# Quantifying Security Issues in Reusable JavaScript Actions in GitHub Workflows

Hassan Onsori Delicheh
hassan.onsoridelicheh@umons.ac.be
University of Mons
Mons, Belgium

Alexandre Decan*
alexandre.decan@umons.ac.be
University of Mons
Mons, Belgium

Tom Mens
tom.mens@umons.ac.be
University of Mons
Mons, Belgium

## ABSTRACT

GitHub's integrated automated workflow mechanism called GitHub Actions promotes the use of Actions as reusable building blocks in workflows. The majority of those Actions are developed in JavaScript and depend on packages distributed through the npm package manager. Those packages can suffer from security vulnerabilities, potentially affecting the Actions that rely on them. Using a dataset of 8,107 JavaScript Actions, we analysed to which extent dependencies on npm packages expose these Actions to vulnerabilities. We observed that JavaScript Actions tend to rely on dozens of npm packages, and that the vast majority of them depend on npm package releases with known vulnerabilities. Most of these vulnerabilities are caused by indirect dependencies, making it difficult for Actions maintainers to analyse their exposure to security vulnerabilities. Moreover, indirect dependencies are more likely to suffer from vulnerabilities of higher severity. We also studied to which extent security weaknesses occur in the source code of JavaScript Actions. To do so, we used CodeQL to detect security weaknesses, revealing that more than 54% of the studied JavaScript Actions contain at least one security weakness, and a small subset of these weaknesses recur frequently in their code. This justifies the need for further studies and more advanced tool support for addressing security issues in the GitHub Actions ecosystem.

## KEYWORDS

GitHub Actions, security vulnerabilities, security weaknesses, npm, dependency network, CodeQL

**ACM Reference Format:**
Hassan Onsori Delicheh, Alexandre Decan, and Tom Mens. 2024. Quantifying Security Issues in Reusable JavaScript Actions in GitHub Workflows. In *21st International Conference on Mining Software Repositories (MSR '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3643991.3644899

## 1 INTRODUCTION

Open-source software (OSS) constitutes a significant portion of modern software applications, often comprising between 70% and 90% of their codebase [35]. It has become straightforward for software producers to incorporate reusable OSS components with just a few lines of code. This practice has lead to the emergence of software supply chains, where software products consist of both core components under full control of the organisation and reusable third-party components that originate outside of an organisation's trusted domain. While component reuse provides several benefits [14, 39], depending on third-party components can result in deep and complex dependency networks [9, 20, 40]. These networks induce important security risks, since software supply chains can become a deliberate target for attacks in which malicious actors actively implant vulnerabilities into reusable components, thereby compromising build and deployment pipelines. The reuse of such insecure third-party components has raised significant concerns [1, 5]. For this reason, OWASP listed "vulnerable and outdated components" as one of the top 10 security risks.[1] Well-known security instances related to the propagation of security vulnerabilities are the equifax [26] and Log4Shell [18] incidents.

We posit that GitHub Actions,[2] the workflow automation tool that is fully integrated into the GitHub social coding platform, is likely to be confronted with such insecure software supply chains. Given that GitHub Actions has become the leading workflow automation service on GitHub in less than 18 months after its public release [15], security concerns in its software supply chain can potentially affect millions of repositories that rely on this workflow mechanism. Indeed, automated workflows in GitHub repositories often rely on reusable components, called Actions [11]. Tens of thousands of such Actions are distributed through the GitHub Marketplace and through public repositories. These Actions may depend on vulnerable external software packages, or may contain security weaknesses in their code.

To address this issue, various tools have been proposed to aid developers in producing more secure code. For instance, GitHub introduced security alerts for Common Vulnerabilities and Exposures (CVEs) in dependencies on 16 November 2017, and on 30 September 2020 it introduced CodeQL (https://codeql.github.com) to provide security alerts for a wide range of Common Weakness Enumerations (CWEs). However, the efficacy of these and related security mechanisms remains uncertain [3, 10]. For example, repository maintainers may not be aware of such tools, may have decided not to rely on them, or may simply consider security to be unimportant.

Therefore, this paper aims to quantify how and where security issues manifest themselves in the code of JavaScript Actions and their dependency networks. To achieve this goal we rely on a dataset of 8,107 JavaScript Actions and their 47,906 releases distributed

*F.R.S.-FNRS Research Associate


*MSR '24, April 15–16, 2024, Lisbon, Portugal*

ACM ISBN 979-8-4007-0587-8/24/04.
https://doi.org/10.1145/3643991.3644899

[1] https://owasp.org/Top10/ (OWASP Top 10 - 2021)
[2] We refer to https://docs.github.com/en/actions/quickstart for readers that are unfamiliar with GitHub Actions.

between November 2019 and June 2023. We target four research questions:
*RQ1: To what extent do JavaScript Actions rely on npm packages?* As a preliminary step before assessing the exposure of JavaScript Actions to vulnerabilities originating from their dependencies, we provide quantitative evidence that the large majority of JavaScript Actions depend on npm package releases.
*RQ2: What are the characteristics of JavaScript Action dependencies?* We show that a large proportion of JavaScript Action dependencies are deeply nested and packages within the @actions namespace are heavily relied upon, forming potential single points of failure.
*RQ3: To what extent do JavaScript Actions have vulnerabilities in their dependency network?* We show that the large majority of JavaScript Action releases depend on at least one vulnerable npm package release, and most of the frequent CVEs were discovered in npm packages that JavaScript Actions indirectly depend on.
*RQ4: To what extent do JavaScript Actions have code weaknesses?* Focusing on their latest releases, we show that the majority of JavaScript Actions were affected by at least one security weakness type and that some CWE types are also more prevalent.

This research serves as an essential step for researchers and practitioners aiming to understand security issues in the ecosystem of Github Actions. It also helps workflow maintainers and tool developers to assess and reduce the security risks associated with the reuse of Actions within GitHub repository workflows.

## 2 RELATED WORK

This section presents the relevant related work, with a primary emphasis on software composition analysis and supply chain security (Section 2.1) and empirical research related to GitHub Actions (Section 2.2).

### 2.1 Software supply chain security

The common software development practice of relying on reusable components is facilitated through a multitude of package managers and registries for popular programming languages, such as npm for JavaScript, PyPI for Python, and Maven for Java. Component reuse has lead to the emergence of software supply chains, where software products depend on upstream components that are developed and maintained by external parties. These components and their own dependencies could be vulnerable, whether due to malicious intent or accidental flaws [2, 8, 23, 24, 38, 39, 42].

Software supply chain security entails securing the end-to-end process of developing, distributing, and maintaining software to mitigate vulnerabilities, malicious code, or other potential threats. This includes protecting software at every stage, from development to deployment to ensure the integrity and safety of software throughout its lifecycle. Enck and Williams [13] investigated the primary obstacles in ensuring software supply chain security by drawing insights from 30 industry and government entities. They identified two main challenges: how to manage vulnerable dependencies, and how to select trusted supply chain components.

In order to tackle the risks associated with reusing insecure third-party components, various Software Composition Analysis (SCA) [19] tools have been proposed, like GitHub's Dependabot, Snyk, and OWASP Dependency check. These tools identify and report known vulnerabilities by scanning and cross-referencing dependency networks with vulnerability databases like NVD[3] and the GitHub Advisory Database.[4] Developers can receive notifications of vulnerable dependencies, allowing them to update the dependencies to more secure versions. SCA tools typically do not confirm whether a vulnerable dependency poses a real threat or whether the vulnerable code remains unused [12]. In addition, open source SCA tools rely on public vulnerability databases, whereas commercial tools may maintain their own proprietary database, occasionally causing information mismatches [12]. Precision issues, such as false positives, can also arise [12, 28], potentially requiring more resource-intensive approaches like code-centric call graph analysis, and usage-based analysis to address them [17, 29–31].

Code scanning tools have been introduced to identify issues, bugs, coding style violations and security weaknesses in source code or binary code [37]. For example, GitHub's CodeQL code quality analysis engine identifies security, correctness, maintainability, and readability issues in source code for a wide variety of programming languages (including JavaScript), along with their respective libraries and frameworks.[5] The static analysis performed by CodeQL does not require to execute the code. In the context of research question *RQ4* we will rely on CodeQL's security query pack for JavaScript to identify security weaknesses in the JavaScript code of reusable Actions.[6] Assessing their security is important, since GitHub workflows frequently rely on such Actions [10, 11]. This implies that security weaknesses in the Actions code have the potential to induce security flaws in the workflows in which they are used, and by extension in the repositories to which these workflows belong, thereby impacting software supply chain security.

Research indicates that only a small proportion of GitHub repositories relying on CI/CD practices actually use code scanners and SCA tools for security analysis. Angermeir et al. [3] analysed the adoption of security tools in a large sample of 8,423 enterprise-driven GitHub repositories, and observed that only 6.83% of these repositories showed signs of integrating such tools. 169 of these repositories were found to use Dependabot, and at least 68 repositories were using CodeQL for security purposes. Decan et al. [10] studied over 22,000 GitHub repositories and reported that only 5% of them had configured Dependabot. Moreover, only 3.1% of them used Dependabot for monitoring Actions within their workflows.

### 2.2 GitHub Actions

Golzadeh et al. [15] quantitatively examined 91,810 GitHub repositories to observe changes in the CI/CD landscape over time. They observed that the adoption of GitHub Actions coincided with a significant decrease of the use of other CI/CD tools such as Travis. A qualitative study based on interviews with 22 experienced software practitioners complemented this quantitative analysis to understand the rationale behind the usage, co-usage, and migration between 31 different CI/CD tools [32]. The study identified a clear migration trend towards GitHub Actions.

[3] https://nvd.nist.gov
[4] https://github.com/advisories
[5] https://codeql.github.com/docs/codeql-overview/supported-languages-and-frameworks
[6] https://codeql.github.com/codeql-query-help/javascript

Prior empirical studies on GitHub Actions have mainly concentrated on how it is used and adopted. Kinsman et al. [21] examined 3,190 GitHub repositories to investigate changes in various development activity indicators related to GitHub Actions adoption. This adoption was discovered to lead to more pull requests being rejected each month and a decrease in the number of commits on successfully merged pull requests. This information is especially important for practitioners as it assists them in understanding and preventing negative consequences on their projects. The effects of how GitHub Actions usage relates to project features were investigated by Chen et al. [6]. Using statistical models, they analysed how GitHub Actions affects the frequency of changes and the efficiency of resolving pull requests and issues. The results provided insights in how practitioners are adjusting to, and gaining advantages from, GitHub Actions.

Decan et al. [11] quantitatively studied GitHub repositories and their workflows by analysing their jobs, steps, and reused Actions. They observed that almost all workflows use Actions, implying that any issues with these Actions (such as bugs and security vulnerabilities) could potentially impact the workflows relying on them. In a follow-up work, Decan et al. [10] investigated to which extent GitHub workflows rely on outdated Action releases. They found that the majority of workflows use an Action release that is lagging behind the latest available release by at least 7 months, and that could have been updated for bug or security fixes during at least 9 months.

Saroar and Nayebi [33] studied the motivations, decision criteria, and challenges related to the development, publication, and usage of reusable Actions. They observed that, when given choices of similar quality, Actions produced by verified individuals with a greater number of stars tend to be preferred. Moreover, users frequently switch to alternative reusable Actions when faced with problems such as bugs and inadequate documentation. Additionally, one of the most prevalent challenges encountered by users is the troubleshooting of workflow configuration files.

Some researchers have focused specifically on security issues in the context of CI workflows. Benedetti et al. [4] investigated the impact of security issues in GitHub workflows and proposed a security assessment methodology. To put their method into practice, they developed a tool called GHAST and tested it on 50 OSS projects. They uncovered a total of 24,905 security issues in workflows. Koishybayev et al. [22] examined GitHub Actions security issues, discovering that 99.8% of GitHub workflows have excessive privileges. Additionally, 23.7% were found vulnerable to arbitrary code execution, and 97% used unverified reusable Actions. These issues pose significant supply chain attack risks. Gu et al. [16] discovered token leakage risks and identified over-privileged tokens in seven popular CI platforms. They revealed four novel attack vectors that allowed privilege escalation and code injection in CI workflows. Large-scale analysis on code hosting platforms showed the vulnerability's impact on popular repositories and large organisations.

As shown above, many studies have explored the usage of GitHub workflows and the security issues that accompany it. However, we are not aware of any research having focused specifically on security issues in the reusable Actions used by GitHub workflows. Our current research is the first to quantify the potential exposure of Actions to vulnerabilities within their dependency network and to quantify security weaknesses in the source code of these Actions.

# 3 DATA EXTRACTION

To study security issues related to reusable JavaScript Actions, we extracted a dataset of such Actions and their releases developed in GitHub repositories between November 2019 (the official release date of GitHub Actions) and June 2023 (Section 3.1). We extracted the direct and indirect dependencies of these Actions on npm package releases (Section 3.2) and identified the security vulnerabilities in these package releases (Section 3.3). We also analysed the security weaknesses that were present in the JavaScript code of the Actions (Section 3.4). The characteristics of the dataset obtained after these steps, are summarised in Table 1. It also shows the characteristics of the latest release of Actions at the end of observation period. The quantitative analysis in this paper is based on this dataset. The data and code to replicate the analysis are available on *https://doi.org/10.5281/zenodo.10521914*.

**Table 1: Characteristics of the considered dataset.**

| | full dataset | latest release |
|---|---|---|
| JS Actions | 8,107 | 8,107 |
| JS Action releases | 47,906 | 8,107 |
| JS Action dependencies | 2,239,815 | 334,084 |
| npm packages | 8,609 | 7,920 |
| npm package releases | 31,509 | 22,830 |
| vulnerabilities (CVEs) | 351 | 305 |
| vulnerability (CVE) occurrences | 1,797 | 1,242 |
| CodeQL weakness types | - | 38 |
| weakness occurrences | - | 9,700 |

## 3.1 JavaScript Actions and their releases

Reusable Actions are developed in GitHub repositories and can be distributed through the GitHub Marketplace. In June 2023, approximately 19K Actions were available on the Marketplace. Since many Action developers do not share their Actions on the Marketplace, we aimed to obtain a larger dataset of Actions by relying on the open API service ecosyste.ms[7]. In June 2023, this service provided access to a list of 30,678 Actions known to be used in GitHub Actions workflows.

16,263 of these Actions (i.e., 53%) were JavaScript Actions, referred to as *JS Actions* in the remainder of the article. They will be our main focus because JavaScript is the only programming language directly supported by GitHub for developing Actions.

To analyse the evolution of JS Actions we study their releases. Actions can have multiple releases, each representing a deployable version that can be used by GitHub repository contributors. Since not every commit in an Action's repository corresponds to an actual deployable version, we relied on GitHub's release management system[8] to identify the Action releases that are expected to be reused in practice. In this way, we collected 67,627 releases of 10,501 JS Actions. We excluded the 5,756 JS Actions that have no releases on GitHub.

[7]https://packages.ecosyste.ms/registries/githubactions/packages
[8]https://docs.github.com/en/repositories/releasing-projects-on-github

### 3.2 npm package dependencies

The use of JavaScript enables Action developers to rely on JavaScript packages distributed through package managers such as *npm*. A JS Action declares its dependencies on npm packages through a `package.json` manifest file (the standard way to define a JavaScript project and its required dependencies, if any). We relied on this file to identify the dependencies used by a JS Action, if any.

To determine the *exact* versions required to built the Action, we looked in each Action's repository for a `package-lock.json` file (the standard way to declare the exact version of each dependency). This file provides the exact version of each required dependency, although the format may vary.[9] We retrieved this file for 47,906 out of 67,627 (i.e., 71%) Action releases belonging to 8,107 distinct Actions.

We dropped 2,394 Actions for which we could not determine their exact dependencies, either because they did not use a `package.json` file or because they had dependencies in `package.json` but did not declare a `package-lock.json` file. Based on the lock files, we extracted the direct and indirect dependencies for each release of each of the 8,107 JS Actions.

### 3.3 Security vulnerabilities

In order to determine which releases of which npm packages have known vulnerabilities, we relied on the OSV open source vulnerability database.[10] Its API provides security advisories for 50M+ versions of open source packages, including npm packages. It provides an accurate description of vulnerabilities that precisely corresponds to specific open-source package versions. Querying this API for the 31,509 npm package releases identified as direct and indirect dependencies of the 8,107 JS Actions in the previous step, we obtained a total of 351 distinct vulnerabilities and their corresponding CVEs.[11] These 351 vulnerabilities occurred 1,797 times, affecting 1,157 npm package releases out of 31,509 (i.e., 3.7%).

### 3.4 Security weaknesses in Actions code

Any JS Action that is used by some GitHub repository workflow must specify a JavaScript code file to be executed. The link to this file is provided after the `main:` keyword in the action.yml file stored at the root of the GitHub repository implementing the Action. To identify weaknesses in this JavaScript code, we used CodeQL because it offers a wide range of code queries that analyse the abstract syntax graph of the code. These queries can be used to detect a wide range of quality issues in the code, including bugs, bad smells, potential errors, and security weaknesses. CodeQL comes with a wide range of open source query packs that are available in the CodeQL repository.[12] We relied on the security query pack for JavaScript[13] to analyse the code of each JS Action in our dataset. To do so, we extracted the paths to the code files from the action.yml files in the latest release of all Actions. We extracted these files from the repositories of the Actions and analysed them with the security query pack. In this way, we identified 38 types of security weaknesses. These weaknesses occurred 9,700 times across 4,409 JS Actions (i.e., in 54.4% of the JS Actions in our dataset).

[9] https://docs.npmjs.com/cli/v10/configuring-npm/package-lock-json
[10] https://osv.dev
[11] https://cve.mitre.org
[12] https://github.com/github/codeql
[13] https://codeql.github.com/codeql-query-help/javascript

## 4 RQ1: TO WHAT EXTENT DO JAVASCRIPT ACTIONS RELY ON NPM PACKAGES?

JS Actions enable the execution of JavaScript code within a Node.js runtime environment. This unlocks the potential to reuse a vast number of JavaScript libraries distributed through package managers like npm.

As a first step towards assessing the exposure of JS Actions to vulnerabilities originating from their dependencies, RQ1 intends to quantify how many such Actions have dependencies on npm packages. To achieve this, we analysed the Action's releases during the observation period. Each month, we calculated the proportion of JS Actions that depended on npm packages between the start date of the observation period (November 1, 2019) and the end of that month. This allowed us to track the evolution of the proportion of Actions depending on npm packages, as shown in Figure 1. We observe that more than 95.2% and, on average, 96.5% of all considered Actions have at least one dependency on a npm package. This indicates that depending on npm packages is a common practice for JS Actions.

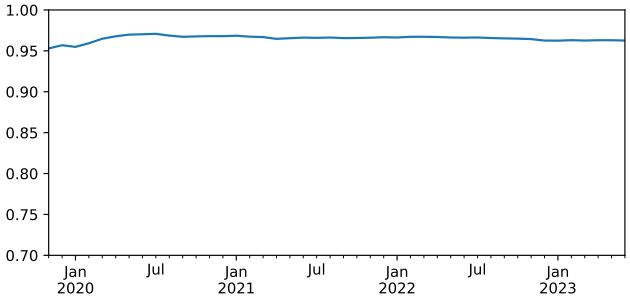


**Figure 1: Evolution of the proportion of considered JS Actions depending on npm packages.**

As a second step we aimed to understand whether these Actions relied on a substantial subset of npm packages or only a limited number of them. Such knowledge offers preliminary insights for characterising the dependency network of Actions in the subsequent research questions. We therefore analysed the evolution of the number of Actions and the number of npm packages directly or indirectly required as dependencies by these Actions. Figure 2 reports the results. At the end of the observation period (June 2023), 8,107 JS Actions depended on 8,609 distinct npm packages, including 31,509 distinct releases. This reveals that, although most of the considered JS Actions require npm packages, they tend to depend on a very small subset of the 2.5M+ packages in the npm registry.

**Summary:** The large majority of considered JS Actions (>95%) depend on npm package releases, but only a very small subset of npm packages is actually required by these Actions.

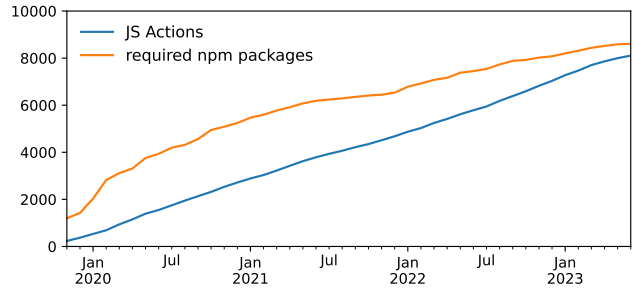


**Figure 2: Evolution of the number of JS Actions and (directly or indirectly) required npm packages.**

## 5 RQ2: WHAT ARE THE CHARACTERISTICS OF JAVASCRIPT ACTION DEPENDENCIES?

RQ1 revealed that the overwhelming majority of the 8,107 JS Actions directly or indirectly depend on 8,609 npm packages. It can be quite challenging for Action maintainers to assess the security of all those packages. This situation is consistent with the observations made in JavaScript projects [8, 23, 42]. Therefore, RQ2 aims to characterise the dependency network of JS Actions.

For each considered JS Action, we computed its npm dependency network, identifying the npm packages that were required as direct and indirect dependencies, and computing the number of dependencies at each depth in the network. Based on the latest release of Actions, Table 2 reports the number and proportion of JS Actions together with the number of dependencies they have at a given depth level in the dependency network. For instance, the second row shows that 7,396 of the 8,107 JS Actions in our dataset (i.e., 91.2%) have indirect dependencies in their latest releases, with a mean of 9.2 and median of 6 dependencies to npm package releases at a depth of 2. The first row shows that for direct dependencies (depth 1) the mean is only 3.9 and the median is only 3 dependencies to npm package releases.

**Table 2: Dependencies of JS Actions on npm package releases in June 2023.**

| | JS Actions | | # dependencies | | |
|---|---|---|---|---|---|
| **depth** | **#** | **%** | **mean** | **median** | **max** |
| 1 | 7,804 | 96.3 | 3.9 | 3 | 33 |
| 2 | 7,396 | 91.2 | 9.2 | 6 | 200 |
| 3 | 7,083 | 87.4 | 13.1 | 9 | 244 |
| 4 | 5,836 | 72.0 | 10.1 | 5 | 303 |
| 5 | 5,301 | 65.4 | 6.2 | 2 | 199 |
| 6 | 3,585 | 44.2 | 5.9 | 2 | 126 |
| 7 | 1,845 | 22.8 | 7.9 | 7 | 95 |
| 8 | 1,507 | 18.6 | 6.0 | 6 | 69 |
| 9 | 1,133 | 14.0 | 3.9 | 3 | 49 |
| 10+ | 904 | 11.1 | 2.5 | 1 | 54 |

A more detailed analysis reveals that a significant proportion of JS Action dependencies are deeply nested. For example, up to 65.4% of the Actions include indirect dependencies at a depth of 5. Additionally, releases of 2,303 distinct npm packages are directly required as dependencies and releases of 6,561 distinct npm packages are indirectly required as JS Action dependencies, accounting for a total of 22,830 npm package releases that are transitively required as dependencies.

Among the npm packages required as *direct* dependencies by the Actions in our dataset, package @actions/core is used by nearly all of them (94.6%). This is unsurprising since this package provides the necessary building blocks for creating a JS Action. More generally, we observed that packages belonging to the @actions[14] namespace are heavily depended upon. Those packages are provided by GitHub to facilitate the development and maintenance of Actions.

The package named tunnel emerged as the most prevalent npm package on which JS Actions *indirectly* depend, with approximately 79% of the Actions in our dataset indirectly depending on it. This suggests that some packages, including those that are deeply nested, may represent potential single points of failure. To assess to which extent JS Actions are exposed to such potential single points of failure, we quantified how many npm packages are required to satisfy the dependencies of all considered Actions.

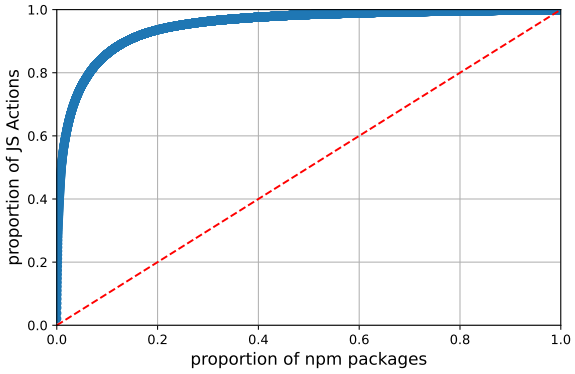


**Figure 3: Inverted Lorenz curve of the cumulative proportion of npm packages required as dependencies in JS Actions.**

Figure 3 presents an inverted Lorenz curve that shows the cumulative proportion of npm packages required to satisfy the dependencies of Actions, sorted by decreasing usage. It illustrates the extent of the inequality in using npm packages as Action dependencies. For example, only 20% of the npm packages are enough to cover all direct and indirect dependencies of more than 93% JS Actions, suggesting a strong concentration of reuse.

**Summary:** A significant proportion of JS Action dependencies are deeply nested. More than 91% of the latest releases of the considered JS Actions have indirect dependencies. Packages within the @actions namespace are heavily relied upon, exposing Actions to potential single points of failure. Only 20% of npm packages are needed to cover the dependencies of over 93.5% of Actions, signaling a high reuse concentration.

[14] https://github.com/actions/toolkit

## 6 RQ3: TO WHAT EXTENT DO JAVASCRIPT ACTIONS HAVE VULNERABILITIES IN THEIR DEPENDENCY NETWORK?

RQ2 revealed that a large majority of JS Actions directly or indirectly depend on multiple packages. The releases of these packages may have security vulnerabilities, exposing the Actions that (transitively) depend on them to those vulnerabilities. RQ3 therefore aims to quantify to what extent JS Actions are exposed to vulnerabilities in their dependency network, and how this evolves over time. To accomplish this, we extracted the exact versions and depths of npm packages specified as runtime dependencies of Actions. We ignored all development dependencies.

We also gathered information on security advisories known to affect the package releases, including their corresponding CVEs. By cross-referencing the package releases in the Actions dependency network with known vulnerability databases we identified 351 distinct CVEs in 1,157 out of 31,509 package releases. Breaking down these CVEs by severity level, we found 65 *critical* CVEs (18.5%), 159 CVEs (45%) of *high* severity, 119 CVEs (34%) of *medium* severity and 8 CVEs (2.5%) of *low* severity.

Vulnerabilities can arise in many npm packages and may manifest themselves at different depths of a JS Action dependency network. To shed light on the impact of vulnerable npm packages and to understand how JS Actions depend on these packages, we analysed the most frequent CVEs discovered within the dependency networks of the latest release of all considered JS Actions. Table 3 presents the top 10 CVEs detected, along with the extent to which JS Actions are potentially exposed to them, either directly or indirectly. One can observe that, apart from the two known vulnerabilities CVE-2022-35954 and CVE-2020-15228 in the @actions/core package, the remaining eight CVEs were discovered in npm packages that JS Actions mostly relied upon in an indirect way.

**Table 3: Top 10 vulnerabilities (CVEs) found in the dependency networks of JS Actions in June 2023.**

| | vulnerable | direct | | indirect | | JS Actions | |
|---|---|---|---|---|---|---|---|
| vulnerability | package name | # | % | # | % | # | % |
| CVE-2022-35954 | @actions/core | 4,458 | 55.0 | 38 | 0.5 | 4,496 | 55.5 |
| CVE-2022-0235 | node-fetch | 187 | 2.3 | 2,029 | 25.0 | 2,216 | 27.3 |
| CVE-2022-25883 | semver | 422 | 5.2 | 1,749 | 21.6 | 2,171 | 26.8 |
| CVE-2020-15228 | @actions/core | 1,196 | 14.7 | 18 | 0.2 | 1,214 | 14.9 |
| CVE-2020-15168 | node-fetch | 53 | 0.6 | 649 | 8.0 | 702 | 8.6 |
| CVE-2022-3517 | minimatch | 42 | 0.5 | 620 | 7.6 | 662 | 8.1 |
| CVE-2023-0842 | xml2js | 55 | 0.7 | 428 | 5.3 | 483 | 6.0 |
| CVE-2022-0536 | follow-redirects | 4 | 0.1 | 475 | 5.8 | 479 | 5.9 |
| CVE-2023-26136 | tough-cookie | 3 | 0.1 | 463 | 5.7 | 466 | 5.8 |
| CVE-2022-0155 | follow-redirects | 2 | 0.1 | 444 | 5.4 | 446 | 5.5 |

We analysed all 47,906 JS Action releases, and observed that a large majority of 37,084 of them (i.e., 77.4%) are exposed to at least one CVE. To study this phenomenon over time, we quantified the proportion of JS Actions exposed to a known CVE. Figure 4 presents this evolution for the considered observation period.

The blue line shows the proportion of all JS Actions that depend on a package release with a known vulnerability in their dependency network. This proportion oscillates between 0.83 and 0.96 before August 2022, and then starts to decrease to 0.71 at the end of the observation period. Investigating the sudden change trend

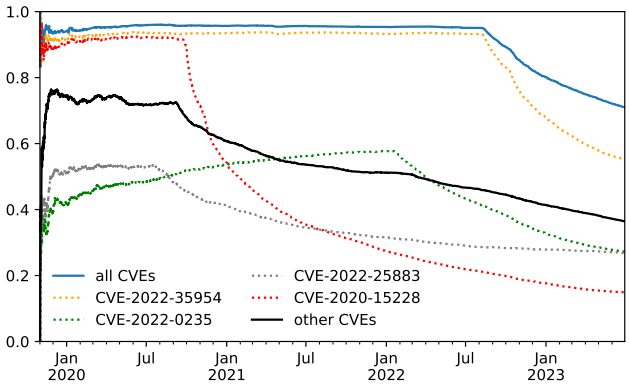


**Figure 4: Proportion of JS Actions exposed to CVEs. The dotted coloured lines correspond to the top four CVEs that were found in dependency networks of many JS Actions.**

in mid-August 2022, we found that most JS Actions were exposed to a single vulnerability CVE-2022-35954 in releases of the @actions/core package below version 1.9.1. This constitutes another example of a package that corresponded to a single point of failure. The orange dotted line in Figure 4 shows the evolution of the proportion of JS Actions exposed to this vulnerability. Given that @actions/core was required by nearly all Actions (see RQ2), a vulnerability fix in version 1.9.1 lead to an important overall decrease in exposure to this CVE. The red dotted line in Figure 4 indicates the evolution of the proportion of JS Actions exposed to another vulnerability CVE-2020-15228 in the @actions/core package, which was fixed in version 1.2.6 in September 2020.

We also identified two other CVEs that could potentially affect more than 10% of the latest JS Actions releases: CVE-2022-0235 for package node-fetch and CVE-2022-25883 for package semver. The evolution of the proportion of JS Actions exposed to these CVEs is shown by the green and grey dotted lines in Figure 4. The availability of a fix for these CVEs in August 2020 and January 2022, respectively, resulted in a significant reduction in vulnerability exposure afterwards.

The black curve in Figure 4 shows the evolution of the proportion of JS Actions exposed to vulnerabilities, excluding the aforementioned four CVEs from the analysis. This proportion decreases from 0.73 in September 2020 to 0.36 in June 2023. We examined both the absolute number of JS Actions and the number of exposed JS Actions to comprehend the reasons behind this declining proportion. Our investigation revealed that while the number of exposed JS Actions has been increasing gradually (rising from 1,415 in September 2020 to 2,957 in June 2023), the total number of available JS Actions also witnessed a surge, going from 1,959 to 8,107 during the same period. This explains why the proportion of exposed JS Actions decreased, despite an increasing number of exposed JS Actions.

In addition, we conducted a more detailed analysis to determine at which level of dependency depth, JS Actions depend on vulnerable npm packages, and what is the severity and frequency of these vulnerabilities. Table 4 provides an overview of this analysis, conducted on the latest release of Actions exposed to CVEs. The table

**Table 4: Overview of latest JS Action releases exposed to vulnerabilities, categorised by dependency depth and severity.**

| | JS Actions | exposed JS Actions | | vulnerable packages | | CVE | occurrence (CVE) | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| dependency type | # | # | % | # | % | # | # low | # medium | # high | # critical |
| direct (depth 1) | 7,804 | 4,885 | 62.6 | 97 | 4.2 | 159 | **1,252 (4)** | **5,484 (50)** | 1,116 (70) | 173 (35) |
| indirect (depth > 1) | 7,396 | 4,055 | 54.8 | 183 | 2.8 | 252 | 668 (3) | 5,405 (83) | **5,729 (119)** | **1,334 (47)** |
| depth 2 | 7,396 | 2,393 | 32.3 | 127 | 3.3 | 157 | 24 (2) | 2,404 (57) | 1,452 (73) | 277 (25) |
| depth 3 | 7,083 | 1,237 | 17.5 | 122 | 3.4 | 176 | 56 (3) | 1,189 (64) | 1,256 (83) | 260 (26) |
| depth 4 | 5,836 | 2,068 | 35.4 | 87 | 3.1 | 130 | 571 (1) | 458 (50) | 2,169 (59) | 200 (20) |
| depth 5 | 5,301 | 426 | 8.0 | 72 | 3.5 | 103 | 11 (1) | 351 (40) | 395 (44) | 179 (18) |
| depth 6 | 3,585 | 335 | 9.3 | 52 | 3.6 | 71 | 6 (1) | 273 (22) | 261 (34) | 330 (14) |
| depth 7 | 1,845 | 161 | 8.7 | 32 | 3.2 | 50 | 0 (0) | 112 (17) | 130 (21) | 47 (12) |
| depth 8 | 1,507 | 615 | 40.8 | 23 | 3.8 | 33 | 0 (0) | 582 (11) | 45 (14) | 21 (8) |
| depth 9 | 1,133 | 44 | 3.9 | 13 | 3.6 | 22 | 0 (0) | 32 (6) | 19 (7) | 14 (9) |
| depth 10+ | 904 | 10 | 1.1 | 5 | 2.0 | 5 | 0 (0) | 4 (2) | 2 (2) | 6 (1) |

reports the number of Actions with direct and indirect dependencies, the number and proportion of Actions exposed to CVEs, the number and proportion of vulnerable npm packages, the number of distinct CVEs known in vulnerable npm packages, and the number of occurrences of these CVEs, broken down by severity level. It also reports this information at each depth level in the dependency networks of JS Actions. For instance, it reports that 1,507 JS Actions have dependencies up to a depth of 8 in their dependency network. 615 (i.e., 40.8%) of these Actions are exposed to vulnerabilities from 23 vulnerable npm packages at depth 8, accounting for 3.8% of all npm packages at this depth. Additionally, it shows that there are 33 distinct CVEs known at this depth. We found 582 occurrences of 11 CVEs with *medium* severity, 45 occurrences of 14 CVEs with *high* severity, and 21 occurrences of 8 CVEs with *critical* severity.

As observed in Table 4, the direct dependencies of JS Actions experienced 4 CVEs with *low* severity occurring 1,252 times, 50 CVEs with *medium* severity occurring 5,484 times, 70 CVEs with *high* severity occurring 1,116 times, and 35 CVEs with *critical* severity occurring 173 times. Similarly, the indirect dependencies of JS Actions encountered 3 CVEs with *low* severity happening 668 times, 83 CVEs with *medium* severity happening 5,405 times, 119 CVEs with *high* severity happening 5,729 times, and 47 CVEs with *critical* severity happening 1,334 times. This implies that the majority of CVEs with *low* and *medium* severity are concentrated and frequently occurring in direct dependencies, while CVEs with *high* and *critical* severity are primarily occurring in indirect dependencies. This highlights the importance for JS Actions developers and maintainers to thoroughly assess not only the direct dependencies but also to carefully evaluate indirect dependencies.

**Summary:** The large majority of JS Action releases (i.e., 77.4%) depend on at least one vulnerable npm package release. We identified several single points of failures in the past, corresponding to vulnerabilities in earlier releases of individual packages that were widely used, and hence exposed a significant portion of Actions to those vulnerabilities. 8 out of 10 of the most frequent CVEs were discovered in npm packages that JS Actions indirectly depend on. Moreover, indirect dependencies are more likely to suffer from vulnerabilities of high and critical severity.

**Table 5: Proportion of latest JS Action releases affected by the top 10 CodeQL security weaknesses. The severity of each weakness type is given between parentheses. Weakness types marked with * are associated to CWE-20.**

| | affected JS Actions | | occurrence |
|---|---|---|---|
| security weakness type (severity) | # | % | # |
| missing regular expression anchor* (7.8) | 3,261 | 40.2 | 3,712 |
| incomplete string escaping or encoding* (7.8) | 824 | 10.2 | 1,946 |
| overly permissive regular expression range* (5.0) | 439 | 5.4 | 850 |
| regular expression injection (7.5) | 215 | 2.6 | 380 |
| potential file system race condition (7.7) | 188 | 2.3 | 208 |
| indirect uncontrolled command line (6.3) | 179 | 2.2 | 424 |
| incomplete URL substring sanitization* (7.8) | 168 | 2.1 | 190 |
| useless regular-expression character escape* (7.8) | 148 | 1.8 | 676 |
| untrusted data passed to external API* (7.8) | 145 | 1.8 | 281 |
| incomplete multi-character sanitization* (7.8) | 97 | 1.2 | 142 |
| *all other 28 weakness types (mean: 7.2)* | 596 | 7.3 | 895 |

## 7 RQ4: TO WHAT EXTENT DO JAVASCRIPT ACTIONS HAVE CODE WEAKNESSES?

Workflows in GitHub repositories often depend on Actions [10, 11]. This suggests that security weaknesses in the source code of these Actions can potentially impact the workflows that use them, as well as the repositories in which these workflows are being run. As a consequence, weaknesses in Actions code have the potential to impact the overall security of the software supply chain.

To quantitatively study this phenomenon, we relied on GitHub's CodeQL static code quality analysis engine for detecting security weaknesses in JS Actions code. More specifically, we used CodeQL's command-line interface to execute the security query pack for JavaScript,[15] aiming to identify security weaknesses in the code of the latest releases of JS Actions. These security queries are available as open source in the CodeQL repository. We used 96 distinct queries for JavaScript that were available in September 2023 and that were tagged with security. Comprehensive information about each query (e.g., tags and severity) can be found in their metadata.[16] Every security query is associated with one or more Common Weakness Enumerations (CWE).[17] We leveraged these associations to determine whether the code of a JS Action is affected by a security weakness.

[15] https://codeql.github.com/codeql-query-help/javascript
[16] https://codeql.github.com/docs/writing-codeql-queries/metadata-for-codeql-queries
[17] https://codeql.github.com/codeql-query-help/javascript-cwe

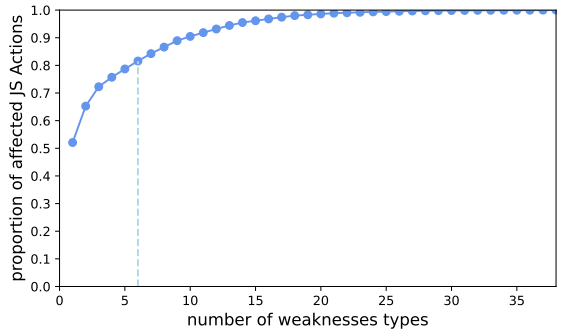


**Figure 5: Cumulative proportion of affected JS Actions in function of the number of weakness types. The dotted horizontal and vertical lines highlight that only 6 weakness types are responsible for over 80% of the affected JS Actions.**

In the latest releases of the 8,107 considered JS Actions, we identified 38 distinct security weakness types. Table 5 shows the proportion of the latest JS Action releases affected by the top 10 most frequently found weakness types. It provides information on their severity (a value ranging from 0 to 10), the number and proportion of JS Actions affected by them, and the number of occurrences that were found for each weakness type. For example, the first row of the table shows that 3,261 (i.e., 40.2%) of the latest JS Action releases are affected by the security weakness "missing regular expression anchor" that occurred 3,712 times in the code of JS Actions. The weakness types that do not belong to the top 10 are aggregated in the last row of the table. The average severity of these remaining weaknesses is 7.2, occurring 895 times in JS Actions code, and affecting the latest releases of 596 distinct JS Actions (i.e., 7.3%).

We also found JS Actions code to be affected by at least one security weakness in 4,409 out of 8,107 JS Actions (i.e., 54.4%). Furthermore, we observed multiple weaknesses in some JS Action releases, resulting in a total of 9,700 occurrences of weaknesses across the 4,409 JS Actions. A more thorough analysis of the distribution of the number of distinct weaknesses within affected JS Actions code revealed that the majority of JS Action releases were associated with a single weakness type, while more than 27% had more than one weakness type in their code, with a maximum of 9 different weakness types. We also observed that, on average, there are 2.2 occurrences of weaknesses within the code of a JS Action, with a median value of 1. We found one outlier JS Action containing 5 different weakness types in its code, accounting for a total of 81 occurrences.[18] The average severity of weaknesses occurring in the code of each JS Action was 7.6.

Table 5 reveals that the number of affected JS Actions per weakness type is highly skewed. For example, the single weakness "missing regular expression anchor" affected more than 40% of the latest JS Action releases. Therefore, we conducted a Pareto analysis, reported in Figure 5, revealing that only 6 out of the 38 weakness types (i.e., 15.7%) are responsible for over 80% of the affected JS Action releases.

[18]We do not reveal the name of this Action to avoid compromising its users.

Analysing the most frequent weakness types, we discovered that 7 of the top 10 weakness types (marked with * in Table 5) are associated with CWE-20, also known as *Improper Input Validation*.[19] This common weakness enumeration refers to the failure of an application to correctly validate input data, leaving it vulnerable to security issues such as injection attacks. For instance, the "missing regular expression anchor" weakness type relates to the sanitization of untrusted input using regular expressions.[20] While regular expressions are widely used for validating input data, they are susceptible to errors when attempting to match untrusted input without the presence of anchors (e.g., $). Malicious input can exploit this vulnerability by incorporating allowed patterns in unexpected places, thus circumventing security checks.

Concrete recommendations for mitigating the detected weakness types are provided in the CodeQL documentation of the corresponding security queries and their CWE coverage. Following these recommendations would allow JS Action developers to resolve many of the common security weaknesses, leading to a significant improvement in the security of these Actions. For example, addressing only the seven weakness types associated with CWE-20 in the top 10 of Table 5 would result in the resolution of 3,502 out of 4,409 affected JS Actions. As a consequence, only 907 out of the original 8,107 JS Actions (i.e., 11.2%), would remain affected by some other security weakness.

**Summary:** The latest releases of the considered JS Actions are affected by 38 distinct types of CodeQL security weaknesses. The majority of Actions (54.4%) was affected by at least one security weakness type. A total of 9,700 weakness occurrences were identified across the 4,409 affected Actions. Some weakness types are considerably more prevalent: 6 out of the 38 weakness types are responsible for over 80% of the affected Actions. Some CWE types are also more prevalent: 7 out of the top 10 most frequent weakness types are associated with CWE-20 (*Improper Input Validation*), and resolving them could potentially lead to an improvement by almost a factor 5 in the security of JS Actions.

# 8 DISCUSSION

## 8.1 Beyond JavaScript Actions

The focus of the current article was on JS Actions, since they constituted the majority (53%) of the 30,678 Actions in the considered dataset (cf. Section 3). These Actions were of particular interest due to their dependence on npm packages, and the fact that the npm dependency network is known to induce security issues in the software supply chain [7, 8, 24, 42].

However, this only reveals one part of the security landscape, since Actions can also be developed in other ways. Firstly, so-called *Docker Actions* execute tasks that are specified within a Docker image and its associated components. We found 9,507 of such Actions (i.e., 31%) in our dataset. Secondly, so-called *Composite Actions* are reusable workflow components that are essentially constructed

[19]https://cwe.mitre.org/data/definitions/20.html
[20]https://codeql.github.com/codeql-query-help/javascript/js-regex-missing-regexp-anchor

using steps that resemble those found in regular GitHub workflows. We found 4,908 Composite Actions (i.e., 16%) in our dataset.

Similar to our study in RQ3 of the impact on JS Actions of security vulnerabilities in the npm ecosystem, it would be very insightful to understand to which extent vulnerabilities in Docker images could compromise the security of the Actions relying on them. For instance, Docker Hub, a prominent and widely used Docker registry, serves as the primary registry for acquiring Docker images used in Docker Actions [27]. Yet, research indicated that Docker images available on Docker Hub rely on numerous packages with known vulnerabilities [25, 34, 43].

Composite Actions are also worthy of study, since they have started to gain popularity due to their reusability, customisability, and extensibility [27]. Their use is likely to increase the attack surface of security vulnerabilities even further, and make the dependency network and security propagation problem more complex. Indeed, it now becomes possible for Actions to depend on Composite Actions that can be reused across multiple workflows, and these composite Actions may themselves further rely on either JS Actions, Docker Actions or other Composite Actions. As such, any security issues that occur somewhere in the dependency chain will have a higher potential to make more workflows vulnerable.

**Summary:** GitHub repositories can be exposed to vulnerabilities through transitive dependencies of its workflows, be it through JS Actions, Docker Actions or Composite Actions. Analysing and hardening the security of each of these automation components is needed to reduce the attack surface.

## 8.2 Exposure of GitHub repositories

We reported in RQ3 that the vast majority (i.e., 71%) of the latest releases of JS Actions depend on at least one vulnerable npm package release, and that the code of more than 54.4% of JS Actions was affected by at least one security weakness. This exposure to security flaws is worrisome, because these problems have the potential to extend to the workflows that rely on such potentially vulnerable Actions. This could pose a threat to the software repositories in which these workflows are being executed, and even to all applications being developed and deployed in these repositories. Therefore, it is important to quantify the number of repositories that are using potentially vulnerable JS Actions.

In our vulnerability analysis we observed that out of 8,107 JS Action releases, 3,147 (i.e., 38.8%) were simultaneously exposed to at least one vulnerability in their dependency network and at least one security weakness in their code. We also observed that (workflows in) repositories heavily rely on some of these 3,147 JS Actions. According to the information found in the Insights tab of each GitHub repository, at the time of writing more than 1.5M repositories have workflows that depend on the actions/setup-node Action, which is exposed to 5 vulnerabilities in its dependency network and 4 security weaknesses in its code. Likewise, workflows of over 0.8M repositories rely on actions/setup-python which is exposed to 4 vulnerabilities in its dependency network and affected by 4 code security weaknesses. These two concrete examples only scratch the surface of the extent to which GitHub repositories, through their associated workflows, depend on potentially vulnerable Actions.

**Summary:** A huge amount of GitHub repositories are potentially exposed to security issues in their associated workflows.

## 8.3 Other security risks of workflows

The problem is even considerably worse, since workflows in GitHub repositories can suffer from many other kinds of security risks. The popularity of GitHub Actions has attracted hackers that see new opportunities to exploit security flaws in GitHub repositories.

For instance, when a workflow is initiated, GitHub creates a temporary token generated exclusively for that workflow to interact with the repository. If malicious actors gain access to this token, it will grant them permissions equivalent to those required to execute code within a workflow.[21] They may gain access to this token through compromised Actions and methods like command injection and secret dumping.[22]. They could exploit the information that workflows receive about the event that triggered them, including details like issue titles, pull request titles, and commit messages. Since attackers can manipulate some of these details, workflow developers and maintainers need to be cautious and treat them as potentially risky input. Workflows that use this untrusted input in bash commands can fall victim to command injection vulnerabilities, exposing the secrets.

Dumping secrets from a GitHub runner's memory is another way to steal a maintainer's access token or credentials. When a workflow job starts, all the secrets required for the job are received by the GitHub Actions runner. Since the secrets are received by the runner at the beginning of the job, the runner's memory can be manipulated to expose all the secrets specified in the job, even before they are used. This implies that by compromising an Action used in the job, all secrets referenced in the job can be read from memory. Therefore, secrets need to be secured by encrypting them within the organisation, repository, or environment settings. This threat can be mitigated by setting up the GitHub secret scanning tool[23] and receiving alerts for exposed secrets.

Moreover, it is crucial to adhere to GitHub's recommendations when reusing Actions in workflows in the form of owner/repo@ref. The ref key can either reference a branch (e.g., @main), a tag (e.g., @v1) or a commit hash (e.g., @8f4b7848644...). It is the latter choice, often referred to as *Action pinning*, that is recommended by GitHub to harden the security of using an Action.[24] It ensures that the Action's version remains consistent, reducing the risk of supply chain attacks. By referring to a specific commit of an Action through its designated hash, confidence can be gained in reusing the exact version that has been previously reviewed and approved. Referring to an Action's commit hash also aligns with one of the top 10 OWASP CI/CD security risks[25], namely CICD-SEC-9 that concerns the issue of Improper Artifact Integrity Validation. This measure guarantees data integrity throughout the entire pipeline. Even if malicious actors would succeed in compromising an Action by

[21] https://www.paloaltonetworks.com/blog/prisma-cloud/github-actions-opt-out-permissions-model
[22] https://www.paloaltonetworks.com/blog/prisma-cloud/github-actions-worm-dependencies
[23] https://docs.github.com/en/code-security/secret-scanning
[24] https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions
[25] https://owasp.org/www-project-top-10-ci-cd-security-risks

inserting malicious code into it, it would not impact any workflows that are using the still secure previous Action versions if they are referred to through a commit hash.

**Summary:** Beyond security vulnerabilities in the code of workflows and their dependencies, workflows can get compromised in many other ways, such as improper access through identity theft, artifact integrity risks, or exposed tokens, credentials and other secrets. Security hardening recommendations such as prioritising Actions with verified creators, strict access control and permission settings, and the use of secret scanning tools help to reduce these risks.

### 8.4 Dependency and security monitoring tools

A wide range of dependency and security monitoring tools is available to allow repository maintainers keeping their dependencies up to date and reducing their exposure to known vulnerabilities. Well-known examples of such tools are GitHub's Dependabot and the CodeQL analysis that can be enabled in the settings of a GitHub repository. Such tools can help in assessing the transitive security exposure, facilitating a more precise analysis of the impact of security vulnerabilities within the dependency network of Actions.[26] Repository maintainers should consider integrating such tools into their development and maintenance processes to enhance the security of their software supply chain by proactively receiving security alerts for CVEs in their project dependencies.

In practice, however, many repositories do not take advantage of using such tools, possibly due to a lack of awareness or training, or perhaps because of the perception that it might lead to an increased maintenance effort. Even though Dependabot supports monitoring of GitHub Actions workflows, only a small fraction of GitHub repositories have been found to incorporate it for that purpose. Decan et al. reported that, out of over 22,000 GitHub repositories relying on workflows, only 5.0% of them had configured Dependabot [10].

**Summary:** There is a need to increase awareness of using dependency and security monitoring tools proactively for securing workflows.

## 9 THREATS TO VALIDITY

We report on the four types of validity threats suggested by Wohlin et al. [41] pertaining to our research.

**Construct validity** discusses the connection between the theory behind the experiment and the observed results. A first threat of this type concerns the fact that we relied on GitHub's release management feature to identify Action releases. However, not all Action repositories use this feature to distribute different versions of an Action; some may have used alternative methods such as tags, git commits, or branches. We have excluded these Actions from our dataset since there is no precise means to identify which git tags, commits or branches should be considered as actual releases.
Another threat to construct validity arises from the fact that we only considered dependencies for JS Actions as specified in their *package.json* and *package-lock.json* manifest files. While this way of studying package dependencies is widely accepted within the research domain, we cannot claim that the resulting set of dependencies is complete, since there may have been other (often more informal) ways of specifying dependencies (such as textually in the repository's README file). Conversely, even when a manifest file specifies a package as a dependency, this dependency may not necessarily be actively used by the Actions. Such so-called "bloated" dependencies refer to third-party libraries packaged within the application binary that are not essential for the application's functionality [36]. Including such bloated dependencies might have inflated the reported number of JS Actions that are potentially exposed to vulnerabilities in npm packages.
A final threat to construct validity stems from the use of CodeQL queries to identify security weaknesses in JS Action code. Even though this represented the most complete list of security weaknesses we could find, it may not encompass the full spectrum of possible security weaknesses and Common Weakness Enumerations (CWEs) in JavaScript code. As a consequence, the findings for RQ4 might underestimate the true extent of security weaknesses affecting JS Actions.

**External validity** concerns the extent to which the findings can be generalised or extended beyond the specific research boundaries. The restriction of the analysis to JS Actions implies that the findings cannot be generalised to Docker Actions and Composite Actions. Also, the restriction of the analysis to GitHub repositories prevents us from generalising the results to other platforms that started supporting Actions, such as Gitea's self-hosted git service.

**Internal validity** addresses the decisions and factors internal to the study that have the potential to impact the observations. One such threat pertains to the use of the OSV vulnerability database for linking npm package releases to known vulnerabilities. Like any other vulnerability database, there is no guarantee that it contains all existing known vulnerabilities. As a consequence, the findings for RQ3 could underestimate the actual exposure of JS Actions to vulnerabilities in their dependency network.

**Conclusion validity** pertains to the extent to which the conclusions drawn from the study are reasonable. To assess whether a JS Action is affected by a vulnerability in a (direct or indirect) dependency, we adopted a conservative approach, considering the Action to be potentially vulnerable if a vulnerable transitive dependency was present. This is likely to be an overestimation, since relying on on a vulnerable package release does not necessarily make the Action vulnerable; the outcome depends on the specific nature of the vulnerability, how it manifests itself, and whether it can actually propagate through the dependency chain. Therefore, we used the terms "potentially vulnerable" and "exposed" (instead of, e.g., "affected" or "compromised") throughout the paper to reflect this uncertainty. To assess to which extent an exposed JS Action is truly affected by a vulnerability in one of its dependencies, more resource-intensive methods would be required such as code-centric call graph analysis and usage-based analysis [29–31]. This would be very challenging given the size of our dataset, composed of thousands of JS Actions, tens of thousands of package releases, and hundreds of thousands of dependencies. Moreover, even a code-centric analysis would not be sufficient. Just because a vulnerable piece of code is called in a dependent project does not necessarily mean that the corresponding vulnerability can be exploited. Only a

[26] https://github.blog/2023-01-19-unlocking-security-updates-for-transitive-dependencies-with-npm

rigorous and partly manual analysis of attack vectors would allow to assess this, but this is impractical at the considered ecosystem scale. As a scalable alternative we relied on GitHub's CodeQL for automated scanning of code weaknesses.

## 10 CONCLUSION

Github Actions, the popular workflow automation tool integrated into the GitHub social coding platform, relies on reusable building blocks called Actions. The majority of them are developed in JavaScript and depend on packages distributed through the npm package manager. Security issues affecting reusable JS Actions have the potential to introduce vulnerabilities in the workflows that use them, and consequently, in the repositories in which these workflows are used, and in the applications developed in these repositories, thus affecting the security of the software supply chain.

The objective of this study was to quantify vulnerable dependencies within the dependency networks of JS Actions, as well as security weaknesses in the Actions' JavaScript code. To achieve this, we analysed 8,107 JS Actions over a period of 3,5 years. 95% of these Actions were found to depend on npm packages, and over 65% of these Actions even had indirect dependencies up to a depth of 5. Moreover, the reuse was concentrated in few packages: only 20% of the packages were required to cover the dependencies of over 93% of Actions.

With respect to security vulnerabilities, over 77% of the Action releases depended on at least one vulnerable npm package release. We identified single points of failure in earlier releases of specific widely used packages, corresponding to vulnerabilities that potentially exposed a significant portion of JS Actions to these vulnerabilities. We also observed that indirect dependencies are more likely to suffer from high and critical vulnerabilities, underscoring the importance of security monitoring for such indirect dependencies.

With respect to security weaknesses in JS Action code, the majority of JS Actions (54.4%) was found to be affected by at least one weakness. Considering the top ten most frequent weakness types, seven of them were found to be associated with CWE-20 known as *Improper Input Validation*. Addressing this CWE therefore has the potential to result in a fivefold improvement in the security of JS Actions code. Putting everything together, 38.8% of all considered Actions were found to be simultaneously exposed to at least one vulnerable dependency and at least one code security weakness. Moreover, many repositories —by means of their automated workflows— rely on these potentially vulnerable Actions.

All these findings urge for the need for workflow maintainers and Action providers to pay more attention to security issues in the GitHub Actions ecosystem. Going even one step further, beyond the scope of the current analysis, these is a need to focus on a wider range of security threats that GitHub repositories and their workflows may be susceptible to. Well-known examples are the theft of repository access tokens or credentials through methods like command injection and secret dumping. This highlights the importance of raising awareness and proactively using security monitoring tools (e.g., CodeQL, Dependabot, and secret scanning tools). These measures are crucial for hardening the security of workflows and repositories and call for additional research and more advanced security analysis tools within and beyond GitHub.

## ACKNOWLEDGMENTS

This research is supported by the Fonds de la Recherche Scientifique - FNRS under grant numbers T.0149.22, F.4515.23 and J.0147.24.

## REFERENCES

[1] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. 2016. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *Symp. Security and Privacy*. IEEE, 289–305. https://doi.org/10.1109/SP.2016.25

[2] M. Alfadel, D. E. Costa, and E. Shihab. 2021. Empirical Analysis of Security Vulnerabilities in Python Packages. In *Int'l Conf. Software Analysis, Evolution and Reengineering*. https://doi.org/10.1109/saner50967.2021.00048

[3] F. Angermeir, M. Voggenreiter, F. Moyón, and D. Méndez. 2021. Enterprise-Driven Open Source Software: A Case Study on Security Automation. *Int'l Conf. Software Engineering: Software Engineering in Practice* (2021), 278–287. https://doi.org/10.1109/ICSE-SEIP52600.2021.00037

[4] G. Benedetti, L. Verderame, and A. Merlo. 2022. Automatic Security Assessment of GitHub Actions Workflows. In *Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. ACM, 37–45. https://doi.org/10.1145/3560835.3564554

[5] M. Chen, F. Fischer, N. Meng, X. Wang, and J. Grossklags. 2019. How Reliable is the Crowdsourced Knowledge of Security Implementation?. In *Int'l Conf. Software Engineering*. 536–547. https://doi.org/10.1109/ICSE.2019.00065

[6] T. Chen, Y. Zhang, S. Chen, T. Wang, and Y. Wu. 2021. Let's Supercharge the Workflows: An Empirical Study of GitHub Actions. In *Int'l Conf. Software Quality, Reliability and Security Companion*. IEEE. https://doi.org/10.1109/QRS-C55045.2021.00163

[7] E. Constantinou and T. Mens. 2017. An empirical comparison of developer retention in the RubyGems and npm software ecosystems. *Innovations in Systems and Software Engineering* 13, 101 (2017). https://doi.org/10.1007/s11334-017-0303-4

[8] A. Decan, T. Mens, and E. Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Int'l Conf. Mining Software Repositories*. 181–191. https://doi.org/10.1145/3196398.3196401

[9] A. Decan, T. Mens, and P. Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empir. Softw. Eng.* 24, 1 (2019), 381–416. https://doi.org/10.1007/s10664-017-9589-y

[10] A. Decan, T. Mens, and H. Onsori Delicheh. 2023. On the outdatedness of workflows in the GitHub Actions ecosystem. *J. Systems and Software* 206 (2023). https://doi.org/10.1016/j.jss.2023.111827

[11] A. Decan, T. Mens, P. Rostami Mazrae, and M. Golzadeh. 2022. On the Use of GitHub Actions in Software Development Repositories. In *Int'l Conf. Software Maintenance and Evolution*. IEEE. https://doi.org/10.1109/ICSME55016.2022.00029

[12] J. Dietrich, S. Rasheed, and A. Jordan. 2023. *On the Security Blind Spots of Software Composition Analysis*. Technical Report. https://doi.org/10.48550/arXiv.2306.05534

[13] W. Enck and L. Williams. 2022. Top Five Challenges in Software Supply Chain Security: Observations From 30 Industry and Government Organizations. *IEEE Security and Privacy* 20, 2 (2022), 96–100. https://doi.org/10.1109/MSEC.2022.3142338

[14] W. B. Frakes and K. C. Kang. 2005. Software reuse research: status and future. *Trans. Softw. Eng.* 31 (2005), 529–536. https://doi.org/10.1109/TSE.2005.85

[15] M. Golzadeh, A. Decan, and T. Mens. 2021. On the rise and fall of CI services in GitHub. In *Int'l Conf. Software Analysis, Evolution and Reengineering*. IEEE. https://doi.org/10.1109/SANER53432.2022.00084

[16] Y. Gu, L. Ying, H. Chai, C. Qiao, H. Duan, and X. Gao. 2023. Continuous Intrusion: Characterizing the Security of Continuous Integration Services. In *Symposium on Security and Privacy*. IEEE, 1561–1577. https://doi.org/10.1109/SP46215.2023.10179471

[17] J. Hejderup, M. Beller, K. Triantafyllou, and G. Gousios. 2022. Präzi: from package-based to call-based dependency networks. *Emp. Softw. Eng.* 27, 5 (2022), 102. https://doi.org/10.1007/s10664-021-10071-9

[18] R. Hiesgen, M. Nawrocki, T. C. Schmidt, and M. Wählisch. 2022. The Race to the Vulnerable: Measuring the Log4j Shell Incident. *ArXiv* abs/2205.02544 (2022). https://doi.org/10.48550/arXiv.2205.02544

[19] N. Imtiaz, S. Thorn, and L. A. Williams. 2021. A comparative study of vulnerability reporting by software composition analysis tools. *Int'l Symp. Empirical Software Engineering and Measurement* (2021). https://doi.org/10.1145/3475716.3475769

[20] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl. 2017. Structure and Evolution of Package Dependency Networks. In *Int'l Conf. Mining Software Repositories*. 102–112. https://doi.org/10.1109/MSR.2017.55

[21] T. Kinsman, M. Wessel, M. A. Gerosa, and C. Treude. 2021. How do software developers use GitHub Actions to automate their workflows?. In *Int'l Conf. Mining Software Repositories*. https://doi.org/10.1109/MSR52588.2021.00054

[22] I. Koishybayev, A. Nahapetyan, R. Zachariah, S. Muralee, B. Reaves, A. Kapravelos, and A. Machiry. 2022. Characterizing the Security of Github CI Workflows. In *USENIX Security Symposium*.

[23] T. Lauinger, A. Chaabane, and C. B. Wilson. 2018. Thou shalt not depend on me. *Comm. ACM* 61, 6 (2018), 41–47. https://doi.org/10.1145/3190562

[24] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng. 2022. Demystifying the Vulnerability Propagation and Its Evolution via Dependency Trees in the NPM Ecosystem. In *Int'l Conf. Software Engineering*. 672–684. https://doi.org/10.1145/3510003.3510142

[25] P. Liu, S. Ji, L. Fu, K Lu, X. Zhang, W.-H. Lee, T. Lu, W. Chen, and R. A. Beyah. 2020. Understanding the Security Risks of Docker Hub. In *European Symp. Research in Computer Security*. https://doi.org/10.1007/978-3-030-58951-6_13

[26] J. Luszcz. 2018. Apache Struts 2: how technical and development gaps caused the Equifax breach. *Network Security* 2018, 1 (2018), 5–8. https://doi.org/10.1016/S1353-4858(18)30005-9

[27] H. Onsori Delicheh, A. Decan, and T. Mens. 2023. A Preliminary Study of GitHub Actions Dependencies. *CEUR Workshop Proceedings* 3483 (2023), 66–77.

[28] D. Pashchenko, I.and Vu and F. Massacci. 2020. A Qualitative Study of Dependency Management and Its Security Implications. *SIGSAC Conf. Computer and Communications Security* (2020), 1513–1531. https://doi.org/10.1145/3372297.3417232

[29] H. Plate, S. E. Ponta, and A. Sabetta. 2015. Impact assessment for vulnerabilities in open-source software libraries. In *Int'l Conf. Software Maintenance and Evolution*. 411–420. https://doi.org/10.1109/ICSM.2015.7332492

[30] H. Ponta, S. E.and Plate and A. Sabetta. 2020. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empir. Softw. Eng.* 25 (2020), 3175 – 3215. https://doi.org/10.1007/s10664-020-09830-x

[31] S. E. Ponta, H. Plate, and A. Sabetta. 2018. Beyond Metadata: Code-Centric and Usage-Based Analysis of Known Vulnerabilities in Open-Source Software. In *Int'l Conf. Software Maintenance and Evolution*. IEEE, 449–460. https://doi.org/10.1109/ICSME.2018.00054

[32] P. Rostami Mazrae, T. Mens, M. Golzadeh, and A. Decan. 2023. On the usage, co-usage and migration of CI/CD tools: A qualitative analysis. *Empir. Softw. Eng.* 28, 2 (2023), 52. https://doi.org/10.1007/s10664-022-10285-5

[33] S. G. Saroar and M. Nayebi. 2023. Developers' Perception of GitHub Actions: A Survey Analysis. In *Int'l Conf. Evaluation and Assessment in Software Engineering*. https://doi.org/10.1145/3593434.3593475

[34] R. Shu, X. Gu, and W. Enck. 2017. A Study of Security Vulnerabilities on Docker Hub. In *Conference on Data and Application Security and Privacy*. ACM. https://doi.org/10.1145/3029806.3029832

[35] Snyk. 2022. State of Open Source Security 2022. https://snyk.io/reports/open-source-security/. [Online; accessed on September 1, 2023].

[36] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry. 2021. A comprehensive study of bloated dependencies in the Maven ecosystem. *Empir. Softw. Eng.* 26, 3 (2021), 45. https://doi.org/10.1007/s10664-020-09914-8

[37] D. Stefanović, D. Nikolic, D. Dakic, I. Spasojević, and S. Ristić. 2020. Static Code Analysis Tools: A Systematic Literature Review. In *Int'l Symp. on Intelligent Manifacturing and Automation*. DAAAM International. https://doi.org/10.2507/31st.daaam.proceedings.078

[38] H. H. Thompson. 2003. Why Security Testing Is Hard. *IEEE Securiy and Privacy* 1, 4 (2003), 83–86. https://doi.org/10.1109/MSECP.2003.1219078

[39] L. Williams. 2022. Trusting Trust: Humans in the Software Supply Chain Loop. *IEEE Security and Privacy* 20, 5 (2022), 7–10. https://doi.org/10.1109/MSEC.2022.3173123

[40] E. Wittern, P. Suter, and S. Rajagopalan. 2016. A Look at the Dynamics of the JavaScript Package Ecosystem. *Working Conf. Mining Software Repositories* (2016), 351–361. https://doi.org/10.1145/2901739.2901743

[41] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. 2012. *Experimentation in Software Engineering*. Springer.

[42] A. Zerouali, T. Mens, A. Decan, and C. De Roover. 2022. On the impact of security vulnerabilities in the npm and RubyGems dependency networks. *Empir. Softw. Eng.* 27, 5 (2022), 1–45. https://doi.org/10.1007/s10664-022-10154-1

[43] A. Zerouali, T. Mens, A. Decan, J. M. Gonzalez-Barahona, and G. Robles. 2021. A multi-dimensional analysis of technical lag in Debian-based Docker images. *Empir. Softw. Eng.* 26 (2021). https://doi.org/10.1007/s10664-020-09908-6