Toward Understanding the Security of Plugins in Continuous Integration Services

Xiaofan Li*
The University of Delaware
Newark, DE, USA
xiaofan@udel.edu

Yacong Gu*
Tsinghua University
QI-ANXIN Group
Beijing, China
guyacong@tsinghua.edu.cn

Chu Qiao
The University of Delaware
Newark, DE, USA
qiaochu@udel.edu

Zhenkai Zhang
Clemson University
Clemson, SC, USA
zhenkai@clemson.edu

Daiping Liu
Palo Alto Networks
Santa Clara, CA, USA
dpliu@paloaltonetworks.com

Lingyun Ying
QI-ANXIN Technology
Research Institute
Beijing, China
yinglingyun@qianxin.com

Haixin Duan
Tsinghua University
Zhongguancun Laboratory
Beijing, China
duanhx@tsinghua.edu.cn

Xing Gao
The University of Delaware
Newark, DE, USA
xgao@udel.edu

ABSTRACT

Mainstream Continuous Integration (CI) platforms have provided the plugin functionality to accelerate the development of CI pipelines. Unfortunately, CI plugins, which are essentially reusable code snippets, also expose new attack surfaces as plugins might be developed by less trusted users. In this paper, we present an in-depth study to understand potential security risks in existing CI plugins. We conduct a comprehensive analysis of plugin implementations on four mainstream CI platforms (GitHub Actions, GitLab CI, CircleCI, and Azure Pipelines), and investigate several weak links in existing plugin distributions and isolation mechanisms. We investigate seven attack vectors that can enable attackers to hijack plugins and distribute malicious code without plugins users being aware, and further exploit hijacked plugins to manipulate the workflow execution. Additionally, we find that plugin dependency (a plugin references other plugins) might further amplify the attack impact of our disclosed attacks. To evaluate the potential impact, we conduct a large-scale measurement on GitHub and GitLab, covering a total of 1,328,912 repositories using the aforementioned CI platforms. Our measurement results show that a large number of repositories and existing plugins, including many widely used ones, are potentially vulnerable to the proposed attacks. We have duly reported the identified vulnerabilities and received positive responses.

CCS CONCEPTS

• **Security and privacy**;

KEYWORDS

Software Supply Chain, Continuous Integration, Security

*Both authors contributed equally to this research.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670366

ACM Reference Format:
Xiaofan Li, Yacong Gu, Chu Qiao, Zhenkai Zhang, Daiping Liu, Lingyun Ying, Haixin Duan, and Xing Gao. 2024. Toward Understanding the Security of Plugins in Continuous Integration Services. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670366

1 INTRODUCTION

Continuous Integration (CI) workflows have been widely used for automated code build, integration, and testing in the existing software industry. Many organizations have adopted CI workflows for packaging and releasing software updates. For example, the curl repository on GitHub [13], pytorch repository on GitHub [72], and GitLab itself on GitLab [47] have adopted CI for code integration. It has been reported that the CI tools market is estimated at USD 1.43 billion in 2024 and will reach USD 3.72 billion by 2029 [1]. To accelerate the development of custom CI pipelines and enable seamless integration with external systems, many CI Platforms (CIPs) allow developers to create reusable code snippets (i.e., plugins) that can be shared within an organization or with the public. CI plugins have been a core component in existing CI ecosystems and are widely used by many repositories. For example, in GitHub Actions, almost all CI pipelines have adopted plugins (i.e., actions). GitHub Actions has operated an official plugin marketplace [36] maintaining public open-source plugins and providing official actions for many basic functionalities such as code cloning. Other mainstream CIPs including GitLab CI [46], CircleCI [11], and Azure Pipelines [67], have also provided the plugin functionality.

Unfortunately, the wide adoption of CI plugins also exposes new attack surfaces, potentially enabling attackers to inject malicious code during the build process. Researchers have discovered several GitHub Actions plugins that are vulnerable to code injection vulnerabilities [64]. In 2023, thousands of public GitHub repositories were found to be vulnerable to malicious code injection via self-hosted GitHub Actions runners, allowing attackers to compromise PyTorch and Microsoft Deepspeed releases [2]. Moreover, sensitive data (e.g., *secrets*) is often used and passed to CI plugins. If a plugin improperly uses secrets, it can lead to a serious risk of sensitive data leakage. However, despite that several recent works have studied CI

security [48, 64], we still lack a systematic study on understanding the security of CI plugins.

In this paper, we present an in-depth study to reveal potential security vulnerabilities that could be exploited by CI plugins to abuse existing CI services. We first conduct a comprehensive analysis of existing plugin implementations on four mainstream CIPs (GitHub Actions, GitLab CI, CircleCI, and Azure Pipelines), from the plugin distribution, parsing, and version control, to resource isolation and sharing. We investigate several weak links in the CI plugin ecosystem that could potentially cause security risks, enabling attackers to perform various software supply chain attacks such as code injection and data leakage. For example, we find that the distribution channel of CI plugins might be unsafe, and existing CIPs generally lack sufficient resource isolation for plugins.

We investigate two types of threats, including seven potential attack vectors, that can enable attackers to (1) hijack plugins without plugin users being aware (i.e., *unsafe plugin distribution*), and (2) further abuse hijacked plugins to potentially manipulate the execution of other plugins, even across different jobs and affecting the entire workflow (i.e., *improper plugin isolation*). For example, CI users largely utilize plugins with unsafe version control methods, allowing plugin maintainers to modify the code of a specific version (e.g., inject malicious code/vulnerability) without changing the version number. For *improper plugin isolation*, we investigate security threats posed by plugins both within a job (inner) and across different jobs (inter). Some CIPs parse plugins using the CI runner, while they only adopt weak isolation (e.g., process-based resource isolation) in the runner. Thus, one plugin process can easily access/manipulate files and environment variables belonging to other plugins inside one job, and/or steal various types of sensitive data (e.g., *secrets*). Particularly, we find attacking methods allowing a malicious plugin to break the plugin order to hijack any plugin within a job. Moreover, although CIPs mostly adopt job-level isolation as a base security policy, vulnerabilities that can be exploited to launch inter-job attacks still exist. A preceding job with malicious plugins might hijack the control flow or inject arbitrary commands into (of) subsequent jobs (e.g., input injection attack). Finally, we find that CI plugins can utilize external resources (e.g., reference other plugins), which form plugin dependency chains. Unfortunately, a plugin and its references will be downloaded and executed in the same step without strong isolation. Thus, plugin dependencies can largely amplify the attack impact of our disclosed attacks.

To evaluate the potential impact of our disclosed attacks, we conduct a large-scale measurement on GitHub and GitLab, covering a total of 1,328,912 repositories using four CIPs. We further analyze the collected repositories and adopt different strategies for collecting plugins in different CIPs, with a total of 43,169 plugins collected. Then, we utilize pre-defined rules to detect potential plugin vulnerabilities. Our measurement results show that a large number of repositories and existing plugins are vulnerable to the proposed attacks. For example, 369 GitHub Actions and 52 GitLab CI standard plugins can be immediately hijacked by any attackers, affecting 4,918 repositories. Also, most of the repositories do not use plugins and version controls in a safe way. In addition, we find both inner-job and inter-job threats are common. 1,097 potentially problematic plugins in GitHub Actions use *secrets* improperly, potentially causing secret leakage. Also, many repositories, including several widely used repositories, are potentially vulnerable to the inter-job input injection attack. Finally, we have discussed practical mitigation and timely disclosed our findings to impacted stakeholders and received positive feedback.

In summary, the major contributions of this work include:

- We present the first systematic study on the security threats in CI plugins. We present an in-depth analysis of existing plugin implementation and identify several risks caused by unsafe plugin distribution and weak resource isolation.
- We show that attackers can exploit multiple attack vectors to potentially distribute malicious code to users without user awareness, break the CI isolation to affect the CI workflow, and steal high-value sensitive data.
- We conduct a large-scale measurement on open-source repositories, revealing that our proposed attacks may present significant security risks to CI users.
- We have disclosed our findings to the impacted stakeholders and received positive feedback.

2 BACKGROUND

2.1 CI Introduction

CI is a software development practice where developers can integrate code changes into a central repository with automated build and test. The pipeline contains a series of steps with different tasks (e.g., code testing, quality assurance, and product releasing) to streamline the software delivery process. For example, a developer can create a new branch of code and commit it after some code modifications. The CI services will automatically build the code, run the automated test suite, and then deploy a new version of the application to a staging server, enabling other teams to test the changes in a production-like environment.

A typical CI workflow involves multiple stakeholders. First, code hosting platforms (CHPs) such as GitHub and GitLab are widely used to manage source code in repositories. CI platforms (CIPs) execute CI tasks of a repository on runners hosts. Popular CHPs often provide their own CIPs (e.g., GitHub Actions and GitLab CI), or they can also integrate independent CIPs (e.g., CircleCI). Third-party services (TPSs) like cloud services might also be involved in storing intermediate files generated during CI tasks.

The owner of a repository can set multiple configurations of CI tasks using a configuration file, including execution triggers events (e.g., a new push event or pull request), runner configuration (e.g., the running environment), secrets, and artifacts. Specifically, *secrets* can be used to access TPSs during the CI task execution. It is recommended to use secrets as key-value pairs so that sensitive data (e.g., plaintext passwords) will not be exposed in the CI task configuration [4, 30]. In addition, *artifacts* are files generated during the execution of a CI task. Examples include package release files, intermediate files, and test-related files (e.g., test reports).

CI workflow. When executing CI tasks, a CI controller will authorize with CHPs, parse the CI task configuration, and further schedule and distribute CI tasks to runners. The runner finally dispatches executors (e.g., shell) to execute CI tasks. Particularly, runners can run on servers provided by CIPs (i.e., *CIP-hosted runners*) or on an organization's own machines (i.e., *self-hosted runners*), which enables the organization to customize the environment and

```yaml
# GitHub Actions Standard Plugin Examples
steps:
  - name: Upload Artifacts
    uses: actions/upload-artifact@694cdab
    with:
      name: homework
      path: artifact/math-homework.txt
# GitHub File Reference Plugin Example
call-workflow-passing-data:
  uses: my-org/my-repo/.github/workflows/demo.yml@main
    with:
      who-to-greet: 'GitHub'
    secrets:
      envPAT: ${{ secrets.envPAT }}
# CircleCI Standard Plugin Example
orbs:
  node: circleci/node@5.1.0
jobs:
  steps:
    - node/install
```

Listing 1: CI plugin examples in CI platforms.

save costs [31]. A CI task consists of a set of *jobs*, which is the smallest unit of CI permissions [32, 45]. A job further has a sequence of *steps*, which share the same permissions. While jobs are typically isolated, they can be dependent on data transfer: a job can take the output of another job as input. Files generated during the CI execution such as dependent packages can be stored in CI caches so that these files can be reused by subsequent tasks.

2.2 CI Plugin Type and Lifecycle

CI plugins essentially are reusable code snippets that can be shared within an organization or with the public. There are two types of plugins based on their development and deployment approaches.
Standard plugin follows the CIP officially defined standard for development. Line 1 in Listing 1 shows an example. Standard plugins contain (1) various attributes (e.g., *name* and *version*), (2) a set of inputs and (optional) outputs, and (3) the execution environment. CIP usually provides an index function for plugins, enabling developers to obtain a specific plugin based on its name and version. The *inputs* specify the data that can be used by the plugin during runtime, while the *outputs* declare the generated output data, which can be further used by subsequent steps in a CI task. Additionally, some CIPs (e.g., GitHub Action and Azure Pipeline) maintain a *metadata* file for each standard plugin defining all the above information, as well as entry points of code files (or commands) that run at different stages in a plugin's lifecycle. Examples include `action.yml` or `action.yaml` in GitHub Actions, and `task.json` in Azure Pipeline.

Standard plugins support three runtime environments: (1) JavaScript engine: GitHub Actions and Azure Pipeline support JavaScript-based standard plugins. (2) Docker container, supported by GitHub Actions, which can customize environment configurations in the `Dockerfile` so that developers can run actions written in any language. (3) Shell commands: the main logic of plugins is implemented using shell commands, used by *composite action* in GitHub Actions, CircleCI Orbs, and GitLab CI *Components*. Particularly, GitHub composite actions can bundle a sequence of actions into one, which means that a composite action can reference other actions.

File Reference Plugin. In addition to standard plugins, GitHub Actions [37], GitLab CI [46], and Azure Pipelines [68] support developers to import a CI configuration file from another repository (and reuse its code) via their syntaxes. Line 8 in Listing 1 shows an example of GitHub Actions using a file reference plugin: the referenced CI task file will be downloaded using the plugin's location information in its hosting repository (line 10). We refer to this type as the file reference plugin, which essentially is a CI configuration file (i.e., YAML file) stored in another repository. Unlike standard plugins, there is no plugin marketplace nor versions for file reference plugins. However, users can specify the git tag or commit hash value in the syntax of a file reference plugin to set restrictions. Particularly, GitHub Actions and GitLab CI can directly reference a file reference plugin, while Azure Pipelines requires establishing a service connection before using it in other repositories.
General CI Plugin Lifecycle. Typically, plugin maintainers first register an account on either CHP or CIP's plugin marketplace, and then can publish, update, and unpublish CI plugins with specific version numbers. Depending on policies, the plugin might be stored in CHPs or directly managed by the CIP (details in Section 3.1). CI plugins can be then introduced into the CI workflow, with input/output set up by developers. For example, GitHub Actions utilizes "`uses:`" for importing plugins and "`with:`" for configuring inputs (lines 4-7 in Listing 1). Developers can further specify the plugin version (as line 17 in Listing 1). Section 3.3 presents a detailed discussion on plugin version control. When executing CI workflows, CIP can execute plugin code after acquiring plugins and initializing inputs. It first reads the plugin's metadata data to extract code entry points and execution environments. Based on the plugin type, the runner uses the corresponding engine (e.g., JavaScript, shell, or Docker) to execute the action. After the execution, the plugin might generate output for subsequent steps in the CI task.

2.3 Threats in Software Development

Software registries have been a major target of various software supply chain attacks [7, 58], and thus have attracted extensive research efforts [49, 50, 60, 76, 84, 86]. Gu et al. [49] investigated 12 threats in package registries. One threat is the package redirection hijacking attack, which enables attackers to hijack packages in decentralized registries without compromising accounts. Particularly, decentralized software registries utilize CHPs to manage packages. When CHP users change their username [34] or transfer their projects to another account [40], CHPs will automatically create a link between the old and new repository locations, so that both locations can be used to download the package. However, the link is terminated immediately if the old location is re-registered by attackers, and future users using the old location will be hijacked. Another threat is the package reference attack. Attackers can reference a malicious package in a normal package. When users download the normal package, the malicious package will also be silently obtained by the users without awareness. Finally, package version reuse is known to be dangerous, as it potentially allows attackers to inject malicious code stealthily (as the version number remains unchanged even when the code has changed). Thus, package registries normally prohibit version reuse, and provide the *version pinning* functionality so that developers can specify versions.

Table 1: Overview of CI plugin implementations. *S*/*R* **denote the plugin type:** *S* **- Standard Plugin,** *R* **- File Reference Plugin.** *D*/*C* **mean the storage of the plugin:** *D* **- Distribute,** *C* **- Centralize.** *F*/*M* **denote the plugin run from:** *F* **- File,** *M* **- Memory.** *P*/*D*/*V* **mean the isolation of plugin:** *P* **- Process,** *D* **- Docker,** *V* **- Virtual Machine.**

CIPs	Plugin Type	Storage	Resolving By	Plugin Run From	Version Control				Isolation	Secrets
					Git Tag	Branch	Commit Hash	Version Code		On-demand?
GitHub Actions	*S*	*D*	Runner	*F*	✓	✓	✓		*P*, *D*	✓
	R	*D*	Controller	*M*	✓	✓	✓		*V*	✓
GitLab CI	*S*	*D*	Controller	*M*	✓	✓	✓		*D*	
	R	*D*	Controller	*M*	✓	✓	✓		*D*	
CircleCI	*S*	*C*	Controller	*M*				✓	*D*	✓
Azure Pipeline	*S*	*C*	Runner	*F*				✓	*P*	✓
	R	*D*	Controller	*M*	✓	✓	✓		*V*	✓

There are some previous works that focus on CI security, particularly on code injection related vulnerabilities. Such vulnerabilities pose a significant threat to CI workflows, as the code executed in CI may come from untrusted users. Argus [64] is a static taint analysis tool for identifying code injection vulnerabilities in GitHub Actions. Argus sets sensitive taint sinks derived from GitHub documentation and tracks data flows in GitHub Workflow and Actions. Argus has discovered code injection vulnerabilities in 4,307 workflows and 80 GitHub Actions. In addition, resource isolation can also cause security risks in CI systems [48]. The weak isolation among CI components can leak various tokens used for authorization, and further cause privilege escalation or code injection.

3 CI PLUGIN IMPLEMENTATIONS

Using plugins in the CI pipeline essentially introduces the execution of code snippets in CI tasks. As plugins might come from anyone including unethical developers, it thus enables new attack surfaces for software supply chain attacks targeting the software build process. Understanding the implementation details of plugins is important for investigating its security issues. This section introduces the details of plugins in four CIPs (i.e., GitHub Actions, GitLab CI, Azure Pipelines, and CircleCI).

We interpret CIP plugin implementations by carefully analyzing their official documents and source code (if open-sourced like GitHub Actions). We also conduct black-box testing on both self-hosted and CIP-hosted runners. Particularly, in each CIP, we create a pipeline for a public repository and set up a runner for this pipeline. In this pipeline, we use mitmproxy [71] to monitor networking traffic when executing our pipeline. We also record the pipeline execution logs (if saved by CIPs). GitLab CI does not save logs, but its debug mode can print the details in the terminal. In addition, we create two types of secrets, a group/organization secret, and a repository/project secret, to observe how secrets are processed and isolated. The overall features of different CIPs are shown in Table 1.

3.1 Plugin Storage and Distribution

Azure Pipelines and CircleCI provide a *centralized* marketplace for storing standard plugins, and standard plugins must be published in their official marketplace to become available. Other plugins can be used in a *distributed* approach: users can directly download plugin code from CHPs, although there are still official plugin marketplaces (e.g., GitHub Marketplace for GitHub).

The accessibility of distributed plugins depends on their hosting repositories: if the hosting repository is public, the plugin can also be used by any users; Otherwise, for a repository that can only be accessed within an organization, its plugin can only be used within the organization. For centralized plugins, plugin maintainers can choose to publicize the plugin. For example, Azure Pipelines maintainers can set the `public` attribute in the `vss-extension.json` (an extension manifest file including plugin information on the marketplace) to publish a standard plugin.

3.2 Plugin Parsing

We find that there are two ways for CIPs to acquire and parse plugins during CI task execution, either by (1) CI runner or (2) CI controller. GitHub Actions and Azure Pipelines adopt the first approach for standard plugins: at the beginning of a CI task (before executing any code), the CI runner will download all required plugins (into the runner host), and later execute them during the CI task execution. Particularly, GitHub Actions and Azure Pipelines download standard plugins into the default folders *_actions* and *_tasks* in the runner, respectively. For this approach, if plugins are not properly isolated in the runner, one plugin might modify the code of other plugins. CircleCI adopts the second approach for parsing standard plugins. The code of plugins will be downloaded into the controller for initial parsing. Then the controller will distribute the complete CI job workflow to the runner.

For file reference plugins, CIPs generally apply the second approach: download plugins into the CI controller first for initial processing (e.g., parsing CI jobs). Then, the CI controller will distribute the complete CI job flow, including file reference plugins, to the corresponding CI runner. In particular, GitHub Actions and Azure Pipeline import file reference plugins into separate jobs. GitLab CI instead introduces file reference plugins as parts of a job. The controller parses the code and variables into the job workflows, which are sent to the runners.

Input Initialization. Some plugins need input arguments for execution. For example, `circleci/aws-cli` plugin needs to read AWS access token for login. These arguments that are explicitly declared by plugins can be passed by the CI runner via environment variables. In addition, the CI controller might dispatch some hidden sensitive data, which might be accessible by plugins, to runners. For example, GitHub Actions generates an `AccessToken` for each CI task to access various resources. Such token can be read by plugins.

3.3 Version Control

There are four different approaches for plugin version control: (1) via git tag; (2) via git branch name; (3) via commit hash value; and

(4) via the version code in centralized plugin markets. The first three methods are supported by GitHub Actions and GitLab CI, as well as the file reference plugins in Azure Pipelines. The (4) method is adopted by Azure Pipeline and CircleCI for standard plugins. Particularly, they both utilize *semantic versioning*, which consists of `Major.Minor.Patch` versions [69]. In addition, Azure Pipeline and CircleCI provide developers with the option to only specify the `Major` version. In this case, subsequently, they will retrieve the latest version within that `Major` version. For instance, if developers specify the plugin `circleci/aws-cli@4`, it will retrieve the latest version within the 4.x.x series, such as 4.1.2.

3.4 Plugin Isolation and Sharing

Inner-Job Isolation. From our experiments, we find that almost all plugins adopt a process-based isolation strategy inside one job: a new process is spawned for launching a plugin. However, CI runners use one user namespace for all running plugin processes. Also, in our tested CIPs, the CI runners do not offer sandboxes for executing plugins. Thus, inside one job, all plugins and the runner have the same permission for accessing system resources (e.g., files).

The only exception is the Docker mode of standard plugins in GitHub Actions: the runner starts a Docker container, and runs the plugin inside the container. Thus, plugins are better isolated from each other as they are running in different containers.

Inter-Job Isolation. All CIPs adopt job-level isolation in a CI workflow: a separate virtual machine (VM) is launched for each job in self-hosted runners. For CIP-hosted runners, GitLab CI and CircleCI use Docker containers for inter-job isolation, while GitHub and Azure Pipeline still adopt VM-based isolation. With the isolation, a plugin cannot access system resources of other jobs by default.

We find that CIPs provide three methods for data transfer between jobs. (1) Supposing job_1 would like to transfer data to job_2, job_1 can configure the data as outputs (e.g., using the `outputs` variables), and then job_2 can read it as inputs. For example, GitHub Actions will first store job_1's outputs. When executing job_2 later, GitHub Actions will include job_1's outputs into job_2 during the parsing procedure, and then launch the VM for running. (2) The second approach is via artifacts. job_1 can generate artifacts for storing data, which can be loaded when job_2 is executing. For example, CircleCI can set a workspace (e.g., `persist_to_workspace`) to specify the path for storing data (e.g., AWS S3 storage), and other jobs can download it from using the workspace (e.g., `attach_workspace`). (3) The third approach is using cache objects for inter-job data transfer. Similar to artifacts, jobs can generate cache objects in TPSs, which can be used by other jobs later. We find that the (2)(3) methods are supported by all CIPs, while the (1) method is supported by GitHub Actions and Azure Pipeline.

Cached Plugin. Interestingly, we find that Azure Pipelines will cache plugins when running in a self-hosted runner. Particularly, Azure Pipelines automatically store used plugins into the runner host machine (i.e., *_task*), which can be used by subsequent CI tasks.

Plugin Cleanup. After finishing a CI task, all involved plugins should be cleanup, including deleting plugin files and cleaning input/output environment variables. The purpose is to prevent plugins from affecting subsequent CI tasks, which might be from a different repository.

3.5 Secrets Accessibility

As *secrets* are one of the most valuable data in CI workflow [62], CIPs may adopt additional mechanisms to secure secrets. We find that GitHub Actions, Azure Pipeline, and CircleCI take an *on-demand* approach for managing secrets for plugins in CI jobs: a plugin can access a secret only if the plugin explicitly uses this secret (e.g., Line 14 in Listing 1). Among them, CircleCI and Azure Pipeline support using secrets in plugins directly. The CI controller will pass the requested secret to the runner via the corresponding API, and further pass it to the plugin as environment variables. GitHub Action adopts a stricter way: all secrets must be passed to plugins as inputs. We find that GitLab CI has not adopted the on-demand mechanism. By default, it passes all secrets in the pipeline to all jobs and plugins.

4 CI PLUGIN THREATS

This paper investigates potential security vulnerabilities in the CI plugin ecosystem, which might be exploited to perform various software supply chain attacks (e.g., injecting malicious code/vulnerabilities into a repository [48, 64] and sensitive data leakage [23, 62, 74, 78]). We aim to understand what potential threats could be brought when integrating CI plugins, which essentially are reusable code snippets and might be developed by less trusted users. In the following, we first introduce the threat model (Section 4.1) and then elaborate on potential vulnerabilities related to unsafe code distribution (Section 4.2) and improper plugin isolation (Section 4.3). Finally, we also discuss how plugin dependency can amplify the attack impact (Section 4.4).

4.1 Threat Model

Overall, we consider a typical CI scenario adopted by an organization, similar to recent research [48]. The organization utilizes code hosting platforms to maintain multiple repositories, and CI with plugins enabled for software development. We consider all main stakeholders, including CHPs, CIPs, and TPSs, to be trustworthy. Also, the communication channels among them are secure and cannot be exploited by attackers. In addition, unless explicitly mentioned, we assume vulnerabilities disclosed by [48] have been fixed and cannot be exploited.

With the above common assumptions, we consider adversaries as *unethical plugin maintainers*, who have almost no privileges in the victim repository. We assume the adversaries can control a CI plugin that is used by the target organization. To achieve this, attackers can hijack an existing used plugin (by the target) by exploiting vulnerabilities in the plugin distribution system. Or attackers can bait the target to use their published plugins. For example, attackers can first publish a benign plugin to attract victims, and upgrade the plugin later for malicious activities. Finally, they can also exploit typosquatting techniques [20, 49, 55, 61] to attract victims into downloading their plugins.

After controlling a used plugin, attackers then attempt to secretly inject malicious code into the plugin without being suspicious. For example, if attackers upgrade the plugin with malicious code injected, the modification should not be noticed by victims throughout the CI pipeline unless victims specifically look into the code in detail. Moreover, we assume the adversaries' capabilities

Table 2: Overview of potential threats on each CI platform. The ✓ means vulnerable.

Threats	GitHub Actions	GitLab CI	CircleCI	Azure Pipelines
D1	✓	✓		
D2	✓	✓	✓	✓
D3				✓
A1	✓	✓		✓
A2 - A	✓	✓	✓	✓
A2 - B	✓	✓	✓	✓
A2 - C	✓	✓		✓
A3	✓		✓	✓
A4	✓		✓	✓
A5	✓	✓	✓	✓

are limited within the CI plugin, which means that all malicious activities can only be initiated from the controlled plugin. Thus, attackers need to escape the sandbox of plugins and launch attacks against different levels of isolation.

Particularly, referencing a malicious CI plugin exposes new attacking surfaces compared to introducing a malicious third-party library (e.g., linking a malicious library statically). With the extensive functionality provided by the CI task, CI plugins can not only inject malicious code into software artifacts (similar to a library), but also manipulate the software development process (e.g., affect the release of an artifact that has been injected with malicious code), as well as steal high-value data (i.e., secrets) that is only visible during CI execution.

In summary, we focus on CI plugin vulnerabilities that allow attackers to (1) hijack existing benign plugins; (2) update plugins without being noticed; and (3) escape the isolation and gain access to unauthorized resources. Table 2 presents a summary of investigated CIPs with potential threats.

Ethical Concerns. All experiments are conducted in an ethical way. First, we never attempt to compromise existing CIP systems or vulnerable repositories. Instead, all security threat tests are conducted and confirmed on our own resources, including repositories, projects, and plugins. Particularly, we register all accounts legally through the web interface, and create corresponding resources in a legitimate way. During our experiments, repositories and plugins are set as private when possible. At the end of our study, we have manually deleted all of our published resources. Moreover, Section 6.2 details our disclosure.

4.2 Unsafe Plugin Distribution

Unsafe code distribution is a classical security risk in the software supply chain [9, 49, 57]. Unfortunately, CI plugins also suffer from various threats that can potentially distribute malicious code to users without plugin users being aware.

4.2.1 ***Plugin Redirection Hijacking Attack (D1).*** Plugins that are decentralized stored rely on CHPs (e.g., GitHub and GitLab) for plugin storing and distribution. When the CHP account hosting the plugin has changed (e.g., user rename or user account deletion), such change might be unknown to plugin users, who might still use the old location (e.g., account name) for downloading. Thus, CHP will create a redirection to link the old and new locations, and both locations can be used to download the plugin.

The problem is that the old location becomes available and can be hijacked by attackers, similar to the package redirection hijacking attack in decentralized software registries [49]. As a result, CI users who still use the plugin's old location will also be hijacked and download malicious plugins maintained by attackers.

In this attack, attackers simply need to locate the redirection where the old location is available (e.g., the user account for the old location is available for re-register). Attackers can take over the old location and hijack CI users without (1) compromising any existing accounts or (2) executing a CI task in the victim repository.

We find that the standard plugin of GitHub Actions, and GitLab CI are vulnerable to this attack. CircleCI and Azure pipeline adopt a centralized plugin marketplace for standard plugins, and thus not vulnerable. In addition, the file reference plugins of GitHub Actions, GitLab CI, and Azure Pipeline are also vulnerable to this attack.

4.2.2 ***Plugin Version Reuse Attack (D2).*** CI Users can use *version pinning* to specify the version of a CI plugin that can be adopted, ensuring the same codebase is always used so that potentially vulnerable/malicious versions can be avoided. However, if the plugin maintainer can change the code of a specific version without changing the version number, the version pinning functionality becomes invalid. Thus, allowing plugin version reuse can be problematic and even cause potential security risks. Obviously, directly publishing a malicious plugin is suspicious as users can easily identify malicious code. Instead, attackers can first publish a normal plugin to attract users. After victim users have referenced this normal plugin, attackers can update the plugin with malicious code injected, while keeping the version number unchanged. Similarly, with the same version number, users might not notice the malicious code change.

Typically, mainstream package registries, such as npm, PyPI, and Maven, explicitly prohibit changing the code of a package's published version. However, similar restrictions might not be applied to plugins. As mentioned in Section 3.2, CIPs adopt four different approaches for plugin version control, using: (1) git tag; (2) git branch name; (3) commit hash value; (4) version code in a centralized marketplace.

Among them, only the (3) approach is a safe choice, as the commit hash is bound with the code. Git used to use SHA-1 hash function, which was demonstrated to be vulnerable to practical hash collision (i.e., the SHAttered attack [77]) on 2/23/2017. Git later switched to a hardened SHA-1 implementation by default, which is not vulnerable to the SHAttered attack [29]. In late 2018, Git eventually picked SHA-256 as the hash method [26, 27]. In addition, we find that some users use the abbreviated commit hash (e.g., the first seven characters) to reference a plugin. In this case, Git mentioned that a short and unique abbreviation for the commit hash is enough to be unique within a project, but still suggests users use a longer commit hash to avoid ambiguity [28]. Thus, we consider the hash method is not vulnerable to the version reuse attack.

Both methods (1) and (2) are vulnerable to version reuse. Particularly, Git allows deleting a git tag. Attackers can simply delete a git tag, modify the code, and then re-publish the same git tag.

When using the git branch/tag name for version control, we find that CIPs will always obtain the plugin with the latest code in that branch/tag. So users will always get the latest plugin when the branch/tag name has remained the same. It means that users will always get the latest plugin even if the branch name has remained the same. It affects GitHub Actions, GitLab CI, and file reference plugins in Azure Pipeline.

Additionally, we observe that both GitHub Actions and GitLab CI allow the creation of branches and tags with the same name. In the case that both branch and tag exist with the same name when referencing a standard plugin, GitHub Actions prioritizes the branch version by default, while GitLab CI prefers the tag version instead. When referencing a file reference plugin, both GitHub Actions and GitLab CI prioritize the tag version. This mechanism potentially enables attackers to launch the version reuse attack without modifying the existing version. For example, suppose GitHub users utilize tag as plugin version control, plugin maintainers can publish a new branch with the same version number, which will lead to the following users obtaining the newly published branch version without being noticed by users.

Finally, whether the (4) approach is vulnerable or not depends on the CIP's implementation, which is unknown to us. We find that neither Azure Pipelines nor CircleCI allows a published plugin (in the marketplace) to change its version code. However, if developers only specify the `Major` version of a standard plugin, the associated repository may still be vulnerable as it always automatically retrieves the latest `minor` or `patch` versions under the `Major` version (as discussed in Section 3.2). In this case, the `Major` version is reused, while its `minor` or `patch` versions may contain potentially malicious code. In summary, all CIPs are potentially vulnerable to the plugin version reuse attack.

4.2.3 ***Cached Plugin Poisoning Attack (D3).*** As mentioned in Section 3.4, CI plugins might be cached by CIP to enhance CI tasks' performance. A cached plugin can be reused by subsequent tasks without re-downloading it again. However, if there is no proper checking/protection mechanism on cached plugins, a poisoned plugin (e.g., with malicious code injected) can also pose security risks on subsequent CI tasks. For example, if the runner can be shared by different repositories in an organization, a poisoned plugin might be reused by CI tasks from a different repository that attackers have no permissions. Thus, depending on the plugin caching policy (e.g., cross-repository or cross-job), it enables attackers to launch attacks to break different isolation mechanisms.

We find that the standard plugin of Azure Pipeline with a self-hosted runner is vulnerable to this attack. In Azure Pipeline, the self-hosted runner (i.e., agent) will check the existence of the target plugin's latest version in the *_task* folder before downloading. If exists, the agent will simply skip the download process, and reuse the one stored in *_task*. Even worse, Azure Pipeline manages an agent pool containing multiple agents. If the agent pool belongs to an organization, Azure Pipeline will select an agent (from the pool) that meets the requirement (e.g., has a plugin) for running a task. Thus, within the agent pool, a malicious plugin from one repository might be selected later for running CI tasks from other repositories. Particularly, when a job containing a malicious plugin is running on a self-hosted agent, the malicious plugin can iterate the cached plugins in the *_task* folder, and then inject malicious code to them (e.g., their code files). The cached tampered plugins will be executed in another repository in the same organization if it also picks this agent to run a job.

4.3 Improper Plugin Isolation

Below we introduce several inner-job and inter-job security threats caused by weak plugin isolation.

4.3.1 ***Inner-Job Cross-Plugin Hijacking Attack (A1).*** A CI job might contain multiple plugins from different sources. Some CIPs parse plugins using the CI runner (discussed in Section 3.2). The plugin's code is first downloaded and stored locally in the CI runner's hosting machine, and will be loaded later when executing particular steps. Obviously, this will cause security issues if a CIP only adopts process-based resource isolation in the runner. Basically, the processes of all plugins and the runner are running in the same user namespace and share similar privileges. Thus, one plugin process can access files and environment variables belonging to other plugins.

In this situation, during the loading/executing procedures by the CI runner, a malicious plugin can easily modify other plugins' code files to inject malicious code. Injecting malicious code into other plugins can help attackers to hide themselves to some extent. But more importantly, it may enable attackers to further escalate their privileges, such as accessing secrets or modifying artifacts. As we tested in Section 3.5, some CIPs such as GitHub Actions adopt the on-demand approach for managing secrets, and only pass secrets to plugins that explicitly use secrets. Malicious attackers can then inject code into benign plugins to steal sensitive data.

Breaking the plugin order. The aforementioned method can only enable a malicious plugin to hijack its subsequent plugins. If the malicious plugin attempts to hijack a preceding plugin, although the malicious code can be successfully injected, the code will not be executed in practice. However, we find that the *pre-steps* for plugins can be exploited by attackers to hijack all plugins inside a job, regardless of the plugin executing sequence.

Particularly, the plugin execution can be further divided into three stages: pre-steps, main steps, and post-steps. The *main steps* contains plugin's core function, so every plugin must have main steps. The *pre-steps* allow a plugin to run code at the start of a job (before the main step begins) after the job setup. The *post-steps* allow a plugin to run code at the end of a job (once the main steps have been completed). For example, for two plugins (A and B) that contain both pre- and post-steps, if A locates before B, the execution sequence is: A.`pre`, B.`pre`, A.`main`, B.`main`, B.`post`, and A.`post`. As pre-steps will always be executed before the first plugin's main code, attackers can simply define a pre-step to inject malicious code and hijack any other plugins.

We find that, the standard plugins of GitHub Actions and Azure Pipelines, as well as both types of GitLab CI, are vulnerable to this attack. Particularly, in GitHub Actions, both JavaScript and shell execution environments are vulnerable to this attack; but the Docker execution environment is not. The reason is that the plugin's code is isolated by containers, which are not accessible to malicious plugins without root privilege. Azure Pipeline and GitLab CI also support setting *pre-steps* (or similar concepts).

4.3.2 ***Sensitive Data Leakage Threat (A2).*** Sensitive data leakage is a severe security risk in CI. Unfortunately, we find there are vast spaces for malicious plugins to steal various types of sensitive data, affecting all CIPs.

Inner-Job Secrets Leakage (A2-A). As introduced before, CIPs have adopted on-demand secret management to mitigate secret leakage. The problem is that, even secrets are isolated between plugins, a malicious plugin can still mount cross-plugin hijacking attacks to extract secrets from any plugins in the same CI job.

Token Leakage (A2-B). Tokens are largely utilized in CI for authorization, and token leakage can cause various security risks such as privilege escalation [48]. With only process-based isolation, a malicious plugin in a CI task can also obtain vulnerable tokens introduced in the previous work [48]. We have two new findings.

First, we find a new vulnerable GitHub Actions token, namely *ACTION_RUNTIME_TOKEN*, that can be exploited by attackers to break the cache isolation of branches and tags. This token is used to access CI cache objects, identifying the cache's scope (e.g., branches or tags). Particularly, the token is preserved in the runtime environment variables (i.e., *process.env*) during executing CI actions, which can be directly read by malicious plugins as they are running in the same namespace as the runner.

Second, in self-hosted runners, some essential tokens are stored in local files (e.g., *.credentials*). These tokens are generated when self-hosted runners register with the CI controllers, and are used for CI task assignments. The runners do not restrict plugins from accessing local files. Thus, attackers can inject codes to traverse the files in the runner to obtain these credentials. Even worse, we find that runners are not properly isolated, and plugins can access files beyond their own runner. If multiple runners are running on the same host machine, credentials from all runners can be leaked to malicious plugins.

Source Code Leakage (A2-C). In self-hosted runners, the source code of repositories is downloaded before executing the CI actions, and the downloaded code remains in the runners. Similarly, the source code in runners can also be leaked to malicious plugins. Even if there are multiple runners (e.g., a runner for a public repository and a runner for a private repository), because the filesystems of runners are not properly isolated, malicious plugins in one runner can obtain source code stored in all runners in the host.

In summary, we find that all four CIPs are vulnerable to the secrets leakage. All four CIPs support using self-hosted runners, which register and restore tokens in local files, are vulnerable to the token leakage. For the source code leakage, GitHub Actions, GitLab CI, and Azure Pipelines are vulnerable to this threat.

4.3.3 ***Inter-Job Control Flow Hijacking Attack (A3).*** CI job is the smallest unit of CI permission, and jobs are typically isolated. All CIPs investigated in this paper have adopted VM-level or Container-level isolation for isolating jobs. Thus, a plugin can execute code within its hosting job in a CI task, but cannot affect other jobs or tasks. The problem is that CIPs allow data transfer between jobs: the subsequent job can take the output of a previous job as input, and further take different actions correspondingly. Without a proper sanitation method on jobs' outputs, a malicious plugin might escape the sandbox of its hosting job and affect the entire CI workflow (e.g., affecting subsequent jobs), potentially causing severe security risks. GitHub also states that "*a compromise of a single action within a workflow can be very significant.*" [43]

```
# Vulnerable CI configuration file
steps:
  - name: Check PR title
    run:  title="${{ github.event.pull_request.title }}"
# Based on a malicious input (i.e., a"; ls /""), the code that
      is actually executed is as follows
run: title="a"; ls /""
```

Listing 2: Example of a normal input injection attack in GitHub Actions.

Malicious plugins in one job cannot directly affect other jobs (e.g., cross-job attacks). However, if the execution of subsequent jobs depends on the job containing the malicious plugin (e.g., the subsequent job can take the output of a previous job as input, and take different actions corresponding to different inputs), a malicious plugin can inject code in its hosting job to hijack the control flow of the CI workflow, potentially causing serious security risks.

For example, it is quite common to deploy a static application security testing (SAST) job to conduct security testing before releasing the software. The subsequent release job will take the SAST job's output, and only release software when the output shows that the SAST is passed. A malicious plugin can then modify the SAST job's output and execute software release if the SAST is failed, so that malicious code can be injected into the victim repository.

We find that the output of a job can easily be tempered by a malicious plugin in multiple CIPs, including GitHub Actions, Azure Pipeline, and CircleCI. In particular, for GitHub Actions and Azure Pipeline that support job outputs, attackers can simply launch the aforementioned inner-job cross-plugin hijacking attack (e.g., exploiting *pre-steps*) to hijack the benign plugin that generates outputs. In CircleCI, inter-job communication is achieved by workspace. Attackers can modify the shared workspace before it is attached to subsequent jobs.

```
# Vulnerable CI configuration file
jobs:
  info:
    outputs:
      ...                    ❸💣 Passed to the output job
      mod_id: ${steps.mod_id.outputs.value}
    steps:
      ...
      - name: Action 1
        id: mod_id    ❷💣
        uses: dreamli0/my-action@main
        ...
     - name: Action 2
        id: action_2
        uses: ciplugins-poc/my-action@v1.0.1  ❶😈 Inject code
              to the mod_id step to rewrite mod_id's value in
              the pre-step
        ...
  output:
    needs: [info]
    steps:              ❹💣 Cross-job input injection triggered
        - run: echo "${{needs.info.outputs.mod_id}} version
              ${{needs.info.outputs.version}};""
```

Listing 3: PoC (based on a real example) of an inter-job input injection attack in GitHub Actions [19].

4.3.4 ***Inter-Job Input Injection Attack (A4).*** If CIPs do not properly sanitize inputs, they may be vulnerable to code injection, similar to SQL injection. Input injection in the CI task is a well-known security threat [64]. Listing 2 demonstrates an input injection example in GitHub Actions. The *run* step (Line 4) is vulnerable, as it reads input from GitHub (`"${{ github.event.pull_request.title}}"`. The input indicates that the pull request name (which can trigger the CI task) can be set by the pull request initiator, who could potentially be an attacker. The attacker can develop a string `a";ls/""` as the pull request title. During the CI task execution, it actually executes `title="a";ls/""`, allowing attackers to execute any command. To mitigate the input injection attack, CIPs have adopted many defense mechanisms. For example, GitHub Actions have listed 10 types of input that might be exploited by attackers [41].

However, CI plugins can largely expand the attacking surface of input injection (e.g., not limited to some special inputs like `title`), as malicious plugins can launch cross-step or cross-job attacks to tamper with any inputs. Particularly, if a plugin's output is directly used as the input to be run (i.e., use the 'run' keyword) by the next job, the plugin can inject commands into the next job.

Even worse, a plugin's output might be modified by another plugin in the same job using inner-job plugin hijacking attack. Basically, if the output is generated from a JavaScript plugin, the output is set using JavaScript code (i.e., `core.setOutput`). Then, a malicious JavaScript plugin can inject code into victim plugin's code file in the *pre-step* to modify the output (which is controlled by victim plugin). Listing 3 demonstrates a proof-of-concept that can be abused by a plugin to mount the inter-job input injection attack. In the job *info* (Line 3), the plugin `ciplugins-poc/my-action@v1.0.1` (Line 15), which is developed by a different user, can inject code (`core.setOutput("value",'";id;echo "');`) to the *mod_id* action in the pre-step (step ❶), when *mod_id* action is executed, the injected code is executed to set the output of `mod_id` with the malicious value (i.e., `";id;echo "`) (step ❷). After the job *info* is finished, the malicious output of `mod_id` is passed (step ❸) to the *output* job (Line 17) within the same CI task. Then, the malicious output is injected into the command in the `run` step (Line 20) to form a malicious command (`echo "";id;echo " version main"`), thus the cross-job input injection (i.e., `id`) is triggered (step ❹). Due to the page limit, the detailed steps are illustrated in our PoC [19].

4.4 Plugin Dependencies: Attack Amplification (A5)

Like software packages, CI plugins also utilize external resources (e.g., other plugins) for accelerating development, which forms plugin dependency chains. Unfortunately, plugin dependencies might largely enlarge the attack surface. When executing a CI workflow, a plugin and its reference plugins will both be executed in the same *step*, which lacks strong isolation. Thus, if a plugin is malicious/vulnerable, all its dependent plugins and their CI workflows become vulnerable. For example, if a plugin is vulnerable to the redirection hijacking attack or version reuse attack, attackers can hijack this plugin, and further attack its dependent plugins.

Cross-plugin Sensitive Data Leakage. A malicious plugin can leak sensitive data via its reference plugins. For example, a plugin can pass secrets to its reference plugin (as input), which further leaks the secret to external servers (e.g., sending an HTTP request).

Dependency. There are three different methods for external dependency. The first method is to directly reference other plugins. For example, GitHub composite plugins can reference other plugins in the metadata file (i.e., `action.yml`). Other CIP plugins supporting this approach include GitLab CI and CircleCI. The second method is to reference an external software package (e.g., npm packages) to extend functions in JavaScrpt plugins. Lastly, a plugin can also rely on external web resources by sending HTTP requests.

5 MEASUREMENT

To investigate the current security status of plugins and understand the potential impact in the open source community, we conduct a large-scale measurement on GitHub and GitLab.

5.1 Data Collection and Methodologies

Repositories Collection. We adopt multiple strategies for repository data collection, similar to previous works [48, 56]. Our target is repositories (in CHPs) that contain a CIP configuration file (e.g., `.github/workflows/` for GitHub Actions, `.gitlab-ci.yml` for GitLab CI, `azure-pipelines.yml` for Azure Pipelines, and `.circleci/config.yml` for CircleCI). For repositories hosted on GitLab, we retrieve all public repositories and their contents through GitLab's public APIs [17, 18]. For repositories hosted on GitHub, we are unable to perform a full crawl since GitHub has adopted API rate limits (i.e., 5,000 requests/hour) with only the first 1,000 results returned for each search query [56]. We then extract repositories using GitHub Actions from the *GHArchive* [70] data. In addition, we parsing the *GitHub Activity Data* [8] to gather information on repositories that utilize CircleCI and Azure Pipelines.

In GitHub, we have collected 686,896 repositories using GitHub Actions, 12,771 repositories using CircleCI, and 2,773 repositories using Azure Pipelines. In GitLab, the number of repositories for GitLab CI, CircleCI, and Azure Pipelines is 612,469, 10,619, and 3,384, respectively. We find that the number of repositories using Azure Pipelines is limited in the open-source community. It is likely because most of their customers are large enterprises, based on recent reports [21]. Unfortunately, we cannot collect them as most of them are private repositories.

Plugin Collection. For standard plugins, we adopt two strategies for collecting plugins in different CIPs. First, CircleCI and Azure Pipelines provide centralized marketplaces. We thus crawl both marketplaces, and obtain 1,000 and 1,625 standard plugins in CircleCI and Azure Pipelines, respectively. For GitHub Actions, although there is an official plugin marketplace [36] maintaining 20,943 standard plugins, maintainers actually do not have to publish their plugins in the marketplace. Developers can use plugins that are not published in the market but exist in the public repositories. Thus, instead of crawling the market, we analyze all repositories using GitHub Actions and successfully extract 29,755 standard plugins via interpreting the *uses* statement (which is used for referencing plugins). GitLab CI has not provided a plugin marketplace yet, so we adopt a similar method and extract 52 standard plugins.

For file reference plugins, none of the CIPs provide plugin markets. We thus extract file reference plugins from repositories' configuration files based on their CIP's referencing rules. For example, GitHub references it using `uses: .../reusable-workflow.yml`).

Table 3: Some vulnerable repositories that reference plugins with redirection. (Gray color indicates that the repository has fixed the issue after our reports.)

Repositories	Stars	Referenced Plugins w/ Redirection
vercel/hyper	42.2k	jungwinter/comment
vlang/v	35.1k	jungwinter/split
pankod/refine	20.1k	probablyup/wait-for-netlify-action
symless/synergy-core	9.8k	jungwinter/comment
devicons/devicon	8.6k	jungwinter/comment
ripperhe/Bob	8.5k	ripperhefork/git-mirror-action ripperhefork/gitee-pages-action

The numbers are 4,840, 5,810, and 87, for GitHub Actions, GitLab CI, and Azure Pipelines, respectively.

After obtaining plugins, we further extract their inputs and outputs based on the formats defined in CIPs. Additionally, for plugins that are potentially vulnerable to the redirection hijacking attack, we use HTTP to download each of them and record their return status code (e.g., `HTTP 301`).

Threat Analysis. For the plugin redirection hijacking attack, we first identify all plugins that have redirection using their HTTP return code. We then utilize CHP APIs [38, 44] to check whether or not the username on the old location can be registered. If there is no username on the old location, attackers can simply register the old username and hijack the redirection by publishing a repository with the same name (as the target plugin).

For the plugin version reuse attack on GitHub Actions and GitLab CI, we analyze their configuration files. We collect potentially vulnerable repositories if they use plugins but do not employ the commit hash for version control. For repositories using standard plugins in Azure Pipelines and CircleCI, we consider repositories as vulnerable if they only specify the `Major` version of plugins.

For the cached plugin poisoning attack, we consider repositories using Azure Pipelines with plugins on self-hosted runner as vulnerable. Particularly, we analyze the configuration files to examine the usage of the self-hosted runner using the keyword '`pool: MyPool`', which indicates the pipeline is running on one runner from MyPool.

For inner-job cross-plugin hijacking attack, we consider a repository is vulnerable if multiple plugins are used in one job. We further evaluate the attack impact if one of the plugins is vulnerable to redirection hijacking attack or version reuse attack.

For both inter-job control flow hijacking attack and input injection attack, we conduct data flow analysis on configuration files. Particularly, we extract jobs that take other jobs' outputs as inputs, and further analyze their operations. If there is a conditional branch based on previous job's output, we consider the repository as potentially vulnerable to control flow hijacking. If the job executes shell scripts (e.g., `run:` in GitHub Actions), the repository is marked as vulnerable to input injection attack.

Finally, for sensitive data leakage attack, we focus on analyzing GitHub Actions to identify potential secret leakage in existing actions. We have integrated and modified Argus [64], a static taint analysis tool for GitHub Actions to identify code injection vulnerabilities, to perform source code analysis on GitHub Actions, with the dependency being considered. The details are presented in Sections 5.8 and 5.9.

Ethical Concerns. All measurement studies are conducted in an ethical way. We only collect data on open-source repositories and public plugin markets, which are publicly available to everyone. We either legally use the official public APIs or query from public datasets. As some platforms have rate limits on API queries, the entire data collection process lasts a couple of months to minimize any potential impacts. Our method is consistent with previous works [48, 56]. We have never attacked any collected repositories/-plugins. The detailed disclosure is presented in Section 6.2.

5.2 Plugin Redirection Hijacking Attack

From 29,755 GitHub Actions standard plugins, we find 1,391 (4.67%) plugins with redirection. Among them, we find 369 plugins are potentially vulnerable : their accounts corresponding to the old location have been deleted. In addition, many repositories still use the vulnerable plugin's old location. For example, the old location for plugin `stupidloud/cachewrtbuild` (i.e., `klever1988/cachewrtbuild`) is used by at least 726 distinct repositories. Table 3 shows some additional affected repositories, their number of stars on GitHub, and vulnerable plugins. For file reference plugins in GitHub Actions, we find 2 file reference plugins whose repositories have been deleted. For GitLab CI, among 52 standard plugins and 5,758 file reference plugins, there are 266 (4.58%) file reference plugins with redirection (i.e., vulnerable), affecting 449 repositories.

5.3 Plugin Version Reuse Attack

We consider repositories that use plugins in GitHub Actions or GitLab CI, but do not use git commit hash for version control as vulnerable. 684,558 GitHub repositories use at least one plugin in their configuration files. Surprisingly, 682,872 (99.75%) repositories use plugins with either branch or tag, and only 18,974 repositories use git commit hash. The number is similar in GitLab CI: among 96,776 repositories that use at least one plugin, 76,694 (79.2%) repositories use plugins with either branch or tag.

In addition, we use GitHub Archive data to identify whether or not existing plugins have modified code but still use the same git tag or branch. We have collected GitHub tag and branch creation and deletion events between January, 2019 and October, 2023. From 29,755 GitHub Action plugins, we find that 6,610 (22.24%) plugins have at least one git tag based version reuse that occurred in their plugins. 10,380 (34.88%) plugins have at least one git branch based version reuse occurred in their plugins. For example, scala-steward-org/scala-steward-action repository [75] is a popular plugin that has been starred by 130 times. One tag (i.e., `v2`) was recreated 26 times between Feb/2021 and Nov/2022. Our results indicate that not only version reuse is actually quite common in plugins, but developers also have no sense that plugin version reuse might be dangerous (largely use the unsafe method for referencing plugins).

Branch-Tag Mix Reuse. We find that 264 plugins have both the branch and tag sharing the same name on GitHub Actions. These plugins are referenced by 14,105 repositories. One of the most referenced actions is `codecov/codecov-action@v3` (used by 7,681 repositories), where 'v3' serves as both the branch and tag name.

`Major` Version Reuse. We find that 274 repositories use the `major` version to reference plugins in CircleCI. Among them, 93 repositories use `volatile` to reference plugins, which always pull the latest version of the plugins. In Azure Pipeline, it is very common to use

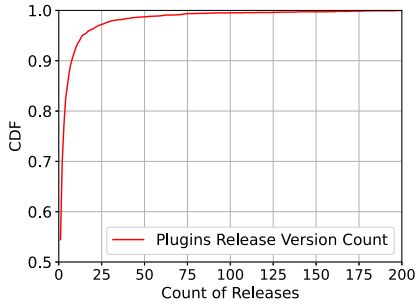

Figure 1: CircleCI Plugins' Release Version Count.

the `major` version for referencing plugins. We only detect 4 repositories to reference plugins using full version (i.e., major.minor.patch), which is not vulnerable. Most of them (4,087 repositories) reference plugins only using `major` versions (e.g., `UseDotNet@2` in the repository configuration file [5]).

We further crawl the version information of 1,000 collected plugins in the CircleCI marketplace. Then, we use the `major` number as the key to count the version release number under the same `major` version. Figure 1 shows the CDF of the plugins' release versions under the same `major` version among CircleCI plugins. It shows that such version updates are very common, and some of the plugins have been released under the same `major` version for hundreds of times. Thus, if attackers mount a version reuse attack abusing the `Major` version, it might affect many victims without being aware.

5.4 Cached Plugin Poisoning Attack

The repositories using Azure Pipelines with plugins on self-hosted runner are considered vulnerable to the cached plugin poisoning attack. In open source projects, we find 233 such repositories. While this number is not significant, it does not include private repositories such as enterprises that we are unable to measure. As Azure Pipelines are largely used by large enterprises/organizations (e.g., 38% of their customers have revenue larger than 1 billion dollars per year [21]), the attack might still cause severe consequences.

5.5 Inner-Job Cross-Plugin Hijacking Attack

Theoretically, if multiple plugins are used in one job, the repository is potentially vulnerable to this attack. We find many repositories fall in this category. Among them, we further evaluate the practical impact of this threat in two aspects.

Impact of Redirection Hijack-able Plugins. For a repository, if one plugin can be hijacked by the plugin redirection hijacking attack, other plugins (in the same jobs) are likely under threat. Among 369 plugins that can be hijacked in GitHub Actions, we find 4,469 repositories have used these hijack-able plugins, and 1,846 repositories have adopted other plugins in the same job. Furthermore, in 456 repositories, their "other" plugins (i.e., non-hijack-able plugins) have used secrets, which could be leaked. Similarly, we find that 318 hijack-able GitLab CI plugins potentially affect 423 repositories, and 26 of them might have secrets leaked.

Impact of Version Reuse Plugins. Similarly, we evaluate the impact if one plugin (in GitHub Actions) is vulnerable to the version reuse attack. Among 26,058 repositories that have used such GitHub Actions plugins, we find 2,031 repositories that have other plugins read secrets. Thus, if plugin maintainers launch version reuse attacks, they can potentially exploit the inner-job cross-plugin hijacking attack to further amplify the attack.

5.6 Inter-Job Control Flow Hijacking Attack

We first use the following conditions to find potentially vulnerable repositories: there are multiple jobs in the repositories; the preceding job uses at least one plugin; and the subsequent job relies on the output of the previous job for determining branch conditions. Our results show that this threat is not uncommon: the number of repositories is, 8,701 (1.27%), 577 (2.47%), and 23 (0.37%) in GitHub Actions, CircleCI, and Azure Pipelines, respectively.

To understand the impact of control flow hijacking, we further analyze the branch conditions and find several categories. One common usage is that the previous job's output is used by the subsequent job to decide whether or not to conduct unit tests or static code analysis for detecting potential code errors. For example, the `home-assistant/core` [3] has configured that the unit test and static code analysis tool are only launched when the changes are pushed to predetermined branches. Attackers can then exploit the vulnerability to pass the testing procedures. Another common use is to decide whether or not to create releases or further publish the package to registries (e.g., npm). For example, the `JanDeDobbeleer/oh-my-posh` repository [54] has configured a `skipped` output in the `changelog` job. Based on this output, the following `artifacts` job will execute packaging/releasing operations. Similar usages include deploying websites and other products. Attackers can then force the release even if it fails particular tests.

5.7 Inter-Job Input Injection Attack

We have analyzed repositories that are potentially vulnerable to the cross-job input injection attack in GitHub Actions. Particularly, we consider the input from other jobs as potential malicious inputs, and examine whether or not the input will be used in the `run:` statement. In addition, the preceding job should use actions, which can modify the job's output (e.g., exploit inner-job cross-plugin hijacking attack). We further remove the case that the preceding job only uses official actions maintained by GitHub. In total, we find 7,033 such repositories, including several widely used repositories that have thousands of stars on GitHub.

5.8 Sensitive Data Leakage

In general, plugins can obtain secrets as input. As introduced before, many CIPs adopt on-demand secrets management on plugins. GitHub limits the accessibility of plugin inputs to the individual plugin level. A plugin can only access its own inputs, but not the inputs of other plugins in the same workflow. However, a plugin might accidentally or intentionally leak its secrets. We consider a plugin to be problematic and there is a potential secrets leakage if it has one of the following four behaviors:

- **Set secrets as an output.** Other plugins after this plugin in the same workflow can access the output (i.e., secrets).
- **Set secrets as environment variables.** In general, the environment variables can be accessed by the plugins in the same workflow. Thus, secrets should not be set as environment variables, otherwise, it can cause potential leakages.

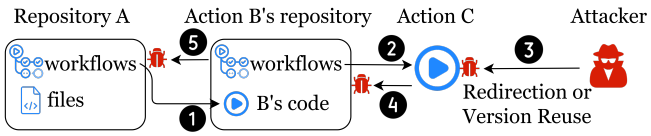

Figure 2: Dependent Actions in Action Repository Workflow.

- **Write secrets into local files.** Local files can be accessed by all the plugins in the workflow.
- **Use secrets as a parameter in the outgoing HTTP request.** The secrets are potentially leaked to external resources.

Method. We extend Argus to analyze the source code of GitHub Actions to identify potential problematic actions. Argus employs CodeQL [12] (a JavaScript code analysis engine) to perform static analysis on GitHub workflows (take repository as input). We extend Argus to track the dataflow of secrets in plugins, and add the above sinks (e.g., HTTP). With detected problematic plugins, we further find potentially vulnerable repositories by checking the configuration file.

We do not consider cleanups in the data leakage analysis. One reason is that data might be leaked even with cleanup, since attackers could monitor generated files. For example, a malicious plugin can abuse the pre-step to monitor files generated in the following steps during the execution (before cleanup).

Results. We find 1,097 potentially problematic plugins in GitHub Actions. Among them, 149 plugins use the secrets as a parameter in the outgoing HTTP request; 130 of them write the secrets to the local files; 161 set the secrets as environment variables; and 749 set the secrets as an output. Note that some plugins may set the secrets as environment variables and an output at the same time.

Moreover, we find 384 repositories pass secrets to these problematic plugins. For a detailed breakdown, 4 repositories pass secrets to plugins that use secrets as a parameter in outgoing HTTP requests. 15 repositories pass secrets to plugins that write secrets into local files. 330 repositories pass secrets to plugins that set secrets as environment variables. 299 repositories pass secrets to plugins that set secrets as an output.

5.9 Plugin Dependency Amplification

We further analyze GitHub Actions to understand the impact of plugin dependency on several identified threats. We focus on two dependency ways in GitHub Actions: (1) Dependent actions in composite actions. A GitHub composite action can run a combination of multiple commands or plugins. So, in a composite action, users can reference other actions, such as Javascript actions and other composite actions. (2) JavaScript actions can import npm packages.

We first extend Argus to analyze both composite actions and npm packages. We first check whether plugins have passed secrets to their referenced action or npm package, then utilize Argus to analyze them. Specifically, for npm packages, we extract them from JavaScript actions using regular expressions and further obtain their source code. We hook all functions (in npm packages) used by plugins, and check whether the npm package leaks its parameters to dangerous sinks (e.g., HTTP). Finally, we also check whether these referenced actions are vulnerable to redirection hijacking attacks and version reuse attacks.

Results. We find 6,565 composite actions that reference JavaScript actions. Among the referenced JavaScript actions, 169 potentially leak sensitive data to dangerous sinks, affecting 603 composite actions. Specifically, the number of plugins that leak secrets via HTTP, local files, environment variables, and outputs are 3, 16, 78, and 95, respectively. In addition, we find 17 repositories indeed pass secrets to them.

8 composite actions suffer from the plugin redirection hijacking attack: their referenced plugins are vulnerable to redirection. For the version reuse threat, unfortunately, only 627 out of 6,565 composite actions reference other plugins using commit hash. The rest can potentially be affected by the version reuse attack.

For npm packages referenced by JavaScript Actions, we find a total of 2,404 npm packages. No sensitive data leakage issues are found in them. However, 109 npm packages introduced by 337 plugins are not found in the npm registry. As these plugins rely on these npm packages, they might be malfunctioned in future versions, since the npm package is no longer available.

Dependent actions in actions' workflow files. We further find another interesting dependency among GitHub actions: dependent actions can exist in an action's workflow file. GitHub recommends storing a plugin in an independent repository, which can have a workflow file (i.e., the plugin's repository workflow). This workflow file can further reference other plugins. As shown in Figure 2, repository A references action B (step ❶), which is stored in an independent repository. In the workflow file of action B, it further references another action C (step ❷). In this case, the CI pipeline execution of repository A will not directly involve action C. However, action C might affect the build of action B, which in turn affects repository A. So, if action C is vulnerable to the redirection hijacking attack or the version reuse attack (step ❸), both action B (step ❹) and repository A (step ❺) might be affected. We find 19,503 plugins contain dedicated plugin repository workflows. Unfortunately, we find that 121 have referenced plugins that are vulnerable to the redirection hijacking attack. 460 plugins have referenced plugins using branch or tag names in their dedicated plugin repository workflows, which are vulnerable to the version reuse attack.

6 COUNTERMEASURES AND DISCLOSURE

6.1 Defense Practices

In this section, we propose several defensive practices that can potentially mitigate the discovered threats.

Improve the Plugin Management Mechanism. CIP can enhance plugin management practices following the software package management in software registries (such as adopting Multi-Factor Authentication). Particularly, for decentralized stored plugins, CIPs should regularly check plugin redirection on CHPs to prevent plugin redirection attacks. In addition, CIPs should enforce stricter version control mechanisms and explicitly prohibit version reuse in plugins. For plugins relying on external resources (e.g., external plugins), CIPs can provide additional services to examine external resources and their dependencies, similar to `Open Source Insights` [53] and `DependaBot` [16], notifying developers about potential threats and consequences.

Enforce Proper Isolation Between Plugins. This paper demonstrates that the process-level isolation adopted by plugins has severe

security risks, as a malicious plugin can easily modify others. Thus, CIPs should adopt stronger isolation mechanisms on plugins. For example, running plugins in Docker containers is generally better isolated than other mechanisms. CIP can also borrow other sandbox techniques (such as from Web browsers) to enforce isolation while keeping lightweight. Overall, a plugin should be limited in its own scope, without having the capability to access plugins' file systems or environment variables.

Protect Valuable Secrets. *Secrets* are one of the most sensitive data in CI workflows, and their access should be strictly confined. CIPs may adopt allowlists on secrets: only when developers explicitly specify in the CI configuration file that secrets can be accessed by a plugin, CI runner can transfer the secrets to the target plugin as inputs. Meanwhile, CI runners should carefully store and process secrets to avoid secret leakage. For example, CI runners should not store/set secrets into a shared global file or environment variables that are accessible by plugins.

6.2 Disclosure and Response

We have promptly disclosed our findings with affected CIPs and vulnerable repositories. We have followed the 90-day vulnerability disclosure policy [10]: we started our disclosure in November 2023, and completed all of them in May 2024, which is more than 90 days before the potential publication date of the paper (October 2024). Specifically, we did not communicate a time window but considered the time between disclosure and publication to be sufficient.

For vulnerable repositories, we attempted to collect their contact information from CHPs through their corresponding APIs [35]. However, not all repositories have contact emails (i.e., the repo's email field is empty [42]), as CHPs allow users to hide their contact emails [39]. In total, we have collected 365 repositories' emails. For these repositories, we then sent individual emails, by (1) mentioning the vulnerable repositories and plugins; (2) explaining the vulnerability; (3) indicating our recommended actions (e.g., update the plugin name to its latest location). During the disclosure, we got some email delivery failures as these emails were already outdated. For the rest emails, we have followed up with them if we have not received any response. In total, we have received 58 responses, with 39 of them replying that they have fixed the issue after receiving our emails. The rest replied that (1) they would fix it in the future; or (2) the repository is no longer in use thus there is no need to fix it. One repository owner has provided us with a redemption code as a gift to express their appreciation. For repositories that we could not collect emails, we find that 81% of them are not starred. Thus, we believe the impact is minimal.

For disclosures to CIPs, we have summarized all uncovered vulnerabilities (for each CIP), including the threat model, the ways to exploit them (e.g., version reuse, malicious code injection), impacts, and suggestions, and submitted them to their official channels (e.g., GitHub Support [33], Microsoft Developer Community [63]) and/or their bug bounty platforms (e.g., HackerOne [51]). For all CIPs, we have followed up with them if there is no response. For most findings, CIPs indicate that they are intentional designs and working as expected. However, they state that they will make some functionalities more strict in the future, or make corresponding suggestions in their official documents. For example, GitHub responded that "*This is an intentional design and is working as expected, we may make this functionally more strict in the future*". Azure Pipeline has converted our report to a suggestion, which "*will allow other developers to easily find it and engage on it*". Particularly, for the version reuse related issues, CircleCI responded that in the document they strongly suggest users reference the orbs (i.e., plugins) in the full version, which is the safest way. GitHub has confirmed our report and plans to update their documents to clarify the branch and tag case (i.e., same name issue).

In addition, for cached plugin and token leakage issues, we have further provided additional materials and tested repositories (with detailed explanations) as requested. GitHub has acknowledged our findings and stated that they might make changes to the token in the future. However, at the current stage, they do not plan to announce it. Azure Pipeline replies that they have investigated the issues and have forwarded our report/feedback to the appropriate engineering team.

7 RELATED WORK

CI Security Analysis. Extensive research efforts have been devoted to understanding CI security in recent years. As the most popular CI platform for open source projects, GitHub Actions also attracted much research attention. Benedetti et al. [6] identified 7 types of security issues in GitHub Actions, and developed GHAST tool to analyze GitHub Actions configuration files. Koishybayev et al. [56] found that most GitHub Actions' workflows are over-privileged and many secrets are leaked in plaintext. Li et al. [59] measured GitHub Actions workflows, revealing that lots of CI jobs were actually abused for illicit crypto-mining. In addition, improper configurations in CI workflow have been studied extensively [22, 25, 66, 73, 82, 83, 85]. For example, Vassallo et al. [82] developed `CD-Linter`, a static analysis tool, to identify and fix configuration smells in CI configuration files. Unlike previous works focusing on vulnerable/buggy workflows, we are the first to systematically study security risks focusing on CI plugins. Our work shows that, even with proper-privileged workflows, a malicious plugin can still cause security risks (e.g., leak secrets).

One closely related work is ARGUS [64], which is a static taint analysis tool to detect command injection vulnerabilities in GitHub Actions workflows. They focused on the command injection attack that the source can be initiated by CI users, and later are passed to various sinks for execution (e.g., shell `run` or `exec` in JavaScript). For example, `github.event.pull_request.title` allows users to initiate a pull request, and its title might contain code that can be executed later. Different from them, our research analyzes command injection vulnerabilities that cannot be triggered by users, but from malicious plugins. We demonstrate that, a malicious plugin can modify the output generated by other plugins (via the inner-job cross-plugin hijacking attack), and then inject command cross-jobs.

Gu et al. [48] studied the authentication/authorization process of CI and unveiled multiple security issues related to tokens. With insecure workflows, malicious users or collaborators (as attackers) can inject code into the insecure workflows to steal tokens. The leaked tokens might be further exploited to escalate the privileges of attackers. In contrast, we assume secure workflows in which malicious users/collaborators cannot inject code, but a malicious

plugin can exploit some vulnerabilities to inject code and potentially affect the entire pipeline. Particularly, we find that malicious plugins can steal some essential tokens stored in local files (e.g., .credentials). Finally, we also conduct an analysis of the impact of plugin dependencies, which are not studied by previous works.

Software Supply Chain Security. With the ever-increasing significance of the software supply chain in software development, many recent research works have been focused on understanding and enhancing the security of the software supply chain, including vulnerable packages [14, 15, 52, 79, 80], attack vectors in open-source software supply chains [49, 86], and malicious code in popular registries [24, 50, 60, 76, 84]. For example, Ladisa et al. [57] introduced a taxonomy of attacks on open-source software supply chains by surveying 17 domain experts and hundreds of software developers. Duan et al. [20] designed a tool to identify security threats in three popular registries, and detected 339 new malicious packages. Our work focuses on the security of CI plugins in the software development process, which is another perspective of software supply chain security that has not been systematically explored before.

Particularly, Gu et al. [49] identified twelve potential attack vectors in software registries, and uncovered many popular registries are threatened through a large-scale measurement study spanning one year over six software registries and seventeen popular mirrors. One of the threats disclosed by [49] is the *package redirection hijacking attack*. They find that package redirection can be hijacked in the Go registry, where package maintainers utilize GitHub or GitLab repositories to manage packages. One attack (plugin redirection hijacking attack) disclosed in this paper is motivated by their work. We demonstrate that, CI plugins in CIPs are also vulnerable to this threat. We have conducted a large-scale measurement study and demonstrated that many plugins used by many repositories can be hijacked immediately.

Finally, version reuse is known as a security risk in software package management, and thus has been disabled/banned by most package registries, including PyPI [81] and npm [65]. The semantic versioning specification has explicitly declared that *"Once a versioned package has been released, the contents of that version must not be modified. Any modifications must be released as a new version"* [69]. Gu et al. [49] also investigated the version reuse problem in existing software registries, and found that npm has reused package versions even after the ban of version reuse features in 2014. Unlike software packages, little attention has been paid to the CI plugin version control. However, we demonstrate that existing CI users largely utilize branch/tag for referencing specific versions of CI plugins. These methods are unsafe as CHPs allow plugin owners to delete a branch/tag and then recreate one with the same name. To the best of our knowledge, we are the first to comprehensively investigate the version reuse problem in CI plugins.

8 CONCLUSION

This paper systematically analyzes plugin-related security vulnerabilities in existing CI services. We have presented a detailed study of existing plugin implementation on mainstream CI platforms, and investigated seven security threats that can be exploited by attackers to inject malicious code and steal sensitive information. We have conducted a large scale measurement study on GitHub and GitLab, covering 1,328,912 open source repositories using CI plugins. Our experimental results show that many repositories and plugins are vulnerable to identified security threats. We have discussed potential mitigation, reported our findings to corresponding stakeholders, and received positive responses.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments. The University of Delaware team is partially supported by National Science Foundation (NSF) grants CNS-2054657, CNS-2317830, and OAC-2319975. Yacong Gu is partially supported by the Postdoctoral Fellowship Program of CPSF grants GZC20231361.

REFERENCES

[1] 2024. *Continuous Integration Solutions Market Size.* https://www.mordorintelligence.com/industry-reports/continuous-integration-tools-market
[2] Ionut Arghire. 2024. Major IT, Crypto Firms Exposed to Supply Chain Compromise via New Class of CI/CD Attack. https://www.securityweek.com/major-it-crypto-firms-exposed-to-supply-chain-compromise-via-new-class-of-ci-cd-attack/?utm_source=dlvr.it&utm_medium=twitter
[3] Home Assistant. 2024. Open source home automation that puts local control and privacy first. https://github.com/home-assistant/core
[4] Atlassian. 2023. *Bitbucket Cloud Variables and Secrets.* https://support.atlassian.com/bitbucket-cloud/docs/variables-and-secrets/
[5] AvaloniaUI. 2024. *Avalonia.* https://github.com/AvaloniaUI/Avalonia/blob/master/azure-pipelines.yml#L9
[6] Giacomo Benedetti, Luca Verderame, and Alessio Merlo. 2022. Automatic Security Assessment of GitHub Actions Workflows. In *2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses.*
[7] Bertus. 2018. Cryptocurrency Clipboard Hijacker Discovered in PyPI Repository. https://bertusk.medium.com/cryptocurrency-clipboard-hijacker-discovered-in-pypi-repository-b66b8a534a8
[8] GitHub Blog. 2017. *GitHub Data, Ready for You to Explore with BigQuery.* https://github.blog/2017-01-19-github-data-ready-for-you-to-explore-with-bigquery/
[9] Justin Cappos, Justin Samuel, Scott Baker, and John H Hartman. 2008. A Look in the Mirror: Attacks on Package Managers. In *2008 ACM Conference on Computer and Communications Security.*
[10] IEEE S&P 2024 CFP. 2024. Ethical Considerations for Vulnerability Disclosure. https://sp2024.ieee-security.org/cfpapers.html
[11] CircleCI. 2024. *CircleCI Orb Registry.* https://circleci.com/developer/orbs
[12] CodeQL. 2024. *CodeQL.* https://codeql.github.com/
[13] curl. 2024. *curl.* https://github.com/curl/curl
[14] James C Davis, Eric R Williamson, and Dongyoon Lee. 2018. A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning. In *2018 USENIX Security Symposium.*
[15] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Impact of Security Vulnerabilities in the npm Package Dependency Network. In *2018 International Conference on Mining Software Repositories.*
[16] Dependabot. 2024. *Dependabot Automated dependency updates built into GitHub.* https://github.com/dependabot
[17] GitLab Documentation. 2023. *Projects API - GitLab.* https://docs.gitlab.com/ee/api/projects.html
[18] GitLab Documentation. 2023. *Repository Files API - GitLab.* https://docs.gitlab.com/ee/api/repository_files.html
[19] dreamli0. 2024. *dreamli0/Inter-Job-PoC.* https://github.com/dreamli0/Inter-Job-PoC/blob/main/.github/workflows/blank.yml
[20] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *2021 Network and Distributed System Security Symposium.*
[21] enlyft. 2024. Companies using Azure Pipelines. https://enlyft.com/tech/products/azure-pipelines
[22] Wagner Felidré, Leonardo Furtado, Daniel A da Costa, Bruno Cartaxo, and Gustavo Pinto. 2019. Continuous Integration Theater. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.*
[23] Runhan Feng, Ziyang Yan, Shiyan Peng, and Yuanyuan Zhang. 2022. Automated Detection of Password Leakage from Public GitHub Repositories. In *2022 International Conference on Software Engineering.*
[24] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2021. Containing Malicious Package Updates in npm with a Lightweight Permission System. In *2021 IEEE/ACM International Conference on Software Engineering.*

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

[26] Git. 2018. *Doc Hash-function-transition: Pick SHA-256 as NewHash.* https://lore.kernel.org/git/20180725083024.16131-3-avarab@gmail.com/

[27] Git. 2024. *Choice of Hash.* https://git-scm.com/docs/hash-function-transition#_choice_of_hash

[28] Git. 2024. *Git Tools - Revision Selection.* https://git-scm.com/book/en/v2/Git-Tools-Revision-Selection#Short-SHA-1

[29] Git. 2024. *Hash Function Transition Background.* https://git-scm.com/docs/hash-function-transition#_background

[30] GitHub. 2023. *Encrypted secrets - GitHub Docs.* https://docs.github.com/en/actions/security-guides/encrypted-secrets

[31] GitHub. 2024. About Self-hosted Runners - GitHub Doc. https://docs.github.com/en/actions/hosting-your-own-runners/about-self-hosted-runners

[32] GitHub. 2024. Assigning permissions to jobs - GitHub Doc. https://docs.github.com/en/actions/using-jobs/assigning-permissions-to-jobs

[33] GitHub. 2024. *Bug Report.* https://support.github.com/contact/bug-report

[34] GitHub. 2024. Changing Your GitHub Username - GitHub Doc. https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-github-user-account/managing-user-account-settings/changing-your-github-username

[35] GitHub. 2024. *Get a user.* https://api.github.com/users/{USERNAME}

[36] GitHub. 2024. *GitHub Actions Marketplace.* https://github.com/marketplace?type=actions

[37] GitHub. 2024. *GitHub Actions Reusing workflows.* https://docs.github.com/en/actions/using-workflows/reusing-workflows

[38] GitHub. 2024. *GitHub Docs - Get a repository.* https://docs.github.com/en/rest/repos/repos?apiVersion=2022-11-28#get-a-repository

[39] GitHub. 2024. *Setting your commit email address on GitHub.* https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-personal-account-on-github/managing-email-preferences/setting-your-commit-email-address

[40] GitHub. 2024. Transferring a Repository. https://docs.github.com/en/repositories/creating-and-managing-repositories/transferring-a-repository

[41] GitHub. 2024. Understanding the risk of script injections - GitHub Docs. https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions#understanding-the-risk-of-script-injections

[42] GitHub. 2024. *A user does not set a public email address.* https://api.github.com/users/dynamoose

[43] GitHub. 2024. *Using Third-party Actions.* https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions#using-third-party-actions

[44] GitLab. 2024. *GitLab Docs - Get single project.* https://docs.gitlab.com/ee/api/projects.html#get-single-project/

[45] GitLab. 2024. Job permissions - Permissions and roles. https://docs.gitlab.com/ee/user/permissions.html#job-permissions

[46] GitLab. 2024. *Use CI/CD configuration from other files.* https://docs.gitlab.com/ee/ci/yaml/includes.html

[47] GitLab.org. 2024. *GitLab.* https://gitlab.com/gitlab-org/gitlab

[48] Yacong Gu, Lingyun Ying, Huajun Chai, Chu Qiao, Haixin Duan, and Xing Gao. 2023. Continuous Intrusion: Characterizing the Security of Continuous Integration Services. In *2023 IEEE Symposium on Security and Privacy.*

[49] Yacong Gu, Lingyun Ying, Yingyuan Pu, Xiao Hu, Huajun Chai, Ruimin Wang, Xing Gao, and Haixin Duan. 2023. Investigating Package Related Security Threats in Software Registries. In *2023 IEEE Symposium on Security and Privacy.*

[50] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. 2023. An Empirical Study of Malicious Code In PyPI Ecosystem. In *2023 IEEE/ACM International Conference on Automated Software Engineering.*

[51] HackerOne. 2024. *HackerOne | #1 Trusted Security Platform and Hacker Program.* https://www.hackerone.com

[52] JI Hejderup. 2015. *In Dependencies We Trust: How Vulnerable Are Dependencies in Software Modules?*

[53] Open Source Insights. 2024. *Open Source Insights Understand your dependencies.* https://deps.dev//

[54] JanDeDobbeleer. 2024. oh-my-posh. https://github.com/JanDeDobbeleer/oh-my-posh

[55] Panagiotis Kintis, Najmeh Miramirkhani, Charles Lever, Yizheng Chen, Rosa Romero-Gómez, Nikolaos Pitropakis, Nick Nikiforakis, and Manos Antonakakis. 2017. Hiding in Plain Sight: A Longitudinal Study of Combosquatting Abuse. In *2017 ACM SIGSAC Conference on Computer and Communications Security.*

[56] Igibek Koishybayev, Aleksandr Nahapetyan, Raima Zachariah, Siddharth Muralee, Bradley Reaves, Alexandros Kapravelos, and Aravind Machiry. 2022. Characterizing the Security of Github CI Workflows. In *2022 USENIX Security Symposium.*

[57] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2023. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In *2023 IEEE Symposium on Security and Privacy.*

[58] Ravie Lakshmanan. 2021. Malicious NPM Libraries Caught Installing Password Stealer and Ransomware. https://thehackernews.com/2021/10/malicious-npm-libraries-caught.html

[59] Zhi Li, Weijie Liu, Hongbo Chen, XiaoFeng Wang, Xiaojing Liao, Luyi Xing, Mingming Zha, Hai Jin, and Deqing Zou. 2022. Robbery on DevOps: Understanding and Mitigating Illicit Cryptomining on Continuous Integration Service Platforms. In *2022 IEEE Symposium on Security and Privacy.*

[60] Wentao Liang, Xiang Ling, Jingzheng Wu, Tianyue Luo, and Yanjun Wu. 2023. A Needle is an Outlier in a Haystack: Hunting Malicious PyPI Packages with Code Clustering. In *2023 IEEE/ACM International Conference on Automated Software Engineering.*

[61] Guannan Liu, Xing Gao, Haining Wang, and Kun Sun. 2022. Exploring the Unchartered Space of Container Registry Typosquatting. In *2022 USENIX Security Symposium.*

[62] Michael Meli, Matthew R McNiece, and Bradley Reaves. 2019. How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories. In *2019 Network and Distributed System Security Symposium.*

[63] Microsoft. 2024. *Developer Community.* https://developercommunity.visualstudio.com

[64] Siddharth Muralee, Igibek Koishybayev, Aleksandr Nahapetyan, Greg Tystahl, Brad Reaves, Antonio Bianchi, William Enck, Alexandros Kapravelos, and Aravind Machiry. 2023. ARGUS: A Framework for Staged Static Taint Analysis of GitHub Workflows and Actions. In *2023 USENIX Security Symposium.*

[65] npm Docs. 2022. *npm-unpublish.* https://docs.npmjs.com/cli/v8/commands/npm-unpublish

[66] Christina Paule, Thomas F Düllmann, and André Van Hoorn. 2019. Vulnerabilities in Continuous Delivery Pipelines? A Case Study. In *2019 IEEE International Conference on Software Architecture Companion.*

[67] Azure Pipelines. 2024. *Azure Pipelines Extensions.* https://marketplace.visualstudio.com/search?target=AzureDevOps&category=Azure%20Pipelines

[68] Azure Pipelines. 2024. *Azure Pipelines Templates.* https://learn.microsoft.com/en-us/azure/devops/pipelines/process/templates?view=azure-devops&pivots=templates-includes#use-other-repositories

[69] Tom Preston-Werner. 2023. Semantic Versioning 2.0.0. https://semver.org/

[70] GH Archive Project. 2023. *GH Archive.* https://www.gharchive.org/

[71] Mitmproxy Project. 2024. mitmproxy - an interactive HTTPS proxy. https://mitmproxy.org/

[72] pytorch. 2024. *pytorch.* https://github.com/pytorch/pytorch

[73] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In *2019 IEEE/ACM International Conference on Software Engineering.*

[74] Aakanksha Saha, Tamara Denning, Vivek Srikumar, and Sneha Kumar Kasera. 2020. Secrets in Source Code: Reducing False Positives using Machine Learning. In *2020 International Conference on COMmunication Systems & NETworkS.*

[75] scala-steward action. 2024. *scala-steward-org/scala-steward-action.* https://github.com/scala-steward-org/scala-steward-action

[76] Adriana Sejfia and Max Schäfer. 2022. Practical Automated Detection of Malicious npm Packages. In *2022 International Conference on Software Engineering.*

[77] SHAttered. 2017. *SHAttered.* https://shattered.io/

[78] Vibha Singhal Sinha, Diptikalyan Saha, Pankaj Dhoolia, Rohan Padhye, and Senthil Mani. 2015. Detecting and Mitigating Secret-Key Leaks in Source Code Repositories. In *2015 IEEE/ACM Working Conference on Mining Software Repositories.*

[79] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *2018 USENIX Security Symposium.*

[80] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2018. SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS. In *2018 Network and Distributed System Security Symposium.*

[81] Donald Stufft. 2015. *Closing the Delete File + Re-upload File Loophole.* https://mail.python.org/pipermail/distutils-sig/2015-January/025683.html

[82] Carmine Vassallo, Sebastian Proksch, Harald C Gall, and Massimiliano Di Penta. 2019. Automated Reporting of Anti-Patterns and Decay in Continuous Integration. In *2019 IEEE/ACM International Conference on Software Engineering.*

[83] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C Gall, and Massimiliano Di Penta. 2020. Configuration Smells in Continuous Delivery Pipelines: A Linter and a Six-Month Study on GitLab. In *2020 Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.*

[84] Duc-Ly Vu, Zachary Newman, and John Speed Meyers. 2023. Bad Snakes: Understanding and Improving Python Package Index Malware Scanning. In *2023 IEEE/ACM International Conference on Software Engineering.*

[85] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. 2020. An Empirical Characterization of Bad Practices in Continuous Integration. *2020 Empirical Software Engineering* (2020).

[86] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *2019 USENIX Security Symposium.*