. https://support.atlassian.com/bitbucket-cloud/docs/use-oauth-on-bitbucket-cloud/#Scopes. Last Accessed: 01-26-2022.

[58] N. Tomas, J. Li, and H. Huang. An empirical study on culture, automation, measurement, and sharing of devsecops. In *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, 2019.

[59] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. in-toto: Providing farm-to-table guarantees for bits and bytes. In *Proceedings of the USENIX Security Symposium*, 2019.

[60] twbs. Bootstrap. https://github.com/twbs/bootstrap.

[61] C. Vassallo, S. Proksch, H. C. Gall, and M. Di Penta. Automated reporting of anti-patterns and decay in continuous integration. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[62] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C. Gall, and Massimiliano Di Penta. Configuration smells in continuous delivery pipelines: A linter and a six-month study on gitlab. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020. Association for Computing Machinery, 2020.

[63] H. Yasar. Experiment: Sizing exposed credentials in github public repositories for ci/cd. In *2018 IEEE Cybersecurity Development (SecDev)*, 2018.

[64] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering*, 25, 2020.

[65] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017. Association for Computing Machinery, 2017.

[66] Yaqin Zhou, Jing Kai Siow, Chenyu Wang, ShangQing Liu, and Yang Liu. Spi: Automated identification of security patches via commits, 2021.

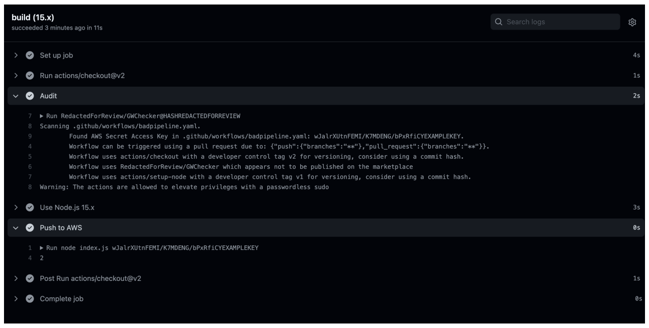
A GWChecker

To assist in mitigating simple security mistakes in the YAML configuration for CI/CD workflows, we developed a workflow auditing GitHub action, GWChecker[10]. GWChecker audits the workflow files by looking for plaintext secrets using regular expressions [47], tags for versioning, non-verified

[10] https://kapravelos.com/projects/githubactions/GWChecker/

actions or actions not published on the marketplace, and insecure triggers. In addition to this GWChecker enforces a pre commit hook that ensures that the files committed are not in '.github/workflow' To avoid having workflows that commit other workflow-related files to the repository.

Listing 3 shows a configuration file that is triggered via pull request on any branch, a plaintext AWS secret key, and uses developer-controlled version tags. The output of the workflow after GWChecker was added as the second step (after github/checkout) is shown in Figure 2.



```
build (15.x)
succeeded 3 minutes ago in 11s                                   Search logs

>  Set up job                                                                    4s
>  Run actions/checkout@v2                                                       1s
v  Audit                                                                         2s
   ▶ Run RedactedForReview/GWChecker@HASHREDACTEDFORREVIEW
   Scanning .github/workflows/badpipeline.yaml.
           Found AWS Secret Access Key in .github/workflows/badpipeline.yaml: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY.
           Workflow can be triggered using a pull request due to: {"push":{"branches":"**"},"pull_request":{"branches":"**"}}.
           Workflow uses actions/checkout with a developer control tag v2 for versioning, consider using a commit hash.
           Workflow uses RedactedForReview/GWChecker which appears not to be published on the marketplace
           Workflow uses actions/setup-node with a developer control tag v1 for versioning, consider using a commit hash.
   Warning: The actions are allowed to elevate privileges with a passwordless sudo
>  Use Node.js 15.x                                                              3s
v  Push to AWS                                                                   0s
   ▶ Run node index.js wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
   2
>  Post Run actions/checkout@v2                                                  1s
>  Complete job                                                                  0s
```

Figure 2: CI/CD log after adding GWChecker

```yaml
name: Node.js CI
on:
  push:
    branches: '**'
  pull_request:
    branches: '**'
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [15.x]
    steps:
    - uses: actions/checkout@v2
    - name: Use Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v1
      with:
        node-version: ${{ matrix.node-version }}
    - name: Push to AWS
      run: node index.js wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

Listing 3: Sample YAML file with plain-text secrets, broad triggers, and actions versioned with tags.

B Additional Tables

In this appendix we list the additional tables that can provide extra information, but not critical to the final results.

C Vulnerability Disclosure Details

To disclose the vulnerabilities we opted to open the issues to repositories that depend on vulnerable version of the action instead of sending an email to the repository owners. This is because not all of the owners of the repositories decide to make his/her email visible to public. Also, it is common process inside GitHub open source projects to open an issue to notify the owners of the repository about vulnerability in the code base.

We also indicated our contact information in the issue, *i.e.*, email address, for owners in case they want to contact with us. This actually, helped us to detect one of false positives, when maintainer of the third-party action himself contacted with us through the email and said that even though previous version does not contain CVE fixing commit, workflows that depend on *vulnerable* version can not be exploited.

Action name	**Total**	**VC**
actions/checkout	499,840	✔
actions/cache	104,563	✔
actions/setup-node	97,236	✔
actions/setup-python	76,906	✔
actions/upload-artifact	75,476	✔
actions/upload-release-asset	27,605	✔
actions/download-artifact	26,979	✔
actions/setup-java	26,630	✔
actions/setup-go	23,183	✔
actions/create-release	23,175	✔
Janealter/branch-pr-comment	20,030	✖
peaceiris/actions-gh-pages	19,051	✖
JamesIves/github-pages-deploy-action	16,670	✖
ad-m/github-push-action	15,452	✖
actions-rs/toolchain	12,367	✖
codecov/codecov-action	11,021	✔
actions/github-script	10,667	✔
JasonEtco/create-an-issue	10,376	✖
r-lib/actions/setup-r	9,839	✖

Table 7: Top 20 actions by number of how many times it was used. Note that sometimes action can be used multiple times inside the same workflow. Here VC column stands for verified creator. If we do not account for first-party actions maintained by GitHub (*actions* organization), there is only one third-party action (*codecov/codecov-action*) maintained by verified creator.