Assessing and Enhancing the Security of Software Packages

Mahmoud Alfadel

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy (Software Engineering) at

Concordia University

Montréal, Québec, Canada

December 2021

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Mahmoud Alfadel**

Entitled: **Assessing and Enhancing the Security of Software Packages**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Sbastien Le Beux

_____ External Examiner
Dr. Kevin Schneider

_____ Examiner
Dr. Nikolaos Tsantalis

_____ Examiner
Dr. Tse-Hsun (Peter) Chen

_____ Examiner
Dr. Mohammad Mannan

_____ Supervisor
Dr. Emad Shihab

Approved by

_____
Dr. Lata Narayanan, Chair

2021

_____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Assessing and Enhancing the Security of Software Packages

Mahmoud Alfadel, Ph.D.

Concordia University, 2022

Modern software applications are developed with increasing reliance on open source software packages (i.e., dependencies). This dependence on open source packages is highly beneficial to software development, since it speeds up development tasks and improves software quality. However, it also has implications to the security of software applications. Dependencies with security vulnerabilities have the potential to expose hundreds of applications to security breaches, potentially causing huge financial and reputation damages. Hence, it is essential to build a solid understanding of the security health of software packages and how developers react once the vulnerabilities are found in the packages they depend on.

To this end, in this thesis, we conduct empirical studies that shed light on the security state of software packages from two aspects. In the first aspect, we study the lifecycle of security vulnerabilities in packages. We analyse how long it takes to discover and fix security vulnerabilities that affect software packages, to better evaluate the response of software ecosystems to security vulnerabilities. Once the vulnerability is discovered, it is also critical to mitigate its impact on software applications. Therefore, in the second aspect, we evaluate the effectiveness of existing mechanisms in mitigating the impact of package vulnerabilities. We assess the role of two popular mechanisms (i.e., the code review process and software bots) for tackling security vulnerabilities in software packages. The insights from our studies in this thesis can help researchers and practitioners better understand the security implications of adopting software packages. Also, leveraging our findings in the studies, we provide a series of implications that can help improve the process of discovering, fixing and managing package vulnerabilities. Finally, the implications of our work lead us to build several prototype tools to increase developers' awareness to vulnerable packages

that affect their projects, and help them better plan the maintenance of their software packages from a security perspective.

Statement of Originality

I, Mahmoud Alfadel, hereby declare that I am the sole author of this thesis. All ideas and inventions attributed to others have been properly referenced. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Dedications

To my family.

Acknowledgments

I would like to express my greatest gratitude to my supervisor, Dr. Emad Shihab, for his guidance and unconditional support. This thesis would have been impossible to complete without his aid and support. Emad, thank you for giving me the opportunity to tackle this challenge, for believing in me. You were there every time I hesitated to take the next step and guided me to success. I have learned a great deal from you beyond as a researcher.

I want to thank my committee members, Drs. Nikolaos Tsantalis, Tse-Hsun (Peter) Chen, Mohammad Mannan, and Kevin Schneider for taking the time to read and critique my work and for their valuable suggestions and comments.

To all my lab mates, thank you, you made this experience more enjoyable. I wish you all the success through your journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Research Statement

1.1 Introduction

Modern software systems increasingly depend on open source software (OSS) packages. These packages (e.g., dependencies) are available from online repositories and often delivered and managed by package management systems, such as npm for JavaScript packages and PyPi for Python packages. The collection of packages that are reused by a community, together with their history and contributors is denoted as a software ecosystem.

While software ecosystems have many benefits, providing an open platform with a large number of reusable packages that speed up development tasks, such openness may lead to the spread of vulnerabilities through a package network, making the vulnerability discovery and fixing much more difficult. A vulnerability is a flaw that allows unauthorized actions and/or malformed access to a given software project, if exploited (Dowd, McDonald, & Schuh, 2006). For example, Contrast Security, a software security company, reported that 80% of the code written in todays applications depend on external packages, and approximately one fourth of package downloads have known vulnerabilities (Williams & Dabirsiaghi, 2012).

These vulnerabilities can impact an organization if exploited. In fact, there are many examples of such cases. One such example is the Equifax cybersecurity incident (Equifax, 2017), where a vulnerability in the Apache Struts package led to unauthorized access to consumers personal information and credit card numbers. Prior studies, e.g., (Zimmermann, Staicu, Tenny, & Pradel,

2019a) showed that a significant proportion (up to 40%) of all npm packages depend on code with at least one publicly known vulnerability, which increases the risk of a vulnerable package in a software application.

Hence, among the most critical aspects in dealing with package vulnerabilities is how fast developers can discover and fix the vulnerability, and how fast the applications update their packages to incorporate the fixed versions. The delay between discovering a package vulnerability and releasing its fix may increase the likelihood of an exploit. Heartbleed, a security vulnerability in OpenSSL package, is perhaps the most infamous example. The vulnerability was introduced in 2012 and remained uncovered until April 2014. After its disclosure, researchers found more than 692 different sources of attacks attempting to exploit the vulnerability in applications that used the OpenSSL package (Durumeric et al., 2014).

Therefore, it is vital to **provide developers with information regarding the lifecycle of vulnerability discovery and fix.** Once the package vulnerability is discovered and reported, it is also essential to mitigate its harmful impact, and hence, **understanding the effectiveness of existing solutions in addressing the security of open-source packages** is also important, to uncover the possible issues in these solutions to improve them.

To this end, the goal of our research is to assess and enhance the security of open source software packages in software ecosystems. To achieve this goal, we address the following two aspects:

(1) Understand the lifecycle of package vulnerabilities. For this goal, we perform two studies, as follows:

   (a) A study to understand the lifecycle of vulnerability discovery and fix, e.g., how long vulnerabilities take to be discovered and fixed?

   (b) Vulnerable packages have a large negative impact on software applications that rely on them. Hence, we study the impact of vulnerable packages on dependent applications, e.g., how often and for how long are applications exposed to vulnerable packages?

(2) Assess the role of existing mechanisms in improving package security. For this goal, we evaluate two mechanisms:

(a) We explore the role of code review process in finding and mitigating security issues that affect software packages.

(b) We evaluate the role of a popular software bot, called Dependabot, in tracking and fixing vulnerable packages that affect the software application.

1.2 Research Statement

Recent studies have shown that the use of packages boosts productivity and software quality, reducing time-to-market (Inoue, Sasaki, Xia, & Manabe, 2012). However, packages also introduce major problems, as they may increase the impact of security vulnerabilities (Lim, 1994a). A security vulnerability in a highly-used package may directly impact hundreds of applications, leading to significant financial costs and reputation loss. These concerns and motivations led to the formulation of this thesis problem statement, which is stated as follows:

With the popularity of software packages and given the fact that modern software systems are increasingly depending on packages, we hypothesize that it is challenging for developers to manage the security when depending on packages. We conduct empirical studies to advance our understanding and assessment of package security vulnerabilities and understand the effectiveness of existing mechanisms to mitigate the impact of these vulnerabilities. We use our findings to propose tool prototypes to improve the maintenance of software packages in software projects.

1.3 Thesis Overview

In this section, we provide a brief overview of the thesis. The rest of the thesis consists of six chapters, which can be classified into three main parts. In the first part (Chapter 2), we provide background and related work to the thesis. In the second part (Chapters 3, 4, 5, and & 6), we present four empirical studies related to the two aspects aforementioned in Section 1.1. Finally, Chapter 7 concludes the thesis and discusses avenues for future work.

1.3.1 Chapter 2: Background and Literature Review

This chapter contains two main sections. The first section presents a background and terminology related to the concept of package management systems. The second section presents existing work that is related to this thesis. In particular, we focus on studies that investigate software packages in general, and studies that approach the security-related issues in package. Also, we discuss work related to existing studies that evaluate mechanisms used to enhance the package security.

1.3.2 Chapter 3: Analysing the Lifecycle of Package Vulnerabilities

Security vulnerabilities are among the most pressing problems in open source software packages. It may take a long time to discover and fix vulnerabilities in packages.

To better understand how software packages in ecosystems are impacted by security vulnerabilities, in Chapter 3, we present an empirical study of 550 vulnerability reports affecting 252 Python packages in the Python ecosystem (PyPi). In particular, we study the propagation and life span of security vulnerabilities, accounting for how long they take to be discovered and fixed. Our findings show that the discovered vulnerabilities in Python packages are increasing over time, and often take more than 3 years to be discovered. The majority of these vulnerabilities (50.55%) are only fixed after being publicly announced, giving ample time for attackers exploitation. We compare the findings in our study to a similar study on the npm ecosystem. We find similarities in some characteristics of vulnerabilities in PyPi and npm. Yet, we observe some divergences that can be attributed to specific PyPi policies. By leveraging our findings, we provide a series of implications that can enhance the security of software ecosystems by improving the process of discovering, fixing and managing package vulnerabilities. Finally, we support our analysis by building a tool (called DEPHEALTH) that aims to help software developers when selecting their packages, e.g., we employ some of the study analysis and provide it to developers through a public web site. Such a tool can give package users insights about the prioritization of discovering and fixing package vulnerabilities. This work was published in the IEEE International Conference on Software Analysis, Evolution and Re-engineering (SANER) and was invited to submit an extended version of the manuscript for

a special issue of the Empirical Software Engineering journal (EMSE).

1.3.3 Chapter 4: Examining the Discoverability of Package Vulnerabilities Impacting Software Applications

The reliance on vulnerable packages (i.e., dependencies) is a major threat to software systems. Software applications that rely on packages with publicly known vulnerabilities are exposed to a higher risk, since the vulnerability is known to the community and the level of exploitation reaches its peak.

To better understand the impact of a package vulnerability on a software application, in Chapter 4, we examine the package vulnerabilities based on their disclosure timeline (i.e., discoverability aspect). First, we define three discoverability levels for dependency vulnerabilities based on their timeline: hidden (unknown), reported, and public. Then, we conduct a large-scale empirical study involving 6,546 active and mature open-source JavaScript applications. Our results show that 67.9% of the examined applications depend on at least one vulnerable package. Taking discoverability into account, we found that although the majority of the affected applications (99.42%) depend on packages with hidden vulnerabilities at the time of analysis, 206 (4.63%) applications were still exposed to dependencies with public vulnerabilities. The major culprit for the applications being affected by public vulnerabilities is the lack of dependency updates, i.e., in 90.8% of the vulnerable packages, a fix for the vulnerable dependency is available but not patched in the application. Moreover, we find that applications remain affected by public dependency vulnerabilities often for a long time (103 days), which can put their software systems at risk. Finally, we devise DEPREVEAL, a tool that reports the historical exposure of JavaScript projects to dependency vulnerabilities, to help developers better plan the maintenance of their software project. This work was submitted and currently under review at the ACM Transactions on Software Engineering and Methodology, TOSEM. 2021.

1.3.4 Chapter 5: Studying The Role of Code Review in Enhancing Package Security

Modern code review is a widely-used practice that project maintainers adopt to improve the overall quality of software. Much of prior work has shown that code review has an important role

in improving software quality, however, in-depth analysis on the effectiveness of code review in relation with security issues is limited.

Therefore, in Chapter 5, we explore the role of code review in finding and mitigating security issues. In particular, we investigate 10 active and popular JavaScript projects to understand what types of security issues are raised during code review, and what kind of mitigation strategies are employed by project maintainers to address them. We study 171 pull-requests (PRs) with raised security concerns, which represent a small fraction of all PRs in the studied projects. However, we find that such issues are discussed at length by project maintainers. Moreover, we found that code review is effective at identifying certain types of issues, e.g., Race Condition, Access Control, and ReDOS; we observe that dealing with such issues require in-depth knowledge of the project domain and implementation specifics of the issue. When analysing how maintainers respond to the raised security issues, we found that ~ 55% of the issues are frequently addressed and mitigated. In other cases, security issues ended up not being fixed or are ignored by project maintainers, which may put the project users at risk. Leveraging our findings, we offer several implications that support the role of reviewing code for security concerns.

1.3.5 Chapter 6: Evaluating the Use of Dependabot for Patching Package Vulnerabilities

As software projects depend on multiple external dependencies, developers struggle to constantly track and check for corresponding security vulnerabilities that affect their project dependencies. To help mitigate this issue, Dependabot has been created, a bot that issues pull-requests (PRs) to automatically update vulnerable dependencies. However, little is known about the degree to which developers adopt Dependabot to help them update vulnerable dependencies.

In Chapter 6, we investigate 2,904 JavaScript open-source GitHub projects that subscribed to Dependabot. Our results show that the vast majority (65.42%) of the created security-related PRs are accepted, often merged within a day. Through manual analysis, we identify 7 main reasons for Dependabot security PRs being not merged, mostly related to concurrent modifications of the affected dependencies rather than Dependabot failures. Interestingly, only 3.2% of the manually examined PRs suffered from build breakages. Finally, we model the time it takes to merge a

Dependabot security pull-request using characteristics from projects, the fixed vulnerabilities and issued PRs. Our model reveals 5 significant features to explain merge times, e.g., projects that have relevant experience with Dependabot security PRs are most likely to be associated with rapid merges. Surprisingly, the severity of the dependency vulnerability and the potential risk of breaking changes are not strongly associated with the merge time. Our findings indicate that Dependabot provides an effective platform for increasing awareness to dependency vulnerabilities and help developers mitigate vulnerability threats in JavaScript projects. Leveraging some implications of our findings, we build DepCombine, a tool on top of Dependabot, to improve the process of tracking and merging Dependabot security PRs in GitHub repositories. This work was published in the IEEE International Conference on Mining Software Repositories (MSR).

1.4 Thesis Contributions

The main contributions of this thesis are the following:

- We perform an empirical study to analyse the lifecycle of security vulnerabilities in the Python ecosystem. Our study covers 12 years of reported vulnerabilities, affecting 252 Python packages.
- We conduct an empirical study on 6,546 open-source JavaScript applications to determine the prevalence of affected applications that rely on vulnerable dependencies taking into consideration the discoverability aspect. We also examine why these applications end up depending on vulnerable versions of the package in order to better understand how we can mitigate such issues.
- We investigate security issues that are identified through code reviews of popular JavaScript projects. Also, we manually build and validate a code review dataset that contains security-related reviews in the studied project.
- We provide empirical evidence for understanding developers adoption of Dependabot security automated PRs in open source JavaScript projects.

- Leverging our findings in the above studies, we provide actionable implications and suggestions for practitioners and researchers. Besides, we develop several tool prototypes that aim at increasing developers awareness to the security aspects we analysed in our studies, and support developers with the process of maintaining vulnerable packages in their projects.

1.5 Related Publications

Most of the work presented in this thesis has been previously published or submitted to different venues, as follows:

- **Mahmoud Alfadel**, Diego Elias Costa and Emad Shihab. "Empirical Analysis of Security Vulnerabilities in Python Packages". *In IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER*. 2021. [Invited to a paper extension in EMSE]
- **Mahmoud Alfadel**, Diego Elias Costa, Emad Shihab and Mouafak Mkhallalati. "On the Use of Dependabot Security Pull Requests". *In IEEE/ACM 18th International Conference on Mining Software Repositories, MSR*. 2021.
- **Mahmoud Alfadel**, Diego Elias Costa, Emad Shihab and Bram Adams. "On the Discoverability of Vulnerabilities Impacting JavaScript Projects". *In ACM Transactions on Software Engineering and Methodology, TOSEM*. 2021. [Under Review]
- **Mahmoud Alfadel**, Nicholas Nagy, Diego Elias Costa, Emad Shihab, and Rabe Abdalkareem. "The Role of Code Review in Enhancing Projects Security". *In ACM Transactions on Software Engineering and Methodology, TOSEM*. 2021. [Under review]

1.6 Thesis organization

The rest of the thesis consists of six chapters that are organized as follows: Chapter 2 provides terminology related to the concept of package management systems, and research related to software packages and vulnerabilities. Chapters 3, 4, 5, and 6 present the results of four empirical studies

related to the aspects analysed in this thesis. Finally, Chapter 7 summarizes the thesis and discusses some directions of future work.

Chapter 2

Background and Literature Review

The concept of package reuse is becoming an important topic in the field of software engineering research. With the emergence of COTS (components off the shelf) in 2000s, the research started to conduct studies on the evolution and management of software packages (Bohner, 2002; Maiden & Ncube, 1998). These studies advocated the need for building software ecosystems with powerful package managers and repositories to improve the process of package reuse (Frakes & Kang, 2005). Nowadays, software ecosystems have become very popular and most software systems today have many dependencies. A recent industrial report has shown that 96% of the analyzed applications in the study depend heavily on external packages, by making use of them in more than 50% of the application code-base (Synopsys, 2019). However, while package are beneficial for productivity, they have a security implications; a recent report by Snyk.io showed that 83% of organizations use vulnerable packages and that 77% of the 430,000 websites crawled by them, run at least one vulnerable JavaScript package (*Snyk report*, accessed on 12/10/2021). These reported figures are worrisome given our everyday dependence on software systems. That said, much of the aforementioned reports motivated us to assess software packages examining vulnerabilities in them.

First, in this chapter, we present an overview of the relevant terminology to our research. We then present the related works of our thesis, including related studies on: a) software packages and their vulnerabilities; b) existing mechanisms to mitigate the impact of package vulnerabilities.

2.1 Terminology

In this section, we present the main terms used throughout our studies. Inspired by (Decan, Mens, & Grosjean, 2019; Zerouali, 2019), Table 2.1 defines the main terms used in this thesis.

Table 2.1: Definition of terms used in the thesis.

Term	Definition
Package Manager	A coherent collection of software tools that automate the process of installing, configuring, upgrading or removing software packages on a computer's operating system in a consistent manner. The node package manager (or short npm) is the package manager that supports JavaScript developers with reusable code packages. PyPi is the package manager for Python packages.
Package/Library	A computer program providing specific functionalities that are available from online repositories and often delivered by a package manager. A package usually exists in many versions which are called releases.
Package Release	A specific version of a package that can be accessed from a package repository and installed through the package manager. It includes what is needed to build, configure and deploy the package version.
Package Vulnerability	Is a vulnerability that affects the code of a package version.
Package Vulnerability Advisories	a centralized source of vulnerability reports specific to a package manager, e.g., npm advisories are a source of vulnerability reports that are specific to npm packages.
Dependency	A package that is used in an application to compile and build its binaries is called a dependency.
Application/Dependent Application	While packages are a set of computer programs that can be reused in other programs and published by the software ecosystem, Applications are computer programs that use packages to facilitate their development and are NOT published in a software ecosystem.
Dependency Semantic Versioning (SemVer)	A condition that is used by a dependency to restrict the supported versions of the target package.

2.2 Literature review

In this section, we discuss the related literature by focusing on: a) studies that investigate software ecosystems in general and studies that approach the link of security-related issues and software ecosystems; b) studies that evaluate or propose solutions to mitigate the impact of vulnerable packages.

2.2.1 Work Related to Software Packages and Vulnerabilities

Software Ecosystems. Several aspects of *Software Ecosystems* have been subject of great interest in the related literature. For example, some works analysed the ecosystem's growth (Fard & Mesbah, 2017; Wittern, Suter, & Rajagopalan, 2016). Fard et al. (Fard & Mesbah, 2017) showed that the number of dependencies in npm projects is 6 on average and the number is always in a growing trend.

Other works qualitatively studied the fragility and breaking changes in software ecosystems. Bogart et al. (Bogart, Kästner, Herbsleb, & Thung, 2016) compared the ecosystems Eclipse, CRAN, and npm in terms of practices that are used by developers to decide about causes of API breaking changes. They found that all three ecosystems are significantly different in terms of practices towards breaking changes, due to some particular community values in each ecosystem. Raemaekers et al. (Raemaekers, Van Deursen, & Visser, 2014) studied the impact of dependency constraint on the breaking changes. They suggested that developers may need to carefully select and adhere to dependency constraint (semver). A recent study by Decan et al. (Decan & Mens, 2019) empirically compared four ecosystems (Cargo, npm, Packagist and Rubygems) with respect to semver compliance. They concluded that the degree of semver compliance is increasing over time, and this can be partially attributed to the ecosystem-specific policies and characteristics.

Others focused on analysing a special types of packages, what is so-called trivial packages. Abdalkareem et al. (Abdalkareem, Nourry, Wehaibi, Mujahid, & Shihab, 2017) investigated why developers use trivial packages as a dependency in the npm ecosystem, and they showed that this practice is popular among developers. While interviewed developers did not consider those packages as harmful, they were found to be less tested than other packages. Kula et al. (Kula, Ouni, German, & Inoue, 2017) studied the impact of the same type of packages in the npm ecosystem. They found that some micro-packages have long dependency chains and incur just as much usage costs as other npm packages.

A few studies conducted a comparison across software ecosystems. Decan et al. (Decan, Mens, & Claes, 2016; Decan et al., 2019) empirically compared the dependency evolution in 7 ecosystems (including npm). They discovered some differences across ecosystems that can be attributed to

ecosystems' policies. For instance, the CRAN ecosystem has a policy called "rolling release", where packages should always be compatible with the latest release of their dependencies since CRAN can only install the latest release automatically. Hence, developers could face issues when updating because a change in one package can affect many others.

While the aforementioned work served as a motivation to our investigation, the focus of our studies in this thesis is fundamentally different. Our work can be used to complement previous work by providing a view on another important quality metric of software ecosystems: security vulnerabilities.

Security Vulnerabilities in Packages. The potential fragility of the ecosystems shown in previous studies (e.g., (Bogart et al., 2016; Bogart, Kstner, & Herbsleb, 2015)) has motivated researchers to examine security vulnerabilities, as vulnerabilities are one of the most problematic aspects of software ecosystems (Thompson, 2003). A study by Pham et al. (Pham, Nguyen, Nguyen, & Nguyen, 2010) presented an empirical study to analyse vulnerabilities in the source code, and found that most vulnerabilities are recurring due to software code reuse and package adoption.

Other studies focused on analysing vulnerabilities in software ecosystems. Hejderup et al. analysed 19 npm vulnerable packages, and found that the number of vulnerabilities is growing over time (Hejderup, 2015). Zimmermann et al. (Zimmermann et al., 2019a) studied the security risk of the npm ecosystem dependencies and showed that individual packages could impact large parts of the entire ecosystem. They also observed that a very small number of maintainers (20 accounts) could be used to inject malicious code into thousands of npm packages, a problem that has been increasing over time. A study by Zapata et al. (Zapata et al., 2018) assessed the danger of having vulnerabilities in npm packages by analyzing function calls of the vulnerable functions, and found that 73.3% of the 60 studied projects were actually safe because they did not make use of the vulnerable functionality.

The management of package vulnerabilities was also studied in other ecosystems like packages written in Java. Kula et al. (Kula, German, Ouni, Ishio, & Inoue, 2018) explored how developers respond to the available security awareness mechanisms such as library migration, and found that developers were unaware of most vulnerabilities in dependencies and prefer to use outdated

versions to reduce the risks of breaking changes. Ponta et al. (Ponta, Plate, & Sabetta, 2018, 2020) proposed a code-centric approach to detect and mitigate open source vulnerabilities for Java industry grade applications. Pashchenko et al. (Pashchenko, Plate, Ponta, Sabetta, & Massacci, 2018, 2020) proposed an technique that addresses the over-estimation problem of approaches that report vulnerable dependencies in the Java ecosystem. The authors highlighted that many of the vulnerable dependencies were not actually deployed, and hence, their impact was neglected.

Inspired by previous studies, and supported by the fact that different ecosystems have different characteristics and policies, we conducted a study (in Chapter 3) to examine security vulnerabilities in the PyPi ecosystem. We studied several aspects related to the lifecycle of vulnerability discovery and fix. Moreover, little is known about the impact of using vulnerable packages on the dependent application, i.e., how often and for how long are applications exposed to dependencies through the application development history? Therefore, we study (in Chapter 4) the impact of vulnerable packages on software applications, by introducing a new classification for package vulnerabilities based on the vulnerability disclosure timeline.

2.2.2 Work Related to Solutions for Mitigating the Impact of Vulnerable Packages

The Role of Code Review to Prevent Security Issues. A plethora of work on code review topic studied the effect of code review process on finding defects. Thongtanunam et al. found that developers are often most concerned about documentation and structure to enhance evolvability, and fix functional issues (Thongtanunam, McIntosh, Hassan, & Iida, 2015). Beller et al. revealed that most changes of open-source systems in code review are indeed related to the functionality aspect (Beller, Bacchelli, Zaidman, & Juergens, 2014). The study by Bacchelli and Bird (Bacchelli & Bird, 2013) showed that most changes of open-source systems in code review are also related to the functionality aspect. The work by (Mäntylä & Lassenius, 2008) reported similar outcomes for other industrial and academic projects. McIntosh et al. (McIntosh, Kamei, Adams, & Hassan, 2014, 2016) examined the impact of code review coverage and participation on the code review quality. They found that projects with low code review coverage and participation are estimated to produce more post-release defects, meaning that poor code review negatively impacts the software quality.

Spadini et al. (Spadini et al., 2019) examined the impact of a code review practice called

Test-Driven Code Review (TDR), where a reviewer inspects patches by examining the changed test code before the changed production code. Their experiments show that developers adopting TDR find more defects than ones found through examining production code. Spadini at al. (Spadini, Aniche, Storey, Bruntink, & Bacchelli, 2018) also examined how code review is used for ensuring the quality of test code. They find that developers tend to discuss test files significantly less than production files. The paper recommends that the project should set aside sufficient time for reviewing test files.

Other most relevant work to our work in this thesis focused on security code review (Bacchelli & Bird, 2013; Bosu, 2014; di Biase, Bruntink, & Bacchelli, 2016; Paul, Turzo, & Bosu, 2021). For example, Bacchelli and Bird (Bacchelli & Bird, 2013) observed (based on interviews and surveys) that code review is mainly motivated for finding defects and formatting issues while missing the fact that there were security issues. Di Biase et al. (di Biase et al., 2016) analyzed the Chromium system to understand the factors that may lead to find security issues during code review, and found, for example, that reviews conducted by more than 2 reviewers are being more successful at finding security issues. Also, they found that reviewers tend to find domain-specific security issues (e.g., Cross-Site Scripting XSS) more than language-specific issues (e.g., C++ issues).

Common to these studies is that they investigate security issues identified through code review of software projects. One of the main objectives of this thesis is to conduct an empirical study to assess the role of code review in identifying security issues in projects that have been published as packages in software ecosystems. Moreover, we aim to understand the mitigation strategies employed by project maintainers to tackle the issues.

Package management tools for security vulnerabilities. Previous studies (e.g.,(Kula et al., 2018; Zerouali, Constantinou, Mens, Robles, & González-Barahona, 2018)) have shown that projects are slow in terms of responding to security vulnerabilities that are publicly announced, which is sometimes due to factors related to resources and process management. The software development community has proposed several tools that help developers be aware of dependency updates and vulnerabilities. For example, Cadariu (Cadariu, Bouwers, Visser, & van Deursen, 2015) developed a Vulnerability Alert Service (VAS), which scans Maven dependencies against vulnerabilities using

the Common Vulnerabilities and Exposures (CVE) database. Apiwave (Hora & Valente, 2015) is another tool that tracks API migrations in order to help developers be aware of their project dependency updates. The current version of the Apiwave tool provides data for 650 Java projects, from which 320K APIs were extracted. One limitation of these tools is that they only send alerts to notify developers about the vulnerable dependencies without being able to automatically fix them.

Other works focused on identifying dependency vulnerabilities at a more fine-grain level. For example, Ponta et al. (Ponta et al., 2018) proposed a code-centric tool to detect and mitigate dependency vulnerabilities for Java industry applications used in the SAP organization. Also, a study by Bodin et al. (Chinthanet et al., 2020) showed that the code-centric detection tool is viable, however, there are several challenges related to the JavaScript language and the complexity of the application dependencies.

Dependabot (Dependabot, accessed on 12/10/2021) is a bot (acquired by GitHub in 2019) that creates pull requests to monitor project dependencies and help developers automatically integrate dependency updates and vulnerability fixes. Also, it provides information about the vulnerability, such as its severity, versions affected, information about the issue from the advisory report, which developers can analyze to consider the risks of not updating. Moreover, the PR contains information about the compatibility of the PR with the project, calculated based on the outcome of updates done by similar projects (*Dependabot Score*, accessed on 12/10/2021).

While previous works propose several solutions to help developers be aware of vulnerable packages in their projects, little is known about to which extent such tools can convince developers to upgrade out-of-date dependencies in their projects. Given that dependency updates for vulnerability fixes have a critical impact, one of the main goals of this thesis (Chapter 6) is to specifically focus on evaluating a very popular dependency tool (e.g., Dependabot) at coping with security updates in dependencies.

2.3 Chapter Summary

In this chapter, we present a literature review on studies that investigate software packages and vulnerabilities. Also, we review prior research on some existing mechanisms for mitigating the

impact of vulnerable packages. From the literature review, we find that the majority of prior studies focus on analysing software packages in popular software ecosystems, e.g., npm. To complement previous research, and as different software ecosystems embodied different programming languages and particularities, we argue that it is also important to study other popular programming languages to build stronger empirical evidence about vulnerabilities in software ecosystems. Hence, in Chapter 3, we take a new look and provide a wider picture by studying security vulnerabilities in the PyPi ecosystem, and compare our results with the npm ecosystem. Moreover, to our best knowledge, none of the previous studies specifically investigated the impact of a package vulnerability that affects a dependent application. In Chapter 4, we propose a novel methodology to study the presence of package vulnerabilities in JavaScript applications. To mitigate the impact of package vulnerabilities, prior work proposed several mechanisms, e.g., the literature shows that the code review process has an important role in improving software quality, but in-depth analysis on the effectiveness of code review in relation with security issues is limited. In Chapter 5, we explore the role of code review in finding and mitigating security issues in JavaScript projects. Finally, we find that previous research has created several software bots to mitigate the harmful impact of vulnerable packages. However, little is known about their effectiveness. In Chapter 6, we perform an empirical study to evaluate a popular software bot, called Dependabot, for patching package vulnerabilities that affect JavaScript projects.

Chapter 3

Analysing the Lifecycle of Package Vulnerabilities

Software ecosystems play an important role in modern software development, providing an open platform of reusable packages that speed up and facilitate development tasks. However, this level of code reusability supported by software ecosystems also makes the discovery and fix of security vulnerabilities much more difficult, as software systems depend on an increasingly high number of packages. To better understand the impact of security vulnerabilities in packages, this chapter presents an empirical study of 550 vulnerability reports affecting 252 Python packages in the Python ecosystem (PyPi). Taking into account the severity of vulnerabilities, we analyse how and when these vulnerabilities are discovered and fixed. We report our findings and provide guidelines for package maintainers and tool developers to improve the process of dealing with security issues in software packages.

3.1 Introduction

Modern software systems increasingly depend on external reusable code. This reusable code takes the form of packages (e.g., libraries) and is available from online repositories and often delivered by package management systems, such as npm for JavaScript packages and PyPi for Python packages. The collection of packages that are reused by a community, together with their

users and contributors is denoted as a *software ecosystem*. While software ecosystems have many benefits, providing an open platform with a large number of reusable packages that speed up and facilitate development tasks, such openness and large scale leads to the spread of vulnerabilities through package network, making the vulnerability discovery much more difficult, given the heavy dependence on such packages and their potential security problems (Thompson, 2003).

Many software applications depend on vulnerable packages (Williams & Dabirsiaghi, 2012). The two most critical aspects in dealing with package vulnerabilities are how fast developers can discover and fix the vulnerability, and how fast the applications update their packages to accommodate the fixed versions. The delay between discovering a package vulnerability and releasing its fix may expose the applications to threats and increase the likelihood of an exploit being developed. Heartbleed, a security vulnerability in OpenSSL package, is perhaps the most infamous example. The vulnerability was introduced in 2012 and remained uncovered until April 2014. After its disclosure, researchers found more than 692 different sources of attacks attempting to exploit the vulnerability in applications that used the OpenSSL package (Durumeric et al., 2014).

Hence, studying how vulnerabilities propagate, get discovered and fixed is essential for the health of ecosystems. Recent studies (Decan, Mens, & Constantinou, 2018b; Hejderup, 2015) analysed the impact of vulnerabilities in the npm ecosystem. Decan et al. (Decan, Mens, & Constantinou, 2018b) found that it takes 24 months to discover 50% of npm package vulnerabilities, whilst 82% of the discovered vulnerabilities are fixed before being publicly announced, where they are less likely to be exploited.

While npm is one of the largest software ecosystems to date (*Libraries.io - The Open Source Discovery Service*, 2021), the investigation of npm vulnerabilities provides an important but restricted view of the software development ecosystems. How much of the findings are particular to npm's development culture and how much of it can be generalized to other ecosystems? We argue that it is important to study other software ecosystems to contrast with npm and draw more generalizable empirical evidence about vulnerabilities in software ecosystems. Our argument is supported by previous studies (e.g., (Bogart, Kstner, & Herbsleb, 2015; Decan et al., 2016; Decan, Mens, & Claes, 2017; Decan et al., 2019)) that show differences across ecosystems. For instance, Decan et al. Decan et al. (2019) found that PyPi ecosystem has a less complex and intertwined

network than ecosystems such as npm and CRAN. This is partially due to Python's robust standard library, which discourages developers of using too many external packages in contrast to JavaScript and R ecosystems.

This motivated us to take a new look and provide a wider picture by studying security vulnerabilities in the PyPi ecosystem. Furthermore, Python is a major programming language in the current development landscape, used by 44.1% of professional developers according to the 2020 Stack-Overflow survey (*Stack Overflow Developer Survey*, 2020). We conduct an exploratory research to study *security vulnerabilities prevalence and their respective discovery and fix timeline in the Python ecosystem*. Inspired by the study on the npm ecosystem (Decan, Mens, & Constantinou, 2018b), we aim to answer the following research questions (RQs):

- **RQ$_1$**: How are vulnerabilities distributed in the PyPi ecosystem?
- **RQ$_2$**: How long does it take to discover a vulnerability in the PyPi ecosystem?
- **RQ$_3$**: When are vulnerabilities fixed in the PyPi ecosystem?
- **RQ$_4$**: How long does it take to fix a vulnerability in the PyPi ecosystem?

Also, we compare our study, where applicable, to the npm ecosystem (Decan, Mens, & Constantinou, 2018b).

To answer our research questions, we analyzed 550 vulnerability reports that affect 252 Python packages of which 7,536 package versions are affected. We observed several interesting findings. In some aspects, our study yields similar findings to the ones observed in the npm study (Decan, Mens, & Constantinou, 2018b). For example, vulnerabilities in both ecosystems take a significantly long time to be discovered, approximately 2 years in the npm and 3 years in the PyPi ecosystem.

However, in other aspects, our results show a drastic departure from npm's reported findings. For example, unlike npm, the majority of PyPi vulnerabilities (50.55%) were only fixed after being publicly announced, which may increase the chances of having the vulnerability exploited by attackers. Our further investigation attributes such observation to the particularities of the PyPi ecosystem's protocol of disclosing and publishing vulnerabilities.

Based on our empirical findings, we offer several important implications to researchers and practitioners to help them provide a more secure environment for software ecosystems. Besides, employing our study methodology, we developed a tool called DEPHEALTH to provide developers with metrics related to the timespan of vulnerability discovery and fix, which help for the package selection process.

To summarize, this chapter makes the following contributions:

- We perform the first empirical study to analyse security vulnerabilities in the Python ecosystem. Our study covers 12 years of PyPi reported vulnerabilities, affecting 252 Python packages.
- We compare the findings of our study to a previous study conducted on the npm ecosystem. We also provide implications that aim at a more secure development environment for software ecosystems.
- We make our scripts and dataset of this study publicly available to facilitate reproducibility and future research (M. Alfadel, 2019).

Chapter organization.

This chapter is organized as following: Section 3.2 describes the terminology and the process of collecting and curating our dataset. In Section 3.3, we dive into our study by motivating and describing the methods used to investigate each research question, as well as presenting the findings obtained in our study. We discuss the results and implications of our study in Section 3.4. Section 3.5 presents our tool. We state the threats to validity and limitations to our study in Section 3.6. Related work is presented in Section 3.7. Finally, Section 3.8 concludes our study.

3.2 Study Design

In this section, we present an overview of software vulnerabilities and the terminology adopted throughout this study. We also explain how we collect and prepare the data used to investigate our research questions.

3.2.1 Terminology

The lifetime of a vulnerability typically goes through various stages, according to when a vulnerability was first introduced, discovered and publicly announced. To ground our study, we use the various stages and define dates specific to a package vulnerability:

- **package vulnerability introduction date** indicates when the vulnerability was first introduced in the affected package, i.e., the release date of the first affected version by the package vulnerability.
- **package vulnerability discovery date** indicates the date in which the package vulnerability was discovered and reported to the maintainer of the package.
- **package vulnerability publication date** marks the date when the vulnerability information was publicly announced.
- **package vulnerability fix date** indicates the release date of the first fixed version of the package vulnerability.

Next, we explain how we collect and process the data used to answer our RQs.

3.2.2 Data Collection and Processing

Data Collection. To conduct our study, we collect two datasets: (1) the Python (PyPi) packages and (2) the security vulnerabilities that affect those PyPi packages. We obtain the information of PyPi packages from Libraries.io (Nesbitt & Nickolls, 2018), and the security vulnerabilities from the Snyk.io dataset (Snyk.io, 2017).

To collect the PyPi packages, we use the service Libraries.io since it provides the PyPi packages along with their respective metadata. The metadata provides detailed information about each package such as, the existing versions and the creation timestamps of those versions. Such data is needed to map the affected versions given by our vulnerabilities dataset. Also, we need the versions timestamps to perform time-based analyses, such as the time it takes to discover and fix a vulnerability with respect to the first affected package version.

Table 3.1: Example of a security report extracted from Snyk.io for the `pillow` package.

Information	**Example**
Vulnerability type	Buffer Overflow
Affected package name	pillow
Platform type	PyPi
Vulnerable constraint (affected versions)	< 6.2.2
Vulnerability Discovery date	03 Jan, 2020
Vulnerability Published date	10 Jan, 2020
Severity level	High
Remediation	≥ 6.2.2

To collect the vulnerabilities for the PyPi packages, we resort to the dataset provided by Snyk.io (Snyk.io, 2017). Snyk.io is a platform that monitors security reports to provide a dataset for different package ecosystems, including PyPi, and publishes a series of information about vulnerabilities. We show in Table 3.1 an example of a security report extracted from Snyk.io dataset for the package *pillow*. For each affected package, the dataset specifies the type of vulnerability, the vulnerability constraint (this helps us to specify the affected versions) and the fixed versions (remediation range). Moreover, the report contains the dates when the vulnerability was discovered and the date when it was published on Snyk.io dataset. Severity level has three possible values, high, medium, and low, which are assigned manually by the Snyk.io team based on the Common Vulnerability Scoring System (CVSS) (Allodi & Massacci, 2014).

Data Processing. As a pre-processing step, we need to determine all the vulnerable packages and their associated versions. First, we obtain the list of all versions of all vulnerable packages from the Libraries.io dataset. Then, we determine the affected versions of the vulnerable packages by cross-referencing the vulnerability constraint of the Snyk.io report (e.g., < 6.2.2) and resolving the versions by using the SemVer tool (*semver PyPI*, 2020). In the particular example of Table 3.1, we resolve the constraint < 6.2.2 to a list of 68 versions of the `pillow` package affected by the Buffer Overflow vulnerability.

We want to analyze the time needed to discover a package vulnerability, hence, we need to identify the version that was *first affected* by a vulnerability. To that aim, once we identify the list of affected versions, we consider the first affected version as the oldest version of the vulnerable package. In the example of Table 3.1, the first affected version was the package version 1.0.0.

We also aim to investigate the time it takes to fix a package vulnerability once the vulnerability is discovered. This requires that we identify the *first fixed version* of the package vulnerability. Similar to the identification of the first affected version, once we resolve the remediation range by using the SemVer tool, we collect a list of versions in which the vulnerability is considered fixed. We then assign the first fixed version as the oldest package version present in the list of fixed versions. In the example of Table 3.1, the first fixed version is the package 6.2.2.

Our initial dataset contains 622 vulnerability reports on the PyPi packages. From this original set, 62 vulnerabilities do not match any packages in the Libraries.io database and were removed from our analysis. Following the filtration process applied by Decan *et al.* (Decan, Mens, & Constantinou, 2018b), we also removed 10 vulnerabilities of type "Malicious Package", because they do not really introduce vulnerable code. These vulnerabilities are packages with names close to popular packages (a.k.a. typo-squatting) in an attempt to deceive users at installing their harmful packages. At the end of this filtering process, our dataset contains 550 vulnerability reports. Such reports affect 252 Python packages in PyPi. Note that these 252 Python packages have released a total of 12,210 versions, in which, according to the vulnerable constraint of reports, 7,536 versions (61.7%) contain at least one reported vulnerability. Table 3.2 shows the descriptive statistics of our dataset.

As part of our study goal is to compare our results to the npm study, we verify how our dataset compares with the one used by Decan et al. (Decan, Mens, & Constantinou, 2018b). The npm dataset contains 399 vulnerabilities which affect 269 npm packages with a similar number of versions (14,931) and similar number of affected versions (6,752). Both datasets are comparable in terms of the number of vulnerability reports and the number of affected packages and versions. Finally, note that we collect our dataset in the similar timeline as the npm study in order to make our study comparable and to perform a relatively fair comparison between our findings and the ones reported from npm (Decan, Mens, & Constantinou, 2018b), i.e., we collect all vulnerability reports that were published before Jan. 2018.

Table 3.2: Descriptive statistics of the PyPi dataset.

Source	Stats	#
Libraries.io	PyPi packages	116,527
	Versions of PyPi packages	893,978
Snyk.io	Security reports on PyPi	550
	Corresponding vulnerable packages	252
	Versions of vulnerable packages	12,210
	Affected versions by vulnerability	7,536

3.3 Study Results

In this section, we present the findings of our empirical study. For each RQ, we present a motivation, describe the approach used to tackle the research question and discuss the results of our analysis.

RQ_1: How are vulnerabilities distributed in the PyPi ecosystem?

Motivation. Prior work reported a steady growth of packages in software ecosystems (Decan, Mens, & Constantinou, 2018a; Decan et al., 2019). This growth may have serious repercussions for package vulnerabilities, facilitating their spread to high number of packages and applications, and magnifying their potential for exploitation. Therefore, in this RQ we investigate how software package vulnerabilities are distributed in the PyPi ecosystem. We examine the distribution from three perspectives: a) the trend of discovered vulnerabilities over time; b) how many versions of packages are affected by vulnerabilities; and c) what are the most commonly identified types of vulnerabilities in PyPi.

Approach. To shed light on the distribution of software vulnerabilities in the PyPi software ecosystem, we leverage the following approaches:

In the first analysis, we focus on investigating the trend of discovered vulnerabilities over time in the PyPi ecosystem. In essence, we want to investigate how the number of discovered vulnerabilities change and how many packages are affected as the ecosystem grows? To do that, we group the

discovered vulnerabilities by the time they were reported, and present the evolution of the number of vulnerabilities and packages affected per year. We also break the analysis per severity level, provided by Snyk.io, to help us quantify the level of threat of newly discovered vulnerabilities in the ecosystem.

In the second analysis, we investigate the vulnerabilities distribution over package versions. A single vulnerability can impact many versions of a package, making it harder for dependents to select a version unaffected by this vulnerability. To that aim, we utilize the vulnerability constraint provided by the Snyk.io dataset (mentioned in Table 3.1, Section 3.2.2) to identify the list of affected versions by a vulnerability.

The third analysis has the goal of reporting the most commonly identified vulnerability types in the PyPi ecosystem. The Snyk.io dataset associates each vulnerability report with a Common Weakness Enumeration (CWE) (*CWE list*, accessed on 12/10/2021), aiming at categorizing vulnerabilities based on the explored software weaknesses (e.g. Buffer Overflow). Currently, CWE contains a community-developed list of 700 common software weaknesses. We examine the frequency of vulnerability types to establish a profile of the vulnerabilities in the PyPi ecosystem. In adittion, we also break our analysis by severity level to investigate how the threat levels are distributed in each vulnerability type.

Results. Figure 3.1 shows the number of discovered vulnerabilities as well as the number of packages being affected over the years. **We observe a steady increase in the number of vulnerable packages, accompanying the PyPi ecosystem growth**. In 2012, in the middle of this ecosystem lifetime, 27 packages were discovered to be vulnerable, in 2016 this number increased three-fold, i.e., 90 vulnerable packages were newly discovered.

Figure 3.2 presents the introduction of vulnerabilities over time by the severity level, showing that **the majority of newly discovered vulnerabilities are of medium and high severity.** Overall, the vulnerabilities classified with medium severity make the bulk of 71.64% of all vulnerabilities, followed by high severity vulnerabilities representing 20.73% of our dataset. These findings are worrisome to the PyPi community, as such critical vulnerabilities have a higher chance of being exploited, i.e., allow an attacker to execute malicious code and damage the software.

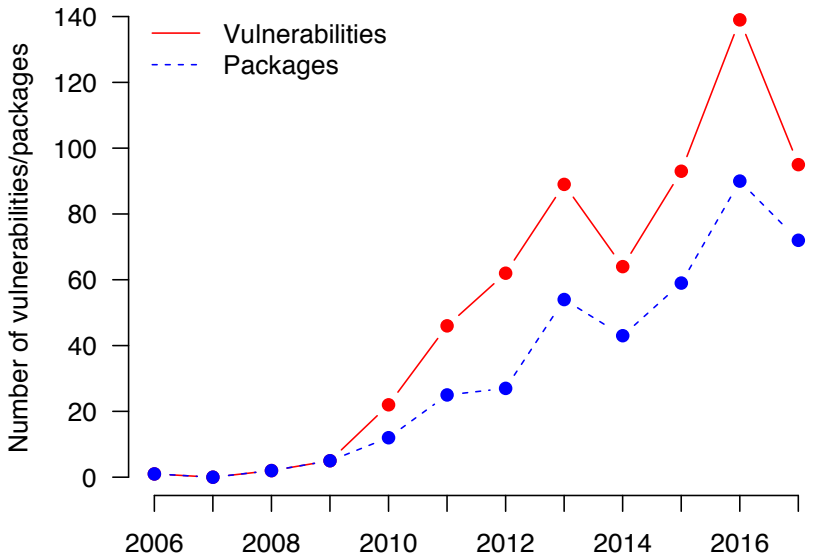


Figure 3.1: Introduction of vulnerabilities and packages being affected per year.

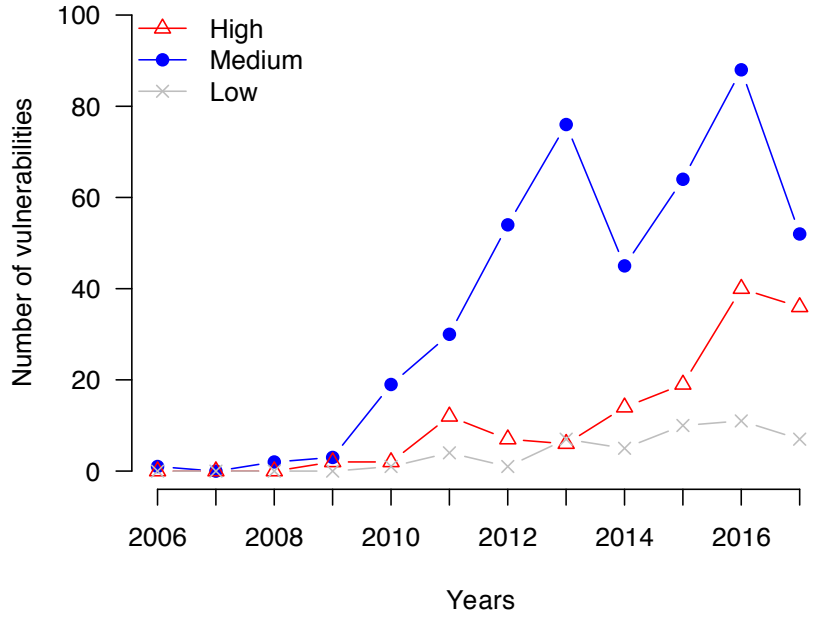
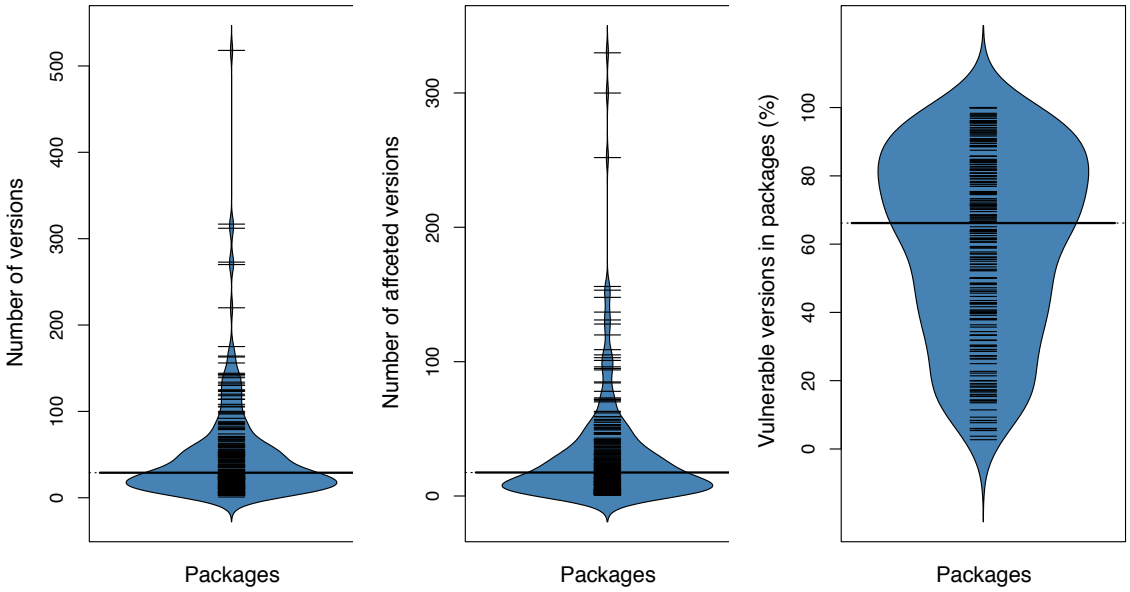


Figure 3.2: Introduction of vulnerabilities per year by the severity levels: high, medium, and low.

Figure 3.3 shows bean plots of three distributions: the number of versions of the 252 vulnerable PyPi packages in our dataset (Figure 3.3a), the number of affected versions in such vulnerable packages (Figure 3.3b), and the percentages of vulnerable versions in the packages (Figure 3.3c). We observe that most packages have dozens of versions (median number of versions is 29), and



(a) Number of versions. (b) Number of affected versions. (c) Percentage of affected versions.

Figure 3.3: Distribution of versions and affected versions of the 252 vulnerable packages of our dataset. In median, packages have 29 versions and 18 affected versions once a vulnerability is discovered.

tend to have, on median, 18 vulnerable versions. The affected versions represent an alarmingly high proportion of all versions in a package, considering the package versions available at discovery time of the vulnerability. Figure 3.3c shows that **half of the packages have at least 68% of their versions affected by a vulnerability, when a vulnerability is first discovered**. In 15% of the packages, the share of vulnerable versions can represent 90% of all released versions at the time the vulnerability was discovered. The result indicates that vulnerabilities are not limited to a few versions of a package, making it difficult for dependents to rollback to an unaffected version if a fix is not available at the time of the vulnerability discovery.

Since vulnerabilities can have different types (e.g., Buffer Overflow and SQL injection), we examine the different vulnerability types given by the Common Weakness Enumeration (CWE) that PyPi packages have. While we found that packages in the PyPi ecosystem are affected by 90 distinct CWEs, **the majority of the discovered vulnerabilities (65.82%) are concentrated on 5 main types**. Table 3.3 shows the distribution of the vulnerabilities over the 5 most commonly found CWEs. As we can see, **XSS is the most common CWE with 130 vulnerabilities**. Also, we observe that most of the XSS vulnerabilities are of medium severity. For the remaining CWEs,

Table 3.3: Ranking of the 5 most commonly found vulnerability types (CWE) in PyPi.

Rank	Vulnerability type (CWE)	Freq.	Frequency by severity		
			High	Medium	Low
1	Cross-Site-Scripting (XSS)	130	4	118	8
2	Denial of Service (DoS)	72	11	59	2
3	Arbitrary Code Execution	66	39	26	1
4	Information Exposure	60	8	44	8
5	Access Restriction Bypass	34	10	23	1

the proportion in each type varies from 72 vulnerabilities of type Denial of Service (DoS) to 34 of type Access Restriction Bypass CWE. Breaking down the proportions of vulnerabilities by severity shows that the majority of vulnerabilities from these types are of medium and high severity, indicating that they represent a serious threat to affected applications. This is particularly severe for the vulnerabilities of Arbitrary Code Execution type, where we found a higher frequency of high severity vulnerabilities than of medium and low severity levels combined.

Comparison to the npm ecosystem. The vulnerabilities found in npm (Decan, Mens, & Constantinou, 2018b) followed a similar distribution to our findings in the PyPi ecosystem. In npm, a) the new discovered vulnerabilities are increasing over the time, and the majority of those vulnerabilities are also of medium and high severity; b) such npm vulnerabilities are not limited to a few versions, i.e., 75% of vulnerable packages have more than 90% of their versions being affected by a vulnerability at the discovery time; c) XSS was also found to be the most common vulnerability among npm vulnerabilities (i.e., 105 occurrences out of 399 vulnerabilities overall).

The number of vulnerabilities is increasing over time in the PyPi ecosystem accompanying the growth of the ecosystem. Newly reported vulnerabilities tend to be of medium and high severity and affect the majority of versions of a software package. The majority of vulnerabilities are concentrated on five vulnerability types, with Cross-Site-Scripting (XSS) being the most common.

RQ_2: How long does it take to discover a vulnerability in the PyPi ecosystem?

Motivation. This question aims to investigate how long it takes to discover package vulnerabilities in the PyPi ecosystem. Answering this question is relevant since the longer a vulnerability remains undiscovered, the higher the chances it will be exploited by attackers. Also, since security maintainers need to discover vulnerabilities as soon as possible to mitigate the harmful impact, providing them with information regarding the life cycle of a vulnerability discovery is vital. Therefore, in this question, we study how long does it take to discover a vulnerability since it was first introduced in the package's source-code?

Approach. Our goal is to calculate the time required to discover a vulnerability in the PyPi ecosystem. To do so, we collect the discovery dates of all the vulnerabilities from the Snyk.io dataset. Then, we obtain the timestamps of the vulnerabilities introduction date from Libraries.io (as described in Section 3.2.2). Note that the vulnerability introduction date is the release date of the first affected version by the package vulnerability. We then calculate the time difference between the vulnerabilities discovery date and the vulnerabilities introduction date.

To gain more insight about the time it takes to discover the vulnerabilities, we conduct a survival analysis method (a.k.a. event history analysis) (Aalen, Borgan, & Gjessing, 2008). The survival analysis is a non-parametric statistic used to measure the survival function from lifetime data where the outcome variable is the "time until the occurrence of an event of interest". For example, it may be used to measure the time duration an employee remains unemployed after a job loss. In the context of our study, we are interested in the time it takes to discover a vulnerability. We use the non-parametric Kaplan-Meier estimator (Kaplan & Meier, 1958) to conduct the survival analysis, as used in previous studies (Decan & Mens, 2019; Decan, Mens, & Constantinou, 2018b).

Results. Figure 3.4 presents the survival probability for the vulnerability before it gets discovered. The Left-side plot of Figure 3.4 reveals that **the probability that a PyPi package vulnerability takes 37 months to be discovered is 50%**. In practice, this shows that vulnerabilities are not

discovered early in the project development. Also, this long process for discovering vulnerabilities explays why a single vulnerability tends to affect dozens of package versions once it is first discovered (RQ1).

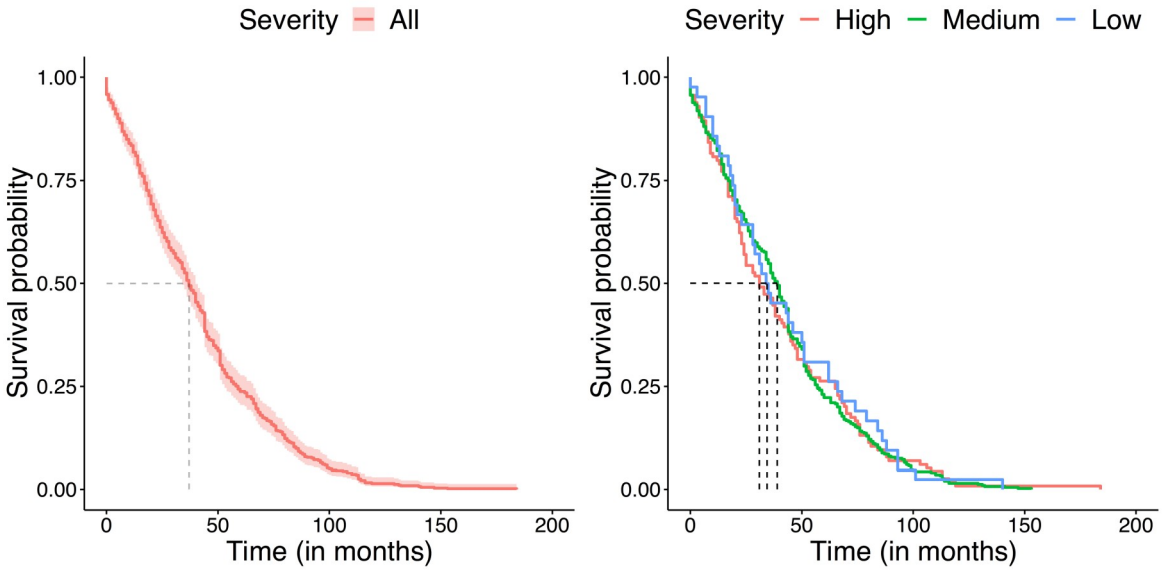


Figure 3.4: Kaplan-Meier survival probability for package vulnerabilities to get discovered for all vulnerabilities (left-side plot) and for vulnerabilities broken by severity (right-side plot).

Since vulnerabilities impact packages at different severity levels, we break down the analysis of discovered vulnerabilities by their severity. The right-side plot of Figure 3.4 presents the survival probability for the event "vulnerability is discovered" by their severity and depicts no significant differences among the severity levels. We confirm this result by using the log-rank statistical method (Bewick, Cheek, & Ball, 2004) to investigate the statistical significance of the results with a confidence level 95% (p-value = 0.94). **PyPi vulnerabilities take a substantial long time to be discovered and reported, independently of their severity.**

Comparison to the npm ecosystem. We found a significant difference on the time it takes to discover a vulnerability between the PyPi and npm packages. Vulnerabilities are discovered with a median of 24 months in the npm ecosystem, considerably sooner than the 37 months required for PyPi package vulnerabilities. Given the popularity of Javascript programs, npm became a prime target for attackers (Zimmermann et al., 2019a), which may have contributed to a faster identification of vulnerabilities. Overall, npm and PyPi vulnerabilities still take considerably long time to

discover vulnerabilities, indicating an issue in the process of testing and detecting vulnerabilities in open source packages.

Package vulnerabilities in the PyPi ecosystem take, on median, more than 3 years to get discovered, regardless of their severity.

RQ_3: When are vulnerabilities fixed in the PyPi ecosystem?

Motivation. Vulnerable packages remain affected even after they are discovered (Decan, Mens, & Constantinou, 2018b; Li & Paxson, 2017). In fact, in many cases, a method of exploitation is reported when the vulnerability is made public, which increases the chances of the vulnerability being exploited by attackers (Sabottke, Suciu, & Dumitra, 2015). Therefore, it is of paramount importance that developers release a fix of the package vulnerability quickly. In open-source ecosystems, a quick fix is the only weapon at developers disposal for minimising the risk of exploitation. Hence, in this question we provide package maintainers and users with information about at which stage a vulnerability fix is released in the PyPi ecosystem with respect to its discovery and publication date, i.e., we investigate whether a vulnerability fixed version is released before or after the vulnerability becomes publicly announced to better understand the threat level of PyPi package vulnerability.

Approach. Our goal is to study when vulnerabilities are fixed. To that aim, we categorise a vulnerability fix based on the stages of a vulnerability lifecycle. In other words, we analyse if the fix version was released before the vulnerability discovery time, in between the discovery time and publication time, or after the vulnerability is made public.

To achieve our goal, we obtain, for each vulnerability, the date of the first fixed version and compare it to the discovery and publication dates. The fix can then be categorized as: "before the vulnerability has been discovered" or short FixBeforeDisc, "between discovery and publication date" or short FixBetweenDiscPub, "after the vulnerability has been made public" or short FixAfterPub, and "Never Fixed". We then report the frequencies of fixes in each category.

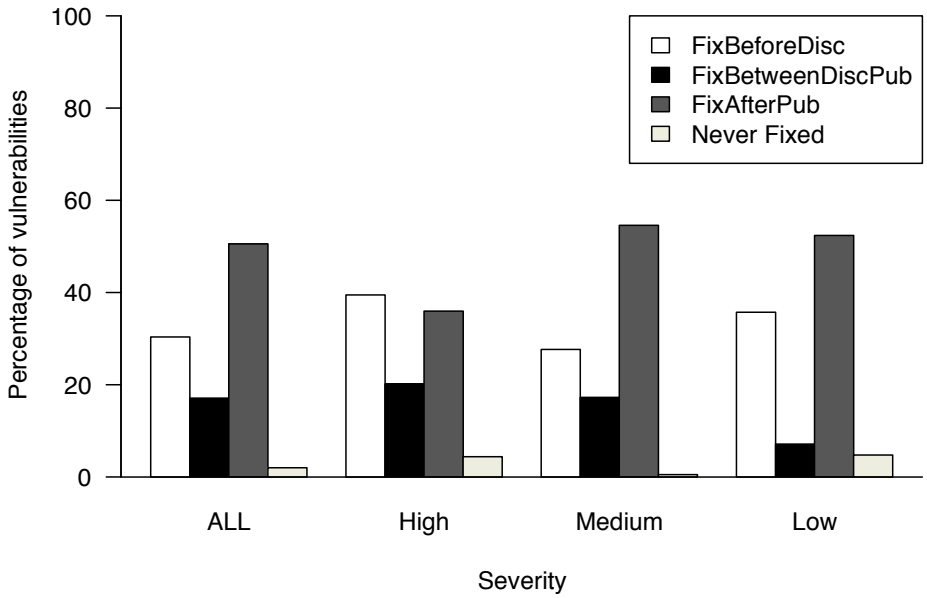


Figure 3.5: Percentages of vulnerabilities according to the release time of the first fixed version by severity.

Results. Figure 3.5 shows the distribution of vulnerabilities according to the four stages in which the first fixed version was released. **We can observe that 50.55% of vulnerabilities were fixed after the vulnerability has been made public**, with the observation being more noticeable for vulnerabilities of medium and low severity (H = 35.96%, M = 54.57%, and L = 52.38%). Such results indicate that the majority of the PyPi package vulnerabilities become public before having any patch addressed to fix them.

For the remaining vulnerabilities, 30.36% of all vulnerabilities were already fixed even before their discovery. One possible explanation is that the maintainers of such affected packages prefer to disclose the vulnerability and report its information while working in silence on a fix to mitigate its impact and reduce the chances of being exploited by potential attackers. Finally, 17.09% of the vulnerabilities were fixed between the vulnerability discovery date and the vulnerability publication date.

Comparison to the npm ecosystem. Unlike npm, our findings show that PyPi package vulnerabilities tend to be fixed only after publication. In npm, 82% of vulnerabilities are fixed after the vulnerability discovery time and before its publication time. Our findings for PyPi show a different

picture, with the close majority of vulnerabilities (50.55%) being fixed after their publication. Such differences can be attributed to community practices and policies in each ecosystem for reporting and disclosing vulnerabilities. We discuss these policies, their limitations, and how to better control them in Discussion.

The majority of vulnerabilities (50.55%) are only fixed after the vulnerability is made public, while 30.36% are fixed before the vulnerability is first discovered, and 17.09% are fixed between the discovery and publication dates.

RQ_4: How long does it take to fix a vulnerability in the PyPi ecosystem?

Motivation. So far, we have observed that the majority of vulnerabilities are fixed after the vulnerability is reported to be discovered, either in between discovery and publication (17.09%) or after the vulnerability publication (50.55%). In this question, we focus on those vulnerabilities and investigate how long it takes for a fix patch to be released after a vulnerability is reported to be discovered. Vulnerabilities that remain un-patched for a long time after being reported and discovered can leave an open channel for successful attacks. Also, a healthy open source package should have a quick response to most vulnerability reports. Therefore, answering this question will give us important insights about the prioritization of fixing vulnerabilities of a package.

Approach. To achieve our goal, we focus now on only those vulnerabilities that get fixed after being discovered, i.e., we omit vulnerabilities that have their fixed versions before the discovery date (30.36%). For the remaining vulnerabilities, we conduct the survival analysis method to provide information about how long it takes to fix a vulnerability after being discovered. We calculate the time difference between the release date of the first fixed version and the vulnerability discovery date. Similarly to the analysis conducted in Section 3.3, we use the Kaplan-Meier estimator (Kaplan & Meier, 1958) for the survival analysis. Furthermore, to understand if the severity level of a vulnerability has any impact on the time required to fix a vulnerability, we also conduct the previous analysis per severity level.

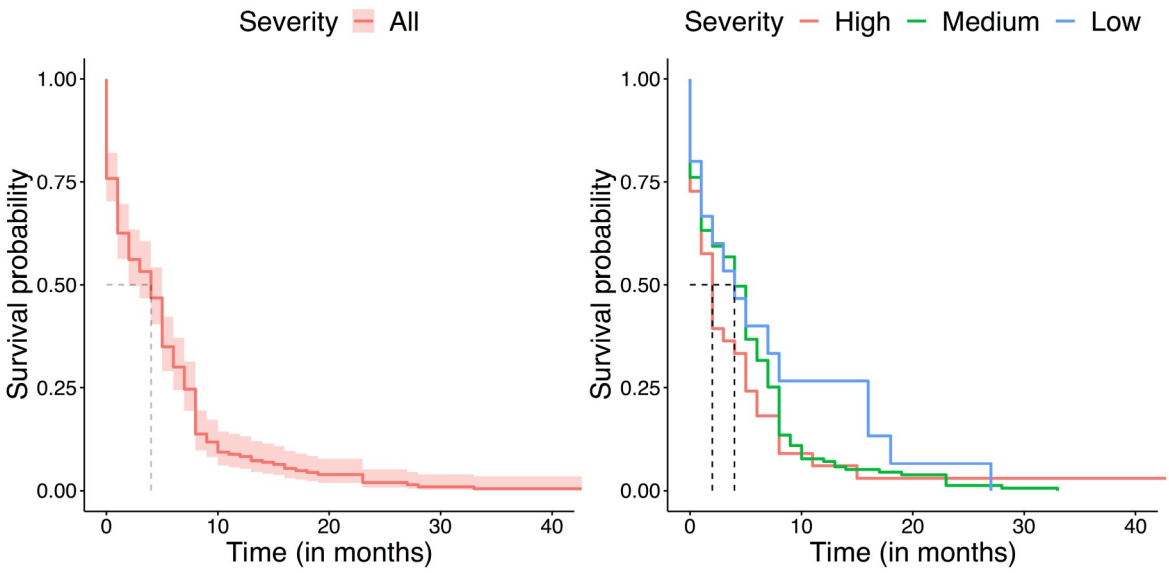


Figure 3.6: Kaplan-Meier survival probability for vulnerable packages to get fixed after being discovered.

Results. Figure 3.6 presents the survival probability for the vulnerabilities to be fixed after being discovered. **As we can observe from the left-side plot, the probability that a vulnerability is fixed 4 months following its discovery is 50%.** Also, we can observe that there is a small share (8.37%) of those vulnerabilities that still take more than a year to get fixed after being discovered.

The right-side plot of Figure 3.6 presents the previous analysis per severity level. Using the log-rank statistical method (Bewick et al., 2004), we found no statistically significant difference in the time to fix vulnerabilities of different severities with a confidence level 95% (p-value = 0.41).

We further analyse the vulnerable PyPi packages that took more than a year for their vulnerabilities to be fixed after the discovery date, to gain insights as to why they take such a long time to address potentially impactful vulnerabilities. Upon close manual inspection, we found that 64.7% of these packages are not popular (i.e., have less than 1000 downloads) and are not actively maintained, with the latest version been released two years ago, in 2018. We expect the developers of those vulnerable packages to be unresponsive to security reports.

Comparison to the npm ecosystem. Our findings show that PyPi package vulnerabilities overall take longer to be fixed than those found in npm. In npm, it takes a median of one month to fix vulnerabilities, regardless of their severity. In PyPi, we found that PyPi vulnerabilities take a median

of 4 months to release a fix after the vulnerability has been discovered.

Vulnerabilities in PyPi take, on median, 4 months to be fixed. The severity level of a PyPi vulnerability does not make a statistically significant difference for the time needed to fix the reported vulnerabilities.

3.4 Discussion and Implications

In this section, we discuss more details about our results with comparison to the npm ecosystem (Section 3.4.1). Then, we highlight the implications of our study to researchers and practitioners (Section 3.4.2).

3.4.1 Comparison to the npm ecosystem

As shown in our comparison to the npm, some of our findings generalized also to the npm ecosystem, while others did not. Therefore, in this section, we delve into some of the reasons both ecosystems exhibit some similar characteristics as well as explanations about the divergent findings.

Vulnerability distribution. Both studies found that the number of newly discovered vulnerabilities are growing over time. We attribute the reason for this increase to the increasing popularity of open source components combined with the awareness of vulnerabilities in such components (Williams & Dabirsiaghi, 2012). At first sight, this is a healthy sign of both ecosystems. The increase in the number of reported vulnerabilities is a result of coordinated efforts in increasing awareness and continuous process testing packages to identify the vulnerabilities before they can be exploited. However, the growth of the ecosystem calls also for the continuous and comprehensive effort for analysing package vulnerabilities to mitigate their negative impact.

We observed that the vast majority of the vulnerabilities identified in the npm and PyPi ecosystems are of medium severity.We believe that this observation is due to the fact that many of the tools used by security package maintainers to discover vulnerabilities in open source packages are not qualified to find more complex and critical issues although they are good at discovering new

vulnerabilities. Robust tools that combine exhaustive techniques like program analysis, testing, and verification are required to find high-hanging vulnerabilities (Godefroid, Levin, & Molnar, 2012).

We observed that Cross-Site-Scripting (XSS type or CWE-79), is the most common vulnerability found in both ecosystems. The dominance of the XSS CWE vulnerability can be justified by 1) its effectiveness in granting unauthorized access into a system and the ease in which the attack method can be applied on a web application (Johari & Sharma, 2012; Thom é, Shar, Bianculli, & Briand, 2018); and 2) the community efforts in taming this well-known vulnerability, as identifying XSS has been a top concern by OWASP (OWASP, 2019) for more than 15 years. We conjecture that other types of vulnerabilities might be not as easily detectable, or easy to exploit, taking away the incentive of attackers in searching such vulnerabilities in the PyPi and npm ecosystems.

Vulnerabilities discovery, publication, and fix. In npm, the majority of reported vulnerabilities (82%) were fixed after they were discovered and before the publication date (Decan, Mens, & Constantinou, 2018b). Contrasting to these findings, we found 50.55% of the PyPi package vulnerabilities to be fixed after the vulnerability has been made public. A possible reason for this is that 3 out of 4 vulnerabilities in PyPi get published right after their discovery, which reduces the time window for a vulnerability to be fixed.

To gain more insights, we investigate the protocol and policies in place for reporting and publishing vulnerabilities of both npm and PyPi ecosystems. We find that npm ecosystem has a protocol for reporting and publishing vulnerabilities, which enforces a 45 days waiting time before the publication of a vulnerability (*Reporting a vulnerability in an npm package*, accessed on 12/10/2021), aiming to give package developers a time window to fix the vulnerability. In PyPi, however, If a vulnerability is assessed to have low risk of being exploited or causing damage, the PyPi security team prefers to publish the vulnerability right after its discovery (PyPi, 2018). We noticed that most vulnerabilities (74.55%) are published as soon as they are discovered, effectively reducing the time window for a vulnerability to be fixed before publication.

An example of that, is a security issue that was found in *elementtree* package (*Issue 27863*, accessed on 12/10/2021). In this issue, the vulnerability could cause serious problems (high-severity level) through a Use-After-Free (UAF) (*CWE-416*, accessed on 12/10/2021) vulnerability related to incorrect use of dynamic memory, where the attacker causes the program to crash by accessing

the memory after it has been freed. Yet, the PyPi security team stated that in this specific case, an attacker could not exploit this vulnerability because it requires a privileged position that is not often possible from the attacker side. Such specificities and policies is a supportive reason behind having majority of vulnerabilities being fixed after the public disclosure. Note that the risk assessment conducted by the PyPi security team is different from the CVSS severity level assigned to a vulnerability in the Snyk.io dataset (*CVSS for CVEs — Snyk*, accessed on 12/10/2021).

3.4.2 Implications

In the following, we highlight the most important implications driven by our findings. We provide implications to both researchers and practitioners by discussing the aspects that the development community needs to address in order to provide a more secure development environment for package ecosystems.

There is a dire need for more effective process to detect vulnerabilities in open source packages. Our findings show that vulnerabilities in Python packages are hidden, on median, 3 years before being first discovered (RQ $_2$). These findings point to inadequacy process of testing open source packages against vulnerabilities. In fact, both npm and PyPi allows to publish a package release to the registry with no security checks exist before publishing the package. An open avenue for future research is the development of a process that ensures some basic security checks (code vetting) before publishing a release of a package. Inspired by other ecosystems, such as mobile application stores (*Android Google Play Protect*, accessed on 12/10/2021; Lu, Li, Wu, Lee, & Jiang, 2012), npm and PyPi could enforce some testing before publishing a new release of a package. Recently, there have been several research attempts to improve the security of the packages uploaded and distributed via the ecosystems, e.g., (Vu, Pashchenko, Massacci, Plate, & Sabetta, 2020a, 2020b) for PyPi, and Synode (Staicu, Pradel, & Livshits, 2016), NoRegrets (Mezzetti, Møller, & Torp, 2018) for npm. The vetting process can start with the most popular packages and move gradually, given the growth of the software ecosystems. Also, the code vetting process can focus on specific categories of security issues, e.g., malicious code or code that steals sensitive information from users, which is triggered by performing XSS attacks, the most common vulnerability found in npm and PyPi (RQ$_1$).

PyPi needs to employ a better protocol of publishing package vulnerabilities. The current process of disclosing and publishing a package vulnerability in Python seems to remain ad-hoc. Our findings show that over 50% of PyPi package vulnerabilities were unfixed when they were first publicly announced (RQ 3), and took a couple of months to be fixed and released (RQ 4). To better control the process of reporting and disclosing package vulnerability information and limit its leakage, practitioners should refine the process to balance the advantages from an early and public-disclosure process of a vulnerability versus private-disclosure process. A possible improvement could be by forestalling the vulnerability publication until valued package users and vendors are privately notified about the vulnerability to give them a little some time to prepare properly before the vulnerability is publicly disclosed. Such controlled process is adopted by various internet networking software packages like BIND 9 and DHCP (*Internet Systems Consortium*, accessed on 12/10/2021). The npm ecosystem defines to some extent a strict timeline for reporting a vulnerability providing only 45 days for package maintainers to fix their vulnerabilities before publishing them. Yet, its efficacy is not known.

PyPi should deprecate packages that suffer continuously from vulnerabilities. In our study, we observed that the vast majority of packages that take longer to fix vulnerabilities are due to project inactivity (RQ_4). A relatively new idea introduced by Pashchenko et al. (Pashchenko et al., 2018) is the concept of "halted package", which is a package where the time to release the latest version surpasses by a large margin the time maintainers took to release previous versions of the package. This concept can be used to identify packages that are becoming less maintained over time, and therefore, should be replaced by a better maintained alternative in the software ecosystem.

PyPi and npm should provide package users with vulnerability information to support them with the selection process of packages. Previous work (Larios-Vargas, Aniche, Treude, Bruntink, & Gousios, 2020) has studied several factors that influence the adoption of packages by developers. Researchers report that the occurrence of vulnerabilities and the number of vulnerabilities not quickly fixed in the packages are two important security-related factors. Currently, both npm and

PyPi package managers provide basic quality metrics on package popularity for each package such as, list of versions, downloads count, stars count, and number of open issues. However, they lack any information on security issues. A methodology, similar to the one used in our study, could be employed to define a lightweight security metrics, to support developers when selecting their packages. An example of such metrics is to calculate the average time to patch a package vulnerability after been reported to be discovered (RQ $_4$). This metric will give package users insights about the prioritization of fixing vulnerabilities of the package.

PyPi should employ tools to audit vulnerabilities when installing the packages. Our findings show that package vulnerabilities remain unfixed for a few months even after being publicly announced. Hence, Python applications that make use of such vulnerable packages could be exposed to vulnerabilities through their dependencies. Therefore, developers should be aware of vulnerabilities in their packages before installing them. Similar to the security audit tool provided by npm (i.e., `npm audit`) (*npm audit*, accessed on 12/10/2021), which warn developers when installing a known vulnerable package, PyPi community could employ a similar tool that instantly warns developers about vulnerable packages once the vulnerable package version is installed. Recently, GitHub acquired DependaBot tool (*Dependabot tool*, accessed on 12/10/2021), a tool that tracks vulnerabilities in several ecosystems. Researchers should work on evaluating such tools to understand their effectiveness and uncover their limitations.

3.5 Tool Support: Dep-Health

A critical issue of vulnerabilities that affect software packages is the lack of developers awareness to them (Kula et al., 2018). Developers should at least understand the security health of the adopted packages in their projects, i.e., how timely package maintainers discover and fix reported vulnerabilities. Moreover, a recent survey with developers indicated a high demand for high-level metrics to assess the maintainability and security of software packages (Pashchenko, Vu, & Massacci, 2020).

To address this problem, we build a tool called DEPHEALTH, which uses the approach described

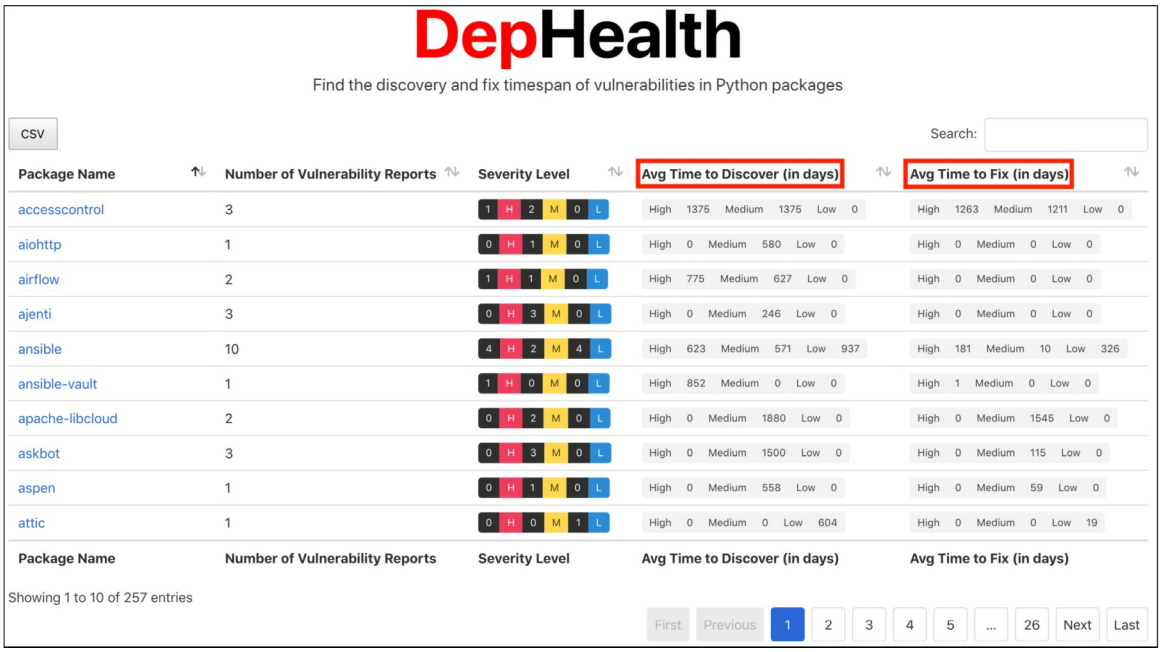


Figure 3.7: Screen-shot of the DEPHEALTH website showing its main page (*Dep Health — Home*, 2021). The columns' names that appeared inside the red-colored borders are the main metrics calculated for each vulnerable package.

in Section 3.2 to generate a report for security vulnerabilities that affect Python packages, i.e., we provide developers with metrics related to the life cycle of vulnerability discovery and fix, which help to show how maintained and secure the packages are.

Our tool generates two main metrics to help developers understand: 1) the discovery timespan of package vulnerabilities, and 2) the fix timespan of package vulnerabilities. The tool also presents the data of the metrics broken by the severity level of package vulnerabilities. Figure 3.7 shows a screen-shot of the DepHealth's interface. As shown in the figure, each row represents the relevant vulnerability data of a vulnerable package.

To facilitate using the tool, the tool provides the user with an option to search for a specific package to view its data, by using the search box in the top-right of Figure 3.7. Also, the user can download (using the button CSV) a complete version of the data presented in the website. It is also possible to download the data for selective packages, by pressing command key + a mouse click to select the desired package rows.

Moreover, the tool provides meta-data for the vulnerable package, e.g., number of total vulnerabilities in the package. By clicking on the package name, more details about each vulnerability

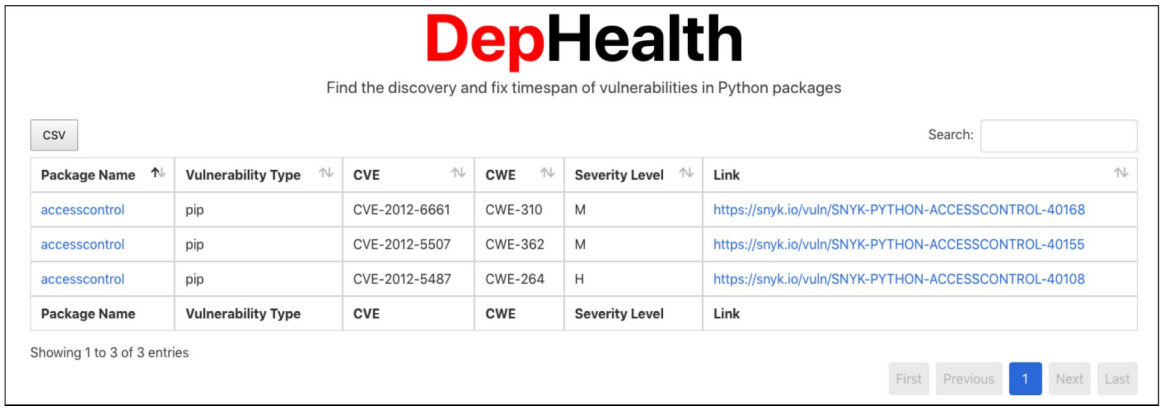


Figure 3.8: A Screen-shot showing some meta-data for vulnerabilities in the *Accesscontrol* package.

report can be shown, e.g., a reference to the report, as shown in Figure 3.8.

Finally, note that to avoid the out-of-date analysis, our pipeline for analysing the data is completely automated, which helps to easily update the website periodically.

3.6 Threats to Validity

In this section, we state some threats to validity that our study is subject to, as well as the actions we took to mitigate these threats.

Internal Validity The internal validity is related to the validity of the vulnerabilities dataset used in our analysis. Our dataset is restricted to a limited number of vulnerabilities (i.e., 550 security reports). We believe that many vulnerable packages may have been discovered and fixed but not yet reported. However, since Snyk team monitors more widely used packages (*How Snyk finds out about new vulnerabilities*, accessed on 12/10/2021), we expect our results to be representative of high-quality Python packages.

Also, we collect the PyPi vulnerabilities reports that were published before Jan. 2018. Results might differ if we consider vulnerability reports published after Jan. 2018. However, since a key point of our study is to compare our findings to the ones reported from npm, we collect our dataset in the same timeline as the npm study in order to make our study comparable. Furthermore, our vulnerability dataset contains more than 500 reports that cover 12 years of PyPi reported vulnerabilities, and many of these reports are related to a popular and most used Python packages

(e.g., Django, Flask, Requests).

Finally, in our analysis, we used the vulnerability severity level provided by Snyk.io to quantify their impact. However, the severity level published by Snyk.io is not necessarily uncontested, as discussed in Section 3.4.1, PyPi security advisories might have had different assessments on the severity of some vulnerabilities. Unfortunately, the severity analysis data provided by the PyPi ecosystem is not publicly available, therefore, we had to rely on the Snyk.io dataset as the only source of information for the severity of vulnerabilities. Also, vulnerability sources other than Snyk could be used, however, our choice of Snyk is influenced by several previous studies (Chinthanet et al., 2019; Decan, Mens, & Constantinou, 2018b; Zapata et al., 2018)

External Validity External validity concerns the generalization of our results to other software ecosystems and programming languages. As shown in our comparison to the npm, some of our findings generalized also to the npm ecosystem, while other findings did not. Although our methodology and approach could be applied to other software ecosystems, results might be (and unsurprisingly so) quite different from PyPi, due to characteristics such as policies, community practices, programming language features and other factors belonging to software ecosystems (Bogart et al., 2016; Decan et al., 2017). Therefore, a replication of our work using packages written in programming languages other than PyPi and npm is required to establish a more complete view of vulnerabilities in software ecosystems.

3.7 Related Work

We reviewed prior research on software packages and vulnerabilities in Chapter 2. In this section, we discuss the work that is most closely related to this chapter.

Hejderup et al. (Hejderup, 2015) conducted an empirical study on the impact of security vulnerabilities in the npm ecosystem. They analysed 19 npm vulnerable packages, and found that the number of vulnerabilities is growing over time. Neuhaus and Zimmermann (Neuhaus & Zimmermann, 2009) analysed vulnerabilities that affect RedHat packages and their dependencies and build a model to predict packages with vulnerabilities. Ruohomen (Ruohonen, 2018) conducted a release-based time series analysis for vulnerabilities in Python web applications, and found the

appearance probabilities of vulnerabilities in different versions of the applications followed the Markov model property. Also, Pashchenko et al. (Pashchenko, Vu, & Massacci, 2020) conducted interviews with developers of C/C++, Java, JavaScript, and Python to understand how they manage their packages with respect to security vulnerabilities. The results indicated a high demand for high-level metrics to show how maintained and secure the package is, when selecting an external package. Our study methodology (in Section 3.4.2) and the tool (in Section 3.5) can be employed to provide developers with such metrics for package selection process.

The work that is most close to our study is the npm study by Decan et al. (Decan, Mens, & Constantinou, 2018b). Their work focused on analysing vulnerability in the npm package ecosystem. Inspired by their study, and supported by the fact that different ecosystems have different characteristics, we conducted our study to examine security vulnerabilities in the PyPi ecosystem. We studied several aspects related to vulnerability propagation, their discovery and fix timeline. By comparing our findings with the ones reported by Decan et al. (Decan, Mens, & Constantinou, 2018b), we identified some particularities of the PyPi ecosystem and devise important recommendations to improve the safety of PyPi.

3.8 Chapter Summary

In this chapter, we conduct an empirical study of security vulnerabilities in the PyPi ecosystem, evaluating over than 500 package vulnerabilities that affect 252 packages. In particular, we explore vulnerabilities propagation, discovery, and fixes. Also, we compare our findings with the npm ecosystem (Decan, Mens, & Constantinou, 2018b). Our results show that PyPi vulnerabilities are increasing over time, affecting the large majority of package versions. Moreover, our findings reveal shortcomings in the process of discovering vulnerabilities in PyPi packages, i.e., they take more than 3 years to discover them. Additionally, we observe that the timing of vulnerability patches does not closely align with the public disclosure date, leaving open windows and chances for an attacker exploitation. We note that over 50% of the vulnerabilities were patched only after public disclosure. Finally, our comparison to npm vulnerabilities reveals in some aspects a departure from the npm's findings, which can be attributed to ecosystems policies. Finally, we build a tool called

DEPHEALTH, which uses the analysis approach in our study to generate analytical report of security vulnerabilities that affect Python packages, i.e., we provide developers with metrics related to the lifecycle of vulnerability discovery and fix, to help show how maintained and secure the packages are.

The focus of this chapter is to study security vulnerabilities that affect software packages. However, package vulnerabilities may propagate to dependent applications that rely on such packages, making them vulnerable too. In the next chapter, we shift our focus to examine software applications that rely on packages to study the degree that applications rely on vulnerable dependencies and understand how threatening such vulnerable dependencies really are.

Chapter 4

Examining the Discoverability of Package Vulnerabilities Impacting Software Applications

Many software applications depend on external packages. These packages can threaten the dependent software applications if exploited. Package vulnerabilities are common and remain hidden for years. However, once the vulnerability is discovered and publicly known to the community, the risk of exploitation reaches its peak, and developers have to work fast to remediate the problem. While there has been a lot of research to characterize vulnerabilities in software ecosystems, none have explored the problem taking the discoverability into account, i.e., how often and for how long are applications exposed to dependencies with public vulnerabilities? In this chapter, we focus on applications that use vulnerable packages and define three discoverability levels for package vulnerabilities impacting the application based on their disclosure lifecycle. Then, we conduct an empirical study on 6,546 open-source JavaScript applications to determine the prevalence of affected applications that rely on vulnerable dependencies taking into consideration the discoverability levels. We also study for how long these the application take before the vulnerabilities are mitigated. Our results show that 67.9% of the examined applications depend on at least one vulnerable package. Taking discoverability into account, we found that although the majority of

the affected applications (99.42%) depend on packages with hidden vulnerabilities at the time of analysis, 206 (4.63%) applications were still exposed to dependencies with public vulnerabilities. Moreover, we find that in the case of applications affected by public package vulnerabilities, it is the application's lack of updating that makes them vulnerable, i.e., it is not the existence of the vulnerability that is the real problem. Finally, we present DepReveal, a tool prototype to report the historical exposure of JavaScript projects to dependency vulnerabilities, to help developers better plan the maintenance of their software project.

4.1 Introduction

Modern software systems are developed with increasingly reliance on open source software packages (dependencies). This dependence on open source packages is highly beneficial to software development, as it speeds up development tasks and improves software quality (Basili, Briand, & Melo, 1996a; Lim, 1994a), but has implications to the security of those systems. Dependencies with security vulnerabilities have the potential to expose hundreds of applications to security breaches, causing huge financial and reputation damages. One such example is the Equifax incident (Equifax, 2017), where a vulnerability on a single dependency of Equifax (the Apache Struts package) led to unauthorized access to hundreds of millions of consumers' personal information and credit card numbers.

The recent popularity of software packages has only magnified the problem. For example, npm (the main package manager used by JavaScript applications) hosts more than 1.73M npm packages available for the JavaScript community. Prior studies (e.g. (Zimmermann et al., 2019a)) showed that a significant proportion (up to 40%) of all npm packages use code with at least one publicly known vulnerability, which increases the risk of a vulnerable package in a software application. In fact, an essential factor to evaluate the impact of vulnerable packages in an application is the *discoverability* of vulnerabilities, i.e, how publicly known is the package vulnerability (OWASP, 2020 (accessed 10/10/2020)). As an example, the vulnerability that caused the Heartbleed incident was hidden in the OpenSSL package for years (*Heartbleed Bug*, accessed on 12/10/2021), but once published, more than 4 thousand exploit attempts were registered by researchers (Durumeric et

al., 2014). While hidden (unknown) vulnerabilities can be exploited by attackers who are aware of the breach, once vulnerabilities become public, the chances of exploitation reach its peak and developers need to act fast to mitigate the security risks.

To our best knowledge, none of the previous studies has explored the problem taking the discoverability into account, considering the application development history. Hence, to shed light on this aspect and better understand the impact of a dependency vulnerability on an application, we examine the vulnerabilities based on their discoverability. To achieve this goal, we classify software vulnerabilities into three discoverability levels: *hidden*, indicating that a vulnerability that affects a dependency was not discovered (reported) yet at a specific point in the application lifetime; *reported*, indicating that a vulnerability was discovered and reported but not yet published; *public*, indicating that a vulnerability has been published and/or a proof-of-concept of how to exploit it is given. Note that this is a post-mortem classification, using information only available after the fact, for the purpose of evaluating dependency vulnerabilities impacting the applications.

We use our discoverability levels and perform an empirical study involving 6,546 active and mature open source JavaScript applications. First, to better understand the threat of dependencies on the software applications, we examine **(RQ1)** how the discoverability levels of vulnerable dependencies are distributed in the studied applications. Our findings show that although the majority (99.42%) of the affected applications (in one of their latest versions) are classified as having *hidden* dependency vulnerabilities, 4.63% of these applications depended on a *public* dependency vulnerability, where the discoverability is at its highest. This means that those applications depend on vulnerable versions of dependencies even after the vulnerability reports have been published.

Therefore, to better understand the reason for the existence of the threat due to the public dependency vulnerability (i.e., is it the application that did not update a dependency or is it the package that did not provide a fixing update), we examine the responsibility for the dependence on public vulnerabilities in **(RQ2)**. We find that the vast majority (90.8%) of the *public* dependency vulnerabilities were due to lack of dependency updates from applications, i.e., vulnerable dependencies had an available vulnerability fix (patch) but developers did not update their application to a newer (safer) version of the vulnerable dependency.

It is critical that applications patch public dependency vulnerabilities as soon as possible to

avoid potential exploits. Hence, to understand how fast vulnerable dependencies are patched in the applications, we examine **(RQ3)** how long does it take for public dependency vulnerabilities to be removed from the applications? We find that the applications take substantially long time (103 days) before *public* dependency vulnerabilities are fixed in the applications.

Finally, levering the findings in this study, we develop DEPREVEAL, a tool that informs developers of the historical exposition to public vulnerabilities in their JavaScript application. Our tool generates historical analytical reports to increase developers awareness to the discoverability of their JavaScript application dependencies.

In summary, this chapter makes the following main contributions:

- We define three discoverability levels for software dependency vulnerabilities based on their disclosure timeline.
- To the best of our knowledge, we conduct the first empirical study on 6,546 open-source JavaScript applications to determine the prevalence of affected applications that rely on vulnerable dependencies taking into consideration the discoverability levels. We also examine why these applications end up depending on vulnerable versions of the package in order to better understand how we can mitigate such issues.
- We develop DEPREVEAL, a tool that generates historical analytical reports to increase developers awareness to the discoverability of their JavaScript application dependencies.

Chapter organization.

The rest of this chapter is organized as follows: Section 4.2 describes how npm manages dependencies in JavaScript applications. Section 4.3 introduces our vulnerability classification used in this study. Section 4.4 describes our study design. Section 4.5 explains how we identify and classify vulnerable dependencies in JavaScript applications. Our results are presented in Section 4.6. Section 4.7 discusses our results further. Section 4.8 presents our tool. Section 4.9 discusses the implications of our findings. Section 4.10 presents the threats to validity. Section 4.11 discusses the related work. Section 4.12 concludes our study.

4.2 NPM Dependency Management

Since determining vulnerable dependencies in JavaScript applications heavily relies on the management of the dependencies and how they are resolved (i.e., the dependency constraints), in this section, we highlight how npm dependency management works.

Node Packages Manager (npm) is the de-facto package manager used by JavaScript applications to handle their dependencies (npm, accessed on 12/10/2021). npm has a registry where packages are published and maintained. To date, npm registry hosts more than 1.3M packages, and has had the highest growth rate in terms of packages amongst all known programming languages (*npm - Libraries.io*, 2021).

To determine the discoverability of vulnerable dependencies in JavaScript applications, we need to understand two important mechanisms of the npm ecosystem: 1) how JavaScript applications specify their npm dependencies and 2) how npm resolves a dependency version, i.e., find the dependency version to install in a JavaScript application. JavaScript applications specify their dependencies in a JSON-format file, called package.json, which lists the dependencies and their versioning constraints. The versioning constraint is a convention to specify the dependency version(s) of the package that an application is willing to depend upon. The version constraints can be static, requiring a specific version of the dependency (e.g., "P:1.0.0"), or dynamic specifying a range of versions of the dependency (e.g., "P:> 1.0.0"). Typically, developers use dynamic versioning constraints if they want to install the latest version of a dependency, allowing them to get the latest updates/security fixes of the package. When a dynamic version is used, the resolved version (i.e., the actual version) corresponds to the latest installable version that satisfies the constraint (Cogo, Oliva, & Hassan, 2019).

JavaScript applications can specify two sets of dependencies in their package.json file: development and production dependencies. Development dependencies are installed only on development environments, and consequently, issues that may arise from them (e.g., vulnerabilities and bugs) have no impact on production environments. On the other hand, production dependencies (also called runtime dependencies) are installed on both production and development environments. In our work, we only consider direct production dependencies since they are the ones that impact the

production environment (Decan, Mens, & Grosjean, 2018).

4.3 Classifying Vulnerabilities

In this section, we explain the stages of a vulnerability lifecycle and how that influences the levels of discoverability we investigate in our study.

4.3.1 Vulnerability Lifecycle

Typically, a vulnerability goes through a number of different stages (*A Day in the Life of npm Security*, accessed on 12/10/2021; *Disclosure of security vulnerabilities*, accessed on 12/10/2021).

- **Introduction.** This is when the software vulnerability is first introduced into the package code. At this stage, no one really knows about its existence, assuming that the introduction is not malicious.
- **Report.** When a vulnerability is discovered, it must be reported to the npm security team. The npm team investigates to ensure that the reported vulnerability is legitimate. At this stage, only the security team and the reporter of the vulnerability know about its existence.
- **Notification.** Once the reported vulnerability is confirmed, the security team triages the vulnerability and notifies the vulnerable package maintainers. At this stage, only the reporter, npm team, and package maintainers know about the vulnerability.
- **Publication without a known fix.** Once the package maintainers are notified, they have 45 days before npm publishes the vulnerability publicly. Alongside with publishing the vulnerability, the npm team may also publish a proof-of-concept showing how the vulnerability can be exploited (*npm publications*, accessed on 12/10/2021). At this stage, the vulnerability is known publicly and its potential risk is higher.
- **Publication with a fix.** Another (and more common) way that a vulnerability can be published is when a fix is provided by the package maintainers. If a fix is provided (before 45 days), then npm publishes the vulnerability along with the version of the package that fixes the vulnerability.

4.3.2 Discoverability Levels

The different stages of a vulnerability significantly impact its chance to be discovered by an attacker. Our study is based on the idea that vulnerabilities should be examined while taking their discoverability into consideration to better assess their potential for exploitation. We use the various stages to ground our argument and define three specific levels:

(1) ***Hidden*: before report.** Since very little (or nothing at all) is known about a vulnerability before it is found and reported, i.e., dependency vulnerabilities are hidden in the applications, we believe that the chances of being exploited are low. We classify all dependency vulnerabilities in the application at this stage as *hidden* dependency vulnerabilities.

(2) ***Reported*: after report & before publication.** Once a vulnerability has been discovered and reported, the general public is not yet aware of the vulnerability, as the process is conducted internally by the npm team. Still, there is a chance that others may know about the vulnerability and has the capability to exploit, so we consider the chances of exploit to be at a medium level. We classify dependency vulnerabilities in the application at this stage as *reported* dependency vulnerabilities.

(3) ***Public*: after publication.** After publication the chance of exploitability is at its highest. A proof-of-concept is often published (*npm publications*, accessed on 12/10/2021) alongside the vulnerability report, explaining how the vulnerability could be exploited. The threat of this vulnerability can only be mitigated once package maintainers release another version fixing the vulnerability and the application developers update their dependency accordingly. Failing to perform both these tasks in a timely fashion may put the application at higher security risk. We classify dependency vulnerabilities in the application at this stage as *public* dependency vulnerabilities.

4.4 Study Design

In this section, we describe the research questions (RQs) that drive our investigation and our process to collect a dataset of mature and active JavaScript applications for our study.

4.4.1 Research Questions

We leverage the collected data to answer the following research questions:

- RQ_1: How often JavaScript applications depend on vulnerable dependencies? How discoverable are their vulnerable dependencies?
- RQ_2: Who is responsible for the dependence on publicly known dependency vulnerabilities?
- RQ_3: For how long do applications depend on publicly known dependency vulnerabilities?

4.4.2 Data Collection

Our study examines vulnerable dependencies in JavaScript applications, particularly applications that use the Node Packages Manager (npm) as dependency management (npm, accessed on 12/10/2021). We opt to focus on JavaScript applications due to its popularity and importance in the current development landscape. JavaScript is currently the most popular programming language in the world (SOF, 2020) with a vibrant and fast growing ecosystem of reusable software packages (*npm - Libraries.io*, 2021).

To perform our study, we leverage two datasets: (1) JavaScript applications that use npm to manage their dependencies, and (2) Security vulnerabilities that affect npm packages. To do so, we **(i)** obtain the JavaScript applications from GitHub, **(ii)** extract their dependencies, and **(iii)** obtain the security vulnerabilities for npm packages from npm advisories (npm, 2021).

(i) Applications Dataset. To analyse a large number of open source JavaScript applications that depend on npm packages, we mine the GHTorrent dataset (Gousios, 2013) and extract information about all JavaScript applications hosted on GitHub. The GHTorrent dataset contains a total of 7,863,361 JavaScript projects hosted on GitHub, of which 2,289,130 use npm as their package management platform (i.e., these projects contain a file called package.json). Moreover, since both JavaScript *packages* and *applications* can use GitHub as their development repository, and our applications dataset should only contain JavaScript *applications*, we filter out the GitHub projects

Table 4.1: Statistics of the 6,546 studied JavaScript applications.

Metric	Min.	Median($\tilde{x}$)	Mean(μ)	Max.
Commits	100	326	1035.47	77,271
Dependencies	3	23	27.93	134
Developers	3	5	6.33	62
Age (in years)	5	7.24	7.53	12.81

that are actually npm *packages* by checking their GitHub URL on the npm registry. The main reason that we focused on applications and not packages is that packages become exploitable when used and deployed in an application. This filtering excludes 328,343 projects from our list of GitHub projects as they are identified as packages and not JavaScript applications.

As shown in previous studies (Kalliamvakou et al., 2014; Kula et al., 2018), projects in GitHub are not always representative of mature software projects we aim to investigate. Hence, we refined the dataset to focus on projects that are active and more likely to be mature software projects, by including applications that satisfy the following criteria:

- Non-forked applications, as we do not want to have duplicated project history to bias our analysis.
- Applications that depend on more than two dependencies.
- Applications that have at least 100 commits by more than two contributors, which indicates a minimal level of commit activity.
- Applications that have had their creation date (first commit) before January 1st 2017. Since vulnerabilities take on median 3 years to be discovered (Decan, Mens, & Constantinou, 2018b), applications in our dataset need to have a development history long enough to have had a chance for their vulnerabilities to be discovered.
- Applications that have at least one commit after January 1st 2020, as we want to analyze applications that had some level of development activity recently.

After applying these refinement criteria, we end up with 6,546 JavaScript applications that make use of npm packages. Table 4.1 shows the descriptive statistics on the selected JavaScript

applications in our dataset. Overall, the applications in our dataset have a rich development history (a median of 326 commits made by 5 developers and 7.24 years of development lifespan) and make ample use of external dependencies (a median of 23 dependencies).

(ii) Application Dependencies. After obtaining the applications dataset, we extract the history of dependency changes of all applications. This is necessary to identify the exact dependency versions that would be installed by the JavaScript application at any specific point-in-time. JavaScript applications specify their dependencies in a JSON-format file, called package.json, which contains the dependency list, a list of the depended upon packages and their respective version constraints. A version constraint is a configuration that specifies the dependency version(s) of the package that an application is willing to depend upon (*Semantic Versioning 2.0.0*, 2021). Hence, we extract all changes that touched the package.json file and associate each commit hash and commit date to their respective package.json dependency list, creating a history of dependency changes for all applications. Note that these dependencies are not yet resolved, that is, we only have the version constraints (not the versions) for the dependencies of each application.

(iii) npm Advisories Dataset. To identify the JavaScript applications that depend on vulnerable packages, we need to collect information on npm vulnerable packages. We resort to the *npm advisories* registry to obtain the required information about all npm vulnerable packages (npm, 2021). The npm advisories dataset is the official registry for all vulnerability reports related to JavaScript packages. This dataset provides some key information on vulnerable packages, such as the affected package, the affected package versions, and the first version in which the vulnerability has been fixed (safe version), if available. This dataset also contains the vulnerability discovery (report) time and publication time, which we use in our approach for identifying and classifying vulnerabilities (Section 4.5).

Our initial dataset contains 1,456 security reports that cover 1,234 vulnerable packages. Following the criteria filtration process applied by Decan et al. (Decan, Mens, & Constantinou, 2018b), we removed 312 vulnerable packages of the type “Malicious Package”, because they do not actually introduce vulnerable code. These vulnerabilities are packages with names close to popular packages

Table 4.2: Descriptive statistics on the npm advisories dataset.

Vulnerability reports	1,144
Vulnerable packages	925
Versions of vulnerable packages	38,562
Affected versions by vulnerability	20,206

(a.k.a. typo-squatting) in an attempt to deceive users at installing harmful packages. The 312 vulnerable packages account for 312 vulnerability reports. At the end of this filtering process, we are left with 1,144 security vulnerabilities reports affecting 925 distinct vulnerable packages. These packages have combined 38,562 distinct package versions of which 20,206 are affected by vulnerabilities from our report. The collected advisories dataset covers vulnerability reports created between October 2015 and May 2020. Table 4.2 shows the summary statistics for vulnerability

and how we use these levels to classify JavaScript applications.

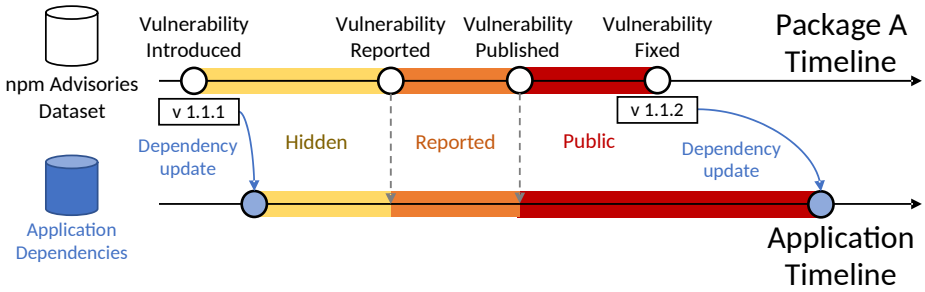


Figure 4.1: Illustration of the methodology for classifying the discoverability level of a single vulnerable dependency (Package A) for an application.

We illustrate our methodology in 4.1, on an example of an application with a single vulnerable dependency. As we can observe, the timelines of the discoverability levels of both the vulnerable package and the application are different. In the example of 4.1, the application is only affected by a vulnerable dependency once it starts depending on the first vulnerable version (v 1.1.1). Similarly, even if the package latest release contains a fix to the vulnerability, the application can

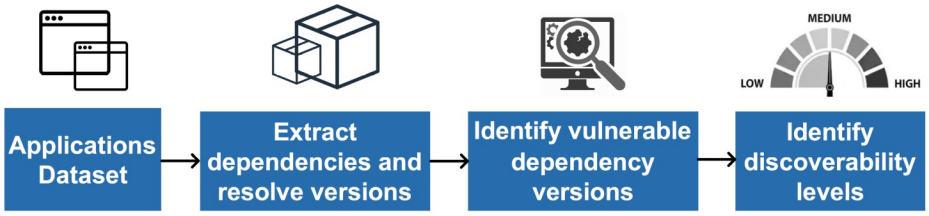


Figure 4.2: Approach for identifying and classifying JavaScript applications affected by vulnerable dependencies.

only benefit from it once it updates to the fixed version (v 1.1.2). This is different for the changes of discoverability levels once the vulnerability is made public. Due to the open nature of open source software, as soon as a vulnerability is published, any attacker in potential can identify that the application depends on the vulnerable version of Package A.

The goal of our study is to investigate how often JavaScript applications depend on vulnerabilities that are hidden, reported and public. To make our analysis feasible, we focus on classifying applications at one specific point in time of the application development history, which we call the *analyzed snapshot time*. We accomplish this by leveraging a 3-step approach. Figure 4.2 provides an overview of our approach, which we detail below:

Step 1. Extract dependencies and resolve versions. The goal of this step is to extract the application dependencies and find the dependency version that would be installed at the analyzed snapshot time. For each application, we extract the dependency list (with the versioning constraints) at that snapshot time from the history of dependency changes. After that, to find the actual version of each dependency at the analyzed snapshot, we utilize the *semver* tool(semver, 2021). This tool is used by npm to resolve versioning constraint in JavaScript applications. We included one additional restriction to *semver*, that the satisfying version should have been released (in the npm registry) before the analyzed snapshot time. For example, an application can specify a versioning constraint ("P:> 1.0.0") at the snapshot May 1st 2016. Hence, the actual installed version is the latest version that is greater than 1.0.0 and also has been released in the npm registry before May 1st 2016. This step allows us to find the installed version of the dependency at the analyzed snapshot time.

Step 2. Identify vulnerable dependency versions. After determining the resolved (and presumably installed) version at the analyzed snapshot time, we check whether the resolved version is vulnerably or not. To do so, we cross-reference the resolved versions with the advisories dataset. If the resolved version is covered by the advisories dataset, we label it as a vulnerable dependency version. We skip the whole next step if the dependency version has not been mentioned in any advisory, i.e., the dependency version is not known to be vulnerable.

Step 3. Identify discoverability levels of vulnerable versions. Once we identify the vulnerable dependency versions at the analyzed snapshot time, we classify each vulnerable dependency version using one of the discoverability levels we defined in Section 4.3.2. To that aim, for each vulnerable version, we compare its vulnerability *discovery (report)* and *publication* time to the analyzed snapshot time. As we stated previously (in Section 4.3.2), if the vulnerability was made public before the snapshot time, we mark the dependency version as having a *public* vulnerability. If the vulnerability of the dependency was not published but only discovered (reported) before the application's snapshot time, the vulnerable dependency version is considered to have a *reported* vulnerability. And finally, if the vulnerability was neither published nor discovered (reported) before the analyzed snapshot time, then we classify the dependency version as a *hidden* vulnerability. In cases where more than one vulnerability affects the vulnerable dependency version, we label the vulnerable dependency version with the highest level. For example, if we find that the vulnerable version of the dependency is affected by two vulnerabilities, one classified as hidden and the other classified as public, we label the dependency version as having a public vulnerability, at that snapshot time.

4.6 Study Results

In this section, we present the motivation, the approach and the findings that answers our 3 research questions (RQs).

RQ_1: How often JavaScript applications depend on vulnerable dependencies? How discoverable are their vulnerable dependencies?

Motivation: Previous studies have shown that security vulnerabilities are very common in the npm ecosystem, with nearly 40% of all npm packages relying on code with known vulnerabilities (Zimmermann, Staicu, Tenny, & Pradel, 2019b). However, vulnerable dependencies can only be exploited once deployed in applications: how many of our studied applications depend on vulnerable dependencies? Moreover, given that the discoverability is essential in assessing the threat of a security vulnerability (OWASP, 2020 (accessed 10/10/2020)), we want to quantify how many studied JavaScript applications depend on hidden (low risk), reported (medium risk), and public vulnerabilities (high risk), at the analyzed time. Answering these questions will give us a better assessment on the exposure of JavaScript applications to dependency vulnerabilities.

Approach: To reduce the biases in our analysis, we need to account for the time it takes to discover a vulnerability. Prior work showed that vulnerabilities in npm packages take on median 3 years to be discovered and publicly announced (Decan, Mens, & Constantinou, 2018b). Consequently, selecting snapshots of our applications in 2021 will paint an incomplete picture, as most vulnerabilities recently introduced in the package's code could remain hidden for a median of 3 years. Since our collected applications contain their latest commits between Jan 2020 and May 2020, we chose to evaluate our applications as of May 1st 2016 (more than 3 years prior), which ensures that at least half the dependency vulnerabilities introduced in the applications are reported in the current npm advisories dataset.

Then, we answer our RQ in two steps. In the first step, we examine if the *selected snapshot* of the application had at least one dependency that contains a vulnerability (irrespective of its discoverability level). In the second step, we analyze only the applications containing at least one vulnerable dependency and use the methodology described in Section 4.5 to classify the discoverability levels of all vulnerable dependencies. In addition, since some applications have more than one vulnerable dependency, we further analyze the distribution of vulnerable dependencies in the applications under each discoverability level.

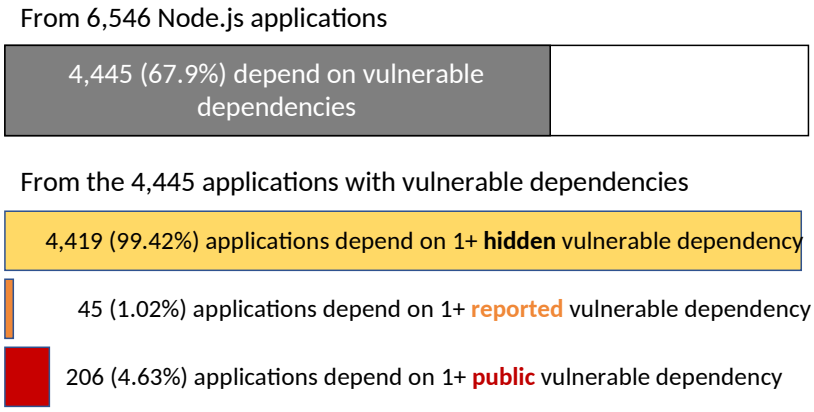


Figure 4.3: Bar-plots showing the share of the examined applications with one or more (1+) vulnerable dependency, overall and per discoverability levels.

Results: As shown in Figure 4.3, we found that of the 6,546 studied applications **4,445 (67.90%) applications depend on at least one vulnerable dependency**. From the 4,445 affected applications, we break down the dependency vulnerabilities by the discoverability levels and evaluate how many applications contain one or more hidden, reported and public dependency vulnerabilities. We show this break down also in Figure 4.3. Note that the total percentage of hidden, reported and public surpasses 100%, as one application might contain dependency vulnerabilities on different discoverability levels. We observe that the majority of the affected applications, 4,419 (99.42%), depend on one or more dependency vulnerabilities that were hidden at the analyzed snapshot time. In fact, on 94.26% of the cases (4,190 applications), the applications were affected only by hidden vulnerabilities. Still, **206 (4.63%) applications depended on at least one package with a public vulnerability** and 45 (1.02%) applications depended on packages with a vulnerability reported to package maintainers.

Given that applications may have multiple vulnerable dependencies, we analyze proportion of vulnerable dependencies in each application. Figure 4.4 shows the distribution of the percentage of vulnerable dependencies per application in each discoverability level (public, reported, hidden). For instance, if an application has 10 dependencies, of which only 2 contained public vulnerabilities, this application would have 20% of its dependencies affected by public vulnerabilities.

In terms of **public** vulnerabilities, the 206 applications with at least one public dependency

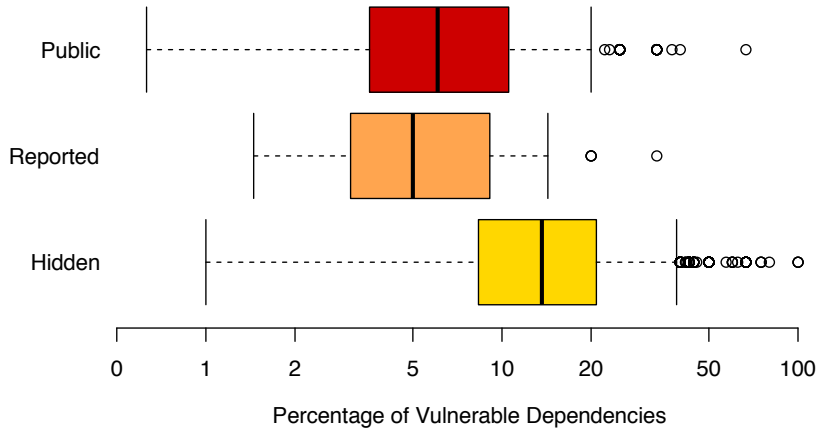


Figure 4.4: Box-plots showing the distributions of the percentages of vulnerable dependencies in the applications, per discoverability level.

vulnerability had, on median, 6.25% of their dependencies affected by a public vulnerability, or, 1 out of 16 dependencies. The majority (80.1%) of the 206 applications depend on a single vulnerable dependency with a public vulnerability. For example, one of the applications affected by a public dependency vulnerability is the project *Atom*, a popular text editor, which has more than 40 dependencies but it was affected by a public vulnerability on a package called *marked* (*marked@v0.3.4*, 2020 accessed on 12/10/2021). Upon close inspection we found that, while the 206 applications depended on a total of 2,438 different packages, the public dependency vulnerabilities were found in only 17 packages. That is, the public dependency vulnerabilities occurred in less than 1% of total dependencies, but could, nevertheless represent the highest threat of exploitation on those applications. Several of these packages are very popular dependencies in the npm ecosystem, such as semver, express, moment, sequelize, marked, tar.

Also, from Figure 4.4 we can observe that **reported** vulnerabilities are present in only 45 applications (1% of the affected applications). The median rate of dependencies with reported vulnerabilities in these 45 applications is 5.5% (1 out of 18 dependencies). It is notable that we find such a small share of applications that depend on reported dependency vulnerabilities. This is attributed to the npm policy for managing vulnerabilities: the policy states that the reported period of a vulnerability lasts at most 45 days, i.e., the vulnerability is published after 45 days of being reported to maintainers (*AppSec on Dependency Management*, 2020 (accessed 2020)). This limits how long a vulnerability can remain reported, thus, explaining the small occurrence of

vulnerabilities at this stage.

Finally, Figure 4.4 shows that half of the 4,419 applications had at least 13.63% of their dependencies affected by **hidden** vulnerabilities. That is, on median, 3 out of 22 dependencies are affected by a hidden vulnerability that would be reported and published after May 2016.

Our findings show that 67.9% of the studied applications depend on vulnerable packages. The majority (94.26%) depended only on hidden dependency vulnerabilities. Still, 206 applications (4.63%) depended on packages with public vulnerabilities.

RQ_2: Who is responsible for the dependence on publicly known dependency vulnerabilities?

Motivation: While most of the affected applications depend on dependencies with hidden vulnerabilities at the studied snapshot time (RQ_1), there is a sizeable number of the affected applications (206 applications = 4.63%) that depend on packages with public vulnerabilities.

In such cases, the developers of the applications could know about the presence of the vulnerability in the affected dependency, and hence, should avoid using that vulnerable version, if a fix is available. Specifically, in this question we want to know who is to blame - the package maintainers for not providing a version that fixes a public vulnerability - or the application maintainers for not keeping their applications up-to-date. Answering this will help us pinpoint the causes for public vulnerabilities in JavaScript applications and develop further strategies to solve this problem.

Approach: To perform our investigation and answer who is responsible for the public vulnerabilities in applications, we check - for each vulnerable package - the availability of a safe (non-vulnerable) version of the package at the analyzed snapshot time. Note that we analyze this RQ at the same snapshot that we analyzed in RQ_1 (i.e., May 2016). Depending on such availability, our analysis has one of two outcomes:

- **Package-to-blame:** if at the analyzed snapshot, no safe version has been provided by the

Table 4.3: The percentage of vulnerabilities caused by the lack of available fix patch (Package-to-blame) vs. caused by the lack of dependencies update (Application-to-blame).

Snapshot	Package-to-blame	Application-to-blame
1st May 2016	**9.24%**	**90.76%**

package maintainers for the public vulnerability. As the publication of a vulnerability comes after a period of 45 days, we consider the package maintainers the responsible for the dependency public vulnerability in applications.

- **Application-to-blame:** if there is already a released safe version of the vulnerable package but the application continues to rely on an (old) version with a public vulnerability. Application developers should monitor their dependencies and update to releases without public vulnerabilities, hence, we consider the application maintainers responsible for depending on a vulnerable package version.

Results: Table 4.3 shows the percentage of public vulnerabilities based on our responsibility analysis. We observe that **for public dependency vulnerabilities, the application is to blame in 90.76% of the cases**. That means that in 9 out of 10 cases the public vulnerability had an available fix, but developers did not update their application dependencies accordingly to receive the latest fix patch.

Therefore, and perhaps counter-intuitively, applications are not exposed to public vulnerabilities because packages have unfixed vulnerabilities. Instead, the real cause is the fact that application developers fail to keep up or at least to inform themselves well enough about a given dependency version. Hence, a major implication of our study is that application developers struggle with keeping their dependencies up-to-date, which may have serious effects in the security of their systems.

To have a better understanding of our results, we investigate how much effort would developers need to migrate to a safe version of their packages. npm adopts a semantic version scheme (semver, 2021) where package maintainers are encouraged to specify the extent of their updates in three different levels: 1) patch release, which indicates backward compatible bug fixes, 2) minor release, which indicates backward compatible updates and 3) major release, which informs developers of

backwards incompatible changes in the package release. Hence, patch and minor updates are deemed backwards compatible and may be performed at a lower migration cost, while major release updates incur on a high migration cost, as developers have to adapt their code to the new package API.

Once we take the update levels into consideration, we found that, **in 43.07% of the public vulnerabilities, the fix is only available in another major release of the package**. For instance, an application depends on P:1.0.0, and the fix patch was only released for a major version 2.0.0. Hence, to benefit from a fix patch in such a case, developers are required to adapt their code, imposing significant migration costs, especially for large projects that depend on dozens of packages. Furthermore, this shows that relying on automatic updates at the level of patch and minor releases (as recommended by npm (*Semantic Versioning 2.0.0*, (accessed 2020))) does not completely prevent public vulnerabilities for affecting JavaScript applications.

In 9 out of 10 cases, the main cause of dependence on packages with public vulnerabilities is the lack of dependency updates. However, in 43% of the cases, the fix is only available on another major version of the package, incurring in significant migration costs for application developers.

RQ_3: For how long do applications depend on publicly known dependency vulnerabilities?

Motivation: Previous RQs show that a small but significant number of applications are exposed to public dependency vulnerabilities, mostly due to lack of dependency updates. It is not clear, however, for how long the applications remain affected by a public dependency vulnerability. Public dependency vulnerabilities that affect the dependent application for a long time can leave an open channel for successful attacks, as shown in cases such as the Heartbleed incident (Durumeric et al., 2014). Hence, we investigate how long applications remain depending on a package version affected by a public vulnerability. Answering this question will give us insights about the prioritization of

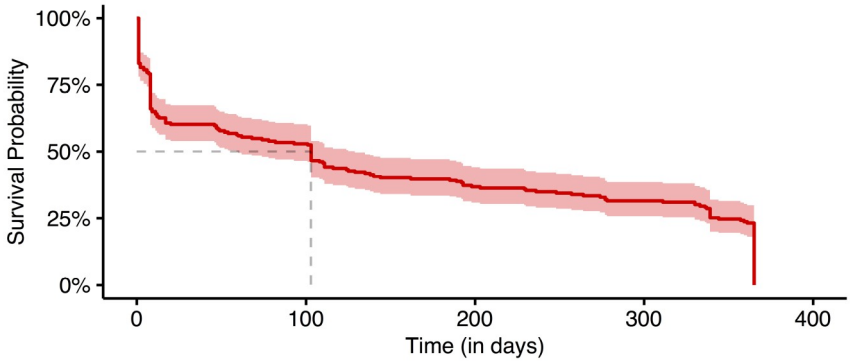


Figure 4.5: Kaplan-Meier survival probability for affected applications with a publicly known vulnerability.

patching public dependency vulnerabilities that affect an application.

Approach: We continue to focus our analysis on the 206 applications that depend on public dependency vulnerabilities. Then, for each application, we measure the time period (in days) of which the application remained affected by a public dependency vulnerability. We constrain our analysis to one year, from Jan 1st, 2016 to the December 31st, 2016, to have an easy to interpret and comparable time-frame. Note that an application could have been affected by different public vulnerabilities in different segments, e.g., from 1st May 2016 to 1st June 2016 and then from 1st Sept. 2016 to 1st December 2016. In such cases, we sum all such periods (i.e., add up the number of days).

To present this analysis, we conduct a survival analysis method (a.k.a. event history analysis) (Aalen et al., 2008). The survival analysis is a non-parametric statistic method used to measure the survival function from lifetime data where the outcome variable is the time until the occurrence of an event of interest. In the context of our study, we are interested in the time period that an application remains (survives) depending on a public dependency vulnerability. We use the non-parametric Kaplan-Meier estimator (Kaplan & Meier, 1958) to conduct the survival analysis, as used in previous studies (Decan & Mens, 2019; Decan, Mens, & Constantinou, 2018b).

Results: Figure 4.5 presents the survival probability for the applications depending on a public dependency vulnerability in the year of 2016. As we can observe, **half the applications remain**

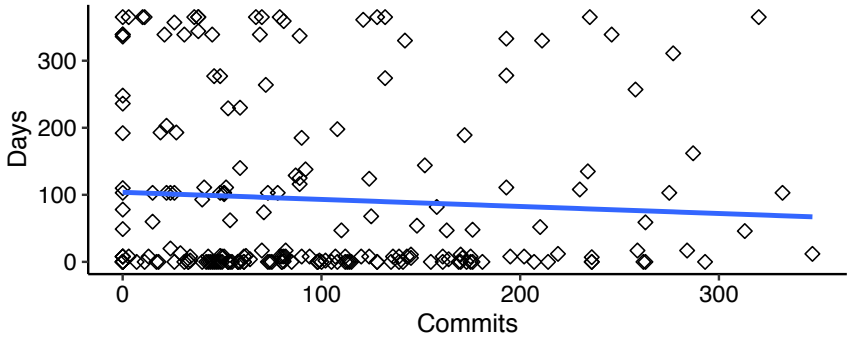


Figure 4.6: Scatter plot showing the correlation analysis of number of commits vs. number of days.

exposed to a public dependency vulnerability for at least 103 days. Such a long exposure of applications is deeply concerning, as it gives a considerable time window for attackers' exploitation.

One possible reason that such applications had not fixed the public vulnerability for a long time is that those applications are not actively being developed during the year of 2016. To investigate this possibility, we examine the development activity of applications during the period that they remained affected by public vulnerabilities. To do so, we measure the application activity by counting the number of commits an application had during the time being affected by a public vulnerability. For example, if an application was affected by a public dependency vulnerability between May 1st 2016 and August 15th 2016, then we calculate the total number of commits within that period. Then, we plot both the number of commits and the number of days during which an application had been affected by a public vulnerability.

Figure 4.6 shows the scatter plot of both variables number of commits vs. number of days (affected by a public vulnerability). We draw a trendline in Figure 4.6 in order to study the relationship between the variables. We can observe that there is no clear pattern of the dots, indicating no correlation between the application activity and the duration of which an application had been affected by a public dependency vulnerability (Pearson corr = -0.066).

In a period of a year, half of the applications with public dependency vulnerabilities remain exposed for a long time (103 days) before vulnerabilities are removed from the applications.

Table 4.4: The share of applications with one or more (1+) public dependency vulnerabilities per severity levels.

Severity Levels	Affected Applications
Low	172 (83.49%)
Medium	167 (81.06%)
High	140 (67.96%)
Critical	59 (28.64%)

4.7 Discussion

In this section, we discuss our results further by reflecting two aspects: the severity of public dependency vulnerabilities and the evolution of discoverability levels in the studied applications.

4.7.1 Severity levels of public vulnerabilities

As we observed in all our RQs, around 5% of the affected applications depend on *public* dependency vulnerabilities at a specific point in time, however, what is the severity of these vulnerabilities? Our study is centered on the discoverability of vulnerabilities in software dependencies, that is, their potential for being exploited. However, a public vulnerability can have a high chance of exploitation according to our classification but cause a low impact if exploited (low severity level). Hence, we discuss the severity of the public vulnerabilities to better understand the potential impact of these cases.

The npm advisories associates each package vulnerability report with its severity level (npm advisories, (accessed 2020)). Severity level has four possible levels, Low, Medium, High, and Critical, which are assigned manually by the npm team. Vulnerabilities clasified as High or Critical are considered of high impact and need to be addressed immediately by software maintainers (npm advisories, (accessed 2020)). By cross-referencing our dataset with the severity reports, we report in Table 4.4 the distribution of the severity levels of the 206 application with public dependency vulnerabilities. Once again, the total percentage of Low, Moderate, High and Critical surpasses 100%, as some applications contain multiple dependency vulnerabilities on different severity levels. As shown in Table 4.4, of the 206 applications that are affected by public dependency vulnerabilities, 172 (83.49%) applications are affected by at least one vulnerable dependency of *low severity*. Still,

Table 4.5: The percentage of vulnerable applications at different historical snapshots, per discoverability level.

Application Snapshot	Affected Applications	Applications		
		Hidden	Reported	Public
20%	4,215 (64.39%)	4,202	16	101
40%	4,277 (65.34%)	4,261	27	122
60%	4,372 (66.79%)	4,352	32	142
80%	4,421 (67.54%)	4,398	41	171
100%	4,445 (67.90%)	4,419	45	206

a majority of 140 (67.96%) applications are affected by public vulnerabilities classified as high severity. In 59 (28.64%) applications, the public vulnerability was classified as critical, given their potential for exploitation. These results dismiss the idea that applications only depend on public dependency vulnerability with low impact of exploitability. More than 140 applications had dependencies with public vulnerabilities where analysts classified them as of high and critical impact, a dangerous combination for the health of those software projects.

4.7.2 Project evolution vs. discoverability levels

Our study thus far has been conducted on one snapshot of the examined applications (RQ $_1$). However, our results might change if the study would be performed at different stages of a projects development cycle. We would like to determine whether our results generalize to different historical snapshots of the application development. Hence, we investigate the evolution of discoverability levels across different snapshots of applications' development.

Since each application has different lifespans, we want to find a measure that makes comparing them feasible. To do so, we normalize the applications by segmenting the lifetime of each application into five equal intervals (each containing 20% of an application's lifetime by time in days). Then, we perform the same analysis conducted in RQ $_1$ on the last snapshot of these five intervals. For this analysis, we only consider the 4,445 applications with at least one vulnerable dependency, as identified in RQ$_1$.

Table 4.5 shows the percentage of applications that have at least one vulnerable dependency for the 5 analyzed snapshots across their lifetime, along with the distribution of their discoverability

levels. We recall that the snapshot of 100% represents applications analyzed on May 2016, the same snapshot analyzed in RQ1. Overall, we found that the proportion of applications with one vulnerable package remain steady between 64 to 67% of the analyzed applications. The major findings in RQ1 holds for all snapshots, there is a predominance of applications with hidden vulnerabilities, followed by a small share with public and even smaller share with reported dependency vulnerabilities. While the number of affected applications have increased as the applications evolve, it is noteworthy that the number of applications exposed to public vulnerabilities more than doubled since the snapshot 20% (from 101 to 206 applications). To conclude, our complementary analysis shows that the trends observed in RQ_1 hold at different stages of the projects.

4.8 Tool Support: Dep-Reveal

A major problem of vulnerabilities in software dependencies is the lack of developers awareness to security vulnerabilities in their dependencies (Kula et al., 2018). Developers need better tools to help them identify the occurrence of vulnerabilities and how timely package maintainers respond to reported vulnerabilities, which affects the discoverability we studied in this work. To address this problem, we build a tool called DEPREVEAL , which uses the approach described in Figure 4.5 to generate analytical reports of dependency discoverability levels for a GitHub JavaScript project. DEPREVEAL is open-source, publicly available, and can be easily integrated in any GitHub npm project.

Our tool generates 4 different reports to help developers understand: 1) the discoverability level of dependency vulnerabilities, 2) the frequency of dependency vulnerabilities per discoverability level, 3) the period of package exposition to discoverability levels, and 4) what package versions account for those dependency vulnerabilities. Figure 4.7 shows a screen-shot of the DepReveal's interface.

Next, we explain 2 of the most insightful reports generated by our tool. Inspired by the Github's Contributions Activity Graph, DEPREVEAL generates a **Dependency Discoverability Graph**, which

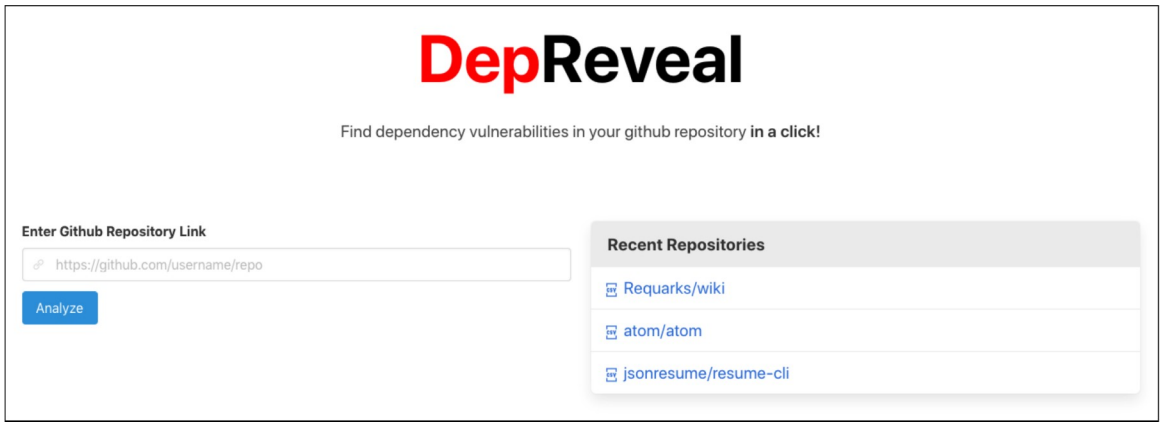


Figure 4.7: Screen-shot of the DepReveal website showing its interface and the recently analysed repositories.

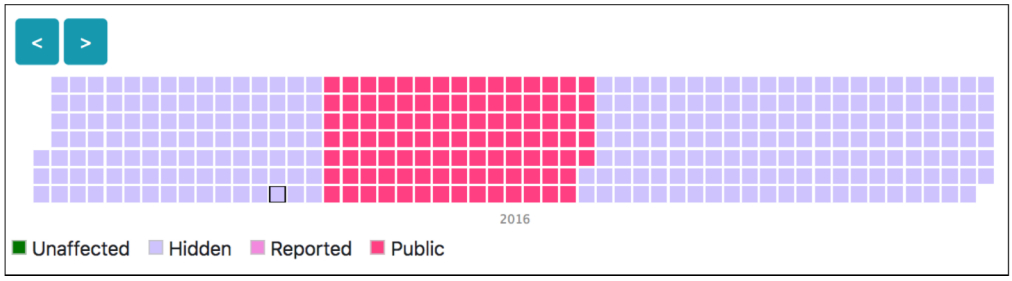


Figure 4.8: Screen-shot of the generated Dependency Discoverability Graph for the atom application using DEPREVEAL .

shows the historical exposure of the application to dependency vulnerabilities. We show in Figure 4.8 a screenshot of this report generated for the atom application. Each cell represents a day in the history of the application during a year and the colors represent the discoverability level, with dark red indicating exposure to public vulnerabilities. In the example, it is easy to see that atom was exposed for 14 weeks to public dependency vulnerabilities in 2016, by seeing how many columns show the darker red color. Users can get more information about the date by hovering the mouse over the cell.

Period of Package Discoverability is another report generated by DEPREVEAL to show the time period (in days) in which a vulnerable package affected the application, per discoverability level. Figure 4.9 shows a screenshot of this report, generated also for the atom application. From the Figure, we can observe, for example, that the package *jquery* was affected by a public, reported and hidden vulnerabilities through the project lifetime. Hovering the mouse over the tip of the red

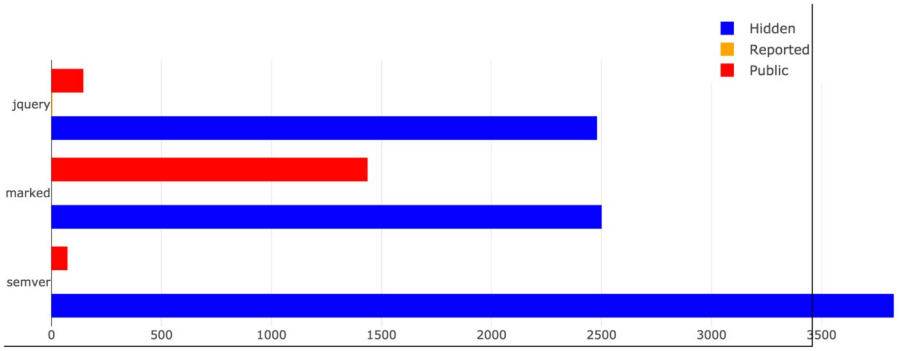


Figure 4.9: Screen-shot of the generated report Period of Discoverability for the atom application using DEPREVEAL .

bar for the *jquery* package, it is possible to notice that the application remained depending on a public vulnerability in the jquery for 145 days through the entire application lifetime. Users can also enable/disable one of the discoverability levels by clicking on the legends at the right-side of the report plot.

Furthermore, the tool generates a CSV file that contains the analysis details for the entire application lifetime to help a further investigation. Finally, note that we provide a command-line version of the DEPREVEAL , which is available from our open-source GitHub repository (Alfadel, 2021). We also provide a web user-interface for the tool to facilitate using and interacting with it (*DepReveal*, 2021).

4.9 Implications

In this section, we discuss some implications of our findings to researchers and practitioners.

Research should account for the discoverability of vulnerabilities to better assess the threats of dependencies in a software ecosystem. A public vulnerability poses a much higher threat of being exploited than an undisclosed (not-published) vulnerability. However, research so far has mainly focused on the occurrence of vulnerabilities in software packages (Decan, Mens, & Constantinou, 2018b; report, (accessed 2020)), which outputs a low-grained worst-case scenario. Our methodology is completely automated and can be replicated. Researchers can use our proposed approach to study the prevalence of vulnerable dependencies and how harmful they are in the

applications taking the vulnerability timeline into account. Future studies could contrast our findings with a similar investigation on different ecosystems (Python, Go, Java) to provide a more complete picture of the problem of vulnerable dependencies.

Developers should not solely depend on semantic versioning (SemVer) to catch security updates in dependencies. Our results show that using semantic versioning for automatic updates at the level of patch and minor releases is not a silver bullet to prevent public vulnerabilities of dependencies in JavaScript applications. Our manual inspection (RQ_2) revealed that in many cases the vulnerability fix is only available in a major release of the package, which comes at the cost of "breaking changes". To overcome this problem, DependaBot (*Dependabot Page*, accessed on 12/10/2021), an automated dependency tool, has proposed a way to estimate the migration cost of security updates for dependencies. It provides a measure for the compatibility of the version update with the application, which is calculated based on the outcome of similar updates that were already done by other applications. A high compatibility score may give developers confidence to upgrade their dependencies without incurring in further maintenance costs due to breaking changes. Other techniques can work at the application source code level to provide an estimate of how much existing code would break due to that change. These challenges emphasize the need for further research on software updating mechanisms.

Developers should adopt automated dependency tools to help them keep their dependencies secure. Our study reveals that application developers rather than package maintainers are the main responsible for the exposure to public dependency vulnerabilities (RQ_2), and they often take a long time before updating to a fixed version (RQ_3). Hence, developers need to constantly track and fix vulnerable dependences. Automated dependency tools (e.g., DependaBot) can be of great help to catch security updates as short as possible (Alfadel, Costa, Shihab, & Mkhallalati, 2021).

Our tool provides developers with a fine-grain look at vulnerable dependencies through the application history. We developed a tool (DEPREVEAL) that goes into a fine-grained level of dependency analysis for vulnerabilities by considering the entire lifetime of the application. Out

tool can be used by developers to identify the discoverability levels of the project dependencies to help them partially observe and monitor their dependencies and understand the dependency health through the application history. For example, (i) our tool can be used by developers to identify vulnerable dependencies with respect to the discoverability levels (as defined in our study) and present their distribution across the application lifetime; (ii) our tool can also be used by developers to identify how long applications have become vulnerable because of publicly known vulnerabilities; (iii) our tool lists the packages that causes the publicly disclosed vulnerabilities along with the used version constraint through the application lifetime, this can also help developers prioritize their work for monitoring specific vulnerable dependencies that exposed the application to the risk of a high threat vulnerability in the past.

4.10 Threats to Validity

Internal Validity considers the relationship between theory and observation. Our dataset contains 925 vulnerability report available in the npm advisories dataset. There might be other vulnerable packages that have been discovered but not yet reported. However, we leveraged up-to-date dataset from npm advisories, which we believe contains the most accurate information about the vulnerable packages reported to them.

This study only considered direct dependencies. Our results may vary if indirect dependencies are considered, however, due to very intensive computation requirements, we focused on the direct dependencies of applications. Indirect dependencies are usually out of the control of the application developer, hence, the problem would even be more serious, which needs to be addressed in the future. Our technique can be extended to analyse indirect dependencies considering the discoverability levels. We did not consider whether the vulnerable functionality in the package actually affects the application, i.e., whether the applications use the vulnerable code of the package. Considering this would be challenging, since our dataset is composed of thousands of applications. That said, our analysis is in line with prior work in the area of software ecosystems, which also examine dependencies in the package.json file to associate packages to applications.

Finally, note that our study relies heavily on the coverage of vulnerability advisories in npm, as

well as on the consistent application of semantic versioning, which may lead to an under-approximation that supports the argument of our study. However, our main argument is that similar studies that use similar datasets to tackle the same aspect paint a less accurate picture of the studied aspect.

External Validity is related to the generalizability of our findings. Our study is based on JavaScript applications that use npm. Hence our results may not generalize to applications written in other languages. However, the key concepts and design of our study can be applied to other package dependency networks to expand the investigation on vulnerable dependencies. Our dataset contains 6,546 JavaScript applications that use npm packages. Our dataset might be considered small when it is compared to the whole population of JavaScript applications. However, our dataset is of high quality, since we filtered out applications that are immature and have less development history, by using the filtering criteria used by Kalliamvakou et al. (Kalliamvakou et al., 2014). Also, to our knowledge, our dataset is considered to be among the largest number of JavaScript applications analyzed.

4.11 Related Work

In this section, we discus the work that is related to the study in this chapter.

Pashchenko et al. (Pashchenko et al., 2018; Pashchenko, Plate, et al., 2020) studied the vulnerability impact of 200 open-source Java libraries commonly used in SAP organisation (SAP, 1972), and found that 20% of the vulnerable dependencies are not deployed, and hence, they are not exploitable in practice. Pashchenko et al. (Pashchenko, Vu, & Massacci, 2020) indicated (based on interviews with developers) a high demand for high-level metrics to assess the maintainability and security of software packages. Our proposed tool DEPREVEAL partially fulfils such a demand since it generates analytical reports to inform developers how vulnerable their dependences are, considering the discoverability levels.

More specifically, several studies focused on analyzing the impact of security vulnerabilities in the npm ecosystem (Chinthanet et al., 2020; Decan, Mens, & Constantinou, 2018b). Recently, Bodin et al. (Chinthanet et al., 2021) analysed npm packages to study lags of vulnerable release and its fixing release, and found that the fixing release is rarely released on its own; 85.72% of the

bundled commits in the fixing release are unrelated to a fix. Similar to npm packages, Wang et al. (Wang et al., 2020) found that Java packages contained dependencies which lag for a long time and never been updated. Our study complements previous studies by analysing npm vulnerable dependencies throughout the JavaScript application lifetime, aggregating the vulnerability lifecycle through the discoverability level metric.

Other studies perform a code-based analysis to assess the danger of dependency vulnerabilities (Pashchenko et al., 2018; Plate, Ponta, & Sabetta, 2015; Ponta et al., 2020; Zapata et al., 2018). A study by Zapata et al. (Zapata et al., 2018) manually analysed 60 projects that depend on vulnerable npm packages, and found that 73.3% of them were actually safe because they did not make use of the vulnerable functionality, showing that there is an overestimation on previous reports. Our study includes another aspect that impacts vulnerable dependencies in applications, by including the discoverability levels.

There were several efforts to assess the impact of vulnerable dependencies in dependent applications (Plate et al., 2015; Ponta et al., 2020). Plate et al. (Plate et al., 2015; Ponta et al., 2020) proposed a code-centric tool that determines whether or not a Java application executes the fragment of the dependency where the vulnerable code is located. Their proposed approach is implemented in a tool called, Eclipse Steady (aka VULAS), which is an official software used by SAP to scan its Java code. Furthermore, Ponta et al. (Ponta et al., 2018, 2020) built upon their previous approach in (Plate et al., 2015) to generalize their vulnerability detection approach by using static and dynamic analysis to determine whether the vulnerable code in the library is reachable through the application call paths. Bodin et al. (Chinthanet et al., 2020) implemented an extension of the Eclipse-Steady tool to support JavaScript. They analysed analysed 42 applications to find their vulnerable constructs, showing that a code-centric approach is viable, yet, there are challenges given the dynamic nature of the JavaScript and the complexity of the npm dependencies (Chinthanet et al., 2020). Our tool (DEPREVEAL) complements these tools by looking at vulnerable dependencies through the history of a JavaScript application. DEPREVEAL aims to increase developers awareness on how often their application project is exposed to vulnerable dependencies.

Our tool could be extended to include a code-centric analysis and report the vulnerable constructs per discoverability analysis. However, the analysis at this level is indeed problematic due to

execution costs needed to analyse the code. Other automated code analysis tools work on vetting the changes in the releases of packages to analyse their lines of code. Recently, there were several efforts for auditing npm security vulnerabilities, both from academia (Gong, 2018; Staicu, Pradel, & Livshits, 2018) and from industry practitioners (*DeepScan*, (accessed 2020); *R2C*, (accessed 2020)).

4.12 Chapter Summary

The goal of the study in this chapter is to examine vulnerable dependencies in JavaScript applications based on their disclosure lifecycle. First, we define three discoverability levels for dependency vulnerabilities in JavaScript applications. Then, we perform an empirical study on 6,546 JavaScript applications to assess how discoverable vulnerable dependencies are. Our findings show that 67.9% of the examined applications depend on at least one vulnerable package. 99.42% of the affected applications depend on hidden dependency vulnerabilities. Though, 206 (4.63%) applications were still affected by a public dependency vulnerability, and they often remain affected for a substantial long time (103 days) during the application lifetime. Moreover, we examined why these applications end up depending on public dependency vulnerabilities. We observed that the application developers are mostly to blame, i.e., a fix for the vulnerable dependency is available but not patched in the application. Our findings indicate that understanding the discoverability of dependency vulnerabilities that exist in applications is key. That said, we developed a tool that supports our analysis approach for npm projects, to help developers better understand and characterize package vulnerabilities that affect their applications. Leveraging our findings, we also provide some implications to practitioners and future work.

In the previous two chapters (Chapters 3, 4), we have focused on understanding the impact of security vulnerabilities in software packages and dependent applications. While the previous studies provide tremendous knowledge that developers can use to understand security vulnerabilities that impact software packages, it is also essential to mitigate the risk of these vulnerabilities. Thus, it is important to study the effectiveness of existing mechanisms that aim to mitigate the impact of security vulnerabilities in software packages. In the following chapter, we turn our attention to examine the role of a popular mechanism (i.e., code review) in enhancing package security.

Chapter 5

Studying The Role of Code Review in Enhancing Package Security

Modern code review has been found to be effective in improving the overall quality of software. However, the effectiveness of code review at finding and mitigating security issues remains unknown. In this chapter, we explore the role of code review process in enhancing package security. In particular, we investigate 10 active and popular JavaScript projects to understand what types of security issues are raised during code review, and what kind of mitigation strategies are employed by project maintainers to address them. Our study examines 171 security issues identified in the projects during code review. We find that such issues are affecting a small fraction of project files (0.25% - 3.63%). However, raised security issues are discussed at length by project maintainers. Our investigation showed 14 types of security issues raised in code review. Issue type varies from common issues across the projects, e.g., Race Condition, Access Control, XSS, Documentation, and Overflow, to other types more frequently affecting specific projects, e.g., ReDOS, SQL injection, and Authentication. We found that most of the issues (≈ 55%) are frequently addressed and mitigated. In other cases, security concerns ended up not being fixed or are ignored by project maintainers, which may put the project users at risk. Based on our findings, we offer several suggestions that aim to motivate improvements at the process of reviewing code for security concerns.

5.1 Introduction

Security vulnerabilities have a large negative impact on the software systems. In fact, the impact of vulnerabilities is only magnified if they are identified in production, i.e., after the project version is released, which increases the chance for exploitation. An example of that is the Equifax cybersecurity incident, caused by a web-server vulnerability in Apache Struts library, which led to illegally access sensitive information from almost half of the US population (143 million American citizens) (Equifax, 2017). Therefore, project maintainers must check their code against security issues and vulnerabilities before being introduced to the project repository in order to reduce their impact as much as possible.

Nowadays, modern code review is a well-adopted practice in industrial and open source projects, especially with the support of the pull-based development model, for the purpose of ensuring software development quality. Pull Requests (PRs) are a major collaborative feature in GitHub, which allows multiple users to work simultaneously on a project and make changes and contribute to a GitHub project, and allows other contributors in the project to review, discuss, and even push follow-up commits (Chacon & Straub, 2014; *Git - Contributing to a Project*, accessed on 12/10/2021).

Previous research (e.g., (Dey & Mockus, 2020; McIntosh et al., 2016; Yu, Wang, Yin, & Wang, 2016)) provided evidence on the effect of code review process on the overall software quality level. For example, Dey and Mockuse (Dey & Mockus, 2020) examined the impact of code review characteristics on accepting PRs in the npm packages, and found that some measures (e.g., PR author and reviewer experience) could influence the PR quality. Little is known, however, of the effect of code review in relation to software security issues. In addition, studying this is important to increase the awareness of project maintainers and researchers to the role of code review in identifying and dealing with security issues. Moreover, it is important to understand how security issues are being tackled during code review by project maintainers.

Therefore, to shed light on the role of code review from a security perspective, our study aims to analyse code reviews in a set of 10 active, mature, and popular JavaScript GitHub projects. For those projects, we mine the security issues being discovered during code reviews of PRs in the projects,

and manually examine more than 4K discussion and review comments in the projects. With these data, we explore three main questions:

(1) **RQ1: How often are security issues identified during code review?**
Out of the studied 10 projects, we find 9 projects in which security issues are raised during code review, with 171 PRs containing evidence of security discussion. Moreover, we observe that such issues are raised in a small fraction of PRs (0.25% - 3.63%), affecting also a small fraction of project files (0.25% - 3.63%). However, project maintainers discuss the issues at length in the PRs, i.e., 4.82% - 28% of all PR comments are related specifically to the security related concerns.

(2) **RQ2: What types of security issues are identified during code review?**
Our manual investigation shows that the identified security issues in the projects belong to 14 types. The types of these issues range from more common types across the projects, e.g., Race Condition, Access Control, Sensitive Data Exposure, XSS, Documentation, and Overflow. Other types (e.g., ReDOS, SQL-injection, and Authentication) are more frequently affecting specific projects.

(3) **RQ3: How do developers respond to the identified security issues during code review?**
We find that the majority (54.96%) of the raised security issues are fixed and mitigated. However, many of the issues seem to be considered as having less threats (28.65%). In a few cases, we find that the project maintainers decided not to fix the issues (8.18%) or even respond to them (4.67%). We also find some issues that are not directly related to the reviewed PRs but still discussed during code review (3.54%).

In summary, the work in this chapter makes the following main contributions:

- We investigate security issues that are identified through code reviews of PRs in popular JavaScript projects.
- We manually build and validate PRs dataset that contains security-related reviews in the studied project.

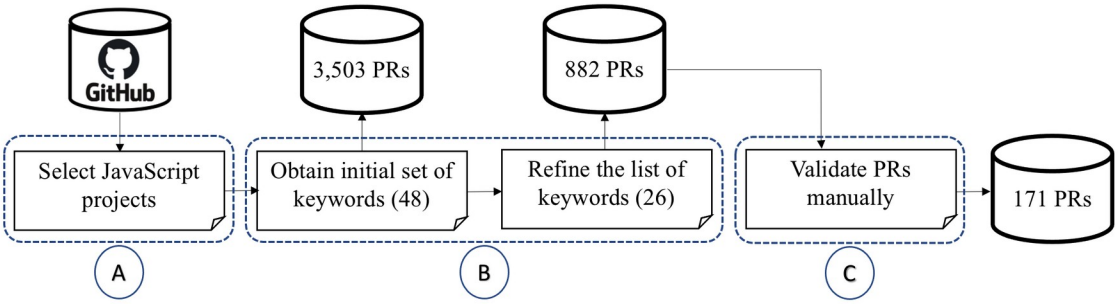


Figure 5.1: An overview of our study approach.

- Based on our findings, we offer some suggestions and implications that support the role of code review for the security of open-source projects.

Chapter organization.

This chapter is organized as following: Section 5.2 describes our study methodology. Section 5.3 presents the results of our study. Section 5.4 presents our discussion and discuss how our findings lead to implications to practitioners and future research. Section 5.5 presents the threats to validity. Section 5.6 presents the related work. Section 5.7 concludes our study.

5.2 Study Design

The main goal of our study is to examine the role that code review plays in identifying and fixing security issues that exist in open source software projects.

To achieve our goal, we resort to analyzing the discussion of code reviews through the development of open source JavaScript projects. Our study focuses on analyzing JavaScript projects that have been published as packages in the Node Package Manager (npm ecosystem). We mine and analyse Pull-Requests (PRs) that exist in the projects. Developers use the PR feature in the GitHub project as a platform for code reviews. To this end, we first (a) collect a representative set of JavaScript GitHub projects. Then, (b) we identify candidates for security-related PRs. Finally, we (c) manually validate the identified PRs, which we use for our study analysis.

Figure 5.1 provides an overview of our general approach, detailed in the remainder of the section.

5.2.1 Project selection

We analyze code review discussions in JavaScript projects that develop packages published in the npm package ecosystem. We chose to focus on JavaScript projects due to its wide popularity amongst the development community as JavaScript has consolidated itself as the most popular programming language (*Stack Overflow Developer Survey*, accessed on 12/10/2021). The npm ecosystem is the largest software package ecosystem to date, surpassing 1.9M packages published in the ecosystem (*npm - Libraries.io*, accessed on 12/10/2021), with popular packages being used in thousands of program applications (Zerouali, Mens, Decan, & De Roover, 2021). Its popularity and reach, makes the npm ecosystem a prime target for attackers, and maintainers of JavaScript projects have to act fast to identify and remediate software vulnerabilities before they are exploited (Alfadel, Costa, & Shihab, 2021). Consequently, npm has a well-renowned advisory for reporting security vulnerabilities that affect its projects (*npm advisory reports*, accessed on 12/10/2021).

We collect all npm projects available in the npm advisories dataset. These projects have some level of popularity and are known to have concerns about security as vulnerabilities have been identified and remediated in their code base. Our initial dataset contains 1,219 unique projects. Then, we filter out projects that do not have links to their repository. Of these projects, only 666 projects had links to GitHub repositories. Furthermore, to further curate the dataset for our manual analysis, we apply a filtration process on the projects. We select projects that satisfy all the following filtration criteria:

- **Security concern**: we choose projects that have a minimum number of 2 vulnerability advisories, as we want to include projects that have had a history of identifying vulnerabilities and hence, should be discussing security in their development history (Walden, 2020). Applying this filtration step leaves us with a dataset of 89 projects.

- **Popularity**: we choose projects with over 10,000 downloads per month, as popular projects are critical to the community, and many developers rely on them for their software development projects (Synopsys, 2019). Upon applying this criteria, our dataset contains 67 projects.

- **Recent activity**: we choose projects with at least ten commits made in the last month prior to the time of collecting our dataset (i.e., August, 2020), as we want to avoid projects that are no

longer active or relevant. Relevancy and activity are essential ways to ensure that the projects we are using for our analysis are current and modern (Kalliamvakou et al., 2014). This step ends us with 37 projects.

It is important to note that our study analysis is very time consuming, given that such projects contain thousands of PRs, and authors need to recognize the context in which the security issues in the PRs are being identified. The in-depth analysis of the role of code review requires extensive manual work, across multiple rounds and verified at least by two investigators. Hence, we refine our projects dataset further by selecting the top 10 most active projects our of the 37 projects (recent activity criteria) since such projects tend to be rich in pull requests, and consequently have better chances for finding and discussing security issues (Walden, 2020). The size of our projects dataset is inline with related studies that has performed similar work (Bosu, 2014; Ebert, Castor, Novielli, & Serebrenik, 2019; Paul et al., 2021); the number of analysed projects in these studies varies between one to ten projects.

Table 5.1 depicts the selected ten projects for our study. For each project, we present the project's domain, the programming languages used through the development lifetime, the age, and the total number of the PRs in the project. As seen, these projects cover multiple languages and application domains. Furthermore, the projects have a considerable long development lifespan and most projects have thousands of PRs in the project repository.

5.2.2 Identification of PR candidates

The goal of this phase is to identify PRs with security-related reviews in the selected projects. To that aim, we conduct a three-step methodology: 1) we elicit a set of security-related keywords, 2) we refine this list of keywords, and 3) we use the refined list to identify the set of PRs with security-related reviews . In the following, we describe each step in details.

1) Obtaining initial dataset of security-related keywords. To identify relevant PRs that are of interest to our study, we use security-related keywords. Influenced by the related literature, we initially adopt a dataset of 48 security-related keywords used in the previous studies (Bosu, 2014; Paul et al., 2021). We use this initial set of keywords and apply each of them, using a regular

Table 5.1: Overview of Projects.

Project	Project Description	Language	Age (in years)	# PRs
Marked	A low-level compiler for parsing markdown without caching or blocking for long periods of time (*marked - npm*, accessed on 12/10/2021).	JavaScript, HTML	10 years	758
Moment	A lightweight date library for parsing, validating, manipulating, and formatting dates (*moment - npm*, accessed on 12/10/2021).	JavaScript	10 years	1,812
Parse-Server	An open source backend that can be deployed to any infrastructure that can run JavaScript (*parse-server - npm*, accessed on 12/10/2021).	JavaScript	5 years	2,869
Sequelize	A promise-based JavaScript ORM for Postgres, MySQL, MariaDB, SQLite and Microsoft SQL Server (*sequelize - npm*, accessed on 12/10/2021).	JavaScript, TypeScript	11 years	3,359
Node-Red	A framework that provides a browser-based editor that makes it easy to write APIs and online services (*node-red - npm*, accessed on 12/10/2021).	JavaScript	8 years	1,140
Strapi	A fully customizable open-source software that provides a headless content management system (*strapi - npm*, accessed on 12/10/2021).	JavaScript	6 years	2,878
Infor-Design	A framework-independent UI library consisting of CSS and JS that provides tools to create user experiences (*infor-design - npm*, accessed on 12/10/2021).	JavaScript, TypeScript, HTML, CSS	5 years	2,524
Electron	A framework that helps build cross-platform desktop apps with JavaScript, HTML, and CSS (*electron - npm*, accessed on 12/10/2021).	JavaScript, TypeScript, C++, Python, HTML	8 years	11,794
React	A library for building user interfaces (*react - npm*, accessed on 12/10/2021).	JavaScript, TypeScript, C++, CSS, HTML	8 years	9,673
Uglify-js	A parser, minifier, compressor and beautifier toolkit (*uglify-js - npm*, accessed on 12/10/2021).	JavaScript, HTML, Shell	9 years	1,197

expression, to identify security-related reviews. This regular expression identifies all keywords by exact match that have whitespace delimiting it before the keyword. To this aim, we collect all the comments and discussions from all the PRs in the projects and identify PRs that contain any keyword from our previously selected keywords dataset. After running the 48 keywords against all the PRs (i.e., 38,004 PRs), we identify 3,503 PRs as candidates for our study.

2) Refining the list of keywords. We identify 3,503 candidates of PRs, however, through a preliminary manual inspection of the identified PRs, we observe that a considerable share of PRs were not relevant for our study, i.e., they do not discuss security-related issues. This is because some of the used keywords are specific to languages other than JavaScript, and hence, include a lot of noise in our PRs dataset. For example, the initial list of keywords contained keywords such as “css”, which in many of the cases for our selected projects, e.g., (*Pull Request #2984*,

accessed on 12/10/2021; *Pull Request #4561*, accessed on 12/10/2021; *Pull Request #685*, accessed on 12/10/2021), refers to the cascading style sheets, the styling language for web pages, and not Cross Site Scripting (XSS) vulnerabilities, as it is commonly referred to in the security domain.

Therefore, to reduce the number of irrelevant PRs from our candidate set, we further verify and refine the list of keywords. First, we randomly select a statistically significant sample size of the candidate set for each keyword. The chosen sample size is significant enough to satisfy a 95% confidence level with a 5% confidence interval for each designated population. Once the PRs sample for each keyword is selected, we verify whether the identified PRs use the keywords in a discussion about security or not, by manually going through the PRs' comments. If the keyword was used in a security discussion of the PR, then we consider the PR a relevant case at this stage of the study and irrelevant otherwise. After going through all the samples of the PRs for each keyword, we decide the inclusion of the keyword by setting a reasonable threshold, i.e., if a keyword retrieves less than 10% relevant cases in its sample, we remove it from our list of keywords. Otherwise, the keyword is included. We chose 10% because we wanted to use a low threshold to preserve as large a variety of keywords as possible, as limiting the amount of keywords could bias the dataset towards only finding issues related to those specific keywords.

This process yields 23 *refined keywords*. Note that as we go through the PRs of the relevant cases, we identified keywords that were being used in security-related reviews and were not in our initial list of keywords. This step ends us with three newly added keywords, namely, denial of service, DDOS, redos. Table 5.2 shows the list of the final 26 unique keywords used in our study. The keywords are associated with a Common Weakness Enumeration (CWE) (*CWE List Version*, accessed on 12/10/2021). CWEs are well-defined classifications for the explored software weaknesses (e.g. CWE-121 corresponds to Buffer Overflow security vulnerabilities).

3) Identifying PR candidates for our study. We then use the list of refined 26 keywords to build the dataset of PRs for our study, by searching for the keywords in the PRs comments. This process yields a total of 882 PRs as candidates for our study.

Table 5.2: List of refined security-related keywords.

Vulnerability Type	CWE-ID	Keywords
Race Condition	362 - 368	race, racy
Buffer Overflow	120 - 127	overflow
Integer Overflow	190, 191, 680	overflow, underflow
Improper Access	22, 264, 269, 276, 281 -290	unauthenticated, gain access, permission
Cross Site Scripting (XSS)	79 - 87	cross site, XSS
Denial of Service (DoS) / Crash	248, 400 - 406, 754, 755	denial service, DOS, *denial of service**, *DDOS**, *redos**
Deadlock	833	deadlock
SQL Injection	89	injection
Cross Site Request Forgery	352	cross site, CSRF, forged
Common keywords	-	security, vulnerability, vulnerable, overrun, exploit, insecure, breach, threat

* Keywords in *italic* are our additions to this list.

5.2.3 Manual validation of the identified PR candidates

In the previous phase, we were able to curate a refined list of keywords to identify PR candidates (882 PRs), and reduce the number of irrelevant PRs for our study. However, some of those PRs might still be wrongly identified due to the limitation of our keyword search technique. For example, our technique flags this PR (*Pull Request #991*, accessed on 12/10/2021) as relevant because one of the review comments in the PR contains the "permission" keyword:

> "*...when users change the provider config from UI (Roles and Permission page), we will save the new config into db, and also sync it into JSON file as well, right?*"

As shown in the quote, the "permission" keyword is used in the context of a webpage name, and has no security implications to the project. Examples like this motivated us to further manually validate the set of 882 PRs. Hence, in this step we conduct a thorough manual analysis on the 882 PRs to filter out the irrelevant cases from our further analysis.

To filter out PRs from our candidate set that did not discuss security-related reviews, we manually look through all the 882 PRs. We examine whether the contributors and reviewers of each PR

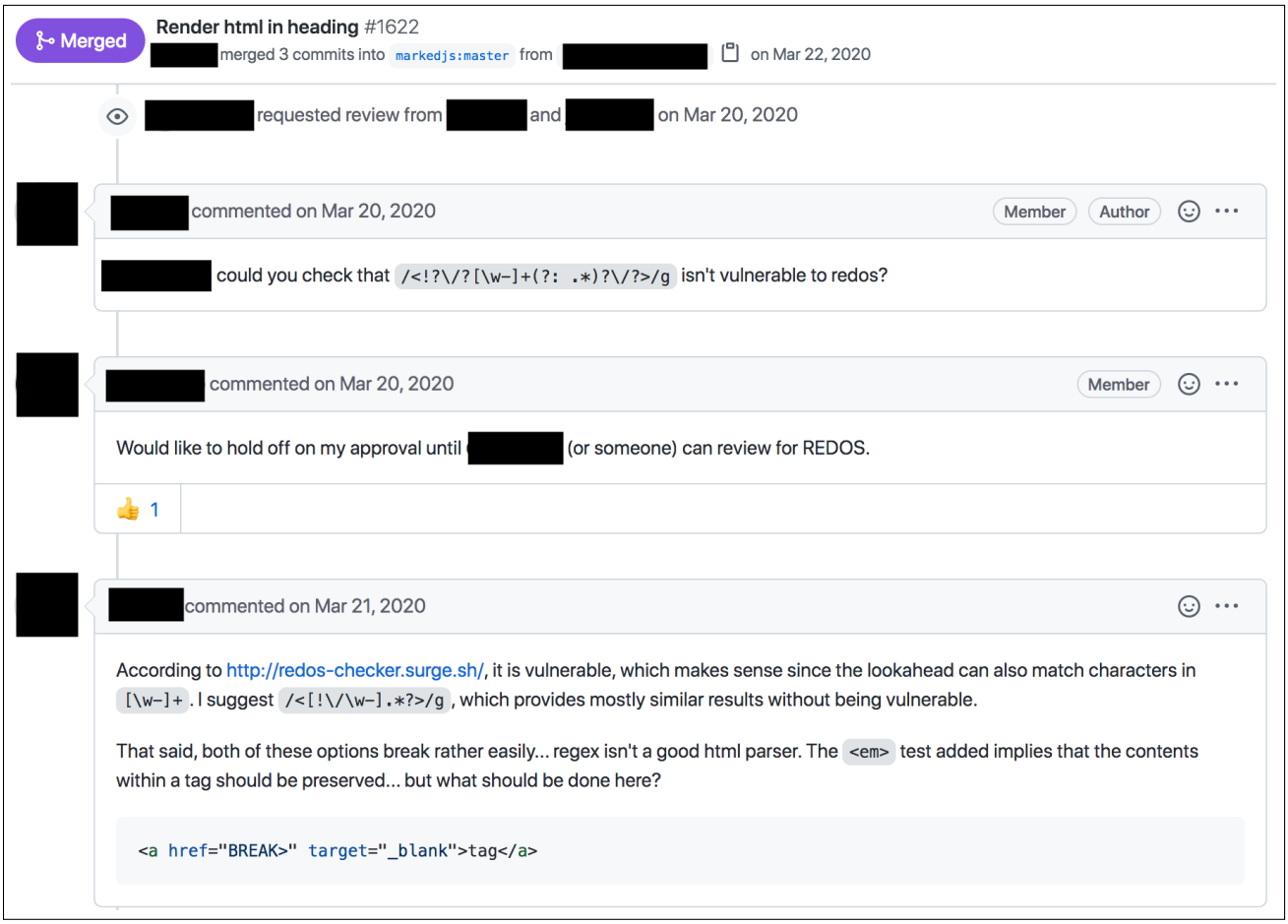


Figure 5.2: Example of a security issue raised during code review.

discuss security-related topics in the PRs' comments. If so, the PR is included in the dataset for our later analysis. To evaluate the agreement, we invite an external annotator to manually label the PRs; we used Cohens kappa coefficient (Cohen, 1960), which is a well-known statistical method that evaluates the inter-rater agreement level for categorical scales. In our manual labelling of the PRs, the level of agreement between the two annotators was of +0.92. At a macro scale, we extract the PRs information in eight rounds (almost 110 PRs per batch). Upon completion of each round, we invite an external annotators to manually validate the PRs. We meet and discuss any conflicts about including the PR. The goal of these meetings is to address any inconsistencies and to work together to resolve them.

Our in-depth manual analysis identifies 171 validated PRs with security-related reviews (out of 882 PRs), which span across all the projects in our dataset. Figure 5.2 shows an example of

a relevant PR that discusses a security issue during code review. In this figure, one of the PR's reviewers in the Marked project asked the contribution to check whether the code is vulnerable against a ReDOS security issue.

5.3 Study Results

In this section, we answer our RQs. For each RQ, we provide a motivation, describe the approach, and present the results.

RQ1: How often are security issues identified during code review?

Motivation: The goal of this RQ is to gain an initial overview of the prevalence of security issues identified in a code review, allowing us to quantify the degree of effectiveness of code review for security purposes. In turn, this will help project maintainers and the community to set realistic expectations on the effort that is put in security-related code reviews and how often security issues are raised in regular code review processes.

In particular, we examine how often a security-related concern is raised in code review under three different granularity levels: a) number of PRs; b) number of files touched by the PRs; and c) number of security-related comments in the PRs.

Approach: We employ the methodology explained in Section 5.2.3 to manually identify the PRs with security-related reviews in each of the studied projects. To identify the files changed in the PRs and the prevalence of security comments in the overall PR discussion, we use the meta data of each PR, to get the files that contain the security issue and the comments that discuss the security issue. More specifically, we use the following methodology to gather our results:

- To find the distribution of security issues at the PR level, we quantify the number of security-related PRs per project.
- At the file level, we quantify the number of unique files that contain the identified security issue per PR, and sum up the number of unique files for each project.

Table 5.3: Distribution of security-related issues distributed at different granularities, per project.

Granularity	Project	# Total	# Security-Related	%
PRs	Marked	758	27	3.56%
	Moment	1,812	2	0.11%
	Parse-Server	2,869	19	0.66%
	Sequelize	3,359	19	0.57%
	Node-Red	1,140	6	0.53%
	Strapi	2,878	18	0.63%
	Electron	11,794	47	0.39%
	React	9,673	31	0.32%
	Uglify-js	1,197	2	0.17%
Files	Marked	327	9	2.75%
	Moment	786	2	0.25%
	Parse-Server	413	15	3.63%
	Sequelize	491	13	2.65%
	Node-Red	1,187	6	0.50%
	Strapi	3,360	15	0.45%
	Electron	2,130	42	1.97%
	React	2,105	30	1.43%
	Uglify-js	276	2	0.72%
Comments in Security-PRs	Marked	665	108	16.24%
	Moment	25	7	28%
	Parse-Server	380	49	12.89%
	Sequelize	809	39	4.82%
	Node-Red	65	8	12.31%
	Strapi	309	39	12.62%
	Electron	1,250	137	10.96%
	React	777	63	8.11%
	Uglify-js	35	5	14.29%

- Finally, at the comment level, we quantify the number of comments that specifically discuss the security issue per PR, and then sum up the total number of such comments for each project.

We normalize the result by the number of total PRs in the project, total files that the project has, and total comments in the PRs, respectively.

Result: Table 5.3 shows the distribution of the 171 PRs identified in the studied projects at different levels of granularity. Surprisingly, we observe that the `Infor Design` project is the only project that had no security issues raised during code review, i.e., the 171 cases are distributed across the remaining 9 projects in our dataset. As a result, in terms of the number of security-related PRs, we

find that **the 171 PRs correspond to 0.49% (less than 1%) of all PRs in the projects.** Concretely, the project `Electron` has the largest occurrence of security issues raised during code review (47 PRs), while there are only 2 PRs with raised security concerns in `Moment` and `Uglify-js` projects. Overall, the rate of PRs with security-related reviews varies from 3.56% in `Marked` to only 0.11% in `Moment`, showing that only a minority of PRs raise any concerns about security. Consequently, **the identified security issues are concentrated on a small fraction of the projects' files (0.25% - 3.63%).** For example, marked has 27 security issues that are identified in 9 files out of the 327 the project currently contains.

Although the PRs with security-related reviews seem rare at both the PR and file granularity levels, once a maintainer expresses a security concern on the PR, maintainers discuss it at length in the PRs. Between 4.82% - 28% of all comments in the 171 PRs are related specifically to the security related concern.

Security issues are raised in a small fraction of PRs (0.25% - 3.63%), affecting also a small fraction of project files (0.25% - 3.63%). However, raised security issues are discussed at length by project maintainers (4.82% - 28% of all PR comments).

RQ2: What types of security issues are identified during code review?

Motivation: The goal of this RQ is to understand the types of security issues that project maintainers discover and discuss during the code review process. This investigation is important to help researchers and practitioners compare and contrast these results with other approaches, such as bug bounties, code inspections, all the way to software inspection tools that have been very well studied (e.g., (Aloraini, Nagappan, German, Hayashi, & Higo, 2019; Imtiaz, Thorne, & Williams, 2021; Yang, Tan, Peyton, & Duer, 2019)). More important, such comparison will help us understand the types of issues code review can be effectively employed and where maintainers may need better assist to identify issues at code review time.

Approach: To categorize the identified security issues, we resort to in-depth manual analysis of the 171 PRs in our dataset. In particular, we manually inspect the description of the PR, the review

comments and discussion, and the PR commits. We independently analyze the PRs using an open card-sort method (Fincher & Tenenberg, 2005), where labels for the security issues are created during the labeling process and each new label is discussed among annotators. We extract the PRs' labels in multiple rounds (note that the rounds are performed during the manual inspection phase that we described in Section 5.2.3). Then, we invite an external annotator to manually validate the PRs. We report the agreement level of the labelling given by both annotators. We found that the annotators have a high-level agreement with Cohens kappa coefficient of +0.79. Finally, both annotators meet and discuss any conflicts, i.e., conflicting labels were resolved through discussions to reach a consensus. The annotators provide one single label for each PR utilizing online search for the issue type in public advisories databases, e.g., the CWE classification (*CWE - Common Weakness Enumeration*, accessed on 12/10/2021), to help us classify the issue according to the CWE specification.

Result: Through our manual analysis, **we find 14 different types of security issues identified in the PRs during code review.** Table 5.4 shows the description and the frequency of each type represented in our dataset, sorted by frequency in descending order.

As seen from Table 5.4, some types are common across projects, i.e., they exist in a high number of projects. Others are more frequent within certain projects. Next, we explain some types in more details.

Common security types across the projects. From Table 5.4, we observe that 7 types are common across the projects, i.e., they exist in 3 or more projects. For example, we find **23 Race Condition** security issues, which affect three projects in our dataset, namely, `Electron`, `Parse-Server` and `Sequelize`. A race condition basically means that two threads are trying to access the same data, which results in having wrong data in the threads or causing errors because of trying to use the same resource. A race condition can compromise the application security where it sometimes causes a thread to access security sensitive information from another thread, through some shared memory, which would result in leaking information if the second thread isn't secured before the data is being accessed. In some of those cases, like in the case of the `Sequelize` project, it is possible

Table 5.4: Types of security issues identified during code review and their frequency.

Type	Description	Frequency	# Projects
ReDOS	A regular expression denial of service (ReDoS) is an attack that produces a denial of service by providing a regular expression that takes a very long time to evaluate, which may lead to either slowing down the system or becoming unresponsive.	23	1
Race Condition	Occurs when two or more threads can access shared data and they try to change it at the same time, which may lead to multiple issues, e.g., alter, manipulate, steal data, and malicious code.	23	3
Access Control	A system that does not restrict or incorrectly restricts access to a resource from an unauthorized actor suffer from Access Control security issue.	23	6
XSS	XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user.	22	3
SQL Injection	SQL injection attack consists of insertion or injection of a SQL query via the input data from the client to the application to spoof identity.	14	2
Documentation	In such cases, developers discuss issues related to enriching the project documentation for a better and more secure use of the project.	14	5
Improper Authentication	A weakness that allows an attacker to either capture or bypass the authentication methods that are used by a web application.	13	2
Sensitive Data Exposure	Occurs as a result of not adequately protecting a database where information is stored	10	5
Remote Code Injection	Occurs when an attacker has the ability to run system commands remotely on the vulnerable application.	9	4
Overflow	Occurs when the entered data in a buffer overflows its capacity to adjacent memory location causing the program to crash.	7	5
Deadlock	Occurs when the software contains multiple threads or executable segments that are waiting for each other to release a necessary lock, resulting in deadlock.	4	2
Improper Input Validation	Occurs the project does not validate the input properties that are required to process the data safely and correctly.	4	1
Vulnerable Package	A third-party vulnerability contains a vulnerability.	3	2
DDOS	Occurs when an attacker floods the target application with traffic or sending it information that triggers a crash.	2	1

that two callbacks are made, and there is a race condition between them for which will perform some functionality first. However, the majority of race condition cases are found in the C++ layer of those JavaScript projects, e.g., package maintainers of the `Electron` project re-use a lot of the code of Chromium, which is C/C++ based project, to render the package user-interface (UI). As an example, in this PR (*Pull Request #20818*, accessed on 12/10/2021), the project maintainers

found a race condition stemming from the C++ code that was spinning up threads to perform some functionality. Developers suggested to extract the code into the JavaScript layer and use callbacks instead in order to mitigate the issue and ensure that the correct functionality is still being called on a particular failure.

Access Control (23) is another security issue type that seems to be common across the studied projects; we find 23 cases distributed across 6 projects. The reason for this common security concern is that most projects have to manage several permission options, which may result in issues related to Permission and Access Control. For example, the `Strapi` project manages various permissions, and has encountered an issue in this PR (*Pull Request #4790*, accessed on 12/10/2021a). The proposed PR had vulnerable changes, allowing certain users to access files beyond their permission scheme, or even allow users to register themselves as admins without further control (*Pull Request #3201*, accessed on 12/10/2021; *Pull Request #5330*, accessed on 12/10/2021).

In addition, our manual analysis finds several cases of **Sensitive Data Exposure (10)** issue, affecting 5 projects. We found that issues related to Sensitive Data Exposure can range from storing information in plaintext, logging sensitive information or even exceptions and error messages (e.g., (*Merge pull request from parse-server@da905a3*, accessed on 12/10/2021; *Pull Request #2402*, accessed on 12/10/2021; *Pull Request #2576*, accessed on 12/10/2021a)). For example, in some cases, maintainers raised the issue that the system was logging sensitive information that should not be exposed. As an example in this PR (*Pull Request #2576*, accessed on 12/10/2021a), the session id is used as a token generated on the server and stored on the client by means of a cookie, used in later communications to identify the user. Developers may log the session id to track the user interactions with the application during a session, however, if an attacker gets access to live logs, he could use the session id to impersonate active users.

Cross-site scripting (22) is another common issue, which is triggered at the client-side when a potential attacker sends a malicious code in the form of a browser side script, e.g., to hijack user's account and credentials. In our analysis, we find that Cross-site scripting (XSS) type affects three web-based projects in our dataset. We find 22 cases of XSS issues where a lot of HTML components are rendered in the projects. A lot of these issues arise when the project is failing to properly escape the inserted HTML, which can cause unwanted cross-site scripting attacks. As a

result, many developers spend a lot of time discussing and trying to figure out the best approach to escape potentially malicious HTML, such as the case for this PR (*Pull Request #3152*, accessed on 12/10/2021).

Other commonly discussed security types include Overflow, Remote Code Injection and Documentation. In fact, we find the **Documentation (14)** cases to be important; they act as the communication medium between package maintainers and application developers who use the package. In such cases, package maintainers discuss inadequate or incomplete documentation of critical usages of the package functionality. For example, in this PR (*Pull Request #23650*, accessed on 12/10/2021), a project maintainer states the following:

> *"..it can lead people to the incorrect assumption that they should actually run a server in the main process, which will confuse folks with this comparison. The relationship between browser/renderer process has fundamentally different security, performance and communication constraints than a traditional client-server model that can not be sufficiently explaining in this comparison".*

As shown, the case specifies that the proposed documentation fails to properly explain the security constraints of the system, which can mislead package users.

Other cases are related to attributes misnaming. Such issues indicate, for example, misnaming a variable to make it seem more secure than it actually is. An example of this case can be seen in this PR (*Pull Request #13367*, accessed on 12/10/2021a), where a variable is called SafeValue, which has security connotations, while it actually is not necessarily secure (*Pull Request #13367*, accessed on 12/10/2021b). Such a case can cause users of the package to misuse its API and introduce vulnerabilities into their applications.

Frequent security types within specific projects. From Table 5.4, we also observe that some types are more frequent in in specific projects. For example, we find **23 ReDOS** cases. ReDOS is one form of denial of service attacks, which occurs by providing a regular expression that takes long time to process which causes a system to crash or take a disproportional amount of time. We find ReDOS issues affecting only the `Marked` project, a compiler for parsing markdown formats.

When parsing markdown text, `Marked` uses a lot of regular expressions, and as such, is very prone to creating such issues, as shown in various PRs (*Pull Request #1305*, accessed on 12/10/2021; *Pull Request #1598*, accessed on 12/10/2021a; *Pull Request #1683*, accessed on 12/10/2021a). In fact, we observe that the project maintainers of Marked seem to be employing a static analysis tool (called *vuln-regex-detector (*vuln-regex-detector*, accessed on 12/10/2021)*), integrating the tool into their CI pipeline, as shown in this PR (*Pull Request #1220*, accessed on 12/10/2021). Moreover, the maintainers seems to seek the help of someone in the team who is regarded as a security expert to help detect whether certain regular expressions are actually vulnerable. We observed his involvement in most ReDOS issues (e.g., (*Pull Request #1083*, accessed on 12/10/2021; *Pull Request #1305*, accessed on 12/10/2021; *Pull Request #1598*, accessed on 12/10/2021a; *Pull Request #1683*, accessed on 12/10/2021b)). This indicates that such issues of ReDOS are not straightforward to spot during code review, and may require automated tools as well as "security experts" to better identify and validate the cases.

We find several cases of **SQL injection (14)**, which affect two projects only, `Sequelize` and `Strapi`. SQL injection issues generally produce attacks by injecting SQL queries via the input data to spoof identity. We observe that 10 out of 14 cases are present in `Sequelize`, which is a powerful package in the JavaScript, which deals with managing SQL databases, and hence, it is more susceptible to SQL injection issues. Most of SQL injection cases that need to be dealt with in our dataset are cases that require escaping characters from potentially malicious strings that would inject SQL queries that grant unauthorized access to the database (*Pull Request #7160*, accessed on 12/10/2021; *Pull Request #783*, accessed on 12/10/2021a).

Issues related to **Authentication (13)** are mostly present in the `Parse-Server` project (6/12 cases), where user authentication is managed. In such cases where the authentication is not properly managed, a potential attacker would gain access to a sensitive data or functionality. For example, in this PR (*Pull Request #4305*, accessed on 12/10/2021a), a reviewer noticed that a public unauthenticated request was returning information about the server version. This can be risky as attackers are allowed to send requests to servers to check if they are running a vulnerable version of the server and take advantage to perform further exploits.

We find other less frequent types, e.g., Deadlock, DDOS, Improper Input Validation, and Vulnerable Packages. Finally, note that we compare the types of security issues identified during code review with the post-release security issues (i.e., advisories) in the Discussion Section. Security advisories are security vulnerabilities that affect the post-release version of the projects and have been announced in public databases.

Our investigation showed 14 types of security issues raised in code review. Issue type varies from common issues across the projects, e.g., Race Condition, Access Control, XSS, Documentation, and Overflow, to other types more frequently affecting specific projects, e.g., ReDOS, SQL injection, and Authentication.

RQ3: How do developers respond to the identified security issues during code review?

Motivation: As shown in RQ $_2$, although various types of security-related issues are brought up during the code review, how the issue is tackled, if at all, is crucial to understand how effective the process of reviewing code is. Therefore, in this RQ, we investigate how developers respond to the identified security issues and the mitigation strategies employed. Doing so is important to motivate improvements at code review process, with the aim of increasing its effectiveness.

Approach: To find out how developers respond to the identified security issues in our dataset, we manually inspect the discussion and reviews comments associated with the 171 PRs in the dataset. In particular, we examine whether the security issue is resolved in a way that increases the overall security of the related project, and how the issue was tackled and mitigated during the discussion. Similarly to RQ$_1$, two authors independently classify the responses using an open card-sort method (Fincher & Tenenberg, 2005), where labels are created during the labeling process, by looking through the discussions, code changes and commit history that occurred through the code review process. For this manual labelling, a high level of agreement is reported with Cohens kappa coefficient of +0.93. Once again, when different labels were assigned to the same PR, the annotators discuss them to reach a consensus.

Table 5.5: Response themes for handling the 171 identified security issues.

Response Theme	Description	# Total (%)
Fixed	The security issue is raised during code review and evidence for a related fix is observed.	79 (46.19%)
No Threat	Project maintainers come to a conclusion that the security issue has no actual threats to the project.	49 (28.65%)
PR Rejected	The raised security issue caused the PR to be rejected by project maintainers.	15 (8.77%)
Not Fixed	Project maintainers opted not to fix the raised security issue, often due to very complex technical difficulties.	14 (8.18%)
No Response	The security issue is raised during code review, but no discussion and changes are made to reflect on these concerns.	8 (4.67%)
General Security Discussion	Project maintainers discuss a security issue that is not directly relevant to the reviewed PR but rather a general security concern to the project.	6 (3.54%)

Result: Table 5.5 presents 6 themes of the responses to the issues raised in the 171 PRs, identified by our manual analysis. Below, we provide more details about each response theme.

Fixed (79 cases). This is the most common way of responding to the identified security issues. In such cases, we find that the security concern was discovered during code review in addition to an evidence for a related fix. In most cases, we observe that the issue is fixed, and the PR is merged (e.g., (*Pull Request #2576*, accessed on 12/10/2021b)). In other cases, we observe that the security issue is fixed, but the code is flawed for some other non-security reason, which led the PR to be closed (*Pull Request #4895*, accessed on 12/10/2021). In a few cases (*Pull Request #783*, accessed on 12/10/2021b), we observe that the reviewers suggest to open a new PR to properly design the solution that tackles the raised security issue. An example of a fix is given in this PR (*Pull Request #4305*, accessed on 12/10/2021a). In this example, the maintainers found an authentication issue where a potential attacker could have access to a public unauthenticated request, as stated:

> "*... Returning information about the server version on a public unauthenticated request makes it really easy to develop bots that check for a version of Parse Server that is vulnerable to an attack, lowering the cost of effort for a random attacker to locate vulnerable servers.*"

The same maintainer also suggests a solution for the issue:

> "*... If the Health Check is going to return structured data, I think that's a feature that should be possible to disable for security hardening. I'd prefer to see the health check do more - but still just return OK. Specifically, it would be nice if it did a simple round-trip to the database that does nothing but confirm the database server is up...*"

Other maintainers agreed on the relevance of the issue and the suggested fix as well. Hence, the issue was fixed by removing the related information of the version in the JSON response, as shown in this commit (*Pull Request #4305*, accessed on 12/10/2021b).

No Threat (49 cases). Of the total number of the analysed security issues, we observe that 49 cases do not impact the corresponding part in the project. In such cases, project maintainers did not reach a consensus on the identified security issue. We observe that a reviewer pointed out a security issue, but was effectively deemed by other reviewers and/or by the contributor that it was actually not a security concern, e.g., the identified issue was not a real threat to the project. For example, as shown in this PR (*Pull Request #5951*, accessed on 12/10/2021), a reviewer discussed some flaws related to the design of the permission feature and how it works. However, after the discussion, the reviewers agreed that it didn't seem to be any security concern. One maintainer stated:

> "*... the pattern of code was actually widely used in the project and known to not have security concerns".*

In other examples where the raised issue is not considered as harmful (*Pull Request #1224*, accessed on 12/10/2021), project maintainers find that the issue can not be triggered accidentally in a normal context, i.e., the package users should know how to use the functionality in a secure manner:

> "*...we should educate our dependents on the safe way to deal with parsing user input. (i.e. web worker/vm.runInNewContext).*"

In other cases, project maintainers find that the identified security issue has no direct impact on the project. For example, in this PR (*Pull Request #1472*, accessed on 12/10/2021) that concerns

about a vulnerable dependency, the vulnerability doesn't affect the end users, since the dependency is solely used as a development dependency:

> "*... for users who are using marked, they do not see (and are not affected) by dev dependency vulnerabilities".*

PR Rejected (15 cases). In 15 PRs, the raised security issues caused the PR to be rejected by their respective project maintainers. For example, in those PRs (e.g., (*Pull Request #3163*, accessed on 12/10/2021; *Pull Request #4790*, accessed on 12/10/2021b; *Pull Request #4822*, accessed on 12/10/2021)), the proposed changes in the PR are vulnerable, and not easy to fix. Given that the proposed changes in the PR are discussed to be lower on the priority list, the team decided to close the PR to avoid the raised security concern. In this example (*Pull Request #4790*, accessed on 12/10/2021b), the maintainer stated:

> "*Closing this PR because of security issue...With this, we can access to all the users base of a group."*

Not Fixed (14 cases). We observe that in 14 PRs, the project maintainers opted for not fixing the raised security issue, often due to very complex technical difficulties. For example, in this PR (*Pull Request #2375*, accessed on 12/10/2021), the project maintainers clearly discuss a race condition in the code. However, through our manual inspection, we observe no action was taken in the PR to fix the race condition since the maintainers don't seem to be able to find where the issue is coming from and don't seem willing to invest time into it at this point of the project, as stated:

> "*...imperfection is to be expected when there's only been one iteration :)"*

Another example (e.g., (*Pull Request #16254*, accessed on 12/10/2021)) is where project maintainers decide not to fix the issue in the current project release, as it would cause a breaking change and the fix could require significant code changes that replace entire underlying requirements of the

PR. In this case, the project maintainers prioritize respecting the release deadlines over fixing the security issue.

In some other cases, the project maintainers are offloading the responsibility of security to its users (the dependent applications). In this example (*Pull Request #9224*, accessed on 12/10/2021), the maintainers stated:

> "*...there is a responsibility up to the developers of Electron projects to ensure the content they are pulling in is safe and trusted"*. And that *"Electron intentionally breaks security and the sandbox to make applications possible"*.

This indicates that in certain cases, the project maintainers need to weigh the pros and cons of securing their project, as there is a tradeoff between usability for the project users and security of the project itself.

No Response (8 cases). The cases under this category are concerns that are raised by maintainers but were completely ignored in the discussion. For example, in these PRs (*Pull Request #1598*, accessed on 12/10/2021b; *Pull Request #18673*, accessed on 12/10/2021), we observe that a specific reviewer raised an issue related to ReDOS, but we could not find any evidence of a discussion or any response back. In this example (*Pull Request #1598*, accessed on 12/10/2021b), one maintainer raised a potential issue related to ReDOS issue, and asked another maintainer to validate it. However, we didn't observe any response back from other maintainers. In other cases, we observe that the security issue was not raised early enough, i.e., the issue was identified only after the PR's decision was already taken (closed/merged). As an example, in this PR (*Pull Request #10816*, accessed on 12/10/2021), we find that, after merging the PR, a developer adds a comment concerning a potential SQL injection. In such cases, we find no evidence of discussing or addressing the security issue after being raised.

General Security Discussion (6 cases). In such cases, reviewers discuss issues that are not directly relevant to the reviewed PR, i.e., they discuss general issues that come along the discussion of other related issues. This can be some improvements for security-features or potential approaches

to fix security issues. In such cases, no actions is taken in the PR since the discussion in not specific to the PR changes. For example, in this PR (*Pull Request #714*, accessed on 12/10/2021), the project maintainers discuss various ideas and approaches to escaping characters to prevent potential XSS issues, though no actual XSS issue is raised. Throughout this discussion, they identify potential security issues in each others' ideas and refine them to come up with an optimal solution. This helps project maintainers plan out a secure approach to prevent a potential vulnerability from an un-discussed plan. As shown in the following quote:

> "*...I'd be curious to see if an indexOf('.') !== -1 check before escaping would help perf in the common case. Since the common case is no dot. We could also escape to a format that doesn't need re-escaping when it goes into the DOM attribute. Since we already have one escape pass, we can utilize that for both. Might be dangerous though. Easy to open up XSS vulnerabilities.*

The author is relating the content of a PR to some future work or feature, and discussing their security concerns for the possible approaches. This provides an entryway for a discussion to further discuss how they should handle these future works, before they begin tackling them.

The majority (54.96%) of the identified security issues during code review are fixed and mitigated. However, many of the issues seem to be considered as having less threats (28.65%). In a few cases, the project maintainers do not fix the issues (8.18%) or even respond to them (4.67%). Interestingly, some of the issues are not directly related to the reviewed PRs, but still discussed during code review (3.54%).

5.4 Discussion and Implications

In this section, we present a discussion on the comparison of security issues identified during code review to the post-release security issues (advisories) that are only identified after the project release. Then, we provide insights about how our findings can improve the practice for practitioners and researchers.

5.4.1 Comparison with advisories dataset

Code review identifies security issues before these issues are merged in the codebase and go to production. However, security issues uncaught by code review may later become known vulnerabilities in the projects. To better understand the effectiveness and limitations of code review, we compare issues identified during code review to post-release security vulnerabilities (advisories) that have been reported after the project release production. Such comparison will help us understand whether there are certain types of issues that code review can be effectively employed to identify.

The npm registry maintains a security advisories database to provide regular updates on post-release security vulnerabilities in the JavaScript projects (*npm advisories*, accessed on 12/10/2021). We collect all npm security advisories of the studied projects in the same timeline of the collected security-related PRs, i.e., we collect all advisories that have their publication date before August, 2020.

We report the results of our comparison by cross referencing the security types identified during code review with advisories types that affect the projects in our dataset. We manually check whether each one of the 14 types identified during code review exist in the advisories dataset. Table 5.6 shows the types identified in our study, and whether they are mentioned in the advisories dataset. From the table, **we can observe that four types in our study are not mentioned by advisories dataset, namely *Race Condition*, *Access Control*, *Documentation*, and *Deadlock*.**

The nature of the types that were more commonly found in code review requires in-depth knowledge of the project domain and implementation specifics. For example, issues related to Race Condition and Deadlock stress this point as their identification and solution require an in-depth understanding of the problematic code and the concerned threads, how they collaborate to deliver the functionality and more importantly how to add code that puts proper constraints on their collaboration to fix the security issue. Similarly, Access Control issues are difficult to spot in advance. In the case of Access Control (e.g., in this PR (*Pull Request #1681*, accessed on 12/10/2021)), we observe that in order to understand whether certain resources can be exposed or not, a deep understanding of the project users requirements is necessary to understand whether those resources are sensitive and need special access to use. In fact, automated tools (e.g., static and dynamic testing tools) can help

Table 5.6: Cross-reference the types of security issues identified during code review with advisories dataset for the studied projects. The values in parentheses represent the number of affected projects.

Types in Code Review	Mentioned in Advisories
ReDOS (1)	X (4)
Race Condition (3)	-
Access Control (6)	-
XSS (3)	X (3)
SQL Injection (2)	X (1)
Documentation (5)	-
Improper Authentication (2)	X (1)
Sensitive Data Exposure (5)	X (2)
Remote Code Injection (4)	X (3)
Overflow (5)	X (3)
Deadlock (2)	-
Improper Input Validation (1)	X (3)
Vulnerable Package (2)	X (4)
DDOS (1)	X (2)

to detect the absence/missing of access control in a system (Aloraini et al., 2019; *Broken Access Control — OWASP*, accessed on 12/10/2021), but cannot determine whether it functions properly when in use, which is the case for several issues identified during code review. Issues classified under "Documentation" type refer to some security constraints on the project usage. Some missing documentations may make clients (i.e., project users) use the API in unsafe manners. Hence, documenting such cases are extremely important as project users rely on this Documentation when using the project, and any missing details on security may lead users to misuse the project, causing a vulnerability in their applications. Yet, such issues are not really exploitable vulnerabilities that may affect the project itself, and hence, the "Documentation" category is not present in the advisories dataset.

While some types of security issues are frequently identified through code reviews, we find that other types are more frequently detected in the advisories dataset. For example, as seen in Table 5.6, we find that code reviews identified ReDOS in one project only (Marked). However, the advisories dataset mentions four projects (including the Marked project) affected by ReDOS, namely, `Moment`, `Uglify-js`, and `Sequqlize`. This result indicates that some other types of issues like ReDOS are easier to detect by means of tools. We observe in RQ that project maintainers

integrate a static analysis tool in the project pipeline and periodically invite a security expert to validate and fix specific issues like the ReDOS type.

5.4.2 Implications

In this section, we provide some implications to practitioners and researchers.

Security issues are raised on a small fraction of project files. Our results show that security issues identified during code review of the studied projects are very localized, only appearing in a small fraction of project PRs and files (see RQ1). For example, we find that all ReDOS issues in the Marked project are identified in one file. This indicates that project maintainers concentrate on these parts and pay more attention to them during security code review. Therefore, one way to support the code review process is to build tools that rank files based on their security sensitiveness. For instance, files that have had security issues identified in them can be flagged by the tool as security sensitive. Such tools can help project maintainers for prioritizing code review for security issues, e.g., a PR that touches a file that has been flagged as security sensitive before may require the review of a security code expert.

Code review approach is more effective to find certain types of security issues over other methods (i.e., advisory method). We found a substantial variety of security issues found through code review. However, our results also show that 4 types of security issues, i.e., issues relating to Race Conditions, Deadlocks, Access Controls and Documentation, are issues in the studied projects which were not found once in the advisories dataset. Through our manual analysis (RQ 2 & Section 5.4.1, we observe that dealing with such security issues are highly complex and hard to locate. For example, in the case of Race Conditions and Deadlocks, the reason due to these issues being difficult to identify and fix is that they require a deep understanding of how the project uses multi-threading, in order to understand whether specific implementations in the code can cause Race Conditions or Deadlocks. In the case of Access Control, this is due to the need for a solid understanding of the requirements, which is necessary to understand whether the accessed resources are sensitive and need special access to use. Unlike other types of security issues, cases

of Documentation clarity are not considered as vulnerabilities, and also are very subjective, which makes it easier to identify and improve through the code review process. That said, code review is considered a critical approach for dealing with such security issues. Hence, it would be advisable that project maintainers pay attention to such types to employ better security code review and understand what would be required to catch them and ensure that the code is free from these types. Also, the development teams should have the required expertise to solve security issues, e.g., issues related to thread synchronization stress this need as their solutions require an in-depth understanding of the problematic code.

Developers should integrate automated tools in the project development cycle to target security issues that affect their projects. Our results (RQ 2) show that in some cases, project maintainers integrate automated tools in the pipeline of the project development cycle. For example, we observe that the project maintainers of Marked integrate a static analysis tool into the project CI pipeline, called *vuln-regex-detector (*vuln-regex-detector*, accessed on 12/10/2021)*, which led to the identification of most ReDOS security issues in the project. Also, through our manual analysis, we observe that, in several cases (e.g., (*Pull Request #1414*, accessed on 12/10/2021; *Pull Request #844*, accessed on 12/10/2021)), the project enables tools for dependency management to upgrade outdated and vulnerable dependencies. For instance, in this PR (*Pull Request #1420*, accessed on 12/10/2021) of the project Marked, several outdated dependencies were automatically updated by the Snyk tool. This result indicates that it is of great help for the project to use automated tools to target security issues that may affect the project. Further research should explore different tools that can be integrated in the development cycle of the project to target security concerns. Recently, GitHub (in August 2018) has created *CodeQL* (*About CodeQL code scanning in your CI system*, accessed on 12/10/2021; *github/codeql*, accessed on 12/10/2021), a code analysis platform for finding zero-days and critical vulnerabilities in pull requests. Future research should examine the efficiency and effectiveness of such code review tools across projects for different types of security concerns. Such research is important to increase developers awareness to code review tools that can be employed in the development pipeline to identify security concerns in the project.

Overlooked security issues during code review should be better evaluated and disclosed for project users. Our findings show that a non-negligible share of issues identified during code review ended up not being fixed or are ignored by maintainers (see RQ3). We observe in several examples (e.g., (*Pull Request #1598*, accessed on 12/10/2021b; *Pull Request #16254*, accessed on 12/10/2021; *Pull Request #9224*, accessed on 12/10/2021)) that such issues generally take great effort to mitigate or may contradict the goal of the project, and the responsibility of the security issue is on the user to mitigate. However, in all these cases, no action is taken from the project maintainers to attempt to advise developers of such security issues. This can make vulnerabilities proliferate to the project users unbeknownst to them. Therefore, we recommend to all project maintainers to document all potential security issues that could come about by using their project in a way that is easy to understand and easy to access. Project users cannot be expected to sift through project history to gain a better understanding of what is their responsibility for security, and to understand what is not being handled by the project. Having some easily accessible documentation, such as in the README of a project, in the package description (like on npm registry) or on the project website can help give a high-level overview to prospective users, further allowing them to handle and mitigate such vulnerabilities in their own projects.

Code review is effective in fixing raised security issues. Our findings show that issues identified are frequently fixed, either by applying new patches, or by rejecting the proposed PRs (see RQ3). While previous studies show that project maintainers lack energy or enthusiasm to review source code for security issues (Howard, 2006), code review is a critical component of shipping secure software, given that it is much less costly to fix security issues before a new version of the project is released. Hence, project maintainers should be encouraged to consider reviewing the PR changes for security issues before merging them. While fixing issues during code review is time-consuming and may obstruct meeting the project release deadline (e.g., as stated by project maintainers (*Pull Request #16254*, accessed on 12/10/2021; *Pull Request #2375*, accessed on 12/10/2021)), project managers should support maintainers by reserving a sufficient amount of time for fixing raised security issues during code review.

5.5 Threats to Validity

Internal validity concerns factors that might affect the casual relationship and experimental bias. In RQ2 and RQ3, we conducted major manual process to extract the required information for analysing the security issues in the PRs. Like any human activity, our manual process is subject to some bias. To mitigate this, we invite an external annotator to independently analyse the PRs using an open card-sort method. Morover, we report a high-level of agreement which indicates that our results are more likely to hold. Additionally, both annotators meet and discuss any conflicts to reach a consensus. This gives us a high confidence of the data used in our study.

Our keyword-based technique to identify security-related PRs is another limitation. We may miss security issues in the PRs if the review comments do not contain any of the keywords that we used. However, our keyword set is curated in an extensive process, by utilizing a well-known set of security-related keywords, which has been used in prior studies. Then, we manually examine the relevance of each keyword and include ones that yield a good relevance (see Section 5.2.2). Hence, we believe that our keyword set is of high quality, and that the potentially missed security issues will not significantly impact our results.

Finally, in our analysis, we used the PR feature in GitHub to search for security issues raised by project maintainers during code review. However, there might be other security issues that are not discussed through PR feature. However, through our manual analysis, we did not observe a case where a project maintainer refers to an issue being discussed through other platforms. Therefore, we had to rely on the PR discussion as the main source of information for the security issues raised during code review.

External validity are related to the generalizability of our findings. Our projects dataset contains 10 JavaScript projects available in the npm advisories dataset. Hence, it is possible that there are other projects not included in our dataset, which might also be of our interest in this study. However, our projects dataset is of high quality, since we leveraged some filtration criteria to provide a good representation of the projects we are interested in studying. The projects chosen for our study include popular open-source projects that vary across domains, languages, age, and having high activity level. Also, the number of projects in our dataset is in-line with the similar studies that also

require similar manual process, given the extensive manual analysis required for the study analysis and data collection process, which makes it infeasible to include a lot of projects. Therefore, we believe most of these results can hold for other OSS projects.

5.6 Related Work

In this section, we discus the work that is related to the study in this chapter. We focus mostly on the work that approach the link between code review and security.

Some studies focused on factors that improve the code review quality. For example, Kononenko et al. (Kononenko et al., 2018) empirically examined what factors influence the pull request (PR) review quality and outcome in the Active Merchant project. They found that the quality of a PR is strongly associated with the quality of its description, its complexity and revertability, while the quality of the review process is linked to the feedback quality, tests quality, and the discussion among developers. Bernardo et at. (Bernardo, da Costa, & Kulesza, 2018) examined the impact of adopting CI on the time to integrate PRs. They found that the time to merge PRs increased after adopting CI. In the context of our study, we observed some projects adopting and integrating static analysis tools in the CI pipeline to help identifying specific security issues (e.g., ReDOS) and other general issues related to fixing the code style and structure.

A recent study by (Paul et al., 2021) built a regression model on the Chromium project to identify factors that differentiate code reviews with successfully identified vulnerabilities from reviews that missed vulnerabilities. They found, for example, that the number of directories under review correlates negatively with identifying vulnerabilities. Bosu et al. (Bosu, 2014) performed an empirical study, where they analyzed more than 400 vulnerable code changes with the aim to identify their characteristics. They found, for example, the changes by less experienced contributors were significantly more likely to introduce vulnerabilities. Also, they found that new files are less likely to contain vulnerabilities compared to frequently modified files. In our study, we find that the identified security issues are concentrated on a small fraction of the project files (RQ_1). This should encourage researchers in the future to understand the nature of such files that frequently contain security issues, which would help practitioners and developers better improve the process

of identifying security issues in code review.

In many ways, our study in this chapter complements the previous work since, we specifically focus on JavaScript projects that have been published in the npm ecosystem. We also add to previous work by studying how developers discuss the raised security issues and tackle them during the review phase. Our study aims to help the community better understand the types of security issues discovered during code review in order to pay attention to them in the future, and understand the mitigation strategies employed by project maintainers to tackle the issues. Moreover, our results highlights several important observations that aim to increase the awareness of practitioners and researchers to the role of code review in relation with security, and improve the practice of code review for spotting security issues.

5.7 Chapter Summary

This chapter conducts a study to explore the role of code review from a security perspective, by analysing 10 JavaScript open-source GitHub projects.

First, we quantify the prevalence of security issues raised in the project Pull Requests (PRs). Our manual analysis (RQ_1) identified 171 security issues, which represents a small proportion of all PRs in the studied projects. However, such issues are discussed by project maintainers at length. Between 4.82% - 28% of all comments in the 171 PRs are related specifically to the security related concern. Moreover, our manual analysis showed 14 types of security issues raised in code review (RQ_2). In particular, we observe that code review is effective at identifying certain types of security issues, e.g., Race Condition, Access Control, and ReDOS. When analysing how project maintainers respond to the raised security issues (RQ_3), we find that the majority of the identified security issues are fixed and mitigated. Yet, the project maintainers sometimes do not fix the issue, due to its technical complexity. Interestingly, sometimes the project maintainers discuss security issues that are not directly related to the reviewed PR. Finally, we present some implications for practitioners and researchers, which aim to support the role code review in enhancing project security.

The focus of this chapter is to study the role of code review from a security perspective. However, many open-source projects that adopt code review still encounter a large number of

post-release security vulnerabilities. These vulnerabilities can then impact the software project users (e.g., dependent applications). To help mitigate this issue, the community has created some mechanisms (e.g., software bots) to automatically track and fix vulnerable dependencies in project. In the next chapter, we shed the light on a popular software bot, called Dependabot, to examine its effectiveness for tackling vulnerable dependencies in software projects.

Chapter 6

Evaluating the Use of Dependabot for Patching Package Vulnerabilities

Developers struggle to keep updating and fixing vulnerable dependencies in their projects. Emerging tools attempt to address this problem by introducing bots that can automatically inform the developer of stale and vulnerable dependencies in the project. To understand whether these tools actually help developers, we evaluate the use of a popular software bot, called Dependabot, a bot that issues pull-requests (PRs) to automatically update vulnerable dependencies. We investigate a quality set of 2,904 JavaScript open-source GitHub projects that subscribed to Dependabot. Our results show that the vast majority (65.42%) of the created security-related pull requests are accepted, often merged within a day. Also, we identify 7 main reasons why security pull requests are not merged, mostly related to concurrent modifications of the affected dependencies rather than Dependabot failures. Finally, we model the time it takes to merge a Dependabot security pull-request. Our model reveals several significant features to explain merge times, e.g., projects that have relevant experience with Dependabot security pull requests are most likely to be associated with rapid merges. Our findings indicate that Dependabot can be of great help to increase awareness of developers to dependency vulnerabilities in their projects.

6.1 Introduction

Modern software systems are increasingly depending on the reuse of code from external dependencies (i.e., packages). While the use of dependencies boosts productivity (Basili, Briand, & Melo, 1996b) and software quality (Lim, 1994b), it also increases the impact of security vulnerabilities (Zimmermann et al., 2019a). A security vulnerability in a highly-used dependency may directly impact hundreds of applications, leading to significant financial costs and reputation loss. An infamous example is the Equifax cybersecurity incident in 2017, caused by a web-server vulnerability in the Apache Struts package, which led to illegal access to sensitive information of almost half of the US population (143 million citizens) (Equifax, 2017).

The open source community has taken active measures to deal with security vulnerabilities in dependencies. For example, Dependabot is a very popular GitHub bot that creates pull-requests (PRs) to help developers automatically integrate dependency updates and vulnerability fixes into their projects (*dependabot-core*, accessed on 12/10/2021). Dependabot monitors the GitHub Vulnerability Advisories dataset to identify the vulnerable dependencies of the target project. As soon as a dependency vulnerability is identified, Dependabot sends a notification through a PR that updates the vulnerable dependency version to non-vulnerable version that has fixed the security issue, and developers can simply merge the PR to adopt the suggested update. Currently, more than 6 million security and non-security related PRs have been merged in projects from 15 languages supported by Dependabot (*Dependabot*, accessed on 12/10/2021).

Previous work (Mirhosseini & Parnin, 2017) investigated to which extent dependency management tools can convince developers to upgrade out-of-date dependencies, showing that such tools are not yet widely adopted by developers. However, they focus on the general problem of outdated dependencies and do not pay particular attention to security vulnerabilities in dependencies. Given that dependency updates for vulnerability fixes have a critical impact, we specifically focus on studying a very popular dependency tool (e.g., Dependabot) at coping with security vulnerabilities in dependencies. To our best knowledge, little is known about the receptivity and level of adoption of Dependabot security PRs in real open-source software projects.

Therefore, our main goal is to understand the degree to which developers adopt Dependabot

security PRs that tackle dependency vulnerabilities in open source projects.

To achieve our goal, we perform an empirical study involving data from 15,243 Dependabot security PRs that belong to 2,904 active open-source JavaScript projects from GitHub. In the first stage of our study, we examine how often Dependabot security PRs are accepted (merged) and how long it takes to merge them **(RQ_1)**, in order to determine to what extent developers of open-source projects adopt and respond to Dependabot security PRs. We observe that the majority (65.42%) of the Dependabot security PRs in our dataset are merged, often within a day. Still, a significant minority (34.58%) of PRs are not merged.

As such, to understand the motives that led developers to not merge Dependabot security PRs, we qualitatively examine the reasons for Dependabot security PRs not being merged (**RQ_2**). Our manual analysis identifies 7 main reasons, showing that, by in large, the majority of non-merged PRs are turned-over by Dependabot itself. For example, in 50.8% of the manually studied PRs, Dependabot closes a former security PR in favor of a newer PR that updates to a newer version.

Although the majority of the PRs are merged within a day (RQ_1), we observe a non-negligible proportion of PRs that took longer to be merged. Hence, to understand what would lead to take a longer time to respond to Dependabot security PRs, we examine the features that influence the time to merge a Dependabot security PR, given that the time is crucial and that the longer a package remains affected, the longer the application that uses it will remain vulnerable to malicious users **(RQ_3)**. We observe, using our mixed-effects regression model, five highly important features to explain merge time durations of Dependabot security PRs. While some common wisdom features (e.g., the project activity and the past experience with Dependabot security PRs) are strongly associated with the timespan of the merged PRs, the severity of dependency vulnerability and the level of patch update are not.

To summarize, this work makes the following contributions:

- To the best of our knowledge, this is the first work to provide an empirical evidence for understanding developers adoption of Dependabot security automated PRs in open source projects, while also discussing the implications of our findings to practitioners and Dependabot maintainers.

- We qualitatively uncover the possible issues developers could face when adopting Dependabot PRs. Such evaluation can advance the future work, i.e., researchers can direct their efforts to identify the cause of the issues and propose solutions to overcome the limitations.
- We build a logistic regression model that could identify relative importance of various factors explaining merge times of Dependabot security PRs.

Chapter organization.

This chapter is organized as following: Section 6.2 provides a brief background about Dependabot workflow. Section 6.3 describes our study design. Section 6.4 presents the results of our study. Section 6.5 discusses how our findings lead to implications to practitioners and future research directions. Section 6.6 presents DEPCOMBINE , a proposed tool that combines Dependabot security PRs on GitHub repositories. Section 6.7 presents the threats to validity. Section 6.8 presents the related work. Section 6.9 concludes our study.

6.2 Background

In this section, we provide a background on Dependabot and its workflow for fixing vulnerable dependencies through pull-requests.

Dependabot is a popular GitHub bot in the field of dependency management (*Dependabot*, accessed on 12/10/2021) . The goal of Dependabot is to help developers maintain and protect their projects from outdated and vulnerable dependencies, by automatically alerting developers about the vulnerable release of dependencies and suggesting security updates for dependencies using Pull Requests (PRs). It supports almost all popular programming languages, such as JavaScript, Python, Ruby, Java, PHP, Elixir, etc.

Dependabot Workflow. To understand how Dependabot works for suggesting security updates for vulnerable dependencies through pull-requests, we briefly discuss the procedures Dependabot uses to find a vulnerable dependency and suggest a fix. To start using Dependabot, a project maintainer needs to enable Dependabot to inspect the project dependencies and submit PRs accordingly. To facilitate adoption, Dependabot provides a method for easy integration, where project maintainers

can sign-in and add their projects to be monitored by Dependabot. Once Dependabot is integrated in the project, its procedure for submitting security pull request is as follows:

(1) Dependabot monitors Github's Security Advisory API (GitHub, accessed on 12/10/2021) to identify security updates of various dependencies in different package ecosystems underlying different programming languages.

(2) Dependabot monitors the dependencies of the target project on a daily-basis by inspecting the dependency management files (e.g., package.json for NodeJS applications, requirements.txt for Python applications).

(3) As soon as a new vulnerability is published in the advisory API, Dependabot then verifies: i) if the monitored project depends on affected package versions; and ii) if there is a version of the package with a fix-patch for said vulnerability. If both criteria are met, Dependabot issues a PR bumping the current vulnerable version of the dependency to the closest non-vulnerable version (to reduce build breakage), by updating the dependency management file. Updating to the minimal fixed version makes that easier, i.e., there is less to review, and less chance of breaking changes. Dependabot distinguishes minor bug fixes and feature enhancements from security fixes, i.e., if the dependency update contains a security fix, the corresponding PR adds several information related to the vulnerability of the affected dependency, e.g., the PR body indicates that the update includes a security fix. Figure 6.1 shows an example of Dependabot security PR.

(4) After that, the project developer can simply upgrade the vulnerable dependencies with a single click by merging the PR or ignore and close the PR without having any effect on the project. In fact, Dependabot comes with other features in order to convince developers at accepting the PRs. For example, alongside the suggested changes, each PR contains information about the vulnerability (e.g., severity, versions affected) and the issue from the advisory report, which can help developers analyze and understand the implications of changes and consider the risks of not updating. Finally, Dependabot provides an auto-merge feature, which automatically merges Dependabot PRs. A project can enable this feature in case it uses a

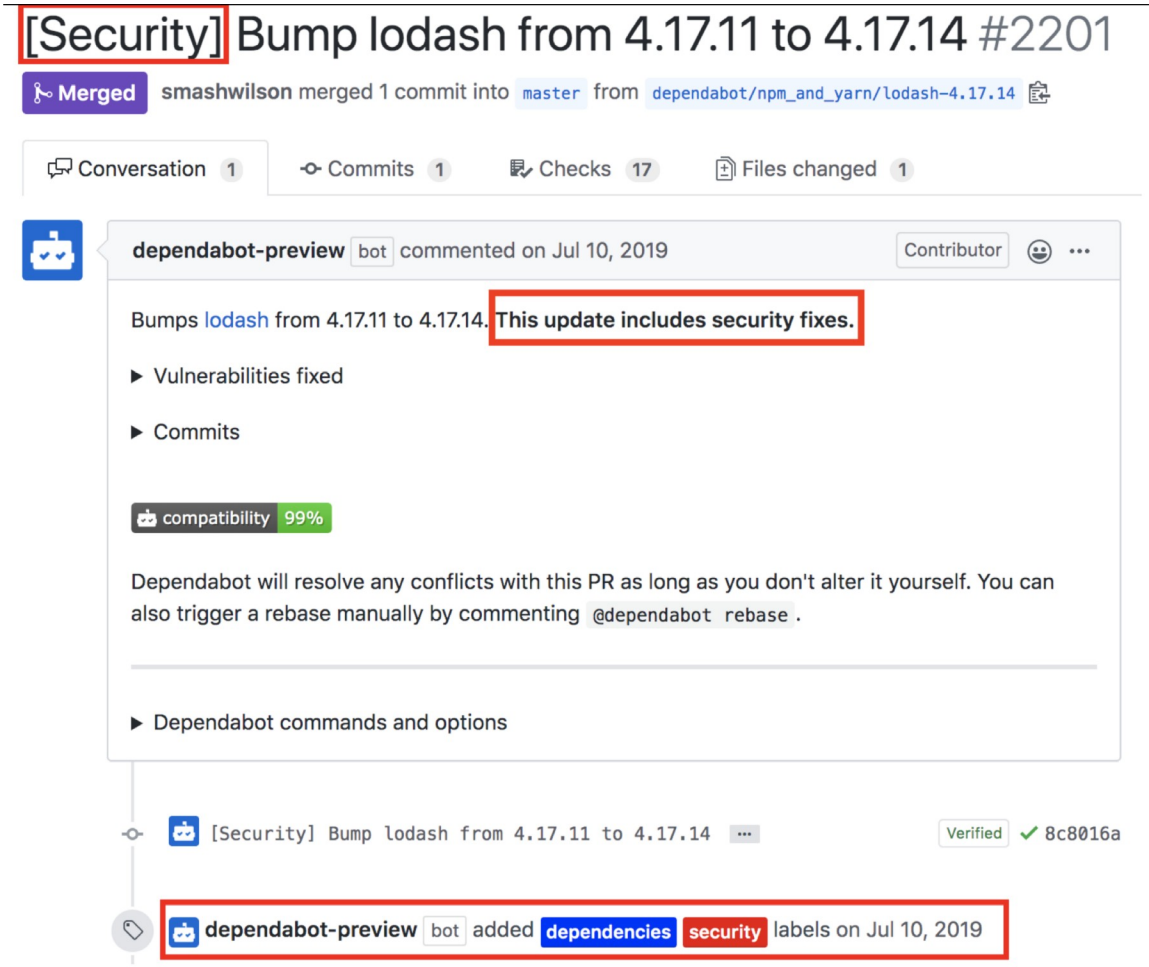


Figure 6.1: An example of Dependabot security PR.

Continuous Integration (CI) infrastructure to prevent possible breaking changes. By default no PRs are auto-merged.

6.3 Study Design

Dependabot aims to help developers automatically update their dependencies through PRs. There are numerous reasons to update a dependency, such as making the use of new features, accessing bug fix patches, etc., which led to the creation of millions of Dependabot PRs in open-source projects. Updates that include security issues fixes are among the most critical reasons developers should update their dependencies, as applications frequently depend on packages containing vulnerabilities (Snyk.io, accessed on 12/10/2021). Therefore, we focus in our work on studying

Dependabot security PRs, i.e., to what extent open source developers adopt Dependabot security PRs to help them keep their dependencies secure. Hence, we first need to identify and collect the dataset of Dependabot security PRs, and use this data to answer the following research questions:

- RQ_1: How often and how fast are Dependabot security pull requests merged?
- RQ_2: What are the reasons for Dependabot security pull requests being not merged?
- RQ_3: What factors are associated with rapid merge times?

Our study examines security PRs created by Dependabot in JavaScript projects. We chose to focus on JavaScript due to its wide popularity amongst the development community (SOF, 2020). In addition, considering the dynamic nature of JavaScript and the rapidly growing environment (with more than 1.3M packages (*Libraries.io - The Open Source Discovery Service*, accessed on 12/10/2021)), the problem of maintaining and updating dependencies is especially challenging, as evidenced by a recent survey of JavaScript developers (Abdalkareem et al., 2017). Hence, dependency management in JavaScript is challenging, which makes Dependabot effectiveness even more crucial.

To perform our study, we leverage the GitHub API to collect security PRs that were created by Dependabot for the purpose of fixing a vulnerable dependency in a JavaScript project.

Obtaining Dependabot security PRs. Dependabot distinguishes minor bug fixes and feature enhancements from security fixes, i.e., whether the dependency update contains a security fix or not. Security PRs submitted by Dependabot contain information related to the vulnerability of the affected dependency, such as the list of vulnerabilities in the security fix. Using the GitHub API, we are able to obtain security PRs by collecting PRs that are: (i) created by Dependabot; (ii) submitted to JavaScript projects in GitHub, and (iii) for the purpose of fixing a security vulnerability (i.e., the PR body refers to a security update). In total, we obtained 36,561 Dependabot security PRs from 6,853 JavaScript projects.

Project selection. It is known that GitHub contains some toy projects (Kalliamvakou et al., 2014), which are not representative of the software projects we aim to investigate. Therefore, once the dataset of Dependabot security PRs is collected, we apply some filtering criteria for selecting a set of higher-quality projects. We only include JavaScript projects that are starred, non-forked,

Table 6.1: Statistics of the 2,904 studied JavaScript projects.

Metric	Min.	Median ($\tilde{x}$)	Mean (μ)	Max.
Commits	20	153	465.7	28,486
Age (in days)	146	652	808.3	3,828
Security PRs	1	6	7.3	48

and contain more than 20 commits, as recommended by prior studies (Kalliamvakou et al., 2014; Mirhosseini & Parnin, 2017). After applying these refinement criteria, we end up with 15,243 PRs, which belong to 2,904 open-source JavaScript projects that have at least one vulnerable dependency identified by Dependabot and a security PR was already created for the purpose of fixing it. The affected dependencies contain a set of 167 distinct vulnerable packages. This set contains some popular packages, such as lodash, eslint-utils, jquery, debug, and merge.

Table 6.1 shows the descriptive statistics on the selected JavaScript projects in our dataset. Overall, the projects in our dataset have a rich development history and are long-lived projects (median of 153 commits and 652 days of development lifespan), and have received a median of 6 security PRs from Dependabot. Finally, our dataset contains Dependabot security PRs for the period between June 2017 and April 2020. Note that Dependabot launching was on May 26, 2017 (*Dependabot introduction*, accessed on 12/10/2021).

6.4 Study Results

In this section, for each RQ, we present our motivation, describe the approach used, and discuss our findings.

RQ_1: How often and how fast are Dependabot security pull requests merged?

In this RQ, we examine the degree to which open source developers are responsive to Dependabot security PRs in the studied projects. Our examination contemplates two main aspects, namely: how many Dependabot security PRs are merged (accepted)?, and how long does it take for these security PRs to be merged?.

Table 6.2: Analysis of the merged and not merged Dependabot security PRs.

Dependabot security PRs	#	%
Total	13,003	100.00%
Merged	8,506	65.42%
Not Merged	4,497	34.58%

Acceptance of Dependabot security PRs

Motivation. Given the critical problem of vulnerable dependencies in the current JavaScript landscape, we want to understand how receptive to Dependabot security PRs the open-source projects are. A high adoption rate of Dependabot security PRs indicates that developers value Dependabot contributions and agree with its assessment on the importance of updating their dependencies due to security concerns. Also, given that updating dependencies comes at the risk of breaking the project's own code, the adoption rate shows how often developers are willing to risk breaking their code to use a dependency that is free of vulnerabilities.

Approach. To examine the number of merged PRs, we need first to find the state of each PR in our dataset. PRs have three different states in GitHub: open, merged and closed (i.e., not merged). Open PRs indicate that the PR is not yet processed by developers and the decision about such PRs is not yet taken, hence they are not meaningful for this analysis and have been excluded. To identify whether the PR status is merged (accepted) or not, we extract the value of the key *merged at* timestamp that is returned from the GitHub API for each PR. For the closed (not merged) PRs, this timestamp is null, while for the merged PRs the merged at timestamp carries an actual date-time value. After that, we count the frequency of each PR state.

Results. The total number of Dependabot security PRs in our dataset after excluding the ones with *open* state is 13,003. **Of the 13,003 examined Dependabot security PRs in our dataset, 65.42% are merged.** Table 6.2 shows the proportion of each state of the Dependabot security PRs in our dataset. We observe that the majority of security PRs are merged, indicating that developers are highly receptive to Dependabot security PRs in their projects.

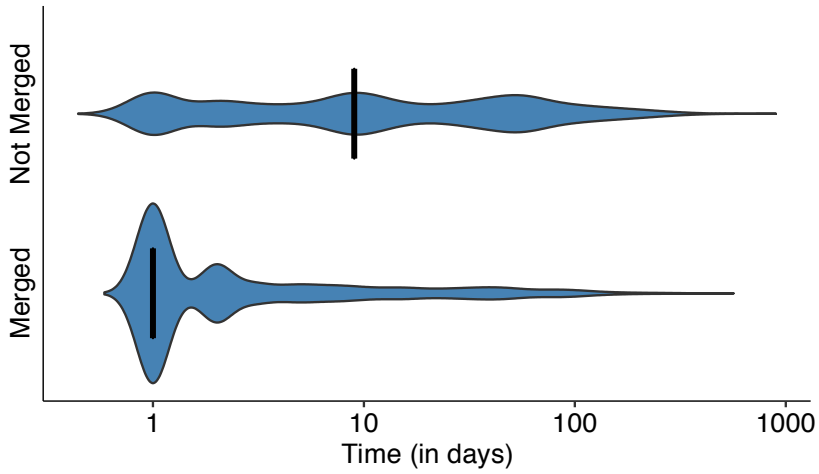


Figure 6.2: Violin-plot showing the distribution of the amount of time for Dependabot security PRs to be processed (merged and not merged). Note the logarithmic scale on the x-axis.

Lifecycle of Dependabot security PRs

Motivation. The time needed to process (merge or close) Dependabot security PRs is an important property, as the longer an application remains depending on vulnerable versions of packages, the higher the likelihood of having the vulnerability exploited by attackers. So, to advance our insights, we study whether developers are responsive at merging Dependabot security PRs, i.e., if the time that these security PRs take to be processed is as short as possible. Therefore, we investigate 1) how long does it take to *merge* a security PR since it was first created? and 2) how long does it take to *close* a security PR since it was first created?

Approach. To measure the amount of time it takes for Dependabot security PRs to be processed (merged or closed), we calculate the time difference (in days) between the creation date and the merge date for merged PRs, and the time difference between the creation date and the close date for closed PRs.

Results. Figure 6.2 presents a violin-plot containing the distribution of the amount of time for the merged and not merged (closed) security PRs, measured in days. From the Figure, we can observe that **the vast majority of the merged Dependabot security PRs are processed within one day (median = 1 day).**

Figure 6.2 also shows that the closed security PRs tend to take longer time to process than the merged ones, i.e., on median, the closed security PRs took 8 days before being closed. Comparing the merged and closed security PRs using the unpaired Mann Whitney test (Mann & Whitney, 1947) shows that this difference is statistically significant (p-value = 2.2e-16), with an effect size (Cliff's: 0.48) for the differences between merged and closed PRs, which is a large size of the effect. This ensures that Dependabot security PRs are either processed and merged fast or left to linger before they are closed without being merged.

The majority (65.42%) of Dependabot security PRs are merged and integrated in the projects, often within a day. Non-merged Dependabot security PRs take, on median, 8 days to be closed.

RQ_2: What are the reasons for Dependabot security pull requests being not merged?

Motivation. While most PRs are merged (as shown in RQ_1), a non-negligible share (34.58%) of the PRs are closed (not merged) in the studied projects. It is crucial to understand why such PRs are not merged, to grasp the motives that led developers to dismiss them, especially because such security-related PRs are meant to free open-source projects from known vulnerabilities. In turn, this can be used to motivate improvements at Dependabot, with the aim of increasing its effectiveness. Therefore, we examine why some PRs are not merged, by performing an in-depth manual analysis.

Approach. To find out why Dependabot security PRs are not merged in our dataset, we qualitatively examine them based on the discussion and reviews associated with these PRs. We collect the discussion and review comments related to each closed PR. Out of overall closed PRs (4,497), 1.27% have no discussion or review comments on them, hence, we exclude them from our analysis since it is very hard to judge such PRs without any extra information. We manually inspected all remaining closed PRs (4,440) by looking at the discussion and review comments to determine the reason for the closing, and (if possible) summarize the reason for not merging the PR into one

Table 6.3: The manually extracted reasons for not merging Dependabot security PRs.

ID	Reason	Description	%	% Closed by	
				Dependabot	Others
R1	Superseded	A newer PR contains a newer fix version of the affected dependency	50.8%	49.74%	1.06%
R2	Up to date	The affected dependency is already updated	30.1%	30.1%	-
R3	No longer a dependency	The affected dependency is removed	6.6%	6.6%	-
R4	No longer updatable	The affected dependency has a peer requirement on another dependency	6.4%	6.4%	-
R5	Tests	Tests run failed	3.2%	-	3.2%
R6	Errors	Incorrect implementation for handling the dependency fix in the PR	1.4%	1.4%	-
R7	Quality Requirement	The PR does not comply to the project standards for handling the PRs	1.1%	-	1.1%
R8	Unknown	The PR could not be classified due to lack of information in the discussion	0.4%	-	0.4%

sentence. Through this manual analysis, we identified 7 different groups of reasons for the PRs not being merged.

To alleviate the potential bias due to our manual classification for these PRs, we obtain a statistically significant sample of 354 PRs (of the 4,440 PRs) with 95% confidence level and 5% confidence interval. Then, we invite an external annotator to independently examine the 354 PRs. Note that the number of comments that span over the discussion of the closed PRs is two, on median, which makes the manual inspection indeed feasible.

To evaluate the agreement between the two annotators, we used Cohen's Kappa coefficient (Cohen, 1960), which is a well-known statistic that measures the inter-rater agreement level for categorical scales, and takes into consideration the possibility of the agreement occurring by chance. In our categorization of the manually extracted reasons, the level of agreement between the two annotators was of +0.96, which is considered to be an excellent agreement (Fleiss & Cohen, 1973).

Results. Table 6.3 summarizes why Dependabot security PRs are not merged, identified by our manual analysis. Below, we provide more details about each reason.

- **R1. PR is superseded by another newer PR (50.8%):** This is the most common reason for not merging a Dependabot security PR. In this case, the PR is closed because another Dependabot security PR updates the affected dependency to even a newer version that contains fixes to other problems but not necessarily a new vulnerability. In such cases, Dependabot itself closes the former PR in favor of the new and more up-to-date PR. Figure 6.3 shows an example of R1 (*Pull Request #91*, accessed on 12/10/2021). A few cases (1.06%) of superseded PRs are closed by project maintainers where they close a set of PRs and create a single PR that combines all of the

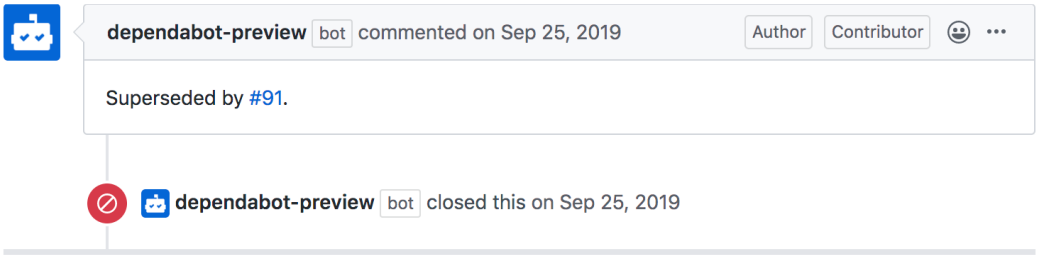


Figure 6.3: Example of Dependabot PR closed for being superseded by another Dependabot PR (R1).

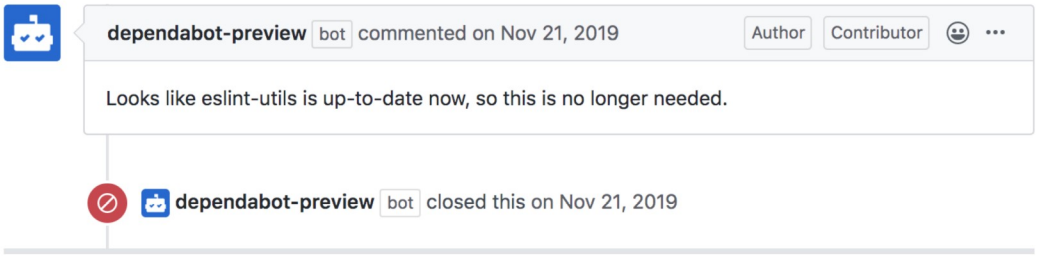


Figure 6.4: Example of Dependabot PR closed because the dependency was already updated (R2).

changes (*Pull Request #245*, accessed on 12/10/2021).

- **R2. PR is not merged because the update was applied manually on the dependency file (30.1%)**: Dependabot detects that the fixed version has been applied on the dependency file, hence, it closes the corresponding PR. Figure 6.4 shows an example of R2 where Dependabot closed the PR that fixes the vulnerable version of the dependency *eslint-utils*. We manually searched for the commit that applied the same fix suggested by Dependabot. In this commit (*steelbrain/babel-cli@c8c9859*, accessed on 12/10/2021), we observe that the same fix version, suggested by Dependabot, was manually added through the commit that also has the same date as the closed PR date (*Pull Request #85*, accessed on 12/10/2021).

- **R3. PR is not merged because the affected dependency is removed and no longer exists in the project (6.6%)**: Dependabot will close a PR once the corresponding vulnerable dependency is removed from the project, and hence, the PR is no longer needed (*Pull Request #33*, accessed on 12/10/2021).

- **R4. PR is not merged due to a peer dependency requirement (6.4%).** Another reason Dependabot closes a PR is when there is a peer requirement between the affected dependency and

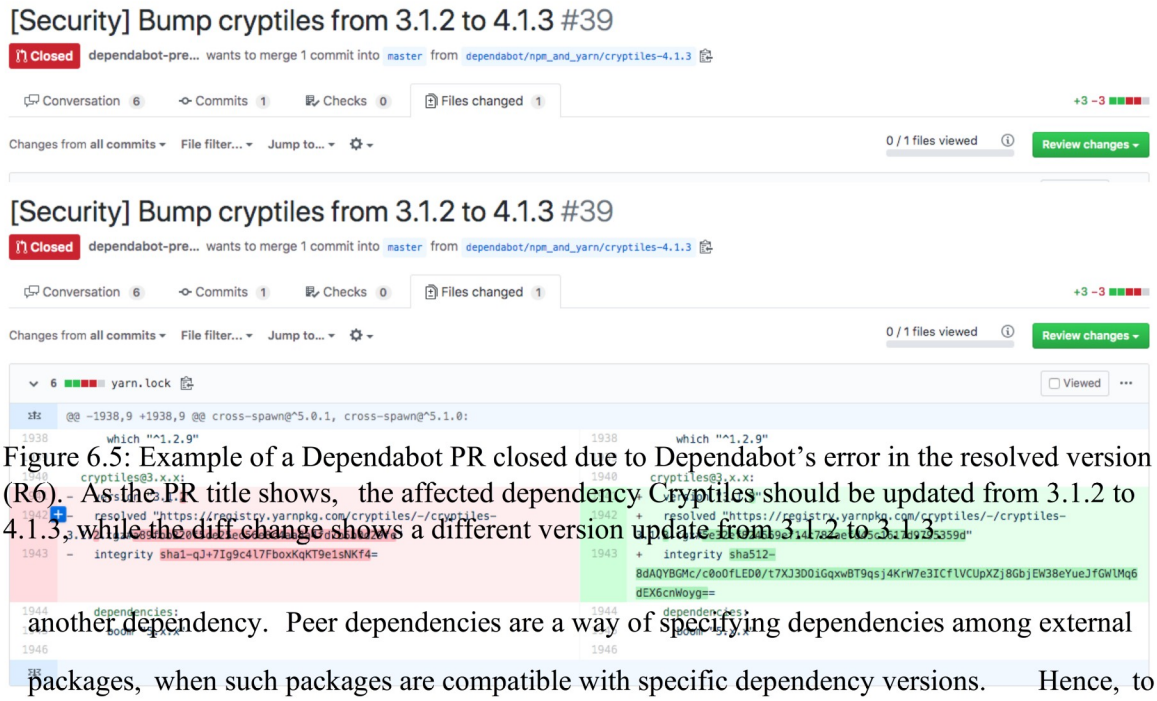


Figure 6.5: Example of a Dependabot PR closed due to Dependabot's error in the resolved version (R6). As the PR title shows, the affected dependency Cryptiles should be updated from 3.1.2 to 4.1.3, while the diff change shows a different version update from 3.1.2 to 3.1.3.

another dependency. Peer dependencies are a way of specifying dependencies among external packages, when such packages are compatible with specific dependency versions. Hence, to update/fix an affected dependency, its peer dependency should also be updated, which may lead to version conflicts (*Pull Request #3*, accessed on 12/10/2021). For example, if the dependency *eslint-config-airbnb@16.1.0* have a peer requirement on *eslint@^4.9.0*, so it is required to update this (eslint) until *eslint-config-airbnb* is updated. In such cases, Dependabot opens a PR to update *eslint-config-airbnb* but later it closes the PR due to having the peer dependency. We found a Github issue in Dependabot repository itself about this problem (*Issue #1138*, accessed on 12/10/2021), however, the problem seems not yet properly resolved by Dependabot according to the issue discussion.

- **R5. PR is not merged due to test failures (3.2%).** In such cases, the PR is closed after automated tests have failed during the CI pipeline run (*Pull Request #105*, accessed on 12/10/2021). For example, after the Travis tests have failed in this PR (*Job #231.2 - Travis CI*, accessed on 12/10/2021), the project maintainer closed the PR. When the project maintainer closes the PR, Dependabot will stop notifying the project about the current affected dependency version, however, it opens a new PR when a new fix version of the affected dependency is available.

- **R6. Error in Dependabot (1.4%).** We found cases where the submitted PRs were opened, however, they do not perform the correct fix update, and hence, Dependabot closed such PRs (*Pull Request #39*, accessed on 12/10/2021). For example, in Figure 6.5 we can notice from the PR title that the affected dependency *cryptiles* should be updated from the vulnerable version 3.1.3 to the fixed version 4.1.3. However, Dependabot was not able to resolve the dependency

to the fixed version, i.e., the PR commit changes show a different version update than the one should be. This issue is caused by the challenge of resolving dependency conflicts of transitive dependencies. Consider an application that depends on package A, and package A (transitively) depends on package B. Package A has a version constraint for depending on package B (^**1**.0.0) which contains a vulnerability, and the vulnerability was only fixed in another major version (e.g., **2**.0.1) of package B. In this case, Dependabot cannot find a version of package B that complies with the requirement of package A and is not vulnerable. This type of issues render the R6 reason. The has now been fixed by Dependabot maintainers (*JS: Handle version resolution*, accessed on 12/10/2021).

- **R7. PR does not comply with the project standards for handling PRs (1.1%).** A small share of open-source projects specifies what is called Contributor License Agreement (CLA) that should be signed by the contributor before merging the corresponding PR. In such projects, developers tend to close the Dependabot security PR after it is submitted. To gain more insights about whether such PRs may be still useful for the projects (e.g., project maintainers may manually adopt and apply the dependency fix suggested by the Dependabot PR), we manually analyze a sample of such PRs. In particular, we perform our analysis on 15 PRs of a popular and very active project namely *box/box-ui-elements* (*box/box-ui-elements*, accessed on 12/10/2021). We could find 8 Dependabot PRs that are manually applied to the dependency file by a project maintainer even after closing the Dependabot PRs. For example, in this PR (*Pull Request #1521*, accessed on 12/10/2021), Dependabot suggests updating the vulnerable dependency *atob* from version 2.0.3 to 2.1.2. Although the project maintainer closed the PR, we find that the same dependency update was actually applied as shown in this commit (*Upgrades most dev dependencies (#1753)*, accessed on 12/10/2021), probably to circumvent the licensing issue. One way to overcome the issue of legal side of contributions (i.e., contributor license agreement (CLA) requirements) is to white-list Dependabot in the CLA checker. Some CLA providers (e.g., cla-assistant (cla assitant, accessed on 12/10/2021)) allow to white-list specific contributions to a repository.

- **R8. Unknown (0.4%).** In a small minority of cases (0.4%), we could not identify the reason of not merging a Dependabot PR because its discussion and comments provided insufficient

information relevant to closing the PR (*Pull Request #860*, accessed on 12/10/2021).

Overall, the vast majority of the examined PRs (93.9%) are not merged due to four primary reasons related to concurrent modifications of dependencies: **R1 (superseded), R2 (already up-to-date), R3 (no longer a dependency), R4 (no longer updatable)**. Approximately 4% of the PRs are not merged by project maintainers due to factors related to the project process and quality specifications (testing, license agreement). Only 1.4% of the PRs are not merged due to technical errors in Dependabot. Finally, note that the reasons mentioned above are not strictly related to security-related Dependabot PRs.

The large majority of the closed Dependabot security PRs (93.9%) are turned over by Dependabot due to concurrent modifications of the affected dependencies. Approximately 4.3% of the non-merged PRs are closed by developers due to a specific project's process. Only 1.4% are not merged due to technical issues with Dependabot.

RQ_3: What factors are associated with rapid merge times?

Motivation. While most of the merged Dependabot PRs are accepted and integrated within one day (RQ1), there is a sizeable proportion (32.82%) of the merged PRs which took much longer time to be merged. The time taken to handle a Dependabot security PR is crucial given that a quick fix is the only weapon at developers disposal for minimising the risk of the application being affected by external vulnerable dependencies. For example, Heartbleed, a security vulnerability in OpenSSL package, is perhaps the most infamous example. It was introduced in 2012 and remained uncovered until April 2014. After its disclosure, researchers found more than 692 different sources of attacks attempting to exploit the vulnerability in applications that used the OpenSSL package (Durumeric et al., 2014; *Heartbleed flaw*, accessed on 12/10/2021). Hence, in this RQ we aim to study the features that are highly important and associated with the merge time of a Dependabot security PR. Doing so is important to gain understanding of why developers take so long to merge a Dependabot

PR that fixes some publicly discovered vulnerabilities.

Approach. Our goal is to study the most important features associated with the merge time of a Dependabot security PR. In particular, we aim to understand the features that are associated with *rapid* merge times. To that aim, we perform a logistics regression analysis that can discriminate whether a Dependabot security PR is merged rapidly or not. Therefore, we first classify merge times into rapid vs. not-rapid. We determine a threshold that discriminates the PRs merge times in our dataset into rapid vs. not-rapid merge times, by evaluating the merge time distribution of the PRs. We find the third quantile (4 days) to be an appropriate threshold. Note that, influenced by prior studies (Ghaleb, Da Costa, & Zou, 2019; Vasilescu et al., 2016), we perform several scenarios for choosing our threshold, i.e., we experimented with different segmentation thresholds (lower quantile, median, upper quantile). For each scenario, we measure the logistics model performance using R-squared (R^2) metric (Nakagawa & Schielzeth, 2013). We use the threshold obtained by the top performing modelling scenario (i.e., the upper quantile). That said, 6,546 PRs belong to the lower 75% of the data points (i.e., those are rapid PR merge times), whereas 1,960 PRs belong to the upper 25% of the data point (i.e., those are not-rapid PR merge times).

To conduct our logistic regression model we first collect a set of features by reviewing the related research on the pull-based software development modelling. Then, we conduct correlation and redundancy analyses to remove highly correlated features because the existence of correlated and redundant features can affect regression models (Domingos, 2012). Finally, we fit a generalized mixed-effects model for logistic regression. These steps are detailed in the following paragraphs.

(i) Features Selection. To determine our set of features, we consult the related literature on the field of pull-based development model, e.g., areas of patch submission and acceptance (Gousios, Pinzger, & Deursen, 2014; Weißgerber, Neu, & Diehl, 2008), code reviewing (Rigby & Bird, 2013), and also dependency vulnerability analysis (Bogart, Kästner, & Herbsleb, 2015; Bogart et al., 2016). The initial list of computed features (described in Table 6.4) comprises features that span over three main dimensions as follows:

PR features. These features attempt to capture the influence of Dependabot security PR characteristics on the merge time. For example, the size of the patch in the PR could affect the merge

Table 6.4: The 15 features selected to model the time to merge Dependabot security PRs.

Feature Name	Data Type	Description
PR Features		
changed lines	Numeric	Number of lines changed (added + deleted) in the dependency file by Dependabot PR
auto merge	Category	Status of auto-merge method for Dependabot PR. Binary value: True or False
Project Features		
sloc	Numeric	Number of executable source lines of code in the project at Dependabot PR creation time
team size	Numeric	Number of the active team members in the project at the PR creation time
num_submitted PRs	Numeric	Number of submitted Dependabot security PRs to the project at the PR creation time
num_accepted PRs *	Numeric	Number of accepted Dependabot security PRs in the project at the PR creation time
perc accepted PRs	Numeric	Percentage of merged Dependabot security PRs in the project at the PR creation time
num_dependencies	Numeric	Number of total proejct dependencies at the PR creation time
num_recent commits	Numeric	Number of commits in the project during the last month prior to the PR creation time
age (days)	Numeric	Project age at Dependabot PR creation time (i.e., the time interval between project creation time and Dependabot PR creation time)
total commits *	Numeric	Number of total project commits at the PR creation time
num_issues	Numeric	Number of total project issues at the PR creation time
num_PRs	Numeric	Number of total project PRs at the PR creation time
Vulnerability Patch Features		
severity	Category	Severity of the vulnerability in the affected dependency (Critical, High, Moderate, Low) associated with the Dependabot PR
patch level	Category	Patch level of the dependency update (Major, Minor, Patch) associated with the Dependabot PR

* Features removed after further step-wise feature selection (e.g., correlation).

time (Gousios et al., 2014; Weißgerber et al., 2008), i.e., the time needed to examine an external contribution could vary depending on the size of the contribution. Dependabot security PRs have varying size depending on the updated dependencies, such as the number of lines being changed (chanegd lines) in the dependency file, which may affect the time it takes for developers to review and validate the applied changes. In fact, Dependabot triggers one security PR for each direct vulnerable dependency, by default. However, if the direct vulnerable dependency requires transitive dependencies that are also vulnerable, Dependabot applies additional changes in the same PR, increasing the impact of the changes, e.g., there is more risk in breaking changes when transitive dependencies are vulnerable. Another PR feature is the auto-merge . Dependabot provides an auto-merge feature, which automatically merges Dependabot PRs. A project can enable this feature in case it uses a Continuous Integration (CI) infrastructure to prevent possible breaking

changes. By default, no PRs are auto-merged. Note that we assign the `auto_merge` as a PR feature, as it can be enabled/disabled during the project lifecycle. Also, enabling the auto-merge feature does not assure that the PR will be merged instantly, given that Dependabot will only merge the PR if the CI tests pass without issues.

Project features. Project features quantify how responsive to Dependabot security PRs the project is. Essentially, such features capture how open the project is to accepting such PRs and its past experience with Dependabot security PRs, by quantifying past Dependabot merged security PRs (e.g., perc accepted PRs). Other common wisdom features that can explain the merge time are related to the project size (e.g., sloc , team size) and maturity of the project (e.g., age). We obtain the project features from previous studies in the field of PR acceptance, as the majority of these features have been successfully used in prior studies (Gousios et al., 2014; Rahman & Roy, 2014; Rigby & Bird, 2013; Soares, de Lima J únior, Murta, & Plastino, 2015; Yu, Wang, Filkov, Devanbu, & Vasilescu, 2015) to explain the merge time of a PR.

Vulnerability patch features. The vulnerability patch features quantify the characteristic of the suggested dependency update in the Dependabot PR. There are three main levels of dependency update: 1) patch release indicates backward compatible bug fixes, 2) minor release indicates backwards compatible new features and 3) major release informs developers of backwards incompatible changes in the package release. Therefore, dependency updates that happens at the patch and minor levels are most likely to have minimal impact on the project and can be merged faster by developers. The opposite will happen in updates that bump the dependency to a major release, which have a higher risk of breaking changes and thus may take longer to be merged (Bogart, Kästner, & Herbsleb, 2015; Bogart et al., 2016). Additionally, the severity of the dependency vulnerability is another feature to explain the project response to a security PR (Decan, Mens, & Constantinou, 2018b). Dependabot builds upon GitHub Advisory dataset (GitHub, 2017) to provide a severity level of a dependency vulnerability (i.e., Critical, High, Moderate, and Low).

(ii) Correlation and Redundancy Analysis. The initial list of features included 15 features, shown in Table 6.4. To make sure that our selected features are not correlated, which could distort their importance in the model, we conduct a pairwise correlation analysis. Specifically, we use the Spearman rank correlation $|\rho|$ metric (Sarle, 1990). A pair of features that have a correlation

of $|\rho| \geq 0.7$ should have one of the features removed. We remove 2 features using that cut-off, namely, num_accepted PRs , total commits .

Furthermore, we perform RDA (redundancy analysis) on the remaining 13 non-correlated features. A feature can be redundant if it can be modelled using the other independent features. That said, we should eliminate independent features that can be estimated with an $R^2 >= 0.9$ (Harrell Jr, 2015). We observe no redundant features found in the remaining 13 features.

Table 6.4 shows the final 13 selected features (the features without * sign) along with their data type and description. Since the original distributions for most of the features were on different scales, we decided to re-scale the data (standardization scaler) before using them in the model.

(iii) Statistical Analysis. Since our dataset contains PRs from different projects (i.e., PRs merging times vary from one project to the next), we use the generalized mixed-effects logistic model to control the variation between projects. Mixed-effects logistic model, unlike the traditional logistic model, can model the individual differences between the projects by assigning (and estimating) a different intercept value for each project (Lewis, 2009; Winter, 2013). This allows to capture a project-to-project variability in the dependent feature (PR merge time). We use the *glmer* function in the *lme4* R package to conduct a mixed-effects logistic model.

To evaluate the fitness of our model, we use the R-squared metric for generalized mixed-effects models (Nakagawa & Schielzeth, 2013), which describes the proportion of variance considering the *project* variable effect. Also, to measure the explanatory power of the features in the model, and influenced by prior studies (Ghaleb et al., 2019; Ruangwan, Thongtanunam, Ihara, & Matsumoto, 2019), we use χ^2 (Chi-Squared). The value of χ^2 indicates whether the model is statistically different from the same model in the absence of a given independent variable according to the degrees of freedom in the model. The higher the χ^2 value, the greater the explanatory power of the feature in distinguishing rapid PR merge times.

Results. Our mixed-effects logistic model achieves a good performance of discriminating the rapid PR merge times of Dependabot security PRs using our determined threshold. The model fits the data well; it explains 67% of the variability in the data (PR merge times) when considering the project variable ($R^2 = 0.67$); and 22% when only considering the independent features without

Table 6.5: Results of the mixed-effects logistic model - sorted by X^2 in descending order.

Feature	Coef.	X^2	p-value	Sign.[+]
perc_accepted_PRs	0.63	88.46	< 0.001	***
auto_merge (TRUE)	1.32	64.57	< 0.001	***
num_recent_commits	0.71	32.20	< 0.001	***
num_dependencies	-0.23	7.83	0.005	**
age	0.17	4.32	0.037	*
sloc	-0.09	2.94	0.086	
severity	-	1.49	0.683	
patch_level	-	1.06	0.588	
num_PRs	-0.11	0.55	0.459	
num_submitted_PRs	0.03	0.25	0.617	
changed_lines	0.02	0.17	0.681	
num_issues	-0.05	0.12	0.733	
team_size	0.04	0.09	0.765	

[+] Significance codes: 0 '***' 0.001 '**' 0.01 '*' 0.05

the project variable, showing that the mixed-effect model is more effective at modelling time to merge PRs across different projects.

Table 6.5 presents the findings of the features importance derived from the mixed-effects model. All the independent features are ordered on the basis of their explanatory power (X^2). With each independent feature, we report its estimated coefficient, its explanatory power (X^2), its p-value, and its statistical significance code (using asterisks) to model the rapid PR merge times. Our results reveal 5 important features to have a strong association with the time to merge a Dependabot security PR. **The top three features are: (1) the percentage of past accepted Dependabot security PRs in the project, (2) the adoption of the auto-merge feature, (3) the level of project activity prior to the PR creation time.** Next, we explain the important features derived from our model.

As shown in Table 6.5, we observe several features that led to merge the Dependabot security PRs fast. For example, the past experience of the project with Dependabot is a major feature that have a strong association with the PR merge time. Projects that have had success in accepting and merging security Dependabot PRs in the past are more likely to merge Dependabot PRs faster in the future, as indicated by the positive coefficient of the perc accepted PRs in Table 6.5. This also shows that projects that have experienced issues in the past are less inclined to merge the PRs without its due investigation, which may explain the long PR processing time.

Also, enabling the auto-merge feature is strongly associated with merging the PRs rapidly. Other highly important features are related to the project activity. For example, the level of project activity, denoted by num_recent commits, has a strong association with rapid merges, i.e., the model indicates that the more active the project, the more likely a Dependabot security PR will be merged within 4 days. Moreover, our model shows that the project age is another important feature that explains the rapid PR merges, although to a lower degree. Projects that have been in development for years are more likely to merge Dependabot PRs within 4 days, as opposed to more recent projects.

Furthermore, we observe that the number of project dependencies (num_dependencies) is a feature that correlates with Dependabot security PRs being merged in more than 4 days. This indicates that projects with a high number of dependencies tend to take longer to merge a Dependabot security PR. Projects with many dependencies are more susceptible to dependency vulnerabilities (Gkortzis, Feitosa, & Spinellis, 2020), which may lead to an overwhelmingly high number of Dependabot security PRs, taking much longer for developers to address all updates.

Finally, it is also surprising to note that some observed features, such as the vulnerability severity and the dependency patch-level, do not play a significant role in how rapid a Dependabot security PR will be merged. This shows that developers do not necessarily prioritize Dependabot security PRs depending on the severity of the vulnerability or the likelihood of a breaking change (patch level).

The rapid merge time of Dependabot security PRs is directly associated with the project activity level, the project past experience with Dependabot security PRs and the adoption of the auto-merge feature. In contrast, a project with a high number of dependencies is more likely to take longer to process the merges. Surprisingly, neither the severity of the vulnerability nor the risk of breaking changes (patch level) seems to significantly influence the PR merge time.

6.5 Implications

In this section, we discuss implications of our findings to practitioners (Section 6.5.1) and Dependabot (Section 6.5.2).

6.5.1 Implications to practitioners

Open-source JavaScript projects are highly receptive to Dependabot security PRs. Our results (RQ_1 & RQ_2) show that a large proportion (65.42%) of the Dependabot security PRs are accepted, and 50.8% of the closed (not merged) security PRs are not triggered by developers, but rather by Dependabot itself in favor of more updated dependency versions. The high level of acceptance of Dependabot security PRs indicates that developers are willing to trust external automated tools for important preventive tasks (security dependency updates), given that the tool provides sufficient information for developers to decide. That said, developers should use Dependabot not just to make their dependencies up-to-date but also to keep them secure and vulnerability-free. Dependabot can be seen as a success case to be replicated by tools that assist developers on a variety of tasks like security updates through PRs.

Developers are encouraged to enable the auto-merge feature for improving the merging time of Dependabot security PRs. Our results (RQ_2) show that more than half (50.8%) of the non-merged PRs in our dataset are superseded for not being merged on time. Therefore, we recommend maintainers to review and respond to security updates as quickly as possible to avoid being affected by publicly known vulnerabilities. One way to achieve that is by using the auto-merge feature. Our model (RQ_3) shows the importance of the auto-merge feature. Additionally, our results show that the security dependency updates of Dependabot rarely break the tests in the CI pipeline (3.2%), given the fact that Dependabot issues a PR bumping the current vulnerable version of the dependency to the closest (minimum) non-vulnerable version (to reduce the likelihood of build breakage) (*Dependabot security updates*, accessed on 12/10/2021). In fact, projects can configure the auto-merge feature to be only enabled for security PRs (*Dependabot Schedule*, accessed on 12/10/2021). That said, developers are better off setting a CI pipeline to automatically merge Dependabot security PRs,

particularly in projects that are not in active development or suffer from lack of resources.

6.5.2 Implications to Dependabot maintainers

Dependabot needs to properly handle peer dependencies. Our results (RQ_2) show that 6.4% of the closed security PRs are accidentally closed by Dependabot when there is a peer dependency. If a vulnerable dependency A has a peer dependency to B (i.e., the semantic version of the dependency A allows only specific versions to be compatible with the dependency B), creating a PR to update the dependency A would produce version conflicts, effectively leading Dependabot to close the PR after opening it. In such cases, to avoid version conflicts for the peer dependency, the dependency B in the previous example should be updated prior, to be compatible with the new version update of the vulnerable dependency A. At current stage, Dependabot is not fully able to handle such peer dependency updates (*Issue #1138*, accessed on 12/10/2021). Therefore, and given that security updates are essential, Dependabot should find a mechanism to be able to resolve the version conflicts among the peer dependencies in the target project, by updating them to compatible versions.

Dependabot needs to be more efficient for projects with a high number of dependencies. Our model (RQ_3) pinpoints the num_dependencies as one of the significant factors for taking a long time to merge a security Dependabot PR. In fact, we have seen several cases (e.g., (*Pull Request #250*, accessed on 12/10/2021; *Pull Request #49*, accessed on 12/10/2021)) where developers have manually consolidated multiple Dependabot PRs into a single PR, to only then update the dependencies all at once. Dependabot can be more efficient by providing ways of grouping PRs to reduce security notification fatigue in large projects. Also, such feature would be more essential in case multiple dependencies need to be updated at the same time or they can break the application.

Dependabot needs to prioritize security updates by more fine-grained analysis of dependency vulnerabilities. Currently, Dependabot provides the maintainers with a way to prioritize receiving notifications for Dependabot security PRs (*Configuring notifications*, accessed on 12/10/2021). This is done by using the vulnerability severity level of the suggested security updates. Our model (RQ_3) shows that the security PRs are treated independently of the severity level, indicating a need

Figure 6.6: Screen-shot of the DEPCOMBINE website showing its main interface (*DepCombine — Home*, 2021).

for a better way of prioritization. A potential improvement to Dependabot is to give priority to updates where the vulnerability part of the dependency is actively used by the project's code. This is admitedly difficult, particularly in dynamically typed languages such as JavaScript, but a conservative approximation can be used to hint developers they need to act fast. Techniques discussed in the literature might be used to achieve this fine-grained prioritization, e.g., SAP organization had recently created a tool that applies static and dynamic analyses to detect and mitigate the use of vulnerable dependencies at the code-level (Chinthanet et al., 2020; Ponta et al., 2020).

6.6 Tool Support: Dep-Combine

The implications of our work suggests ways in which Dependabot notifications can be improved to better align with developers' need, e.g., to reduce notification fatigue and improve developers productivity. This is more critical for Dependabot security updates, given that such updates should be merged in the project as short as possible. In our work, we have seen several cases (e.g., (*Pull Request #250*, accessed on 12/10/2021; *Pull Request #49*, accessed on 12/10/2021)) where developers have manually consolidated multiple Dependabot PRs into a single PR, to only then update the dependencies all at once. To address this problem, we build a tool called DEPCOMBINE , which combines Dependabot security PRs that are open in a GitHub repository, by creating a new single PR that merges all Dependabot PRs together. Figure 6.6 shows a screen-shot of DEPCOMBINE 's interface. The tool is publicly available through this website (*DepCombine — Home*, 2021). Next, we briefly describe the workflow of DEPCOMBINE using an example.

First, the user provides a Github repository link, as shown in Figure 6.6. We assume that the

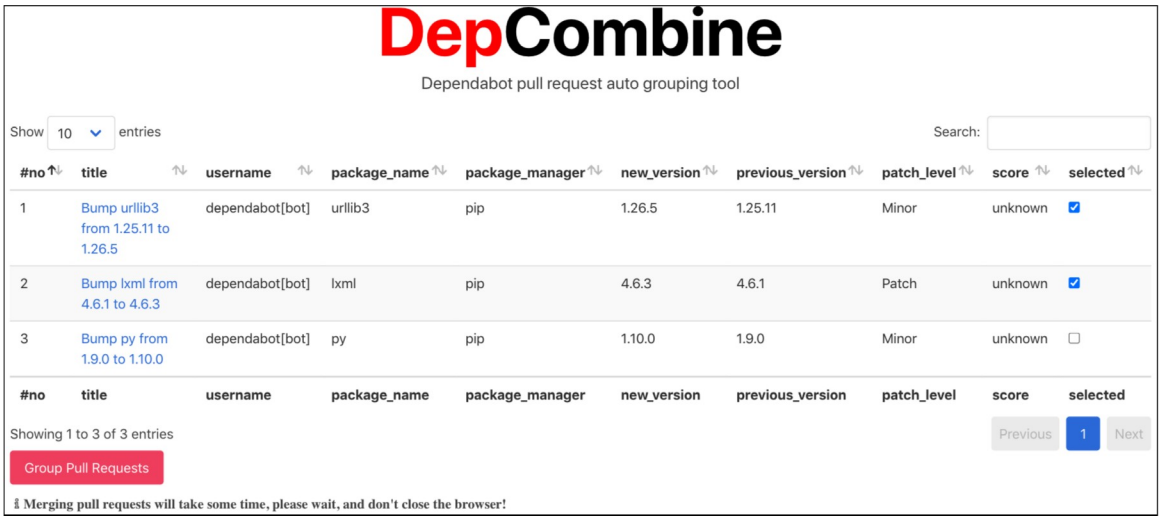


Figure 6.7: Screen-shot of the DEPCOMBINE website showing the fetched Dependabot pull requests.

analysed repository have already integrated Dependabot for dependency management. Once the user press "Fetch Pull Requests" button, DEPCOMBINE will fetch all the open Dependabot security pull requests and list them in a table, as shown in Figure 6.7.

As shown in Figure 6.7, the tool allows the user to view some information related to the PR, i.e., the user can search for keywords, package name, package manager, patch level, compatability scores of the update, etc, to filter the pull requests based on the search criteria. The users can then pick any pull requests in any order for the grouping (using check boxes shown under the column "selected"). Once satisfied, when the user presses "Group Pull Requests", DEPCOMBINE will create a new PR that combines all the changes of the previously selected PRs, as shown in Figure 6.8. The Figure shows a single PR created in the analyzed repository, and the PR combines the changes of the two previously selected PRs from Dependabot. Note that the PR author (i.e., malfadel) is a GitHub test user that we created for the purpose of testing the tool.

Finally, note that DEPCOMBINE also considers specific cases where a PR from the tool is already created in the repository but not yet merged, and the user would like to perform another grouping activity using the tool. For example, assume that the user already created a grouped pull request which has not been merged in the repository yet. If the user creates a new pull request using DEPCOMBINE , the tool will only update the un-merged (open) PR with the latest changes, without

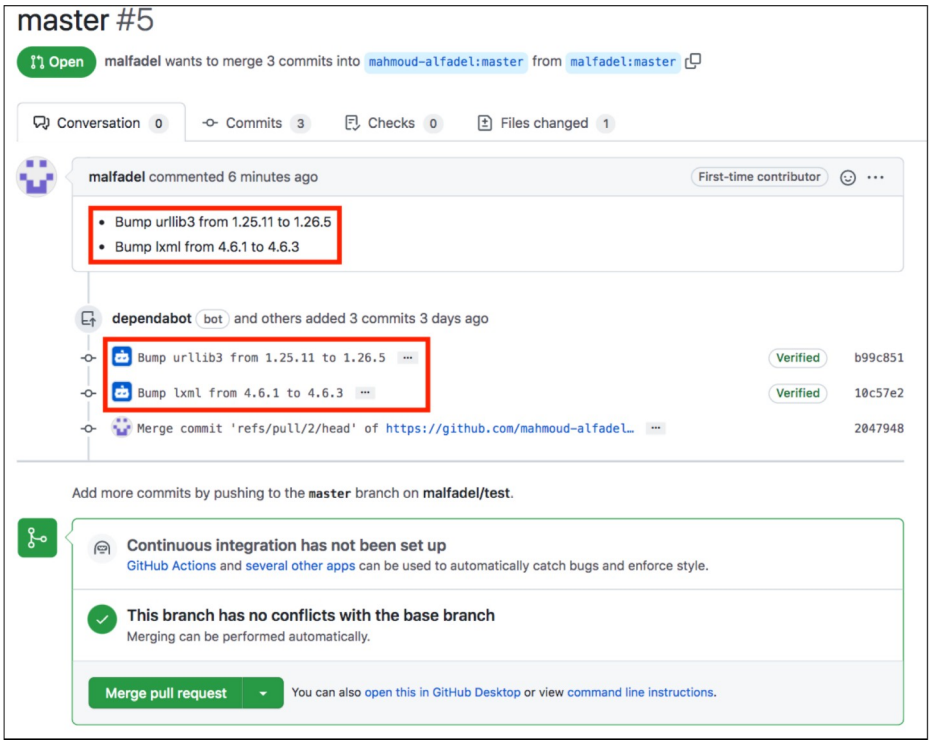


Figure 6.8: Screen-shot showing the new PR created by the tool D EPCOMBINE , which combines the selected PRs in the analysed repository (*Pull Request #5 by mahmoud-alfadel/test*, 2021).

creating another new PR for the new changes, which could reduce the number of open PRs in the repository.

6.7 Threats to validity

Internal validity: Threats to internal validity concern factors that might affect the casual relationship and experimental bias. In RQ2, we manually analyze the non-merged PRs to identify the reasons of their apparent rejection by developers. This analysis is subjected to the human bias, as every investigator has a subjective method when classifying a PR. We mitigate this threat by asking a second annotator to independently classify the reasons for not merging and calculate the inter-rater agreement in our methodology (Cohens Kappa coefficient (Cohen, 1960)). The level agreement (+0.96) indicates that our results are more likely to hold.

Another concern is related to the conclusion drawn from the built model by studying the association between the independent and dependent variables. In our work, we study the features that

influence the time it takes to merge a Dependabot security PR. To achieve that, we built our model using 13 features that span over three dimensions. However, our set of features are not exhaustive, and other features can be added and show influence for the PR merge times. Still, our model is able to explain 67% of the data variation, which for our purposes is a good initial model for understanding the factors that correlate with the merge time.

External validity: Threats to external validity concern the generalizability of our findings. Our study analyses only JavaScript projects that subscribe to Dependabot. Therefore, our results cannot be generalized to projects of different languages and other ecosystems. Still, given that JavaScript was the first major language supported by Dependabot, it has had a more widespread adoption, which enable us to assess its use on a larger dataset of projects. Furthermore, our methodology can be applied to investigate Dependabot in projects from other programming languages.

6.8 Related Work

In this section, we present the work most related to the study in this chapter. The works most related to ours are studies that propose or discuss dependency management tools for security vulnerabilities. Previous works (e.g., (*David*, accessed on 12/10/2021; *Greenkeeper*, accessed on 12/10/2021)) aim to help project maintainers automatically track and update their dependencies. For example, David-DM (*David*, accessed on 12/10/2021) is a tool that uses what is called "coloured (badges)", trying to convince developers to update their outdated dependencies (regardless of the package being afected by a vulnerability). The tool checks for outdated dependencies and colours a dependency badge with red, indicating that an outdated dependency version is used. Greenkeeper (*Greenkeeper*, accessed on 12/10/2021), an automated pull-requests bot, is another tool that helps developers keep their project dependencies up-to-date by creating PRs that make the required changes for the dependency version update.

A study by Mirhosseini and Parnin (Mirhosseini & Parnin, 2017). Their work investigated the use of pull requests and badges in the tools David-DM (badges) and Greenkeeper (PRs) to understand whether such tools help developers upgrade outdated dependencies. They analyzed more than 6K GitHub projects that used these tools, and found that projects using the PR tool

(i.e., Greenkeeper) tend to upgrade more often than projects that use the Badge tool (David-DM). Nevertheless, the Greenkeeper tool could convince developers of the examined projects to accept only a third of the submitted PRs with a relativity high rate of build breakages (i.e., 25%), indicating the need for better automated dependency tools to convince developers respond to these PRs. The analysis in the study focused on only seven npm packages in the studied projects.

Dependabot (Dependabot, accessed on 12/10/2021) is a bot (acquired by GitHub in 2019) that creates pull requests to monitor project dependencies and help developers automatically integrate dependency updates and vulnerability fixes. Also, it provides information about the vulnerability, such as its severity, versions affected, information about the issue from the advisory report, which developers can analyze to consider the risks of not updating. Moreover, the PR contains information about the compatibility of the PR with the project, calculated based on the outcome of updates done by similar projects (*Dependabot Score*, accessed on 12/10/2021).

The work that is most close to ours is the study by Mirhosseini and Parnin (Mirhosseini & Parnin, 2017). Their work investigated the use of pull requests and badges in the tools David-DM (badges) and Greenkeeper (PRs) to understand whether such tools help developers upgrade outdated dependencies. They analyzed more than 6K GitHub projects that used these tools, and found that projects using the PR tool (i.e., Greenkeeper) tend to upgrade more often than projects that use the Badge tool (David-DM). Nevertheless, the Greenkeeper tool could convince developers of the examined projects to accept only a third of the submitted PRs with a relativity high rate of build breakages (i.e., 25%), indicating the need for better automated dependency tools to convince developers respond to these PRs. The analysis in the study focused on only seven npm packages in the studied projects.

While previous work investigated to which extent dependency management tools can convince developers to upgrade out-of-date dependencies, they focus on the general problem of outdated dependencies and do not pay particular attention to security vulnerabilities in dependencies. Our study complements previous works since we specifically focus on studying security updates, i.e., we study a large dataset of Dependabot security pull requests. Moreover, we examine the reasons for Dependabot security PRs being not-merged. Our case study shows that developers make a good use of dependency tools such as Dependabot, responding quickly to the majority of Dependabot security

pull requests (less than a day), suffering from a low rate of build breakages. Additionally, our study adds to the literature through, for example, understanding what factors influence the merge time of a Dependabot security PR. Lastly, we do believe our observations from evaluating Dependabot can also help application developers choose effective security dependency tools and persuade other tool builders and researchers to address limitations of the current tools.

6.9 Chapter Summary

This chapter conducts an empirical study to investigate the use of Dependabot security pull requests, by examining 15,243 pull requests submitted to 2,904 JavaScript open source GitHub projects. Our results show that a large proportion (65.42%) of Dependabot security PRs are merged, often in one day. Furthermore, our manual analysis leads us to identify that most of the non-merged security PRs (93.9%) are actually closed by Dependabot itself, mostly related to concurrent modifications on the affected dependencies, rather than Dependabot failures. Finally, we build a mixed-effects regression model to understand why some of the pull requests take longer to be merged. Our results reveal 5 important features, e.g., the project past experience with Dependabot security PRs is the most influential feature. We note, however, that the severity of the vulnerability and the risk of breaking changes are not significantly associated with rapid merges. Our findings indicate that Dependabot provides an effective platform to help developers secure their dependencies. Leveraging our findings, we provide a series of implications that is of interest for practitioners and Dependabot maintainers alike. Finally, to enhance the process of merging Dependabot security PRs, we build a tool called DepCombine, which combines multiple Dependabot security PRs, creating a new PR that merges all Dependabot PRs together in a GitHub repository.

Chapter 7

Conclusion and Future Work

In this chapter, we conclude the thesis by summarizing the main work and contribution in each chapter of the thesis. At the end of the chapter, we discussed some directions for future research.

7.1 Conclusion

The work presented in this thesis has emerged from the observation that software packages have become popular in software development and that software developers increasingly rely on these packages. However, such packages also increase the impact of security vulnerabilities and may directly impact hundreds of applications, leading to significant financial costs and reputation loss. In this thesis, we focused on examining some of the most critical aspects for dealing with security vulnerabilities that affect the software package. We described and reported on a series of empirical studies that investigate the lifecycle of package vulnerabilities and their impact on dependent applications. Our finding showed that package vulnerabilities often take a long time to be discovered. Furthermore, software applications that rely on external packages remain affected by public vulnerabilities for a long time, giving ample time for attackers exploitation. We also performed empirical studies that investigate some popular mechanisms for mitigating the impact of package vulnerabilities (i.e., code review process and software bots). We qualitatively uncover the possible issues developers could face when adopting these mechanisms. Such evaluation can also advance the future work, i.e., researchers can direct their efforts to identify the cause of the issues

and propose solutions to overcome the limitations.

Based on the findings in our studies, this thesis attempts to improve the practice of relying on software packages from a security perspective, by building several prototype tools to enhance the maintenance of vulnerable packages in software projects. More specifically, the presented research provides the following main contributions:

7.1.1 Analysing the Lifecycle of Package Vulnerabilities

We perform the first empirical study to analyse security vulnerabilities in the Python ecosystem (PyPi). Our study covers 12 years of PyPi reported vulnerabilities, affecting 252 Python packages. Our results show that PyPi vulnerabilities are increasing over time, affecting the large majority of package versions. Additionally, we observe that the timing of vulnerability patches does not closely align with the public disclosure date, leaving open windows and chances for an attacker exploitation. We compare the findings of our study to a previous study conducted on the npm ecosystem. Our comparison shows a drastic departure from npm's reported findings in some aspects, which can be attributed to ecosystems policies. Finally, we build a tool called DEPHEALTH, which uses the analysis approach in our study to generate analytical report of security vulnerabilities that affect Python packages, i.e., we provide developers with metrics related to the lifecycle of vulnerability discovery and fix, which help to show how maintained and secure the packages are.

7.1.2 Examining the Discoverability of Package Vulnerabilities Impacting Software Applications

The thesis presents the first empirical study on 6,546 open-source JavaScript applications to determine the prevalence of affected applications that rely on vulnerable dependencies, taking into consideration the vulnerability disclosure timeline (i.e., the discoverability aspect). We also examine why these applications end up depending on vulnerable versions of the package in order to better understand how we can mitigate such issues. We show that 1) the majority of the affected applications depend on hidden dependency vulnerabilities. Though, a non-trivial number of the applications were still affected by a public dependency vulnerability. 2) Such applications often

remain affected for a substantial long time during the application lifetime, and 3) the application developers are mostly to blame, i.e., a fix for the vulnerable dependency is available but not patched in the application. Finally, we develop DEPREVEAL , a tool that generates historical analytical reports to increase developers awareness to the *discoverability* of their JavaScript application dependencies.

7.1.3 Studying The Role of Code Review in Enhancing Package Security

To help developers understand the role of code review in relation with security, we investigate 10 active and popular JavaScript projects. In particular, we aim to understand what types of security issues are raised during code review, and what kind of mitigation strategies are employed by project maintainers to address them. Our study examines 171 pull requests (PRs) with raised security concerns. We find that 1) such issues represent a small fraction of all PRs in the studied projects. However, we find that such issues are discussed at length by project maintainers. 2) Code review approach is more effective to find certain types of security issues over other methods, i.e., advisory method. 3) Although the majority of identified issues are frequently fixed, we find a non-negligible share of issues ended up not being fixed or are ignored by maintainers. Our findings help the community better understand the role of code review from a security perspective, trying to improve the practice of code review for the software security, by understanding the types of security issues discovered during code review in order to pay attention to them in the future, and understand the mitigation strategies employed by project maintainers to tackle the issues. Leveraging our findings, we offer several implications that support the role of reviewing code for security concerns.

7.1.4 Evaluating the Use of Dependabot for Patching Package Vulnerabilities

We examine the use of Dependabot security pull-requests for tackling vulnerable packages in 2,904 JavaScript projects. We show that 1) a large proportion (65.42%) of Dependabot security PRs are merged, often in one day. 2) Most of the non-merged security PRs are actually closed by Dependabot itself, mostly related to concurrent modifications on the affected dependencies, rather than Dependabot failures. 3) We also found that several factors could influence the PR merge time, e.g, we find that the project past experience with Dependabot security PRs is the most influential feature. Our findings illustrate that Dependabot provides an effective platform to help developers

secure their dependencies. The implications of our work suggests ways in which Dependabot notifications can be improved to reduce the problem of notification fatigue and improve developers productivity. We build a tool called DEPCOMBINE , which combines multiple Dependabot security PRs in a GitHub repository, and create a new single PR that merges all the PRs together.

7.2 Future Work

Although this thesis work has made many contributions towards understanding security vulnerabilities that affect open-source software packages, many different avenues for future work remain unexplored. Next, we summarize some of the main directions for future work.

7.2.1 Examining Fine-Grained Solutions to Address Package Vulnerability in Software Applications

The work of this thesis focused on analysing security vulnerabilities that affect software packages, and how they may impact software applications. Most of the explored solutions for such analysis consider coarse-grained methods. Whenever a vulnerable dependency is found, the common mitigation action consists of updating the dependency to another non-vulnerable version. While this solution seems reasonable and easy to adopt, it can be difficult particularly when the library to be updated is not compatible with or has some production issues in the application. There are other solutions that have tackled the problem by providing fine-grained code analyses to reduce the number of false alerts (i.e., dependencies flagged as vulnerable but that do not expose the dependent application to any threat) (Pashchenko et al., 2018; Plate et al., 2015; Ponta et al., 2020; Zapata et al., 2018). Such solutions provide a combination of both static (i.e., call graph-based) and dynamic analyses (i.e., test-based) to maximize the reachability of known vulnerable package constructs (e.g., method, class) starting from the client application code. However, one limitation of this solution is that it requires a significant amount of data from the application test suite (i.e., execution traces) to make an effective vulnerability assessment. Unfortunately, many software projects are not adequately tested. Therefore, further research should explore more efficient and automated solutions to tackle this problem. Software fuzzers (Manès et al., 2019) are known as efficient and

effective software testing techniques that are used to test programs against unexpected behaviors to expose corner cases that have not been properly dealt with, which might overcome the problem of inadequate test suite in the application. The research should investigate the possibility of applying these techniques (e.g., fuzzers) for analysing known package vulnerabilities that impact software applications.

7.2.2 Exploring Different Data Sources for Package Vulnerabilities

Throughout our work in this thesis, we utilized several databases for package vulnerabilities. For instance, in Chapter 3, to collect security vulnerabilities for the PyPi packages, we resort to the dataset provided by Snyk.io. Also, in Chapter 4, we resort to the npm advisories registry to obtain the required information about all npm vulnerable packages. In the Dependabot study, we observe that the bot utilizes GitHub's advisory database to track and pull patched version of vulnerable packages. Besides, many other tools monitor vulnerable dependencies. Tools supporting npm and JavaScript include commercial services (Gemnasium, SRC:CLR), as well as free/open-source solutions (RetireJS, Node Security Project, OSSIndex). While all these tools search existing databases for vulnerability information, some of them also recover information from their own private databases and from mailing lists, bug-tracking systems and blogs. This diversity of tools and data sources when reporting security issues makes it impossible to get a complete historic view on the ecosystem's vulnerabilities. Therefore, one direction for future work is to investigate the differences among such tools and data-sources, and also examine the possibilities to maintain and contribute to a single and common vulnerability database reporting all vulnerabilities related to the package dependency network.

7.2.3 Replication in an Industrial Setting

The results included in this thesis showed that reusing packages affects the quality of software from a security perspective. However, these results are based on analyzing only open source projects. While we have done our best to select appropriate representative large platforms such as PyPi and npm and data analysis techniques in order to reduce the threats to internal validity, we believe that practitioners need to understand how the problem impacts their project qualities and

what challenges arise when relying on such software packages. A future research that investigates and studies the impact of vulnerable packages in an industrial setting would allow us to further generalize our results.

7.2.4 Evaluating the Proposed Tools

The implications of our work in this thesis lead us to build several prototype tools that aim to increase the awareness of developers to the impact of vulnerable packages. However, the actual usefulness of these tools and the degree to which developers would adopt them remains unknown. Future work should explore how developer perceive such tools, e.g., by conducting surveys with software developers to understand their point of view on the tools, and how they can be improved further to encourage developers adopt them.

7.2.5 Understanding Developers Perception on the Studied Aspects

Another direction for the future is to gain more insights on the aspects studied in this thesis from developers point of view. For instance, future work should focus on improving our work by conducting surveys with developers to understand their perception on the role of code review in identifying and fixing security issues. Another example is to conduct survey with developers to obtain more insights on the use of Dependabot and what kind of challenges developers face while adopting the bot.

7.2.6 Other Languages of Software Ecosystems

While this thesis work has focused on the impact of software packages from popular package ecosystems, e.g., PyPi and npm, however, there are other important package ecosystems. We argue that it is important to study other software ecosystems to contrast with our studies and draw more generalizable empirical evidence about vulnerabilities in software ecosystems. Future work should focus on broadening our studies to other ecosystems and work on the development of package security tools that help practitioners at selecting and using secure software packages in other ecosystems.

References

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

Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., & Shihab, E. (2017). 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* (pp. 385–395). ACM. Retrieved from http://doi.acm.org/10.1145/3106237.3106267 doi: 10.1145/3106237.3106267

About codeql code scanning in your ci system. (accessed on 12/10/2021). https://docs.github.com/en/code-security/code-scanning/using-codeql-code-scanning-with-your-existing-ci-system/about-codeql-code-scanning-in-your-ci-system. (accessed on 10/04/2021)

Alfadel, M. (2021). *Depreveal.* https://github.com/mahmoud-alfadel/DepReveal. ((accessed on 07/17/2021))

Alfadel, M., Costa, D. E., & Shihab, E. (2021). Empirical analysis of security vulnerabilities in python packages. In *Proceedings of the 2021 ieee international conference on software analysis, evolution and reengineering (saner)* (pp. 446–457).

Alfadel, M., Costa, D. E., Shihab, E., & Mkhallalati, M. (2021). On the use of dependabot security pull requests. *Proceedings of the 2021 International Conference on Mining Software Repositories (MSR 21)*.

Allodi, L., & Massacci, F. (2014). Comparing vulnerability severity and exploits using case-control studies. *ACM Transactions on Information and System Security (TISSEC)*, *17*(1), 1–20.

Aloraini, B., Nagappan, M., German, D. M., Hayashi, S., & Higo, Y. (2019). An empirical study of security warnings from static application security testing tools. *Journal of Systems and Software*, *158*, 110427.

Android google play protect. (accessed on 12/10/2021)https://www.android.com/intl/en_ca/play-protect/.

Appsec on dependency management. (2020 (accessed 2020)). https://blog.npmjs.org/post/187496869845/appsec-pov-on-dependency-management

Bacchelli, A., & Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. In *2013 35th international conference on software engineering (icse)* (pp. 712–721).

Basili, V. R., Briand, L. C., & Melo, W. L. (1996a). How reuse influences productivity in object-oriented systems. *Communications of the ACM*, *39*(10), 104–116.

Basili, V. R., Briand, L. C., & Melo, W. L. (1996b). How reuse influences productivity in object-oriented systems. *Communications of the ACM*, *39*(10), 104–116.

Beller, M., Bacchelli, A., Zaidman, A., & Juergens, E. (2014). Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th working conference on mining software repositories* (pp. 202–211).

Bernardo, J. H., da Costa, D. A., & Kulesza, U. (2018). Studying the impact of adopting continuous integration on the delivery time of pull requests. In *2018 ieee/acm 15th international conference on mining software repositories (msr)* (pp. 131–141).

Bewick, V., Cheek, L., & Ball, J. (2004). Statistics review 12: survival analysis. *Critical care*, *8*(5).

Bogart, C., Kästner, C., & Herbsleb, J. (2015). When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *30th ieee/acm international conference on automated software engineering workshop (asew), 2015* (pp. 86–89).

Bogart, C., Kästner, C., Herbsleb, J., & Thung, F. (2016). 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* (pp. 109–120).

Bogart, C., Kstner, C., & Herbsleb, J. (2015, Nov). 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)* (pp. 86–89). doi: 10.1109/

ASEW.2015.21

Bohner, S. A. (2002). Extending software change impact analysis into cots components. In *27th annual nasa goddard/ieee software engineering workshop, 2002. proceedings.* (pp. 175–182).

Bosu, A. (2014). Characteristics of the vulnerable code changes identified through peer code review. In *Companion proceedings of the 36th international conference on software engineering* (pp. 736–738).

box/box-ui-elements. (accessed on 12/10/2021). https://github.com/box/box-ui-elements.

Broken access control — owasp. (accessed on 12/10/2021). https://owasp.org/www-project-top-ten/2017/A5_2017-Broken Access Control

Cadariu, M., Bouwers, E., Visser, J., & van Deursen, A. (2015). Tracking known security vulnerabilities in proprietary software systems. In *2015 ieee 22nd international conference on software analysis, evolution, and reengineering (saner)* (pp. 516–519).

Chacon, S., & Straub, B. (2014). *Pro git*. Springer Nature.

Chinthanet, B., Kula, R. G., McIntosh, S., Ishio, T., Ihara, A., & Matsumoto, K. (2019). Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering*.

Chinthanet, B., Kula, R. G., McIntosh, S., Ishio, T., Ihara, A., & Matsumoto, K. (2021). Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering*, *26*(3), 1–28.

Chinthanet, B., Ponta, S. E., Plate, H., Sabetta, A., Kula, R. G., Ishio, T., & Matsumoto, K. (2020). Code-based vulnerability detection in node. js applications: How far are we? In *2020 35th ieee/acm international conference on automated software engineering (ase)* (pp. 1199–1203).

cla assitant. (accessed on 12/10/2021). *Cla assistant.* https://github.com/cla-assistant/cla-assistant.

Cogo, F. R., Oliva, G. A., & Hassan, A. E. (2019). An empirical study of dependency downgrades in the npm ecosystem. *IEEE Transactions on Software Engineering*.

Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and psychological*

measurement, *20*(1), 37–46.

Configuring notifications. (accessed on 12/10/2021). https://docs.github.com/en/free-pro-team@latest/github/managing-subscriptions-and-notifications-on-github/configuring-notifications#filtering-email-notifications.

Cvss for cves — snyk. (accessed on 12/10/2021). https://snyk.io/blog/scoring-security-vulnerabilities-101-introducing-cvss-for-cve/.

Cwe-416. (accessed on 12/10/2021). https://cwe.mitre.org/data/definitions/416.html.

Cwe - common weakness enumeration. (accessed on 12/10/2021).https://cwe.mitre.org/index.html.

Cwe list. (accessed on 12/10/2021). https://cwe.mitre.org/about/index.html.

Cwe list version. (accessed on 12/10/2021). https://cwe.mitre.org/data/index.html.

David. (accessed on 12/10/2021). http://freestyle-developments.co.uk/blog/?p=457.

A day in the life of npm security. (accessed on 12/10/2021). https://blog.npmjs.org/post/190665497245/a-day-in-the-life-of-npm-security.html.

Decan, A., & Mens, T. (2019). What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*.

Decan, A., Mens, T., & Claes, M. (2016). On the topology of package dependency networks: A comparison of three programming language ecosystems. In *Proccedings of the 10th european conference on software architecture workshops* (p. 21).

Decan, A., Mens, T., & Claes, M. (2017). An empirical comparison of dependency issues in oss packaging ecosystems. In *2017 ieee 24th international conference on software analysis, evolution and reengineering (saner)* (pp. 2–12).

Decan, A., Mens, T., & Constantinou, E. (2018a). On the evolution of technical lag in the npm package dependency network. In *2018 ieee international conference on software maintenance and evolution (icsme)* (pp. 404–414).

Decan, A., Mens, T., & Constantinou, E. (2018b). On the impact of security vulnerabilities in the npm package dependency network. In *International conference on mining software repositories*.

Decan, A., Mens, T., & Grosjean, P. (2018). An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 1–36.

Decan, A., Mens, T., & Grosjean, P. (2019). An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, *24*(1), 381–416.

Deepscan. ((accessed 2020)). https://deepscan.io/.

Depcombine — home. (2021). http://104.237.154.205:8449/. (accessed on 10/04/2021)

Dependabot. (accessed on 12/10/2021). https://dependabot.com/.

Dependabot, A. (accessed on 12/10/2021). *Dependabot*. https://dependabot.com/.

dependabot-core. (accessed on 12/10/2021). https://github.com/dependabot/dependabot-core.

Dependabot introduction. (accessed on 12/10/2021). https://dependabot.com/blog/introducing-dependabot/.

Dependabot page. (accessed on 12/10/2021). https://dependabot.com/.

Dependabot schedule. (accessed on 12/10/2021). https://dependabot.com/docs/config-file/#update_schedule-required

Dependabot score. (accessed on 12/10/2021). https://dependabot.com/compatibility-score/?dependency-name=bootstrap&package-manager=npm_and_yarn&version-scheme=semver

Dependabot security updates. (accessed on 12/10/2021). https://docs.github.com/en/free-pro-team@latest/github/managing-security-vulnerabilities/about-dependabot-security-updates

Dependabot tool. (accessed on 12/10/2021). https://github.com/dependabot.

Dep health — home. (2021). http://104.237.154.205:8443/?fbclid=IwAR3qdZPNXISqK7VkPNXYQaEhtdxKR8nBEbmqGJI7Z-nHw9f6_oSNAjLc_dI

(accessed on 12/10/2021)

Depreveal. (2021, July). https://bit.ly/3emg5w3. ((accessed on 07/17/2021))

Dey, T., & Mockus, A. (2020). Effect of technical and social factors on pull request quality for the npm ecosystem. In *Proceedings of the 14th acm/ieee international symposium on empirical software engineering and measurement (esem)* (pp. 1–11).

di Biase, M., Bruntink, M., & Bacchelli, A. (2016). A security perspective on code review: The case of chromium. In *2016 ieee 16th international working conference on source code analysis and manipulation (scam)* (pp. 21–30).

Disclosure of security vulnerabilities. (accessed on 12/10/2021). https://docs.github.com/en/code-security/security-advisories/about-coordinated-disclosure-of-security-vulnerabilities#about-disclosing-vulnerabilities-in-the-industry.

Domingos, P. (2012). A few useful things to know about machine learning. *Communications of the ACM*, *55*(10), 78–87.

Dowd, M., McDonald, J., & Schuh, J. (2006). *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education.

Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., . . .others (2014). The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference* (pp. 475–488).

Ebert, F., Castor, F., Novielli, N., & Serebrenik, A. (2019). Confusion in code reviews: Reasons, impacts, and coping strategies. In *2019 ieee 26th international conference on software analysis, evolution and reengineering (saner)* (pp. 49–60).

electron - npm. (accessed on 12/10/2021). https://www.npmjs.com/package/electron.

Equifax. (2017). *Equifax releases details on cybersecurity incident, announces personnel changes — equifax,*. Retrieved from https://investor.equifax.com/news-and-events/news/2017/09-15-2017-224018832 (accessed on 01/12/2021)

Fard, A. M., & Mesbah, A. (2017). Javascript: The (un) covered parts. In *Ieee international conference on software testing, verification and validation (icst), 2017* (pp. 230–240).

Fincher, S., & Tenenberg, J. (2005). Making sense of card sorting data. *Expert Systems*, *22*(3), 89–93.

Fleiss, J. L., & Cohen, J. (1973). The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability. *Educational and psychological measurement*, *33*(3), 613–619.

Frakes, W. B., & Kang, K. (2005). Software reuse research: Status and future. *IEEE transactions on Software Engineering*, *31*(7), 529–536.

Ghaleb, T. A., Da Costa, D. A., & Zou, Y. (2019). An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering*, *24*(4), 2102–2139.

Git - contributing to a project. (accessed on 12/10/2021). https://git-scm.com/book/en/v2/GitHub-Contributing-to-a-Project.

GitHub. (2017). *Github advisory database.* https://github.com/advisories?page=1. ((accessed on 01/12/2021))

GitHub. (accessed on 12/10/2021). *Advisory database.* https://github.com/advisories.

github/codeql. (accessed on 12/10/2021). https://github.com/github/codeql. (accessed on 10/04/2021)

Gkortzis, A., Feitosa, D., & Spinellis, D. (2020). Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. *Journal of Systems and Software*, 1–14.

Godefroid, P., Levin, M. Y., & Molnar, D. (2012). Sage: whitebox fuzzing for security testing. *Communications of the ACM*, *55*(3), 40–44.

Gong, L. (2018). *Dynamic analysis for javascript code* (Unpublished doctoral dissertation). UC Berkeley.

Gousios, G. (2013). The ghtorrent dataset and tool suite. In *Proceedings of the 10th working conference on mining software repositories* (pp. 233–236). Piscataway, NJ, USA: IEEE Press. Retrieved from http://dl.acm.org/citation.cfm?id=2487085.2487132

Gousios, G., Pinzger, M., & Deursen, A. v. (2014). An exploratory study of the pull-based software development model. In *Proceedings of the 36th international conference on software*

engineering (pp. 345–355).

Greenkeeper. (accessed on 12/10/2021). https://greenkeeper.io/.

Harrell Jr, F. E. (2015). *Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis*. Springer.

Heartbleed bug. (accessed on 12/10/2021). https://heartbleed.com/#:~:text=The%20Heartbleed%20Bug%20is%20a,used%20to%20secure%20the%20Internet.

Heartbleed flaw. (accessed on 12/10/2021). https://www.computerworld.com/article/2605220/heartbleed-flaw-was-unknown-before-disclosure.html.

Hejderup, J. (2015). *In dependencies we trust: How vulnerable are dependencies in software modules?*

Hora, A., & Valente, M. T. (2015). apiwave: Keeping track of api popularity and migration. In *2015 ieee international conference on software maintenance and evolution (icsme)* (pp. 321–323).

Howard, M. A. (2006). A process for performing security code reviews. *IEEE Security & privacy*, *4*(4), 74–79.

How snyk finds out about new vulnerabilities. (accessed on 12/10/2021). https://support.snyk.io/hc/en-us/articles/360003923877-How-Snyk-finds-out-about-new-vulnerabilities.

Imtiaz, N., Thorne, S., & Williams, L. (2021). A comparative study of vulnerability reporting by software composition analysis tools. *arXiv preprint arXiv:2108.12078*.

infor-design - npm. (accessed on 12/10/2021). https://github.com/infor-design/enterprise-ng.

Inoue, K., Sasaki, Y., Xia, P., & Manabe, Y. (2012). Where does this code come from and where does it go?-integrated code history tracker for open source systems. In *Proceedings of the 34th international conference on software engineering* (pp. 331–341).

Internet systems consortium. (accessed on 12/10/2021). https://www.isc.org/#.

Issue #1138. (accessed on 12/10/2021). https://github.com/dependabot/dependabot-core/issues/1138.

Issue 27863. (accessed on 12/10/2021). https://bugs.python.org/issue27863.

Job #231.2 - travis ci. (accessed on 12/10/2021). https://travis-ci.org/github/joshghent/blog/jobs/577015435.

Johari, R., & Sharma, P. (2012). A survey on web application vulnerabilities (sqlia, xss) exploitation and security engine for sql injection. In *2012 international conference on communication systems and network technologies* (pp. 453–458).

Js: Handle version resolution. (accessed on 12/10/2021). https://github.com/dependabot/dependabot-core/commit/b917ac195748f2d2812071c3cffaf7b77b6b5489.

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

Kaplan, E. L., & Meier, P. (1958). Nonparametric estimation from incomplete observations. *Journal of the American statistical association*, *53*(282), 457–481.

Kononenko, O., Rose, T., Baysal, O., Godfrey, M., Theisen, D., & De Water, B. (2018). Studying pull request merges: a case study of shopify's active merchant. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice* (pp. 124–133).

Kula, R. G., German, D. M., Ouni, A., Ishio, T., & Inoue, K. (2018). Do developers update their library dependencies? *Empirical Software Engineering*, *23*(1), 384–417.

Kula, R. G., Ouni, A., German, D. M., & Inoue, K. (2017). On the impact of micro-packages: An empirical study of the npm javascript ecosystem. *arXiv preprint arXiv:1709.04638*.

Larios-Vargas, E., Aniche, M., Treude, C., Bruntink, M., & Gousios, G. (2020). Selecting third-party libraries: The practitioners' perspective. *arXiv preprint arXiv:2005.12574*.

Lewis, A. J. (2009). *Mixed effects models and extensions in ecology with r*. Springer.

Li, F., & Paxson, V. (2017). A large-scale empirical study of security patches. In *Proceedings of the 2017 acm sigsac conference on computer and communications security* (pp. 2201–2215).

Libraries.io - the open source discovery service. (2021). Retrieved from http://libraries.io/

Libraries.io - the open source discovery service. (accessed on 12/10/2021). https://libraries.io/.

Lim, W. C. (1994a). Effects of reuse on quality, productivity, and economics. *IEEE software*(5), 23–30.

Lim, W. C. (1994b). Effects of reuse on quality, productivity, and economics. *IEEE software*, *11*(5), 23–30.

Lu, L., Li, Z., Wu, Z., Lee, W., & Jiang, G. (2012). Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 acm conference on computer and communications security* (pp. 229–240).

Maiden, N. A., & Ncube, C. (1998). Acquiring cots software selection requirements. *IEEE software*, *15*(2), 46–56.

M. Alfadel, E. S., D. Costa. (2019, August). *On the Unexploitability of Security Vulnerabilities: A Case Study on Node.js.* Retrieved from https://doi.org/10.5281/zenodo.3376290 doi: 10.5281/zenodo.3376290

Manès, V. J. M., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., & Woo, M. (2019). The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*.

Mann, H. B., & Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, 50–60.

Mäntylä, M. V., & Lassenius, C. (2008). What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering*, *35*(3), 430–448.

marked - npm. (accessed on 12/10/2021). https://www.npmjs.com/package/marked

marked@v0.3.4. (2020 accessed on 12/10/2021). https://github.com/atom/atom/commit/41b799aebc7f40201219d8ec435d1520cf057285

McIntosh, S., Kamei, Y., Adams, B., & Hassan, A. E. (2014). The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th working conference on mining software repositories* (pp. 192–201).

McIntosh, S., Kamei, Y., Adams, B., & Hassan, A. E. (2016). An empirical study of the impact of

modern code review practices on software quality. *Empirical Software Engineering*, *21*(5), 2146–2189.

Merge pull request from parse-server@da905a3. (accessed on 12/10/2021). https://github.com/parse-community/parse-server/commit/da905a357d062ab4fea727a21eac231acc2ed92a

Mezzetti, G., Møller, A., & Torp, M. T. (2018). Type regression testing to detect breaking changes in node. js libraries. In *32nd european conference on object-oriented programming (ecoop 2018)*.

Mirhosseini, S., & Parnin, C. (2017). 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)* (pp. 84–94).

moment - npm. (accessed on 12/10/2021). https://www.npmjs.com/package/moment.

Nakagawa, S., & Schielzeth, H. (2013). A general and simple method for obtaining r2 from generalized linear mixed-effects models. *Methods in ecology and evolution*, *4*(2), 133–142.

Nesbitt, A., & Nickolls, B. (2018). *Libraries.io Open Source Repository and Dependency Metadata. v1.2.0.* https://doi.org/10.5281/zenodo.808273 ([Online; accessed 10/10/2020])

Neuhaus, S., & Zimmermann, T. (2009). The beauty and the beast: Vulnerabilities in red hat's packages. In *Usenix annual technical conference*.

node-red - npm. (accessed on 12/10/2021). https://www.npmjs.com/package/node-red.

npm. (2021). *npm advisories*. https://www.npmjs.com/advisories. ((accessed on 07/17/2021))

npm. (accessed on 12/10/2021). *npm-registry*. https://docs.npmjs.com/using-npm/registry.html.

npm advisories. ((accessed 2020)). *About audit reports — npm docs*. https://bit.ly/3uqn0uv.

npm advisories. (accessed on 12/10/2021). https://www.npmjs.com/advisories.

npm advisory reports. (accessed on 12/10/2021). https://www.npmjs.com/advisories.

npm audit. (accessed on 12/10/2021). https://docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities.

npm - libraries.io. (2021). https://libraries.io/NPM ((accessed on 07/17/2021))

npm - libraries.io. (accessed on 12/10/2021). https://libraries.io/npm.

npm publications. (accessed on 12/10/2021). https://www.npmjs.com/advisories/33.

OWASP.(2019). *Owasp*. https://www.owasp.org/index.php/Main_Page ((accessed on 10/10/2020))

OWASP. (2020 (accessed 10/10/2020)). Open web application security project [Computer software manual]. Retrieved from https://www.owasp.org/index.php/Main_Page

parse-server - npm. (accessed on 12/10/2021).https://www.npmjs.com/package/parse-server.

Pashchenko, I., Plate, H., Ponta, S. E., Sabetta, A., & Massacci, F. (2018). Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th acm/ieee international symposium on empirical software engineering and measurement* (pp. 1–10).

Pashchenko, I., Plate, H., Ponta, S. E., Sabetta, A., & Massacci, F. (2020). Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering*.

Pashchenko, I., Vu, D.-L., & Massacci, F. (2020). A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 acm sigsac conference on computer and communications security* (pp. 1513–1531).

Paul, R., Turzo, A. K., & Bosu, A. (2021). Why security defects go unnoticed during code reviews? a case-control study of the chromium os project. *arXiv preprint arXiv:2102.06909*.

Pham, N. H., Nguyen, T. T., Nguyen, H. A., & Nguyen, T. N. (2010). Detection of recurring software vulnerabilities. In *Proceedings of the ieee/acm international conference on automated software engineering* (pp. 447–456).

Plate, H., Ponta, S. E., & Sabetta, A. (2015). Impact assessment for vulnerabilities in open-source software libraries. In *2015 ieee international conference on software maintenance and evolution (icsme)* (pp. 411–420).

Ponta, S. E., Plate, H., & Sabetta, A. (2018). Beyond metadata: Code-centric and usage-based

analysis of known vulnerabilities in open-source software. In *2018 ieee international conference on software maintenance and evolution (icsme)* (pp. 449–460).

Ponta, S. E., Plate, H., & Sabetta, A. (2020). Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, *25*(5), 3175–3215.

Pull request #105. (accessed on 12/10/2021). https://github.com/joshghent/blog/pull/105.

Pull request #10816. (accessed on 12/10/2021). https://github.com/sequelize/sequelize/pull/10816.

Pull request #1083. (accessed on 12/10/2021).https://github.com/markedjs/marked/pull/1083#issuecomment-368726539.

Pull request #1220. (accessed on 12/10/2021).https://github.com/markedjs/marked/pull/1220.

Pull request #1224. (accessed on 12/10/2021).https://github.com/markedjs/marked/pull/1224.

Pull request #1305. (accessed on 12/10/2021).https://github.com/markedjs/marked/pull/1305.

Pull request #13367. (accessed on 12/10/2021a). https://github.com/facebook/react/pull/13367.

Pull request #13367. (accessed on 12/10/2021b). https://github.com/facebook/react/pull/13367#issuecomment-412349795.

Pull request #1414. (accessed on 12/10/2021).https://github.com/markedjs/marked/pull/1414. (accessed on 10/04/2021)

Pull request #1420. (accessed on 12/10/2021).https://github.com/markedjs/marked/pull/1420. (accessed on 10/04/2021)

Pull request #1472. (accessed on 12/10/2021).https://github.com/markedjs/marked/pull/1472.

Pull request #1521. (accessed on 12/10/2021). https://github.com/box/box-ui-elements/pull/1521.

Pull request #1598. (accessed on 12/10/2021a). https://github.com/markedjs/

marked/pull/1598.

Pull request #1598. (accessed on 12/10/2021b). https://github.com/markedjs/marked/pull/1598.

Pull request #16254. (accessed on 12/10/2021).https://github.com/facebook/react/pull/16254.

Pull request #1681. (accessed on 12/10/2021). https://github.com/strapi/strapi/pull/1681.

Pull request #1683. (accessed on 12/10/2021a). https://github.com/markedjs/marked/pull/1683.

Pull request #1683. (accessed on 12/10/2021b). https://github.com/markedjs/marked/pull/1683.

Pull request #18673. (accessed on 12/10/2021).https://github.com/facebook/react/pull/18673.

Pull request #20818. (accessed on 12/10/2021). https://github.com/electron/electron/pull/20818.

Pull request #23650. (accessed on 12/10/2021). https://github.com/electron/electron/pull/23650.

Pull request #2375. (accessed on 12/10/2021). https://github.com/sequelize/sequelize/pull/2375.

Pull request #2402. (accessed on 12/10/2021). https://github.com/node-red/node-red/pull/2402.

Pull request #245. (accessed on 12/10/2021). https://github.com/FromDoppler/doppler-webapp/pull/245.

Pull request #250. (accessed on 12/10/2021). https://github.com/FromDoppler/doppler-webapp/pull/250.

Pull request #2576. (accessed on 12/10/2021a). https://github.com/sequelize/sequelize/pull/2576.

Pull request #2576. (accessed on 12/10/2021b). https://github.com/sequelize/sequelize/pull/2576.

Pull request #2984. (accessed on 12/10/2021). https://github.com/infor-design/enterprise/pull/2984.

Pull request #3. (accessed on 12/10/2021). https://github.com/codeparticle/react-visible/pull/3.

Pull request #3152. (accessed on 12/10/2021). https://github.com/facebook/react/pull/3152.

Pull request #3163. (accessed on 12/10/2021). https://github.com/strapi/strapi/pull/3163.

Pull request #3201. (accessed on 12/10/2021). https://github.com/strapi/strapi/pull/3201.

Pull request #33. (accessed on 12/10/2021). https://github.com/4Catalyzer/graphql-validation-complexity/pull/33.

Pull request #39. (accessed on 12/10/2021). https://github.com/hinaloe/public-toot-viewer/pull/39.

Pull request #4305. (accessed on 12/10/2021a). https://github.com/parse-community/parse-server/pull/4305.

Pull request #4305. (accessed on 12/10/2021b). https://github.com/parse-community/parse-server/pull/4305/commits/be4aef061d7b1967ce020e286304e08a6ac389d2.

Pull request #4561. (accessed on 12/10/2021). https://github.com/facebook/react/pull/4561.

Pull request #4790. (accessed on 12/10/2021a). https://github.com/strapi/strapi/pull/4790.

Pull request #4790. (accessed on 12/10/2021b). https://github.com/strapi/strapi/pull/4790.

Pull request #4822. (accessed on 12/10/2021).https://github.com/parse-community/parse-server/pull/4822.

Pull request #4895. (accessed on 12/10/2021).https://github.com/parse-community/parse-server/pull/4895.

Pull request #49. (accessed on 12/10/2021). https://github.com/edm00se/emoji-transmogrifier/pull/49/commits.

Pull request #5330. (accessed on 12/10/2021). https://github.com/strapi/strapi/pull/5330.

Pull request #5951. (accessed on 12/10/2021).https://github.com/parse-community/parse-server/pull/5951.

Pull request #5 by mahmoud-alfadel/test. (2021). https://github.com/mahmoud-alfadel/test/pull/5. (accessed on 10/04/2021)

Pull request #685. (accessed on 12/10/2021). https://github.com/strapi/strapi/pull/685.

Pull request #714. (accessed on 12/10/2021). https://github.com/facebook/react/pull/714.

Pull request #7160. (accessed on 12/10/2021). https://github.com/sequelize/sequelize/pull/7160.

Pull request #783. (accessed on 12/10/2021a). https://github.com/sequelize/sequelize/pull/783.

Pull request #783. (accessed on 12/10/2021b). https://github.com/sequelize/sequelize/pull/783.

Pull request #844. (accessed on 12/10/2021). https://github.com/markedjs/marked/pull/844. (accessed on 10/04/2021)

Pull request #85. (accessed on 12/10/2021). https://github.com/steelbrain/babel-cli/pull/85.

Pull request #860. (accessed on 12/10/2021). https://github.com/PrideInLondon/pride-london-web/pull/860.

Pull request #91. (accessed on 12/10/2021). https://github.com/slothpixel/ui/pull/91.

Pull request #9224. (accessed on 12/10/2021). https://github.com/electron/electron/pull/9224.

Pull request #991. (accessed on 12/10/2021). https://github.com/strapi/strapi/

pull/991.

PyPi. (2018). *Security pypi.* https://pypi.org/security/. ((accessed on 10/10/2020))

R2c. ((accessed 2020)). https://r2c.dev/.

Raemaekers, S., Van Deursen, A., & Visser, J. (2014). Semantic versioning versus breaking changes: A study of the maven repository. In *Ieee 14th international working conference on source code analysis and manipulation (scam), 2014* (pp. 215–224).

Rahman, M. M., & Roy, C. K. (2014). An insight into the pull requests of github. In *Proceedings of the 11th working conference on mining software repositories* (pp. 364–367).

react - npm. (accessed on 12/10/2021). https://www.npmjs.com/package/react.

report, S. ((accessed 2020)). *77% of 433,000 sites use vulnerable javascript libraries — snyk.* https://snyk.io/blog/77-percent-of-sites-still-vulnerable/. Retrieved from https://snyk.io/blog/77-percent-of-sites-still-vulnerable/

Reporting a vulnerability in an npm package. (accessed on 12/10/2021)https://docs.npmjs.com/reporting-a-vulnerability-in-an-npm-package

Rigby, P. C., & Bird, C. (2013). Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering* (pp. 202–212).

Ruangwan, S., Thongtanunam, P., Ihara, A., & Matsumoto, K. (2019). The impact of human factors on the participation decision of reviewers in modern code review.*Empirical Software Engineering*, *24*(2), 973–1016.

Ruohonen, J. (2018). An empirical analysis of vulnerabilities in python packages for web applications. In *2018 9th international workshop on empirical software engineering in practice (iwesep)* (pp. 25–30).

Sabottke, C., Suciu, O., & Dumitra, T. (2015). Vulnerability disclosure in the age of social media: Exploiting twitter for predicting real-world exploits. In *24th {USENIX} security symposium ({USENIX} security 15)* (pp. 1041–1056).

SAP. (1972). *Sap software solutions — business applications and technology.* https://www.sap.com/canada/index.html. ((accessed on 07/17/2021))

Sarle, W. (1990). Sas/stat user?s guide: The varclus procedure. sas institute. *Inc., Cary, NC, USA*.

Semantic versioning 2.0.0. (2021). https://semver.org/. ((accessed on 07/17/2021))

Semantic versioning 2.0.0. ((accessed 2020)). https://semver.org/.

semver. (2021). *semver - npm.* https://www.npmjs.com/package/semver. ((accessed on 07/17/2021))

semver pypi. (2020). https://pypi.org/project/semver/. ((accessed on 10/10/2020))

sequelize - npm. (accessed on 12/10/2021). https://www.npmjs.com/package/sequelize.

Snyk.io. (2017). *The state of open-source security.* ([Online; Available: https://snyk.io/])

Snyk.io. (accessed on 12/10/2021). *The state of open source security.* https://snyk.io/open-source-security/. Retrieved from https://snyk.io/open-source-security/

Snyk report. (accessed on 12/10/2021). https://snyk.io/blog/77-percent-of-sites-still-vulnerable/.

Soares, D. M., de Lima Júnior, M. L., Murta, L., & Plastino, A. (2015). Acceptance factors of pull requests in open-source projects. In *Proceedings of the 30th annual acm symposium on applied computing* (pp. 1541–1546).

SOF. (2020). *Stack overflow developer survey 2019.* https://insights.stackoverflow.com/survey/2020#overview. ((accessed on 01/12/2021))

Spadini, D., Aniche, M., Storey, M.-A., Bruntink, M., & Bacchelli, A. (2018). When testing meets code review: Why and how developers review tests. In *2018 ieee/acm 40th international conference on software engineering (icse)* (pp. 677–687).

Spadini, D., Palomba, F., Baum, T., Hanenberg, S., Bruntink, M., & Bacchelli, A. (2019). Test-driven code review: an empirical study. In *2019 ieee/acm 41st international conference on software engineering (icse)* (pp. 1061–1072).

Stack overflow developer survey. (2020). https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages-all-respondents. ((accessed on 01/10/2021))

Stack overflow developer survey. (accessed on 12/10/2021). https://insights

.stackoverflow.com/survey/2019.

Staicu, C.-A., Pradel, M., & Livshits, B. (2016). *Understanding and automatically preventing injection attacks on node. js* (Tech. Rep.). Tech. Rep. TUD-CS-2016-14663, TU Darmstadt, Department of Computer Science.

Staicu, C.-A., Pradel, M., & Livshits, B. (2018). Synode. In *Ndss*.

steelbrain/babel-cli@c8c9859. (accessed on 12/10/2021). https://github.com/steelbrain/babel-cli/commit/c8c985925c3513f2dd26241a75d71953dd5e1d39#diff-b9cfc7f2cdf78a7f4b91a753d10865a2.

strapi - npm. (accessed on 12/10/2021). https://www.npmjs.com/package/strapi.

Synopsys. (2019). *Synopsys black duck open source security and risk analysis*.

Thomé, J., Shar, L. K., Bianculli, D., & Briand, L. (2018). Security slicing for auditing common injection vulnerabilities. *Journal of Systems and Software*, *137*, 766–783.

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

Thongtanunam, P., McIntosh, S., Hassan, A. E., & Iida, H. (2015). Investigating code review practices in defective files: An empirical study of the qt system. In *2015 ieee/acm 12th working conference on mining software repositories* (pp. 168–179).

uglify-js - npm. (accessed on 12/10/2021). https://www.npmjs.com/package/uglify-js.

Upgrades most dev dependencies (#1753). (accessed on 12/10/2021). https://github.com/box/box-ui-elements/commit/7c8bde43917e9bef50c38ef5e7af3fe168412b1d#diff-8ee2343978836a779dc9f8d6b794c3b2.

Vasilescu, B., Blincoe, K., Xuan, Q., Casalnuovo, C., Damian, D., Devanbu, P., & Filkov, V. (2016). The sky is not the limit: multitasking across github projects. In *Proceedings of the 38th international conference on software engineering* (pp. 994–1005).

Vu, D.-L., Pashchenko, I., Massacci, F., Plate, H., & Sabetta, A. (2020a). Poster: Towards using source code repositories to identify software supply chain attacks. In *Ccs 20*.

Vu, D.-L., Pashchenko, I., Massacci, F., Plate, H., & Sabetta, A. (2020b). Typosquatting and

combosquatting attacks on the python ecosystem. In *2020 ieee european symposium on security and privacy workshops (euros&pw)* (pp. 509–514).

vuln-regex-detector. (accessed on 12/10/2021). https://github.com/davisjam/vuln-regex-detector#readme.

Walden, J. (2020). The impact of a major security event on an open source project: The case of openssl. In *Proceedings of the 17th international conference on mining software repositories* (pp. 409–419).

Wang, Y., Chen, B., Huang, K., Shi, B., Xu, C., Peng, X., . . Liu, Y. (2020). 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)* (pp. 35–45).

Weißgerber, P., Neu, D., & Diehl, S. (2008). Small patches get in! In *Proceedings of the 2008 international working conference on mining software repositories* (pp. 67–76).

Williams, J., & Dabirsiaghi, A. (2012). The unfortunate reality of insecure libraries. *Asp. Secur. Inc*, 1–26.

Winter, B. (2013). Linear models and linear mixed effects models in r with linguistic applications. *University of California, Merced, Cognitive and Information Sciences*.

Wittern, E., Suter, P., & Rajagopalan, S. (2016). A look at the dynamics of the javascript package ecosystem. In *Ieee/acm 13th working conference on mining software repositories (msr), 2016* (pp. 351–361).

Yang, J., Tan, L., Peyton, J., & Duer, K. A. (2019). Towards better utilizing static application security testing. In *2019 ieee/acm 41st international conference on software engineering: Software engineering in practice (icse-seip)* (pp. 51–60).

Yu, Y., Wang, H., Filkov, V., Devanbu, P., & Vasilescu, B. (2015). Wait for it: Determinants of pull request evaluation latency on github. In *2015 ieee/acm 12th working conference on mining software repositories* (pp. 367–371).

Yu, Y., Wang, H., Yin, G., & Wang, T. (2016). Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information and Software Technology*, *74*, 204–218.

Zapata, R. E., Kula, R. G., Chinthanet, B., Ishio, T., Matsumoto, K., & Ihara, A. (2018). 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)* (pp. 559–563).

Zerouali, A. (2019). *A measurement framework for analyzing technical lag in open-source software ecosystems* (Unpublished doctoral dissertation). PhD thesis, University of Mons.

Zerouali, A., Constantinou, E., Mens, T., Robles, G., & González-Barahona, J. (2018). An empirical analysis of technical lag in npm package dependencies. In *International conference on software reuse* (pp. 95–110).

Zerouali, A., Mens, T., Decan, A., & De Roover, C. (2021). On the impact of security vulnerabilities in the npm and rubygems dependency networks. *arXiv preprint arXiv:2106.06747*.

Zimmermann, M., Staicu, C.-A., Tenny, C., & Pradel, M. (2019a). Small world with high risks: A study of security threats in the npm ecosystem. In *28th {USENIX} security symposium ({USENIX} security 19)*.

Zimmermann, M., Staicu, C.-A., Tenny, C., & Pradel, M. (2019b). Small world with high risks: A study of security threats in the npm ecosystem. *USENIX Security Symposium*.