Scientific Research Publishing

Journal of Software Engineering and Applications, 2024, 17, 321-339
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Sher: A Secure Broker for DevSecOps and CI/CD Workflows

Pranau Kumar, Vijay K. Madisetti

School of Cybersecurity and Privacy, Georgia Institute of Technology, Atlanta, USA
Email: pranau@gatech.edu, vkm@gatech.edu

How to cite this paper: Kumar, P. and Madisetti, V.K. (2024) Sher: A Secure Broker for DevSecOps and CI/CD Workflows. *Journal of Software Engineering and Applications*, **17**, 321-339.
https://doi.org/10.4236/jsea.2024.175018

Received: April 25, 2024
Accepted: May 26, 2024
Published: May 29, 2024

Abstract

GitHub Actions, a popular CI/CD platform, introduces significant security challenges due to its integration with GitHub's open ecosystem and its use of flexible workflow configurations. This paper presents *Sher*, a Python-based tool that enhances the security of GitHub Actions by automating the detection and remediation of security issues in workflows. Self-Hosted Ephemeral Runner, or Sher, acts as a broker between GitHub's APIs and a customizable, isolated environment, analyzing workflows through a static rules engine and automatically fixing identified issues. By providing a secure, ephemeral runner environment and a dynamic analysis tool, Sher addresses common misconfigurations and vulnerabilities, contributing to the resilience and integrity of DevSecOps practices within software development pipelines.

Keywords

CI/CD Pipelines, GitHub, GitOps, DevSecOps, Isolation, Security, SAST

1. Introduction

The flexibility and responsiveness offered by modern software development practices such as DevOps and DevSecOps has enabled rapid development cycles [1]. Typically, there are several components and processes associated with DevOps—Continuous Integration (CI), Continuous Delivery (CD), Version Control, Monitoring, Collaboration, Infrastructure as Code (IaC), Configuration Management, and Automation to mention some of them [2]. Similarly, DevSecOps comprises a number of steps: 1) Identify DevSecOps needs, 2) Verify Code Dependencies, 3) Adopt the Right Tools, 4) Identify Threat Models, 5) Adopt Automation Tools, 6) Build Security Controls and Vulnerability Detection into CI/CD Pipelines, and 7) Monitor & Deploy.

Continuous Integration and Continuous Delivery (CI/CD) are a set of

processes that have become synonymous with DevOps. CI lets developers regularly merge their code changes to a central version-controlled repository, where automated tests and builds can be run against them. This helps improve software quality by detecting issues early in the development process.

CD, on the other hand, helps deploy and host the code changes made by developers to various infrastructure locations such as production, staging, or testing environments. CI and CD together form a sequential series of steps starting from code changes to deployment called a "pipeline" [3].

Tools that enable the setup of CI/CD pipelines are generally not developed by engineering teams, with most of them opting to use offerings of commercial CI/CD providers. This is primarily because of the complexity involved in deploying and maintaining the software that controls this infrastructure. A few popular CI/CD providers include Jenkins [4], TravisCI [5], CircleCI [6], GitLab CI [7], and GitHub Actions [8]. These providers let developers define "workflows" or automation tasks that can be run in response to various events or "triggers". An example of such a workflow can be running a static analysis tool against the codebase whenever a developer merges changes to the repository and then reporting the results.

While these tools let development teams be very productive, a major shortcoming associated with current tools is the inclusion of security as an afterthought. This is a concern because supply chain security is directly dependent on the security of CI/CD pipelines. A compromise in the pipeline can let attackers poison any artifacts generated by the pipeline, steal sensitive secrets, and can also let them laterally move across a team's infrastructure. OWASP defines many such attack vectors in their Top 10 CI/CD Security Risks [3].

Security researchers have demonstrated many practical attacks on CI/CD platforms. They range from code injection to lateral movement to control of domain controllers [9]. Unfortunately, it's very easy to make mistakes while configuring pipelines and issues can exist in the configuration of the infrastructure responsible for running the pipelines as well.

1.1. GitHub Actions

GitHub, a popular source-code management platform introduced their CI/CD product called Actions. GitHub Actions is tightly integrated with the GitHub ecosystem. It lets users define configurations for their workflows and save them alongside their code as simple YAML files. Second, users can run their workflows on GitHub's infrastructure for free (with some limits [10]. At the same time, GitHub lets teams freely bring in their own infrastructure to run workflows or do a mix of both. Further, Actions lets developers create workflows and publish them to the public so that commonly used workflows can be re-used by other developer [8]

These workflows, defined in a special folder within every repository (.github/workflows), can be run against "triggers" or events that are defined in the confi-

guration file by the developer. For example, triggers may be a push, pull request, fork, etc. [11].

These features let most developers, experienced and unexperienced, easily get started with building their CI/CD pipelines. This ease-of-use is one of the reasons why GitHub Actions is very popular and is continuing to increase in popularity. Large organizations ranging from Google to Microsoft to open-source organizations like Pytorch to government agencies like the NSA, all use GitHub Actions [12].

The power and versatility exposes GitHub Actions to some of the same security problems with CI/CD pipelines discussed above. This is compounded by the fact that many of the workflows can be re-used and the open-source, collaborative nature of GitHub allows for complex dependency chains to exist. Further increasing the chance of security vulnerabilities is the feature that lets developers use their own infrastructure for running workflows [13] [14] [15].

1.2. Contributions

These security issues can have potentially disastrous consequences and it is imperative to enable developers to protect against common misconfigurations.

In this paper, we introduce "*Sher*", a tool designed to automatically detect and remediate some of the security issues present in GitHub Actions discussed earlier. *Sher* is a Python-based tool that hooks into the GitHub REST API to act based on repository events.

Our main contributions through *Sher* are as follows:

1) A broker that acts as a middleman between GitHub's APIs and a customizable, isolated and stateless environment to run Actions workflows.

2) An extensible rules engine to perform static analysis on workflows.

3) A tool to automatically remediate certain issues based on the results of the static analysis.

In the next section, we describe details of the GitHub Actions ecosystem—what workflows are, how they're defined and how they can be triggered. In doing so, we also explain the attack vectors covered by *Sher*. In Section 3, we cover some of the previous research and existing tools in this area. We also briefly explain how these works compare to mine. In Section 4, we describe how *Sher* works and in Section 5, its architecture and implementation. Later in Section 6, performance and effectiveness are compared against existing approaches. Lastly, we conclude with future extensions of *Sher* and how it can be generalized along with a summary of the paper.

2. DevSecOps Workflows

A CI/CD pipeline typically consists of a version control system, an automation server, a testing framework and deployment tools. In GitHub Actions parlance, the version control system used is Git, the automation server is called a "run-

ner", while testing framework and deployment tools are choices left to the developer based on their needs. Each "Action" consists of workflows, an automation consisting of a series of steps executed on a runner. These runners are VMs or Docker containers that can be run on GitHub's infrastructure or developers can choose to self-host it. In this section, we describe what a workflow is, how it is written, how it is triggered and the environment where it is run, and we also describe common ways of adversarial attack.

2.1. Workflows

A workflow is a configuration file written in YAML that defines what the automation task is. GitHub defines it as a "configurable automated process that will run one or more jobs" [11].

A repository's workflows are stored in a pre-defined directory within the root directory of the repository called .github/workflows. A repository can have multiple workflows each performing independent tasks or they can also be interdependent on each other.

The following are essential components of a workflow (See **Figure 1**):

- *Trigger:* An event that causes a workflow to run.
- *Job:* A series of *steps* that execute on a runner.
- *Step:* Each step in a job is a small task that builds up progress for the job. This can be a shell command, a third-party Actions workflow (simply called an "action"), docker containers or external programs.

Each workflow can include one or more of the above components and each of these components lets the developer define fine-grained conditions allowing them to have a lot of control how exactly the automation is executed.

Listing 1 shows an example workflow that runs when there is a push to a repository. The workflow checks out the repository, sets up node.js 20, installs a package and runs a program called bats. The workflow consists of multiple key-value pairs that each have a special meaning. The name key, as the name suggests, defines the name of the workflow which in this case is learn-github-actions. The on key defines the triggers. Triggers are events that GitHub generates and can be consumed by Actions. Next, we have the jobs key which defines one or more jobs that need to be run. In this case, the job's name is check-bats-version. The key for each job lists the steps to be taken (steps) and the kind of machine it needs to be run on (runs-on). In this case, the runner is a GitHub hosted Ubuntu VM. The steps for this workflow includes 4 parts with the first two re-using a third-party workflow to "check out" the repository and install node.js. The remaining steps involve running shell commands to install and run bat.

```
name: learn-github-actions
run-name: ${{ github.actor }} is learning
GitHub Actions
on: [push]
jobs:
  check-bats-version:
    runs-on: ubuntu-latest
```

```
steps:
  - uses: actions/checkout@v4
  - uses: actions/setup-node@v4
    with:
      node-version: '20'
  - run: npm install -g bats
  - run: bats -v
```

Listing 1. A simple GitHub Actions workflow.

2.2. Triggers

Workflow triggers are events that cause a workflow to run [16]. Events can be broadly categorized into events that occur in a repository; events that occur outside GitHub; scheduled events; manual events.

When an event that GitHub recognizes occurs, GitHub iterates through all the workflows present in the workflows directory in a repository and matches the event against the one defined using the on key. Some events also have types and developers can choose to trigger their workflows for one or more of these types.

2.3. Runners

Each job in a workflow needs to specify what kind of machine the job can be run on using the runs-on key. The machine that executes a job is called a runner. Runners are of two types—Github-hosted and self-hosted. GitHub uses a tag system to label the runners available to a repository and match a job to a runner. For example, a job that specifies a runs-on value of ubuntu-latest causes Actions to choose a GitHub-hosted runner using the latest available Ubuntu version. Similarly, if the value is self-hosted, Actions chooses a self-hosted runner [11].

GitHub-hosted runners are runners that GitHub provides to users for free with limits based on number of minutes of compute used. Each GitHub-hosted runner is a VM that is instantiated with an operating system (Windows, macOS, Linux), some pre-installed packages, and the runner software. These VMs are ephemeral and are destroyed after the job is complete. GitHub is also responsible for the maintenance of these runners.

On the other hand, self-hosted runners are machines that developers can set up by themselves to accept and execute jobs from GitHub, on the pre-condition that they are responsible for maintaining and upgrading the machines that run the jobs. The main advantage of self-hosted runners is that it offers developers control over the hardware, software, environment, and tools. For example, if a machine learning engineer wants to run workflows that require a GPU, they can create a machine that has a GPU and install the runner software that GitHub provides so that it can be registered to execute the GPU-bound workflows that the engineer requires.

However, self-hosted runners despite their ease-of-setup and flexibility is insecure in its default configuration. Runners, by default, have access to the same resources as the user of the machine they run on. If the runner software is running as root, all jobs executed on the machine run as root. Similarly, self-hosted runners, by default, are stateful. Any artifacts from previous jobs such as files

and orphan processes are left behind and continue to stay on the machine unless the developer takes special care to clean up. Lastly, as mentioned earlier, upgrading and maintaining the machine and the runner software is the responsibility of the developer.

2.4. Attack Vectors

GitHub Actions is unique in that it allows Actions workflows to be run by not only the repository owner but also any user who has previously contributed to the repository when they open a pull request Actions works and is by design. However, it also opens up the following attack vectors:

1) *Actions workflows from malicious PRs*—When a contributor creates a pull request, they can include a malicious workflow that may be run when the PR is submitted. This PR may potentially be able to steal tokens used by the repository and send them to the attacker.

2) *Using untrusted Actions workflows*—When a repository owner uses an untrusted workflow from the GitHub Marketplace, they are potentially allowing the workflow to run arbitrary code in the context of their repository. Once again, this may allow the workflow to steal tokens, secrets and modify the output of the workflow.

3) *Usage of non-ephemeral self-hosted runners*—When a developer sets up a runner on their own infrastructure to run Actions workflows, the code used by the workflow runs alongside other code on the machine with no isolation. Further, by default, unless the developer configures a method to clean up the runner, artifacts from a previous workflow are left behind. This can be exploited by a malicious actor to run arbitrary code on a repository owner's infrastructure. When combined with attack vector 1, this can potentially allow any contributor to run arbitrary code on a repository owner's infrastructure.

3. Related Work

The importance of security issues in CI/CD pipelines is new and increasing with the realization that software supply chain security is critical. Most research papers discussing these issue are recently published within the last two years. In this section, we discuss some of the existing literature, published work, and open-source tools present in this space. We also discuss some of the gaps present in their works that we have tried to address in our work.

NIST, in their paper [17] on "*Strategies for the Integration of Software Supply Chain Security in DevSecOps CI/CD Pipelines*" describe measures to include software supply chain security measures into a CI/CD pipeline. They describe measures such as secure build, secure pull-push operations with repositories, integrity of builds and audit logs, secure commits, and security of workflows. These measures show an accurate picture of the threats associated with pipelines. However, the paper is prescriptive and does not provide any reference implementation of how one might integrate these security measures.

Pan *et al.* [18], in their paper, "*Ambush from All Sides: Understanding Security Threats in Open-Source Software CI/CD Pipelines*", conduct a large-scale measurement study of over 320 k CI/CD pipelines to define a threat model and approaches attackers may take towards these pipelines. They go on to validate the threat model with case studies.

Koishybayev *et al.* [19], in their paper, "*Characterizing the Security of GitHub CI Workflows*", scan over 440 k workflows on GitHub Actions and found that over 99.7% of workflows re-use external third-party actions. Further, they found that 99.8% of workflows execute with more privileges than needed—the workflow can read and write data to the source code of the repository. This paper underscores the need for tools that analyze workflows and provide measures to developers to secure their configurations.

Dakic *et al.* [20], in their paper, "*CI/CD Toolset Security*", list CI/CD pipeline security concerns in Jenkins, another popular tool provider like GitHub Actions.

Further, outside of academia, there are many open-source tools and companies trying to address some of the shortcomings of GitHub Actions by hooking into the rich APIs that it provides.

In this paper, we studied 5 tools and solutions that currently exist—Gato [21], Ubicloud [22], StepSecurity, Actions Runner Controller [10], Runs On [10].

1) *Actions Runner Controller*—This is a Kubernetes controller maintained by GitHub that lets you deploy self-hosted runners as pods on a cluster. It gives the runners statelessness when jobs capable running entirely in containers are used. However, it cannot be easily used with custom hardware configurations.

2) *Gato*—Gato, or Github Attack Toolkit, is an open-source attack tool from Praetorian [21] that scans for CI/CD issues in a GitHub repository. It also lets researchers exploit pipeline vulnerabilities. However, it doesn't have the capacity to actually protect a pipeline by itself.

3) *StepSecurity*—StepSecurity is a startup that is focused on improving the security of GitHub actions. "Harden runner" is an action that lets you harden your self-hosted runners by restricting network traffic and applying other restrictions on the code that is run during a workflow. It also logs all network traffic during a workflow for later use. They also have products that lets users scan workflows. However, they don't address the use-case of isolating the environment for a self-hosted runner [23].

4) *Ubicloud*—This is a startup that provides a cheaper and more secure alternative to GitHub-hosted runners. They provide a service that lets developers run their GitHub Actions workflows on their infrastructure.

5) *Runs on*—Similar to Ubicloud, they let developers run GitHub Actions workflows on AWS with the option to use custom AWS VM images if needed.

Having looked at the various approaches taken by different tools and companies to secure pipeline workflows, it is clear that there is no "correct way" to improve the security. There are a list of tradeoffs based on the priorities of each tool and its intended use case.

Table 1. Comparison of Sher with some existing solutions.

Features	Sher	Gato	Ubicloud	Step Security	Actions Runner Controller	Runs On
Ephemeral Runners	Yes	No	Yes	No	Yes	Yes
Custom Runner Environments	Yes	No	No	No	Yes	Yes
Scan Workflows	Yes	Yes	No	Yes	No	No
Fix Workflows	Yes	No	No	Yes	No	No
Generate Exploits	No	Yes	No	No	No	No
Hardened Runners	No	No	Yes	Yes	No	NO

4. Methodology

In this paper, we propose an efficient solution to two major steps within GitHub Actions—first, the ability to easily create *customizable* self-hosted and ephemeral runners; second, the ability to scan Actions workflows for potential security issues, report them, and automatically take action on them. In doing so, we create a broker-based architecture between GitHub's APIs and a developer's infrastructure giving them more control over how to use their own hardware while preserving the ease-of-use of GitHub Actions Self-Hosted Ephemeral Runner

Sher, short for Self-Hosted Ephemeral Runner, is an open-source tool that acts as a broker between GitHub's APIs and the underlying infrastructure that runs a workflow's jobs. The primary objective is to provide users the ability to easily and securely customize the environment according to their hardware and software requirements while also preserving the flexibility in workflow definitions afforded by GitHub Actions. [24]

The tool works by registering itself as an authorized GitHub App. GitHub Apps are officially supported ways to extend the functionality of GitHub by using its REST APIs. GitHub also lets Apps subscribe to "events". These events are different from events that trigger workflows as described earlier. These are events that describe any changes happening in a repository. For example, an event is emitted if an account installs an app, a workflow is started, a workflow is completed, a new branch is made, an issue is created, and many such other events can be triggered. Each time an event is triggered, GitHub makes a POST HTTP call to an endpoint defined by the developers where they can get details of the event. This is called a "webhook" and lets developers add custom logic in response to an event [25].

Similarly, *Sher* is a GitHub app that subscribes to events about GitHub Actions workflows. In response to an event describing the start of a workflow, the tool creates a virtual machine just-in-time based on a pre-defined configuration.

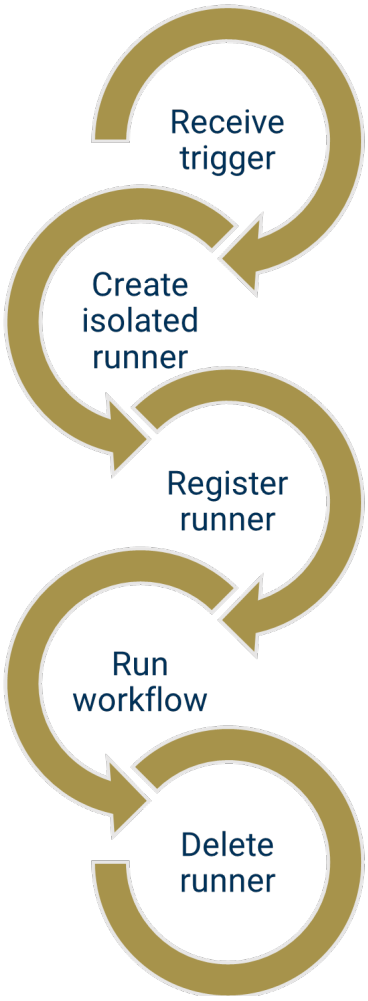

Figure 1. Self-Hosted ephemeral runner.

It then loads the runner software on the virtual machine and registers the runner with GitHub. Once the runner has executed its job, GitHub sends out another event signaling that the job has completed and the tool deletes the VM and its data. This process repeats every time a job needs to run on a self-hosted runner.

The advantages of doing it this way are evident. The first is that it lets workflow jobs run in an isolated environment. Further, since the VM is deleted after the completion of every run, there is no possibility for an attacker to leave behind artifacts on the infrastructure that can later be used by the attacker to compromise infrastructure. Next, since the developer can choose to configure the VM however they want (including the ability to passthrough desired PCIe hardware), they can easily run custom workflows securely that would have otherwise not been possible.

Scanning Tool

The second objective of *Sher* is to provide an open-source tool that can analyze a developer's workflows according to pre-defined rules and evaluate compliance

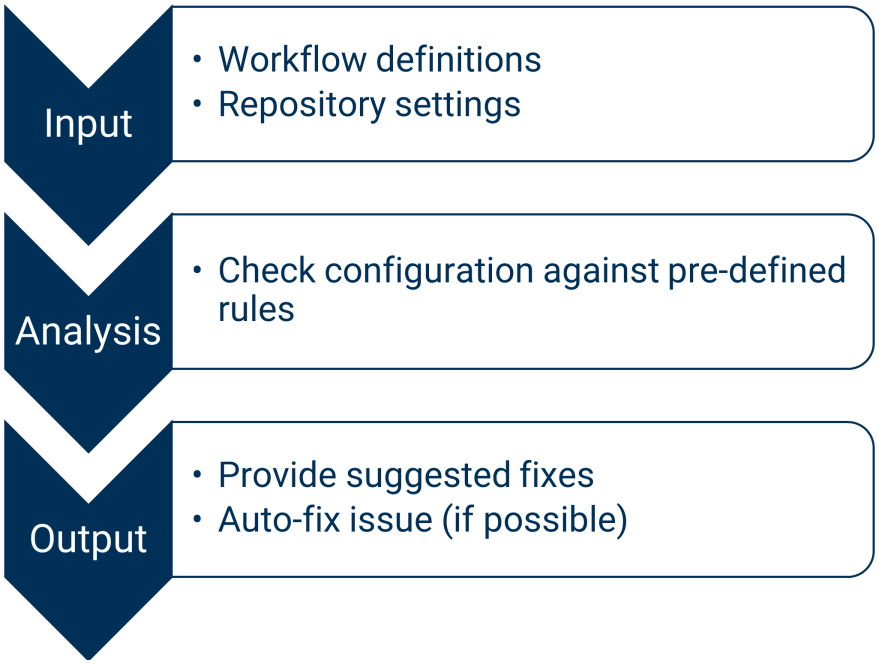

Figure 2. Scanning tool.

with these rules. When possible, the tool also suggest remediations to the developer.

The scanning tool (See **Figure 2**) makes is currently available as a website at https://ghasecurity.xyz. Anyone who has authorized the GitHub app for their account can use it.

When a developer installs the GitHub app, they are able to grant access to the app to scan their repositories as well. Once access is granted, the app can read the repository's workflows defined in the .github/workflows folder and provide a report on the website for each defined rule. In select cases, the app also lets the user automatically fix workflows according to best practices highlighted in the report.

The scanning tool supports three checks at present. These rules were carefully selected after considering a tradeoff between ease of implementation and impact. Since the tool is open-source, interested members of the community may add more checks to the tool as needed to extend it. The checks are defined in the form of a self-contained Python function.

The three rules currently present in the tool are as follows:

1) The first rule checks if a workflow re-uses third-party actions via its ref instead of its commit hash. GitHub recommends [8] developers do this because it is possible for a malicious actor to have a ref point to a different malicious commit. When this happens, it can lead to a supply chain attack when the ref pointing to a malicious commit is imported by another workflow.

2) The second rule checks if a workflow is using a secret in a step that is not necessary. Specifically, it checks if all the secrets accessible to a repository are being passed to another workflow or third-party action. While the uses of this are genuine, it can help to understand how repository secrets are being accessed

and used. When all repository secrets are directly passed by inheritance to another workflow or third-party action, it is generally not necessary and requires the developer to audit that configuration.

3) The third rule reads the repository settings and checks the permissions of the GITHUB_TOKEN token and if Actions are configured to approve pull requests. The GITHUB_TOKEN token controls the access of Actions workflows to the repository's contents. By default, this setting only lets workflows read the repository's code and not modify it. But, many developers change this default permission to allow workflows to both read *and* write even when it's not necessary. Similarly, the second setting that's checked by this rule warns a user if Actions workflows are allowed to merge pull requests to the repository without approval. This setting, if enabled, is dangerous as it can let any Actions workflow to create a pull request against the repository and automatically merge it without any human intervention.

The advantage of using an extensible tool like this is that it lets users immediately known if there are any glaring issues with their configuration. However, not every rule might be potentially applicable to a workflow, and the tool will let users disable rules not relevant to their workflows and prevent potential false-positives.

5. Architecture

In this section, we describe the architecture of both parts of *Sher* in depth. In doing so, we highlight how it acts as a broker between control-plane (GitHub) and data-plane (infrastructure) and how it can potentially be generalized to other CI/CD pipelines.

Sher is built as a simple FastAPI [26] server on Python to listen and respond to webhooks from GitHub. It's designed to run on any system that supports running Python and Vagrant. The source code for the tool is available at https://github.com/pranau97/sher.

5.1. Self-Hosted Ephemeral Runner Broker

From start to finish, there are around 10 steps (See **Figure 3**) every workflow needs to take to complete a job. These steps are as follows:

- The first step starts with a developer making a change to the repository. This can typically be a push to the repository.
- When GitHub sees this push, it reads the workflows in the repository to see if there are any that are triggered by a push.
- If yes, GitHub starts running those workflows. Further, it sends out a webhook to *Sher* that a workflow has started.
- *Sher*'s FastAPI server (See **Figure 4**) listens on https://ghasecurity.xyz/webhook for incoming requests. When it receives a webhook, the first thing that it does is verify that it is legitimate and came from GitHub by checking a signature.

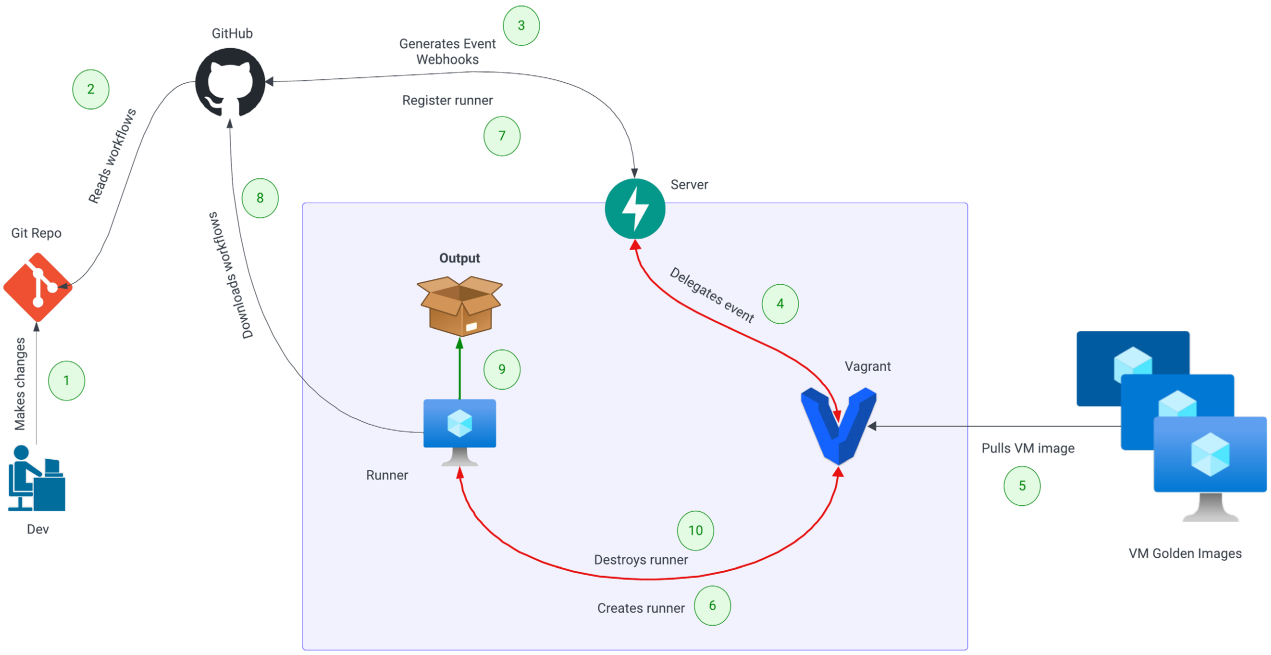

Figure 3. Architecture of runner broker.

```
❯ uvicorn auto-runner:app --reload

INFO:     Will watch for changes in these directories: ['/home/pranau/apps/auto-runner']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [192636] using StatReload
INFO:     Started server process [192685]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     140.82.115.169:0 - "POST /webhook HTTP/1.1" 200 OK
INFO:     140.82.115.173:0 - "POST /webhook HTTP/1.1" 200 OK
INFO:     140.82.115.242:0 - "POST /webhook HTTP/1.1" 200 OK
INFO:     auto-runner label detected.
DEBUG:    JWT generated, eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpYXQiOjE3MTAwMjgyNjQsImV4cCI6MTcxMDAyODg2NCwiaXNzIjoiODIxMzk3In0.TqYshZvH7QymJhottrcvHI_5G8JQBfn2UEGMyyEkQ_XZgTmhgsch8RjrxN-DvwgDJveY2qKbZe7iMKRxlZDZSVYVf_4RQzHMWQbdyZXd3DF88GUd884TXX61I-hlsrChY4K_qAxNu4mUkrk3_N0Bt6byAEyAiIJhK8qixgLkruZbEMuulXuiGhhCxYDs1xMpk9ixXt_E-PhtEw4idcCxqGUnywjCcvGuHXrjRENapMA2FdOX961JwAS5xq7Et5P-gIChmeQecrQitEaYdCfCjxqDiUBWHZg03RMNy9-jLqFWZLtcx9V_guRpq8xqnWXBnNe_jstVRAdqtdbvUM7thQ
DEBUG:    Token generated, ghs_9VwxBbTDaqhcI7OLPue3zEzlnMI7BO1Z9eVy
DEBUG:    Registration token generated, BF72BAGZSLNXTJJ6KOXPBBDF5UB7RAVPNFXHG5DBNRWGC5DJN5XF62LEZYBM36F3WFUW443UMFWGYYLUNFXW4X3UPFYGLN2JNZ2GKZ3SMF2GS33OJFXHG5DBNRWGC5DJN5XA
INFO:     Initialized runner directory work/22474262677
INFO:     VM provisioned for 22474262677 in 35.378612995147705 seconds
INFO:     Runner started for 22474262677 in 14.5056791305542 seconds
INFO:     140.82.115.241:0 - "POST /webhook HTTP/1.1" 200 OK
INFO:     140.82.115.27:0 - "POST /webhook HTTP/1.1" 200 OK
INFO:     auto-runner label detected.
INFO:     140.82.115.35:0 - "POST /webhook HTTP/1.1" 200 OK
INFO:     VM destroyed for 22474262677
INFO:     Folder 'work/22474262677' and its contents deleted successfully
```

Figure 4. Output of the FastAPI server showing received webhooks from GitHub and the corresponding processing.

- If the request is legitimate, the server starts processing the request. It checks the request payload to see if the workflow that has been started uses a runner with the tag, "auto-runner". If not, the request is just ignored.
- Next, it sets up an job specific folder where it creates a Vagrantfile that defines how the VM needs to be set up. Listing 2 shows the default Vagrantfile.
- The server then contacts GitHub to retrieve a registration token for a runner

and boots the VM up and registers the runner as *ephemeral* with GitHub.

- At this stage, the workflow execution proceeds as normal as GitHub recognizes the new runner and assigns the job to it.
- Once the job has completed, GitHub fires off another webhook to *Sher* stating that the workflow has completed.
- The server recognizes this event and in response terminates the VM and deletes the job specific folder.
- Since the runner was registered as *ephemeral*, GitHub also removes the runner from the list of configured runners so that it's not re-used.

Thus, *Sher* acts as a broker interpreting events from GitHub and translating it into commands for Vagrant. This way, the control-plane (*i.e.*, GitHub Actions) remains the same). However, the data-plane (*i.e.*, infrastructure) can be customized to the developer's liking [27].

```
Vagrant.configure("2") do |config|
  config.vm.box = "generic/ubuntu2204"
  config.vm.provider "libvirt"
  config.vm.synced_folder ".",
"/vagrant", disabled: true
  config.vm.hostname = "runner"

  config.vm.provision "shell", inline:
<<-SHELL
    apt-get update
    apt-get install -y jq
    wget $(curl -s
https://api.github.com/repos/actions/r
unner/releases/latest | \
    jq -r '.assets[] | select(.name |
contains ("linux-x64"))
| .browser_download_url')
    mkdir actions-runner
    tar xzf ./actions-runner-linux-x64*
-C actions-runner
    chown -R vagrant: vagrant ac-
tions-runner
  SHELL
end
```

Listing 2. Default vagrantfile.

5.2. Scanning Tool

From start to finish, there are six steps involved in the process of scanning a workflow. These steps are as follows:

- The process starts with a user who has installed the app going to the website, https://ghasecurity.xyz.
- Once on the website, they are prompted to add their repository's full name in the format owner/name.
- The FastAPI server parses the repository name and looks it up to check if it has the requisite permissions to access the repository's contents. If yes, it queries GitHub's APIs to list all workflows in the default branch of the repo-

sitory.

- The user is again prompted to select a workflow as shown in Figure 5 and Figure 6.

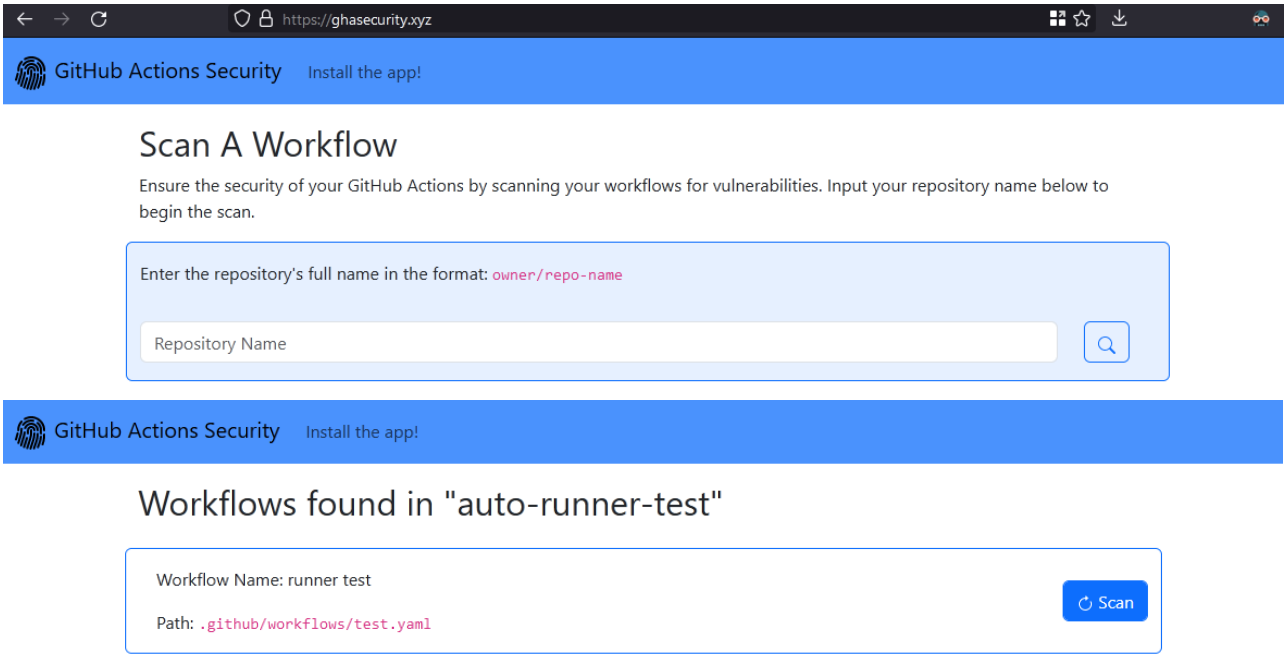

Figure 5. Screenshots showing the Scanning tool's search and workflow selection interface.

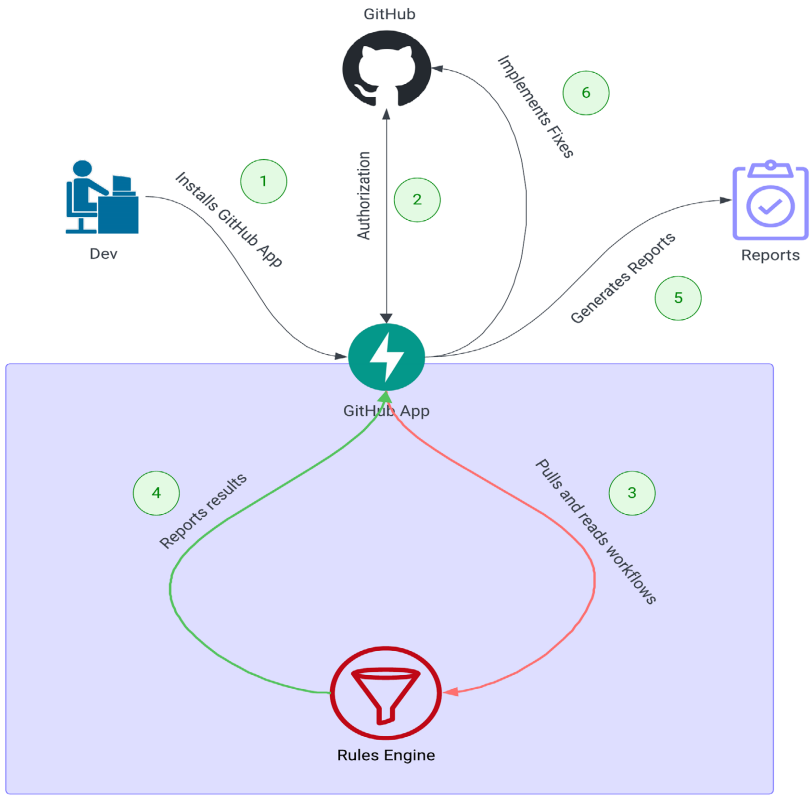

Figure 6. Architecture of the scanning tool.

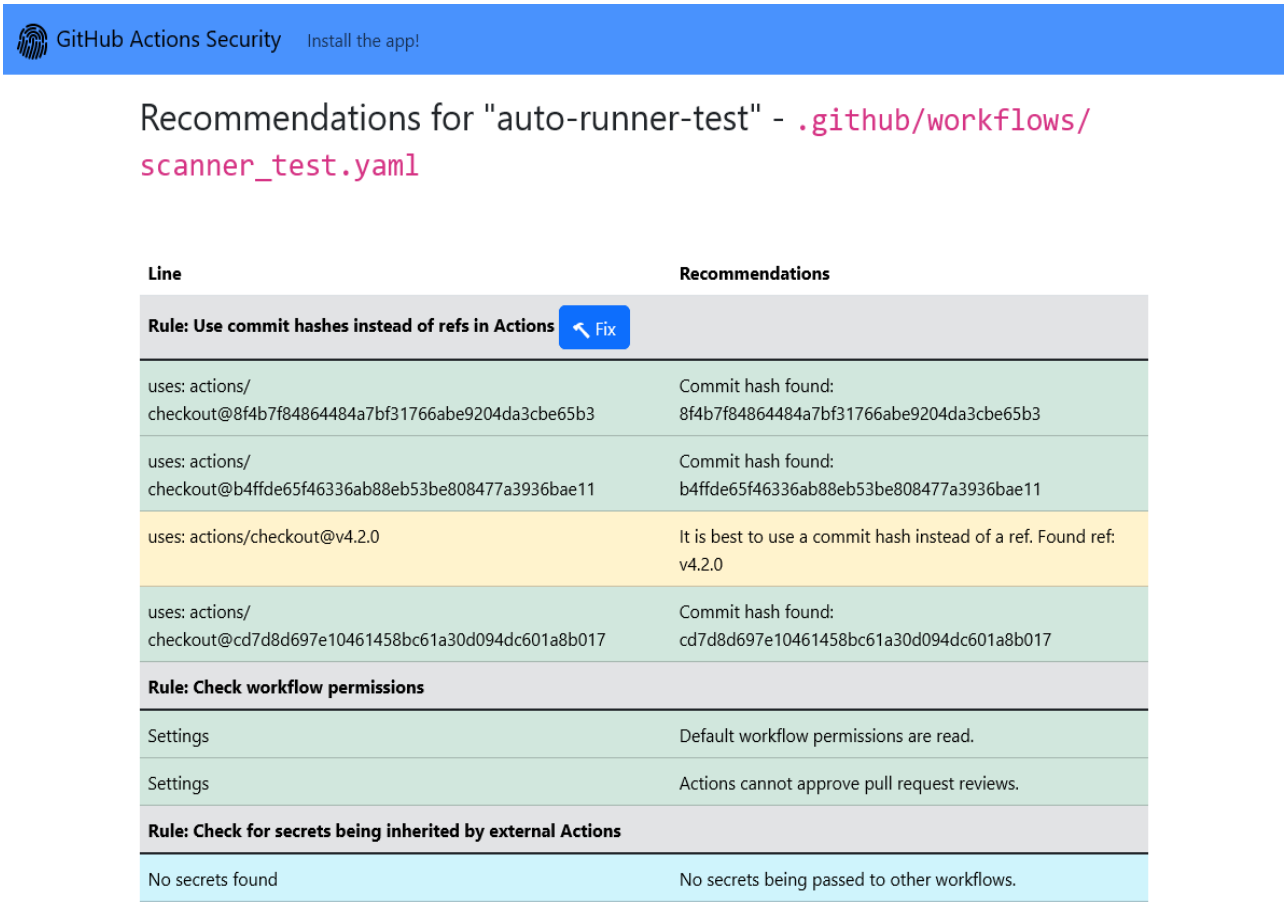

Figure 7. List of recommendations of the scanning tool.

- Once the user selects a workflow, the server reads the workflow's contents and runs each rule's functions against the workflow and tabulates a list of recommendations.
- These recommendations are then tabulated and sent to the website to display. The recommendations are also color coded according to the potential security risk posed by the configuration. **Figure 7** shows how the list of recommendations looks.
- For each category of recommendations, there is a button to apply a fix. If the user selects the fix button, the tool then automatically applies a fix.
- If a fix is not available, the button is simply not displayed.

This way, *Sher* can be used as a tool to educate its users about potential issues along with an explanation of why their configuration is potentially insecure. Where possible, *Sher* also makes it very easy to automatically apply a fix.

6. Evaluation

We devised multiple test cases that are designed to emulate real-world usage of *Sher*. We have already discussed in the preceding sections how *Sher* addresses gaps in current solutions. The following test cases will prove how the gaps were met by Sher.

6.1. Test Cases for Runner Tool

Test cases for the runner tool were divided into two categories. One category was used to test ephemeral runners while the other was used to test the same but with custom VM images. The test cases are as follows:

1) *Test Case* 1: Deploying and destroying VM images on-demand in response to webhooks from GitHub.

2) *Test Case* 2: Running successfully when fed a single-step, single-job workflow.

3) *Test Case* 3: Running successfully when fed a multiple-step, single-job workflow.

4) *Test Case* 4: Running successfully when fed a multiple-step, multiple-job workflow.

The same test cases were repeated to test the functionality of the tool with custom VM images. In all test cases, *the tool was able to cleanly create a VM image*, *run the job*, *and clean up after the job was marked complete*.

6.2. Test Cases for Scanning Tool

For the scanning tool, we designed the following test cases:

1) *Test Case* 1: Scan workflows and settings. The backend should be able to scan workflows for the defined rules and correctly generate recommendations.

2) *Test Case* 2: Parse repository names. The website should be able to correctly parse the repository name. If the repository is incorrect, it should be able to display an error message as appropriate.

3) *Test Case* 3: List workflows in the default branch of the repository. The website should be able to read the workflows present in .github/workflows and list them correctly.

4) *Test Case* 4: Display table of recommendations for the three currently configured rules.

5) *Test Case* 5: Automatically fix issues according to the generated recommendations.

The tool was able to successfully pass the first 3 test cases. However, for test case 4 and 5, not all rules had an automatic fix and there were some false positives detected too. These results for each test case are captured in Table 2.

These results are in-line with expectations. Trying to fix all false positives will result in the rules becoming too complex and hard to maintain. Similarly, not all rules are able to be fixed automatically. For example, a human is needed to audit

Table 2. Results of scanning rules.

RULE	SCAN %	FIX %
R1: Usage of git refs instead of commits	80%	100%
R2: Secrets usage	75%	N/A
R3: Non-Default workflow permissions	100%	100%

which secrets can be safely inherited by third-party actions and which rules cannot be.

7. Summary

This paper has introduced *Sher*, a novel tool designed to enhance the security infrastructure of GitHub Actions by automating the detection and remediation of potential security issues in CI/CD workflows. Through a code development process carried out by the first author as part of his research, Sher provides a dual solution: it acts as a secure, ephemeral runner to isolate and execute jobs, and it incorporates a static analysis tool to identify and fix vulnerabilities in workflow configurations. The combination of these functionalities addresses a significant gap in current CI/CD security practices, which often overlook the necessity for dynamic and proactive security measures.

Sher's architecture allows it to function as a middleman between GitHub's APIs and the user's custom infrastructure, facilitating a controlled and secure environment for running potentially vulnerable CI/CD tasks. The implementation of *Sher* as a GitHub App, leveraging GitHub's extensive webhook capabilities, ensures that it integrates smoothly with existing GitHub workflows, maintaining user familiarity and ease of use while significantly elevating the security posture of the development pipeline.

7.1. Future Work

Looking forward, the *Sher* tool presents several avenues for extension and enhancement that could further solidify its utility and applicability in a broader range of CI/CD security scenarios. One potential extension could involve expanding the rule set of the static analysis tool to cover more comprehensive security checks. As the landscape of CI/CD security evolves (See [28] [29]), continuously updating and refining the rules based on the latest security research and threat intelligence will enhance *Sher*'s effectiveness at identifying and mitigating emerging threats.

Another promising direction could be to extend *Sher*'s functionality to other CI/CD platforms beyond GitHub Actions. By adapting the underlying principles and mechanisms of Sher to platforms like GitLab CI, Jenkins, or CircleCI, the tool could offer a versatile security solution that benefits a wider array of users and use cases in the DevOps community. This cross-platform capability would involve developing adapters or plugins that translate Sher's security checks and runner management features to the APIs and workflow specifications of these other systems.

Moreover, integrating machine learning techniques to predict potential vulnerabilities based on historical data and common configuration errors could significantly advance Sher's preemptive security measures. This proactive approach would allow developers to avoid common pitfalls before they manifest as security risks, thereby strengthening the CI/CD pipeline's defense against at-

tacks.

7.2. Conclusion

In conclusion, *Sher* represents an advancement in the security of GitHub Actions by providing a solution to a critical aspect of modern software development: secure automation in CI/CD environments. As this tool evolves, it has the potential to set new standards for security in DevSecOps practices, ensuring that security is a cornerstone of the development process, not an afterthought. With continued development and expansion, *Sher* could play a crucial role in shaping the future of secure software deployment and management in an increasingly complex cyber threat landscape.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Oracle (2024) What is DevOps? https://www.oracle.com/devops/what-is-devops/

[2] Wikipedia (2024) GitHub. https://en.wikipedia.org/w/index.php?title=GitHub&oldid=1196345392

[3] GitHub Resources (2024) DevSecOps Explained. https://resources.github.com/devops/fundamentals/devsecops/

[4] Jenkins (2024) https://www.jenkins.io/

[5] Travis CI (2024) Test and Deploy with Confidence. https://www.travis-ci.com/

[6] Circle CI (2024) Build Anything Fast. The CI/CD Platform for the AI Future. https://circleci.com/

[7] GitLab (2024) GitLab CI-Documentation. https://docs.gitlab.com/ee/ci/

[8] GitHub (2024) What is Github Actions? https://web.archive.org/web/20211203130324/https://resources.github.com/downloads/What-is-GitHub.Actions_.Benefits-and-examples.pdf

[9] Stawinski IV, J. (2024) Fixing Typos and Breaching Microsoft's Perimeter. https://johnstawinski.com/2024/04/15/fixing-typos-and-breaching-microsofts-perimeter/

[10] GitHub (2024) GitHub Marketplace-GitHub Actions. https://github.com/marketplace?category=&query=updated%3A%3E2023-07-21+sort%3Apopularity-desc&type=actions&verification

[11] GitHub (2024) Understanding GitHub Actions. https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions

[12] Epling, J. (2024) Powering Community-Led Innovation with GitHub Actions. https://github.blog/2019-11-14-powering-community-led-innovation-with-github-actions/

[13] Smart, I. and Gazdag, V. (2024) RCE-as-a-Service: Lessons Learned from 5 Years of Real-World CI/CD Pipeline Compromise.

https://www.blackhat.com/us-22/briefings/schedule/#rce-as-a-service-lessons-learned-from-5-years-of-real-world-cicd-pipeline-compromise-27541

[14] Stawinski IV, J. (2024) Playing with Fire—How We Executed a Critical Supply Chain Attack on Pytorch. https://johnstawinski.com/2024/01/11/playing-with-fire-how-we-executed-a-critical-supply-chain-attack-on-pytorch/

[15] Khan, A. (2024) One Supply Chain Attack to Rule Them All—Poisoning GitHub's Runner Images. https://adnanthekhan.com/2023/12/20/one-supply-chain-attack-to-rule-them-all/

[16] GitHub (2024) Events that Trigger Workflows. https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows

[17] Chandramouli, R., Kautz, F. and Torres-Arias, S. (2024) Strategies for the Integration of Software Supply Chain Security in DevSecOps CI/CD Pipelines. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-204D.pdf

[18] Pan, Z., *et al*. (2023) Ambush from All Sides: Understanding Security Threats in Open-Source Software CI/CD Pipelines. *IEEE Transactions on Dependable and Secure Computing*, **21**, 403-418. https://ieeexplore.ieee.org/abstract/document/10061526

[19] Koishybayev, I., Nahapetyan, A., Zachariah, R., Muralee, S., Reaves B., Kapravelos A. and Machiry, A. (2022) Characterizing the Security of Github CI Workflows. *Proceedings of the* 31*st USENIX Symposium*, Boston, 10-12 August 2022, 2747-2763.

[20] Dakic, V., Redzepagic, J. and Basic, M. (2024) CI/CD Toolset Security. *Proceedings of the* 33*rd DAAAM International Symposium on Intelligent Manufacturing and Automation*, Vienna, 27-28 October 2022, 161-164. https://www.daaam.info/Downloads/Pdfs/proceedings/proceedings_2022/working_papers/dpn34029_a_2_Dakic.pdf

[21] Praetorian Inc. (2024) Gato (GitHub Attack Toolkit). https://github.com/praetorian-inc/gato

[22] Ubicloud (2024) Open, Free, and Portable Cloud. Elastic Compute, Block Storage (Non Replicated), Virtual Networking, Managed Postgres, and IAM Services in Public Beta. https://github.com/ubicloud/ubicloud

[23] StepSecurity (2024) Secure Your GitHub Actions with StepSecurity Platform. https://www.stepsecurity.io/

[24] Kumar, P. (2024) Sher. https://github.com/pranau97/sher?tab=readme-ov-file#tiger-sher

[25] GitHub (2024) About GitHub Marketplace for Apps. https://docs.github.com/en/apps/github-marketplace/github-marketplace-overview/about-github-marketplace-for-apps

[26] Ramírez, S. (2024) FastAPI. https://fastapi.tiangolo.com/

[27] HashiCorp (2024) Vagrant by HashiCorp. https://www.vagrantup.com/

[28] Github (2024) About Billing for GitHub Actions. https://docs.github.com/en/billing/managing-billing-for-github-actions/about-billing-for-github-actions

[29] Sharma, A. (2024) GitHub Actions Being Actively Abused to Mine Cryptocurrency on GitHub Servers. https://www.bleepingcomputer.com/news/security/github-actions-being-actively-abused-to-mine-cryptocurrency-on-github-servers/