EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

Eindhoven University of Technology

MASTER

On Resolution of Vulnerable Dependencies with Dependabot Security Updates in JavaScript Projects

Agaronian, Andrei

Award date:
2021

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Department of Mathematics and Computer Science
Software Engineering and Technology Research Group

On Resolution of Vulnerable Dependencies with Dependabot Security Updates in JavaScript Projects

Master Thesis

Andrei Agaronian

Graduation Supervisor:
dr. Eleni Constantinou

Co-supervisors:
dr. Alexander Serebrenik
dr. Nicola Zannone

– Version 1.0–

Eindhoven, September 2021

Abstract

The utilization of third-party packages is a consolidated practice in software engineering that significantly alleviates the development. Nevertheless, introducing dependencies into software projects is also a source of security concerns. Third-party packages may contain security vulnerabilities that propagate to the dependent applications through code reuse. To counter this, maintainers of dependent projects should monitor their dependencies and security reports to ensure that only patched releases of the upstream applications are in use. Unfortunately, previous studies demonstrate that developers fail to maintain their dependencies secure and up-to-date. As such, recent years have seen several software bots designed to assist developers in this task and support the rapid identification and resolution of vulnerable dependencies. One of such bots, Dependabot, automatically notifies the developers of a vulnerability in their dependencies and generates a pull request that remediates it - a security update.

However, no studies investigate the developer interaction with the security updates and the level of Dependabot adoption by the community. In this work, we address this gap by combining quantitative and qualitative techniques to analyse security updates in mature and actively maintained JavaScript projects. Our findings show that the task of fixing vulnerable dependencies is, to a large extent, delegated to Dependabot and that developers merge the majority of security updates within several days. On the other hand, when developers do not merge a security update, they usually address the identified vulnerability manually, which often takes up to several months. Based on this finding, we encourage developers to perform manual security fixes more rapidly. Furthermore, we discover many problems the users experience when using the bot. We analyse the findings and formulate five recommendations for Dependabot maintainers to elevate user satisfaction, urging them to provide better means for the developers to familiarise themselves with the bot and increase its configurability.

Preface

I would like to thank dr. Eleni Constantinou, dr. Alexander Serebrenik, and dr. Nicola Zannone for their guidance, advice, feedback, and collaboration during this project. I would also like to thank dr. Alexander Serebrenik and dr. Eleni Constantinou one more time for the opportunities they have provided me throughout my Master's study. Working on these projects was a great experience, and I have learned more than I ever did before.

I would like to thank my parents for always pushing me forward and helping me with the means for my studies and living. Thank you, Father, for being a role model to me. Thank you, Mother, for your care and support.

Finally, I would like to thank my dear friend and fellow student, Thorn, for our friendship and collaboration. I had a very hard time adapting to the new environment, subconsciously refusing everything and everyone, but despite this, you were always kind to me, inviting me to study or spend time together.

Contents

Contents vii

List of Figures xi

List of Tables xiii

1 Introduction 1

2 Background 3
2.1 Dependencies in JavaScript projects . . . 3
2.1.1 Package managers . . . 3
2.1.2 Manifest file . . . 4
2.1.3 Lock file . . . 4
2.1.3.1 package-lock.json . . . 5
2.1.3.2 yarn.lock . . . 6
2.2 Dependabot . . . 6
2.2.1 GitHub Advisory Database . . . 6
2.2.2 Security alerts . . . 7
2.2.3 Security updates . . . 7
2.2.4 Version updates . . . 8

3 Related work 9
3.1 Vulnerabilities and Dependency Management . . . 9
3.2 Software Bots . . . 10

4 Methodology 12
4.1 Data Collection . . . 12
4.1.1 Collection period . . . 12
4.1.2 Project selection . . . 13
4.1.3 Security Advisory Records . . . 15
4.1.4 Detecting Vulnerabilities in Dependencies . . . 16
4.1.4.1 Deducted Algorithm . . . 17
4.1.4.2 Validation . . . 19
4.1.5 Fixing Cases Discovery . . . 21
4.2 Data Analysis . . . 22
4.2.1 Addressing RQ_1 . . . 23
4.2.2 Addressing RQ_2 . . . 24
4.2.3 Addressing RQ_3 . . . 24
4.2.4 Addressing RQ_4 . . . 25

5 Results **27**
5.1 How often do developers merge Dependabot security updates? 27
5.2 How frequently do developers fix a vulnerable dependency manually in the presence of the Dependabot security update? 28
5.3 Why do developers ignore the suggestion of Dependabot or decide to address the vulnerability manually? 30
5.4 How long does it take to address a vulnerable dependency identified by Dependabot? 31

6 Discussion **34**
6.1 Implications to Practitioners 34
6.2 Implications to Dependabot Maintainers 35
6.3 Implications to Researchers 36
6.4 Future work 36
6.5 Threats to Validity 37

7 Conclusion **39**

Bibliography **41**

Appendix **49**

A Rectification of the security advisory records **49**
A.1 minimist and acorn 49
A.1.1 Summary 49
A.1.2 Evidence 49
A.1.3 Remediation 49
A.2 concat-with-sourcemaps 50
A.2.1 Summary 50
A.2.2 Evidence 50
A.2.3 Remediation 50
A.3 eslint 50
A.3.1 Summary 50
A.3.2 Evidence 50
A.3.3 Remediation 50
A.4 ecstatic 51
A.4.1 Summary 51
A.4.2 Evidence 51
A.4.3 Remediation 51
A.5 ws 51
A.5.1 Summary 51
A.5.2 Evidence 51
A.5.3 Remediation 51
A.6 kind-of 52
A.6.1 Summary 52
A.6.2 Evidence 52
A.6.3 Remediation 52
A.7 marked 52
A.7.1 Summary 52
A.7.2 Evidence 52
A.7.3 Remediation 52

B Rater guideline **53**
B.1 First stage . . . 53
B.1.1 Susceptibility to a vulnerability . . . 53
B.1.2 Fixing commit . . . 54
B.1.3 Candidate commits . . . 54
B.1.4 Provided information . . . 54
B.2 Second stage . . . 55
B.2.1 Column A . . . 55
B.2.2 Column B . . . 55
B.2.3 Column C . . . 55
B.2.4 Column D . . . 57

C Additional Figures **59**

List of Figures

2.1 Screenshot of a security update. 7

4.1 Structure of a security update title. 16
4.2 Decision tree routine for a manifest file; $[x, y]$ stands for the range defined by the dependency constraint, $[z, k]$ defines the range of the vulnerable releases, and finally, p is the first patched release. 17
4.3 Screenshot of a security alert. 20
4.4 Candidate and fixing commits scenarios. 22
4.5 Distribution of the number of security updates. 23

5.1 Violin plots for the distributions of the merge ratios across the four groups of projects. 27
5.2 Survival curve for the event “vulnerability identified by Dependabot is addressed”. 32
5.3 Survival curves for the event “vulnerability identified by Dependabot is addressed” based on the severity level. 33
5.4 Violin plots for the distributions of the bot and manual fixing times. 33

B.1 Candidate and fixing commits scenarios 55

C.1 Summary for ***RQ3*** 60

List of Tables

2.1 Semantic oriented constraints. 4

4.1 Assessment of the classifier performance on the different sets of dimensions. 14
4.2 Resource path resolution for the set of identified bots. 14
4.3 Summary of the collected security updates. 15
4.4 Characteristics of the selected projects. 15
4.5 Project classification based on the number of security updates. 23

5.1 The percentages of human/bot addressed and non-resolved vulnerabilities per project group. 28
5.2 Computed p-values for the pairwise comparisons between the project groups. Significance: '***' < 0.001, '**' < 0.01, '*' < 0.05. Blue cells highlight the cases when $p < 0.05$. 29
5.3 Survival probability P_S of a vulnerability identified by Dependabot. 32
5.4 Survival probability P_S of a vulnerability identified by Dependabot based on its severity. 32

B.1 Column B - Labels. 56
B.2 Column C - Labels 57
B.3 Column D - Labels. 57

Chapter 1

Introduction

Modern open-source software is increasingly developed and deployed in highly interdependent environments, relying on reusable software packages that are distributed through online registries, each targeting a particular programming language (*e.g.,* `npm`, `RubyGems`, `Maven`, and `CRAN`). Indeed, software reuse is a well-established practice which facilitates the development of complex systems [52]. By reducing costs, efforts, and delivery times, it also contributes to rapid software evolution.

However, while providing benefits, utilization of third-party libraries may introduce security concerns. The reason is that a downstream application that depends on reusable software packages not only inherits their functionality but also security vulnerabilities [79]. Although reusable packages are as well subjects to constant evolution, resulting in mitigation of known vulnerabilities and bugs with newer releases, Kula *et al.* [65] concluded that developers are reluctant to update stale and vulnerable dependencies, perceiving dependency management to be extra workload and responsibility. In fact, after interviewing the developers of open-source projects with known vulnerable dependencies, Kula et al. report that 69% of the interviewees were simply unaware of them.

The problem of vulnerable dependencies does not remain overlooked, and one recommended measure to address it is to facilitate dependency management through the use of automation [44]. As such, recent years have seen several software bots [9,10,14,16,17] designed to monitor releases and/or security reports to identify stale and/or vulnerable dependencies, and in response, generate pull requests to update them. GitHub, the largest host of code in the world [54], has also acknowledged the problem, and since October 2017 [12], started to monitor the dependency graphs of the hosted dependent projects and notify the developers when a vulnerability is detected in one of the dependencies, suggesting known fixes to resolve it. Taking it a step further, on May 2019 [11], GitHub has acquired one the most popular dependency management bots, *Dependabot-preview* [9], resulting in a new natively integrated service, *Dependabot security updates* [2]. In turn, GitHub's own security notification service was renamed to *Dependabot security alerts*. When developers receive an alert, Dependabot automatically opens a pull request, *i.e.*, security update, to upgrade the dependency to the minimum required non-vulnerable version.

Being natively integrated into GitHub and distributed completely free of charge, irrespective of project visibility and team size, Dependabot security updates hold the lead as the most accessible service that provides automated pull-requests for remediation of vulnerable dependencies. Moreover, in less than two years since its introduction, GitHub's Dependabot has generated more than 23M pull requests, suggesting that it is the most widely used dependency management bot. However, no work investigates the usage of Dependabot security updates and their role in fixing vulnerable dependencies in software projects. To address this gap, we analyse 4,538 security updates associated with 1,004 mature and actively maintained JavaScript projects hosted on GitHub. First, we ask:

RQ*$_1$*: *How often do developers merge Dependabot security updates?*

The interaction intended by the design of the bot suggests that in response to a generated security update, the recipient developers should merge it. As such, we analyse the distribution of merge rates across the considered projects to get an understanding of how frequently project maintainers adhere to this practice. We find that regardless of the number of security updates received, the projects' maintainers tend to either accept and merge every suggestion of Dependabot or none of them.

As suggested by the findings of the previous studies [23, 74], developers may prefer to close an automated pull request and implement the bot suggestions manually. Alternatively, developers may choose to eliminate the dependency on a vulnerable package altogether. Envisioning multiple scenarios in which a security update is rejected, but the identified vulnerability is nonetheless removed from the project, we ask:

RQ_2: *How frequently do developers fix a vulnerable dependency manually in the presence of the Dependabot security update?*

Using a semi-automated approach, for each vulnerability associated with a security update raised by Dependabot, we track the earliest commit in which the concerned vulnerability is resolved. Distributing the fixes between Dependabot and developers, we find that majority of the fixes are contributed by the bot. Furthermore, most of the vulnerabilities that are not addressed by merging a security update do not remain ignored and, eventually, get resolved manually.

Aiming to obtain further insights on what interaction challenges do developers face when interacting with Dependabot, and accordingly, how can user satisfaction with the bot get improved, we ask:

RQ_3: *Why do developers ignore the suggestion of Dependabot or decide to address the vulnerability manually?*

Manually investigating any textual artefacts associated with the security update or a fixing commit, we find 22 unique reasons to ignore a vulnerability in dependencies or mitigate it without the bot. Among the most frequent reasons, we outline the use of the third-party services for development and the risks of breaking changes.

Finally, we attempt to capture the degree to which the developers react to the identified vulnerabilities in a timely manner. Therefore, we formulate the following research question:

RQ_4: *How long does it take to address a vulnerable dependency identified by Dependabot?*

Taking into account that some vulnerabilities may remain unaddressed by the end of the observable period, to answer this question, we rely on the statistical technique of the survival analysis [20] and estimate the expected time duration until the remediation measures are taken. We find that, to a large extent, developers are proactive to address a vulnerability when a corresponding security update is generated. Nevertheless, this mostly concerns the fixes made with the bot, as they, on average, take one or two days. When developers decide to implement the fix manually, the observed statistic is not as optimistic.

The remainder of the paper is organised as follows. In Sections 2 and 3, we discuss the background and survey the related work. Following this, in Sections 4 and 5, we describe the methodology and report the results. Accordingly, in Section 6, we discuss the implications and present the threats to validity. At last, in Section 7, we conclude.

Chapter 2

Background

In this chapter, we cover the requisite background knowledge and concepts embedded in the foundation of this work. Specifically, the technologies and core definitions that pertain to dependency management in JavaScript projects (Section 2.1) and Dependabot and its services (Section 2.2).

2.1 Dependencies in JavaScript projects

To provide the context for the methodology exploited in this work, first, we discuss the package managers leveraged in JavaScript projects for dependency management (Section 2.1.1). Next, we explain the functionality, the working, and the structure of the two files used to capture JavaScript dependencies and control their installation. Namely, a manifest file (Section 2.1.2) and a lock file (Section 2.1.3).

2.1.1 Package managers

To ease the installation, upgrading, removal, and distribution of software packages, developers rely on package managers [18]. As of March 2021, there are two core package managers available for JavaScript software. The first one is `npm`[1], which is the most widely used package manager. It comprises two main components: (1) a remote registry, *i.e.*, an online database for publishing and fetching JavaScript packages, and (2) a command-line interface for users to interact with the registry and manage dependencies. In `npm` registry, package releases follow the versioning policy called *semantic versioning*[2], which uses a sequence of three digits `Major.Minor.Patch` to signal the compatibility of a change [29]. Under this scheme, version numbers and the way they change have a special meaning about what has been modified from one version to the next. An increase in the `Major` digit indicates breaking API changes, whereas changes to `Minor` and `Patch` version numbers are intended as backward compatible new features and bug fixes, respectively.

Another popular package manager is `yarn`[3], an alternative to `npm` released by Facebook [19], actively maintained and steadily growing in popularity to this date [13]. While remaining compatible with `npm` registry, `yarn` was developed to offer increased performance, security, and functionality for installation and resolution of dependencies. That is, `yarn` provides only a command-line interface but does not maintain its own registry, relying on the one of `npm` instead [15]. In our work, we do not restrict the scope to one particular package manager but consider the dependent projects that rely on either (or even both) of the two interfaces to manage dependencies.

[1] https://www.npmjs.com/. Last accessed March 10, 2021.
[2] https://semver.org/. Last accessed February 24, 2021.
[3] https://yarnpkg.com/. Last accessed March 10, 2021.

Table 2.1: Semantic oriented constraints.

Constraint	Translation
`1.0` or `1.0.x` or `~1.0.0`	Allow `Patch` release updates
`1` or `1.x` or `^1.0.0`	Allow `Minor` release updates
`x` or `*`	Allow `Major` release updates

2.1.2 Manifest file

All the dependent JavaScript projects contain a *manifest file*, `package.json`, that includes the relevant metadata, such as name, description, and release number. Most importantly, this file is used to specify the set of *direct* dependencies, *i.e.*, upstream packages referenced within the source code directly. Each dependency is captured through the name of the required upstream package mapped onto the *dependency constraint*, which explicitly states the range of the allowed releases, effectively excluding the versions that are deemed undesirable or incompatible. Such constraints can be used to specify a minimal (*e.g.*, $\geq$ 1.0.0) or a maximal (*e.g.*, $\leq$ 1.0.4) version of the dependency but can also be exploited for the semantic compatibility oriented ranges (see Table 2.1). Using specific notations, one can also express the combinations of constraints to declare a more complex restriction. By default, when a user runs the installation command, for every required dependency declared in the manifest, `npm` and `yarn` install the most recent release satisfying the constraint. This process is recursive, as every module on which the targeted project depends directly has dependencies of its own, resulting in *transitive* (a.k.a. indirect) dependencies.

Above, we emphasised that only required dependencies are installed. The reason is that there are four types of dependencies defined in `npm`. First, *runtime* dependencies, which are requisite for installation and execution of a project in the production environment. Next, there are *optional* dependencies that do not hamper the installation procedure if not found or cannot be retrieved. Additionally, there are *peer* dependencies, which are never fetched amid the execution of the installation command. Instead, the expectancy is to use the version already located in the root of the user folder hosting the deployed packages. In case it is not present or its version does not satisfy the constraint, a warning is raised. At last, there are *development* dependencies, required only during the development (i.e., testing). Accordingly, users can signal `npm` (with a flag `--production`) to ignore the development dependencies upon installation.

2.1.3 Lock file

Aside from a manifest file, it is also encouraged to commit a *lock file* into the source repository. This file is generated upon execution of the installation command on a manifest file and stores the exact calculated *dependency tree* along with the relevant metadata for each node (*e.g.*, integrity hash and the resource path in the registry). In a dependency tree, nodes represent software packages with an assigned release number determined upon installation, whereas edges capture the dependency relationship between them. A lock file that stores it is designed to be solely machine-processable rather than human-readable. Thus, contrary to a manifest file, a lock file is not intended to be modified by developers manually.

As discussed, in a manifest file, developers declare the allowed releases for direct dependencies, whereas transitive dependencies remain outside of developer control. Consequentially, when running the installation command twice at different times, two almost entirely different dependency trees may be produced. In turn, the purpose of a lock file is to lock down the versions of both direct and indirect dependencies, allowing the exact reproduction of the dependency tree across different users and systems. This allows maintainers of the project to ensure that other contributors and end-users leverage the application in an environment identical to the one in which it was

developed and tested.

Each of the two discussed package managers for JavaScript software supports a different implementation of a lock file. For `npm`, there are `package-lock.json` and `npm-shrinkwrap.json`. Both of the files have no differences in structure, yet, unlike the first, `npm-shrinkwrap.json` can be published not only to the source repository but also `npm` package registry. For `yarn`, in turn, there is `yarn.lock`. We cover the structure of both `package-lock.json` and `yarn.lock` in Sections 2.1.3.1 and 2.1.3.2, respectively.

2.1.3.1 package-lock.json

The `package-lock.json` file describes the dependency tree generated upon deployment as a JSON object. This lock file captures the exact structure and hierarchy of the automatically compiled folder containing the installed packages. Consider an excerpt of an arbitrary `package-lock.json` given by Listing 2.1.

```
1   "lodash": {
2     "version": "1.0.0",
3     "resolved": "https...",
4     "integrity": "sha512...",
5     "dev": true,
6     "requires": {
7       "kind-of": "^6.0.0"
8     }
9   }
10  "kind-of": {
11    "version": "6.0.1",
12    "resolved": "https...",
13    "integrity": "sha512...",
14    "dev": true
15  }
16  "mnim": {
17    "version": "0.1.4",
18    "resolved": "https...",
19    "integrity": "sha512...",
20    "dev": true,
21    "requires": {
22      "kind-of": "3.2.2"
23    },
24    "dependencies": {
25      "kind-of": {
26        "version": "3.2.2",
27        "resolved": "https...",
28        "integrity": "sha512...",
29        "dev": true
30      }
31    }
32  }
```

Listing 2.1: Excerpt of a `package-lock.json` file.

The first object (lines 1-9) corresponds to a node of the dependency graph representing the *lodash* package. As can be observed, on line 2 of the file excerpt, the version locked for this package is declared, *i.e.*, release “1.0.0”. Lines 3-4 specify the resource to download the package from, and an integrity hash used to determine whether the package contents have been tampered with since the author originally published them. Next, on line 5, it is identified whether the dependency on this package is of development kind or not. Finally, lines 6-8 specify the dependencies of this package, *i.e.*, the name of an upstream package and an associated dependency constraint. In the concerned case, the *lodash* package at version “1.0.0” is dependant on any release of the *kind-of* package at the sixth major version. Accordingly, the next object in the file excerpt (lines 10-15) represents the *kind-of* package node in the dependency graph, resolving it to version “6.0.1”.

Further examining the given excerpt of the `package-lock.json` file, one can observe that the *mnim* package at version ”0.1.4” in the dependency graph (lines 16-33) also requires the *kind-of* package. However, unlike the *lodash* package, the dependency constraint specifies that only version “3.2.2” of the *kind-of* package satisfies the requirements. Since version “6.0.1” of the *kind-of* package, deployed at the top level, does not satisfy the aforementioned requisite, the *mnim* package will contain its own local deployment of the *kind-of* package at version “3.2.2” (lines 25-32).

Under the described scheme, representations of the direct dependencies will always be defined at the top level of the JSON object. Whereas the representations of transitive dependencies can be defined either within the “dependencies” field inside another upstream package object or also at the top level. The latter scenario applies (1) when no direct dependency onto the upstream package concerned by a transitive dependency is declared or (2) when the version deployed for

the direct dependency satisfies the constraint of the transitive dependency, implying that a nested package will rely on the upstream placed as the top-level package.

2.1.3.2 yarn.lock

The `yarn.lock` file can be described as a collection of *dependency resolution blocks*, which dictate the version that will be installed for a certain combination of an upstream package and a dependency constraint in a given dependency tree. Consider an example of such a block given in Listing 2.2.

```
1  axios@0.18.1, axios@^0.18.0:
2    version "0.18.1"
3    resolved: "https...",
4    integrity: "sha512...",
5    dependencies:
6      follow-redirects "1.5.10"
7      is-buffer "^2.0.2"
```

Listing 2.2: Excerpt of a `yarn.lock` file.

The first core element of the block, the header, is defined at the first line of the listing. It is a set of package-constraint pairs separated by the "`@`" sign. Despite the syntax, the package name in each pair for a single block must be the same. That is, each block is dedicated to no more than a single package. However, a dependency constraint in each of the headers with identical package names is unique. Therefore, a header reads as a package and a collection of the dependency constraints. The logic behind this element is that a block to which a header belongs applies to every node in a dependency graph that corresponds to the package and one of the dependency constraints defined in it. Therefore, in the case of Listing 2.2, any instance of the *axios* package with a constraint "0.18.1" or "^0.18.0", regardless of its position in a dependency graph, will get resolved with version "0.18.1" (see line 2).

Similarly to `package-lock.json`, lines 3-4 specify the resource path and an integrity hash. Finally, lines 6-7 give a list of the dependencies for the specified release of the concerned package, *i.e.*, the axios package at version "0.18.1", retrieved from its own manifest file. Accordingly, for each dependency listed in a block, there is another block that contains it in a header, defining the locked version for this dependency.

2.2 Dependabot

GitHub's Dependabot is associated with four interconnected services. Specifically, *database of security advisories* (Section 2.2.1), *security alerts* (Section 2.2.2), *security updates* (Section 2.2.3), and *version updates* (Section 2.2.4). While our work focuses solely on Dependabot security updates, an explanation of the other three features is required to understand the identified bottlenecks in the methodology and our approach to address them.

2.2.1 GitHub Advisory Database

GitHub maintains a curated list of security vulnerabilities in software packages that belong to six different ecosystems, including `npm`. As of March 2021, GitHub exploits three main sources of information to construct its database: (1) the National Vulnerability Database[4], (2) the `npm` Security advisories database[5], and (3) the advisories reported by the project maintainers. Additionally, GitHub leverages a combination of human review and machine learning capabilities to detect vulnerabilities in public commits. Each advisory record includes the description of the concerned vulnerability, name of the package and the ecosystem it belongs to, affected and patched releases, and optional information (references, workarounds, credits). Additionally, each security

[4] https://nvd.nist.gov/. Last accessed February 21, 2021.
[5] https://www.npmjs.com/advisories. Last accessed February 21, 2021.

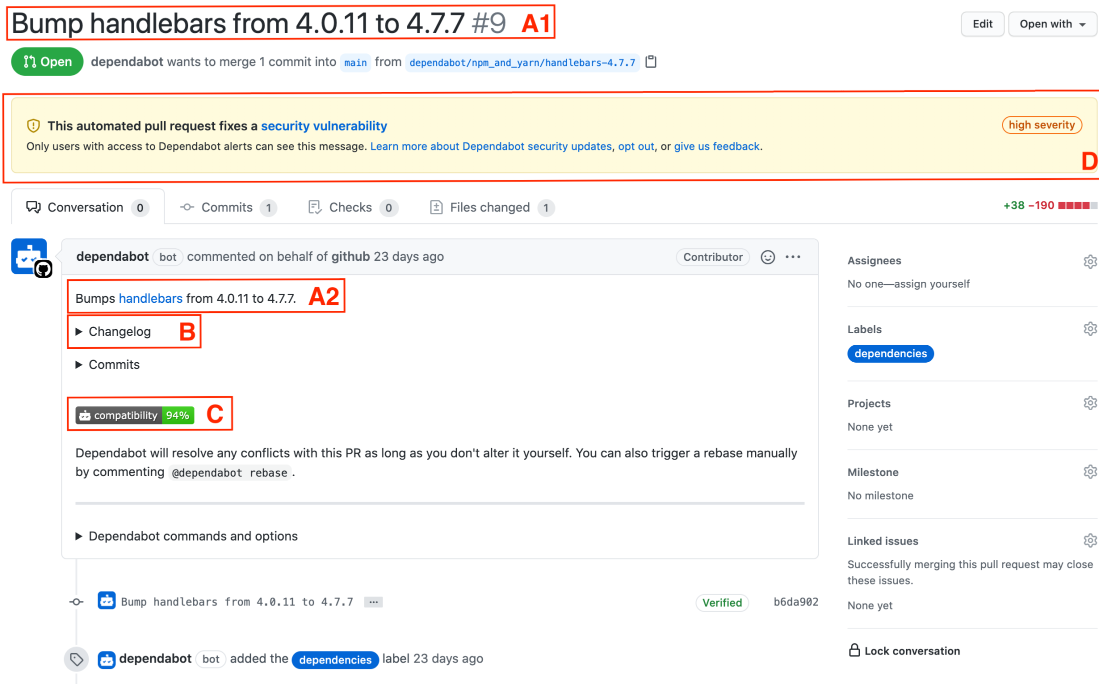


Figure 2.1: Screenshot of a security update.

advisory is assigned with a severity level, one of *low*, *moderate*, *high*, and *critical*. These are used to signal the extent of the risks incurred on the affected system, enabling the maintainers to prioritise the vulnerabilities.

2.2.2 Security alerts

In October 2017, GitHub introduced a new natively integrated service to the community, Dependabot security alerts. Upon any changes to the dependency files or the security advisory database of GitHub, Dependabot performs a scan of the dependency graph, and if a vulnerable version of a dependency is identified, raises a security alert. To perform the scan, Dependabot leverages the *dependency graph feature*, which reads the dependency files (both manifest and lock) located in the *default branch* of a repository, *i.e.*, the base branch for pull requests and commits, and uses them to construct an exact representation of the entire dependency tree. This implies that Dependabot is only capable to monitor and interact with the default branch.

A raised alert can be accessed at the dedicated section on the web page of a repository, visible only to the project owners and collaborators. Additionally, by default, Dependabot sends a notification to the users with the required permissions. An alert contains a link to the corresponding security record in the database and a summary describing the vulnerability, including its severity level, the file in which the vulnerable dependency is declared, and possible remediation measures if any. Dependabot raises the alerts in public repositories by default, but developers can opt-out.

2.2.3 Security updates

On May 23rd, 2019, GitHub announced the acquisition of Dependabot-preview, leading to a new natively integrated service, Dependabot security updates. When enabled, upon receiving a security alert, Dependabot constructs a pull request with the modification(s) to the dependency file(s) necessary to eliminate the identified vulnerability (if possible). It is worth noting that, in case of a vulnerable dependency, Dependabot always suggests an update to the minimum required non-vulnerable release of the upstream package, averse the most recent. An example of a security

update is presented in Figure 2.1. As can be observed at the top of the figure, the suggested change is displayed both in the title (A1 & A2) and body of the pull request. Additionally, Dependabot provides the changelog for each of the versions in between the current and the suggested release (B) and a badge that communicates the compatibility score (C). The latter is calculated dynamically based on a percentage of successful CI runs in other public repositories, where an identical security update has been instantiated. The information concerning the concerned vulnerability is not presented in the pull request itself. Instead, a yellow paned window is displayed to a user, signalling that the pull request concerns a security vulnerability (D). On the leftmost side, it provides a link to the corresponding security alert, and on the rightmost side, it displays the severity level of the vulnerability. This information, however, is not visible to the users without the required access rights, *i.e.*, external observers.

Although, it is expected that the developers respond to a generated security update manually, under certain circumstances, Dependabot can reject the corresponding pull request by itself. First, in case the dependency files are modified manually, updating the vulnerable dependency to a patched version, Dependabot closes the dedicated security update leaving a comment that the dependency has been upgraded and so the pull request is no longer needed (*e.g.*, [7]). Similarly, in case the dependency on the vulnerable package is removed entirely, Dependabot signals through a comment that the dependency has been removed and closes the pull request (*e.g.*, [5]). Another scenario concerns the case when after a security update has been generated, the dependency files are changed such that there is now a new dependency on the concerned vulnerable package imposed by a certain node in the dependency graph, whose constraint prevents the remediation (*e.g.*, [4]). In response, Dependabot leaves a comment that the dependency is no longer updatable and closes the pull request. Furthermore, in case a new vulnerability of the concerned upstream package is published, requiring an update to a more recent release, Dependabot supersedes the previous security pull request with a newer one, effectively closing the former (*e.g.*, [6]). At last, by commenting a command `@dependabot ignore this [patch|minor|major] version` or `@dependabot ignore this dependency` on a security update generated by Dependabot, developers trigger a corresponding action, which causes Dependabot to close the pull request (*e.g.*, [3]).

Concerning availability, Dependabot security updates is, generally, an opt-in service. However, provided that a project satisfies the given set of the prerequisites, such as (1) repository is public and not a fork, (2) contains a dependency file for a supported ecosystem, (3) does not already use an integration for dependency management, and (4) has security alerts enabled, the service of security updates is enabled automatically, with an option to opt-out.

2.2.4 Version updates

Continuing to transfer the functionality of the acquired tool, Dependabot-preview, on June 1st, 2020, GitHub has introduced another service, Dependabot version updates. Unlike security updates, version updates are automatically generated pull requests that aim at keeping the dependencies up-to-date. That is, Dependabot monitors the dependency graph of the project and the list of the releases for the upstream packages, and with a specified frequency, generates a pull request with an update in case a newer version is available. Both the security and the version updates can be enabled at a single repository simultaneously. However, to an external observer with no permissions, the two kinds of pull requests are indistinguishable. In fact, a version update can effectively turn into a security update, in case a recently published security advisory suggests that an unaddressed version update resolves a vulnerability in dependencies.

As of March 2021, Dependabot version updates remains in beta as an opt-in service. To enable it, the project maintainers are required to deploy a dedicated configuration file, `dependabot.yml`, in which developers specify, *e.g.*, the packages they want Dependabot to monitor, the schedule to generate the updates, the maximum amount of the simultaneously open automated pull requests, *etc.* Some of the configuration options may also affect the security updates.

Chapter 3

Related work

We discuss the relevant literature in two steps. First, in Section 3.1, we overview the studies on the propagation of vulnerabilities in dependency networks. These identify the widespread and non-negligible impact of this problem across numerous software projects while revealing the complexity of dependency management, perceived by the developers, which combined, explain the necessity of tools like Dependabot. In Section 3.2, we discuss the research on dependency management bots and describe the relationship between the prior studies and this work, effectively outlining the gaps in the knowledge that we aim to address.

3.1 Vulnerabilities and Dependency Management

A plethora of studies [22,26,35,42–44,46,65,82,90,91] conducted within the domain of the dependency networks conclude that updates in dependencies suffer from considerable time lags, sometimes even measured in the orders of years [67]. This phenomenon can be observed across various ecosystems, both decentralized, such as Android [26, 46], and centralized, such as Java [65, 82] and JavaScript [35, 43, 44, 90]. In turn, as concluded by Cox *et al.* [40], who studied the relationship between outdated dependencies and vulnerabilities in Java systems, reluctance to update ultimately leads to security issues [40].

In the attempt to gauge the problem, Decan *et al.* [44] examined the propagation of security vulnerabilities and their fixes in the dependency network for JavaScript software. Following their findings, over 133K projects, out of 610K considered, directly depend on a package with a known vulnerability. Whereas 52% of these projects have at least a single release that relies on an affected version of a vulnerable package. Considering that this share accounts for more than 72K dependent projects, while only 269 distinct upstream packages with known vulnerabilities were taken into account, the scale and importance of the problem become more than evident.

Further findings of Decan *et al.* [44] suggest that it takes nearly 14 months for 50% of the dependent packages to fix a vulnerable dependency. In fact, the dependent packages take not only a lot of time to be freed from vulnerabilities but also significantly more than the upstream packages. Identically, the study of Prana *et al.* [77] reveals that the high survivability of a vulnerable dependency is primarily caused by the delayed updates in the dependent application rather than the persistence of vulnerabilities across the releases of the upstream package. In their study of 450 software projects, Prana *et al.* [77] also analysed the relationship between vulnerabilities in the dependencies of a project and its attributes, such as activity level, popularity, and contributors, only to observe that the strongest correlation factor for the number of vulnerable dependencies is total dependency count. Indeed, as the dependency network grows, it becomes increasingly more complex for the developers to examine and manage it, despite the number of contributors, their experience, or project size. This suggests the need for automation to support developers in these duties.

Careful management of a project's dependencies, although is highly encouraged, not always

practised. By surveying the developers of the software projects with known vulnerabilities in dependencies, Kula *et al.* [65] report that, for developers, the effort needed to migrate a vulnerable dependency is of greater importance than the persistence of the security issue. Namely, developers are not likely to prioritize an update, perceiving it as an extra workload and responsibility. To make matters worse, 69% of the respondents were simply unaware of their vulnerable dependencies. This highlights that the community could benefit from the automated notifications on the security issues in projects' dependencies, stipulating our interest to study to what extent developers respond to them.

3.2 Software Bots

Software bots are rapidly becoming a commodity in software development. They assist developers with coding activities, play an integral role in testing, speed up code deployment, address slow feedback loops, bridge communication between users and developers, and generate documentation. In response, the research community has carried out a considerable effort on investigating these tools. Previous studies focus on identifying the challenges in interacting with software bots [69, 83, 85, 86], the impact of their usage on the development artifacts and software quality [63, 70, 84], and quantitatively measuring the extent of their adoption and developer receptivity towards their assistance [23, 32, 88].

In their recent study, Wyrich *et al.* [88] extracted over 20M GitHub pull requests to analyse and compare whether maintainers are more receptive to manually created pull requests averse those created automatically by bots. The authors found that the pull requests from humans are accepted and merged almost twice as often as bot pull requests (72.53% vs 37.38%), suggesting that such tools are not leveraged to their full potential. Furthermore, despite the fact that bot pull requests contain less changes on average, they take significantly longer to be interacted with and to be merged. However, Wyrich *et al.* have neither scoped their analysis to a particular bot nor restricted the domain based on the properties of the projects that use them, opting to go wide with a large collection of pull requests. On the contrary, in our work, we go deep and focus specifically on GitHub's Dependabot and mature and well-maintained JavaScript projects, aiming to mitigate the potential impact of confounding variables.

Despite the consensus of the research community on the complexity and burdensome of dependency management experienced by developers, there is an evident scarcity of studies that analyse the tools, which automate and alleviate this task. Similarly to our work, Mirhosseini *et al.* [70] analysed JavaScript projects that use a dependency management bot. Specifically, the authors investigated the impact of Greenkeeper, a bot that generates pull requests to upgrade out-of-date dependencies. The results acquired by Mirhosseini *et al.* show that, on average, projects that used the dependency management bot upgraded 1.6 times more often than projects that did not use any tools, suggesting a significant utility of such a suggestion mechanism. Although the authors report that 32% of automated pull requests in their dataset were merged, it remains unknown whether the developers disregarded the updates or performed them manually; the authors did not examine to what extent developers delegate the bot with the dependency updates. In turn, the survey of Pashchenko *et al.* [74] suggests that bots can be used solely for the identification of issues within dependencies, while the update itself is performed by a developer. In our work, we take it a step forward and not only analyse the developer receptivity to bot suggestions but also determine quantitatively how frequently they resolve the problem manually despite an automated pull request.

The only other study that investigates the usage of a dependency management bot, the recent study of Alfadel *et al.* [23], is the most relevant to our work. The authors examined the receptivity and level of adoption of the security pull requests authored by Dependabot-preview, the predecessor of GitHub's Dependabot. On the contrary to the previously discussed studies, they found that majority of such automated suggestions are merged (65.42%), and highly likely, within a day. However, some of the design properties of Dependabot-preview may affect the representativity of the acquired statistic. The principal functionality of the bot are version updates, and, as a

consequence, Dependabot-preview may frequently supersede the previous security update with a new one when a more recent release of the vulnerable package is available. Indeed, Alfadel *et al.* report that the majority of the rejected pull requests are closed by the bot itself due to superseding. Most importantly, Dependabot-preview ships with the *auto-merge* feature, which allows the bot to merge its own pull requests without developer intervention. Thus, the acquired findings do not necessarily reflect the developer reception and reaction to the security pull requests since many merges and majority of rejections are performed automatically. Examining GitHub's Dependabot, on the other hand, allows us to account for these confounding factors to a much greater extent - Dependabot always suggests the minimum required non-vulnerable version and does not feature auto-merge functionality.

Furthermore, Alfadel *et al.* [23] examined the factors that affect the rapid merges of the security pull requests generated by Dependabot-preview. Exploiting the regression modelling, the authors observed that the severity level of identified vulnerability has no significant impact, contradicting the other existing studies that conclude the relationship between the severity of a vulnerability and the time it takes to resolve it in both the upstream package [44] and the downstream applications [36]. Further results suggest that the risk of breaking changes as well has no significant influence, disputing the concerns shared by the developers [29, 56, 60]. We argue that the counter-intuitive findings could be a result of the misalignment between the subject of analysis and the research goals due to the design of the examined tool. Since authors report that the majority of the pull requests are superseded, it is very plausible that a rapidly merged update could be the last in a long chain of closed and newly opened pull requests. Therefore, it can be no rapid as the vulnerable dependency addressed by it could have been resolved much earlier with the pull request that got superseded due to non-responsiveness. In our work, we account for this and choose not the pull requests but the vulnerabilities specifically as the subjects of the analysis, ensuring the alignment between the analysis and the goals.

Chapter 4

Methodology

This chapter presents the data collection procedures we follow (Section 4.1) and the analysis techniques we leverage (Section 4.2) to answer the research questions posed in this work.

4.1 Data Collection

In this section, we describe the data collected for answering the research questions we raise in this work and cover the approaches and procedures we leverage to mine it. To analyse how often developers merge security updates of Dependabot (***RQ₁***), we argue and establish a collection period for their collection (Section 4.1.1). Furthermore, we assess the projects associated with the collected security updates based on their quality and properties, effectively excluding the undesirable ones (Section 4.1.2). Next, to investigate the fixes of vulnerable dependencies (***RQ₂***, ***RQ₃***, ***RQ₄***), we gather the cases of vulnerability resolution associated with the advisories flagged by Dependabot through an automated pull request in the selected projects. This requires extracting the security advisory records of GitHub (Section 4.1.3), defining the algorithm to detection of vulnerabilities in dependencies (Section 4.1.4), and finally, locating the cases of vulnerability fixing by traversing the repositories' histories. (Section 4.1.5).

4.1.1 Collection period

As discussed in Section 2.2.4, a year after introducing the security updates, GitHub extended the functionality of Dependabot by providing developers with version updates that aim at keeping dependencies up-to-date. Having no access to the project settings of public repositories, it becomes extremely challenging to identify whether a project that makes use of version updates also exploits security updates. Moreover, the presence of both services in a single project may contribute to confounding factors, effectively convoluting our findings that should solely target security updates.

One option to avoid this is to disregard the repositories that deploy a configuration file signalling the adoption of version updates. However, under such constraint, we would also eliminate the innovation-friendly projects that, after a successful adoption and continuous usage of Dependabot security updates, decided to opt-in for the version updates. As such, we instead constraint the collection period by the interval between the introduction of security updates to the community and the subsequent event of introducing version updates. In particular, the collection period marks the start as of June 1st, 2019 and the end as of May 31st, 2020, resulting in a time-window of exactly a year. Following this definition, we only consider the pull requests that were instantiated during this period.

4.1.2 Project selection

For our empirical study, we focus on JavaScript projects. The motivation for this is threefold. First, the annual surveys that GitHub conducts among its developers[1] suggest that JavaScript is taking the lead over other alternatives, as the most popular programming language. In turn, `npm` package manager, the official registry for JavaScript packages, is the largest ecosystem, comprising more than 1.68M packages[2], which is > 4 times greater in comparison to its closest predecessor (`Maven`). Combined, these two factors make our results relevant for a large community of developers. Moreover, JavaScript projects were observed to have the highest distribution of package dependencies in comparison to other programming languages that benefit from centralised library ecosystems [1]. As a consequence, for the maintainers of such projects, dependency management is a challenge of a much greater scale. Since the identical observation holds for the `npm` hosted packages [42] [45], JavaScript projects are more prone to inheriting vulnerabilities through dependencies. At last, JavaScript projects, and the `npm` packages in particular, are the primary targets of the recent studies on dependency networks [21, 22, 35, 37, 42–45, 58, 87, 89, 92] and tools [23, 70, 80].

Using the *GitHub Search API*[3], we identify 155,065 starred and non-forked repositories that were created before the start of the collection period and had at least a single update after it. To ensure that immaturity or inconsistent levels of activity do not introduce bias to our analysis, we select the projects that were actively maintained during the entire collection period and had no less than 100 commits at the start of the collection period [65]. We operationalise active maintenance as the presence of at least a single commit each month that belongs to the collection period. Furthermore, to identify whether the collected projects have dependencies on `npm` packages, for each of them, we track the presence of the manifest file in the root of the repository, leaving us with 3,587 projects.

The study of Kalliamvakou *et al.* [61] reveals that a large portion of GitHub repositories are used for experimental, storage, or academic purposes. For the sake of our study, we want to avoid such projects since their inclusion may introduce noise to the analysis. Indeed, the security concerns of the repositories hosting homework assignments or blogs are not necessarily applicable to the projects with general-purpose utility. Therefore, we proceed with the selection procedure by leveraging `Reaper`, a tool designed by Munaiah *et al.* [72] to identify engineered projects, averse the personal ones, such as homework assignments. The authors of the tool developed an evaluation framework consisting of eight quantifiable software engineering practices, called *dimensions*, to characterise a repository: *architecture*, *community*, *continuous integration*, *documentation*, *history*, *issues*, *license*, and *unit testing*. Accordingly, `Reaper` computes the values for these dimensions using the source code of the targeted repository and the history of the project by extracting the relevant data from GHTorrent [53], an offline mirror of GitHub metadata. In turn, to classify the projects for affiliation to the group of the engineered ones, the authors exploit a *random forest* classifier [30].

The application of `Reaper` to our scenario, however, requires a number of modifications. First, we remove a dependency on GHTorrent, since the latest image it provides is largely outdated. Instead, to mine the history of the identified projects, we leverage *GitHub GraphQL API*[4]. Secondly, the architecture dimension was not originally designed towards projects written in JavaScript. This dimension is operationalised by a metric, *monolithicity*, which measures the ratio of the number of files in the largest connected sub-graph to the number of files in the entire inter-file call-graph of the project. The approach the authors employed to estimate the inter-file call-graph, while applicable to the other programming languages, such as Python and Java, cannot be used for JavaScript source files due to the dynamic nature of this language. A possible measure to address this issue is to extend the tool by implementing an algorithm for the approximation of the call-graphs for JavaScript (*e.g.*, [50]). However, since for the training data, the call-graphs were computed using the author's own technique, we expect that utilization of a different approach would lead to a

[1] `https://octoverse.github.com/`. Last accessed February 21, 2021.
[2] `https://libraries.io/platforms/`. Last accessed February 21, 2021.
[3] `https://docs.github.com/en/rest/reference/search`. Last accessed February 21, 2021.
[4] `https://docs.github.com/en/graphql`. Last accessed February 21, 2021.

Table 4.1: Assessment of the classifier performance on the different sets of dimensions.

Classifier	FPR	FNR	Precision	Recall	F-measure
Baseline [72]	18%	17%	82%	83%	83%
Modified	25%	11%	78%	89%	83%

Table 4.2: Resource path resolution for the set of identified bots.

Bot	Path	Repositories
Dependabot [2]	/apps/dependabot	1,492
Dependabot-preview [9]	/apps/dependabot-preview	572
Greenkeeper [10]	/apps/greenkeeper	276
Snyk-bot [17]	/snyk-bot	228
Renovate [16]	/apps/renovate /renovate-bot	380

large non-conformity between the monolithicity values for the JavaScript projects and the ones written in the languages originally targeted by `Reaper`, effectively degrading the accuracy. Hence, in line with the study of Cassee *et al.* [33], we disregard the architecture dimension. Following the approach of Cassee *et al.*, we also remove the issue dimension, operationalised as the issue frequency across the time-span between the first and the last commit. While the tool authors rely on this dimension as an evidence to the project management, many of such tasks are carried out with external applications, and thus, not visible in the GitHub records. To ensure that removal of these two dimensions does not vastly degrade the performance, we benchmark the tool using the training and validation datasets supplied by Munaiah *et al.* [72]. The authors provide two datasets for the training procedure, *organisation* and *utility*. In contrast to the organisation dataset, which comprises of the public repositories owned by widely known companies, such as Google, Amazon, Facebook, *etc.*, the utility dataset contains the projects that were manually labelled for a general-purpose utility to users other than the developers themselves, irrespective of the organisation behind them. Since Munaiah *et al.* report to achieve a better performance when using the latter collection, to benchmark the tool and label the projects in our study, we train a classification model using the utility dataset. The validation dataset contains a set of 200 repositories manually labelled for the ground truth, *i.e.*, engineered (100) vs. non-engineered (100) projects. To conduct the training procedure, we rely on `scikit-learn` [76], one of Python's most widely used machine learning libraries. In line with the original work [72], we assess the classification performance based on false positive rate (FPR), false negative rate (FNR), precision, recall, and F-measure. Table 4.1 presents the results for the *baseline* classifier that utilises all of the eight dimensions (as reported by Munaiah *et al.* [72]) and the *modified* model that excludes the architecture and issue dimensions. As can be observed, at the expense of a 4% drop in precision, the modified classifier achieves a 6% increase in recall. Overall, observing no change in the F-measure, we are confident that the removal of the architecture and issue dimensions has no significant negative impact on the tool accuracy. Applying the trained classifier to the feature dimensions computed for the 3,587 selected repositories, accounting for their state as of the end of the collection period, we identify 3,151 engineered projects.

Accounting for the confounding factors that stem from the utilization of multiple dependency management bots, we remove the projects that exercise this practice from our collection. Surveying the existing literature [37, 47–49, 63, 88] and the *GitHub Marketplace*[5], the official registry for the third-party GitHub applications, we identify four other bots that provide automated pull requests that update dependencies in JavaScript projects: Dependabot-preview [9], Greenkeeper [10], Snyk-bot [17], and Renovate [16]. Following this, for each selected repository, we extract all the pull requests that were instantiated during the collection period and inspect their authors. Specifically, we track the resource path of the author, *i.e.*, the HTTP path for this actor. Consider Table 4.2

[5] `https://github.com/marketplace`. Last accessed February 21, 2021.

Table 4.3: Summary of the collected security updates.

State	Security updates
Open	221
Closed	1,847
Merged	2,446

Table 4.4: Characteristics of the selected projects.

Metric	Min.	Max.	Median	Mean
Forks	0	33,022	33	391.88
Stars	2	180,228	63	2,121.08
Core contributors*	1	477	4	8.58
Security updates	1	67	3	4.50
Commits before col. period	101	48,807	890	2,019.81
Commits during col. period	25	15,306	346	670.80

* *Computed in line with [72]*

that presents the mapping between the bot name and its corresponding resource path(s) but also displays the number of the repositories in which pull request history the traces of this actor were encountered. Accordingly, from the set of 1,492 repositories that have at least a single pull request issued by Dependabot, we filter out 390 projects that, aside from this dependency management bot, employ one or more other ones.

At last, we filter out the projects that have received Dependabot security updates targeting ecosystems other than `npm` (*e.g.*, `Maven`, `RubyGems`). The final dataset consists of 4514 security updates (see Table 4.3), associated with 1,003 JavaScript projects. The collection includes software projects maintained by Google, Mozilla, Facebook, eBay, SAP, and vastly popular `npm` packages, such as *react*, *vue*, and *mongoose*. In Table 4.4, we report the characteristics of the collected projects.

4.1.3 Security Advisory Records

We retrieve the security advisories from the database of GitHub using the *GitHub GraphQL API* on March 27th, 2021. Some of the advisories concern the vulnerabilities with no known fixes (a fixing patch has not been released) or those that originate from the ultimately malicious packages. These advisories either comprise no entry for the field dedicated to the first patched release or contain a collocation "Malicious Package" in the summary. Since such vulnerabilities cannot be addressed by Dependabot, we remove the corresponding entries from the collection, leaving us with 1,063 security advisory records. However, the database of the security advisories is constantly evolving, and as such, some vulnerability records may be taken down, republished or modified. Therefore, the advisories that incurred the creation of some of the collected Dependabot security updates may have already be taken down by the time we have mined the database. Accordingly, we verify the extent to which this problem applies to our data. To this end, for each retrieved security update, we identify whether there is at least a single associated security advisory from the collected set.

As discussed in Section 2.2.3, the information about the vulnerabilities targeted by a security update cannot be accessed without specific project permissions. In fact, once the concerned vulnerability is eliminated, this information becomes unavailable to the project maintainers as well. The reason is that each security update is linked to a security alert, and, if the latter is addressed, the record is erased, effectively breaking the link. Therefore, to identify the association of an advisory with a security update, we implement an algorithm that extracts this information based on the title of an automated pull request.

The title of a security update captures the vulnerable upstream package, its currently installed

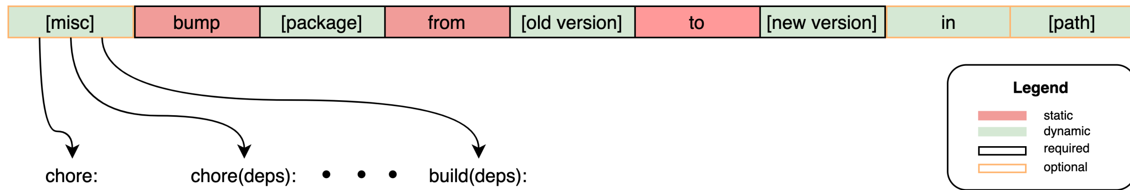


Figure 4.1: Structure of a security update title.

version, *i.e.*, declared in the manifest or lock files, and the release to which Dependabot suggests to upgrade. Consider Figure 4.1 that demonstrates its structure. Omitting the optional constructs, such as semantic commit message prefix ("misc") and the path to the dependency files, one can produce a regular expression to extract the relevant information from the title of a security update. We use the name of a vulnerable package to locate the security advisory records that correspond to it. Following this, we filter out the records which were published after the pull request that represents the security update has been raised since they could not have induced its generation. Accordingly, for each advisory, the range of the vulnerable releases is matched against the currently installed version ("old version") to identify whether the latter belongs to a list of the affected releases. To this end, and for any further operation on evaluation of dependency constraints and comparators in our work, we rely on the semver[6] module, leveraged by `npm`. Furthermore, to ensure that the security update concerns the identified vulnerability, we verify that the suggest release ("new version") is outside of the range of the affected ones.

After applying the algorithm to the set of the collected security updates, we find that for 479 of them, the associated security advisory could not be deducted. To address the problem and restore the missing records, we manually investigate these cases and inspect the past commits, comments, and discussions on GitHub. We find advisory records, associated with eight vulnerable packages, that were altered, removed, or re-entered into the database (resulting in an imprecise publication date). These cases, together with the evidence to them, are presented in Appendix A. Re-applying the algorithm on the extended collection of the security advisories, we manage to reduce the number of unlabelled security updates to 28 (0.6%). Accordingly, we remove them from the further analysis.

4.1.4 Detecting Vulnerabilities in Dependencies

To recognise the cases of vulnerability resolution, first and foremost, there is a need to define a set of rules that dictate whether a project is affected by a concerned security vulnerability through its dependencies or not. One option is to adopt the algorithm introduced by Decan *et al.* [44] in their study on the propagation of vulnerabilities through dependencies in the `npm` ecosystem. Another option is to leverage the logic behind the *audit* command of the `npm` client that, given a description of the configured dependencies, returns a report of the detected vulnerabilities. However, in our work, we decide to adopt the technique used by GitHub's own services. In particular, Dependabot security alerts and security updates. The reason for this is that Dependabot acts as the uniform source of the security vulnerability information shared among all of the selected projects.

Even though for the original tool, Dependabot-preview, the codebase is exposed to the public, GitHub's Dependabot, as the collection of services, remains a closed-source project. Specifically, the security alerts service, used to determine whether there is a need to update a dependency, *i.e.*, whether it is vulnerable or not. Therefore, to identify GitHub's logic in vulnerability detection, we reverse-engineer it without access to the source code, *i.e.*, based on its inputs and outputs. To this end, we generate a plethora of GitHub repositories with the security services enabled, to which we deploy the dependency files of over 50 different projects from our dataset, chosen randomly. As such, the deduction routine comprises of manually modifying the dependency files to observe the impact of the changes onto the security alerts. Additionally, at each modification, we examine the

[6] `https://semver.org/`. Last accessed February 24, 2021.

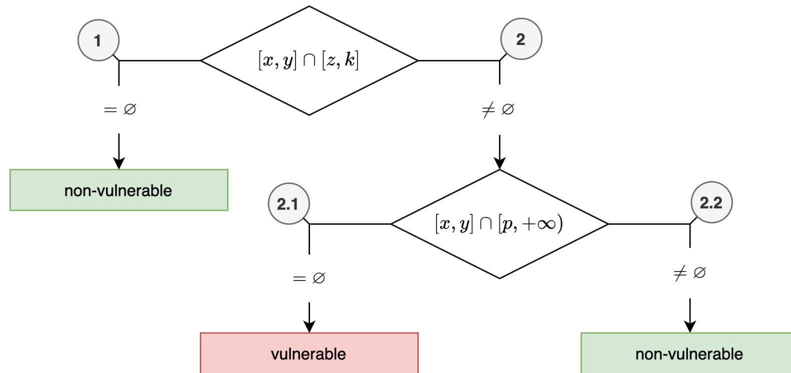


Figure 4.2: Decision tree routine for a manifest file; $[x, y]$ stands for the range defined by the dependency constraint, $[z, k]$ defines the range of the vulnerable releases, and finally, p is the first patched release.

security updates generated by Dependabot, provided the dependency files declare a dependency on the affected releases of a package with a known vulnerability. The entire procedure is performed iteratively, until the deducted algorithm, described in Section 4.1.4.1, passes the validation routine, discussed in Section 4.1.4.2.

4.1.4.1 Deducted Algorithm

The algorithm takes as input (1) the name of the concerned upstream package with known security vulnerabilities, (2) the database of the security advisories, and (3) the dependency files of the selected repository. Based on the provided input, the algorithm computes the set of vulnerabilities inherited through the dependency on the specified upstream package. The hosted repository is deemed to be affected by a vulnerability if at least one dependency file is found to declare a dependency on a vulnerable release of an upstream package. However, the conditions of the latter definition differ between the manifest and the lock files.

Manifest File The manifest file is said to declare a dependency on a vulnerable release if the specified upstream package is present as a direct runtime or development dependency, while its most recent version that satisfies the defined dependency constraint is affected by the concerned vulnerability. In fact, this definition is almost entirely identical to one derived by Decan *et al.* [44], with only one difference - development dependencies are also taken into account. Under the described definition, the identification of a vulnerable dependency requires the possession of the meta-information on each release of each package with known vulnerabilities. One possibility to obtaining this data requires querying the `npm` registry, which would consume a large amount of time. An alternative is to leverage the source discovery service Libraries.io[7], known to publish the datasets on package releases for several registries, including `npm`. However, the latest image is dated by January 2020, which does not satisfy our needs. Nevertheless, after examining the database of the security advisories maintained by GitHub, we identify that the properties of its entries and the way they are published allow us to drop the need in the release information. First, each security advisory is associated with one or more disjoint ranges of vulnerable releases. In turn, each range is either given by (a) an upper bound (*e.g.*, "< 1.5.1"), or (b) both a lower and an upper bounds (*e.g.*, ">= 2.5.5 < 3.0.0", "= 4.0.0"). Furthermore, for each range, an advisory includes the first patched release, that is, the lowest version coming after the right bound of the range, which is not affected by a vulnerability and is available for installation by the time the advisory is published. Indeed, unless the package is ultimately malicious, GitHub only publishes the security advisory if the associated vulnerability has already been patched. Hence, to identify

[7] `https://libraries.io/data`. Last accessed March 10, 2021.

whether the most recent release that satisfies the dependency constraint, given by the range $[x, y]$, is affected by a vulnerability, it satisfies to apply the decision tree routine captured by Figure 4.2 to each pair of a vulnerable releases range $[k, z]$ and the first patched release p associated with the security advisory that corresponds to this vulnerability. First, we determine whether the range defined by the dependency constraint declared in the manifest file intersects with the range of the vulnerable releases of the upstream package.

1 If it does not, then any version that satisfies the constraint is either strictly lower or higher than any version defined by the range of vulnerable releases; thus, the most recent installable release is not vulnerable.

2 If it does, there is a need to verify whether the first patched release or any version that came after it satisfies the dependency constraint. That is, whether the range defined by the dependency constraint intersects with the range defined by the comparator "$\geq p$", where p is the first patched release.

2.1 If not, then the most recent installable version that satisfies the constraint is strictly within the range of vulnerable releases, *i.e.*, the most recent installable release is vulnerable.

2.2 Otherwise, this release is either the first patched version or any version, which is higher than it, *i.e.*, the most recent installable release is not vulnerable.

Accordingly, if for any pair of a range and the first patched release, the most recent version that satisfies the constraint is affected by the concerned vulnerability, then the examined `package.json` file is found to declare a dependency on a vulnerable release of an upstream package. To identify whether the two version ranges intersect, we leverage the eponymous function of the semver module used by the `npm` package manager.

Lastly, an additional rule applies if the repository follows the *monorepo* paradigm [31], *i.e.*, contains more than one project. Commonly, this implies that the sub-modules on which the top-level (or "main") project depends on are hosted within a single multi-package repository. If this is the case, then the direct dependencies of the hosted sub-modules, the relative paths to which are defined through the "workspaces" field in the manifest file, are deemed as direct dependencies of the entire top-level module. Accordingly, we also inspect them to identify whether there is a direct runtime or development dependency on the specified upstream package with a known security vulnerability.

Lock files Concerning the lock files, there are two scenarios. In the first case, the specified upstream package with a known security vulnerability is declared as a direct runtime or development dependency in the manifest file (including the hosted sub-modules defined through the "workspaces" field). To determine whether the examined lock file declares a dependency on a vulnerable release of an upstream package, solely the release assigned to a node in the dependency graph that represents this direct dependency is validated, whereas the nodes associated with the transitive dependencies are ignored. To this end, for a `package-lock.json` (or `npm-shrinkwrap.json`), we retrieve the top-level dependency node object (recall Section 2.1.3) defined by the name of the concerned upstream package and examine the value for the property "version". For a `yarn.lock`, we perform a lookup for the dependency resolution block (recall Section 2.1.3), whose header contains a package-constraint pair that corresponds to the name of the concerned upstream package and the dependency constrain assigned to it in the manifest file. Similarly, we retrieve the value of the "version" field. For both the `package-lock.json` and the `yarn.lock` files, this value represents the version of the upstream package locked for the direct dependency on it.

In the second scenario, the specified upstream package with a known security vulnerability is not declared as a direct runtime or development dependency. Then, conversely to the previous case, every node in the dependency graph that corresponds to a dependency on the concerned upstream package with a known security vulnerability is inspected. If the locked release of at least a single node in the dependency graph belongs to the range of the affected versions, then the file is said to declare a dependency on a vulnerable release of an upstream package. To verify this,

for a `package-lock.json` file, we recursively traverse every dependency node object in the graph, examining those that are defined by the name of the concerned upstream package and collecting the releases locked to them. For a `yarn.lock` file, we perform a lookup for every dependency resolution block with header containing the name of the concerned upstream package, irrespective of the dependency constraint, and retrieve the assigned releases.

4.1.4.2 Validation

To validate the algorithm amid the deduction routine, we leverage the mined Dependabot security updates. Indeed, the *parent commit* of the security update, *i.e.*, the commit on which the modification is based, represents the ground truth state of the project with a vulnerable dependency. Whereas the *merge commit*, *i.e.*, commit resulted by merging the security update, captures the state at which this vulnerable dependency is resolved.

As discussed in Section 2.2.3, each Dependabot security update addresses at least a single vulnerability in dependencies. This vulnerability is publicly known at the time of the update creation and sourced off the upstream package, the name of which is mentioned in the title of the corresponding automated pull request. Furthermore, each security update may not only modify the top-level, *e.g.*, the manifest and lock files deployed at the root level of a project, but also exclusively target the dependency files of the sub-modules at the lower levels. In fact, Dependabot may target multiple (sub-)modules at different levels through a single security update. However, not every single one of them must be affected by vulnerability. The reason is that amid addressing a security vulnerability in one (sub-)module, Dependabot may select a version higher than the first patched release for the dependency (due to other constraints), effectively attempting to also modify it across the entire repository, regardless of susceptibility to the vulnerability. That is, bring the selected directories up to date with the selected non-vulnerable release. Accordingly, given a security update generated at date $\boldsymbol{d}$, targeting upstream package $\boldsymbol{u}$, and modifying the dependency files in the (sub-)modules given by set $\boldsymbol{F}$, for the state of the dependency files associated with the:

parent commit there is at least a single (sub-)module in set $\boldsymbol{F}$ containing one or more dependency files that are found to declare a dependency on a vulnerable release of package $\boldsymbol{u}$, such that this release is affected by at least one vulnerability publicly known at date $\boldsymbol{d}$;

merge commit there is no a (sub-)module in set $\boldsymbol{F}$ containing one or more dependency files that are found to declare a dependency on a vulnerable release of package $\boldsymbol{u}$, such that this release is affected by at least one vulnerability publicly known at date $\boldsymbol{d}$;

Following this, the deducted algorithm is validated through binary classification. For the parent commit, the positive outcome is expected, *i.e.*, a vulnerability is encountered. In this scenario, the deducted algorithm is expected to return at least a single security advisory from the set of the records (1) associated with the selected upstream package and (2) published before the corresponding security update was generated. For the merge commit, in turn, the negative outcome is expected, *i.e.*, no vulnerability is identified. For this case, the deducted algorithm should return an empty set.

The final version of the algorithm, described in this paper, reports no false negatives. That is, for every security update in our collection, the algorithm manages to flag the presence of a vulnerable dependency in the repository's state associated with the parent commit. Nevertheless, when it comes to the merged security updates, we find 133 cases of false positives. In other words, for these 133 security updates, the algorithm suggests that the modification of Dependabot does not eliminate the vulnerability. After carefully examining the false-positive cases, we find that the reason for them is a bug in Dependabot rather than an incorrectly deducted algorithm. The identified bug arises when all of the following conditions are met: (1) the associated advisory comprises multiple disjoint ranges of vulnerable releases; (2) there is no direct dependency on the concerned upstream package with a known vulnerable release(s); (3) in the dependency graph,

acorn

Dismiss

Open GitHub opened this alert 4 days ago

Bump acorn from 5.7.3 to 5.7.4 dependencies

#1 by dependabot bot was merged 23 hours ago

1 **acorn** vulnerability found in **yarn.lock** 4 days ago

Remediation

Upgrade **acorn** to version **`7.1.1`** or later. For example:

```
acorn@^7.1.1:
  version "7.1.1"
```

Always verify the validity and compatibility of suggestions with your codebase.

Details

GHSA-6chw-6frg-f759

moderate severity

Vulnerable versions: `>= 7.0.0, < 7.1.1`
Patched version: `7.1.1`

Affected versions of acorn are vulnerable to Regular Expression Denial of Service.
A regex in the form of /[x-\ud800]/u causes the parser to enter an infinite loop.
The string is not valid UTF16 which usually results in it being sanitized before reaching the parser.
If an application processes untrusted input and passes it directly to acorn,
attackers may leverage the vulnerability leading to Denial of Service.

Figure 4.3: Screenshot of a security alert.

there are multiple nodes corresponding to a transitive dependency on the concerned upstream package; (4) there are at least two such nodes for which a locked release belongs to one of the ranges of vulnerable versions, but for each of these nodes, the range is different; and (5) the file requiring an update is `yarn.lock`. In this scenario, Dependabot accidentally ignores every range of the vulnerable releases but one and modifies the `yarn.lock` file accordingly. Therefore, at least one dependency resolution block remains pointing to a vulnerable release after the modification of the bot.

To verify the bug conjecture, we replicate the aforementioned conditions in a GitHub repository with both the Dependabot security alerts and security updates enabled. We find that after merging the automated pull request generated by Dependabot, the associated security alert persists. Consider Figure 4.3 that captures an example of such an alert. In this instance, the alert concerns a vulnerability in the *acorn* package that covers multiple versions across three different major releases, resulting in three disjoint ranges of the vulnerable versions. Specifically, major version “5.x”, “6.x”, and “7.x”. Accordingly, Dependabot resolved the nodes that were previously locked with a vulnerable version at the fifth major release, as can be deducted from the title of the corresponding pull request given at the top of the figure. However, the nodes that are assigned with a vulnerable version at the seventh major release remained unaddressed. In further support, we also find a discussion over a Dependabot security update [8], in which developers pointed out the aforementioned error and eventually decided to manually revise the suggested modification.

4.1.5 Fixing Cases Discovery

To trace the cases of fixing a vulnerable dependency, we collect the events of identifying a security vulnerability propagated through the dependencies in the selected repositories. To this end, we apply the deducted vulnerability detection algorithm to the parent commits of each security update, and generate a separate event for every reported security advisory. As such, we ensure that for each identified vulnerability, there is a Dependabot security update that targets it, which is the constraint induced by $\boldsymbol{RQ_2}$. Accordingly, each event is identified by the following four properties: (1) the slug, *i.e.*, the name with the owner, of the repository, (2) the associated security advisory, which also includes the name of the affected upstream package, (3) the concerned (sub-)modules containing the dependency files that declare a vulnerable dependency, and (4) the parent commit of the associated security update. The latter property allows capturing the earliest state in the repositories history at which Dependabot has identified the vulnerability.

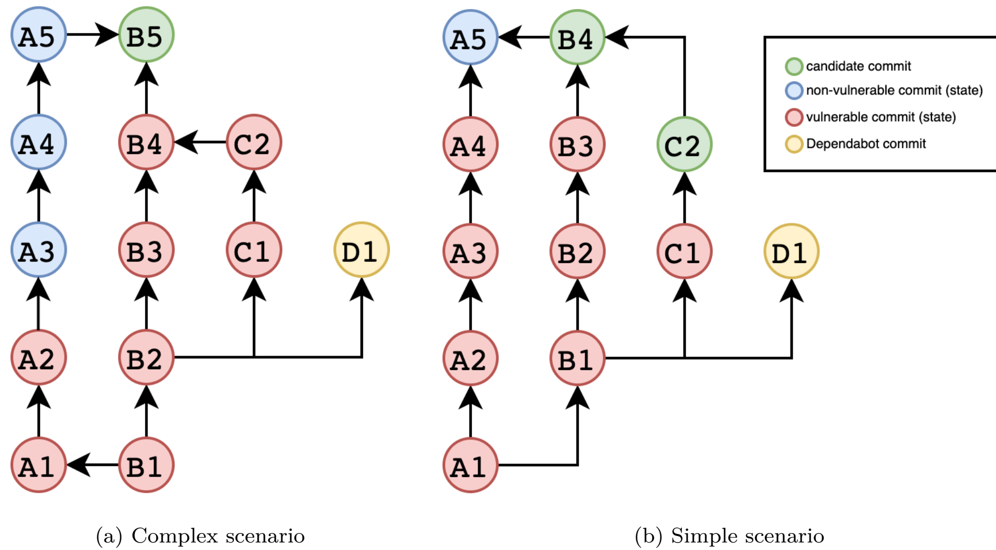
In total we obtained 5395 events of identifying a security vulnerability propagated through the dependencies. However, Dependabot may instantiate a new security update in case a more recent version of the upstream is required, targeting a superset of the vulnerabilities that were concerned by the superseded pull request. Therefore, some of the collected events capture not the earliest state at which Dependabot has identified the vulnerability, but the state at which the security update targeting this vulnerability was re-instantiated. To account for this, we link the superseded and superseding security updates and drop such events, leaving us with 5089 entries.

Finally, to collect the fixing cases, for each event of identifying a security vulnerability, we trace the *fixing commit*. We define a fixing commit as the earliest modification to the dependency files that resolves the specified vulnerability in the dependencies and eventually reaches the default branch of the repository. That is, the first commit that results in a state of the concerned repository such that it is no longer deemed affected by the specified vulnerability. The requirement on affiliation to the default branch is explained by the fact that we only account for the commits that were accepted by the project maintainers, effectively ignoring those that did not reach the users of the repository. To discover the fixing commit, we implement the algorithm that recursively visits the descendants of the commit that served as a parent for the security update generated by Dependabot, *i.e.*, the subsequent states of the repository, and identifies whether the concerned vulnerability is eliminated or not using the technique we deducted. If it is eliminated, then the recursive call halts and returns the commit to the caller. Since the repository history commonly comprises of more than a single branch, the algorithm may return multiple *candidate commits*, *i.e.*, candidates for the fixing commit. The reason is that once the original modification carrying the fix located at a certain development branch reaches another branch through, *e.g.*, a merge, then for the latter branch, the earliest node with the vulnerable dependency resolved is this merge commit. Therefore, due to traversing each development branch independently, the algorithm may return more than one commit. However, it is also possible that none of the candidates is the fixing commit. We regard such a scenario as *complex*, on the contrary to *simple*, where one of the candidates is the fixing commit. Consider Figures 4.4b and 4.4a.

Both of the figures represent excerpts of the repository history, similar to the visualisation of the network graph realized by GitHub[8]. Each node represents a commit, whereas the directed edges capture the parent-child relationships between them. Red nodes depict commits for which the associated state of the dependency files results in the repository being affected by a vulnerability through dependency. The green is used to define the candidate commits, for which the dependency files do not declare a vulnerable dependency. The yellow highlights the commit generated by Dependabot as a part of a security update. Finally, the blue is used to depict that the associated state is non-vulnerable, but the commit does not belong to the list of candidates.

As can be observed from Figure 4.4b, the parent commit of the security update generated by Dependabot is `B1`, while the candidate commits are `B4` and `C2`. In this simple scenario, the fixing commit `C2` is a descendant of `B1`. Whereas examining the complex scenario depicted by Figure 4.4a, one can observe that the fixing commit `A3` does not belong to the list of candidates,

[8] `https://docs.github.com/en/github/visualizing-repository-data-with-graphs/viewing-a-repositorys-network`. Last accessed February 21, 2021.



(a) Complex scenario

(b) Simple scenario

Figure 4.4: Candidate and fixing commits scenarios.

as the commit `B2`, *i.e.*, the parent of the commit instantiated by Dependabot, is not its ancestor. The reason is that the branch `A`, used as the origin for the fix, was forked before `B2` was created.

The existence of the complex scenario explains the reason for no complete automation for the discovery of the fixing commit. Avoiding them requires traversing the network graph backwards, which is expected to increase the number of nodes to be visited by orders of magnitude, effectively impeding the research. Following this, we opt for the semi-automated solution, *i.e.*, given the list of the candidate commits, a human rater determines the fixing commit manually. In the complex scenario, however, the earliest candidate commit is a result of a merge, and as such, a rater is required to manually investigate the ancestors of this candidate to, ultimately, identify the fixing commit. To assess the extent of accidental errors or the potential bias due to our manual assignment, we recruit another independent rater. This second rater is presented with a total of 50 events of identifying a security vulnerability with a list of the candidate fixing commits for each of the events. One-half of the events pertaining to a complex scenario, while the other does not, which is not revealed to the rater. The rater is also familiarized with the vulnerability deduction algorithm and the task at hand through the guideline captured in Appendix B. Accordingly, we find no discrepancy between the fixing commits reported by the original and the second raters, increasing our confidence in the accuracy of collected fixing cases.

4.2 Data Analysis

In this section, we discuss the techniques we employ to translate the collected data into interpretable results that address the questions posed in this work. In particular, in Section 4.2.1, we report on the metric we introduce to operationalize receptivity towards Dependabot security updates and the motivation to split the projects into separate groups to analyse it (***RQ₁***). In Section 4.2.2, we cover the guideline we follow to distinguish between the bot and human fixes and the analysis techniques we use to study the developer response towards vulnerabilities on different levels of granularity (***RQ₂***). Next, in Section 4.2.3, we discuss the procedure and the set of rules defined to collect developer motivation to manually address or ignore a vulnerability and the approach we utilize to compensate for the subjectivity in qualitative analysis (***RQ₃***). Finally, in Section 4.2.4, we describe the operalization framework to measure the time to address a vulnerability and the modelling technique we use in the analysis to account for the security vulnerabilities

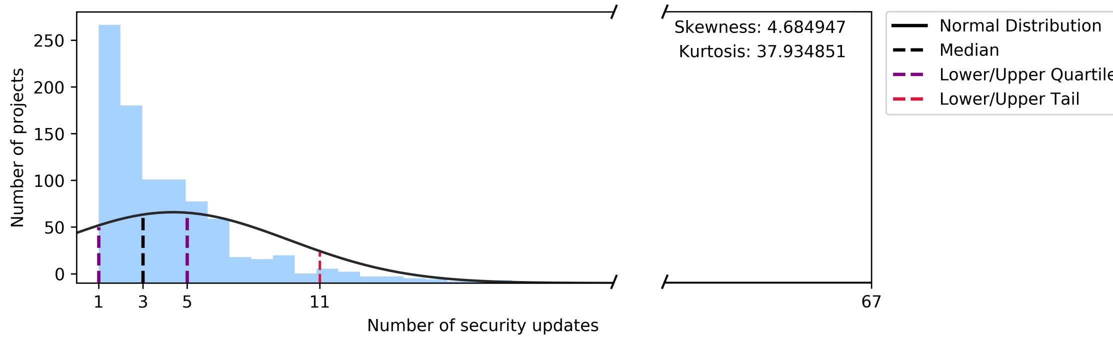


Figure 4.5: Distribution of the number of security updates.

Table 4.5: Project classification based on the number of security updates.

Group	Number of security updates	
	Constraint	Interpretation
Very low	[lower quartile, median)	[1, 3)
Low	[median, upper quartile)	[3, 5)
High	[upper quartile, upper tail)	[5, 11)
Very high	[upper tail, maximum]	[11, 67]

that remain unresolved ($\boldsymbol{RQ_4}$).

4.2.1 Addressing RQ$_1$

The first research question aims at identifying how frequently developers merge the security updates generated by Dependabot. To this end, we introduce a metric, *merge ratio*, defined as the proportion of the non-open security updates that were merged. That is, the ratio is computed over a set of the security updates whose state at the moment of the data retrieval was either "closed", *i.e.*, closed without merging, or "merged" but not "open". The reason to exclude them is that this state indicates that a decision to or not merge a pull request has not been made yet.

We compute the merge ratio over the entire collection of the extracted security updates but also perform a more fine-grained analysis by assessing this metric on a per-project basis. Different projects may adhere to different practices, and thus, respond to the security updates in a non-uniform manner. Therefore, it is crucial to account for this and examine the distribution of merge ratios among the selected projects.

Given the relativity of the metric, the range of the possible values is directly determined by the number of non-open security updates a project has. Hence, if, amid the collection period, a project received a very small number of automated pull requests (*e.g.,* 1 or 2), it is extremely likely that the merge ratio is either 0% or 100%. If this is a predominant case, then the distribution of merge ratios will be mostly determined by these projects, heavily skewing it to both of the extremes and, effectively, impeding the analysis. We plot the distribution of the number of security updates to verify whether the aforementioned scenario applies to our dataset. Additionally, we report the coefficients of *skewness* and *kurtosis* [64], whose acceptable values for normality lie between -1 and 1 and between -2 and 2, respectively. Consider Figure 4.5. As can be observed, both the plot and the computed coefficients indicate that the distribution suffers from a very high positive skew. As a countermeasure, we follow a quantile classification approach [66] to compose four disjoint groups of projects that characterized by the degree of recipiency of the automated pull requests. Namely, *Very low*, *Low*, *High*, and *Very high*. In Table 4.5 we report the constraints, *i.e.*, cut-off points, based on which the classification is performed.

4.2.2 Addressing RQ_2

The second research question aims to assess how often developers fix a vulnerability in dependencies manually despite the availability of the Dependabot security update that addresses it. To answer this question, we measure the percentage of the vulnerabilities in the obtained collection that were (1) fixed by Dependabot, (2) fixed by a developer, or (3) have not yet been addressed. We determine the inclusion to the latter group through the absence of the fixing commit, whereas distinguishing between the first and the second groups is more complex. Indeed, provided the external viewpoint onto developers' actions, one can only conjecture to what extent the fixing modification authored by a developer is influenced by the suggestion of Dependabot. Even in the case that the first is an exact copy of the latter. Following this, in our work, we decide to implement the high precision strategy - we regard a fix as contributed by Dependabot if and only if the fixing commit is authored by it. Otherwise, we attribute the fix to the developers.

To maintain consistency with the analysis for the first research question, we also compute the percentages for the four groups of projects separately. In particular, for each group, we compute two pairs of percentages: (1) percentage of vulnerabilities that were addressed vs not addressed, and (2) out of all fixes, the share that was contributed by a bot vs implemented by a human. In the event of an observable discrepancy in the results for the different groups, we validate whether this difference is statistically significant by computing the contingency table with the absolute values and applying the Pearson's χ^2 test [75]. We reject the null hypothesis H_0, which assumes no relationship between the number of security updates received by a project and the expected response to a vulnerability, if $p < 0.05$ (the traditional 5% significance level). In line with the common guidelines [25], we also report the effect size, *i.e.*, magnitude of this relationship, provided evaluation of the Pearson's χ^2 test suggests the rejection of the null hypothesis. Despite the plethora of approaches to compute the effect size [25], for a contingency table the size of which is not upper-bounded by 2×2, it is recommended [57, 59] to use Cramér's V [41], denoted by ϕ_V. The metric varies between 0 and 1, with the former corresponding to a lack of association. We follow the interpretation of ϕ_V proposed by Jacob Cohen [38], which suggests that for a 4×2 contingency table, the association between two variables is *trivial* if $\phi_V < 0.10$, *small* if $0.10 \leq \phi_V < 0.30$, *medium* if $0.30 \leq \phi_V < 0.50$, and *large* if $\phi_V \geq 0.50$. However, the rejection of the null hypothesis over an entire contingency table and a non-negligible effect size neither imply that the difference in populations is statistically significant between each group nor indicate between which groups specifically it is. As such, we complete the analysis by performing $\binom{4}{2} = 6$ pairwise comparisons, *i.e.*, Pearson's χ^2 tests. To compensate for an increased risk of a type I error when making multiple statistical tests, we control the false discovery rate by adjusting the p values following the Benjamini-Hochberg correction procedure [27].

4.2.3 Addressing RQ_3

To provide further insights, we qualitatively identify the reasons developers decide not to address a vulnerability, implement it manually, or solely reject the proposition of Dependabot. On the contrary to previous studies [23], we do not aim to discover the motivation to the non-merged pull requests but the rationale behind the decisions on vulnerability resolution. For instance, in the first scenario, we identify that a suggested fix is rejected due to being implemented manually, whereas, in the second, we determine that this fix is implemented manually due to breaking changes incurred by the automatically proposed update. To this end, we examine the git commit messages submitted with the fixing commit (if present), the comments left for the associated security update generated by Dependabot, and the other related discussions (*e.g.*, referenced in the logs). Accordingly, for each vulnerability not addressed by merging the original (non-superseding) Dependabot security update, we review the communication artefacts and use the collected cues to deduct and assign a short label that captures the reason. Each label is accompanied by one to two sentences that explain it. When assigning a label, we only rely on the rationale suggested by the developers themselves, without conjecturing on the possible events and intents that are not specified explicitly. However, amid the analysis, we observe that the vast majority of non-merged

security updates are rejected silently, while vulnerability fixes are most commonly accompanied by a superficial message. As such, through the manual review, the first author manages to identify only 213 explicitly motivated and actionable responses to a vulnerability notification. Despite such a low number of cases, we establish 22 unique reasons, *i.e.*, labels, that explain them. We cluster these fine-grained labels into six generalized groups. This allows to strictly outline the domains of the developers' concerns that guide the decision to or not to merge a security update of Dependabot.

To counteract the subjectivity that stems from the manual analysis and ensure the validity of the results, we recruit a second rater to perform a separate round of labelling independently (consult Appendix B for the rater guideline). To assess the agreement between the two raters, we compute Cohen's κ, the de facto standard statistic, which is considered robust since it accounts for the possibility of agreeing by chance. Comparing the classification of the two raters, we observe $\kappa = 0.963$. This, following the interpretation proposed by Viera and Garrett [81], is equivalent to *perfect agreement*, which increases our confidence in the reliability of the acquired results.

4.2.4 Addressing RQ$_4$

In our work, we focus on developer interaction with Dependabot rather than its impact on the users of the studied projects. Therefore, by answering the fourth research question, we aim to capture the degree to which developers react to the vulnerabilities identified by Dependabot in a timely manner. That is, contrary to the study of Decan *et al.* [44], we are not interested in the difference between the time the vulnerability occurred in dependencies and the time the security fix reached the users. Instead, in the case of a human fix, we operationalize the time required to resolve a vulnerable dependency as the difference between (1) the time the vulnerability was reported by Dependabot through a security update and (2) the time the fixing commit was made. On the other hand, when the fix is made by the bot, we measure the difference between (1) the time the vulnerability was reported by Dependabot through a security update and (2) the time the fixing security update was merged.

In line with the previous software engineering studies [39, 42, 44, 68, 78], to assess the time-to-event distribution, we rely on the statistical technique of the survival analysis [20]. The advantage of this statistical approach is that it allows to account for the vulnerabilities that remain unaddressed by the end of the observable period, *i.e.*, *censored* observations. Using the Kaplan-Meier estimator [62], a non-parametric statistic, we fit a survival analysis model to estimate the survival rate of the vulnerabilities in dependencies, *i.e.*, the expected time duration until an actionable reaction to a vulnerability, over time. Accordingly, the survival function represents the probability that a vulnerability survives past a certain time point. To visualize it, we plot the survival curve along the timeline with a 95% confidence interval.

In addition to the overall analysis, for each severity level, we fit a separate survivals analysis model. This allows verifying whether there is a relationship between the risks incurred by a vulnerability and the time it takes for the developers to react to it. That is, whether the developers take into account the severity levels in prioritisation. To verify the significance of any observable difference between each pair of the severity levels, we carry out $\binom{4}{2} = 6$ pairwise comparisons using the log-rank test [51], the de facto testing procedure for comparing time-to-event distributions. The null hypothesis H_0 assumes that there is no difference in the survival distributions of the two groups. Given the absence of a meta-test that considers all four survival curves at once, we choose a more conservative approach for type I risk correction. Specifically, we control the family-wise error rate, *i.e.*, probability of making at least one type I error, and following the Bonferonni approach [71], test each individual hypothesis at a significance level of 0.83% ($= \alpha/T$, where $\alpha = 5\%$ is the desired overall significance level, $T = 6$ is the number of comparisons).

Finally, we measure and compare the vulnerability resolution times between the bot and manual fixes. Since a vulnerability that has not been addressed by the end of the observable period can neither be attributed to a bot nor a human, for this analysis, we have no censored observations. Therefore, performing survival analysis is redundant, and we assess the distributions of the bot and human fixes through violin- and box- plots. To statistically verify the difference, we utilize

the one-sided non-parametric Mann-Whitney U test [73] at a standard 5% significance level. The choice of the one-sided alternative is motivated by intuition that vulnerability fixed attributed to Dependabot take less time than the manually implemented ones.

Chapter 5

Results

5.1 How often do developers merge Dependabot security updates?

We find that 56.75% of 4317 non-open Dependabot security updates ended up being merged. Comparing this to the merge ratio for the security pull requests generated by Dependabot-preview (65.42%), as reported by Alfadel *et al.* [23], we find an almost 9% drop in receptivity. While this difference is not inordinate, we conjecture that there are three core reasons that could explain it. First, the auto-merge functionality, which is unique to Dependabot-preview, allows the bot to merge its own pull requests without human intervention. Hence, it is likely that when a security problem is identified, the version increment that has to be performed to eliminate the vulnerability is very small. The less the increment is, the more plausible that the upgrade will not break the code. Moreover, given its purpose, it is reasonable to expect that the maintainers of the projects, which adopt Dependabot-preview, are more inclined to upgrade their dependencies. At last, unlike the other dependency management tools, Dependabot may be deployed at the repository automatically, given that it meets the prerequisites specified by GitHub. We conjecture that in this scenario, developers can get confused and hesitate to merge a pull request of an unknown bot.

However, analysing the overall merge ratio does not allow to discover and distinguish between different levels of interaction with the bot. As such, consider Figure 5.1 that captures the distribution of the merge ratios for the four groups of projects, categorized based on the number of security pull requests received (see Section 4.2.1). The dotted line connects the medians of the distributions, whereas the straight while line indicates a fair merge ratio (50%). The colours of the violin plots transmit the proportions of the projects that belong to each group. Concerning the first group, *i.e.*, projects with a very low number of security updates, we observe that, predominantly,

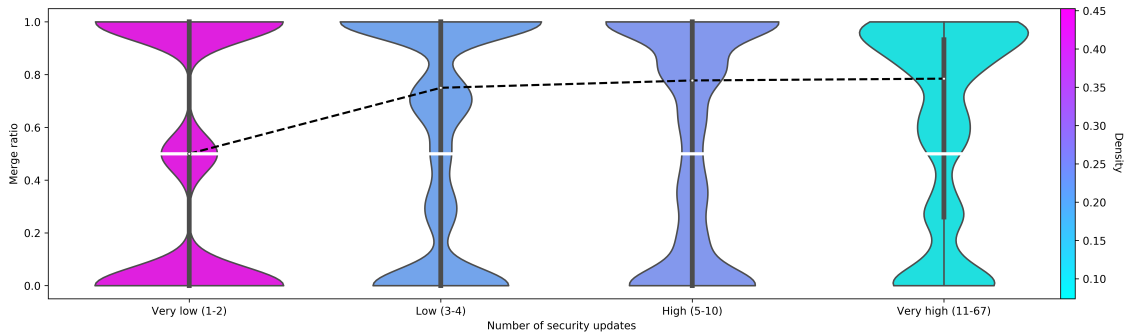


Figure 5.1: Violin plots for the distributions of the merge ratios across the four groups of projects.

the project maintainers either merge all of the automated pull requests or none of them. In fact, the two proportions of projects, each capturing one of these two extremes, are almost identical. A possible explanation to the group of the non-responsive projects is that, as discussed earlier, Dependabot was activated automatically, without the developer consent. Another rationale that explains this observation is that the maintainers of the projects with a very low number of security updates have yet to assert the reliability of the tool.

Shifting the focus to the other three groups, we observe that the majority of the projects are highly willing to merge the pull requests generated by Dependabot. In fact, for the groups with a low and high number of security updates, we find that at least a quarter of the projects merge all of the automated suggestions. On the other hand, at least another quarter of the projects merge none of the pull requests, although there is a notable dominance of the practice of merging every security update. Perhaps surprisingly, this practice can even be observed in case a project has received a very high number of security updates amid the observation period. A possible explanation to this behaviour is that some developers prefer to implement the modifications manually, as evidenced by previous studies [23, 74].

Findings. Developers tend to either merge all or none of the security updates. For the projects with less than or exactly two security updates, this tendency is the most evident, with equally frequent occurrences of both extremes. In turn, for most of the projects that received more than two security updates, developers merge at least the majority, and more often, every security update. Otherwise, most frequently, developers accept none of the automated updates, regardless of the number of suggestions received.

5.2 How frequently do developers fix a vulnerable dependency manually in the presence of the Dependabot security update?

The results suggests that 53.63% of the examined vulnerabilities are mitigated by Dependabot, 30.38% of vulnerabilities in dependencies are resolved manually, and the remaining 15.99% have not yet been fixed. That is, almost a third of the vulnerabilities in our dataset were fixed by a human, despite the availability of the Dependabot security updates targeting them. Nevertheless, the data also suggests that over half of them are fixed by the bot. Taking into account the percentage of unresolved vulnerabilities, the bot fixes occur 1.8 times as often as the manual ones, which implies that, overall, the task of vulnerability resolution is to a greater extent delegated to Dependabot.

Continuing with the analysis, in Table 5.1, we report the results on the project group level. The first meta-column, *i.e.*, the first two sub-columns, provide the proportions of the vulnerabilities that have and have not been fixed. As one can observe, there is a clearly noticeable difference in the proportions of fixed and not fixed vulnerabilities. In fact, evaluation of the Pearson's χ^2

Table 5.1: The percentages of human/bot addressed and non-resolved vulnerabilities per project group.

Group	Responses to vulnerabilities		Fixed by	
	Fixed	Not fixed	Bot	Human
Very low	76.74%	23.26%	53.08%	46.92%
Low	90.96%	9.04%	61.74%	38.26%
High	86.25%	13.75%	63.59%	36.41%
Very high	84.21%	15.79%	71.38%	28.62%
Total	84.01%	15.99%	63.84%	36.16%

Table 5.2: Computed p-values for the pairwise comparisons between the project groups. Significance: '***' < 0.001, '**' < 0.01, '*' < 0.05. Blue cells highlight the cases when $p < 0.05$.

	Very low	*Low*	*High*	*Very high*
Very low		7.18e-15***	4.50e-09***	1.63e-05***
Low	1.27e-03**		6.87e-04***	7.75e-06***
High	1.35e-05***	4.02e-01		1.11e-01
Very high	2.03e-14***	1.35e-05***	2.47e-05***	

Upper triangle: ***fixed vs not fixed***; lower triangle: ***fixed by bot vs fixed by human***

test suggests that this difference is significant ($p = 4.28e\text{-}15$) with a non-trivial (small) effect size $\phi_V = 0.12$. We observe that the lowest percentage of resolved vulnerabilities belongs to the group of projects with a very low number of security updates received. This finding may provide an additional explanation for the highest share of projects with minimal receptivity (merges) to the security updates observed for the projects that pertain to this group - developers' concerns do not intersect with the security issues. That is, developers do not accept the suggestions of the bot as a consequence of indifference to vulnerabilities in dependencies. Perhaps surprisingly, we find that the highest ratio of the resolved vulnerabilities belongs to the group of projects with the low number of security updates - a 14% difference in comparison to the first group. Furthermore, as the number of security updates increases (*low* $\longrightarrow$ *high* $\longrightarrow$ *very high*), the proportion of the unaddressed security vulnerabilities raises as well. A potential explanation to this tendency is that developers get overwhelmed by the vulnerabilities in their dependencies. The high number of vulnerabilities or their frequent occurrence, in turn, could be consequenced by the size and complexity of the project, and effectively, its dependency tree. In support of this conjecture, comparing the percentages of the bot and human security fixes captured through the second meta-column of Table 5.1, we find that for every subsequent group of projects, there is a greater delegation of the security fixes to Dependabot ($p = 6.38e - 14 < 0.05$ and $\phi_V = 0.12$, *i.e.*, the difference is significant with a small effect size). This can be due to a more extensive experience with the bot, as evidenced in the recent study of Alfadel *et al.* [23] that reports the experience to have a significant impact on the time to merge a bot-proposed pull request. On the other hand, the greater extent of delegation of vulnerability resolution to the bot can be well-explained by (1) the complexity of the project and the relationship between its dependencies that obstructs the ability to address a vulnerability manually, without re-computing the dependency tree, (2) the effort and time needed to address the vulnerabilities, given their high number, or (3) both.

However, as evidenced by the results of the post hoc pairwise comparisons reported in Table 5.2, the truth likely lies in between, *i.e.*, both the experience with the bot and the project complexity play a role. We find that in each case but two, the difference is significant (overall significance level $\alpha = 5\%$). The first such case is the trivial difference in the vulnerability response between the *high* and *very high* groups of projects, despite a significant discrepancy in the delegation of the security fixes. This suggests that the majority of the projects that belong to the second group compensate for the increased complexity (or the number of vulnerabilities) by delegating the (added) security workload to a bot; hence, a considerable increase in the proportion of security fixes made by the bot without an improvement in the extent of the vulnerability resolution, *i.e.*, no increase in the proportion of addressed vulnerabilities. The second case is the trivial difference in the delegation of the security fixes between the *low* and *high* groups of projects, despite a significant discrepancy in the vulnerability response. This suggests that the projects associated with the latter group could not compensate for the added complexity by distributing the vulnerability resolution to Dependabot due to lack of experience, or also, trust.

Findings. Almost a third of the vulnerabilities identified by Dependabot are fixed manually. Nevertheless, the majority (two thirds) of the fixes are made by merging a security update generated by the bot. In fact, the greater the number of security updates a project received, the more inclining the maintainers are in delegating the task of resolving vulnerable dependencies to Dependabot. Despite this, the maintainers of the projects that receive more security updates tend to get overwhelmed by the vulnerabilities and address less of them. The exclusions are the maintainers of the projects with less than or exactly two security updates, which have addressed the lowest observed percentage of identified vulnerabilities, suggesting indifference to security vulnerabilities in dependencies.

5.3 Why do developers ignore the suggestion of Dependabot or decide to address the vulnerability manually?

Concerning the developer motivation, we find that in **31.92%** (68) of the cases with an explicitly transmitted rationale (213), the decision to not merge a Dependabot security update and address the vulnerability manually stems from the ***project management peculiarities***. For instance, *external management* of the project (50), *i.e.*, the repository acts as a mirror for the project, whereas the development and management are mediated through another third-party platform (*e.g.*, Gerrit Code Review). Another cause that belongs to this group of challenges is that a security update gets *closed automatically* (11) by another bot, deployed as a consequence of the maintainer reluctance to employ the pull-based development model, *i.e.*, the project developers do not use pull requests and reject them automatically. Developers can also be forced to implement the suggestion of the bot manually when the repository maintainers enforce the policy of only merging the contributions of those who have signed the *contributor license agreement* (6). In this scenario, developers cannot whitelist the bot, and thus, accept a security update. Additionally, we also observe that developers can decide to *postpone the merge* (1) until the next milestone but eventually forget to do so.

In line with previous studies [28, 29, 56], we find that one of the biggest developer concerns when deciding to update are ***compatibility challenges***, which cover **27.70%** (59) cases. Most frequently, developers claim that an update incurs *breaking changes* (40), which require additional modifications to the source code and adjustments to the other project dependencies. That is, a sole upgrade to the dependency files, as proposed by Dependabot, breaks the software. In fact, we also find that developers can decide to dismiss the security update due to the *absence of tests* (1), *i.e.*, without certainty on breaking changes but in fear of compatibility issues. Sometimes developers explicitly asks Dependabot to *ignore dependency* as a whole (14) or *ignore minor* versions of the upstream package specifically (1). Alternatively, in certain scenarios, developers signify aspiration to upgrade to a more recent release, *i.e.*, a *higher version* (3) than suggested by the bot, to also benefit from the increased performance or new features.

Another source of motivation to resolve a vulnerable dependency manually or ignore the automated suggestion is the ***dependency usage***, which accounts for the **18.31%** (39) of the reasoned cases. On several occasions developers recognise that the dependency with a known vulnerability is *unnecessary* (26) to the project, *i.e.*, either not used in the project at all or belongs to an obsolete sub-module in the repository. Under this circumstance, developers decide that it is more convenient and rational to remove the dependency or the redundant sub-module that sources it rather than bring it to a more recent release, as proposed by Dependabot. This suggests that an ability to analyse the usage of the dependency across the repository could make Dependabot a more effective and useful tool. We also observe that developers often reject a security update, stating that they are interested in the *manifest updates only* (13), *i.e.*, transitive dependencies are outside of the developers' concerns.

As expected, we also find that some of the ***bot limitations*** may also impact the decision to not accept its contributions - **10.33%** (22). First, there are scenarios when developers highlight

that the vulnerable dependency has been *already resolved* (9) on the development branch by the moment Dependabot generated a corresponding security update. Indeed, Dependabot monitors only one branch, the default one, which often is the production branch, averse to the one that developers use to prepare a release. Therefore, the fix that is already applied to the development branch has not been transmitted to the production one yet at the time the bot generated the security update. In fact, the absence of an opportunity to select the desired branch results in another problem - developer reject the suggestion claiming that Dependabot proposes a merge to the *wrong branch* (4). That is, developers would have merged the suggestion, provided it was based on the branch of their choice. Furthermore, sometimes developers are not satisfied with the bot committing *redundant changes* (7). For instance, modifying some of the additional fields in the dependency files (*e.g.*, the path to the registry) or the magnitude of the update, as in the cases when Dependabot recomputes almost the entire dependency tree. We also find rare instances when the developers would rather have multiple security updates *grouped* (1) into a single pull request, or the scenario when the developers prefer to instead manually implement an *advanced* fix (1) that addresses more vulnerabilities using other sources of the security advisory information. Overall, majority of the aforementioned issues concern the limited configuration settings provided by Dependabot, which is one of the recurrent issues in bot adoption, following the study of Wessel *et al.* [86].

We also find the developers expressing ***bot dissatisfaction*** in **9.39%** (20) of the cases. In particular, there are cases when developers purposely removed all the lock files to *prevent future security alerts* (11) and the cases when developers explicitly mark security updates as *spam* (5). This suggest that, to an extent, Dependabot generates noise, which is the most recurrent and central problem of interacting with software bots [83, 85, 86]. In addition, we observe several cases when developers explicitly state their *reluctance* (4) to use Dependabot, which was deployed automatically.

Finally, there are three reasons we find that are too rare to form their own group and too distinct to join the others. We put them under the notion of the ***misc*** reasons that cover **2.35%** (5) of the cases. In particular, *confusion* (3) experienced by the developers either due to the title or the contents of the security update (or a mismatch between the two), leading to a rejection of Dependabot's proposition; the case when Dependabot security update was generated as a result of a *trial run* (1) by a developer; and at last, the case when the developers decided decided that update addresses a vulnerability with the *tolerable severity* (1), thus unnecessary. The results are summarised in Figure C.1 (Appendix C).

Findings. The two most common motivations to ignore the suggestion of Dependabot or address the vulnerability manually are the use of the third-party platforms for development and (the risk of) breaking changes. From the side of the tool itself, the limited configuration is the most frequent source of adoption problems. In particular, there is a lack of the choice of the branch for Dependabot to monitor. Additionally, some developers report Dependabot liable to noise generation, which is a recurrent challenge for the majority of software bots. Nevertheless, we do not observe this problem to be reported frequently and widely enough to conclude it as experienced by a significant number of developers.

5.4 How long does it take to address a vulnerable dependency identified by Dependabot?

Figure 5.2 shows the survival curve for the event "vulnerability identified by Dependabot is addressed" and Table 5.3 reports the survival probability for certain times x, *i.e.*, the probability an arbitrary vulnerability in dependencies is addressed after x days since the vulnerability is identified by Dependabot. First, we observe that the likelihood of a vulnerability to remain unaddressed within the first day is less than 70%. In other words, it is expected that almost a third of the

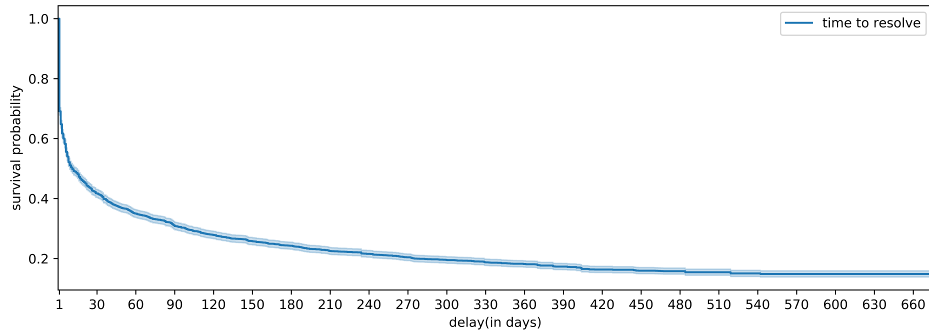


Figure 5.2: Survival curve for the event "vulnerability identified by Dependabot is addressed".

Table 5.3: Survival probability P_S of a vulnerability identified by Dependabot.

x (in days)	1	7	14	21	30	60	90	180	365
$P_S(x)$	0.69	0.54	0.49	0.45	0.42	0.35	0.31	0.24	0.18

vulnerabilities in dependencies are addressed within 24 hours since Dependabot reported the security problem, *i.e.*, instantiated a corresponding security update. Accordingly, the expectancy is that the majority of the vulnerability advisories are responded to within the first two-three weeks. As such, we conclude that predominantly developers respond to the suggestion of Dependabot promptly. Nevertheless, based on the acquired statistic, a vulnerable dependency identified by the bot remains unaddressed for over a year with an 18% likelihood, implying that almost one over five vulnerabilities affects the users of the dependent projects for at least a whole year since the advisory was published. This suggests that there is still a need to strengthen developer motivation to take security vulnerabilities in dependencies with care and urgency.

Concerning the analysis on the relation between the survivability and severity of a vulnerability, consider Figure 5.3 capturing the survival curve and Table 5.4 that reports the survival probability for a set of durations. One can immediately observe that the level of the severity is negatively correlated with the survival probability when comparing critical and high severity vulnerabilities to moderate and low. In fact, there is also a clear difference between the first two classes of vulnerabilities - given an identical time span, a critical severity vulnerability is always less likely to remain unaddressed. The dominance is even more emphasised when comparing to the low and moderate severity vulnerabilities - a critical severity vulnerability reported by Dependabot has, on average, 15-20% more chances to receive an actionable response, given the same period of time. For instance, we observe that a critical severity vulnerability is 1.5 times more likely to be addressed within the first month than a low or a moderate. When it comes to an entire year since the notification, an arbitrary low severity vulnerability is 2.4 more likely to persist than a critical and 1.7 than a high. The evaluation of the pairwise log-rank tests confirms the significant difference between each severity level, excluding the comparison of low and moderate severity vulnerabilities. This suggests that developers consider the risks a vulnerability incurs

Table 5.4: Survival probability P_S of a vulnerability identified by Dependabot based on its severity.

	Severity \ x (in days)	1	7	14	21	30	60	90	180	365
$P_S(x)$	*Low*	0.74	0.60	0.57	0.53	0.50	0.44	0.39	0.33	0.26
	Moderate	0.76	0.61	0.55	0.52	0.48	0.40	0.36	0.30	0.23
	High	0.65	0.50	0.44	0.41	0.37	0.32	0.27	0.21	0.15
	Critical	0.62	0.44	0.40	0.36	0.33	0.26	0.22	0.15	0.11

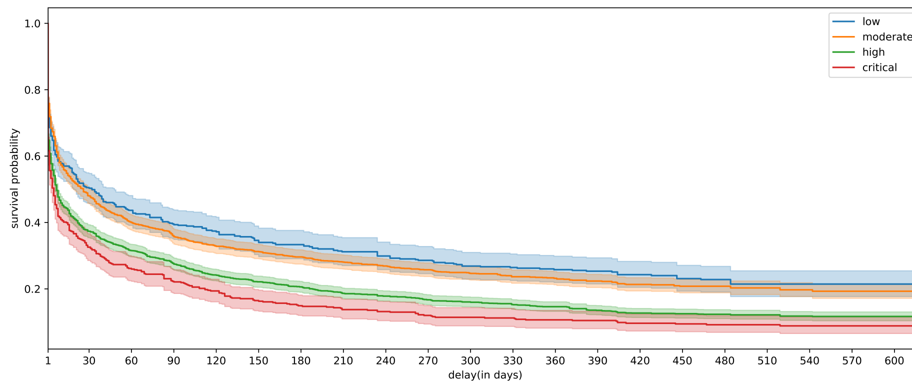


Figure 5.3: Survival curves for the event "vulnerability identified by Dependabot is addressed" based on the severity level.

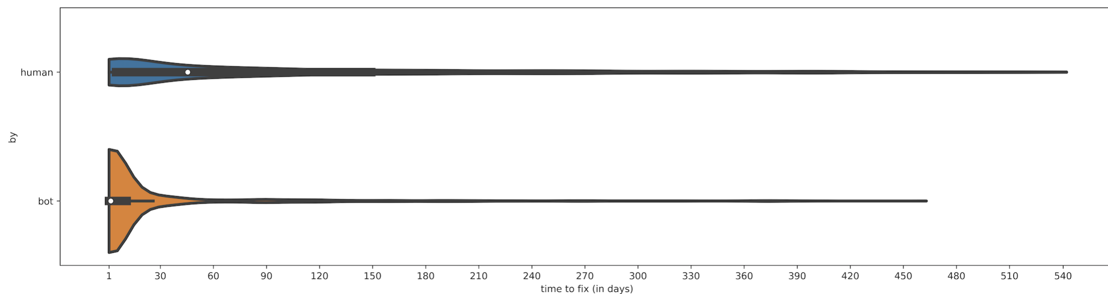


Figure 5.4: Violin plots for the distributions of the bot and manual fixing times.

when prioritizing a dependency update.

As expected, comparing the bot fixes and the ones implemented manually, we find that the first take significantly less time than the latter (0.01% significance level). As can be observed from Figure 5.4 that shows the distributions of the fixing times, the difference between the two classes of vulnerability resolutions is vast. Roughly 50% of the security updates are merged within a day and another 25% within eleven days. On the other hand, it is only less than 25% (precisely, 18%) of the manual fixes that are implemented within a day, whereas half of them take at least 1.5 months. This suggests that either a security fix is executed rapidly with the bot or requires far more time to be addressed by developers manually.

> ***Findings.*** Predominantly, developers are proactive in addressing vulnerable dependencies identified by Dependabot - less than half of the vulnerabilities persist for more than two weeks. Furthermore, developers follow the ground truth advice on treating security concerns, *i.e.*, prioritize the vulnerabilities based on the risks they incur, reflected through the severity level assigned.

Chapter 6

Discussion

In this chapter, we correlate the obtained findings and discuss their implications. First, in Section 6.1, we cover the implications for practitioners. Next, in Section 6.2, we present the actionable advice for the maintainers of Dependabot to increase user satisfaction and engagement with the tool. Accordingly, in Section 6.3, we review the lessons learned that can be helpful for fellow researchers that investigate software bots, and in Section 6.4, we discuss the problems that emerge from this work. Lastly, in Section 6.5, we discuss the validity of our study.

6.1 Implications to Practitioners

Usage of Dependabot is predominantly a positive experience The observed bimodal distribution of merge rates across all the selected projects with the majority of samples concentrated at the extremes (recall Figure 5.1) implies that if project maintainers merge at least one security update, it is likely that they will continue accepting all (or the vast majority of) the future Dependabot suggestions. In other words, it is unlikely that maintainers stop using Dependabot or use it to a lesser extent once committed to it. From this, we conclude that either project maintainers rarely face negative experiences with Dependabot or that its benefits severely outweigh the cons. The qualitative analysis confirms this, as we only observe dissatisfaction with the tool in roughly 9% of explicitly motivated rejections. In fact, these belong to the projects that do not merge the suggestions of Dependabot at all. Furthermore, amid the labelling procedure, we notice positivity as the most frequent sentiment when developers express emotions towards the tool. For example, "Good bot! :)" or "Thanks Dependabot!". Based on this, we advise developers to leverage Dependabot for keeping dependencies secure. We also encourage developers that, to this date, have not merged a single Dependabot security update received to at least attempt to use it for some trial period.

When addressing a vulnerable dependency manually, developers should strive to do so more rapidly. As a whole, developers react to vulnerabilities in a timely manner, addressing most of them within the first two-three weeks since the corresponding pull request is generated (recall Figure 5.2). However, 30.38% of vulnerabilities in dependencies identified by Dependabot are resolved manually, whereas the majority of manual fixes take at least one and half months to get approached (recall Figure 5.4). Despite that a manual update may require additional costs due to breaking changes, which we observe as a common rationale to ignore vulnerability or address it without Dependabot, we advocate that the fixing times could get reduced by prioritizing security concerns over other tasks. Alternatively, we advise project maintainers to perform regular updates to dependencies to minimise the difference between the installed release and the non-vulnerable one, *i.e.*, reduce the risks of breaking changes for when a vulnerability needs to be addressed.

6.2 Implications to Dependabot Maintainers

Familiarise developers with Dependabot. For the maintainers of the projects with less than or exactly two security updates, which is the largest group of projects (roughly 45%), the tendency to either merge all or none of the security updates is the most evident. We suggest that the reason for this behaviour is two-fold, both of which stem from the automatic deployment of Dependabot without the developer consent. First, some developers and maintainers can get confused and refuse to merge a pull request of an unknown bot that starts interacting with the repository without warning and instead address the identified vulnerability manually. This is confirmed by the fact that the highest observed percentage of developer fixes belong to this group of projects (47%, recall Table 5.1). Accordingly, the problem is the lack of information about the bot, or its immediate accessibility upon deployment, which is one of the main challenges in human-bot interactions. To address this, we recommend GitHub to always deploy Dependabot with an introductory or a welcoming message that explains the purpose of the bot, how it works, and how to interact with it, effectively increasing user familiarity with the tool.

Promote the importance to address vulnerable dependencies. The second reason for the tendency to either merge all or none of the bot suggestions is that some developers and maintainers that started receiving Dependabot security updates automatically are not concerned about security issues in their dependencies. In support of this, we find that the highest observed percentage of unaddressed vulnerabilities pertains to this (*very low* number of security updates) group of projects (23%, recall Table 5.1). Following this, GitHub needs to promote the importance to address vulnerable dependencies. One way to achieve this is to publicly display the number of unaddressed security alerts on the repository page, similar to the *badges*, the presence of which is known to correlate with more involved management of vulnerabilities in dependencies [80]. Accordingly, this badge-like element is used as a signal of the project quality to its users, effectively introducing a gamification mechanism to motivate developers [24, 34, 55].

Enable integration of Dependabot with third-party services. Investigating the developer motivation to ignore a vulnerable dependency or mitigate it manually, we observe that the reasoned cases of rejected security updates are often explained by the use of external tools to manage the repositories. In this scenario, the repository hosted on GitHub is not used for development purposes but only as a mirror to increase accessibility to the source code for the users. Since Dependabot is bounded to the pull-request based contribution mechanism of GitHub, maintainers of such projects can not leverage the bot to address security concerns in dependencies. As such, we recommend GitHub to also distribute the services of Dependabot as a command-line interface or through REST API, the uniform programming interface for the web services.

Increase configurability of Dependabot and provide better control over bot actions. Whereas limited configuration is a common issue in bot adoption [86], GitHub does not provide any options to control the behaviour of Dependabot. As concluded through qualitative analysis, the most recurrent need of the developers is to select the branch on which Dependabot should operate. Moreover, as reported by developers, due to the absence of this setting, Dependabot can generate false alarms, *i.e.*, alert the vulnerable dependency only present on the outdated branch. Another setting that would allow for better tailoring towards user needs is to limit the number of open security updates, as we observe that developers can get overwhelmed by them, and ultimately, address a lower percentage of vulnerabilities. Alternatively, Dependabot could allow project maintainers to prioritise reception of security based on the vulnerability severity level. This should be an effective prioritisation factor as we observe a strong correlation between the survivability of a vulnerability and the severity level assigned to it.

Introduce dependency usage analysis. In almost one over five reasoned cases of rejecting Dependabot suggestion, developers argue that either the vulnerable dependency or the module that sources it is not actually used in the project. Therefore, for these scenarios, an alert generated by Dependabot is a false alarm. Therefore, we recommend enhancing the analysis performed by Dependabot by scanning the source files to identify whether the package identified as vulnerable is imported, *i.e.*, used in the code, or not. As a side effect, generating fewer false alarms would

also reduce the overall number of security updates, and so the likelihood of developers getting overwhelmed by them.

6.3 Implications to Researchers

Ensure alignment in the goals of the analysed bots. We observe that developers merge a security update generated by Dependabot in 56.75% of the cases. This statistic does not align with the results reported in previous works that investigate developer receptivity towards pull request bots. For instance, Wyrich *et al.* [88], who aimed to compare the interaction with automatically and manually created pull requests, report that only 37.38% of the bot pull requests in their collection ended up being merged. The difference in the results could be contributed by the extensive project filtering that we perform in our work. For this project, we focus on actively maintained and matured JavaScript projects, contrary to Wyrich *et al.*, which do not define criteria that restrict the scope of their work. However, we are more inclined to consider the tool selection as the main factor contributing to the observed discrepancy - Wyrich *et al.* do not distinguish between the bots that are pull request authors on GitHub and analyse them as a whole. Indeed, different bots support developers in different tasks and goals. This is further confirmed when comparing our results to the ones reported by Mirhosseini *et al.* [70] in their work on the usage of Greenkeeper, which is a bot that provides automated pull requests upgrading stale dependencies. The authors focused on starred and non-forked JavaScript projects with at least 20 commits and found that only 32% of pull requests generated by Greenkeeper were actually merged, which is 1.8 times less than the percentage we observe in this work. Despite that Greenkeeper and Dependabot are designed to update the dependencies and leverage an almost identical developer interaction mechanism, *i.e.*, pull requests, the different goals they support play a role in their receptivity. As such, when analysing the bot usage among developers, we suggest separating them based on the goals and tasks these bots are designed to fulfil.

Small differences between seemingly identical bots may lead to different findings. The alignment in the goals of examined bots is necessary but, perhaps, not sufficient to ensure identical levels of developer interaction with them. Despite that Dependabot was based on Dependabot-preview, and from the first glance is an identical tool, it can be deployed at a repository automatically, lacks auto-merge functionality, and aims to suggest the minimum required increment in the updated release. Accordingly, in our work, we observe a merge ratio that is 1.2 times less than the one reported by Alfadel *et al.* [23] in their study on the usage of Dependabot-preview and its security pull requests. As such, the findings collected for one tool do not necessarily apply to another similar.

Developers rarely comment on rejected bot pull requests. Extracting developer motivation based on the commit messages and pull request discussions allowed us to discover many divergent reasons to remediate a vulnerable dependency manually or ignore it. Nevertheless, we observe that the majority of the collected security updates rejected by the developers are left uncommented. Moreover, such an approach consumes a very large amount of time. Indeed, in this project, this technique is the only feasible option as it is unreasonable to survey developers on their past decisions. However, for researchers that aim to investigate the developer opinion towards pull-based bots, we advise leveraging a series of interviews or an open-ended survey to mine them.

6.4 Future work

Continue the investigation on the reasons to reject suggestions of Dependabot. As discussed in Section 6.3, despite that we have discovered many unique motivations to address a vulnerability manually or ignore it, developers predominantly reject security updates without explaining their decision. This phenomenon suggests that the relevance of the identified reasons to the majority of developers and project maintainers requires further investigation. We argue that

the results acquired in this study can be regarded as a stepping stone for the future work that investigates the interaction between developers and Dependabot. In particular, with the reasons collected, it becomes possible to conduct a close-ended survey among the developers where the respondents are required to rate or order them in line with the experienced relevance. This would provide the maintainers of Dependabot with a more accurate assessment of the priority in which the problems should be addressed.

Identify the advantages of the Dependabot interaction design. It is important to identify the problems with the tool so that they can be addressed and that the developers of the other tools could learn from them. In our work, we identify these problems and suggest the means to mitigate them. However, to further strengthen the user satisfaction with Dependabot and the other similar tools, it is also crucial to understand its convenience and advantages in interaction design. To this end, we envision two strategies. First, we suggest the qualitative approach, which boils down to conducting a series of interviews with the users of Dependabot. The other approach requires reproducing our study for the projects that use the other tools (*e.g.*, Snyk-bot or Renovate). Accordingly, correlating the difference between the tools and the difference in the extent of their adoption may reveal the advantages and disadvantages of the compared tools.

Measure the impact of Dependabot on keeping the project dependencies secure. In our work, we assess the time it takes for the developers to address a vulnerability identified by Dependabot. Nevertheless, it remains unknown to what extent Dependabot has an impact on it. The way the tool aims to assist developers is clear, and the rationale behind its usage is evident, but understanding its effect on the secureness of the projects is still required. In an optimistic scenario that the bot has a strong impact, this evidence can be used to further motivate the projects' maintainers to adopt Dependabot, effectively elevating the secure state of the dependencies. On the contrary, a scenario that there is no impact or its negligible implies that the concept of a bot that generates pull requests to update vulnerable dependencies requires refinement.

6.5 Threats to Validity

Construct validity. We find two core threats that concern the relationship between the theory and the results obtained in this study. The first threat refers to our definition of a security fix made by Dependabot. We employ a high precision strategy and only consider a fix to be contributed by the bot if Dependabot is the author of the fixing commit. Indeed, there could be instances when developers replicate the update generated by the bot or use a proxy to implement the suggested changes. However, such cases can not be identified with high confidence. Accordingly, when identifying the extent of delegation of fixing vulnerable dependencies to Dependabot, we prefer to underestimate rather than overestimate. In fact, we estimate the impact of this decision to be mild, if any, as, amid the manual inspection, we only find 32 (2%) of the cases where one could conjecture that the contents of the commit generated by the bot were exactly replicated by a developer.

The second threat is that we operationalise the left endpoint of the fixing time interval for a vulnerability as the moment the corresponding security update that addresses it was generated. The moment that developers and maintainers of a project became aware of the vulnerability in their dependencies could have taken place earlier, provided they used other sources of information. Nevertheless, as Dependabot is the uniform interface for vulnerability data, shared across all of the selected projects, this choice of the starting timestamp for a fixing event is the most optimal.

Internal validity. We find three main threats that could affect our results and are internal to our study. The first threat pertains to the quality of the obtained collection of projects and has an impact on every research question we pose in this work. For instance, the presence of abandoned projects or immutable forks could have downgraded the overall merge ratio or significantly elevate survival curves for vulnerabilities. To address this threat, we perform an extensive filtering procedure and remove the projects that have less than one commit during each month of the collection period. As such, we ensure that when a project receives a security update, it has recent activity. In fact, the average number of commits during the collection period among the

selected projects is 640, *i.e.*, approximately 53 commits per month.

The next internal threat concerns mining GitHub in general, as a large share of the repositories hosted there are used for personal projects [61], *e.g.*, homework assignments. This poses a serious threat to our analysis, as vulnerabilities in a project with no utility to individuals other than the developers of the project themselves are tolerable. That is, security concerns for the maintainers of such projects are likely to associate with the lowest priority. To mitigate this threat, we only consider engineered repositories selected using the `Reaper` tool. Despite that this tool is not perfectly accurate, given the other criteria we impose on the projects, we are confident the presence of personal repositories in the considered sample is minimal.

The third and final internal threat relates to the use of qualitative techniques. To obtain insights on developer motivation, we manually analyse the cases of identified vulnerabilities that were not addressed or mitigated manually, despite the presence of the corresponding Dependabot security update. The downside of such approach is the author bias and subjectivity. To counter this, first, we enforce a rule that the conclusion can be made only based on the information explicitly specified by the developers themselves. Furthermore, we employ a second independent analyst to repeat the manual study. Measuring the inter-rater agreement, we observe a very high value (0.96) for Cohen's κ coefficient, strengthening our confidence in the objectivity, and hence validity, of the qualitative analysis.

External validity. The last threat concerns the generalization of our findings. In our work, we solely focus on the JavaScript projects and `npm` ecosystem. Since the communities surrounding different centralised package registries deviate in policies, practices, and culture [29,42], our results do not necessarily reflect the use of automated support in the projects written the different languages and ecosystems other than `npm`. Moreover, our analysis is strictly limited to Dependabot, whose interaction traits and core logic need not correlate with the ones of the other tools alike. As such, we encourage replication of our work for the other languages and package dependency networks. We expect that different lessons learnt from other ecosystems can contribute to the better adaptation of the dependency management tools towards specific developer communities. While our results do not reflect the level of adoption and receptivity towards other tools like Dependabot, these findings hold significant relevance since this bot is the most accessible tool that provides automated security updates.

Chapter 7

Conclusion

This work conducts an empirical study of vulnerable dependency resolution with the assistance of Dependabot in the open-source JavaScript projects hosted on GitHub. Literature suggests that maintaining dependencies secure is an overwhelming challenge to the developers, indicating a strong need for support through automation. In turn, Dependabot is the most accessible and widely deployed tool designed to mitigate this problem. However, no prior studies assess the extent of its adoption by the community and investigate the interaction with the bot along with the problems developers face when using it.

To provide insights on the usage of Dependabot and address the gap, we formulated four research questions that we answered by combining quantitative and qualitative techniques through the analysis of 4,538 security updates associated with 1,004 mature and actively maintained JavaScript projects. We studied developer receptivity towards Dependabot on different levels of granularity, investigated the practice of fixing the vulnerability manually in the presence of an automated pull request that addresses it as well as motivation to this phenomenon, and finally, assessed how proactive the developers were in remediating the vulnerable dependencies alerted by Dependabot.

To evaluate developer receptivity, we measured the overall merge ratio but also assessed its distribution across the studied projects. We found that more than half of the bot suggestions are accepted and merged. After splitting the projects into four groups based on the number of security updates received, we observed that developers either merge all or none of the suggestions. This behaviour was encountered in all four groups of projects.

Studying how often developers fix a vulnerability manually, as opposed to merging the corresponding Dependabot security update that addresses it, proved to be a complex problem that requires several steps. First, we restored the missing and modified entries in the public database of security advisories hosted by GitHub. Then, we reverse-engineered and implemented the algorithm leveraged by Dependabot to identify whether the project is affected by a vulnerability in the dependencies. After the validation procedure, we apply the algorithm to the modification history of each studied project. Specifically, we recursively traversed their network graphs for each vulnerable dependency identified by Dependabot to locate the earliest fixing commit that eliminates it. Completing this task required manual inspection, whose correctness we confirmed with the help of the second reviewer. We found that when developers do not accept a security update, they eventually address the associated vulnerability manually - vulnerabilities identified by Dependabot were fixed manually twice more often than ignored. Nevertheless, the vast majority of the identified fixes were performed by merging a security update generated by the bot.

To discover the challenges developers face when interacting with Dependabot, for each rejected security update, we manually investigated the textual artefacts associated with it or a fixing commit, provided the associated vulnerability was addressed manually. To counter the subjectivity in the qualitative analysis, we verified the inter-rater agreement between the original and the second reviewers. We found 22 unique reasons to ignore a vulnerability in dependencies or mitigate it without the bot, which we grouped into six categories. We observed that the two prevailing motivations are the use of third-party platforms for development and (the risk of) breaking changes.

Moreover, the limited configuration of the bot is one of the most frequent sources of adoption problems.

At last, to capture the degree to which the developers react to the identified vulnerabilities promptly, we leveraged the survival analysis. We found that developers are often proactive in addressing vulnerable dependencies identified by Dependabot. In particular, less than half of the vulnerabilities persist for more than two weeks. We also found that developers prioritize the vulnerabilities based on the risks they incur, *i.e.*, there is a relation between the severity level assigned to a vulnerability and the time it takes for developers to react to it. However, when developers implement the fix manually, the results are not as optimistic - the majority of the studied manual fixes took more than a month.

Based on our findings, we conclude that the majority of developers are willing to delegate the resolution of vulnerable dependencies to Dependabot and that the usage of this bot is predominantly a positive experience. However, we urge developers to spend additional efforts on manual fixes as they take a very long time, especially in comparison to automated fixes. Despite that over 50% of the security updates are merged, we identified several ways the maintainers of Dependabot could improve this statistic. First, we recommend them to familiarise developers with Dependabot and promote the importance to address vulnerable dependencies. Furthermore, we encourage to augment the bot by enabling integration with third-party services, increasing configurability of the tool, and introducing dependency usage analysis. Finally, we call for the continuation of our work by extending the findings on the reasons to reject suggestions of Dependabot through additional surveys and analysing the impact of Dependabot on keeping the project dependencies secure.

Bibliography

[1] The 2020 State of the Octoverse: Security Report. https://octoverse.github.com/static/github-octoverse-2020-security-report.pdf/. [Online] Last accessed 21 March 2021. 13

[2] About Dependabot security updates. https://docs.github.com/en/github/managing-security-vulnerabilities/about-dependabot-security-updates/. [Online] Last accessed 21 March 2021. 1, 14

[3] Bump bootstrap-sass from 3.3.7 to 3.4.1 by dependabot · pull request 1 · concur/developer.concur.com. https://github.com/concur/developer.concur.com/pull/2230. [Online] Last accessed 24 March 2021. 8

[4] Bump handlebars from 4.1.0 to 4.7.3 by dependabot · pull request 134 · entrylabs/entry-tool. https://github.com/entrylabs/entry-tool/pull/134. [Online] Last accessed 24 March 2021. 8

[5] Bump lodash-es from 4.17.11 to 4.17.15 by dependabot · pull request 3 · lane711/sonicjs. https://github.com/lane711/sonicjs/pull/3. [Online] Last accessed 24 March 2021. 8

[6] Bump lodash from 4.17.10 to 4.17.13 by dependabot · pull request 1 · ninjadotorg/handshake-i18n. https://github.com/ninjadotorg/handshake-i18n/pull/1. [Online] Last accessed 24 March 2021. 8

[7] Bump minimist from 0.0.8 to 1.2.5 by dependabot · pull request 179 · linode/developers. https://github.com/linode/developers/pull/179. [Online] Last accessed 24 March 2021. 8

[8] chore(deps): bump acorn from 5.6.2 to 5.7.4 · pull request 1008 · carbon-design-system/carbon-addons-iot-react. 20

[9] Dependabot-preview. https://github.com/marketplace/dependabot-preview/. [Online] Last accessed 21 March 2021. 1, 14

[10] Greenkeeper. https://greenkeeper.io/. [Online] Last accessed 2 February 2021. 1, 14

[11] Introducing new ways to keep your code secure. https://github.blog/2019-05-23-introducing-new-ways-to-keep-your-code-secure/. [Online] Last accessed 12 March 2021. 1

[12] Introducing security alerts on GitHub. https://github.blog/2017-11-16-introducing-security-alerts-on-github/. [Online] Last accessed 12 March 2021. 1

[13] npm trends: npm vs yarn. https://www.npmtrends.com/npm-vs-yarn. [Online] Last accessed 5 July 2021. 3

[14] Pyup. https://pyup.io/. [Online] Last accessed 12 March 2021. 1

[15] Questions & answers: Why registry.yarnpkg.com? does facebook track us? https://yarnpkg.com/getting-started/qa#why-registryyarnpkgcom-does-facebook-track-us. [Online] Last accessed 5 July 2021. 3

[16] Renovate. https://github.com/marketplace/renovate/. [Online] Last accessed 2 February 2021. 1, 14

[17] Snyk. https://github.com/marketplace/snyk/. [Online] Last accessed 21 March 2021. 1, 14

[18] What is package manager? https://www.debian.org/doc/manuals/aptitude/pr01s02.en.html. [Online] Last accessed 5 July 2021. 3

[19] Yarn: A new package manager for javascript. ://engineering.fb.com/2016/10/11/web/yarn-a-new-package-manager-for-javascript/. 3

[20] Odd Aalen, Ornulf Borgan, and Hakon Gjessing. *Survival and event history analysis: a process point of view*. Springer Science & Business Media, 2008. 2, 25

[21] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 385–395, 2017. 13

[22] Mahmoud Alfadel, Diego Elias Costa, Mouafak Mokhallalati, Emad Shihab, and Bram Adams. On the Threat of npm Vulnerable Dependencies in Node.js Applications. *arXiv preprint arXiv:2009.09019*, 2020. 9, 13

[23] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Mouafak Mkhallalati. On the Use of Dependabot Security Pull Requests. In *Proceedings of the 18th International Conference on Mining Software Repositories*, pages xxx–xxx, 2021. 2, 10, 11, 13, 24, 27, 28, 29, 36

[24] Bilal Amir and Paul Ralph. Proposing a theory of gamification effectiveness. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 626–627, 2014. 35

[25] APA. *Publication manual of the American Psychological Association*, pages 16–18. American Psychological Association, 4 edition, 1994. 24

[26] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 356–367, 2016. 9

[27] Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal statistical society: series B (Methodological)*, 57(1):289–300, 1995. 24

[28] Christopher Bogart, Christian Kästner, and James Herbsleb. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 86–89. IEEE, 2015. 30

[29] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an API: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120, 2016. 3, 11, 30, 38

[30] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001. 13

[31] Gleison Brito, Ricardo Terra, and Marco Tulio Valente. Monorepos: a multivocal literature review. *arXiv preprint arXiv:1810.09477*, 2018. 18

[32] Chris Brown and Chris Parnin. Sorry to bother you: designing bots for effective recommendations. In *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*, pages 54–58. IEEE, 2019. 10

[33] Nathan Cassee, Gustavo Pinto, Fernando Castor, and Alexander Serebrenik. How swift developers handle errors. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 292–302. IEEE, 2018. 14

[34] Huseyin Cavusoglu, Zhuolun Li, and Ke-Wei Huang. Can gamification motivate voluntary contributions? the case of stackoverflow q&a community. In *Proceedings of the 18th ACM conference companion on computer supported cooperative work & social computing*, pages 171–174, 2015. 35

[35] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering*, 26(3):1–28, 2021. 9, 13

[36] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering*, 26(3):1–28, 2021. 11

[37] Filipe Roseiro Cogo, Gustavo Ansaldi Oliva, and Ahmed E Hassan. An empirical study of dependency downgrades in the npm ecosystem. *IEEE Transactions on Software Engineering*, 2019. 13, 14

[38] Jacob Cohen. *Statistical power analysis for the behavioral sciences*, pages 224–226. L. Erlbaum Associates, 2 edition, 1988. 24

[39] Eleni Constantinou and Tom Mens. An empirical comparison of developer retention in the rubygems and npm software ecosystems. *Innovations in Systems and Software Engineering*, 13(2):101–115, 2017. 25

[40] Joël Cox, Eric Bouwers, Marko Van Eekelen, and Joost Visser. Measuring dependency freshness in software systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 109–118. IEEE, 2015. 9

[41] Harald Cramér. *Mathematical methods of statistics*, chapter 21, page 282. Princeton University Press, 1946. 24

[42] Alexandre Decan, Tom Mens, and Maëlick Claes. An empirical comparison of dependency issues in oss packaging ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 2–12. IEEE, 2017. 9, 13, 25, 38

[43] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the evolution of technical lag in the npm package dependency network. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 404–414. IEEE, 2018. 9, 13

[44] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 181–191, 2018. 1, 9, 11, 13, 16, 17, 25

[45] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24(1):381–416, 2019. 13

[46] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2187–2200, 2017. 9

[47] Tapajit Dey, Sara Mousavi, Eduardo Ponce, Tanner Fry, Bogdan Vasilescu, Anna Filippova, and Audris Mockus. Detecting and characterizing bots that commit code. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 209–219, 2020. 14

[48] Tapajit Dey, Bogdan Vasilescu, and Audris Mockus. An exploratory study of bot commits. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 61–65, 2020. 14

[49] Linda Erlenhov, Francisco Gomes de Oliveira Neto, Riccardo Scandariato, and Philipp Leitner. Current and future bots in software development. In *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*, pages 7–11. IEEE, 2019. 14

[50] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for javascript ide services. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 752–761. IEEE, 2013. 13

[51] Thomas R Fleming and David P Harrington. *Counting processes and survival analysis*, volume 169. John Wiley & Sons, 2011. 25

[52] William B Frakes and Kyo Kang. Software reuse research: Status and future. *IEEE transactions on Software Engineering*, 31(7):529–536, 2005. 1

[53] Georgios Gousios. The ghtorent dataset and tool suite. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 233–236. IEEE, 2013. 13

[54] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean GHTorrent: GitHub data on demand. In *Proceedings of the 11th working conference on mining software repositories*, pages 384–387, 2014. 1

[55] Scott Grant and Buddy Betts. Encouraging user behaviour with achievements: an empirical study. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 65–68. IEEE, 2013. 35

[56] Nicole Haenni, Mircea Lungu, Niko Schwarz, and Oscar Nierstrasz. Categorizing developer information needs in software ecosystems. In *Proceedings of the 2013 international workshop on ecosystem architectures*, pages 1–5, 2013. 11, 30

[57] Joseph F Healey. *Statistics: A Tool for Social Research*, pages 316–317. Wadsworth Cengage Learning, 8 edition, 2009. 24

[58] JI Hejderup. In dependencies we trust: How vulnerable are dependencies in software modules? Master's thesis, Delft University of Technology, 2015. 13

[59] David C. Howell. *Statistical methods for psychology*, page 165. Wadsworth, 5 edition, 2007. 24

[60] Jie Huang, Nataniel Borges, Sven Bugiel, and Michael Backes. Up-to-crash: Evaluating third-party library updatability on android. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 15–30. IEEE, 2019. 11

[61] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101, 2014. 13, 38

[62] Edward L Kaplan and Paul Meier. Nonparametric estimation from incomplete observations. *Journal of the American statistical association*, 53(282):457–481, 1958. 25

[63] David Kavaler, Asher Trockman, Bogdan Vasilescu, and Vladimir Filkov. Tool choice matters: Javascript quality assurance tools and usage outcomes in github projects. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 476–487. IEEE, 2019. 10, 14

[64] Stephen Kokoska and Daniel Zwillinger. *CRC standard probability and statistics tables and formulae.* Crc Press, 2000. Section 2.2.24. 23

[65] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, 2018. 1, 9, 10, 13

[66] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems.* Springer Science & Business Media, 2007. 23

[67] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. *arXiv preprint arXiv:1811.00918*, 2018. 9

[68] Bin Lin, Gregorio Robles, and Alexander Serebrenik. Developer turnover in global, industrial open source projects: Insights from applying survival analysis. In *2017 IEEE 12th International Conference on Global Software Engineering (ICGSE)*, pages 66–75. IEEE, 2017. 25

[69] Dongyu Liu, Micah J Smith, and Kalyan Veeramachaneni. Understanding user-bot interactions for small-scale automation in open-source development. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–8, 2020. 10

[70] Samim Mirhosseini and Chris Parnin. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 84–94. IEEE, 2017. 10, 13, 36

[71] Ron C Mittelhammer, George G Judge, and Douglas J Miller. *Econometric foundations.* Cambridge University Press, 2000. 25

[72] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, 2017. 13, 14, 15

[73] Nadim Nachar et al. The mann-whitney u: A test for assessing whether two independent samples come from the same distribution. *Tutorials in quantitative Methods for Psychology*, 4(1):13–20, 2008. 26

[74] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1513–1531, 2020. 2, 10, 28

[75] Karl Pearson. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302):157–175, 1900. 24

[76] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011. 14

[77] Gede Artha Azriadi Prana, Abhishek Sharma, Lwin Khin Shar, Darius Foo, Andrew E Santosa, Asankhaya Sharma, and David Lo. Out of sight, out of mind? how vulnerable dependencies affect open-source projects. *Empirical Software Engineering*, 26(4):1–34, 2021. 9

[78] Ioannis Samoladas, Lefteris Angelis, and Ioannis Stamelos. Survival analysis on the duration of open source projects. *Information and Software Technology*, 52(9):902–922, 2010. 25

[79] Herbert H Thompson. Why security testing is hard. *IEEE Security & Privacy*, 1(4):83–86, 2003. 1

[80] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In *Proceedings of the 40th International Conference on Software Engineering*, pages 511–522, 2018. 13, 35

[81] Anthony J. Viera and Joanne M. Garrett. Understanding interobserver agreement: the kappa statistic. *Family medicine*, 37(5):360–363, 2005. 25

[82] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45. IEEE, 2020. 9

[83] Mairieli Wessel, Bruno Mendes De Souza, Igor Steinmacher, Igor S Wiese, Ivanilton Polato, Ana Paula Chaves, and Marco A Gerosa. The power of bots: Characterizing and understanding bots in oss projects. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):1–19, 2018. 10, 31

[84] Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A Gerosa. Effects of Adopting Code Review Bots on Pull Requests to OSS Projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–11. IEEE, 2020. 10

[85] Mairieli Wessel and Igor Steinmacher. The inconvenient side of software bots on pull requests. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 51–55, 2020. 10, 31

[86] Mairieli Wessel, Igor Wiese, Igor Steinmacher, and Marco A Gerosa. Don't disturb me: Challenges of interacting with softwarebots on open source software projects. *arXiv preprint arXiv:2103.13950*, 2021. 10, 31, 35

[87] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the JavaScript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 351–361, 2016. 13

[88] Marvin Wyrich, Raoul Ghit, Tobias Haller, and Christian Müller. Bots don't mind waiting, do they? comparing the interaction with automatically and manually created pull requests. *arXiv preprint arXiv:2103.03591*, 2021. 10, 14, 36

[89] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm JavaScript packages. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 559–563. IEEE, 2018. 13

[90] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. An empirical analysis of technical lag in npm package dependencies. In *International Conference on Software Reuse*, pages 95–110. Springer, 2018. 9

[91] Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles, and Jesus M Gonzalez-Barahona. On the impact of outdated and vulnerable javascript packages in docker images. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 619–623. IEEE, 2019. 9

[92] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 995–1010, 2019. 13

Appendix A

Rectification of the security advisory records

A.1 minimist and acorn

A.1.1 Summary

GHSA-7fhm-mqm4-2wp7 got withdrawn and replaced by GHSA-6chw-6frg-f759 (acorn package) and GHSA-vh95-rmgr-6w4m (minimist package), as suggested by the summary of the former. The only change that occurred is that the range of the vulnerable releases for the minimist package used to be defined as $< 1.2.2$ but got modified to $(< 0.2.1) \vee (\geq 1.0.0 < 1.2.3)$

A.1.2 Evidence

As for the acorn package, there are numerous examples of the security updates[1] that suggest updating from the highest vulnerable version to the minimum non-vulnerable one, dated seconds after GHSA-7fhm-mqm4-2wp7 got published. As for the minimist package, the associated entry in the National Vulnerability Database[2], highlights that every release up to (excluding) 1.2.2 is vulnerable. Additionally, using GitHub Search, one can find many examples of the security updates[3], dated minutes after publication of the original security advisory, that suggest upgrading minimist to 1.2.2. There are also examples[4] when a security update that upgrades minimist to 1.2.2 gets superseded by an automated pull request that upgrades it to 1.2.3 on the date of the publication of the replacement advisory (GHSA-vh95-rmgr-6w4m).

A.1.3 Remediation

The publication date for the records GHSA-6chw-6frg-f759 and GHSA-vh95-rmgr-6w4m should be replaced by the one of GHSA-7fhm-mqm4-2wp7. In turn, GHSA-7fhm-mqm4-2wp7 is removed, to avoid identification of multiple vulnerabilities, since the affected release ranges intersect. At last, a special case must be made in the vulnerability deduction algorithm that, in case Dependabot

[1] https://github.com/thedevs-network/the-guard-bot/pull/113,
https://github.com/restorecommerce/handlebars-helperized/pull/4,
https://github.com/yellowled/yl-bp/pull/71

[2] https://nvd.nist.gov/vuln/detail/CVE-2020-7598

[3] https://github.com/BitGo/BitGoJS/pull/703,
https://github.com/kartotherian/kartotherian/pull/122,
https://github.com/workshopper/adventure/pull/18

[4] https://github.com/peeranha/peeranha-web/pull/187,
https://github.com/fakob/MoviePrint_v004/pull/10,
https://github.com/viewstools/morph/pull/185

suggests updating minimist to 1.2.2, the vulnerability `GHSA-vh95-rmgr-6w4m` is assigned as an associated security advisory.

A.2 concat-with-sourcemaps

A.2.1 Summary

The `GHSA-2xv3-h762-ccxv` advisory comprises an incorrect vulnerability constraint, which is used to capture the range of the affected releases.

A.2.2 Evidence

The range of the vulnerable releases is defined as $> 1.0.0.1 < 1.0.6$, while the summary of the advisory suggest that, instead, every version prior to 1.0.6 is affected. The latter is correct, as suggested by the corresponding vulnerability record in the database maintained by `npm`[5], which is included as a reference. Additionally, version 1.0.0.1 of package concat-with-sourcemaps does not exist.

A.2.3 Remediation

Replace the vulnerability range definition with $< 1.0.6$.

A.3 eslint

A.3.1 Summary

The `GHSA-jcgq-xh2f-2hfm` advisory concerning catastrophic backtracking (severity level "moderate") in eslint package of versions prior to 4.18.2 got published on June 20th, 2019, and removed from the database after November 19th, 2020.

A.3.2 Evidence

Numerous secuirty updates suggesting to upgrade eslint package to version 4.18.2[6] were generated on June 20th, 2019. The latest security update suggesting to upgrade to this version[7], which we manage to retrieve, was instantiated on November 19th, 2020. As suggested by an automated pull request raised by Renovate bot on June 20th, 2019[8], which sources GitHub Advisory Database, the `ghsa` identifier of the concerned vulnerability is `GHSA-jcgq-xh2f-2hfm`, and the affected version range is $< 4.18.2$. As derived from the following discussion[9], the severity level assigned to this vulnerability is "moderate".

A.3.3 Remediation

Add an entry to the collection of security advisories following the information provided above.

[5] https://www.npmjs.com/advisories/644/versions
[6] https://github.com/search?o=asc&q=is%3Apr+author%3Aapp%2Fdependabot+bump+eslint+from+to+4.18.2&s=created&type=Issues
[7] https://github.com/saelinklaw/ReverseServer/pull/1
[8] https://github.com/Financial-Times/n-teaser/pull/214
[9] https://github.com/uktrade/githubscan/pull/4

A.4 ecstatic

A.4.1 Summary

The `GHSA-9q64-mpxx-87fg` advisory, originally published on June 7th, 2019, got republished on April 1st, 2020.

A.4.2 Evidence

The corresponding advisory comprises three disjoint vulnerability ranges. Accordingly, the suggestions are to upgrade ecstatic package to version 2.2.2, 3.3.2, or 4.1.2. There are numerous examples of the security updates[10] that upgrade a dependency on ecstatic to one of the aforementioned versions, dated June 7th, 2019.

A.4.3 Remediation

Modify the publication date with June 7th, 2019.

A.5 ws

A.5.1 Summary

The `GHSA-5v72-xg48-5rpm` advisory used to have a different range of the vulnerable releases specified.

A.5.2 Evidence

The advisory suggests that the versions of the ws package that satisfy the constraint $(\geq 0.2.6 < 1.1.5) \vee (\geq 2.0.0 < 3.3.1)$ are vulnerable. That is, release 1.1.5 is patched. However, on the first day of publishing this advisory, there are a plethora of Dependabot security updates[11] that suggest avoiding version 1.1.5 and, instead, upgrading to 3.3.1. Furthermore, we find a pull request generated by Renovate-bot[12] on the day advisory was published that references it and suggesting that the affected versions of the ws package are $(\geq 0.2.6 < 3.3.1)$. Therefore, when the advisory was published, version 1.1.5 was deemed vulnerable. The advisory was modified to exclude the version 1.1.5 from the range of affected releases on December 9th, 2019[13].

A.5.3 Remediation

A special case must be made in the vulnerability deduction algorithm that, in case Dependabot suggests updating ws package from version 1.1.5 to any release that is higher than 3.3.0 before December 9th, 2019, the vulnerability `GHSA-5v72-xg48-5rpm` is assigned as an associated security advisory.

[10] https://github.com/search?o=asc&q=is%3Apr+author%3Aapp%2Fdependabot+ecstatic+from+to+2.2.2&s=created&type=Issues,
https://github.com/search?o=asc&q=is%3Apr+author%3Aapp%2Fdependabot+ecstatic+from+to+3.3.2&s=created&type=Issues,
https://github.com/search?o=asc&q=is%3Apr+author%3Aapp%2Fdependabot+ecstatic+from+to+4.1.2&s=created&type=Issues

[11] https://github.com/search?o=asc&q=is%3Apr+author%3Aapp%2Fdependabot+bump+ws+from+1.1.5+to+3.3.1&s=created&type=Issues

[12] https://github.com/moul/iocat/pull/14

[13] https://github.com/search?o=asc&q=is%3Apr+author%3Aapp%2Fdependabot+bump+ws+from+1.1.1+to+1.1.5&s=created&type=Issues

A.6 kind-of

A.6.1 Summary

The `GHSA-6c8f-qphg-qjgp` advisory used to have a different range of the vulnerable releases specified.

A.6.2 Evidence

The retrieved version of advisory suggests that versions of the kind-of package 6.x prior to 6.0.3 are vulnerable to a Validation Bypass. However, on the day the advisory was published, we find several Dependabot security updates[14] that suggest avoiding version 3.3.2 of the package kind-of in favour of 6.0.3. Indeed, as suggested by the following discussion[15], the advisory used to declare that every version prior to 6.0.3 is vulnerable. The latest security update that advises to avoid versions prior to 6.x is dated by November 6th, 2020.

A.6.3 Remediation

A special case must be made in the vulnerability deduction algorithm that, in case Dependabot suggests updating from version <6.x to any release that is higher than 6.0.2 before November 6th, 2020, the vulnerability `GHSA-6c8f-qphg-qjgp` is assigned as an associated security advisory.

A.7 marked

A.7.1 Summary

The `GHSA-7m7q-q53v-j47v` advisory concerning the regular expression denial of service (severity level "moderate") in marked package of versions $\geq 0.5.0 < 0.6.1$ got published on June 4th, 2019, and removed from the database after August 30th, 2020.

A.7.2 Evidence

Numerous security updates suggesting to upgrade marked package to version 6.0.1[16] were generated on June 4th, 2019. The latest security update suggesting to update to this version[17], which we manage to retrieve, was instantiated on August 30th, 2020. As suggested by a pull request raised by Renovate-bot on June 4th, 2019[18], the `ghsa` of this advisory is `GHSA-7m7q-q53v-j47v`, and the affected version range is $\geq 0.5.0 < 0.6.1$. As derived from the following discussion[19], the severity level assigned to this vulnerability is "moderate".

A.7.3 Remediation

Add an entry to the collection of security advisories following the information provided above.

[14] https://github.com/search?o=asc&q=is%3Apr+author%3Aapp%2Fdependabot+Bump+kind-of+from+3.2.2+to+6.0.3&s=created&type=Issues
[15] https://github.com/near/rainbow-bridge/issues/28
[16] https://github.com/search?o=asc&q=is%3Apr+author%3Aapp%2Fdependabot+bump+marked+to+0.6.1&s=created&type=Issues
[17] https://github.com/admdev8/autocomplete-plus/pull/4
[18] https://github.com/cats-oss/scaffdog/pull/48
[19] https://github.com/docsifyjs/docsify-cli/pull/87

Appendix B

Rater guideline

The labelling procedure consists of two stages. The first stage is dedicated to the identification of the fixing commit given a list of the candidate commits, additional information that pertains to the concerned vulnerability, and access to the repository hosted on GitHub. All the necessary knowledge and in-depth explanation of the methodology are described in Section B.1. In the second stage, a rater is required to assign four (multi-) labels to each case of fixing (or non-fixing) a vulnerable dependency for which Dependabot has generated a security update. The details are discussed in Section B.2.

B.1 First stage

B.1.1 Susceptibility to a vulnerability

Each provided case is accompanied with a list of the affected directories, i.e., (sub-)modules of a repository with a vulnerable dependency. The hosted repository is deemed to be affected by a vulnerability if at least one dependency file in either of the specified directories is found to declare a vulnerable dependency. In our study, we consider four kinds of dependency files. Namely, `package.json`, `npm-shrinkwrap.json`, `package-lock.json`, and `yarn.lock`. A rater is assumed to have a considerable level of familiarity with their structure and function.

For each case, a rater is provided with a name of the upstream package and the range(s) of its vulnerable releases. Additionally, for each range, the first patched release is specified. The `package.json` is said to declare a vulnerable dependency if the specified upstream package is present as a direct runtime or development dependency, whose highest version that satisfies its dependency constraint is vulnerable. Since each vulnerability range is upper-bounded by a first patched version, there is no need to consult the registry for a list of releases. A rater is assumed to be capable of interpreting the dependency constraints and have a knowledge of semantic versioning.

Concerning the other three kinds of dependency files, if the specified upstream package is declared as a direct dependency, then solely the release assigned to a node in the dependency graph that represents this direct dependency is used to determine the susceptibility to a vulnerability. Otherwise, if the locked release of at least a single node that corresponds to the specified package in the dependency graph belongs to a range of vulnerable versions, then the file is said to declare a vulnerable dependency. Furthermore, if a repository follows the *monorepo* structure, then the direct dependencies of the hosted sub-modules, defined through the `workspaces` field in the `yarn.lock`, are deemed as direct dependencies of the entire top-level module.

At last, consider that a repository is no longer regarded as affected by a vulnerability through a dependency, if the dependency files that declare a vulnerable dependency are removed. Whereas renaming a dependency file is treated as removing it.

B.1.2 Fixing commit

The goal of the first stage is to, given a case, identify the fixing commit (its *oid*). A fixing commit is the earliest modification to the dependency files that resolves the specified vulnerability in the dependencies and pertains to the default branch of the repository. That is, the first commit that results in a state of the concerned repository such that it is no longer deemed affected by the specified vulnerability. Recall that only the directories reported together with a provided case contribute to susceptibility to a vulnerability.

B.1.3 Candidate commits

For each case, a rater is presented with a list of candidates for the fixing commit. The algorithm used in the study to retrieve this list is based on the recursion and visits the descendants of the commit that served as a parent for the security update generated by Dependabot. That is, a commit on which security update was based. Due to this and the plethora of ways to interact with the version control system, the fixing commit cannot be identified automatically. Specifically, it is possible that none of the candidates is the fixing commit. We regard such a scenario as *complex*, on the contrary to *simple*, where one of the candidates is the fixing commit. Consider Figures B.1b and B.1a.

Both of the figures represent excerpts of the repository history, similar to the visualisation of the network graph realized by GitHub[1]. Each node represents a commit, whereas the directed edges capture the parent-child relationships between them. Red nodes depict commits for which the associated state of the dependency files results in the repository being affected by a vulnerability through dependency. The green is used to define the candidate commits, for which the dependency files do not declare a vulnerable dependency. The yellow highlights the commit generated by Dependabot as a part of a security update. Finally, the blue is used to depict that the associated state is non-vulnerable, but the commit does not belong to the list of candidates.

As can be observed from Figure B.1b, the parent commit of the security update generated by Dependabot is `B1`, while the candidate commits are `B4` and `C2`. In this simple scenario, the fixing commit `C2` is a descendant of `B1`. Whereas examining the complex scenario depicted by Figure B.1a, one can observe that the fixing commit `A3` does not belong to the list of candidates, as the commit `B2`, *i.e.*, the parent of the commit instantiated by Dependabot, is not its ancestor. The reason is that the branch `A`, used as the origin for the fix, was forked before `B2` was created.

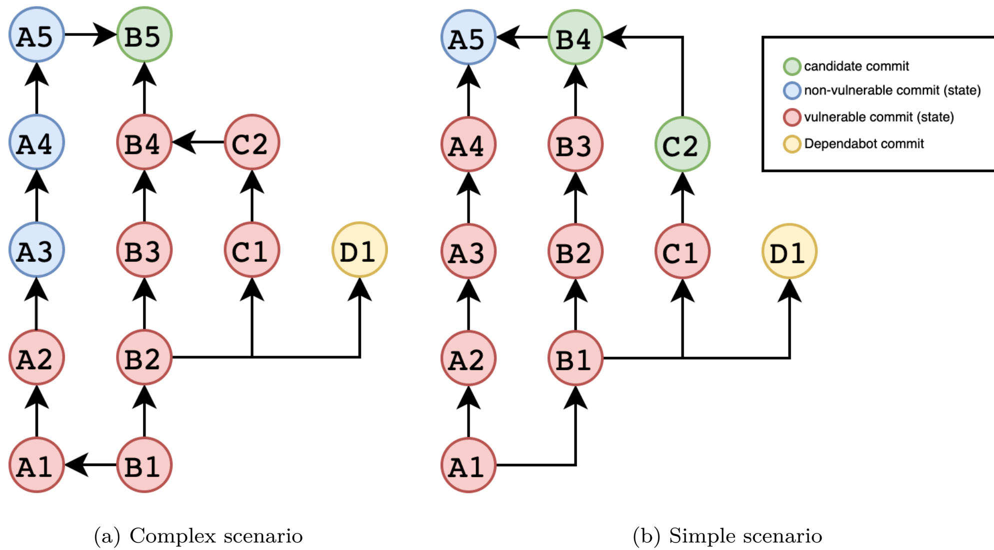
Nevertheless, the traceability of the complex scenario is straightforward. In this case, the earliest candidate commit is a result of a merge. As such, a rater is required to manually investigate the ancestors of this candidate to, ultimately, identify the fixing commit.

Finally, note that merging the corresponding pull request generated by Dependabot does not necessarily result in mitigation of a vulnerability. The reason to this is a supposed bug that arises when the bot attempts to resolve a vulnerability in `yarn.lock` file, while this vulnerability affects multiple disjoint release ranges of the upstream package.

B.1.4 Provided information

Aside from the aforementioned details, as a part of each case, a rater is also presented with the url's to the repository, the first pull request generated by Dependabot to address the vulnerability, and the pull requests that have superseded it (if existing). Additionally, we provide a link to the parent of the commit generated by a Dependabot and the unique identified (`ghsa`) of the concerned vulnerability.

[1] https://docs.github.com/en/github/visualizing-repository-data-with-graphs/viewing-a-repositorys-network



(a) Complex scenario

(b) Simple scenario

Figure B.1: Candidate and fixing commits scenarios

B.2 Second stage

In the second stage, for each case, a rater is provided with the address of the repository, the url's of the first pull request generated by Dependabot to address the vulnerability and the pull requests that have superseded it (if existing), the fixing commit and the pull request associated with it (if existing), the information about the vulnerability and the upstream package, and at last, the list of the directories. A rater is required to assign four labels, one for each column. The in depth description of each is given below.

B.2.1 Column A

The associated variable is a boolean that captures whether or not the fixing commit was created with a security intent. We follow a high-precision approach, and as such, when assigning a value, a rater is prohibited from making conjectures. A fix can only be deemed as a security-intended if and only if this is directly specified by the developers, *e.g.*, through the commit message, the name or the discussion in the associated pull requests, the branch name, *etc.* When in doubt, a rater should always assign FALSE. If a fixing commit is authored by Dependabot, the assigned value is TRUE by default.

B.2.2 Column B

The intent behind this column is to identify by which actions the fix was brought into the repository. Accordingly, a rater is required to assign one of the labels from the predefined pool. Consider Table B.1 that gives the list of possible labels.

B.2.3 Column C

The third column is used to identify the way the dependency (the constraint or the locked version) was altered, such that the vulnerability is eliminated. A rater is required to assign one of the labels listed in Table B.2.

Table B.1: Column B - Labels.

label	**description**
merged	Either the pull request generated by Dependabot to address the vulnerability or one of the pull requests that have superseded it was merged, effectively eliminating the vulnerability.
accompanied	The fix was brought by merging a security update generated by Dependabot to address a vulnerability in another upstream package.
transported	The pull request generated by Dependabot to address the vulnerability and the pull requests that have superseded it are declared as not merged but one of the associated commits is the fixing commit. That is, the developers have migrated the Dependabot commit to another branch, effectively eliminating the vulnerability.
duplicate	The vulnerability alert and the associated Dependabot pull requests were generated as a result of shifting the default branch, while the vulnerability is addressed by merging one of the corresponding security updates instantiated based on the branch regarded as the default previously.
copypaste	The fixing commit is identical to the one generated by Dependabot but authored by a developer. That is, a developer has copied a commit of Dependabot. Alternatively, a portion of the developer commit is a copy of the modifications made by Dependabot.
migrate	The fix is made by a migrating from one dependency file format or registry to another (*e.g.,* from `package-lock.json` to `yarn.lock`).
update dependencies	Either the fixing commit only comprises of modifications to the dependency files (excluding removal or renaming) or the commit/pull request message suggests that (one of) the goal(s) of the commit is to update dependencies.
update upstream	Developers purposely update the direct dependency on the upstream package that induces a transitive dependency on a vulnerable package, effectively removing the latter.
remove upstream	Developers purposely remove the direct dependency on the upstream package that induces a transitive dependency on a vulnerable package, effectively removing the latter.
remove dependency	Developers purposely remove the dependency on the vulnerable package.
remove lock files	Developers removed all of the lock files.
remove one lock file	Developers removed one of the lock files in a directory. The other lock files in the directory remain untouched.
remove manifest	Developers removed the manifest file(s) from the directories sourcing the vulnerability.
remove dependency files	Developers removed all of the dependency files from the directories sourcing the vulnerability.
remove directory	Developers removed the entire sub-module(s) that holds the dependency files sourcing the vulnerability.
audit	Developers either utilized `npm/yarn audit` or `npm/yarn audit fix` commands.
other	None of the categories fit.

Table B.2: Column C - Labels

label	**description**
removed	Either the dependency is removed or the lock files that state the fixed version for it are removed.
introduced as direct	The vulnerable upstream package that was originating only as a transitive dependency, was introduced as a direct one. Accordingly, this has narrowed the susceptibility to a vulnerability to the locked version of the node in the dependency tree that represents the direct dependency on the affected package.
other	None of the two aforementioned categories fit. Similar to the automated message "dependency is up-to-date" of Dependabot.

B.2.4 Column D

The intent behind the final column is to identify the reason why developers decided to not address the vulnerability, implement it manually, or reject the proposition of Dependabot. Similarly to the previous columns, a rater is required to assign one of the labels from the predefined pool. A label can only be assigned based on a strong evidence, such as a discussion in a pull request/issue or a commit message. That is, no conjectures can be made. If no reason can be identified, a rater should leave the field blank. Consider the list of labels given in Table B.3.

Table B.3: Column D - Labels.

grouping	Developer(s) intended to update all dependencies at once, rather than merging the security updates generated by Dependabot.
advanced	Developer(s) indented to perform an advanced modification that eliminates more vulnerabilities than reported by Dependabot.
trial run	The Dependabot security update was generated as a result of trial run by developer(s).
ignore minor	Developer(s) asked Dependabot to ignore the minor versions of the affected package.
ignore dependency	Developer(s) asked Dependabot to ignore the vulnerabilities in the affected package.
no tests	Developer(s) did not merge the Dependabot security update due to lack of testing.
tolerable severity	Developer(s) decided that the severity of the vulnerability is tolerable for the project.
postpone merge	Developer(s) decided to postpone the merge.
confusion	Developer(s) got confused either by the title or the contents of the security update (or a mismatch between the two), rejecting the proposition of Dependabot.
manifest updates only	Developer(s) do not address vulnerabilities in transitive dependencies or prefer to merge modifications that only concern the manifest files.
higher version	Developer(s) want to upgrade to a higher version that proposed by Dependabot.
anti-dependabot	Developer(s) express the reluctance to use Dependabot.
wrong branch	Developer(s) are not satisfied with the choice of the branch to which Dependabot proposes a merge.

spam	Developer(s) regard the security update as spam.
cla	Developer(s) are unable to merge the suggestion since this requires its author to sign the Contributor License Agreement. Accordingly, developer(s) are unaware of how to bypass this.
bot extra changes	Developer(s) are not satisfied with the bot altering some of the additional fields in the dependency files or the magnitude of the change.
already resolved	Developer(s) indicate that the vulnerability has already been addressed on the internal branch.
auto close	The security update is closed automatically by another bot shortly after its creation.
prevention	Developer(s), instead of addressing the vulnerabilities, prefer to prevent the security alerts/notifications/warnings.
obsolete	Developer(s) decided that the dependency is not used in a project or easily removable/replaceable. Alternatively, the (sub-)module in which the vulnerability is identified is not used.
breaking changes	Developer(s) state that merging the security update is not possible due to issues with compatibility.
external management	Developer(s) do not merge the security updates due to external management of the repository (*e.g.,* Gerrit Code Review).
other	none of the categories fit.

Appendix C

Additional Figures

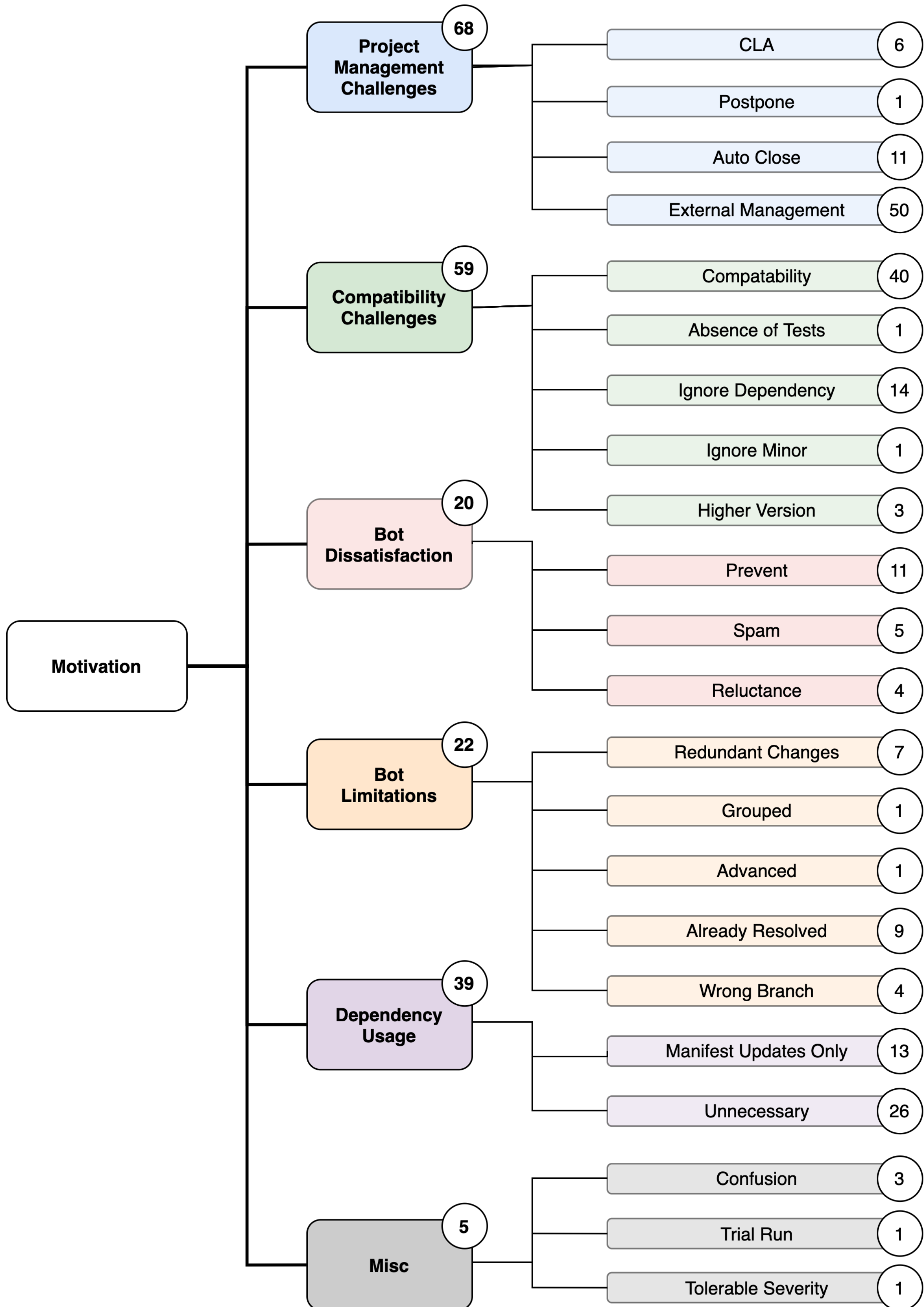


Figure C.1: Summary for $\boldsymbol{RQ_3}$