Automatic Security Assessment of GitHub Actions Workflows

Giacomo Benedetti
giacomo.benedetti@dibris.unige.it
DIBRIS - University of Genoa
Genova, Italy

Luca Verderame
luca.verderame@dibris.unige.it
DIBRIS - University of Genoa
Genova, Italy

Alessio Merlo
alessio@dibris.unige.it
DIBRIS - University of Genoa
Genova, Italy

ABSTRACT

The demand for quick and reliable DevOps operations pushed distributors of repository platforms to implement *workflows*. Workflows allow automating code management operations directly on the repository hosting the software.

However, this feature also introduces security issues that directly affect the repository, its content, and all the software supply chains in which the hosted code is involved in. Hence, an attack exploiting vulnerable workflows can affect disruptively large software ecosystems.

To empirically assess the importance of this problem, in this paper, we focus on the de-facto main distributor (i.e., GitHub). We developed a security assessment methodology for GitHub Actions workflows, which are widely adopted in software supply chains. We implemented the methodology in a tool (GHAST) and applied it on 50 open-source projects.

The experimental results are worrisome as they allowed identifying a total of 24,905 security issues (all reported to the corresponding stakeholders), thereby indicating that the problem is open and demands further research and investigation.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; • **Security and privacy → Software security engineering**.

KEYWORDS

Software Supply Chain, Software Supply Chain Security, GitHub Actions, Workflow security.

ACM Reference Format:
Giacomo Benedetti, Luca Verderame, and Alessio Merlo. 2022. Automatic Security Assessment of GitHub Actions Workflows. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '22), November 11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3560835.3564554

1 INTRODUCTION

Code repositories (also known as CRs) are a crucial part of every software development process. They are used to store, share, distribute, and version software and its dependencies. Indeed, code repositories are widely used in Software Supply Chains (SSCs) as they enable the storage of the core software as well as its distribution. In detail, the software's coding and building phase integrate several pieces of code (e.g., modules and third-party libraries) hosted in separate CRs. To this aim, the corresponding SSC will include a set of CRs contributing to the final software.

Also, in the last years, the push for automation procedures of Continuous Integration/Continuous Delivery (CI/CD) led the distributors of the most common platforms (e.g., GitHub, GitLab, Bitbucket) to introduce engines for the execution of *workflows*. A workflow is a sequence of actions aiming to automatize software's building, testing, and verification. In practice, they allow automatizing CI/CD processes directly into the repository without relying on external services. Workflows can be configured to run when manually triggered, at a scheduled time, or when a particular event on the repository occurs.

GitHub publicly released GitHub Actions (GHA) in 2019, and thanks to the provider's popularity, this service started to be widely used by developers. In a nutshell, GHA consists of an API and a dedicated engine that allow users to define and execute workflows on their repositories. GHA engines support the execution of workflows on GitHub dedicated machines or self-hosted ones.

The ability of workflows to manage and modify the content of CRs makes them an appealing target for attackers. For instance, several technical reports [22, 26] provided examples of attacks targeting GHA workflows to obtain control of GHA engines or inject malicious code into a repository. Through the SSC, the compromised CR is able to affect the other nodes that rely on it, e.g., the final software importing the compromised code in its codebase.

To cope with the security implications of workflows and GHA, in this paper, we provide the following contributions. First, we analyzed the GitHub security guidelines for hardening workflows, and we extracted a set of security constraints regarding confidential information, third-party workflows, permissions, and context variables. Then, we proposed a methodology to evaluate the security posture of GHA workflows, and we presented a prototype evaluation, called GitHub Action Security Tester (GHAST), based on the *Sunset* SSC security framework [1]. Finally, we conducted an in-the-wild experimental campaign on 50 GitHub - publicly available - repositories. The results allowed us to provide an overview of the security status of repositories. From this evaluation, we analyzed the workflows of 646 unique repositories involved in the SSCs of all the 50 projects and identified 20 previously unknown security vulnerabilities and 24,885 security misconfigurations.

In the rest of the paper, we first provide the necessary background (Section 2) to understand the proposed contribution. Then, we present our security assessment methodology to evaluate the security posture of workflows (Section 3). Moreover, we provide an implementation of the methodology (GHAST), and the techniques

SCORED '22, November 11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9885-5/22/11…$15.00
https://doi.org/10.1145/3560835.3564554

applied to rebuild the SSC and extract the required data in Section 4. Section 5 is dedicated to the analysis of results obtained applying GHAST in the wild against 50 open source projects. Section 6 discuss some related work. Finally, Section 8 draws some conclusions and discusses some future extensions of this work.

2 GITHUB ACTIONS: BACKGROUND & SECURITY ISSUES

This section provides a brief description of the concepts discussed in our work. In particular, we focus on GitHub Actions (GHA) and their core elements. Then we discuss vulnerabilities affecting workflows, specifically GHA workflows, and how an attacker can exploit them to compromise code repositories in the SSC.

2.1 GitHub Actions

GHA was introduced in 2019 with the aim to automate, customize, and execute software development workflows in code repositories. GHA attracted developers to ease automation routines in their projects independently of their size. At the time of writing (July 2022), the first 100 most starred repositories on GitHub adopted GHA.

GHA is an event-driven API provided by the GitHub platform to automate development workflows [6]. Among the different concepts introduced by GHA and required for their use, we give a brief explanation of the essential ones. The GHA API enables the definition of workflows in one or more files in YAML format that need to be stored in the `.github/workflows/` directory of the target GitHub repository.

A workflow is composed of one or more jobs. A job allows developers to define the environment and configuration where a sequence of tasks (namely, *steps*) will run. Each workflow can be associated with a list of events that trigger its execution. Examples of events include `pull request`, `push`, and `merge`. Runners are computing elements that host the execution of workflows for repositories. A Runner can be hosted both on GitHub dedicated servers and self-hosted machines.

At the start of each workflow run, GitHub automatically creates a unique `GITHUB_TOKEN` to authenticate the request, granting each runner privileges to interact with the repository on behalf of GHA. Administrators of the repository can set the permissions granted to the token to restrict access to specific resources or jobs. The default permissions can be either permissive or restricted. In the first case, the `GITHUB_TOKEN` has full access to the resources of the repository, while - in the second case - the capabilities are limited to read the content of the repository [5].

GHA offers developers artifacts to manage different aspects of workflow execution. Contexts are a way to access information about workflow runs, runner environments, jobs, and steps. Each context is an object that contains properties, which can be strings or other objects. Among the list of possible contexts, `github` [7] and `secrets` [8] are the most used ones.

The `github` context contains the event that triggered the workflow run plus some further information. It can be involved in the workflow execution to provide information like the actor who triggered the workflow or the body text of a newly created issue. The `secrets` context contains the names and values of secrets that are available to a workflow run. Secrets can be used to manage confidential information, like API keys and passwords. For example, the `GITHUB_TOKEN` is automatically included in the `secrets` context. Finally, GHA offers the possibility to make workflows reusable [9]. This mechanism enables anyone with access to the repository and the reusable workflow to call it from another workflow. Workflow reuse also promotes best practices by helping developers to use workflows that are well designed, have already been tested, and have been proven to be effective. Also, reusable workflows enable the definition of organization-wide libraries of workflows that can be used to speed up the creation of CI/CD pipelines.

2.2 Security Issues of GHA workflows

The introduction of workflows enabled code repositories to become an integral part of the CI/CD pipeline. In particular, GHA workflows can operate on the code repository by programmatically adding, removing, and modifying its content. Those capabilities, however, can directly affect the confidentiality, integrity, and availability of the software and the associated metadata information.

In detail, an attacker can leverage security weaknesses and misconfigurations in the definition and execution of a GHA workflow. As some elements of the workflow can be manipulated by the actor triggering the action, a malicious actor can craft specific inputs to cause unexpected execution flows in the workflow.

For instance, context elements are susceptible to several security weaknesses and misuse, as stated in the official documentation [7]. On one hand, the `secrets` context can contain sensitive information (e.g., a token or a key) that is encrypted using the default key and can be safely manipulated inside a workflow environment. On the other hand, a poorly configured workflow may grant direct access to a secret, thereby allowing an attacker either to send it to unintended hosts or explicitly print it to the log output [12].

As for secrets, direct access to variables of the `github` context can lead to command injection attacks. In detail, the `github` context allows storing information that directly depends on the user's input (e.g., the body of an event). Those data can be manipulated and executed inside by a Runner using the `run` job. Unfortunately, a poorly-configured workflow can allow attackers to inject a crafted input to trigger its direct execution inside the Runner, thereby compromising the repository or the execution environment.

An example of vulnerable scenario is depicted in Figure 1. In this scenario, the `issue.title` element is directly executed in the run step. If an attacker can inject in the title of the issue the string: `New malicious issue title" && bash -i >& /dev/tcp/0.tcp.eu.-ngrok.io/14872 0>&1 && echo "`, she can obtain a reverse shell on the remote machine hosting the Runner.

In addition to the security weaknesses of workflow elements, two other features of GHA have direct impact on the security of workflows, i.e., *permissions* and *reusable workflows*.

In the first case, permissions can affect the capability of a successful attack on a repository [11]. To this aim, developers should enforce the least privilege principle by assigning the least set of permissions tokens and workflows, and differentiate - when possible - the set of permissions granted to each job.

Finally, reusable workflows allow repository owners to import both workflows belonging to their organization and workflows

```yaml
on:
  issues:
    types: [opened]

jobs:
  vuln_job:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2

      - name: vuln_step
        run: |
         echo "ISSUE TITLE: ${{github.event.issue.title}}"
```

Figure 1: Example of Github Workflow vulnerable to command injection attacks.

developed by third parties that are publicly available [13]. In this latter case, the caller workflow does not control the imported one (the callee). The introduction of unmonitored workflows increases the attack surface since the callee may include potential weaknesses and misconfigurations, especially if the imported version is not updated. Also, attackers can manipulate reusable workflows by exploiting tag-reuse attacks to trick the caller workflow into importing a different version of the callee, which has been maliciously re-tagged. To prevent such issues, the best practice suggests selecting the workflow version using a commit hash (e.g., `workflow_name@cdd...08c`) [13].

3 SECURITY ASSESSMENT METHODOLOGY

We propose a novel methodology composed of two phases, namely *Workflow Collection* and *Workflow Security Evaluation* in order to evaluate the security of GHA in a software supply chain. The Workflow Collection phase is devoted to analyze the SSC and extract a model that includes all the involved CRs and - for each repository - the set of GHA workflows. The Workflow Security Evaluation phase performs a security analysis of each GHA workflow to assess five security categories extracted from the analysis of the official documentation [6] and the security hardening guidelines [10]. The result of the analysis is a report containing a list of security issues affecting the workflows used in the SSC, classified according to their exploitability.

3.1 Workflow Collection

The first part of the methodology takes as input a folder containing the software under test (hereafter, SUT). The procedure parses the code and configuration files of the project to recursively identify software dependencies, code repositories, and distribution networks composing the Software Supply Chain of the SUT. From such a piece of knowledge, the methodology builds a direct graph structure where nodes include any code repository involved in the SSC, while edges represent the relationships among them.

Then, the procedure focuses on each CR to extract its GHA workflows (if any). More specifically, the methodology relies on GitHub API to search and retrieve the YAML files of the available workflows. The set of workflows is then linked to the corresponding CR node in the graph.

```
( ISSUE_TITLE_INJECTION,
    "issues[opened]",
    "vuln_job",
    "",
    2,
    "echo \"ISSUE TITLE: ${{github.event.issue.title}}\"",
    1
)
```

Figure 2: Example of a COMMAND_INJECTION security issue detected in the workflow of Figure 1.

3.2 Workflow Security Evaluation

The Workflow Security Evaluation phase scans the set of GHA workflows to detect security vulnerabilities and misconfigurations that can affect code repositories in the SSC.

The detection logic of this phase is based on the evaluation of a set of *security categories* that are mapped in constraints - i.e., security requirements that workflows must comply with - and *security checks* - i.e., technical controls to assess the enforcement of constraints. The list of categories, constraints, and checks applicable to GHA workflows derive from a manual review of the GitHub guidelines for workflow hardening [10] and from the official GHA documentation [6]. Table 1 reports the results of our analysis.

In particular, we identified four categories from the analysis of GH guidelines concerning the security of GHA workflows, namely *Confidential Data Disclosure*, *Command Injection*, *Third-party Workflows*, and *Workflow Permissions*. Also, we extended those categories with *Triggering Events* as they represent the entry point for workflows.

The evaluation phase parses each GHA workflow to *i*) execute the security checks detailed in Table 1 to detect security vulnerabilities and misconfigurations, and *ii)* evaluate the exploitability of the findings by identifying the events and preconditions that allow triggering the corresponding security issue.

The result of the evaluation is a set of tuples containing the detected security issue, some information related to the issue (i.e., the name of the job and the affected step, the position of the issue, and the print of the affected line), the event triggering the issue (i.e., the entry point), and an evaluation on its exploitability. If an issue belongs to a job triggered by multiple events, the evaluation will contain a separate tuple for each event to enable the filtering of specific results.

```
1       ( security_issue_type,
2         triggering_event,
3         job_name,
4         step_name,
5         step_position,
6         issue_line,
7         exploitability_score )
```

Listing 1: Structure of a security issue identified by the Workflow Security Evaluation phase.

Security Categories	Security Constraints	Security Checks
Confidential Data Disclosure	- Registering generated values as secrets. - Registering modified values as secrets.	(SC-1) Check the use of the `secrets` context outside environment. (SC-2) Check the use of `secrets` context for generating new data.
Command Injection	- Do not directly use github context in scripts.	(SC-3) Check the use of `github` subcontexts inside run steps.
Third-Party Workflows	- Audit the use of branch names, emails, and external inputs. - Audit information passed to TP actions. - Keep third-party workflows up-to-date.	(SC-4) Verify the version of reusable workflows. (SC-5) Verify the use of the pinning Commit Tag.
Workflow Permissions	- Specify permissions to avoid uncontrolled access.	(SC-6) Check if the workflows enforce the least privilege principle.
Triggering Events	- Audit the kind of events that trigger the workflow. - Audit the filters applied to triggering events.	(SC-7) Verify which is the exploitability score of the event.

Table 1: List of security categories, requirements, and checks of the Workflow Security Evaluation phase.

Listing 1 provides the structure of a tuple, while Figure 2 shows an example of a tuple for the scenario depicted in Figure 1.

The rest of this section will detail the evaluation of the exploitability of events and the type of security issues supported by the methodology.

3.2.1 Execution of Security Checks. The execution of the security checks listed in Table 1 enables the identification of a set of security issues targeting the workflow under test. The methodology labels an identified security issue into two different groups, i.e.:

Vulnerability. A flaw in the workflow that is directly exploitable. For example, a misuse of the `github.context` API enables attackers to execute command injection attacks.

Misconfiguration. A configuration error that makes the environment, the CR, or the SSC vulnerable. For example, the improper configuration of `tag` variable inside a workflow may lead to the import of an outdated third-party workflow.

In detail, the security issues belonging to the Confidential Data Disclosure and the Command Injection categories can be identified using pattern verification techniques.

Secrets can be used in workflows through the `secrets` context [8]. A particular secret is accessed declaring the context and then its name. Secrets are expected to be accessed inside of an environment. Indeed, when they are accessed from outside, secrets can be exfiltrated by an attacker (e.g., printed in a log or sent to a remote host) that has access to the workflow. Then, the security checks for Secrets consist of assessing if secrets are used in the environment part of the workflows (`SC-1`) and, if it is not the case, if they are manipulated in order to generate confidential data (`SC-2`). Indeed, `SC-2` allows assessing when a secret is involved in a computation and then exposed to potential exfiltration.

Similarly, the methodology can verify the presence of command injection attacks in workflows by evaluating the usage of `github` context [7] in run steps. In detail, the syntax `github.*` is used to access variables contained in the GitHub Context. This context contains many subcontexts linked to different aspects of the workflows. For example, the `github.event` context contains all the information of the triggering event. Some parts of the `github` context can be manipulated externally by an actor. Hence, an attacker can inject a command into a vulnerable instruction through this context. For example, in the scenario of Figure 1, the attacker is able to exploit the run step through the issue title contained in `github.event.issue.title`. For this reason, the methodology evaluates the use of `github` subcontexts that can be exploited by an attacker to affect the workflow run (`SC-3`).

The command injection issues is further distinguished between *conditional* and *unconditional*. The first case consists of command injection attacks in a branch of a conditional statement, thereby requiring the attacker to trigger the appropriate branch to exploit the vulnerability. In the unconditional case, the vulnerable statement will be executed directly.

To evaluate the security checks belonging to the Third-party Workflows category, the methodology relies on the GitHub API to obtain the latest version of the specific TP workflow, and compare it with the version requested by the workflow under test. If the used TP workflow is out-of-date (`SC-4`) or the workflow under test does not pin a specific commit hash (i.e., using the `<wf_name>@<commit_hash>` syntax) (`SC-5`), the methodology marks the issue as a security misconfiguration.

Finally, the methodology evaluates the workflow under test to check if the permissions declared in the workflow follow the principle of least privilege. Following the least privilege principle, the methodology checks if the required permission matches the capabilities required by the workflow (`SC-6`).

3.2.2 Exploitability Score. The only way for an attacker to execute a workflow and, thus, exploit a security issue is by triggering one or more events defined in the workflow. To this aim, the methodology evaluates the potential entry points by assessing the configuration of workflow events and the type of conditions that enables their activation (`SC-7`).

Events are influenced by filters [14] that can be applied to specialize the triggering actions. For example, an *Issues* event can have up to 16 activity types (e.g., opened, deleted, labeled, ...). To this aim, the use of filters directly affects the exploitability of a workflow.

The methodology classifies events using three security levels based on the complexity an attacker has to face to stimulate a particular event. We consider the complexity as the number and type of preparatory activities to get the required assets (e.g., privileges, knowledge) to trigger an event.

Restricted (1) The attacker needs to achieve a multi-stage attack to be able to trigger the specific workflow event. The attacker can succeed only with the help, whether intentionally or not, of one of the owners of the repositories. For example, an attacker cannot trigger a push event unless a repository owner grants him the maintainer status.

Supervised (2) The attacker must match particular conditions in order to trigger the event. For example, a pull request can be triggered, but the maintainer has to accept it and consequently start the workflow.

Unsupervised (3) The attacker does not need any granted permission by the repository owners. He can trigger the workflow at any time by means of external action. For example, an issue opening event can be triggered by any GitHub user with access to the repository.

4 PROTOTYPE IMPLEMENTATION

We implemented the proposed methodology in a command-line tool - the GitHub Actions Security Tester (GHAST) - composed of three modules, as depicted in Figure 3. The source code of GHAST is publicly available[1].

The first module (*SSC Builder*) extracts the model of the SSC starting from a folder containing the source code of SUT. The second one (*WF Extractor*) extracts the list of code repositories from the SSC model. It extracts the set of GHA workflows from these collected repositories. The last module (*WF SecAnalyzer*) executes the workflow security evaluation described in Section 3.2 to produce the final security report.

For the *SSC builder*, we rely on a state-of-the-art research tool for the rebuilding and modeling of software supply chains [1]. In detail, the module requires the folder containing the SUT as input, then it recursively identifies the SSC assets, including code repositories. The output model is a direct graph structure stored in a Neo4J Graph database [19].

The *WF Extractor* queries the SSC model to extract the set of repositories that uses GHA workflows. Then, the module scans each repository using the GitHub API and downloads all the workflow YAML files contained in the corresponding `.github/workflows` folder. The result of this operation is the collection of workflows used in the SSC.

Finally, the *WF SecAnalyzer* evaluates each workflow by applying the security checks listed in Table 1. To do so, the module parses each YAML file to extract the following GHA elements:

- *triggering-events*, for the computation of the exploitability score (SC-7). These elements also contain event filters that are included in the score computation.
- *Runs*, for the evaluation of command injection and confidential data disclosure issues (SC-1, SC-2, SC-3).
- *TP-workflows*, for the evaluation of commit pinning and the workflow versions (SC-4, SC-5).
- *Permissions*, for the evaluation of permissions both at workflow level and at job level (SC-6).

```
"<work_test>":
{
  "events": ["issues",3],
  "issues": [
    ["MISCONF_PERM_GLOBAL", "<job_A>",...],
    ["OUTDATED_WF", "<job_A>",...],
    ["CI_ACTOR","<job_B>",...],
    ["OUTDATED_WF", "<job_A>",...],
    ["OUTDATED_WF", "<job_B>",...]
  ]
}
```

Listing 2: Extract of a sample security report produced by GHAST.

[1]https://github.com/Mobile-IoT-Security-Lab/GHAST

The output of the evaluation is a security report in JSON format that contains - for each repository (and workflow) of the SSC - the set of security issues identified by the tool. Listing 2 shows an extract of an output file. In the example, the tool identified five issues in the workflow `work_test`, associated with the event `issues`, i.e., a command injection exploiting the actor username, a misconfiguration of the permissions, and three outdated third-party workflows. The exploitability score of the event triggering those finding is 3, i.e., unsupervised.

5 EXPERIMENTAL EVALUATION

We conducted an experimental campaign to test the applicability and efficacy of GHAST in the wild on software repositories available on GitHub. In detail, we ran the tool against 50 public repositories randomly taken from the top 100 Python-based projects on GitHub in July 2022. The repositories are actively maintained and used by the community (e.g., the dataset has an average number of stargazers per repository above 20k). Table 2 details the dataset's characteristics regarding the number of stargazers, forks, contributors, open issues, and commits.

	MIN	MAX	AVG
Stargazers	4,365	191,386	22,227
Forks	307	36,334	4,708
Contributors	3	30	25
Issues	1	1,707	232
Commits	40	52,887	4,729

Table 2: Number of stargazers, forks, contributions, issues and commits (min, max, and avg values) of the dataset.

The tool extracts the SSC for each repository, identifies the CRs compatible with GHA workflows, and processes the security report. The experiments were hosted on a virtual machine running Ubuntu 20.04 with 8 processors and 32GB RAM.

5.1 Experimental Results

GHAST was able to reconstruct the SSC of the 50 projects obtaining 646 unique CRs in approximately 14 hours. Then, the tool extracted 131,168 GHA workflows and executed the security checks.

The analysis allowed for the identification of 24,905 security issues, i.e., 20 security vulnerabilities and 24,885 misconfigurations. All the findings have been directly reported to the maintainers of the repositories.

Security Vulnerabilities. Figure 4a reports the distribution of security vulnerabilities identified using SC-1, SC-2, and SC-3 according to their exploitability score.

The security checks on Secrets SC-1 and SC-2 reported 10 jobs that are not using the `secrets` context inside an environment. Such behavior enables jobs to expose those secrets, e.g., by printing in a log output or sending them to an external host. Also, it is worth noticing that most of these vulnerabilities (i.e., 7 out of 10) can be triggered using unsupervised events.

GHAST also reported 9 workflows vulnerable to command injection attacks (SC-3), of which two do not need any granted permission by the repository owner to be exploited (score equals to

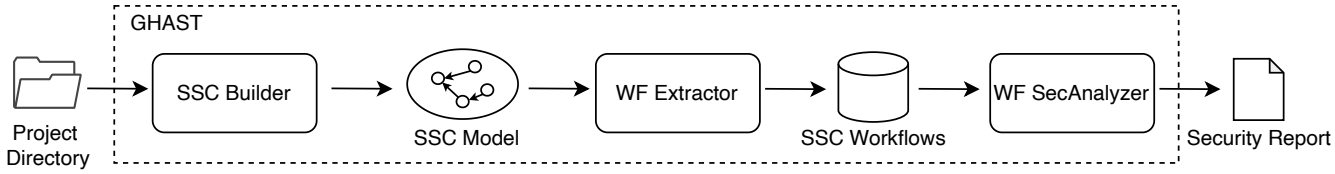
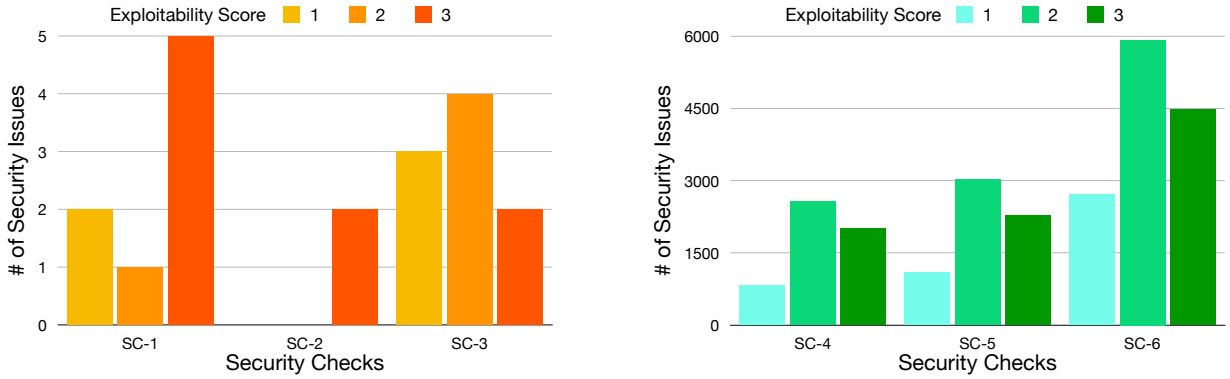


Figure 3: The GHAST architecture.



(a) Vulnerability Issues identified.

(b) Misconfiguration Issues identified.

Figure 4: Issues for different exploitability scores.

3). Also, 3 out of 9 are *unconditional* command injections, i.e., that the affected step is not included in any conditional branch, thereby easing the exploitation process.

Misconfigurations. For SC-4 and SC-5, we considered both checks passed when the workflow uses the commit hash of the latest version available of the third-party workflow. In particular, we found that this condition has never been verified on our sample set, meaning that even if a workflow uses the `latest` tag, it is still vulnerable to tag-reuse attacks. The experimental campaign allowed the identification of 5,384 steps that do not use the latest version of a third-party workflow. Also, the evaluation of SC-5 resulted in 6,388 steps that do not pin a specific version of a third-party workflow.

Finally, SC-6 reported several misconfigurations in the definition of workflow permissions. In detail, Figure 5 shows that 38% of workflows declared the permissions granted for their execution at the workflow level instead of for each job (as suggested by the security guidelines [10]). Furthermore, 62% of workflows do not declare any specific permission, thereby allowing the execution of the workflow with the default permissions assigned to the GITHUB_TOKEN.

5.2 Manual Validation of the Vulnerabilities

We manually revised the 20 identified vulnerabilities in order to verify the presence of false positives. To conduct this analysis, we downloaded the affected repositories locally and instantiated a local GHA Runner for the corresponding workflows to reproduce the exact conditions to test the exploitability of the attack. The result of manual analysis confirmed that all the vulnerabilities identified by GHAST are true positives.

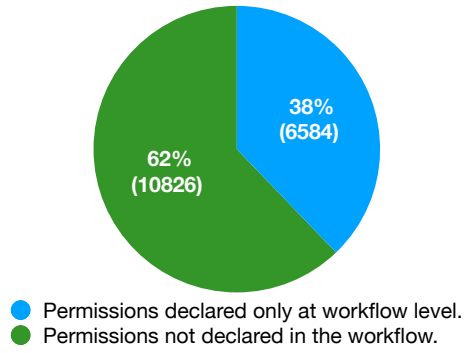


Figure 5: Distribution of Misconfigured Permissions (SC-6).

Listing 3 shows an anonymized GHA workflow containing a command injection issue triggered by a pull request event reported by GHAST. The workflow belongs to a repository with more than 10,000 stars, 500 forks, and 3,000 commits.

```
name: Pull Request Validation
on:
  pull_request:
    types: [opened, synchronize, reopened, edited]
jobs:
  <anon-job-name>:
    name: -------------
    runs-on: ubuntu-latest
    steps:
```

```
10      - name: <anon-step-name>
11        uses: actions/checkout@v2
12        with:
13          ref: ${{github.event.pull_request.head.sha
                }}
14          fetch-depth: 0
15      ...
16      - name: <anon-step-name>
17        run: |
18          cat << EOF | egrep -qsi '^disable-check:
                .*\<commit-count\>'
19          ${{github.event.pull_request.body}}
20          EOF
```

Listing 3: Excerpt (anonymized) of a workflow vulnerable to a command injection attack.

In such a workflow, it is possible to replicate a similar attack that affects the workflow in Figure 1 to initiate a reverse shell and access the Runner executing the workflow.

In detail, we crafted a payload to exploit the interpolation of the `github.event.pull_request.body` variable in the workflow, and we submitted it as a new pull request in the target repository, as shown in Figure 6. If a maintainer accepts the pull request, she will trigger the vulnerable workflow. When the GHA reaches the run step of Listing 3, the Runner executes rows 17-20 using bash.

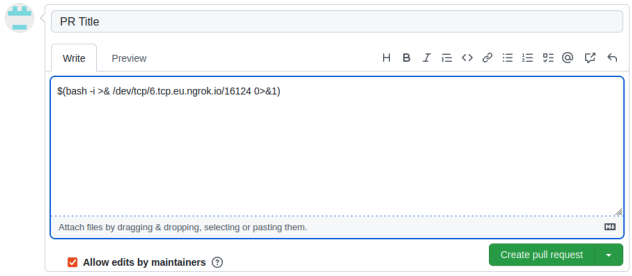


Figure 6: Creation of a malicious pull request on the target repository.

In this case, the run step contains an `here documents redirection` [16] that allows command substitution [15]. Hence, the GHA Runner executes the malicious payload from the pull request and opens a TCP connection towards a remote server on port 16147. Figure 7 shows the listener active on local port 10000 mapped to the remote server, where the reverse shell is open. At this point, the attacker has complete access to the Runner instance, e.g., she can execute any command with Runner's privileges.

6 RELATED WORK

To the best of our knowledge, the research work targeting the evaluation of GHA in the context of the software supply chain is limited. On the one hand, several organizations like ENISA [4], NIST [3], and OWASP [23] discussed the security issues in the Supply Chain, giving a focus on SSCs and the impact of using insecure third-party software in the development pipeline. Also, several technical reports investigated attacks targeting SSCs of popular software like Solarwind [18] and Log4J [24].

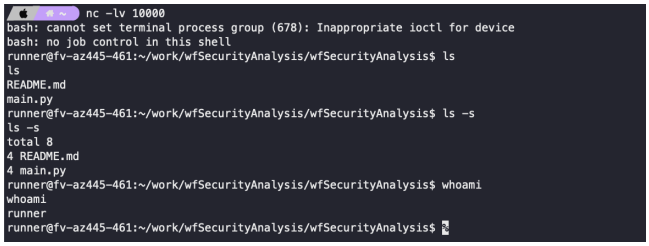


Figure 7: Attacker terminal with the reverse shell to the affected Runner.

On the other hand, the scientific community mainly investigated the problem of ensuring the integrity of the final software in SSCs that include different actors (e.g., code repository platforms, software library binaries, distribution networks). Several works [21, 17] propose the use of reproducible builds to create an independently-verifiable path from source to binary code. To cope with the integrity problem in the last part of the development process, Vu et al. in [29] show how the software can be compromised between the production of source code and the building process and how it is possible to mitigate this security problem. Considering the entire CI/CD pipeline, the in-toto framework [28] proposes holistic integrity enforcement of a software supply chain. It gathers cryptographically verifiable information about the chain to accomplish its objectives.

To the best of our knowledge, only two open source projects explicitly targetting the evaluation of code repositories: GitGat [25] and GitHub Workflow Auditor [27]. The first one is a project released in June 2022 that takes advantage of the Open Policy Agent (OPA) [2] to evaluate security policies for GitHub's organization, repositories, and user accounts. GitHub Workflow Auditor is a command line tool released by Tinder in July 2022 for the security assessment of workflows. The tool scans a specific organization, user, or repository to detect potential security issues in secrets usage and external inputs. Unlike GHAST, GitHub Workflow Auditor can neither reconstruct the SSC of a project nor extract all the workflows associated with different CRs to detect security issues. Also, the tool does not provide any security evaluation of Third-party Workflows, Workflow Permissions, and Triggering Events. Also, it is worth noticing that both tools were released in Q2 2022, thereby suggesting a growing interest in the topic.

Finally, Koishybayev et al. [20] studied the security of GitHub CI workflows in parallel with our research. In detail, their work identified four security properties (permissions, privileges, code controls, and secrets) affecting workflows in GitHub CI and other VCS platforms. Also, they released a PoC tool called GWChecker to assist in analyzing GHA workflows. As for the other works on code repositories, their research focuses on single workflows, and neither takes into account nor models the dependencies among GHA workflows, thereby lacking an evaluation of the workflows on the entire software supply chain of the SUT. In addition, the authors do not consider events and their exploitability to evaluate the impact of vulnerabilities and security misconfigurations of GHA workflows.

7 LIMITATIONS AND DISCUSSION

Limitations. The proposed methodology obtained promising results during the experimental evaluation and allowed us to assess the applicability and efficacy of GHAST. In addition, the manual validation of the identified vulnerabilities confirms the reliability of the approach.

Still, our solution suffers from some limitations. First, the security evaluation methodology has inherited limits. Although some security controls (e.g., SC-4 and SC-5) are based on the definition syntax of GHA workflows, having an intrinsically low rate of false positives. Other SCs (e.g., SC-1 and SC-2) are based on regular expressions and pattern matching techniques that cannot keep into account other information, like, e.g., the workflow execution context. As a consequence, this could, in principle, lead to potential false positives.

In our work, we tried to reduce the rate of false positives by thoroughly reviewing the GHA documentation and real-world GHA workflows. This approach allowed us to catch some potential corner cases and forge a heuristic to evaluate the SCs. However, it is still possible that GHAST does not identify deviant cases; therefore, we argue that an assessment in the wild may lead to false positives.

Moreover, the evaluation of the GHA workflows depends on correctly identifying all the software repositories involved in the SSC and the associated GHA workflows. To this aim, an error in the parsing process of the SUT can lead to not identifying a subset of CRs and the related workflows, thereby leading to potential false negatives. In this respect, GHAST is based on the Sunset Framework [1] and shares the same parsing limitations.

Vulnerability Disclosure and Security Implications. We disclosed all the identified vulnerabilities to the owners of the affected repositories via email. Each email contained a description of the vulnerability, the potential impact of its exploitation, and the GHAST report. At the time of writing, 6 out of 20 repository owners acknowledged our notifications within a period of 1 month after the disclosure.

As with any security assessment tool, attackers can use GHAST to discover vulnerable projects. When releasing a tool, there is a fundamental trade-off between helping repository owners versus facilitating the attackers.

Given the increased attention to software supply chain security and the existence of similar publicly available tools (e.g., [20] [25], and [27]), we feel that the benefits to repository owners for publicly releasing GHAST outweigh the harms. Attackers have the resources and capabilities to replicate GHAST, whereas many repository owners do not. Also, GHAST does not explicitly report how to exploit security misconfigurations. Finally, publicly releasing GHAST will also encourage further research and would help repository maintainers (e.g., tools to patch vulnerable workflows automatically).

8 CONCLUSION

We investigated the security issues affecting GHA workflows to understand their security impact on the software supply chain. We produced an analysis of GHA aspects impacting the security of repositories. Then, we leveraged our analysis into a methodology for automatically assessing the presence of security issues in workflows.

We implemented the methodology in GHAST (GitHub Actions Security Tester). GHAST runs on a project source code, taking advantage of the *Sunset* SSC security framework to retrieve the SSC. Then, it extracts and analyzes the GitHub Actions workflows applying the security checks designed in the methodology.

Using GHAST against 50 open source projects produced relevant experimental results regarding security issues. We analyzed the results, providing an overview of the current security landscape of GHA workflows. We identified and manually revisioned the security vulnerabilities discovered through GHAST.

For future work, we plan to deepen the analysis of third-party workflow to *i)* better understand their involvement in the SSC, *ii)* extend GHAST with automatic security assessment of third-party workflow code.

REFERENCES

[1] Giacomo Benedetti, Luca Verderame, and Alessio Merlo. 2022. Alice in (software supply) chains: risk identification and evaluation. In *Quality of Information and Communications Technology*. Springer International Publishing, Cham, 281–295. ISBN: 978-3-031-14179-9. DOI: 10.1007/978-3-031-14179-9_19.

[2] Open Policy Agent contributors. 2022. Open policy agent. Retrieved July 22, 2022 from https://www.openpolicyagent.org.

[3] Cybersecurity and Infrastructure Security Agency. 2021. *Defending Against Software Supply Chain Attacks*. Retrieved July 22, 2022 from https://www.cisa.gov/sites/default/files/publications/defending_against_software_supply_chain_attacks_508_1.pdf.

[4] European Union Agency for Cybersecurity. 2021. *ENISA threat landscape for supply chain attacks*. Publications Office. Retrieved July 22, 2022 from https://data.europa.eu/doi/10.2824/168593.

[5] GitHub. 2022. Automatic token authentication. Retrieved July 22, 2022 from https://docs.github.com/en/actions/security-guides/automatic-token-authentication.

[6] GitHub. 2022. GitHub actions. Retrieved July 22, 2022 from https://docs.github.com/en/actions.

[7] GitHub. 2022. GitHub contexts - github. Retrieved July 22, 2022 from https://docs.github.com/en/actions/learn-github-actions/contexts#github-context.

[8] GitHub. 2022. GitHub contexts - secrets. Retrieved July 22, 2022 from https://docs.github.com/en/actions/learn-github-actions/contexts#secrets-context.

[9] GitHub. 2022. Reusing workflows. Retrieved July 22, 2022 from https://docs.github.com/en/actions/using-workflows/reusing-workflows.

[10] GitHub. 2022. Security hardening for github actions. Retrieved July 22, 2022 from https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions.

[11] GitHub. 2022. Security hardening for GitHub actions: restricting permissions for tokens. Retrieved July 22, 2022 from https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions#restricting-permissions-for-tokens.

[12] GitHub. 2022. Security hardening for GitHub actions: using secrets. Retrieved July 22, 2022 from https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions#using-secrets.

[13] GitHub. 2022. Security hardening for github actions: using third-party actions. Retrieved July 22, 2022 from https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions#using-third-party-actions.

[14] GitHub. 2022. Using filters. Retrieved July 22, 2022 from https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#using-filters.

[15] GNU. 2020. Bash reference manual - command substitution. https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html#Command-Substitution.

[16] GNU. 2020. Bash reference manual - here documents. https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html#Here-Documents.

[17] Pronnoy Goswami, Saksham Gupta, Zhiyuan Li, Na Meng, and Daphne Yao. 2020. Investigating the reproducibility of npm packages. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 677–681. DOI: 10.1109/ICSME46990.2020.00071.

[18] Trey Herr, Will Loomis, Emma Schroeder, Stewart Scott, Simon Handler, Tianjiu Zuo, and Atlantic Council of the United States. 2021. *Broken trust: lessons from Sunburst*. Retrieved July 22, 2022 from https://www.atlanticcouncil.org/in-depth-research-reports/report/broken-trust-lessons-from-sunburst/.

[19] Neo4j Inc. 2022. Neo4j graph database. Retrieved July 22, 2022 from https://neo4j.com/product/neo4j-graph-database/.

[20] Igibek Koishybayev, Aleksandr Nahapetyan, Raima Zachariah, Siddharth Muralee, Bradley Reaves, Alexandros Kapravelos, and Aravind Machiry. 2022. Characterizing the security of github CI workflows. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, (Aug. 2022), 2747–2763. ISBN: 978-1-939133-31-1. https://www.usenix.org/conference/usenixsecurity22/presentation/koishybayev.

[21] Chris Lamb and Stefano Zacchiroli. 2022. Reproducible builds: increasing the integrity of software supply chains. *IEEE Software*, 39, 2, 62–70. DOI: 10.1109/MS.2021.3073045.

[22] Magno Logan. 2022. GitHub action runners: analyzing the environment and security in action. Retrieved July 22, 2022 from https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/github-action-runners-analyzing-the-environment-and-security-in-action.

[23] OWASP. 2020. *OWASP Software Component Verification Standard*. Retrieved July 22, 2022 from https://owasp.org/www-project-software-component-verification-standard/.

[24] Radware. 2021. Log4shell: critical log4j vulnerability. Retrieved July 22, 2022 from https://www.radware.com/security/threat-advisories-and-attack-reports/log4shell-critical-log4j-vulnerability/.

[25] Scribe. 2022. Gitgat. Retrieved July 22, 2022 from https://github.com/scribe-public/gitgat.

[26] Thomas Segura. 2022. GitHub actions security best practices. Retrieved July 22, 2022 from https://blog.gitguardian.com/github-actions-security-cheat-sheet/.

[27] Tinder. 2022. Gh-workflow-auditor. Retrieved July 22, 2022 from https://github.com/TinderSec/gh-workflow-auditor.

[28] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. 2019. In-toto: providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, (Aug. 2019), 1393–1410. ISBN: 978-1-939133-06-9. https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias.

[29] Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. 2021. Lastpymile: identifying the discrepancy between sources and packages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ESEC/FSE 2021). Association for Computing Machinery, Athens, Greece, 780–792. ISBN: 9781450385626. DOI: 10.1145/3468264.3468592.