Continuous Intrusion: Characterizing the Security of Continuous Integration Services

Yacong Gu*, Lingyun Ying*✉, Huajun Chai*, Chu Qiao†, Haixin Duan‡§, Xing Gao†
*QI-ANXIN Technology Research Institute, †University of Delaware,
‡Tsinghua University, §Tsinghua University-QI-ANXIN Group JCNS
*{guyacong, yinglingyun, chaihuajun}@qianxin.com, †{qiaochu, xgao}@udel.edu, ‡§duanhx@tsinghua.edu.cn

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179471

***Abstract*—Continuous Integration (CI) is a widely-adopted software development practice for automated code integration. A typical CI workflow involves multiple independent stakeholders, including code hosting platforms (CHPs), CI platforms (CPs), and third party services. While CI can significantly improve development efficiency, unfortunately, it also exposes new attack surfaces. As the code executed by a CI task may come from a less-trusted user, improperly configured CI with weak isolation mechanisms might enable attackers to inject malicious code into victim software by triggering a CI task. Also, one insecure stakeholder can potentially affect the whole process. In this paper, we systematically study potential security threats in CI workflows with multiple stakeholders and major CP components considered. We design and develop an analysis tool, CInspector, to investigate potential vulnerabilities in seven popular CPs, when integrated with three mainstream CHPs. We find that all CPs have the risk of token leakage caused by improper resource sharing and isolation, and many of them utilize over-privileged tokens with improper validity periods. We further reveal four novel attack vectors that allow attackers to escalate their privileges and stealthy inject malicious code by executing a piece of code in a CI task. To understand the potential impact, we conduct a large-scale measurement on the three mainstream CHPs, scrutinizing over 1.69 million repositories. Our quantitative analysis demonstrates that some very popular repositories and large organizations are affected by these attacks. We have duly reported the identified vulnerabilities to CPs and received positive responses.**

1. Introduction

Continuous Integration (CI) is a software development practice of automatically integrating code changes with automated build and test. CI can significantly improve software development efficiency while reducing costs, and has been considered as one of the most effective ways to promptly deliver continuous improvements to customers [1]. Therefore, CI services have gained popularity among developers: it is reported that 36% of developers have applied CI services to automatically build and test their code changes in 2022 [2].

Unfortunately, CI also exposes new attack surfaces, as it needs to execute user-defined code (potentially less trusted) in the build and test phases of CI workflows. If CI services are improperly configured without strong isolation enforced, attackers can intentionally inject malicious code into victim software by submitting a pull request and triggering a CI task. As one of the top five challenges in software supply chain security [3], many CI security incidents [4] have happened in recent years. For example, attackers have created a backdoor in the official *PHP* interpreter artifact by exploiting a malicious commit to the *PHP*'s repository, which might trigger a CI workflow that performs an automated deployment [5]. Moreover, CI workflows involve multiple independent stakeholders, and one insecure stakeholder can potentially affect the whole process. It is reported that attackers utilize unknown methods to steal GitHub OAuth tokens on two popular CI platforms (CPs), TravisCI [6] and Heroku [7], and further abuse these tokens to steal source code from multiple organizations' private repositories [8].

In this paper, we systematically study potential security threats in CI workflows, covering three primary stakeholders, i.e., CPs, Code Hosting Platforms (CHPs), and Third Party Services (TPSs). We consider major CP components (e.g., *controller*, *runner*, and *executor*) and extract a basic security model and principles by reading documents and analyzing the source code. We identify a series of security risks in CI workflows that could potentially cause serious consequences, such as privilege escalation. Specifically, as resource sharing is quite common in CI services (e.g., many CPs support running different repositories on the same CI runner), the weak isolation between CP components may allow users with lower authority to access non-authorized resources. Meanwhile, we find that tokens are largely utilized for authorization among stakeholders. If token accessibility is not properly configured, token leakage might occur since the code executed by a CI task may come from a malicious developer [9]. Even worse, over-privileged tokens can potentially enable attackers to access unauthorized resources (e.g., the source code of private repositories). Finally, tokens with improper validity periods can further amplify various security threats.

We design and develop an analysis tool, CInspector, to investigate potential vulnerabilities in existing CPs. CInspector dynamically collects a wide-spectrum of networking data (e.g., traffic generated in the runner machine) and system information (e.g., process and socket port information), and further adopts a two-stage analysis. In the online stage, CInspector performs automated tests using dedicated pre-designed test suites on suspected tokens to identify their

✉Lingyun Ying is the corresponding author.

DOI 10.1109/SP46215.2023.00149

permissions and validity periods. Then, CInspector performs token pedigree analysis based on the network traffic and further identifies token accessibility. Finally, CInspector conducts rule-based threat analysis to uncover the potential security risks (e.g., token leakage).

We analyze seven popular CPs, including three CHP built-in CPs (i.e., GitHub Actions, GitLab CI, and Bitbucket Pipelines) and four independent CPs (i.e., CircleCI, TravisCI, TeamCity, and Jenkins), with three mainstream CHPs (i.e., GitHub, GitLab, and Bitbucket). We identify four types of problems that may cause token leakage. Particularly, we find that tokens can be leaked from command-line information, environment variables, plaintext files with improper permissions, and memory that other processes can inspect. All CPs have the risk of token leakage, and many of them have the problem of over-privileged tokens. Also, improperly long-lived tokens widely exist: some tokens even do not expire after two weeks, potentially allowing attackers (e.g., a fired employee) to access resources after the CI task is finished.

Based on the identified risks, we further reveal four novel attack vectors that allow attackers who can execute code in CI tasks to escalate their privileges and stealthily inject malicious code. Specifically, attackers can utilize a stolen token to register an alternative runner and hijack all following CI tasks. For some CPs, their over-privileged tokens used for code cloning from GitHub can be easily stolen by attackers (without any permissions) to obtain read, write, and admin permissions of all repositories managed by the repository owner. We also find that some leaked tokens enable attackers to arbitrarily tamper with the CI task results, which can bypass existing code-checking mechanisms. Finally, some CPs improperly utilize TPS's long-term access token, allowing attackers to modify artifacts of the repository.

To evaluate the potential impact of our disclosed attacks, we conduct a large-scale measurement on the three mainstream CHPs. We collect 1.69 million repositories that are integrated with at least one of the seven CPs. We find that some very popular repositories and many large organizations are affected by these attacks. For example, 7 out of the top 100 stars repositories on GitHub, and multiple repositories of Google, Microsoft, NVIDIA, and Apache use the self-hosted runner and are vulnerable to the task hijacking attack. We further reveal that 8,464 `secrets` in 2,472 vulnerable repositories are in danger: adversaries can exploit these `secrets` to launch advanced attacks, such as covertly injecting malicious code into many popular packages that are used by millions of developers.

Finally, we discuss several defense practices that CPs should adopt to mitigate these attacks. We have disclosed our findings to all affected service providers and received positive feedback. For example, the TeamCity team have confirmed the reported vulnerabilities and marked two of them as 'show-stopper' (the highest level). The GitHub, GitLab, Bitbucket, Jenkins, and CircleCI teams have also acknowledged our reported issues and the Bitbucket team have awarded us with a $1,200 bug bounty.

In summary, the major contributions of this work include:

- We systematically study potential security threats in CI workflows and identify several risks involving the authorization process and resource isolation among multiple stakeholders.
- We design and develop a fine-grained analysis tool, CInspector, and analyze seven popular CPs using it. We find that all seven CPs suffer from several security risks, potentially causing serious consequences.
- We reveal four new attack vectors allowing attackers to steal source code from repositories without any access permission, arbitrarily tamper with CI task results, and maliciously modify the artifacts generated in CI workflows.
- We conduct a large-scale measurement on three mainstream CHPs, demonstrating that the above attacks could pose severe security threats to the open source community.
- We have disclosed all our findings to the corresponding service providers and received positive feedback.

2. Background

2.1. Primary Stakeholders

Code Hosting Platforms (CHPs) are responsible for storing repositories' source code. Popular CHPs (e.g., GitHub, GitLab, and Bitbucket) generally adopt role-based access control to their repositories. There are three types of roles: *owner*, *collaborator*, and *reporter*. Repository owners have full access to their repositories: they can control the permissions of repository members and provide authorization to third party services (TPSs). Repository collaborators can have read (e.g., browse, clone, and create pull requests) and write (e.g., edit source code and merge pull requests) permissions on the repository. However, they cannot edit settings or provide authorization to any TPSs. Reporters (e.g., software testers) only have read permissions for the source code but not write permissions.

CI Platforms (CPs) are responsible for executing a repository's CI tasks. A CP can be a part of a CHP (e.g., GitHub Actions) or independent of CHPs (e.g., CircleCI).

Repository owners first need to grant a CP permission and configure CI tasks via a configuration file (e.g., `.travis.yml` in TravisCI) or web portal (e.g., TeamCity). Generally, the CI task configuration contains: (1) *Execution Triggers* specify when or which events should trigger the execution of a CI task. For example, based on the configuration, a new push event or pull request can trigger a CI task execution. (2) *Runner Configuration* indicates the running environment (e.g., the OS version) required by a CI task. (3) *Jobs and Steps*. Each CI task is a collection of jobs which can be executed sequentially or in parallel. Each job is a sequence of steps, and a step can execute one or more commands (e.g., run test code or build artifact). (4) *Secrets* can be used to access TPSs during the CI task execution. Repository owners can define secrets as key-value pairs on the CP web portal and use keys instead of values in the CI task configuration to avoid exposing secrets. (5) *Artifacts* are files generated during the execution of a CI task. Mainstream CPs support three categories of artifacts: package release files, intermediate files passed between multiple jobs of the same CI task, and test-related files such as test reports and

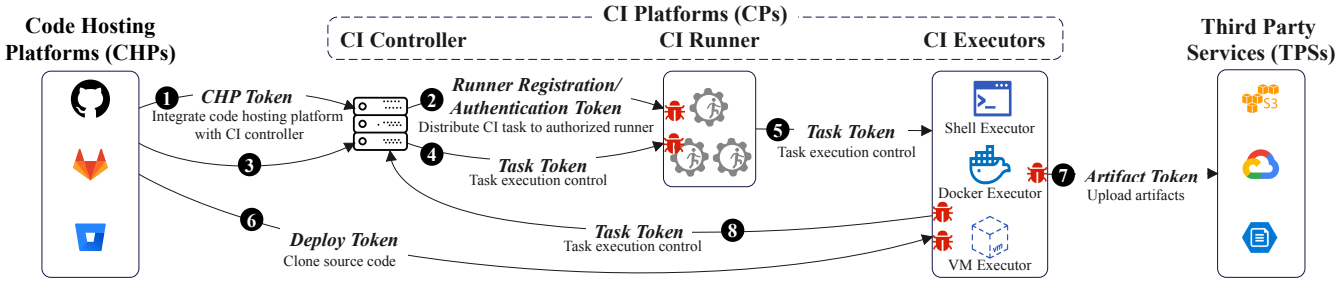

Figure 1: Overview of CI workflow.

screenshots. For example, developers can use artifacts to share data between jobs in GitHub Actions.

Once the configuration is completed, any defined events will trigger the CP to execute a CI task. The execution of a CI task involves three components: *CI controller*, *CI runner*, and *CI executor*. *Controller* is the core of CP. It is responsible for processing authorization with CHPs, parsing the CI task configuration, receiving execution trigger events, and scheduling and distributing CI tasks. The controller will also generate detailed execution logs in real time and display the task execution result (i.e., success or failure).

Runner is an agent installed on a different host machine from the controller. The runner must be registered in the controller to receive instructions. Runners can run on servers provided by CPs (called CP-hosted runners) or on an organization's own machines (called self-hosted runners), which enable the organization to customize the environment and save costs [10].

Finally, *executor* is essentially an environment where CI tasks will be executed. For example, GitLab CI allows users to specify different types of executors, such as shell executors and Docker executors. If the Docker executor is used, the runner will start a Docker container when it receives a new CI task, and then execute the task in the Docker container.

Third Party Services (TPSs) such as cloud services might also be involved during executing CI tasks. For example, CircleCI stores the artifacts generated by CI tasks in AWS S3 [11].

2.2. CI Workflow

Figure 1 shows a typical CI workflow. As multiple stakeholders are involved in the process, tokens are largely utilized for authorization. First, the repository owner grants the CI controller with CHP access (step ❶). A *CHP token* (T_{chp}) is released to allow the CP to access source code in the repository and receive trigger events (e.g., pull requests). Then, a CI runner will be registered in the controller (step ❷), with a *runner registration token* ($T_{register}$) released. Particularly, in the self-hosted runner mode, the controller will generate a unique and permanent registration token for each runner. In some CPs, after a runner is successfully registered, the controller will further provide a runner authentication token to the runner for authenticating subsequent communication. We treat both tokens as the $T_{register}$.

Next, when execution trigger events occur in the repository (step ❸), the CHP will notify the controller to distribute a task to an appropriate runner based on the CI task configuration (step ❹). The CP utilizes a *task token* (T_{task}) for the authorization between runner and controller. This token will also be used for updating task execution logs and returning task results. Particularly, a runner can execute multiple CI tasks in parallel. To avoid interference among different tasks, CPs generally generate an independent task token for each CI task execution.

When a runner receives a CI task, it starts an executor to execute the task (step ❺). CI task execution requires cloning the source code of a repository from the CHP to an executor. For non-public repositories, a *code deploy token* (T_{deploy}) is provided by the CHP for authorizing the cloning process. The executor uses the T_{deploy} to clone source code from the CHP, and then executes jobs in the task (step ❻). During the execution (step ❼), the executor may interact with TPSs (e.g., upload artifacts to cloud storage services). This step needs authentication and authorization with TPSs, which involves an *artifact token* ($T_{artifact}$). Finally, as step ❽, after the CI task is finished, the executor (or the runner for some special cases) returns the task result to the controller.

3. CI Security Model and Principles

In a typical CI scenario, an organization can utilize CHPs to maintain multiple repositories, and then maintain several servers (possibly self-hosted) for operating the CI controller and runners/executors. As employees are typically granted different permissions and authorization levels (to repositories), CPs need to carefully set the resource sharing strategy among the controller, runners, and executors, with resources properly isolated. Otherwise, a user with lower authority could break the isolation to access non-authorized resources (e.g., repositories).

Threat Model. In the above scenario, we assume the controller is properly separated from runners/executors (e.g., running in different machines), as explicitly stated by many CPs for safety concerns [12]. CPs are set with default or widely adopted configurations. Besides, all stakeholders (e.g., CHPs, CPs, and TPSs) and the communication between/inside them are secured.

Overall, we consider adversaries as normal users with low authorization levels or limited permissions. The goal

of adversaries is to gain unauthorized access to resources (e.g., escalating their privileges to read private repository's source code) or to distribute malicious code stealthily (e.g., injecting backdoors into artifacts built by CI workflows). They follow the CI execution flow and can execute a piece of code in a CI task, which involves running an executor process, to steal weak-isolated tokens with high permissions. Specifically, they can either modify the CI configuration files (if they are not protected) or modify the codebase such as unit test code used in a CI task (which is easy as most CPs have no limitations on this).

In particular, we describe three different types of adversaries and their capabilities as follows:

- **TYPE I: Pull request initiator for open source repository.** Pull request is the primary collaborative way for developers to contribute open source repositories. Some CI tasks can be configured to automatically start running after receiving a new pull request. Recent research [9] shows that 44.4% of the public repositories on GitHub, which use GitHub Actions, are set to trigger a CI task when a new `pull_request` event occurs. Adversaries can then exploit this feature to execute their malicious code in the CI task of the victim repository by submitting a pull request. Note that GitHub has disabled the execution of new GitHub Actions created by first-time contributors without the approval from a repository owner [13]. However, adversaries can gain the trust of the repository owner by first submitting a valid pull request and later submitting a malicious pull request. Note that this type of adversary does not require the corresponding code changes of the pull request to be merged into the victim repository. In many cases (based on the configuration), CPs may run the pull request's CI task before the merging. For example, GitHub provides a feature [14] that makes a pull request mergeable only after it passes CI checks, which can be used to mount the attack.
- **TYPE II: Evil repository collaborator.** Adversaries can be collaborators of one repository in the victim organization. They only have limited access (e.g., read and write) to the repository. However, they are not authorized to change the repository settings, nor manipulate software releases like artifacts. Also, they do not have any permissions on other repositories of the victim organization. In this case, adversaries can introduce the malicious code in a CI task execution by modifying the authorized repository's CI build script or testing code.
- **TYPE III: Evil repository owner.** Adversaries are the owner of one repository in the victim organization and have full access to the repository. They can introduce malicious code in the execution of CI tasks by using carefully crafted CI build scripts or testing code. However, they do not have any permissions on other repositories of the victim organization.

Based on our threat model, we introduce several potential security threats in the existing CI workflows. While there are no unified public standards, we extract a basic security model and principles by reading documents and analyzing the source code of relevant open-source components.

3.1. Weak Resource Isolation

An executor is started by a CI runner; thus both components usually run on the same host machine. Existing CPs support running different repositories on the same runner. In this case, if the runner is not properly isolated from the executor, the task of one repository could attack other repositories (e.g., stealing tokens). The isolation should prevent the executor from accessing any sensitive data from the runner (e.g., $T_{register}$) nor affecting the execution of the runner (e.g., kill the runner process).

Besides, some CPs adopt a *continuous* runner reuse strategy: after executing a task, the runner will be reused for the next one. Under this mode, any sensitive data (e.g., tokens) used by the previous task must be cleared. Otherwise, it might be leaked to the subsequent CI tasks, which might belong to a different repository.

If there is no isolation (e.g., both components run on the same system with root privileges), attackers can control one executor to access all resources on the system. They can also control other processes through syscalls like `ptrace` [15]. Thus, existing CPs adopt different runner/executor isolation strategies, such as running them on different virtual machines (VM), Linux containers (namespaces), and native processes. At the process level, two components run under the same non-root namespace and have limited access to the file system. They can still exchange data through various inter-process communication methods. One component can even kill the other component's process. Thus, our threat model considers that the process isolation is not enough. At the container (namespace) level, two components run under different Linux namespaces. For example, the runner runs in the host system, the executor runs in a non-privileged Docker container; or both components run as different Linux users. Even with this isolation, containers should still be carefully configured. For example, if the container runs in the privileged mode, it can be considered as no isolation since Docker has full access to the host system. In addition, data exchange can be performed by setting environment variables for Docker, which might also cause information leakage.

3.2. Improper Code Hosting Platform Token

T_{chp} allows the CP to access source code in the repository and receive trigger events. Improper usages of the T_{chp} can leak CHP data to attackers. Generally, there are two types of T_{chp}: (1) Personal access credentials, which inherit permissions from the CHP user and can access resources on behalf of the user. Examples include username/password from CHPs and Personal Access Token (PAT), used by GitHub [16] and GitLab [17]. (2) Authorization access credentials, issued by a CHP specified for authorization. Repository owners can limit the permissions and lifetimes of these credentials. Authorization credentials are usually implemented based on the OAuth protocol. All the three mainstream CHPs support such credentials for authorization.

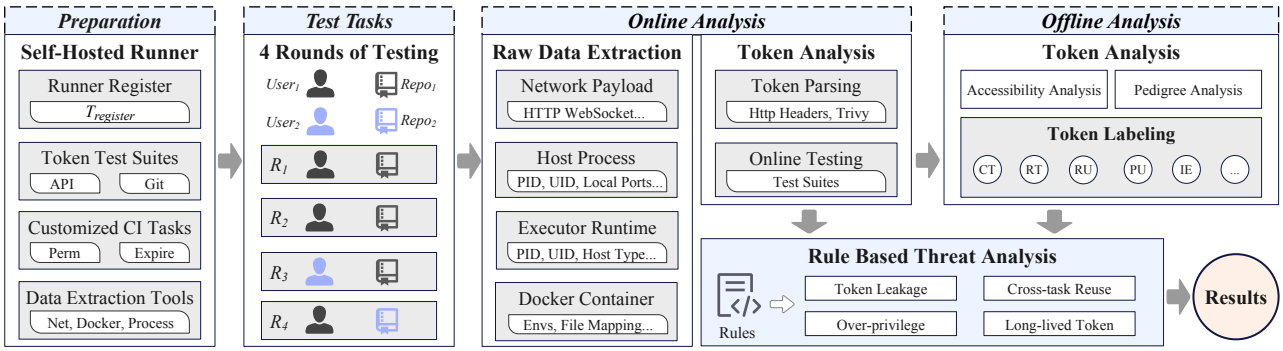

Figure 2: Workflow of CInspector.

Generally, the authorization access credentials are more suitable for CHPs to provide authorization to CPs than personal access credentials. Due to the broad range of permissions that personal access credentials have, using these credentials to authorize CPs is considered a security risk. For example, GitHub's official security guide clearly states that authorizing GitHub Actions "should never use personal access tokens" [18].

Furthermore, T_{chp} must be properly stored and should never be passed to the runner/executor to avoid token leakage. Otherwise, executors running by adversaries may obtain this token and further access repositories in CHPs. For example, it is reported that attackers successfully steal OAuth tokens of TravisCI and Heroku (obtained from GitHub) using some unknown methods and further steal the code of multiple organizations' private repositories hosted on GitHub [8].

3.3. Improper Token Permissions

The accessibility and permissions of all tokens should be carefully configured and limited to avoid potential privilege escalation. For example, T_{task} is used for the authorization between the runner/executor and controller. Since the code executed by the CI task may come from a malicious developer [9], T_{task} should be limited to the originated repository, without access to other repositories (e.g., read the source code of other repositories). Similarly, permissions should be properly configured for T_{deploy}.

For the permission of $T_{artifact}$, this token should not support modifying the artifact after creation. For example, both GitHub and GitLab only provide interfaces for downloading and deleting artifacts, but not for modifying [19] [20]. Also, $T_{artifact}$ should only have access to the artifacts generated by the current task, without permissions for other tasks. Otherwise, attackers who obtained this token might be able to access and modify unauthorized artifacts.

3.4. Improper Token Validity Period

Many tokens, including T_{task}, T_{deploy}, and $T_{artifact}$, should be one-time used only (e.g., expire immediately after the corresponding task ends). Tokens with unnecessarily long validity periods can cause/amplify many security threats, especially if tokens are leaked. For example, both GitHub Actions and GitLab CI expire T_{task} immediately after the CI task ends to improve security [21] [22].

Particularly, similar rules apply for T_{deploy}, which is used by the CHP for authorizing the cloning process on CPs. With a long-lived T_{deploy}, an immoral employee who has been fired might still be able to access the source code of the organization's private repositories. Generally, there are three types of common T_{deploy} [23]: (1) T_{chp}. Executors use the T_{chp} directly. (2) One-time token. Based on T_{chp}, the controller uses the CHP APIs to generate a one-time token for code clone (e.g., GitHub's server-to-server tokens [23]). (3) Private key. The controller generates a pair of public and private keys. The public key will be added to a repository of the CHP, and the private key can be used as a code deploy token. Among them, both T_{chp} and private key are long-lived by default. Therefore, if CPs use these two types of T_{deploy} without properly setting the validity period, they are prone to security breaches.

4. Methodology

To investigate the potential vulnerabilities in CI services, we design an analysis tool, CInspector, to analyze popular CPs and their interaction with related stakeholders (e.g., CHPs and TPSs). The overall workflow of CInspector is illustrated in Figure 2. CInspector conducts black-box testing on CPs using a self-hosted runner (or an on-premise deployed runner) deployed on our own machine. During the execution of the CI task, CInspector records the network traffic generated in the runner machine and collects a wide-spectrum of system data (e.g., process and socket port information). The analysis process further consists of online and offline stages. In the online analysis stage, CInspector extracts suspected tokens from network traffic. It uses pre-designed test suites to perform automated tests on tokens to identify their permissions and validity periods. In the offline analysis stage, CInspector performs token pedigree analysis based on the network traffic. It further performs token accessibility analysis to identify whether a token can be acquired by potential adversaries (e.g., token leakage). Finally, according to a series of rules, CInspector outputs the potential security risks. This section details all components of CInspector. We also discuss ethical concerns.

4.1. Target Platforms

We choose three CHPs, namely GitHub, GitLab, and Bitbucket, which dominate the CHP market and are supported by all mainstream independent CPs [24]. Following several recent researches [9] [25], we select seven CPs with high market share [24], including three CHP built-in CPs (i.e., GitHub Actions, GitLab CI, and Bitbucket Pipelines), and four popular independent CPs (i.e., CircleCI, TravisCI, TeamCity, and Jenkins).

We use a self-hosted runner for all CPs except for TravisCI. TravisCI does not support the self-hosting mode, so we run an on-premise deployment runner (deploying all components in our own machines)[1]. We also deploy the runner and executor on Linux, which is adopted by most CPs. For executor, as most CPs utilize Docker container or shell as the built-in executor, we thus focus on these two types of executors (if supported) for testing. Particularly, if a CP's built-in executor has root privileges, we also grant root privileges to the executor in the self-hosted runner.

4.2. Preparation

We attempt to simulate users with different privileges and test potential misconfigured token permissions.

Repositories. We first create an organization on each of the three CHPs. Within the organization, we further create two private repositories ($Repo_1$ and $Repo_2$) and two users ($User_1$ and $User_2$). $User_1$ is the owner of both repositories. $User_2$ is set as a collaborator of $Repo_1$ with low-level authority (as mentioned in Section 2.1), but has no permission on $Repo_2$.

This setup allows us to test whether $User_2$, as a potential attacker, can access non-authorized resources. It also allows us to examine over-privileged tokens (e.g., whether or not tokens in CI tasks initiated by $User_1$ and $User_2$ are the same).

Runner Registration. We register the self-hosted runner and record the registration token generated by each controller. For CPs supporting multiple repositories to share the same runner, we register both repositories on the same runner.

Data Extraction Tools. We use the tool mitmproxy [26] on the runner host to monitor network traffic and decrypt SSL encrypted traffic by installing a custom certificate. We develop a Linux shell script to regularly collect data such as the UID and PID of the process and the socket port opened by a process. We also use the Docker inspect tool [27] to regularly collect the configuration of running Docker containers, including environment variables and directory mapping. These data are then fed for token analysis.

Customized CI Tasks. We instrument the CI build script and the testing code to output the executor process' UID, PID, and environment variables. CInspector also tests if the process has root privileges and whether the process is inside a Docker container.

Token Test Suites. We design several token test suites to test each token's permissions for accessing relevant resources, as shown in Table 1. For each token, the test covers accessing user profiles, reading/writing both repositories, modifying the repository's settings, reading/writing artifacts, etc. We utilize APIs [28] [29] [30] [31] provided by CHPs, CPs, and TPSs for implementing the test suites. Particularly, after a CI task ends, CInspector further runs the test suites several times at different intervals to test tokens' validity periods.

TABLE 1: Overview of the token test suites.

Targets	Test Operations
Repos	READ, WRITE to the source code of $Repo_1$ and $Repo_2$. READ, WRITE, LIST, DELETE the artifacts of both Repos. WRITE to the settings of $Repo_1$ and $Repo_2$.
Controller	READ, WRITE to the execution logs of the current CI task. WRITE to the execution result of current CI task.
Users	READ, WRITE to user profiles of $User_1$ and $User_2$.

4.3. Online Analysis

We manually conduct 4 rounds of testing for all valid combinations of CHP and CP. Round 1, with $User_1$ as the initiator, we run a CI task of $Repo_1$. Round 2, after 1 hour, we run the CI task of $Repo_1$ again with $User_1$ as the initiator. Round 3, after 1 hour again, we run the CI task of $Repo_1$, but with $User_2$ as the initiator. Round 4, we run a CI task of $Repo_2$ with $User_1$ as the initiator.

In each round of testing, CInspector automatically analyzes the decrypted network traffic in real time. We extract the token from the HTTP `Authorization` header and adopt the open source tool Trivy [32] to extract useful data in the payload (e.g., tokens, public/private keys). For each extracted token, CInspector run the token test suites with both repositories' resources and record the results accordingly.

4.4. Offline Analysis

CInspector maps the extracted tokens with the corresponding processes through the Linux socket port and then conducts offline analysis. Note that a token can be associated with multiple processes since it might be transmitted across processes and containers.

Token Accessibility Analysis. CInspector conducts token accessibility analysis to infer whether or not a token is visible to an executor process based on the following rules: (1) processes in non-privileged Docker containers should not access processes in the host; (2) processes in privileged Docker containers can access processes in the host; (3) processes with root privileges can access any process; and (4) a process can access other processes under the same UID but cannot access other processes under different UIDs.

Token Pedigree Analysis. In some HTTP request-response pairs, there is a unique *token* in the requests, but with one or multiple new $token'$s in the corresponding responses. For example, CircleCI sends an HTTP request[2] with T_{task}, and the HTTP response message includes T_{deploy}. CInspector considers that these new tokens are derived from the *token*. During token accessibility analysis, if a token is visible to the executor process, CInspector also treats all its derived tokens as visible.

[1]We collectively refer to them as self-hosted runners in this paper.

[2]The URL is https://runner.circleci.com/api/v2/task/config.

chp(t) :- t matches one of T_{chps}'s formats

equal($t_1, t_2 \ldots t_n$) :- $t_1 \ldots t_n$'s values pairwise equal

nequal($t_1, t_2 \ldots t_n$) :- $t_1 \ldots t_n$'s values pairwise unequal

used(t, s) :- t is used in s scenario

label(t, $l_1 \ldots l_n$) :- t is labeled with $l_1 \ldots l_n$

expire(t, m) :- t expires after m minutes from the task ends

access(t, r) :- t can access repository r

Figure 3: Token analysis predicates.

CHP Token(CT): label(t, CT) $\leftrightarrow$ chp(t)

Runner Registration Token(RT): label(t, RT) $\leftrightarrow$ equal(t, $\forall\ t' \in \{T_{register}$ recorded$\}$)

Task Token(TT): label(t, TT) $\leftrightarrow$ used(t, 'when sending the task results')

Deploy Token(DT): label(t, DT) $\leftrightarrow$ used(t, 'when cloning the source code')

Artifact Token(AT): label(t, AT) $\leftrightarrow$ used(t, 'when uploading the artifacts')

Runner Unique(RU): label(t, RU) $\leftrightarrow$ equal($t_{R_1}, t_{R_2}, t_{R_3}, t_{R_4}$)

Task Unique(TU): label(t, TU) $\leftrightarrow$ nequal($t_{R_1}, t_{R_2}, t_{R_3}, t_{R_4}$)

Repository Unique(PU): label(t, PU) $\leftrightarrow$ equal(t_{R_1}, t_{R_2}), nequal(t_{R_1}, t_{R_4})

User Unique(UU): label(t, UU) $\leftrightarrow$ equal($t_{R_1}, t_{R_2}, t_{R_4}$), nequal($t_{R_1}, t_{R_3}$)

Immediately Expire(IE): label(t, IE) $\leftrightarrow$ expire(t, 0)

(t_{R_n} indicates a token's value in the round n of the test task.)

Figure 4: Token label rules.

$$\frac{\text{label}(t,\ CT)}{\text{warn}(t)}$$

$$\frac{l \in \{RT, TT, DT, AT\}\,, \text{label}(t,\ CT,\ l)}{\text{warn}(t)}$$

$$\frac{\text{access}(t, \neg Repo_t),\ l \in \{TT, DT, AT\},\ \text{label}(t,\ l)}{\text{warn}(t)}$$

$$\frac{\text{access}(t, \neg Repo_t),\ l \in \{TU, PU, UU\},\ \text{label}(t,\ l)}{\text{warn}(t)}$$

$$\frac{\forall Repo\ \text{access}(t, Repo),\ \text{label}(t,\ RU)}{\text{warn}(t)}$$

$$\frac{\neg\text{label}(t,\ IE),\ l \in \{TT, DT, AT\},\ \text{label}(t,\ l)}{\text{warn}(t)}$$

Figure 5: Threat analysis rules.

Token Labeling. CInspector labels all tokens during the analysis. Figure 3 shows the base rules for token-related predicates, which are straightforward and self-explained. For example, `chp(t)` means a token t's format is the same as a T_{chp}'s (e.g., all GitHub PATs are initiated with `ghp_`). Then, we label tokens following the rules in Figure 4. For instance, if *chp(t)* is true, the token t is labeled with CT. We first assign a label to each type of token (e.g., DT for T_{deploy}). Then, we test whether the token is unique for each task (TU), user (UU), runner (RU), and repository (PU). For example, a token is labeled as PU (repository unique) if both `equal` (t_{R_1}, t_{R_2}) and `nequal` (t_{R_1}, t_{R_4}) are true, meaning that the tokens used in the first two rounds are the same (for $Repo_1$), while different from the fourth round (for $Repo_2$). We also mark a token as IE if it expires immediately. Note a token may be marked with multiple labels.

4.5. Rule-Based Threat Analysis

We mainly consider three types of threats: (1) token leakage: if a token is visible to executors; (2) over-privileged: if a token has unnecessary permissions that could potentially cause privilege escalation; and (3) improper validity period: if a token persists (remains valid) for a long time.

Based on these threats, we propose several detection rules for different tokens. For the task-unique token, it should expire immediately after the task ends. Also, it should only have access rights to the repository that the task belongs to. For the runner-unique token, it can only be used to generate other tokens (e.g., T_{task}) and should not have the read permissions of the source code of any repositories. As multiple repositories might share tokens if they share the same runner, if a token has read access to one repository, it can also access other repositories. For the repository-unique token, it should only have the access permissions to the corresponding repository. Figure 5 lists the detailed rules for detecting security threats. Take the first rule as an example, if a token t is labeled as CT, it indicates that the T_{chp} is transmitted to the runner/executor; thus, there is a security risk of T_{chp} leakage. As another example, in the third rule, the token t is labeled as one of T_{task} / T_{deploy} / $T_{artifact}$ but has access rights to repositories other than the originated one, indicating a potential risk of privilege escalation.

4.6. Ethical Considerations

We believe there are no or only minor ethical issues in our work. We have never conducted any experiments on third-party repositories/projects. All vulnerability analyses and threat validations are done on our-owned repositories/projects or our locally deployed CPs. Moreover, we conduct all the experiments using our own accounts. The black-box testing of the runner and executor is also conducted on our own self-hosted runner machine. To test the risk of privilege escalation, we also use our own resources and never try to access any resources that do not belong to us. Finally, after the testing, we have timely disclosed all our findings to corresponding stakeholders (details in Section 7.2).

5. Identified Issues in CI Platforms

5.1. Token Leakage

Token leakage is highly related to how a token is used and the isolation strategy between the executor and the runner. We have adopted CInspector to semi-automatic unveil several token-leakage threats as follows. CInspector automatically collects and analyzes token leakage for most potential leakage sinks (e.g., .bash_hisotry file, the configuration file of CI runner, environment variables, and command-line arguments). We then manually analyze memory leakages, as various

TABLE 2: The runner isolation strategy, token leakage, and token reuse problems on each CP. ○ means no isolation. ◑ means process-level isolation. ● means Linux namespace-level isolation. ✓ indicates the executor can access the runner's resources. *C*/*E*/*F*/*M* in the *Token Leakage* column mean the token leakage ways: *C* - Command-line, *E* - Environment Variables, *F* - Files, *M* - Memory.

CPs	Executors	Runner Isolation	Access Runner's				Token Leakage					Token Reuse	
			C	*E*	*F*	*M*	T_{chp}	$T_{register}$	T_{task}	T_{deploy}	$T_{artifact}$	as T_{deploy}	as $T_{artifact}$
GitHub Actions	Shell†	○	✓	✓	✓	✓		*C*, *F*, *M*	*E*, *M*	*E*, *M*	*E*, *M*	T_{task}	T_{task}
GitLab CI	Docker	●		✓					*E*	*E*	*E*	T_{task}	T_{task}
	Shell	◑	✓	✓					*E*	*E*	*E*	T_{task}	T_{task}
Bitbucket Pipelines	Docker	●											
	Shell	○	✓	✓	✓	✓		*C*, *M*	*E*, *M*	*M*	*E*, *M*		
CircleCI	Docker	○*	✓	✓	✓	✓			*F*, *M*	*F*, *M*	*M*		
	Shell	○	✓	✓	✓	✓		*C*, *M*	*F*, *M*	*F*, *M*	*M*		
TravisCI	Docker	●		✓				*E*	*F*	*E*			
TeamCity	Shell	◑	✓	✓	✓	✓	*M*	*F*, *M*	*M*	*F*, *M*	*M*	$T_{chp}^{\ddagger}$	
Jenkins	Docker	◑*	✓	✓	✓	✓	*M*	*C*, *M*	*M*	*M*	*M*	$T_{chp}^{\ddagger}$	
	Shell	◑	✓	✓	✓	✓	*M*	*C*, *M*	*M*	*M*	*M*	$T_{chp}^{\ddagger}$	

† The GitHub will generate a clean VM for each GitHub-hosted runner. However, since the runner and the executor are running within the same VM, we consider the isolation of the GitHub-hosted runner to be the same as the Shell executor.
* In CircleCI's and Jenkins's docker executor mode, the runner and the executor are running in the same container (the former has root privileges).
‡ When integrated with GitHub, TeamCity uses T_{chp} as T_{deploy}. When integrated with Bitbucket and GitLab, Jenkins uses T_{chp} as T_{deploy}.

memory layouts (e.g., native, JVM, and .NET) are adopted by CI runners.

5.1.1. Risks of Token Leakage

Using Token on the Command-line. There are several ways that Linux shells can expose sensitive data. When a process is running, it is easy to get the process information through the command `ps -ef`. For instance, the self-hosted runner of Bitbucket Pipelines takes the $T_{register}$ as an argument to the start command[3]. Even after the process exits, one can still observe this command information in the shell's history file (such as `.bash_history` file). Therefore, these tokens can be leaked to adversaries in a CI task. We find that the self-hosted runner of GitHub Actions suffers from the issue.

Setting Token as Environment Variables. Using environment variable is common for data (e.g., token) transfer, especially to transfer data from the host to a Docker container. However, environment variables can be easily read in multiple ways, and thus tokens can be leaked. For example, in GitLab CI Docker mode, a T_{task} is passed into a Docker container as an environment variable for an executor to use. As another example, to protect the T_{task} (i.e., `ACTIONS_RUNTIME_TOKEN` in GitHub) from being read by testing code, GitHub Actions only passes the token to necessary jobs, via setting environment variables in the self-hosted runner. However, an executor process can directly steal the T_{task} by reading the environment variables of other processes through the `/proc/{pid}/environ` file.

Storing Token in Plaintext File. We find that many CPs store plaintext tokens in files directly, which can be easily obtained by other processes under the same namespace. For example, in the Docker executor mode, CircleCI saves the config file of a CI task inside the executor container[4]. As a result, the executor can directly obtain the T_{task} and the T_{deploy} in the file through a CI task. In GitHub Actions, after a self-hosted runner is registered, the controller will send a $T_{register}$ for subsequent authentication. The runner saves this token in three files[5]. Similarly, the executor (in the shell mode) has permission to read these files, so anyone who controls the executor can steal the $T_{register}$ through a CI task. Finally, we find that TeamCity adopts an interesting strategy. TeamCity stores the T_{deploy} in a temporary file[6] and will delete the file immediately after finishing code cloning. However, since TeamCity's runner runs in *continuous* mode, attackers can implant a monitoring program in the runner machine through a CI task to continuously scan the target directory, and still obtain the token.

Keeping Token in Memory. If an executor runs with root privileges, adversaries can directly read the memory of the runner process through a CI task and search for token strings according to the token format. For instance, both executors of GitHub Actions and CircleCI have root privileges [33] [34]. Thus, adversaries can grab the $T_{register}$ and the T_{task} directly from the runner process's memory.

On the other hand, a process without root privileges cannot read the memory of other processes. However, security risks still exist in some CPs. While the executors of TeamCity, Jenkins, and Bitbucket Pipelines (in shell executor mode) do not have root privileges, their runner processes open JVM Tool Interface (JVM TI) [35]. This interface allows a non-root process to inspect the runner process's memory and further controls the execution of the runner process. JVM TI is enabled by default when JVM is started, and it can only be actively disabled [36]. Therefore, various tokens in the memory of runner processes can be leaked. Even worse, all these CPs allow multiple repositories to share the same runner. Thus, all these repositories' tokens kept in memory are at the risk of being stolen.

[3] `start.sh -OAuthClientId -OAuthClientSecret`
[4] `/.circleci-runner-config.json`
[5] `.runner`, `.credentials`, and `.credentials_rsaparams`
[6] `{runner folder}/temp/buildTmp/pass`

5.1.2. Token Leakage Analysis Results

We find that all seven CPs suffer from token leakage problems, and the detailed results are listed in Table 2.

First, all CPs leak the T_{task}. As the executor frequently uses the T_{task} for various operations (e.g., log sending and task result uploading) during the execution of a CI task, most runners simply transfer the T_{task} to the executor. The only exception is the Docker mode executor of Bitbucket Pipelines (its shell mode executor is still vulnerable to the problem). When Bitbucket Pipelines executes a CI task, it starts an executor Docker container and an auxiliary Docker container (named `atlassian/pipelines-auth-proxy`) to handle the authentication with the controller. The runner passes the T_{task} to the auxiliary container rather than the executor container. Unfortunately, although the above method will not leak the T_{task}, Bitbucket Pipelines still suffers from other potential attacks related to the T_{task} (i.e., *Task Result Hijacking Attack*) in Section 6.3.

Similarly, the T_{deploy} might be leaked. Particularly, we find that the T_{deploy} of CircleCI and TravisCI are in the form of public-private key pair, while others are strings. Both have security risks if permissions and validity periods are not carefully configured (discussed in Section 6.2).

In addition, we also find that shell mode executors can easily leak $T_{register}$ through a file or command-line. The only exception is GitLab CI: although its shell mode executor writes the $T_{register}$ to the configuration file (i.e., `/etc/gitlab-runner/config.toml`), the permission of the file is set to only readable by the root user. Thus, executors running under non-root privileges cannot read the file, avoiding token leakage.

Finally, we have another interesting finding: the T_{chp}, which is supposed to only be used between CHP and the controller for authorization, is leaked in the runners of TeamCity and Jenkins. The cause is that these two CPs reuse the value of the T_{chp} as the T_{deploy}. In the next subsection, we show that the misuse of different types of tokens can cause severe security issues.

5.2. Token Misuse

CInspector also shows that the misuse of tokens widely exists in all CPs. Some misuses can cause privilege escalation (i.e., over-privileged tokens), while others grant tokens with improperly long validity periods. We have manually confirmed the potential security risks. For example, after CInspector discovering a long-lived token, we determine its accurate expiration time by reading related documents or analyzing network traffic.

5.2.1. Over-privileged Token

We find that the T_{deploy} of multiple CPs have over-privilege problems. The analysis results are shown in Table 3. In our experiment, the T_{deploy} extracted from the CI task of $Repo_1$ should only have read permission of $Repo_1$, as described in Section 4.2. However, TeamCity and Jenkins have over-privileged T_{deploy} when integrated with some CHPs (i.e., GitHub and GitLab). For example, when TeamCity is integrated with GitHub/GitLab, the T_{deploy} of $Repo_1$ has the read, write, and admin (modify repository settings) permissions of both $Repo_1$ and $Repo_2$. With GitLab, the T_{deploy} of TeamCity can even read user profile of $User_1$. When integrated with Bitbucket, TeamCity's T_{deploy} has the read permissions of both $Repo_1$ and $Repo_2$, as well as the user profile of $User_1$. In addition, the GitLab CI controller grants the T_{deploy} the same permissions as the user who triggered the CI task [37]. Thus, in the CI task of $Repo_1$ initiated by $User_1$, the T_{deploy} also has the read permission of $Repo_2$, which is over-privileged.

TABLE 3: The T_{deploy} over-privilege problems. The columns highlighted in gray color indicate over-privileged permissions. ✓ means the token has this permission. *R*-Read, *W*-Write, *A*-Admin.

CPs	CHPs	$Repo_1$			$Repo_2$			User Profiles
		R	W	A	R	W	A	
GitLab CI	GitLab	✓			✓			
TeamCity	GitHub†	✓	✓	✓	✓	✓	✓	
	GitLab	✓	✓	✓	✓	✓	✓	✓
	Bitbucket	✓			✓			✓
Jenkins	GitHub	✓						
	GitLab†	✓	✓	✓	✓	✓	✓	✓
	Bitbucket†	✓	✓	✓	✓	✓	✓	✓

† Reusing the T_{chp}.

The $T_{artifact}$ of multiple CPs also suffer from over-privilege problems. As discussed in Section 3.3, $T_{artifact}$ should not support modifying the artifacts after they are generated. However, TravisCI, CircleCI, and Jenkins fail to apply the write-once-read-many (WORM) model [38] to protect the artifacts. Their $T_{artifact}$ can be used to modify existing artifacts as long as the token is valid. Even worse, TravisCI directly uses AWS's long-term access token as $T_{artifact}$ [39], which has the permission to modify all artifacts in the repository, regardless of the owner (e.g., CI tasks) of an artifact.

5.2.2. Cross-task Reused Token

Token reuse (e.g., using the same token as T_{task}, T_{deploy}, and $T_{artifact}$) is quite common in all CPs. We do not find any security threats of token reuse if the token is one-time and only valid in a single task. However, with a different life cycle, token reuse might pose security threats. For instance, when TeamCity is integrated with GitHub, the T_{chp} will be reused as a task's T_{deploy}. Similar reuse also happens when Jenkins is integrated with Bitbucket and GitLab. Typically, once authorized, the T_{chp} is valid for a long time with high permissions (e.g., full access to multiple repositories), as the T_{chp} might be used in multiple subsequent tasks. If CPs simply reuse T_{chp} as T_{deploy}, the T_{deploy} then has extra unnecessary permissions, and the T_{deploy} leakage can further cause serious security issues.

Particularly, the problem is even worse in Jenkins. First, as described in Section 3.2, there is a security risk if CPs use personal access credentials as T_{chp}. Unfortunately, Jenkins uses them in the form of username/password when integrated

TABLE 4: Token validity period (in minutes) of CPs. Hyphen (-) means the token expires immediately. Infinity (∞) means the token does not expire after two weeks.

CPs	CHPs	T_{task}	T_{deploy}	$T_{artifact}$
GitHub Actions	GitHub	-	-	-
GitLab CI	GitLab	-	-	-
Bitbucket Pipelines	Bitbucket	180	180	-
CircleCI	GitHub	300	∞	15
	GitLab	300	∞	15
	Bitbucket	300	∞	15
TravisCI	GitHub	-	60	∞
	GitLab	-	∞	∞
	Bitbucket	-	∞	∞
TeamCity	GitHub	-	∞	10
	GitLab	-	120	10
	Bitbucket	-	120	10
Jenkins	GitHub	-	∞	60
	GitLab	-	∞	60
	Bitbucket	-	∞	60

with GitLab and Bitbucket, which violates security guidelines and might lead to source code leakage [37] [40]. Moreover, Jenkins reuses users' T_{chp} as their tasks' T_{deploy}, which can further pose a serious security threat to the users' repositories.

5.2.3. Long-lived Token

For security considerations, T_{task}, T_{deploy}, and $T_{artifact}$ should be one-time and short-lived tokens which expire immediately after a CI task ends. In practice, these tokens in many CPs do not expire immediately. Particularly, in some CPs, these tokens will automatically expire after a period of time once the CI task ends. We consider this type of token as a medium security risk. In some cases, several tokens never expire during our experiments (1 day), unless the repository owner manually revokes them. For those tokens, we further check their validity multiple times within the next 2 weeks. If these tokens are still valid after 2 weeks, we treat them as non-expired with high-security threats. Overall, five CPs (i.e., Bitbucket Pipelines, CircleCI, TravisCI, TeamCity, and Jenkins) have problems, as listed in Table 4.

Specifically, we find that all T_{task}, T_{deploy}, and $T_{artifact}$ suffer from the medium risk. There are two main reasons: (1) Some CPs simply use the default expiration time when generating tokens from CHPs. For example, TravisCI uses GitHub's server-to-server token [41] as T_{deploy}. When this token is generated, GitHub sets a default expiration time of 1 hour. (2) Some CPs set the longest possible validity period for tokens. For example, the maximum execution duration of a CI task in CircleCI is 5 hours [42]. CircleCI then sets the validity period of T_{task} to 5 hours, ensuring that the token will not expire during the CI task (in the worst case), while 80% of workflows are reported to be completed within 10 minutes [43].

Even worse, some T_{deploy} and $T_{artifact}$ never expire. For instance, CircleCI's T_{deploy} is implemented by adding a public key to CHPs, which will never expire unless the repository owner manually revokes it. This also applies to CPs that use T_{chp} (which does not expire) as T_{deploy}. Examples include TeamCity integrated with GitHub and Jenkins integrated with GitLab/Bitbucket. Finally, we find that TravisCI uses AWS' long-term access token as $T_{artifact}$ to access S3, which also suffers from the problem.

TABLE 5: Overview of potential threats on each CP. ✓ means the CP is vulnerable.

CPs	Executors	Attack Vectors			
		A_1	A_2	A_3	A_4
GitHub Actions	Shell	✓		✓	
GitLab CI	Docker		✓	✓	
	Shell		✓	✓	
Bitbucket Pipelines	Docker			✓	
	Shell	✓		✓	
CircleCI	Docker			✓	✓
	Shell			✓	✓
TravisCI	Docker			✓	✓
TeamCity	Shell	✓	✓	✓	
Jenkins	Docker		✓	✓	✓
	Shell	✓	✓	✓	✓

6. Attack Vectors

Based on the token leakage and token misuse results, we propose four novel attacks that enable adversaries to access unauthorized resources, manipulate execution results, and bypass security mechanisms on CHPs. We still consider the same scenarios described in our threat model. In each attack vector, we describe the related tokens, specific attack methods, and affected CPs. We also collect data on the repositories of the open-source community using these CPs and evaluate the potential impact of these attacks. A summary of investigated CPs with potential vulnerabilities is listed in Table 5. Particularly, these attacks (except A_1) can affect not only self-hosted runners but also CP-hosted runners.

6.1. Task Hijacking Attack (A_1)

Related Token: *Runner Registration/Authentication Token*.
Adversaries: *Type I, II, III*.
Attack Method. Attackers can steal the $T_{register}$ and register a malicious runner to connect with the controller. Most controllers will simply approve this registration without any notification, as this $T_{register}$ has been utilized before. The controller will then distribute subsequent CI tasks of victim repositories (using the benign runner) to the malicious runner. The whole process is entirely transparent to CI tasks. The consequence of this attack can be severe. For example, if the victim (benign) runner is shared by multiple repositories, the new registered (malicious) runner can hijack subsequent CI tasks, allowing attackers to access all repositories' source code even without any permissions. In addition, when the controller distributes a CI task, it will also pass sensitive data such as `secrets` to the malicious runner, which can also be leaked to attackers.
Vulnerable CI Platforms. CPs that leaks the $T_{register}$ (shown in Table 2) are vulnerable to this attack, including GitHub Actions, Bitbucket Pipelines (shell mode executor), CircleCI, TeamCity, and Jenkins (the default launch mode). In CircleCI, runners actively pull tasks. When multiple runners

exist, CircleCI adopts a first-come, first-served strategy. A benign runner initiates an HTTP request[7] about every 10 minutes to pull a new task from the controller. The runner authenticates to the controller by placing its $T_{register}$ in the HTTP `Authorization` header. Attackers can steal this token via memory dumping. Once attackers get this token, they can actively pull tasks with higher frequency and more likely obtain tasks from the controller. Similarly, Bitbucket Pipelines using shell mode executor is also vulnerable to this attack.

GitHub Actions, TeamCity, and Jenkins prohibit a new runner from registering with the controller using the same $T_{register}$, if the benign runner is active. Unfortunately, the executor and the runner of these three CPs run in the same namespace. Thus, the executor process can force the runner process to exit (e.g., through commands such as `kill -9`). As a result, attackers can kill the benign runner process after stealing the $T_{register}$ and then immediately register a malicious runner. This attack is stealthy because nothing abnormal can be seen on the web portals of both CHP and CP. The only exception is that the CI task log might be incomplete (a task might not be completed as the runner is killed). However, attackers can modify the task log with the *Task Result Hijacking Attack (A_3)* described later.

6.2. Repository Privilege Escalation Attack (A_2)

Related Token: *CHP Token* and *Code Deploy Token*.
Adversaries: *Type I, II, III*.
Attack Method. Attackers who are only low-privilege collaborators of a repository (called $Repo_{task}$) in an organization can steal over-privileged tokens through CI tasks to access unauthorized resources. They can (1) conduct unauthorized operations on $Repo_{task}$, such as modifying the repository settings or reading the user profile of the repository owner; and (2) access unauthorized resources in other repositories (attackers do not have access permissions, but the owner of $Repo_{task}$ has). Note that this attack is not limited to a self-hosted runner. It also does not require multiple repositories sharing the same runner, as the source code is stored in CHP.
Vulnerable CI Platforms. TeamCity and Jenkins are affected by this attack. These two CPs' T_{deploy} have over-privilege problems when integrated with certain CHPs (as shown in Table 3). The token can be leaked via the JVM TI injection.

6.3. Task Result Hijacking Attack (A_3)

To ensure software quality, CI workflows typically apply a series of CI checks, such as code quality analysis and vulnerability check with the static application security testing tools [44]. The result of a CI task (i.e., success or failure) is one key indicator of whether a project passes quality checks. Based on the results, CPs will trigger different subsequent jobs. For example, all mainstream CHPs have a feature of status checks, i.e., mark a pull request as mergable only after it passes the CI checks [14] [45] [46]. Bitbucket even supports automatically merging a pull request if all checks pass [46]. In this attack, we show that attackers can manipulate the execution results of CI checks.
Related Token: *Task Token*.
Adversaries: *Type I, II*.
Attack Method. All target CPs rely on T_{task} to receive CI task results/logs during task execution. Attackers can steal the T_{task} through a CI task and use the stolen T_{task} to report a fake task result (e.g., success) to the controller before the real result (e.g., failure) is submitted. With a spoofed result, attackers can trick the repository owner into merge code with vulnerabilities even if multiple checks are adopted. Besides, attackers can also use the T_{task} to send modified logs to the controller to make the attack more stealthy.
Vulnerable CI Platforms. All seven CPs are affected by this attack. GitHub Actions, GitLab CI, CircleCI, TravisCI, and TeamCity, utilize the HTTP protocol to transmit logs and task results. Once attackers get the T_{task}, they can directly initiate an HTTP request to submit spoofed task results. Jenkins uses Java Network Launch Protocol (JNLP) [47] to implement the communication between the controller and the runner. Attackers can spoof the task result to the controller following the protocol. Besides, they can also utilize JVM TI injection to control the runner to return the tampered result.

Particularly, although attackers cannot steal the T_{task} of Bitbucket Pipelines in the Docker executor mode, they can still launch the A_3 attack. In this mode, the executor runs in a stand-alone Docker container, separated from both the runner and the auxiliary containers. When a task execution ends, the executor delivers the task result to the auxiliary Docker container based on a script file[8] inside the executor container. Attackers can simply modify the content of the above script to tamper with the task result.

6.4. Artifact Hijacking Attack (A_4)

Related Token: *Artifact Token*.
Adversaries: *Type I, II, III*.
Attack Method. As many CPs improperly configure the $T_{artifact}$'s permissions, attackers can exploit a leaked $T_{artifact}$ to inject malicious code into the artifacts generated in the CI task. In addition, the $T_{artifact}$ in many CPs have unnecessarily long validity periods, allowing attackers to modify an artifact even after the corresponding CI task has ended. Moreover, in some improperly configured CPs, attackers can potentially abuse the $T_{artifact}$ obtained from one CI task to attack other CI tasks of the same repository. Even worse, an over-privileged $T_{artifact}$ might allow attackers to access and modify the artifacts of other repositories.
Vulnerable CI Platforms. All four independent CPs support using AWS S3 as artifact storage. Among them, TravisCI, CircleCI, and Jenkins are vulnerable to this attack. Particularly, TravisCI uses AWS's long-term access token to access S3 [39] and directly passes this token to the executor as the $T_{artifact}$ through environment variables. Attackers can steal this token via reading these environment variables[9]. As all CI tasks of one repository share the same AWS access

[7]The URL is https://runner.circleci.com/api/v2/runner/claim.

[8]`/tmp/{id}/tmp/echoResultToResultFileScript.sh`
[9]`ARTIFACTS_KEY` and `ARTIFACTS_SECRET`

token, attackers can then arbitrarily modify all artifacts of the repository, including those generated by historical, current, or even future CI tasks, as long as the token remains valid. Even worse, if an organization sets artifact storage of multiple repositories to the same AWS S3 bucket, attackers can further use this token to access/modify artifacts belonging to other repositories, even though they do not have any access rights to these repositories.

CircleCI creates an AWS temporary security credential [48] for each CI task as its $T_{artifact}$ and further limits the token's read and write permissions to the artifacts of the current CI task. However, we find that CircleCI's executor has root privileges, allowing attackers to grab $T_{artifact}$ from memory through syscalls such as `ptrace` [15]. Furthermore, we find that the validity period of CircleCI's $T_{artifact}$ (about 15 minutes) is not synchronized with the execution time of the CI task. As a result, attackers have the opportunity to tamper with the generated artifacts after a CI task ends.

Jenkins uses AWS S3 presigned URLs [49] as $T_{artifact}$ [50]. There is no limitation about what files can be uploaded to a presigned URL. Anyone who gets a presigned URL can get temporary access to the specific file in S3. More specifically, Jenkins pre-allocates the storage path on S3 for the artifacts of a CI task. Then, it generates a presigned URL valid for 1 hour and sends the URL to a CI executor for uploading artifacts. Again, we find that the validity period of the presigned URL (1 hour) is not synchronized with the execution time of a CI task. Therefore, attackers can steal $T_{artifact}$ via JVM TI injection and further modify the generated artifacts even after a CI task ends.

Instead, TeamCity, which also uses AWS S3 presigned URLs, is not vulnerable as it adopts a different strategy. When TeamCity generates a presigned URL, it utilizes the `Content-MD5` parameter to specify the MD5 hash value [51] of the artifact to be uploaded. Thus, even if attackers get the presigned URL, they still cannot modify the artifact since the verification of the AWS server will fail.

6.5. Measurement Results

We conduct a large-scale measurement on three mainstream CHPs (i.e., GitHub, GitLab, and Bitbucket) to evaluate the potential impact of the above mentioned attacks in the open source community. Obviously, we are unable to measure private repositories, so the actual situation could be worse.

Data Collection. We employ multiple strategies to collect as many repositories as possible for different combinations of CPs and CHPs. All CPs except TeamCity support setting CI configurations through config files. Therefore, we collect a repository's CP by checking the corresponding CI config file. We are unable to collect repositories that integrate with TeamCity since TeamCity cannot be set through the config file. For GitLab and Bitbucket, we obtain all public repositories and their contents via public APIs [52] [53] [54] [55]. In total, we have collected 3,328,928 repositories on GitLab and 2,355,610 repositories on Bitbucket. The data was collected on November 2022.

We are unable to do a full crawl using GitHub API, facing the same difficulty as the recent work conducted by Koishybayev et al. [9]: GitHub's REST APIs limit the crawling rate to 5,000 requests/hour, with only the first 1,000 results returned for each query. We have then adopted a similar method (e.g., using GitHub Actions from GHArchive data [56] by analyzing `github_bot` generated events) [9] and successfully obtained 669,070 repositories. However, other CPs do not have `github_bot` generated events, we then use *GitHub Activity Data* on BigQuery [57] to collect repositories using other CPs (484,230 in total). Note that *GitHub Activity Data* only contains 3.3M repositories, which is a small part of GitHub's full public repositories (at least 38M). Moreover, in addition to the config file, Jenkins also supports setting CI configuration through the CI controller web portal, which we are unable to collect. Thus, more repositories actually use the above CPs.

Finally, the total numbers of collected repositories on the three CHPs are: GitHub Actions (669,070), GitLab CI (297,529), Bitbucket Pipelines (27,814), CircleCI (122,911), TravisCI (553,362), and Jenkins (19,562).

Task Hijacking Attack (A_1). Repositories using GitHub Actions, Bitbucket Pipelines, CircleCI, TeamCity, and Jenkins, while also using self-hosted runners, are vulnerable to A_1. The numbers of repositories using self-hosted runners are: 22,274 (7.49%) with GitLab CI, 9,290 (1.39%) with GitHub Actions, 178 (0.64%) with Bitbucket Pipelines, and 40 (0.03%) with CircleCI. Jenkins does not provide CP-hosted runners; thus we infer all 19,562 repositories with Jenkins are also vulnerable. Among them, many are very popular repositories. For example, 7 out of the top 100 stars repositories on GitHub use the self-hosted runner; multiple repositories of Google, Microsoft, Docker, Tencent, NVIDIA, and Apache also use GitHub Actions self-hosted runner. Part of the results is shown in Table 8 in the Appendix.

We take GitHub Actions as a case study. For popular repositories threatened by A_1, the `secrets` leakage is a serious threat. GitHub Actions uses `secrets` to transfer sensitive data in CI tasks. For safety concerns, GitHub Actions only transfers secrets to jobs that need them and promptly scrubs secrets from memory [18]. However, these secrets can easily be leaked with A_1. For example, attackers can modify the GitHub Runner source code [58] and compile a custom CI runner to record `secrets`, as the GitHub controller passes `secrets` to the runner during a CI task. In total, we find 8,464 secrets that could be stolen in 2,472 vulnerable repositories. Particularly, the most common secrets are for publishing package releases and for Docker Hub authentication. Due to ethical concerns, we do not actually obtain these secrets to test their validity and functionalities. Instead, we analyze their invocations in the CI build scripts to evaluate the possible damage. We list the threats of some popular repositories in Table 6. We can see that attackers can use the stolen secrets to perform malicious code injection or launch a software supply chain attack, potentially affecting a large number of users.

Repository Privilege Escalation Attack (A_2). In the open source community, if a repository using the GitLab CI/TeamCity/Jenkins has collaborators, it may be vulnerable to A_2. If the repository owner has access rights to multiple

TABLE 6: GitHub repositories* using GitHub Actions that are vulnerable to A_1, their vulnerable secrets, and possible consequences.

Repositories (# of Stars)	Vulnerable Secrets	Possible Consequences of Secrets Leakage
ge**ry/s**y (33K)	GH_RELEASE_PAT	Attackers can inject malicious code into the released package of s**y, a popular tool with 3.5M users.
d**m/d**m (17K)	ENV_D**M_S3_AWS_SECRET_ACCESS_KEY ENV_DOCKERHUB_KEY_MATERIAL	Attackers can steal the long-term AWS and Docker Hub credentials for publishing the package of d**m company.
k**y/k**y (15K)	PYPI_PASSWORD UBUNTU_UPLOAD_KEY	Attackers can publish a malicious version of k**y, which has 90,000 monthly downloads on PyPI.
cu**js/cu**js (14K)	NPM_TOKEN DOCKERHUB_TOKEN	Attackers can publish a malicious version of cu**js, which has 100,000+ monthly downloads on npm.
Ho**ew/ho**re (12K)	HO**EW_CORE_GITHUB_PACKAGES_TOKEN	Attackers can release malicious packages to the Ho**ew repository.

* Name hidden due to ethical concerns.

other repositories, all of these repositories might be accessible by evil collaborators. Among the 317,091 repositories using GitLab CI/Jenkins, there are 180,617 (56.96%) repositories with collaborators. Particularly, the owner of 49,305 (15.55%) repositories have access rights to other repositories.

Task Result Hijacking Attack (A_3). We measure A_3 on GitHub, as it allows us to obtain the branch protection settings of public repositories with public APIs [59]. In total, we have collected 669,070 repositories using GitHub Action, 469,609 using TravisCI, and 9,643 using CircleCI. Among them, we find that 107,859 repositories have enabled the branch protection rule of `required_status_checks` (i.e., merge a pull request only after it passes CI checks). The numbers of repositories with the rule enabled on different CPs are GitHub Actions 83,047 (12.41%), TravisCI 20,831 (4.44%), and CircleCI 2,977 (30.87%). We also notice that many popular repositories have this rule enabled. For example, 31.48% of repositories with more than 5,000 stars have enabled this rule.

Artifact Hijacking Attack (A_4). Repositories using TravisCI, CircleCI, and Jenkins that upload artifacts in CI build scripts are all vulnerable to A_4. We find that 4,463 (3.63%) repositories using CircleCI, 2,845 (0.51%) repositories using TravisCI, and 2,069 (10.58%) repositories using Jenkins are affected by this threat. All affected repositories using TravisCI are also vulnerable to long-term AWS-token leakage.

In addition, if the artifacts (e.g., the package releases or the intermediate files) are tampered with, there will be severe security risks, as attackers might covertly inject malicious code or stealthy replace artifacts. We analyze the top 50 repositories threatened by A_4, and find that 22 (44%) of them use artifacts to store package releases or intermediate files. A list of the top 8 stars vulnerable repositories is presented in Table 7.

TABLE 7: Top 8 stars repositories vulnerable to A_4 and their artifacts' usage. *Rls.*-Releases, *Inter.*-Intermediates, *Rep.*-Reports.

Repositories (# of Stars)	CPs	Rls.	Inter.	Rep.
facebook/react (198K)	CircleCI	✓	✓	✓
facebook/react-native (106K)	CircleCI	✓	✓	✓
angular/angular (85K)	CircleCI	✓		✓
callemall/material-ui (83K)	CircleCI			✓
storybooks/storybook (75K)	CircleCI			✓
kadirahq/react-storybook (74K)	CircleCI			✓
huggingface/transformers (74K)	CircleCI			✓
docker/docker (65K)	Jenkins	✓		

7. Countermeasures and Disclosure

7.1. Defense Practices

We propose several defensive practices to reduce the occurrence of discovered threats.

Proper Isolation Between Runner and Executor. Since the executor may execute code from an untrusted source, CPs should properly isolate the executor from the runner, preventing the executor from interfering with the runner or accessing resources owned by the runner. Our analysis results show that executors in the Docker mode are generally better isolated than those in the shell mode. However, Docker containers still need to be carefully configured to avoid token leakage (e.g., avoid leaking tokens through environment variables). Also, using a separate Docker container to handle executor authentication is an effective method, as adopted by Bitbucket Pipelines.

Limiting Permission and Lifetime of Tokens. When CPs obtain tokens from CHP, they should follow the principle of least privilege, only applying for the needed permissions. In addition, CPs should continuously optimize the authorization with CHP according to the updated security functions. For example, GitHub is publicly testing fine-grained PATs [60], which provides granular control over the permissions and repository access that they grant to a PAT. Meanwhile, CPs that use TPSs as artifact storage should also adopt the security features of TPSs to prevent artifacts from being tampered with. For example, AWS S3 provides not only the `Content-MD5` function [51] (already adopted by TeamCity) but also S3 Object Lock [38], which can be used to prevent artifacts from being deleted or overwritten. Finally, all tokens' validity periods must be strictly controlled, and a one-time token must expire immediately after the task ends.

Scrutinizing Reused Resources. Organizations should carefully design their CI workflows and adopt CPs to balance resource usage and system security. They should thoroughly check reused resources to exclude potential risks of token leakage. For example, a runner should only be shared among repositories maintained by developers with the same privileges within the organization. A self-hosted runner should not be shared between public repositories and private repositories. Also, the continuous runner mode should only be used in a controlled and trusted environment.

7.2. Disclosure

We have initiated responsible disclosure with seven CPs to help them mitigate the detected threats. We have reported detailed information of all identified vulnerabilities and their potential threats, as well as the mitigation proposed in Section 7.1 to these entities. We have received multiple positive feedback. The TeamCity team have confirmed multiple issues, including task hijacking (A_1), repository privilege escalation (A_2), and task result hijacking (A_3). They have marked two issues as 'show-stopper' (the highest level) and have fixed the JVM TI injection vulnerability. The Jenkins team have acknowledged our report and confirmed that not sharing runners across trust boundaries is a necessary step to prevent similar attacks. The Bitbucket team have confirmed task hijacking (A_1) and task result hijacking (A_3) issues. Furthermore, they rewarded us with a bug bounty of $1,200. The GitLab team have appreciated our work and confirmed issues with repository privilege escalation (A_2) and task result hijacking (A_3). The GitHub team have confirmed issues with task hijacking (A_1) and task result hijacking (A_3). For A_1, they recommend using separate runner groups for repositories with different trust levels. For A_3, they said that they will make the functionality (GitHub Actions alters the result of a CI run) more strict in the future. The CircleCI team have confirmed our reported issues and rewarded us with a gift. They have also promised to better protect the tokens.

8. Related Work

CI Security. There are many research works study the impact of CI services [61] [62] [63] and use CI to optimize builds and reduce cost [64] [65] [66] [67]. Meanwhile, CI security has also gradually attracted much researchers' attention [68] [63] [69] [9] [70] [71] [25]. Hilton et al. [68] showed that securing `secrets` is a significant concern in CI security via semi-structured interviews with CI users. Li et al. [69] studied the abuse of CI services and found many illicit cryptomining instances on mainstream CPs. Koishybayev et al. [9] performed a large scale measurement study on GitHub Actions workflows. They found that 99.8% workflows apply the default permission, which is actually over-privileged. They also uncovered that GitHub Actions stores plaintext `secrets` in files on the CI runner, which can be retrieved by third-party actions. Unlike previous works, we are the first to examine the interaction among multiple primary stakeholders of CI and the authentication/authorization process through the entire CI workflow, to the best of our knowledge.

Configuration smells in CI is another security issue that has been studied extensively [25] [72] [73] [74] [75] [76] [77]. These works focus on automatically identifying and fixing improper configurations. Vassallo et al. [75] built a reporting tool to detect anti-patterns by analyzing CI build logs and repository information. Our work focuses on security flaws caused by CPs, which pose threats to users even if they use the proper CI configurations.

Credential Security. OAuth protocol is widely used for authentication and thus attracted extensive research efforts in recent years, including modeling OAuth protocol's security [78] [79] and finding flaws in protocol implementations [80] [81]. With many practices being proposed to help developers securely implement the OAuth protocol [78], Chen et al. [80] demonstrated that many mobile apps still incorrectly implement OAuth. While some T_{chp} studied in our work are implemented with OAuth, other tokens are not. We also mainly focus on the permissions and validity periods, which are unrelated to OAuth implementation.

Many studies have also been conducted on studying potential risks of credential leakage [82], [83]. Bai et al. [84] studied the token leakage on mobile offline payment caused by design flaws. Meli et al. [70] demonstrated that secret leakage is a serious problem affecting over 100,000 public GitHub repositories. Our work shows that existing CI services also suffer from the token leakage problem.

Software Supply Chain Security. Existing software supply chain faces many security threats, thus have recently attracted many research efforts, including vulnerable package dependencies [85] [86] [87] [88] [89], resource reuse [90] [91] [92], and typosquatting [93] [94] [95] [96]. Enck et al. [3] discussed the top five challenges in software supply chain security and argued that "securing the build process" is very important. Zahan et al. [97] analyzed the metadata of 1.63 million JavaScript npm packages and proposed six weak link signals, discovering that many maintainers' accounts can be taken over. Duan et al. [98] designed a comparative framework to identify security threats in PyPI, npm, and RubyGems, and detected hundreds of malicious packages. Zimmermann et al. [99] found that some maintainer accounts could be abused to inject malicious code into many packages. Our work further complements these previous research efforts on understanding CI security in the software supply chain.

9. Conclusion

This paper systematically analyzes potential security threats in CI workflows. We find that weak resource isolation and token misuse exist in CPs. We develop an analysis tool to investigate potential vulnerabilities in seven popular CPs and conduct a large-scale measurement on three mainstream CHPs. Our experimental results show that all CPs have the risk of token leakage, and many of them use over-privileged tokens with improper validity periods. We reveal four novel attack vectors and identify that some popular repositories and large organizations are potentially affected by these attacks. We have discussed potential mitigation, reported all security issues to corresponding CPs, and received positive responses.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. This work is partially supported by the National Natural Science Foundation of China (U1836213, U19B2034). The authors from the University of Delaware were partially supported by the National Science Foundation (NSF) grant CNS-2054657.

References

[1] "Enterprise CI/CD: A Holistic View," https://octo.vmware.com/enterprise-ci-cd-holistic-view/.

[2] "State of Continuous Delivery Report: The Evolution of Software Delivery Performance," https://cd.foundation/state-of-cd-june-2022/.

[3] W. Enck and L. Williams, "Top Five Challenges in Software Supply Chain Security: Observations From 30 Industry and Government Organizations," *IEEE Security & Privacy*, 2022.

[4] "GHSL-2020-235: Arbitrary Command Injection in wayou/turn-issues-to-posts-action," https://securitylab.github.com/advisories/GHSL-2020-235-wayou-turn-issues-to-posts-action/.

[5] "Hackers Backdoor PHP Source Code After Breaching Internal Git Server," https://arstechnica.com/gadgets/2021/03/hackers-backdoor-php-source-code-after-breaching-internal-git-server/.

[6] "Travis-CI," https://www.travis-ci.com/.

[7] "Cloud Application Platform | Heroku," https://www.heroku.com/.

[8] "Security Alert: Attack Campaign Involving Stolen OAuth User Tokens Issued to Two Third-Party Integrators," https://github.blog/2022-04-15-security-alert-stolen-oauth-user-tokens/.

[9] I. Koishybayev, A. Nahapetyan, R. Zachariah, S. Muralee, B. Reaves, A. Kapravelos, and A. Machiry, "Characterizing the Security of GitHub CI Workflows," in *USENIX Security*, 2022.

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

[11] "Storing Build Artifacts - CircleCI," https://circleci.com/docs/artifacts/.

[12] "Controller Isolation," https://www.jenkins.io/doc/book/security/controller-isolation/.

[13] "GitHub Actions Update: Helping Maintainers Combat Bad Actors," https://github.blog/2021-04-22-github-actions-update-helping-maintainers-combat-bad-actors/.

[14] "About Protected Branches," https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/defining-the-mergeability-of-pull-requests/about-protected-branches.

[15] "KernelHardening - Ubuntu Wiki," https://wiki.ubuntu.com/SecurityTeam/Roadmap/KernelHardening#ptrace_Protection.

[16] "GitHub Docs: Creating A Personal Access Token," https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token.

[17] "Personal Access Tokens - GitLab," https://docs.gitlab.com/ee/user/profile/personal_access_tokens.html.

[18] "Security Hardening for GitHub Actions," https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions.

[19] "GitHub Actions Artifacts," https://docs.github.com/en/rest/actions/artifacts.

[20] "Job Artifacts API - GitLab," https://docs.gitlab.com/ee/api/job_artifacts.html.

[21] "GitLab CI/CD Job Token," https://docs.gitlab.com/ee/ci/jobs/ci_job_token.html.

[22] "Automatic Token Authentication," https://ghdocs-prod.azurewebsites.net/en/actions/security-guides/automatic-token-authentication.

[23] "Managing Deploy Keys," https://docs.github.com/en/developers/overview/managing-deploy-keys.

[24] "Team Tools - The State of Developer Ecosystem in 2020 Infographic," https://www.jetbrains.com/lp/devecosystem-2020/team-tools/.

[25] K. Gallaba and S. McIntosh, "Use and Misuse of Continuous Integration Features: An Empirical Study of Projects That (Mis)Use Travis CI," *IEEE Trans. Software Eng.*, 2018.

[26] "mitmproxy - an interactive HTTPS proxy," https://mitmproxy.org/.

[27] "Docker Inspect," https://docs.docker.com/engine/reference/commandline/inspect/.

[28] "Branches API - GitHub," https://docs.github.com/en/rest/branches/branches#rename-a-branch.

[29] "Jobs API - GitLab," https://docs.gitlab.com/ee/api/jobs.html.

[30] "The Bitbucket Cloud REST API," https://developer.atlassian.com/cloud/bitbucket/rest/api-group-users/.

[31] "PutObject - Amazon Simple Storage Service," https://docs.aws.amazon.com/AmazonS3/latest/API/API_PutObject.html.

[32] "Trivy," https://github.com/aquasecurity/trivy.

[33] "About GitHub-hosted Runners," https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners.

[34] "Sudo CircleCI Make Me a Sandwich," https://circleci.com/blog/sudo-circleci-make-me-a-sandwich/.

[35] "Java Virtual Machine Tool Interface (JVM TI)," https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/index.html.

[36] "JVM Tool Interface," https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html.

[37] "GitLab Token Overview," https://docs.gitlab.com/ee/security/token_overview.html.

[38] "Using S3 Object Lock - Amazon Simple Storage Service," https://docs.aws.amazon.com/AmazonS3/latest/userguide/object-lock.html.

[39] "Uploading Artifacts on Travis CI," https://docs.travis-ci.com/user/uploading-artifacts/.

[40] "Personal Access Tokens," https://confluence.atlassian.com/bitbucketserver072/personal-access-tokens-1005335924.html.

[41] "Authenticating with GitHub Apps," https://ghdocs-prod.azurewebsites.net/en/developers/apps/building-github-apps/authenticating-with-github-apps.

[42] "Updates to Maximum Duration of Jobs," https://support.circleci.com/hc/en-us/articles/4411086979867-Updates-to-maximum-duration-of-jobs.

[43] "The Data-Driven Case for CI: What 30 Million Workflows Reveal About Devops In Practice," https://circleci.com/blog/the-data-driven-case-for-ci-what-30-million-workflows-reveal-about-devops-in-practice/.

[44] "Static Application Security Testing (SAST) - GitLab," https://docs.gitlab.com/ee/user/application_security/sast/.

[45] "Merge When Pipeline Succeeds - GitLab," https://docs.gitlab.com/ee/user/project/merge_requests/merge_when_pipeline_succeeds.html.

[46] "Suggest or Require Checks Before a Merge Bitbucket Cloud," https://support.atlassian.com/bitbucket-cloud/docs/suggest-or-require-checks-before-a-merge/.

[47] "Java Network Launch Protocol - The Java Tutorials," https://docs.oracle.com/javase/tutorial/deployment/deploymentInDepth/jnlp.html.

[48] "Using Temporary Credentials with AWS Resources," https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_temp_use-resources.html.

[49] "Generating a Presigned URL to Upload an Object - Amazon Simple Storage Service," https://docs.aws.amazon.com/AmazonS3/latest/userguide/PresignedUrlUploadObject.html.

[50] "[JENKINS-50518] Use Presigned URLs for Upload," https://issues.jenkins.io/browse/JENKINS-50518.

[51] "Generate a Pre-Signed URL for an Amazon S3 PUT Operation with a Specific Payload," https://docs.aws.amazon.com/sdk-for-go/v1/developer-guide/s3-example-presigned-urls.html#generate-a-pre-signed-url-put.

[52] "Projects API - GitLab," https://docs.gitlab.com/ee/api/projects.html.

[53] "List Public Repositories - The Bitbucket Cloud REST API," https://developer.atlassian.com/cloud/bitbucket/rest/api-group-repositories/.

[54] "Repository Files API - GitLab," https://docs.gitlab.com/ee/api/repository_files.html.

[55] "Get File or Directory Contents - The Bitbucket Cloud REST API," https://developer.atlassian.com/cloud/bitbucket/rest/api-group-source/#api-repositories-workspace-repo-slug-src-post.

[56] "GH Archive," https://www.gharchive.org/.

[57] "GitHub Data, Ready for You to Explore with BigQuery," https://github.blog/2017-01-19-github-data-ready-for-you-to-explore-with-bigquery/.

[58] "Github Actions - Runner Authentication and Authorization," https://github.com/actions/runner/blob/c0bc4c02f860d8cf247d42ef4ede0b016dc0007c/docs/design/auth.md.

[59] "Protected Branches," https://docs.github.com/en/rest/branches/branch-protection.

[60] "Introducing Fine-grained Personal Access Tokens for GitHub," https://github.blog/2022-10-18-introducing-fine-grained-personal-access-tokens-for-github/.

[61] N. Cassee, B. Vasilescu, and A. Serebrenik, "The Silent Helper: The Impact of Continuous Integration on Code Reviews," in *IEEE SANER*, 2020.

[62] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu, "The Impact of Continuous Integration on Other Software Development Practices: A Large-Scale Empirical Study," in *IEEE/ACM ASE*, 2017.

[63] D. Wermke, N. Wöhler, J. H. Klemmer, M. Fourné, Y. Acar, and S. Fahl, "Committed to Trust: A Qualitative Study on Security & Trust in Open Source Software Projects," in *IEEE S&P*, 2022.

[64] M. Shahin, M. A. Babar, and L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," *IEEE Access*, 2017.

[65] T. Yu and T. Wang, "A Study of Regression Test Selection in Continuous Integration Environments," in *IEEE ISSRE*, 2018.

[66] B. Chen, L. Chen, C. Zhang, and X. Peng, "BUILDFAST: History-Aware Build Outcome Prediction for Fast Feedback and Reduced Cost in Continuous Integration," in *IEEE/ACM ASE*, 2020.

[67] K. Gallaba, M. Lamothe, and S. McIntosh, "Lessons from Eight Years of Operational Data from a Continuous Integration Service: An Exploratory Case Study of CircleCI," in *IEEE/ACM ICSE*, 2022.

[68] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in Continuous Integration: Assurance, Security, and Flexibility," in *ESEC/FSE*, 2017.

[69] Z. Li, W. Liu, H. Chen, X. Wang, X. Liao, L. Xing, M. Zha, H. Jin, and D. Zou, "Robbery on DevOps: Understanding and Mitigating Illicit Cryptomining on Continuous Integration Service Platforms," in *IEEE S&P*, 2022.

[70] M. Meli, M. R. McNiece, and B. Reaves, "How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories," in *NDSS*, 2019.

[71] V. Gruhn, C. Hannebauer, and C. John, "Security of Public Continuous Integration Services," in *ACM WikiSym*, 2013.

[72] W. Felidré, L. Furtado, D. A. da Costa, B. Cartaxo, and G. Pinto, "Continuous Integration Theater," in *ACM/IEEE ESEM*, 2019.

[73] C. Paule, T. F. Düllmann, and A. Van Hoorn, "Vulnerabilities in Continuous Delivery Pipelines? A Case Study," in *IEEE ICSA-C*, 2019.

[74] A. Rahman, C. Parnin, and L. Williams, "The Seven Sins: Security Smells in Infrastructure as Code Scripts," in *IEEE/ACM ICSE*, 2019.

[75] C. Vassallo, S. Proksch, H. C. Gall, and M. Di Penta, "Automated Reporting of Anti-Patterns and Decay in Continuous Integration," in *IEEE/ACM ICSE*, 2019.

[76] C. Vassallo, S. Proksch, A. Jancso, H. C. Gall, and M. Di Penta, "Configuration Smells in Continuous Delivery Pipelines: A Linter and a Six-Month Study on GitLab," in *ACM ESEC/FSE*.

[77] F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. Gall, and M. Di Penta, "An Empirical Characterization of Bad Practices in Continuous Integration," *Empirical Software Engineering*, 2020.

[78] T. Lodderstedt, M. McGloin, and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations," Tech. Rep., 2013.

[79] D. Fett, R. Küsters, and G. Schmitz, "A Comprehensive Formal Security Analysis of OAuth 2.0," in *ACM CCS*, 2016.

[80] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague, "Oauth Demystified for Mobile Application Developers," in *ACM CCS*, 2014.

[81] P. Philippaerts, D. Preuveneers, and W. Joosen, "OAuch: Exploring Security Compliance in the OAuth 2.0 Ecosystem," in *RAID*, 2022.

[82] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel, "Attacking and Fixing PKCS# 11 Security Tokens," in *ACM CCS*, 2010.

[83] D. Choi and Y. Lee, "Eavesdropping One-Time Tokens Over Magnetic Secure Transmission in Samsung Pay," in *WOOT*, 2016.

[84] X. Bai, Z. Zhou, X. Wang, Z. Li, X. Mi, N. Zhang, T. Li, S.-M. Hu, and K. Zhang, "Picking Up My Tab: Understanding and Mitigating Synchronized Token Lifting and Spending in Mobile Payment," in *USENIX Security*, 2017.

[85] A. Decan, T. Mens, and E. Constantinou, "On the Impact of Security Vulnerabilities in the npm Package Dependency Network," in *MSR*, 2018.

[86] J. Hejderup, *In Dependencies We Trust: How Vulnerable Are Dependencies in Software Modules?* Delft University of Technology, 2015.

[87] C.-A. Staicu and M. Pradel, "Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers," in *USENIX Security*, 2018.

[88] J. C. Davis, E. R. Williamson, and D. Lee, "A Sense of Time for JavaScript and Node.js: First-Class Timeouts as a Cure for Event Handler Poisoning," in *USENIX Security*, 2018.

[89] C.-A. Staicu, M. Pradel, and B. Livshits, "SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS," in *NDSS*, 2018.

[90] A. McDonald, C. Sugatan, T. Guberek, and F. Schaub, "The Annoying, the Disturbing, and the Weird: Challenges with Phone Numbers as Identifiers and Phone Number Recycling," in *ACM CHI*, 2021.

[91] Q. Li, *A Survey Study of Password Setting and Reuse*. University of Delaware, 2020.

[92] K. C. Wang and M. K. Reiter, "How to End Password Reuse on the Web," in *NDSS*, 2019.

[93] P. Agten, W. Joosen, F. Piessens, and N. Nikiforakis, "Seven Months' Worth of Mistakes: A Longitudinal Study of Typosquatting Abuse," in *NDSS*, 2015.

[94] M. T. Khan, X. Huo, Z. Li, and C. Kanich, "Every Second Counts: Quantifying the Negative Externalities of Cybercrime via Typosquatting," in *IEEE S&P*, 2015.

[95] T. Moore and B. Edelman, "Measuring the Perpetrators and Funders of Typosquatting," in *Springer FC*, 2010.

[96] J. Szurdi, B. Kocso, G. Cseh, J. Spring, M. Felegyhazi, and C. Kanich, "The Long 'Taile' of Typosquatting Domain Names," in *USENIX Security*, 2014.

[97] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What are Weak Links in the npm Supply Chain?" in *IEEE/ACM ICSE-SEIP*, 2022.

[98] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages," in *NDSS*, 2021.

[99] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small World with High Risks: A Study of Security Threats in the npm Ecosystem," in *USENIX Security*, 2019.

Appendix

TABLE 8: Some repositories on GitHub, which are threatened by A_1 and use GitHub Actions, and their numbers of stars.

Repositories	# of Stars
tensorflow/tensorflow	169,276
electron/electron	104,610
ant-design/ant-design	82,949
huggingface/transformers	74,705
docker/docker	64,576
pytorch/pytorch	60,492
coder/code-server	57,759
apache/airflow	28,245
ClickHouse/ClickHouse	26,170
google/jax	20,981
Tencent/ncnn	15,973
NVIDIA/nvidia-docker	15,449
microsoft/LightGBM	14,427
Homebrew/homebrew-core	11,867
mozilla-mobile	11,088