Investigating the Resolution of Vulnerable Dependencies with Dependabot Security Updates

Hamid Mohayeji*, Andrei Agaronian†, Eleni Constantinou‡, Nicola Zannone*, Alexander Serebrenik*
*Eindhoven University of Technology, Eindhoven, The Netherlands {h.mohayeji.nasrabadi, n.zannone, a.serebrenik}@tue.nl
†Eindhoven University of Technology, Eindhoven, The Netherlands andrei.agaronian@gmail.com
‡University of Cyprus, Nicosia, Cyprus constantinou.a.eleni@ucy.ac.cy

***Abstract*—Modern software development practices increasingly rely on third-party libraries due to the inherent benefits of reuse. However, libraries may contain security vulnerabilities that can propagate to the dependent applications. 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. As manual maintenance of dependencies has shown to be ineffective, several automated tools (aka *bots*) have been proposed to assist developers in rapidly identifying and resolving vulnerable dependencies. In this work, we focus on Dependabot, a popular bot providing security and version updates, and study developers' receptivity to its security updates in engineered and actively maintained JavaScript projects. Moreover, we carry out a fine-grained analysis of the lifecycle of every vulnerability to manifest how they are dealt with in the presence of Dependabot. 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. This approach, however, often takes up to several months which in turn could expose the projects to security issues.**

***Index Terms*—Dependency Management, Security, Bot, Dependabot**

I. 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` and `Maven`). Despite the benefits of reuse [1], there is also the risk of inheriting security vulnerabilities present in the imported libraries [2]. Although vulnerabilities are mitigated in package newer releases, developers are generally reluctant to update stale and vulnerable dependencies [3]. As dependency management is a time-consuming task, nowadays it is facilitated through automation [4]; several software bots have been designed to monitor releases and/or security reports to identify stale and/or vulnerable dependencies, and in response, generate pull requests to update them [5], [6], [7], [8], [9]. On May 2019 [10], GitHub acquired one of the most popular dependency management bots, *Dependabot-preview* [5], resulting in a new natively integrated service, *Dependabot security updates* [11]. 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 free of charge, Dependabot is among the most accessible and widely used services that provide automated pull requests to remedy vulnerable dependencies [12] and the security issues induced by them.

A recent study by Alfadel *et al.* [13] investigated the level of adoption of security pull requests authored by Dependabot-preview, the predecessor of GitHub's service. They found that most of such automated suggestions are merged within a day. Additionally, they examined the factors that affect the rapid merges of security pull requests generated by Dependabot-preview, observing that the severity level of identified vulnerability has no significant impact. However, the findings of Alfadel *et al.* might not necessarily reflect the developer reception and usage of Dependabot security updates provided by GitHub [14]. Indeed, there are significant differences between the two services. First, the main functionality of Dependabot-preview is version updates, aiming to keep dependencies up-to-date. This implies that Dependabot-preview generates pull requests much more frequently, thus, creating more noise. Furthermore, in case of a vulnerable dependency, Dependabot-preview always suggests upgrading to the most recent non-vulnerable version, unlike Dependabot security updates that propose the minimum required version. As a consequence, it is a common scenario that, in case a more recent version is available, Dependabot-preview supersedes the previous security update with a new one. Indeed, Alfadel *et al.* reported that the majority of the rejected pull requests are closed by the bot itself. Besides, Dependabot-preview ships with the *auto-merge* feature, which allows the bot to merge its own pull requests without developer intervention.

In this study, we aim at understanding the impact of Dependabot security updates on handling vulnerabilities. In this regard, we first analyze the merge ratio of Dependabot security updates for multiple disjoint groups of projects with respect to the number of security updates they receive and also investigate the correlation between the merge ratio and the popularity of the projects. To examine how effective Dependabot is to bring security vulnerabilities to the developers' attention, unlike the qualitative approach in Alfadel *et al.* [13], we quantitatively measure the extent to which developers fix vulnerabilities manually in the presence of Dependabot. We base our study on vulnerabilities rather than on security updates, thus performing a more fine-grained analysis compared to previous work, as security updates might contain fixes for multiple vulnerabilities. To this end, we derive every single vulnerability instance from the parent commit of each security update, actualized by leveraging GitHub Advisory Records. We then introduce a mechanism to recursively mine the commit history of each repository to find the possible fixing commit of each vulnerability. This enables us to track and study the lifecycle of each vulnerability instance separately. Next, we conduct a survival analysis of this data to investigate the degree to which developers react to the vulnerabilities in a timely manner and study persistent cases. Finally, we discover the relationship between the severity of vulnerabilities and the time it takes for the developers to react to them, which contradicts the results provided by Alfadel *et al.* [13].

In this work, we analyze 4,195 security updates associated with 978 mature and actively maintained JavaScript projects that are based on `npm` or `yarn` package managers. We discovered that 57% of the bot security updates are merged. Also, bot fixes are almost 2 times more frequent than manual fixes. While the majority of vulnerabilities associated with ignored security updates were fixed manually, our results reveal that manual fixes take considerably more time. This denotes that rejecting security updates leaves the packages vulnerable for longer periods of time, which could lead to security issues [15]. Our study reveals that, overall, Dependabot is entrusted with the task of vulnerability resolution. Still, in some cases, reasons such as suspicion of compatibility issues, limited configurations, and automatic deployment of the bot without prior awareness deter maintainers from entertaining the security updates.

To summarize, this paper makes the following contributions:

- By investigating the correlation between the merge ratio of Dependabot security updates for several groups of projects w.r.t. the number of security updates they receive and their popularity, we provide an insight into the receptivity of the security updates in different projects.
- We introduce a mechanism to derive every single vulnerability instance from the parent commit of each security update, followed by a recursive solution for detecting their fixes. This helps us study the lifecycle of each vulnerability instance individually.
- We conduct a survival analysis on the vulnerabilities to monitor their persistence in case developers decide to ignore the security updates.

The remainder of the paper is organized as follows. The following section presents background on dependencies in JavaScript projects and Dependabot. Section III provides our research questions. Section IV presents the methodology used for data collection and analysis, and Section V reports the results. Section VI discusses our findings and presents the threats to validity. Finally, Section VII discusses related work, and Section VIII concludes the paper.

II. Background

A *dependency* (also known as *package* or *library*) is a piece of code that can be used directly in a program. To ease the installation, upgrading, removal, and distribution of software packages, developers rely on package managers [16]. A plethora of studies [3], [4], [17], [18], [19], [20], [21], [22], [23], [24], [25] showed that dependency update suffers from considerable time lags, sometimes even measured in the orders of years [26]. The reluctance to update dependencies can ultimately lead to security issues [15].

Decan *et al.* [4] examined the propagation of security vulnerabilities in the `npm` dependency network, reporting that more than 20% of the projects directly depend on a vulnerable package, while most of these projects have at least a single release that relies on an affected version of a vulnerable package. This indicates the abundance of vulnerabilities in `npm` packages and the need to mitigate them. Similarly, Prana *et al.* [27] show that the high survivability of a vulnerable dependency is primarily caused by the delayed updates in the dependent application rather than by the persistence of vulnerabilities across the releases of the upstream package. They also observed that the strongest correlation factor for the number of vulnerable dependencies is the total dependency count, which also implies the complexity of the dependency network. This suggests the need for automation to support developers in dependency management.

Mindful dependency management, although highly encouraged, is not always practiced. By surveying developers of software projects with known vulnerable dependencies, Kula *et al.* [3] report that, for developers, the effort needed to mitigate a vulnerable dependency is of greater importance than the persistence of the security issue, while 69% of them were simply unaware of their vulnerable dependencies. This highlights that the community could benefit from automated notifications of security issues in projects' dependencies, driving our interest to study to what extent developers respond to them.

A. Dependencies in JavaScript projects

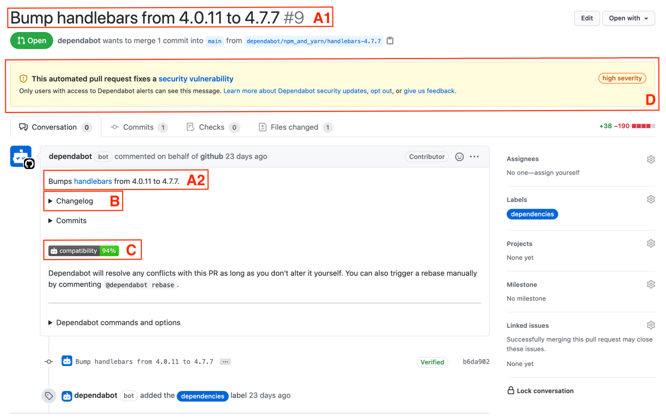
There are two core package managers available for JavaScript software, namely `npm` and `yarn`. In this work, we consider projects that rely on either.

Dependencies are typically declared in a *dependency file* associated with the repository. Specifically, all dependent JavaScript projects contain a *manifest file*, called `package.json`, which specifies the set of *direct* dependencies, *i.e.*, upstream packages referenced within the source code. Aside from a manifest file, developers are 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 *dependency tree*, which specifies both direct dependencies and transitive (*indirect*) dependencies, along with the relevant metadata for each node (*e.g.*, integrity hash and the resource path in the registry).

B. Dependabot

Dependabot is a dependency management tool provided by GitHub, which aims to assist developers in updating dependencies and fixing known vulnerabilities. This tool comprises four interconnected services:

a) GitHub Advisory Database: GitHub maintains a list of security vulnerabilities in software packages belonging to six different ecosystems, including `npm` and `yarn`. Each advisory



Bump handlebars from 4.0.11 to 4.7.7 #9 A1

Open dependabot wants to merge 1 commit into `main` from `dependabot/npm_and_yarn/handlebars-4.7.7`

This automated pull request fixes a security vulnerability

Only users with access to Dependabot alerts can see this message. Learn more about Dependabot security updates, opt out, or give us feedback.

high severity D

Conversation 0 · Commits 1 · Checks 0 · Files changed 1 · +38 −190

dependabot bot commented on behalf of github 23 days ago · Contributor

Bumps handlebars from 4.0.11 to 4.7.7. A2

▸ Changelog B

▸ Commits

compatibility 94% C

Dependabot will resolve any conflicts with this PR as long as you don't alter it yourself. You can also trigger a rebase manually by commenting `@dependabot rebase`.

▸ Dependabot commands and options

`Bump handlebars from 4.0.11 to 4.7.7` Verified b6da902

dependabot bot added the dependencies label 23 days ago

Fig. 1: Screenshot of a security update.

record includes the description of the concerned vulnerability, the name of the package and the ecosystem it belongs to, and affected and patched releases. Additionally, each security advisory is assigned a severity level (*i.e.*, *low*, *moderate*, *high*, and *critical*).

b) Security Alerts: It is a natively integrated service to manage vulnerable dependencies. Upon any change to the dependency files or to the GitHub advisory database, Dependabot scans the dependency graph, and if a vulnerable version of a dependency is identified, raises a security alert.

c) Security Updates: GitHub announced the acquisition of Dependabot-preview in 2019, leading to a new natively integrated service called Dependabot security updates. When enabled, upon receiving a security alert, Dependabot constructs a pull request, *i.e.*, security update, with the modification(s) to the dependency file(s) to update each upstream package to the minimum required non-vulnerable release (if possible). An example of a security update is presented in Figure 1. The suggested change is displayed both in the title and body of the pull request (A1 & A2). Additionally, Dependabot provides the changelog for each version between the current and the suggested release (B) and a badge indicating the compatibility score (C). The latter is calculated dynamically based on the percentage of successful Continuous Integration (CI) runs in other public repositories where an identical security update has been instantiated. The information about the concerned vulnerability is not presented in the pull request itself. Instead, a yellow paned window is displayed to users, signaling 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 users without the necessary access rights, *i.e.*, external observers.

d) Version updates: They are automatically generated pull requests that aim at keeping dependencies up-to-date. Both security and version updates can be enabled in a single repository simultaneously. However, to an external observer, the two kinds of pull requests are indistinguishable.

III. Research Questions

Dependabot, as one of the most accessible tools providing automated security updates to fix vulnerabilities that would otherwise cause security issues, is of great importance to investigate. However, the extent of receptivity towards its security updates in different types of projects and the way vulnerabilities are dealt with in its presence is still unclear. To address this gap, we focus on three research questions, described below.

Our first question aims to understand the degree to which developers take in Dependabot security updates w.r.t. the number of security updates they receive and their popularity:
RQ$_1$: *How often do developers merge Dependabot security updates?*

As suggested by previous studies [13], [28], developers may prefer to close a security update 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 a Dependabot security update?*

Finally, to understand the efficacy of Dependabot in bringing the vulnerabilities to the developer's attention, we attempt to capture the degree to which developers react to the identified vulnerabilities in a timely manner. These concerns are captured by the following research question:
RQ$_3$: *How long does it take to address a vulnerable dependency identified by Dependabot?*

IV. Methodology

This section presents the methodology employed for data collection and the analysis techniques we leverage to answer the research questions posed in this work.

A. Data Collection

To study how often developers merge security updates of Dependabot (***RQ$_1$***), we collect security updates from GitHub. Next, we extract vulnerability instances from each security update, and finally mine the projects commit history to investigate the fixes of vulnerable dependencies (***RQ$_2$***, ***RQ$_3$***).

For our empirical study, we focus on JavaScript projects for two reasons. First, the GitHub annual survey[1] suggests that JavaScript is taking the lead as the most popular programming language. Moreover, JavaScript projects have the highest distribution of package dependencies compared to other programming languages [29], thus making them more prone to inheriting vulnerabilities through dependencies [23], [30].

As we do not have access to the projects' settings, it is not possible to determine whether a project employs both version update and security update services. The presence of both services in a single project may contribute to confounding

[1]https://octoverse.github.com/2022/top-programming-languages. Last accessed January 18, 2023.

TABLE I: 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 Munaiah* et al. *[32]*

factors in the resolution of security updates, as their pull requests are indistinguishable. To this end, we constrain the collection of security updates to the period between the introduction of security updates (June 1, 2019) and the introduction of version updates (May 31, 2020) to ensure that only security updates are available.

Using the *GitHub Search API*[2], we identified 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. We selected the projects that were actively maintained during the entire collection period and have no less than 100 commits at the start of the collection period [3] and at least a single commit at each month of the collection period. As we are interested in projects employing the `npm` and `yarn` package management tools, we only considered projects including a `package.json` manifest file in the root of the repository, leaving us with 3,587 projects.

Kalliamvakou *et al.* [31] show that a large portion of GitHub repositories are used for experimental, storage, or academic purposes. As the inclusion of such repositories can introduce noise into the analysis, we filtered those out using `Reaper` [32], resulting in 3,151 engineered projects. From the set of 1,492 repositories that have at least a single security update issued by Dependabot (out of the 3,151 engineered projects), we also filter out 390 projects that use multiple dependency management bots (*e.g.*, Dependabot-preview [5], Greenkeeper [6], Snyk-bot [7], and Renovate [8]) to account for confounding factors stemming from the use of various tools, and 124 projects that have received Dependabot security updates targeting ecosystems other than `npm` and `yarn` (*e.g.*, `Maven`, `RubyGems`), resulting in 978 projects. An overview of these projects is presented in Table I. For each selected project, we extract all security updates created by Dependabot within the collection period. At this point, our dataset consists of 4,416 security updates, of which 2,391 are in the *merged* state, 1,804 in the *closed* state, and 221 are *open*. Since a decision to or not to merge an open security update has not been made by the developers, we use the 4,195 non-open ones in our analysis. A non-open security update is a security update whose state is "closed" or "merged".

B. Detecting Vulnerabilities in Dependencies

To discover vulnerabilities, we retrieved the security advisories using the *GitHub GraphQL API* on March 27, 2021. Removing vulnerabilities with no known fixes leaves us with 1,063 security advisory records. Given that GitHub advisory database is constantly evolving, for each retrieved security update, we identified whether there is at least a single associated security advisory from the collected set. Since the information about the vulnerabilities targeted by a security update cannot be accessed without specific project permissions, to identify the association of an advisory with a security update, we extract this information based on the title of security updates, which contains the vulnerable upstream package, its currently installed version, and the release to which Dependabot suggests upgrading (cf. Figure 1). We found that 28 (0.6%) records in our dataset could not be matched to any security advisory record and removed them from further analysis.

Our vulnerability detection algorithm takes as input (1) the name of the concerned upstream package with known security vulnerabilities, (2) the database of 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 are different for the manifest and the lock files. 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. Note that this definition extends the one adopted in [4] by also considering development dependencies. To this end, 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 by leveraging 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 [33], *i.e.*, it contains more than one project. In this case, 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. 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. 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, we extract the version of the upstream package locked for direct dependency. In the second scenario, the specified upstream package with a known security vulnerability is not declared as a direct runtime or development dependency in the manifest file. Then, conversely to the previous case, every node in the dependency graph corresponding to a dependency on the concerned upstream package is inspected. If the locked release

[2]https://docs.github.com/en/rest/search. Last accessed January 18, 2023.

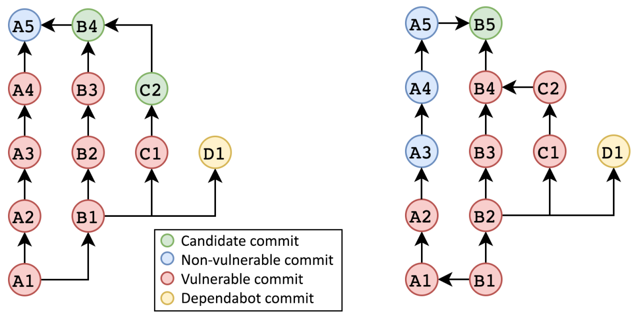
of at least a single node in the dependency graph belongs to the range of the affected versions, then the file is said to declare a dependency on a vulnerable release of an upstream package. For this purpose, we recursively traverse every dependency node object in the graph, examining those defined by the name of the concerned upstream package and collecting the releases locked to them.

The vulnerability detection algorithm is validated through binary classification. For this purpose, we leverage the Dependabot security updates. The *parent commit* of a security update, *i.e.*, the commit on which the modification is based, represents the state of the project with a vulnerable dependency, as each security update addresses at least a single vulnerability in the dependencies. On the other hand, the *merge commit*, *i.e.*, the commit obtained by merging the security update, captures the state at which this vulnerable dependency is resolved. For each repository, we feed the algorithm the dependency file associated with each of these two commits. For the parent commit, it is expected to return at least a single security advisory published before the corresponding security update was generated. On the other hand, no security advisory should be returned for merge commits.

The algorithm 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 a careful examination of these cases, we found out they were due to a bug in Dependabot rather than to an issue in our algorithm. 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 [34]. To verify the bug conjecture, we replicated the aforementioned conditions in a GitHub repository with both the Dependabot security alerts and security updates enabled. We find that after merging the automated security update generated by Dependabot, the associated security alert persists.

C. Fixing Cases Discovery

To trace the cases where a vulnerable dependency was fixed, we applied the vulnerability detection algorithm presented in the previous section to the parent commits of each security update and generated a separate entry for every reported security advisory. Accordingly, each entry is identified by the following four properties: (1) the slug, *i.e.*, the name of the repository owner, (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 us to capture the earliest state in the repositories history at which Dependabot has identified the vulnerability.



(a) Simple scenario (b) Complex scenario

Fig. 2: Candidate and fixing commits scenarios.

In total, we identified 5,089 vulnerabilities. However, Dependabot creates a new security update in case a more recent version of the upstream is required. Therefore, some of the collected events do not capture the earliest stage 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 4,978 entries.

Finally, to collect the fixing cases, for each identified 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. To discover the fixing commit, we designed an algorithm that recursively visits the descendants of each security update's parent commit and identifies whether the concerned vulnerability is eliminated or not. Since the repository history commonly comprises more than a single branch, the algorithm may return multiple *candidate fixing commits*. 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 resolved vulnerable dependency 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.

Figs. 2a and 2b present excerpts of the repository history. Each node represents a commit, and the directed edges capture the parent-child relationships between them. In Fig. 2a, 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 in the complex scenario shown in Fig. 2b, 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.

The existence of the complex scenario explains the reason

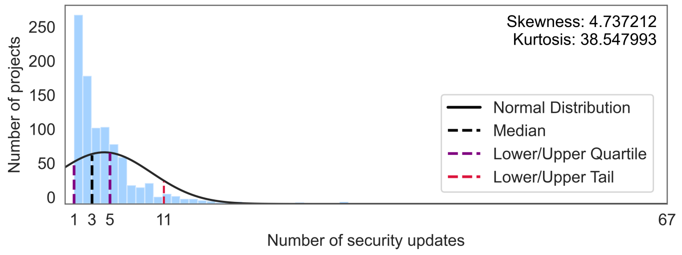


Fig. 3: Distribution of the number of security updates.

for no complete automation for the discovery of the fixing commit. Avoiding them requires traversing the network graph backward, which is expected to increase the number of nodes to be visited by orders of magnitude. To this end, we opted for a semi-automated solution where, given the list of the candidate commits, a human rater determined 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 recruited another independent rater. The second rater was presented with a total of 50 events identifying a security vulnerability with a list of the candidate fixing commits for each event. Half of the events pertain to a complex scenario, while the other does not, which was not revealed to the rater. We found no discrepancy between the fixing commits reported by the original and the second rater, increasing our confidence in the accuracy of collected fixing cases.

D. Data Analysis

This section presents the techniques employed to address our research questions.

*1) **Addressing RQ$_1$**:* To identify how frequently developers merge the security updates created by Dependabot, we computed the *merge ratio* defined as the proportion of the non-open security updates that were merged. However, different projects may adhere to different practices and, thus, respond to security updates in a non-uniform manner. To this end, we also performed a more fine-grained analysis by assessing this metric on a per-project basis.

If a project received a very small number of security updates (*e.g.,* 1 or 2), it is extremely likely that the merge ratio is either 0% or 100%. If this is a predominant case, the distribution of merge ratios will be mostly determined by these projects, heavily skewing it to both extremes. Fig. 3 plots the distribution of the number of security updates for the project in our collection along with the coefficients of *skewness* and *kurtosis* [35], whose acceptable values for normality lie between -1 and 1 and between -2 and 2, respectively. We can observe that the distribution suffers from a very high positive skew. To this end, we followed a quantile classification approach [36] to define four disjoint groups of projects with respect to the number of security updates they receive, namely *Very low*, *Low*, *High*, and *Very high*. Table II reports the cut-off points based on which the classification is performed.

TABLE II: 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]

Given that popular projects are more likely to adopt test automation practices [37], they might have fewer concerns about possible breaking changes caused by merging a pull request. In this regard, we compute Spearman's rank correlation test to inspect whether the merge ratio of pull requests and the popularity of the projects (represented as the number of stars and forks) are correlated.

*2) **Addressing RQ$_2$**:* To assess how often developers fix a vulnerability in dependencies manually despite the presence of a Dependabot security update, we measured 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 latter group through the absence of the fixing commit, whereas distinguishing between the first and the second groups is more complex. We regard a vulnerable dependency as fixed by Dependabot if and only if the fixing commit is authored by the bot. Otherwise, we attribute the fix to the developers.

To maintain consistency with **RQ$_1$**, we also computed the percentages for the four groups of projects separately: (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 Pearson's χ^2 test [38]. 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 best practices [39], we also report the effect size, *i.e.*, the magnitude of this relationship, when Pearson's χ^2 test suggests the rejection of the null hypothesis. Despite the plethora of approaches for computing the effect size [39], for a contingency table whose size is not upper-bounded by 2×2, it is recommended in [40], [41] to use Cramér's V [42], denoted by ϕ_V. This metric varies between 0 and 1, with the former corresponding to a lack of association. We follow the interpretation of ϕ_V proposed by Cohen [43], which suggests that for a 4×2 contingency table, the association between two variables is *trivial* if $\phi_V < 0.10$, *small* if $0.10 \le \phi_V < 0.30$, *medium* if $0.30 \le \phi_V < 0.50$, and *large* if $\phi_V \ge 0.50$. 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 [44].

*3) **Addressing RQ$_3$**:* By answering the third research question, we aim to capture the degree to which developers *react* to the vulnerabilities identified by Dependabot in a timely manner. We operationalize the time required to resolve a vulnerable dependency as the difference between the time the vulnerability was reported by Dependabot through a security update and the time the fixing commit was made.

In line with previous studies [4], [23], [45], [46], [47], we rely on survival analysis [48] to assess the time-to-event distribution. Using the Kaplan-Meier estimator [49], a non-parametric statistic, we fit a survival analysis model to estimate the survival rate of vulnerabilities in dependencies, *i.e.*, the expected time until an actionable reaction to a vulnerability, over time. Accordingly, the survival function represents the probability that a vulnerability survives after a certain time point.

We also performed the analysis with respect to the severity levels of vulnerabilities. This allows verifying whether there is a relationship between the risks due to a vulnerability and the time it takes for the developers to react to it, that is, whether the developers take into account the severity of vulnerabilities in prioritization. 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 [50], 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 control the family-wise error rate, *i.e.*, probability of making at least one type I error, and following the Bonferonni approach [51], test each individual hypothesis at a significance level of 0.83% ($= \alpha/T$ where $\alpha = 5\%$ is the desired overall significance level and $T = 6$ is the number of comparisons).

Finally, we measured and compared 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 to 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 [52] at a standard 5% significance level. The choice of the one-sided alternative is motivated by the intuition that vulnerability fixes attributed to Dependabot take less time than the manually implemented ones.

TABLE III: Distribution of merge ratios for the four project groups.

Group	Min	25%	Median	75%	Max	Avg	Std
Very low	0%	0%	50%	100%	100%	49%	47%
Low	0%	0%	75%	100%	100%	59%	43%
High	0%	0%	80%	100%	100%	57%	42%
Very high	0%	26%	78%	94%	100%	61%	38%
Total	0%	0%	67%	100%	100%	54%	44%

V. Results

A. RQ1: How often do developers merge Dependabot security updates?

To answer **RQ1**, we computed the merge ratio for non-open Dependabot security updates (cf. Section IV-D1). The results show that, of the 4,195 non-open Dependabot security updates in our dataset, 57% were merged.

Table III shows the distribution of merge ratios for the projects considered in our study. We can observe that projects merged, on average, 54% of their non-open security updates (7th column in Table III) and that the median project merged 67% of its non-open security updates (4th column in Table III). Moreover, the mean merge ratio for the projects in all categories except *Very Low*, is 75% or higher. However, we cannot make strong claims on the trend in specific project groups since the average values are not very dispersed (ranging between 49% and 61%) and the standard deviation is quite large (ranging between 38% and 47%).

To verify if there are differences between the distributions of the merge ratios between project groups, we performed multiple Mann-Whitney U tests to validate the null hypothesis that "the samples in category X come from the same population as the ones of category Y", where X and Y are populated with the four project groups (*Very low*, *Low*, *High*, *Very high*). Note that due to multiple comparisons over the same data, we used Bonferroni's correction [53] and report over the corrected p-values. The results show that the null hypotheses (one for each distinct pair of project groups) cannot be rejected (adjusted $p \geq 0.15$), meaning there are no significant differences between repositories of the different project groups w.r.t. their willingness to merge Dependabot security updates. Also, we compute Spearman's rank correlation test for merge ratio and popularity metrics. The correlation with stars and with forks is weak (both ρ = 0.16), suggesting an insignificant correlation between the merge ratio and project popularity.

B. RQ2: How frequently do developers fix a vulnerable dependency manually in the presence of a Dependabot security update?

To answer this question, we measure the percentage of vulnerabilities in the obtained collection that were (1) fixed by Dependabot, (2) fixed by a developer, or (3) have not yet been addressed.

The 4,195 security updates correspond to 4,978 security vulnerabilities. Our results show that 53.48% (2,662 out of 4,978) of the examined vulnerabilities are mitigated by

TABLE IV: Percentages of vulnerabilities addressed by human/bot and non-resolved per project group.

Group	Response to vul.		Fixed by	
	Fixed	Not fixed	Bot	Human
Very low	76.35%	**23.65%**	52.83%	**47.17%**
Low	**90.77%**	9.23%	61.36%	38.64%
High	86.04%	13.96%	64.11%	35.89%
Very high	84.07%	15.93%	**71.11%**	28.89%
Total	84.62%	15.38%	63.85%	36.15%

merging Dependabot security updates, 30.27% (1,507 out of 4,978) of the vulnerabilities are resolved manually, and the remaining 16.25% (809 out of 4,978) have not been fixed. Taking into account the percentage of unresolved vulnerabilities, bot fixes are 1.8 times more frequent than manual fixes. Besides, there is a possibility that manual fixes are also inspired by Dependabot, albeit we could not measure those cases. This implies that, overall, the task of vulnerability resolution is to a great extent delegated to Dependabot.

Table IV reports the percentage of vulnerabilities that have and have not been fixed (meta-column *Response to vulnerabilities*) per project group. We can observe a noticeable difference between these percentages. Pearson's χ^2 test confirms that this difference is significant ($p = 1.19e$-15) with a non-trivial (small) effect size $\phi_V = 0.12$. The lowest percentage of resolved vulnerabilities belongs to the group of projects with a *very low* number of security updates received. On the other hand, 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. Moreover, as the number of security updates increases, the proportion of unaddressed security vulnerabilities raises as well. A possible explanation is that developers get overwhelmed by notifications about vulnerable dependencies. To support this conjecture, we compared the percentages of the security fixes made by the bot and humans (meta-column *Fixed by* in Table IV). We observe that the delegation of the security fixes to Dependabot increases with the increase of security updates ($p = 9.89e-14$ 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 highlighted in [13]. The greater extent of delegation of vulnerability resolution to the bot can be 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 V, 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 case is the trivial difference in the vulnerability response between the *high* and *very high* groups, despite a significant discrepancy in the delegation of the security fixes. This suggests that the majority of projects that belong to the second group compensate for the increased complexity (or the number of vulnerabilities) by delegating the (added) security workload to the 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, 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 a lack of experience or trust in the bot.

TABLE V: 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		6.41e-15***	3.99e-09***	1.57e-05***
Low	1.70e-03**		7.66e-04***	7.43e-06***
High	3.95e-06***	2.08e-01		1.04e-01
Very high	4.32e-14***	1.24e-05***	1.85e-04***	

fixed by bot vs. fixed by human

fixed vs not fixed

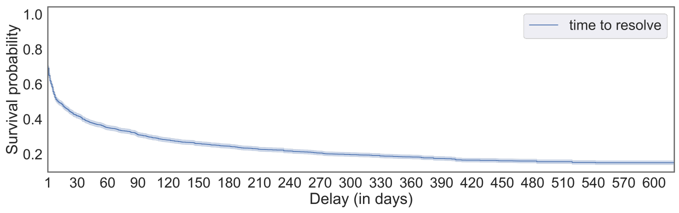


Fig. 4: Survival curve for the event "vulnerability identified by Dependabot is addressed".

C. RQ3: How long does it take to address a vulnerable dependency identified by Dependabot?

Figure 4 shows the survival curve of the vulnerable dependencies identified by Dependabot. First, we observe that the likelihood of a vulnerability remaining unaddressed within the first day is less than 70%. In other words, it is expected that almost a third of the vulnerable dependencies are addressed within 24 hours since the reception of the security update. Moreover, the results show that the majority of the vulnerable dependencies are addressed within the first two weeks. As such, we conclude that developers predominantly respond to the suggestion of Dependabot promptly. Nevertheless, a vulnerable dependency reported by the bot remains unaddressed for over a year with a probability of 18%, implying that almost one over five vulnerabilities affect the users of the dependent projects for at least an entire year since the advisory was published.

The analysis of the relation between the survivability and severity of a vulnerability reveals that the level of severity is negatively correlated with the survival probability when comparing critical and high severity vulnerabilities to moderate and low, contrary to the study conducted by Alfadel *et al.* [13]. In fact, there is a clear difference between the first two classes of vulnerabilities - given an identical time span, a

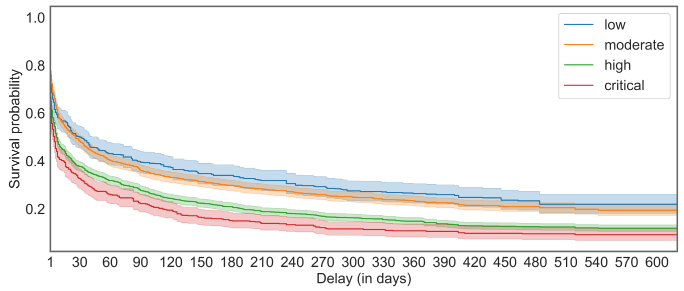


Fig. 5: Survival curves for the event "vulnerability identified by Dependabot is addressed" based on severity level.

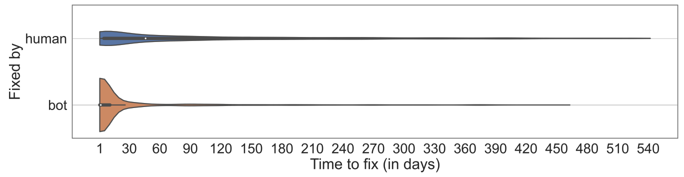


Fig. 6: Violin plots for the distributions of the bot and manual fixing times.

critical severity vulnerability is always less likely to remain unaddressed. The dominance is even more emphasized when compared to 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. When it comes to an entire year since the notification, a low severity vulnerability is 2.3 times more likely to persist than a critical one and 1.7 times more than a high one. 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 severity of vulnerabilities when prioritizing a dependency update. As expected, comparing the bot fixes and the ones implemented manually, we find that the first takes significantly less time than the latter (0.01% significance level). As can be observed from the distributions of the fixing times in Fig. 6, 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 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.

VI. Discussion

Comparing the merge ratio of security pull requests generated by Dependabot to Dependabot-preview (65.42%), as reported in [13], we observe a drop of 9% in receptivity. We conjecture this difference is mainly due to the auto-merge functionality of Dependabot-preview. Also, the Dependabot being activated automatically by GitHub, given that the repository meets certain prerequisites, might explain the hesitance to merge pull requests since developers did not enable the bot themselves.

To investigate possible trends throughout our observation period, such as the changes in the merge ratio of security updates or the resolution time of vulnerabilities in different spells, we split our dataset into three parts (P1, P2, P3), each corresponding to a period of 4 months. Concerning the merged pull requests of **$\mathbf{RQ_1}$**, our analysis in the three time periods showed that projects across all four categories have a larger merge ratio in P1. However, since most projects receive a very small number of security updates, we cannot infer anything about within-project adoption across the observation period. If we repeat the analysis by considering only projects that received security updates throughout the entire observation period, we find that the average merge ratio per category is quite similar in all periods. Also, we ran the **$\mathbf{RQ_3}$** survival analysis on the vulnerabilities created in each period. The results show a trend throughout the periods: vulnerabilities in the earliest period (P1) tend to be resolved faster compared to the following two periods (P2, P3). For instance, after 60 days, about 75% of vulnerabilities were resolved in P1, while 50% and 60% were resolved in P2 and P3, respectively. Hence, vulnerabilities were fixed faster at the beginning of our observation period.

In an effort to identify the reasons developers decide not to address a vulnerability, implement it manually, or solely reject the proposition of Dependabot, we recruit two different raters to examine related textual artifacts, including 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 communication texts. We identified 22 unique reasons, *i.e.*, labels, out of 213 samples analyzed by the first rater, which can be clustered into six groups. Also, the second rater performs a separate round of labeling independently to counteract the subjectivity. To assess the agreement between the two raters, we compute Cohen's κ, the de facto standard statistic, and observe $\kappa = 0.963$. This, following the interpretation proposed by Viera and Garrett [54], is equivalent to *perfect agreement*.

We find that in 31.92% (68) of the cases the decision to not merge a Dependabot security update and address the vulnerability manually stems from the ***project management peculiarities***, such as *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. In line with previous studies [55], [56], [57], we find that in 27.70% (59) of cases, ***compatibility challenges*** are one of the biggest developer concerns. ***dependency usage***, *i.e.*, unused dependencies accounts for the 18.31% (39) of the reasoned cases. As expected, some of the ***bot limitations*** may also impact the decision to not accept its contributions - 10.33% (22), such as the limited configuration settings provided by Dependabot, which is one of the recurrent issues in bot adoption, following the study of Wessel *et al.* [58]. We also find the developers expressing ***bot dissatisfaction*** in 9.39% (20) of the cases, especially complaining about the automatic deployment of the bot and noise generation, which is the most recurrent and central problem of interacting with software bots [59], [60], [58]. Finally, there are 2.35% (5) ***miscellaneous*** cases.

A. Implications to Practitioners

The observed distribution of merge rates 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. Also, our qualitative analysis shows roughly 9% of dissatisfaction in explicitly motivated rejections. Moreover, the survival analysis conducted in this work proves that manual resolution of vulnerabilities might take a long time, leaving the projects susceptible to security issues. Despite popular projects having more tests available [37], our results signify that there is no significant correlation between projects' popularity and the level of security update adoption. This result highlights that developers should be vigilant on security even if their project is popular and possibly has a larger community. Taking everything into account, we encourage developers to try out Dependabot on a trial basis as evidence suggests its benefits in managing security vulnerabilities are considerable.

B. Implications to Dependabot Maintainers

As suggested by our qualitative analysis, some maintainers can get confused and refuse to merge a pull request of an unknown bot that starts interacting with the repository without warning. To address this, we recommend GitHub always deploy Dependabot with an introductory message that explains the purpose of the bot.

Limited configuration is a common issue in bot adoption [58], and Dependabot is no exception. One setting that could 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. Alternatively, Dependabot could allow project maintainers to prioritize the reception of security updates based on the vulnerability severity level. This could be interesting for maintainers, as we observe a strong correlation between the survivability of a vulnerability and the severity level assigned to it. Moreover, 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 it reduces the number of false alarms.

C. Implications to Researchers

We observe that developers merge a security update generated by Dependabot in 57% of the cases. This statistic does not align with the results reported in previous works such as the one of Wyrich *et al.* [61], who report that only 37.38% of the bot pull requests in their collection ended up being merged. The difference in the results could be due to the extensive project filtering performed in our 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 issuing pull requests and analyze them as a whole. This is further confirmed when comparing our results to the ones reported by Mirhosseini *et al.* [62] 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, their different goals play a role in their receptivity. As such, when analyzing the bot usage among developers, we suggest separating them based on the goals and tasks these bots are designed to fulfill.

Other dependency management bots, such as Snyk [7] and Renovate [8], have been proposed to assist developers in managing vulnerable dependencies. While these bots also have limitations, some of their characteristics and features are complementary to Dependabot. For instance, Dependabot security updates have a compatibility score that lets developers know whether updating a dependency could cause breaking changes. This might lead to a higher merge ratio for its security updates. Snyk, on the other hand, provides a priority score feature that can assist developers in filtering and prioritizing the discovered issues according to their level of importance, risk, frequency, and ease of fixing. Using multiple bots in a repository for similar purposes, however, might lead to more noise. As we discuss in our paper, noise is one of the causes of vulnerabilities not being addressed. Using complementary tools might increase the noise and ultimately delay the resolution of vulnerabilities. Nonetheless, further study is needed to discover the efficacy of employing multiple bots in software repositories.

D. Threats to Validity

Construct validity. A threat to construct validity concerns 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. Although developers can replicate the update generated by the bot or use a proxy to implement the suggested changes, such cases cannot be identified with high confidence.

Internal validity. The main internal threat pertains to the quality of the obtained collection of projects. The presence of abandoned projects or immutable forks could have downgraded the overall merge ratio or significantly affected survival curves for vulnerabilities. To address this threat, we applied an extensive filtering procedure and removed projects having less than one commit each month of the collection period. Also, we only considered engineered projects selected using the `Reaper` tool. Despite that this tool may not be 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.

External validity. This work only focuses on JavaScript projects and the `npm` and `yarn` ecosystems. Therefore, our findings might not apply to other types of projects and ecosystems due to the different policies, practices, and culture in each ecosystem [23], [56]. This is supported by the findings of Zerouali et al. [63], who found that vulnerabilities in `npm` are fixed sooner compared to `RubyGems`, which is a much small ecosystem compared to `npm`. Moreover, our analysis is strictly limited to Dependabot, whose interaction traits and core logic can be different from the ones of other bots. Further studies are needed to verify our findings for other ecosystems and bots.

VII. Related work

Previous studies on bots, as one of the most dominant assistant tools in software development, focus on identifying the challenges in interaction [58], [59], [60], [64], the impact of their usage on the development artifacts and software quality [62], [65], [66], and quantitatively measuring the extent of their adoption and developer receptivity towards their assistance [13], [61], [67], [68].

Wyrich *et al.* [61] found that 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. However, this analysis is not scoped to a particular bot nor to the properties of the projects that use them. On the contrary, our work focuses on GitHub Dependabot and mature and well-maintained JavaScript projects, aiming to mitigate the potential impact of confounding variables.

Mirhosseini *et al.* [62] investigated the impact of Greenkeeper, a bot that generates pull requests to upgrade out-of-date dependencies of JavaScript projects. Their results show that, on average, projects that employ the bot upgrade their dependencies 1.6 times more often than projects that do not use the bot, suggesting a significant utility of such a tool. 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 survey of Pashchenko *et al.* [28] suggests that bots can be only used to identify issues within dependencies, while the update itself is performed by developers. Also, He *et al.* [69] reports that Dependabot is effective in reducing technical lag, and developers are highly receptive to its pull requests. In our work, we take a step forward and not only analyze the developers' receptivity to bot suggestions but also determine quantitatively how frequently they resolve the problem manually despite an automated pull request and study the persistence of vulnerabilities.

Alfadel *et al.* [13] is the most relevant to our work. The authors investigated developers' receptivity to the security pull requests generated by Dependabot-preview. The results show that the majority of such pull requests are merged, often within a day, and the severity level of the vulnerabilities has no significant impact on their resolution time. Moreover, their results suggest that the risk of breaking changes has no significant influence on the merging of pull requests, contrary to other studies [4], [55], [56], [70], [71]. The main functionality of Dependabot-preview, studied in [13], is version updates. The bot supersedes a security update with a new one when a more recent release of the vulnerable package is available. Most importantly, Dependabot-preview provides an *auto-merge*, which means the findings do not necessarily reflect developers' reception of security pull requests since many merges and rejections are performed automatically. Examining Dependabot, on the other hand, allows us to account for these confounding factors as Dependabot always suggests the minimum required non-vulnerable version. Moreover, unlike all related works, we focus our analysis on the vulnerabilities rather than on the pull requests to more accurately estimate the time developers take to resolve vulnerabilities.

VIII. Conclusion

In this work, we conducted an empirical study on the usage of Dependabot and its effectiveness on maintaining the dependencies secure in JavaScript projects hosted on GitHub. In particular, we analyzed 4,195 security updates associated with 978 engineered and actively maintained JavaScript projects. We studied developers' receptivity towards Dependabot security updates, the practice of fixing the vulnerability manually in the presence of an automated pull request, and assessed how proactive developers are in solving the vulnerable dependencies alerted by Dependabot.

The results show that more than half of the bot suggestions are merged. Moreover, when developers do not accept a security update, they often address the associated vulnerability manually. By levering a survival analysis, we found that developers are often proactive in addressing vulnerable dependencies identified by Dependabot and prioritize fixes based on the severity level of vulnerabilities. However, when developers implement the fixes manually, the fixes could take more than a month.

We call for the continuation of our work by further analyzing the impact of Dependabot on keeping the project dependencies secure. In the case of manual fixes, for example, it remains unclear to what extent those fixes were inspired by Dependabot security updates. Also, for studying the relation between the Dependabot merge ratio and concern of breaking changes, other than popularity metrics, one could investigate factors such as the availability of tests or CI/CD pipelines. Another possible research avenue is to inspect the actual usage of vulnerable APIs in the repository's code, as a security update or relying on a vulnerable dependency might not necessarily imply that the project is vulnerable. Examining these cases can assist Dependabot in creating fewer security updates and reducing the noise level.

IX. Data Availability

To comply with Open Science policies, all the scripts and data used to produce the results of this work are publicly accessible [3].

References

[1] W. B. Frakes and K. Kang, "Software reuse research: Status and future," *IEEE transactions on Software Engineering*, vol. 31, no. 7, pp. 529–536, 2005.

[2] H. H. Thompson, "Why security testing is hard," *IEEE Security & Privacy*, vol. 1, no. 4, pp. 83–86, 2003.

[3]https://github.com/piwvh/dependabot-msr2023

[3] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[4] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 181–191.

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

[6] "Greenkeeper," https://greenkeeper.io/, [Online] Last accessed 18 January 2023.

[7] "Snyk," https://github.com/marketplace/snyk/, [Online] Last accessed 18 January 2023.

[8] "Renovate," https://github.com/marketplace/renovate/, [Online] Last accessed 18 January 2023.

[9] "Pyup," https://pyup.io/, [Online] Last accessed 18 January 2023.

[10] "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 18 January 2023.

[11] "About Dependabot security updates," https://docs.github.com/en/code-security/dependabot/dependabot-security-updates/about-dependabot-security-updates, [Online] Last accessed 18 January 2023.

[12] M. Wyrich, R. Ghit, T. Haller, and C. Müller, "Bots don't mind waiting, do they? comparing the interaction with automatically and manually created pull requests," in *2021 IEEE/ACM Third International Workshop on Bots in Software Engineering (BotSE)*, 2021, pp. 6–10.

[13] M. Alfadel, D. E. Costa, E. Shihab, and M. Mkhallalati, "On the use of dependabot security pull requests," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 254–265.

[14] "Goodbye dependabot preview, hello dependabot!" https://github.blog/2021-04-29-goodbye-dependabot-preview-hello-dependabot/, [Online] Last accessed 18 January 2023.

[15] J. Cox, E. Bouwers, M. Van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 109–118.

[16] "What is package manager?" https://www.debian.org/doc/manuals/aptitude/pr01s02.en.html, [Online] Last accessed 22 August 2022.

[17] M. Backes, S. Bugiel, and E. 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*, 2016, pp. 356–367.

[18] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, "An empirical analysis of technical lag in npm package dependencies," in *International Conference on Software Reuse*. Springer, 2018, pp. 95–110.

[19] A. Decan, T. Mens, and E. Constantinou, "On the evolution of technical lag in the npm package dependency network," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 404–414.

[20] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. 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*, 2017, pp. 2187–2200.

[21] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. 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)*. IEEE, 2020, pp. 35–45.

[22] B. Chinthanet, R. G. Kula, S. McIntosh, T. Ishio, A. Ihara, and K. Matsumoto, "Lags in the release, adoption, and propagation of npm vulnerability fixes," *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–28, 2021.

[23] A. Decan, T. Mens, and M. Claes, "An empirical comparison of dependency issues in oss packaging ecosystems," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 2–12.

[24] A. Zerouali, V. Cosentino, T. Mens, G. Robles, and J. 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)*. IEEE, 2019, pp. 619–623.

[25] M. Alfadel, D. E. Costa, M. Mokhallalati, E. Shihab, and B. Adams, "On the Threat of npm Vulnerable Dependencies in Node.js Applications," *arXiv preprint arXiv:2009.09019*, 2020.

[26] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," *arXiv preprint arXiv:1811.00918*, 2018.

[27] G. A. A. Prana, A. Sharma, L. K. Shar, D. Foo, A. E. Santosa, A. Sharma, and D. Lo, "Out of sight, out of mind? how vulnerable dependencies affect open-source projects," *Empirical Software Engineering*, vol. 26, no. 4, pp. 1–34, 2021.

[28] I. Pashchenko, D.-L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1513–1531.

[29] "The 2020 State of the Octoverse: Security Report," https://octoverse.github.com/static/github-octoverse-2020-security-report.pdf, [Online] Last accessed 18 January 2023.

[30] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019.

[31] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 92–101.

[32] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[33] G. Brito, R. Terra, and M. T. Valente, "Monorepos: a multivocal literature review," *arXiv preprint arXiv:1810.09477*, 2018.

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

[35] S. Kokoska and D. Zwillinger, *CRC standard probability and statistics tables and formulae*. Crc Press, 2000, Section 2.2.24.

[36] M. Lanza and R. 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.

[37] J.-W. Lin, N. Salehnamadi, and S. Malek, "Test automation in open-source android apps: A large-scale empirical study," ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 1078–1089. [Online]. Available: https://doi.org/10.1145/3324884.3416623

[38] K. 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*, vol. 50, no. 302, pp. 157–175, 1900.

[39] APA, *Publication manual of the American Psychological Association*, 4th ed. American Psychological Association, 1994, pp. 16–18.

[40] J. F. Healey, *Statistics: A Tool for Social Research*, 8th ed. Wadsworth Cengage Learning, 2009, pp. 316–317.

[41] D. C. Howell, *Statistical methods for psychology*, 5th ed. Wadsworth, 2007, p. 165.

[42] H. Cramér, *Mathematical methods of statistics*. Princeton University Press, 1946, ch. 21, p. 282.

[43] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. L. Erlbaum Associates, 1988, pp. 224–226.

[44] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: a practical and powerful approach to multiple testing," *Journal of the Royal statistical society: series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.

[45] B. Lin, G. Robles, and A. 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)*. IEEE, 2017, pp. 66–75.

[46] I. Samoladas, L. Angelis, and I. Stamelos, "Survival analysis on the duration of open source projects," *Information and Software Technology*, vol. 52, no. 9, pp. 902–922, 2010.

[47] E. Constantinou and T. Mens, "An empirical comparison of developer retention in the rubygems and npm software ecosystems," *Innovations in Systems and Software Engineering*, vol. 13, no. 2, pp. 101–115, 2017.

[48] O. Aalen, O. Borgan, and H. Gjessing, *Survival and event history analysis: a process point of view*. Springer Science & Business Media, 2008.

[49] E. L. Kaplan and P. Meier, "Nonparametric estimation from incomplete observations," *Journal of the American statistical association*, vol. 53, no. 282, pp. 457–481, 1958.

[50] T. R. Fleming and D. P. Harrington, *Counting processes and survival analysis*. John Wiley & Sons, 2011, vol. 169.

[51] R. C. Mittelhammer, G. G. Judge, and D. J. Miller, *Econometric foundations*. Cambridge University Press, 2000.

[52] N. 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*, vol. 4, no. 1, pp. 13–20, 2008.

[53] R. A. Armstrong, "When to use the Bonferroni correction," *Ophthalmic and Physiological Optics*, vol. 34, no. 5, pp. 502–508, 2014.

[54] A. J. Viera and J. M. Garrett, "Understanding interobserver agreement: the kappa statistic," *Family medicine*, vol. 37, no. 5, pp. 360–363, 2005.

[55] N. Haenni, M. Lungu, N. Schwarz, and O. Nierstrasz, "Categorizing developer information needs in software ecosystems," in *Proceedings of the 2013 international workshop on ecosystem architectures*, 2013, pp. 1–5.

[56] C. Bogart, C. Kästner, J. Herbsleb, and F. 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*, 2016, pp. 109–120.

[57] C. Bogart, C. Kästner, and J. 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)*. IEEE, 2015, pp. 86–89.

[58] M. Wessel, I. Wiese, I. Steinmacher, and M. A. Gerosa, "Don't disturb me: Challenges of interacting with softwarebots on open source software projects," *arXiv preprint arXiv:2103.13950*, 2021.

[59] M. Wessel, B. M. De Souza, I. Steinmacher, I. S. Wiese, I. Polato, A. P. Chaves, and M. A. Gerosa, "The power of bots: Characterizing and understanding bots in oss projects," *Proceedings of the ACM on Human-Computer Interaction*, vol. 2, no. CSCW, pp. 1–19, 2018.

[60] M. Wessel and I. Steinmacher, "The inconvenient side of software bots on pull requests," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 51–55.

[61] M. Wyrich, R. Ghit, T. Haller, and C. Müller, "Bots don't mind waiting, do they? comparing the interaction with automatically and manually created pull requests," *arXiv preprint arXiv:2103.03591*, 2021.

[62] S. Mirhosseini and C. 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)*. IEEE, 2017, pp. 84–94.

[63] A. Zerouali, T. Mens, A. Decan, and C. D. Roover, "On the impact of security vulnerabilities in the npm and RubyGems dependency networks," *Empir. Softw. Eng.*, vol. 27, no. 5, p. 107, 2022.

[64] D. Liu, M. J. Smith, and K. 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*, 2020, pp. 1–8.

[65] M. Wessel, A. Serebrenik, I. Wiese, I. Steinmacher, and M. 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)*. IEEE, 2020, pp. 1–11.

[66] D. Kavaler, A. Trockman, B. Vasilescu, and V. 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)*. IEEE, 2019, pp. 476–487.

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

[68] H. Mohayeji, F. Ebert, E. Arts, E. Constantinou, and A. Serebrenik, "On the adoption of a todo bot on github: A preliminary study," in *2022 IEEE/ACM 4th International Workshop on Bots in Software Engineering (BotSE)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 23–27. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/3528228.3528408

[69] R. He, H. He, Y. Zhang, and M. Zhou, "Automating dependency updates in practice: An exploratory study on github dependabot," 2022. [Online]. Available: https://arxiv.org/abs/2206.07230

[70] B. Chinthanet, R. G. Kula, S. McIntosh, T. Ishio, A. Ihara, and K. Matsumoto, "Lags in the release, adoption, and propagation of npm vulnerability fixes," *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–28, 2021.

[71] J. Huang, N. Borges, S. Bugiel, and M. Backes, "Up-to-crash: Evaluating third-party library updatability on android," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 15–30.