



Empirical analysis of security vulnerabilities in Python packages

Mahmoud Alfadel¹ · Diego Elias Costa² · Emad Shihab¹

Accepted: 14 December 2022 / Published online: 25 March 2023

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

Abstract

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 of security vulnerabilities much more difficult, as software systems depend on an increasingly high number of packages. Recently, security vulnerabilities in the *npm* ecosystem, the ecosystem of Node.js packages, have been studied in the literature. As different software ecosystems embody 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. In this paper, we present an empirical study of 1,396 vulnerability reports affecting 698 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. In addition, vulnerabilities in packages may affect software projects that depend on them (dependent projects), making them vulnerable too. We study a set of 2,224 GitHub Python projects, to better understand the prevalence of vulnerabilities in their dependencies and how fast it takes to update them. Our findings show that the discovered vulnerabilities in Python packages are increasing over time, and they take more than 3 years to be discovered. A large portion of these vulnerabilities (40.86%) are only fixed after being publicly announced, giving ample time for attackers exploitation. Moreover, we find that more than half of the dependent projects rely on at least one vulnerable package, taking a considerably long time (7 months) to update to a non-vulnerable version. We find similarities in some characteristics of vulnerabilities in *PyPi* and *npm* and divergences that can be attributed to specific *PyPi* policies. By leveraging our findings, we provide a series of implications that can help the security of software ecosystems by improving the process of discovering, fixing and managing package vulnerabilities.

Keywords Python · *PyPi* · Packages · Vulnerabilities · Empirical studies

Communicated by: Rick Kazman, Marouane Kessentini, Yuanfang Cai

This article belongs to the Topical Collection: *Software Analysis, Evolution and Reengineering (SANER)*

✉ Mahmoud Alfadel
alfadelmahmood@gmail.com

Extended author information available on the last page of the article.

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 Node.js 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 and 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 (client projects) update their packages to incorporate 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 (Hejderup 2015; Decan et al. 2018a; Zimmermann et al. 2019) analysed the impact of vulnerabilities in the *npm* ecosystem. Zerouali et al. (2022) found that it takes 26 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 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.

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 (StackOverflow 2020).

Hence, this paper has extended our previous work (Alfadel et al. 2021) replicating and extending our study on security vulnerabilities in Python packages. We conduct an exploratory research to study *security vulnerabilities prevalence and their respective discovery and fix timeline in the Python ecosystem*. In this study, we aim to answer the following research questions (RQs):

- **RQ₁**: How are vulnerabilities distributed in the *PyPi* ecosystem?
- **RQ₂**: How long does it take to discover a vulnerability in the *PyPi* ecosystem?
- **RQ₃**: When are vulnerabilities fixed in the *PyPi* ecosystem?
- **RQ₄**: How long does it take to fix a vulnerability in the *PyPi* ecosystem?
- **RQ₅**: How often are dependent projects affected by vulnerable dependencies?

- **RQ₆**: How long do dependent projects take to update their vulnerable dependencies?

Our extended study provides the following key additions:

- We updated and extended our study of the PyPi security vulnerabilities and increased the vulnerability dataset from 550 security reports to a set of 1,396 reports. Our new dataset covers vulnerabilities reported from 2006 to 2021.
- We extended our analysis by examining the impact of vulnerable packages on a large set of dependent Python projects (RQ₅ & RQ₆).
- We updated our comparison to the npm ecosystem by comparing our findings, where applicable, to a similar study on npm by Zerouali et al. (2022).
- We developed a tool called DEPHEALTH, which illustrates analytical reports of security vulnerabilities that affect Python packages in terms of their discovery and fix lifetime.

To answer our research questions, we analyzed 1,396 vulnerability reports that affect 698 Python packages of which 30,915 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 (Zerouali et al. 2022). 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*, a large number of *PyPi* vulnerabilities (40.86%) 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. To summarize, this paper makes the following contributions:

- We perform the first empirical study to analyse security vulnerabilities in the Python ecosystem. Our study covers 16 years of *PyPi* reported vulnerabilities, affecting 698 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 build a tool called DEPHEALTH, which uses the analysis approach in our study to generate analytical report of security vulnerabilities that affect Python packages.
- We make our dataset of this study publicly available to facilitate reproducibility and future research (Alfadel et al. 2020).

Paper Organization Section 2 presents the motivation of our study. Section 3 presents concepts and terminologies related to software package vulnerabilities, which we adopt throughout our study. Section 4 describes the process of collecting and curating our dataset. In Section 5, 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 6. We present our tool in Section 7. We state the threats to validity and limitations to our study in Section 8. Related work is presented in Section 9. Finally, Section 10 concludes our paper.

2 Motivation

Vulnerabilities in third-party packages are a growing concern for the software developer. GitHub (the report in 2018 Github (2022)) reported that over four million vulnerabilities were raised to the attention of developers of over 500 thousand GitHub repositories. Known examples of package vulnerabilities include the ShellShock and Heartbleed vulnerabilities (Metha 2022), which caused widespread damage to broad and diverse software ecosystems. Durumeric et al. (2014) showed that 24% of the secure sites that adopt the OpenSSL package, the project where the Heartbleed vulnerability originated, were affected and impacted by Heartbleed.

The speed at which ecosystems react to vulnerabilities and the availability of fixes to vulnerabilities is of paramount importance. Several lines of prior works support this intuition. Several studies (Hejderup 2015; Pashchenko et al. 2018; Ponta et al. 2018) encourage developers to use security best practices, e.g., security monitoring, to prevent and detect package vulnerabilities as fast as possible. Other work (Decan et al. 2017, 2018a, 2018b; Zimmermann et al. 2019) focused on studying the impact of package vulnerabilities at the ecosystem level. While these prior studies have made important advances, they have tended to focus on (i) the largest software ecosystems, i.e., npm (NodeJS projects). For example, Decan et al. (2018a) analyzed how and when npm package releases with vulnerabilities are discovered and fixed. Also, Decan et al. (2018b) analyzed npm package releases to explore the evolution of technical lag and its impact. While npm is one of the largest software ecosystems to date, the investigation of npm vulnerabilities is a restricted view of the software development 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 et al. 2015; Decan et al. 2016, 2017, 2019) that show differences across ecosystems. For instance, Decan et al. (2017) found that the 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 from using too many external packages in contrast to JavaScript and R ecosystems. Furthermore, there is also a research gap that relates to (ii) the analysis of the package vulnerability fixes with respect to the dependent projects (downstream clients). Vulnerabilities in packages can only be exploited once used in other dependent projects and once these projects are released and used in production. This is important to understand to what extent dependent projects are affected by vulnerable dependencies.

To bridge these two research gaps, we set out to study the security vulnerabilities prevalence and their respective discovery and fix timelines in the Python ecosystem (PyPi). We compare our analysis to the npm ecosystem study by Zerouali et al. (2022). We incorporate this analysis in the first four research questions (RQ₁ - RQ₄). Then, we expand our analysis of vulnerabilities in the PyPi ecosystem to encompass the dependent projects. We examine the releases of a large set of GitHub open-source Python projects, to better understand the impact of vulnerable packages on dependants, which we demonstrate in RQ₅ and RQ₆.

3 Concept and Terminology

In this section, we present concepts and terminologies related to software package vulnerabilities, which we adopt throughout our study.

3.1 Vulnerability Lifecycle

The lifetime of a vulnerability typically goes through various stages, according to when a vulnerability was first introduced, discovered, and publicly announced (Kula et al. 2018). To ground our study, we use the various stages and define dates specific to a package vulnerability.

Package Side Figure 1 illustrates the lifecycle of a vulnerability of package P (the lower part of the figure). We break the lifecycle into four main stages:

- *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. Figure 1 shows the vulnerability of package P being introduced in the $P_{V1.0.0}$ release.
- *package vulnerability discovery date* indicates the date on which the package vulnerability was discovered and reported. Figure 1 shows the vulnerability of package P being discovered after the $P_{V1.0.1}$ release (orange star).
- *package vulnerability fix date* indicates the release date of the first fixed version of the package vulnerability. In Fig. 1, $P_{V1.1.0}$ is the first fixed release of package P.
- *package vulnerability publication date* marks the date when the vulnerability information (and report) was publicly announced. The vulnerability publication may happen before or after the fixed version ($P_{V1.1.0}$) is released.

Dependent Project-Side Prior work suggests that lags in adopting a fixed version of vulnerable packages could result from migration efforts (Kula et al. 2018). Thus, to quantify this effort, we study the extent to which dependent projects (client) are using vulnerable packages and how long it takes to use the package-side fixing release. Developers of the vulnerable package fulfilled their responsibility, thus, the adoption responsibility is left to

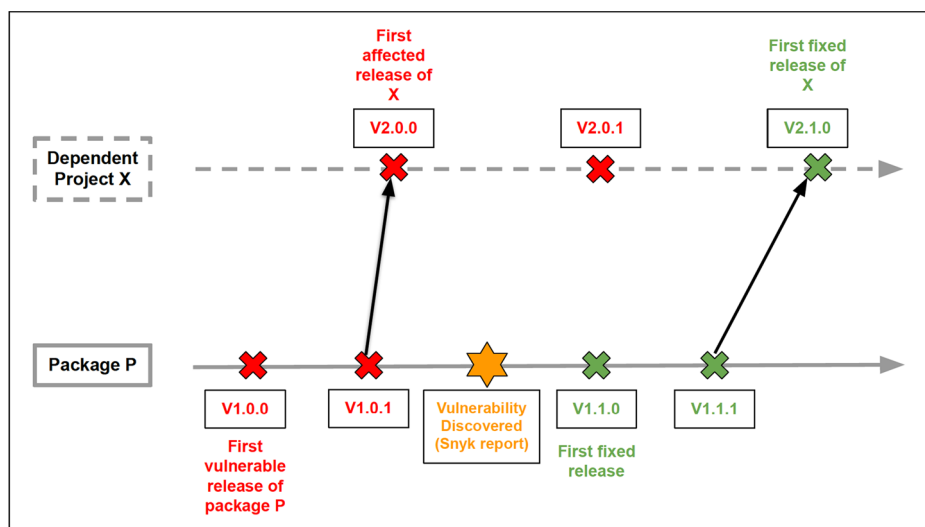


Fig. 1 The lifetime of vulnerability discovery and fix of package P and dependent project X over time. Red and green releases indicate whether releases are vulnerable or not

the dependent project maintainers. Figure 1 shows the timeline of a dependent project (X), illustrating the first affected release of the project as well as its first release that adopts a fixed version of the package P. As illustrated in the figure, project X suffers a lag in the adoption of a package-side fixing release. The first affected release of project X is $X_{V2.0.0}$ which relies on Package $P_{V1.0.1}$, i.e., X was vulnerable in its version V2.0.0 due to its dependency ($P_{V1.0.1}$). To mitigate the vulnerability, package P creates a new branch, i.e., the minor branch, which includes the package-side fixing release ($P_{V1.1.0}$). Project X finally adopts the new release of package P in its version V2.1.0. We consider that project X has a lag because it actually skipped the package-side fixing release ($P_{V1.1.0}$). Instead, client X adopted the next release ($P_{V1.1.1}$). A possible cause of lags is the potential migration effort

Buffer Overflow

Affecting `pillow` package, versions `[6.2.2)`

Share ▾

How to fix?

Upgrade `Pillow` to version 6.2.2 or higher.

Overview

`Pillow` is a PIL (Python Imaging Library) fork.

Affected versions of this package are vulnerable to Buffer Overflow in `libImaging/PcxDecode.c`.

References

- [GitHub Commit](#)
- [RedHat Bugzilla Bug](#)

(a) pillow vulnerability report with high severity.

Snyk ID	SNYK-PYTHON-PILLOW-541324
Published	10 Jan 2020
Disclosed	2 Jan 2020
Credit	Unknown

(b) The report dates of the pillow vulnerability report.

Fig. 2 Example of a vulnerability report extracted from Snyk.io for the package pillow

needed to switch branches, i.e., from $P_{V1.0.1}$ to $P_{V1.1.1}$. The migration effort for major or minor changes may include breaking changes or issues in the release cycle of the project X.

3.2 Package Vulnerability Example

Figure 2 show a practical case of a vulnerability report extracted from Snyk.io (the vulnerability dataset we used in our study). The vulnerability report is for a library that provides an internal representation and powerful image processing capabilities, i.e., pillow. The vulnerability report contains detailed information regarding the identified problem, its severity, and a proof-of-concept (references) to confirm the threat. Figure 2a shows that pillow was vulnerable to high severity. Also, the report contains information related to the vulnerability lifecycle. As shown in Fig. 2b, the vulnerability was disclosed on Jan 2, 2020, and published on Jan 10, 2020.

4 Methodology

In this section, we present the methodology of our study. An overview is also provided in Fig. 3. In particular, we explain how we collect and prepare the data used to investigate our research questions.

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 and Nickolls 2018), and the security vulnerabilities from the Snyk.io dataset (Snyk.io 2017). Our dataset collection considers *PyPi* packages and vulnerability reports that are available till June, 2021, i.e., we collect all vulnerability reports that were published before June, 2021.

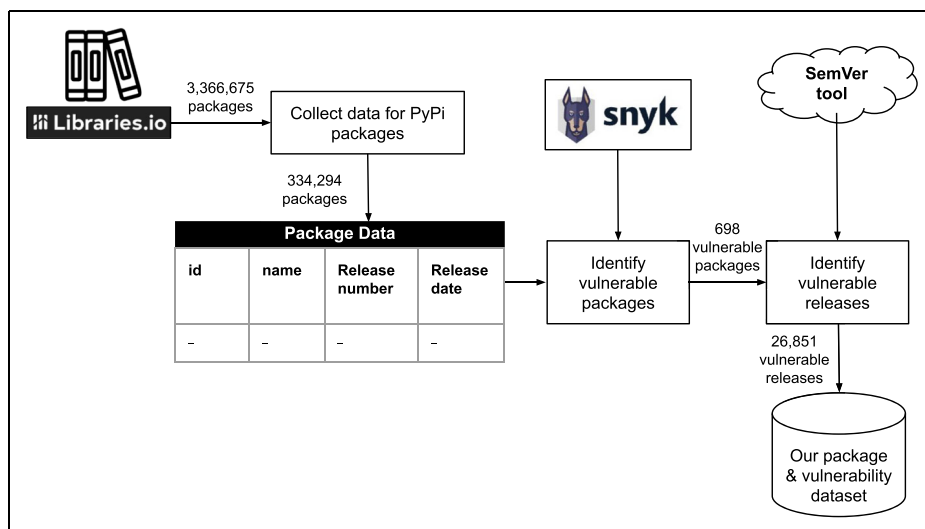


Fig. 3 An overview of our study design showing the process of collecting our package and vulnerability dataset

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.

To collect the vulnerabilities for the *PyPi* packages, we resort to the dataset provided by Snyk.io (Snyk 2020a). 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 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 four possible values, critical, high, medium, and low, which are assigned manually by the Snyk.io team based on the Common Vulnerability Scoring System (CVSS) (Allodi and 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 2020). In the particular example of Table 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 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

Table 1 Example of a security report extracted from Snyk.io for the `pillow` package

Information	Example
Vulnerability type	Buffer Overflow
Affected package name	<code>pillow</code>
Platform type	<i>PyPi</i>
Vulnerable constraint (affected versions)	$< 6.2.2$
Vulnerability Discovery date	03 Jan, 2020
Vulnerability Published date	10 Jan, 2020
Severity level	High
Remediation	$\geq 6.2.2$

Table 2 Descriptive statistics of the *PyPi* dataset

Source	Stats	#
Libraries.io	<i>PyPi</i> packages	334,294
	Versions of <i>PyPi</i> packages	2,955,586
Snyk.io	Security reports on <i>PyPi</i>	1,396
	Corresponding vulnerable packages	698
	Versions of vulnerable packages	30,915
	Affected versions by vulnerability	26,851

version present in the list of fixed versions. In the example of Table 1, the first fixed version is the package 6.2.2.

Our initial dataset contains 1,483 vulnerability reports on the *PyPi* packages. From this original set, 68 vulnerabilities do not match any packages in the Libraries.io database and were removed from our analysis. We also removed 19 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 1,396 vulnerability reports. Such reports affect 698 Python packages in *PyPi*. Note that these 698 Python packages have released a total of 30,915 versions, in which, according to the vulnerable constraint of reports, 26,851 versions contain at least one reported vulnerability. Table 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 Zerouali et al. (2022). The *npm* dataset contains 2,786 vulnerabilities which affect 1,672 *npm* packages. We can observe that the *npm* dataset has more than double the number of vulnerability reports and affected packages in the *PyPi* dataset, given that *npm* has had the highest growth rate in terms of packages amongst all known programming languages (Libraries.io 2021).

5 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.

5.1 RQ₁ How are Vulnerabilities Distributed in the *PyPi* Ecosystem?

Motivation Prior work reported a steady growth of packages in software ecosystems (Decan et al. 2018b, 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*.

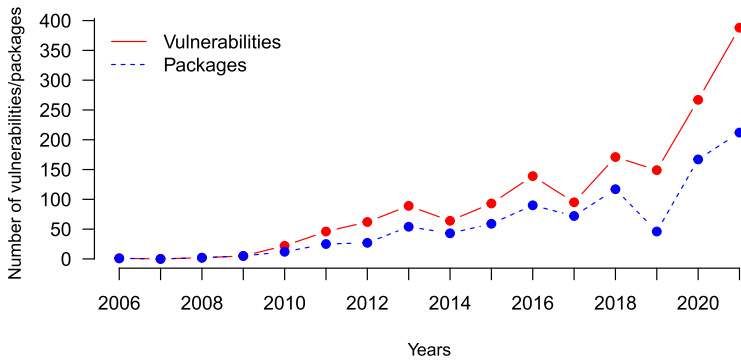


Fig. 4 Introduction of vulnerabilities and packages being affected per year

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 affected packages 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 1) 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) (MITRE 2020), aiming at categorizing vulnerabilities based on the explored software weaknesses (e.g. Buffer Overflow). Currently, CWE contains a community-developed a list of more than 700 common software weaknesses. We examine the frequency of vulnerability types to establish a profile of the vulnerabilities in the *PyPi* ecosystem. In addition, we also break our analysis by severity level to investigate how the threat levels are distributed in each vulnerability type.

Findings Figure 4 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 2014, in the middle of this ecosystem lifetime, 43 packages were discovered to be vulnerable, in 2021 this number increased five-fold, i.e., 212 vulnerable packages were newly discovered.

Figure 5 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 64.03% of all vulnerabilities, followed by high severity vulnerabilities representing 25.42% of our dataset. Still, a non-trivial portion (4.39%) of the vulnerability reports were classified as critical vulnerabilities. These findings are worrisome to the *PyPi* community, as such vulnerabilities

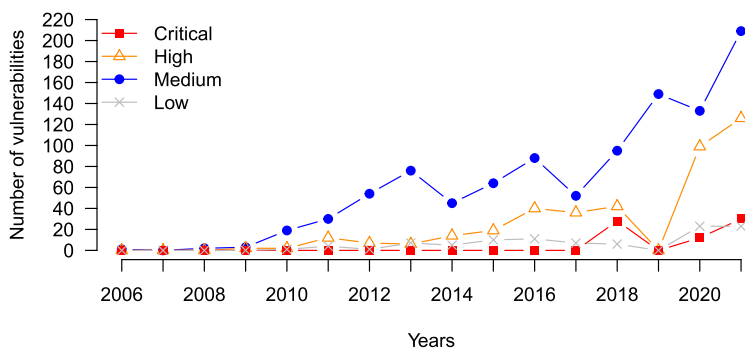


Fig. 5 Introduction of vulnerabilities per year by the severity levels: critical, high, medium, and low

have a higher chance of being exploited, i.e., allow an attacker to execute malicious code and damage the software.

Figure 6 shows bean plots of three distributions: the number of versions of the 698 vulnerable *PyPi* packages in our dataset (Fig. 6a), the number of affected versions in such vulnerable packages (Fig. 6b), and the percentages of vulnerable versions in the packages (Fig. 6c). We observe that most packages have dozens of versions (median number of versions is 40), and tend to have, on median, 37 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 6c shows that half of the packages have at least 83.64% of their versions affected by a vulnerability, when a vulnerability is first 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

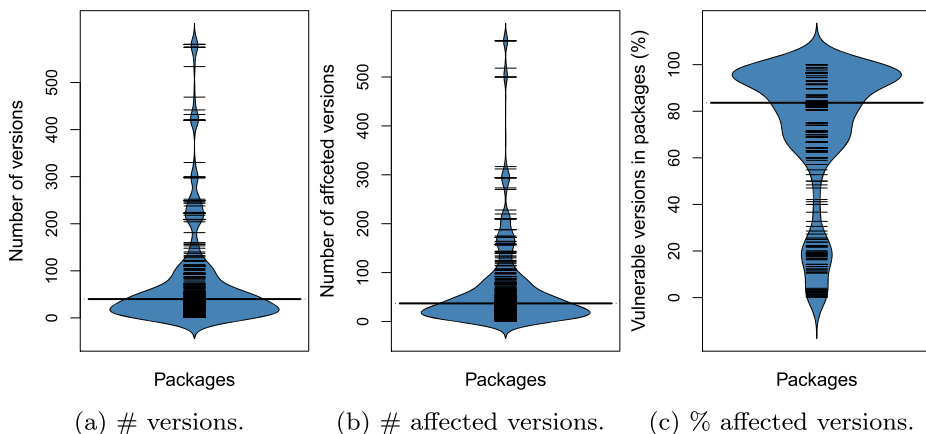


Fig. 6 Distribution of versions and affected versions of the 698 vulnerable packages of our dataset. In median, packages have 40 versions and 37 affected versions once a vulnerability is discovered

Table 3 Ranking of the 10 most commonly found vulnerability types (CWE) in *PyPi*

Rank	Vulnerability type (CWE)	Freq.	Frequency by severity			
			Critical	High	Medium	Low
1	Cross-Site-Scripting (XSS)	266	1	27	227	11
2	Denial of Service (DoS)	151	0	37	107	7
3	Information Exposure	139	4	21	100	14
4	Arbitrary Code Execution	112	16	53	42	1
5	Access Restriction Bypass	41	3	12	24	2
6	Regular Expression Denial of Service (ReDoS)	29	0	14	12	3
7	Improper Input Validation	28	0	8	2	18
8	Directory Traversal	27	1	13	4	9
9	Remote Code Execution (RCE)	23	1	12	10	0
10	Authentication Bypass	21	2	7	12	0

affected by 119 distinct CWEs, the majority of the discovered vulnerabilities (51.79%) are concentrated on 10 main types. Table 3 shows the distribution of the vulnerabilities over the 10 most commonly found CWEs. As we can see, XSS is the most common CWE with 266 vulnerabilities. Also, we observe that most of the XSS vulnerabilities are of medium severity. For the remaining CWEs, the proportion in each type varies from 21 vulnerabilities of type Authentication Bypass to 151 of type Denial of Service (DOS). 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, with a non-trivial proportion of critical vulnerabilities.

Comparison to the *npm* Ecosystem The vulnerabilities found in *npm* (Zerouali et al. 2022) followed a similar distribution to our findings in the *PyPi* ecosystem. In *npm*, a) the newly 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) Directory Traversal and XSS was found to be the most common vulnerability among *npm* vulnerabilities (i.e., 322 and 331 occurrences, respectively).

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 ten vulnerability types, with Cross-Site-Scripting (XSS) being the most common.

5.2 RQ₂: 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 4). 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 into the time it takes to discover the vulnerabilities, we conduct a survival analysis method (a.k.a. event history analysis) (Aalen et al. 2008). Our empirical analysis in the RQs relies on the statistical technique of survival analysis (a.k.a. event history analysis (Kaplan and Meier 1958)). Such technique is useful for our analysis because it models “time to event” data with the aim to estimate the survival rate of a given population, i.e., the expected time duration until the event of interest occurs (e.g., death of a biological organism, failure of a mechanical component, recovery of a disease). Survival analysis models take into account the fact that some observed subjects may be “censored” either because they leave the study during the observation period, or because the event of interest was not observed for them during the observation period. A common non-parametric statistic used to estimate survival functions is the Kaplan-Meier estimator. This test has also been used in previous studies (e.g., Decan and Mens 2019; Decan et al. 2018a).

Findings Figure 7 presents the survival probability for the vulnerability before it gets discovered. The Left-side plot of Fig. 7 reveals that half the vulnerabilities took 39 months to be discovered. In practice, this shows that vulnerabilities are not discovered early in the project development. Also, this long process for discovering vulnerabilities explains why a single vulnerability tends to affect dozens of package versions once it is first discovered (RQ1).

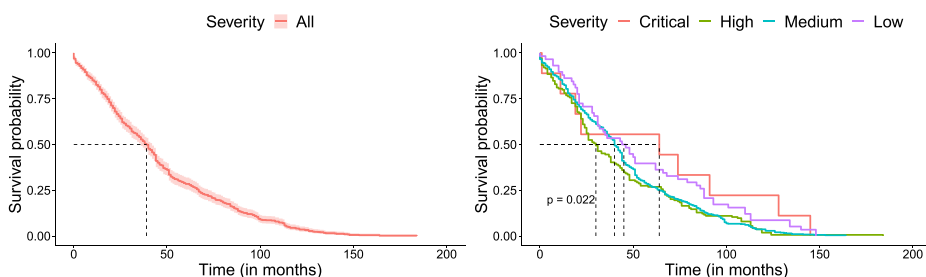


Fig. 7 Kaplan-Meier survival probability for package vulnerabilities to get discovered for all vulnerabilities (left-side plot) and for vulnerabilities broken by severity (right-side plot)

Since vulnerabilities impact packages at different severity levels, we break down the analysis of discovered vulnerabilities by their severity. The right-side plot of Fig. 7 presents the survival probability for the event “vulnerability is discovered” by their severity. We found statistically significant differences when comparing the different severity levels. We confirm this result by using the log-rank statistical method (Bewick et al. 2004) to investigate the statistical significance of the results with a confidence level 95% (p-value = 0.022). Critical vulnerabilities take longer to be discovered (median = 64 months), followed by low severity vulnerabilities (median = 45), medium severity vulnerabilities (median = 40), and high severity (median = 30). One possible explanation for vulnerabilities with critical severity taking significantly longer is that such cases are not immediately reported and disclosed, given their harmful impact on the software and higher chance of being exploited. Also, such cases involve higher complexity (higher CVSS scores), which may require longer to be discovered and validated.

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 26 months in the *npm* ecosystem, considerably sooner than the 39 months required for *PyPi* package vulnerabilities. Given the popularity of Javascript programs, *npm* became a prime target for attackers (Zimmermann et al. 2019), 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. Critical vulnerabilities take longer (64 months) to be discovered than high (30 months) and medium severity (40 months) vulnerabilities.

5.3 RQ3: When are Vulnerabilities Fixed in the *PyPi* Ecosystem?

Motivation Vulnerable packages remain affected even after they are discovered (Decan et al. 2018a; Li and 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 et al. 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:

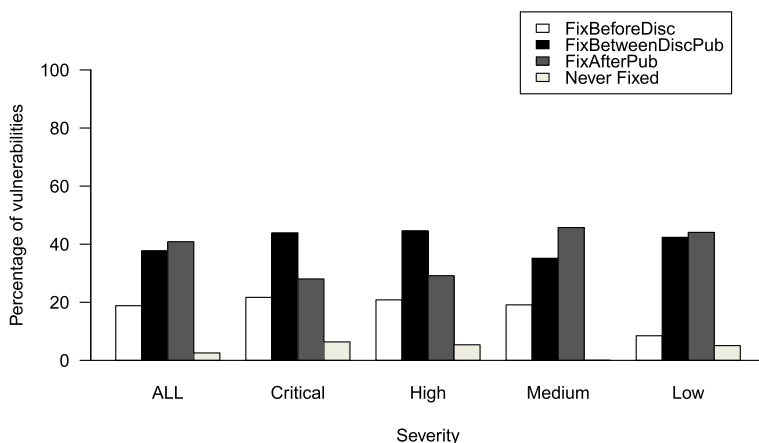


Fig. 8 Percentages of vulnerabilities according to the release time of the first fixed version by severity

“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.

Findings Figure 8 shows the distribution of vulnerabilities according to the four stages in which the first fixed version was released. We can observe that 40.86% of vulnerabilities were fixed after the vulnerability has been made public, with the observation being more noticeable for vulnerabilities of medium and low severity ($C = 28.04\%$, $H = 29.17\%$, $M = 45.73\%$, and $L = 44.07\%$). Such results indicate that a non-negligible proportion of the *PyPi* package vulnerabilities become public before having any patch addressed to fix them.

For the remaining vulnerabilities, 18.82% 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, 37.77% 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 a substantial share of *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 a large number of vulnerabilities (40.86%) 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 Section 6.

A substantial share of vulnerabilities (40.86%) are only fixed after the vulnerability is made public, while 18.82% are fixed before the vulnerability is first discovered, and 37.77% are fixed between the discovery and publication dates.

5.4 RQ4: How long does it take to fix a vulnerability in the *PyPi* Ecosystem?

Motivation So far, we have observed that the vast majority of vulnerabilities are fixed after the vulnerability is reported to be discovered, either in between discovery and publication (37.77%) or after the vulnerability publication (40.86%). 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 (18.82%). 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 5.2, we use the Kaplan-Meier estimator (Kaplan and 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.

Findings Figure 9 presents the survival probability for the vulnerabilities to be fixed after being discovered. As we can observe from the left-side plot, 50% of the vulnerabilities are fixed 2 months following the discovery time. Also, we can observe that there is a small share (7.37%) of those vulnerabilities that still take more than a year to get fixed after being discovered.

The right-side plot of Fig. 9 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 severity levels with a confidence level 95% (p -value = 0.05).

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)

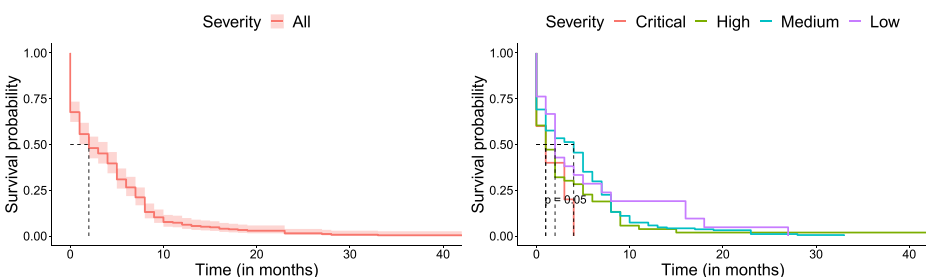


Fig. 9 Kaplan-Meier survival probability for vulnerable packages to get fixed after being discovered and reported

and are not actively maintained, with the latest version been released two years ago. 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 11 days to fix vulnerabilities, regardless of their severity. In *PyPi*, we found that *PyPi* vulnerabilities take a median of 2 months to release a fix after the vulnerability has been discovered.

Vulnerabilities in *PyPi* take, on median, 2 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.

5.5 RQ₅: How Often are Dependent Projects Affected by Vulnerable Dependencies?

Motivation So far, our research has focused on assessing the vulnerability lifecycle at the package ecosystem level. However, vulnerabilities in packages can only be exploited once used in other dependent projects and once these projects are released and used in production. In this RQ, we want to expand our analysis of vulnerabilities in the *PyPi* ecosystem to encompass the dependent projects. To what extent are dependent projects affected by vulnerable dependencies? How often releases carry vulnerable dependencies and what share packages affect dependent projects the most? We examine the releases of a large set of GitHub open-source Python projects, to better understand the impact of vulnerable packages on dependants.

Approach To study the impact of vulnerabilities that affect ecosystem packages from the end-user projects' point of view, we need to collect a dataset of projects. To do so, we collect historical data of Python GitHub projects using the GitHub API.

Since GitHub hosts projects that are not yet immature or of sufficient complexity to warrant analysis, we apply several criteria for selecting a set of high-quality projects. We first choose projects with ten stars at least, as such projects are considered of interest to the community. Prior work reported that a 10 stars threshold is a reasonable mechanism to remove most personal and toy projects unlikely to be relevant for empirical studies (Dabic et al. 2021). Also, a survey of over 700 developers shows that most developers consider the number of stars before using or contributing to GitHub projects (Borges and Valente 2018). After that, we chose non-forked applications with at least 100 commits and 2 developers (with the latest commit pushed after June 1, 2021), as similar filtering criteria were used in prior work (Kalliamvakou et al. 2014; Abdalkareem et al. 2017, 2020; Chowdhury et al. 2021). Finally, using the GitHub meta-data that we obtain after requesting the API, we select projects that use *PyPi* platform as the dependency management tool and that have produced at least one release. A release of a GitHub project exposes release notes and links to download the software or source code from GitHub. This helps us to specifically collect the list of project dependencies at each release.

After this curation process, we are left with 2,224 projects for our investigation. We show in Table 4 the statistics of the projects under a variety of metrics. Our selected dataset contains popular Python projects (median stars of 260), with history of long development (median of 6.24 years), and developed by multiple collaborators (median of 26 collaborators).

Table 4 Statistics of the 2,224 projects selected for our study

Statistics	Min	Median	Average	Max
Stars	10	260	1,532.37	100,709
Project age (years)	3	6.24	6.52	11.83
Number of commits	100	1047	2813.73	157,121
Number of contributors	2	26	51.01	449
Number of files	16	3,317	12,611.02	186,784.9

After selecting the projects for our study, we mine and analyse their releases. For each project, we obtain all releases by using the GitHub API. Then, for each release, we search for the dependency file (e.g., requirements.txt), where the list of project run-time dependencies are listed. After gathering the files, we parse them to extract the list of project dependencies along with their defined version constraints. For each project, we analyse each produced release in the project and check whether the release lists any vulnerable package versions that have been reported and published. Note that in some cases, the dependency is defined with a dynamic version constraint. Hence, to determine the exact version of the package to be installed for each release, we use the dependency constraint resolver (Semver 2020). For this analysis, we consider a release to be vulnerable if it contains a vulnerable dependency as a direct dependency, thus, we disregard vulnerabilities from transitive dependencies. Project maintainers have full control over the direct dependencies of their project and have to monitor their dependencies to be aware of vulnerabilities that may expose their project.

Findings Out of the 2,224 projects in our dataset, 1,237 (55.62%) projects have at least one release depending on vulnerable dependencies. Hence, the majority of the Python projects we investigate are affected by vulnerabilities in the *PyPi* ecosystem.

To get a better understanding of the impact of vulnerable dependencies on the project release productions, we plot in Fig. 10 the bean plots of three distributions: the number of project releases of the 1,237 vulnerable Python projects in our dataset (Fig. 7a), the number

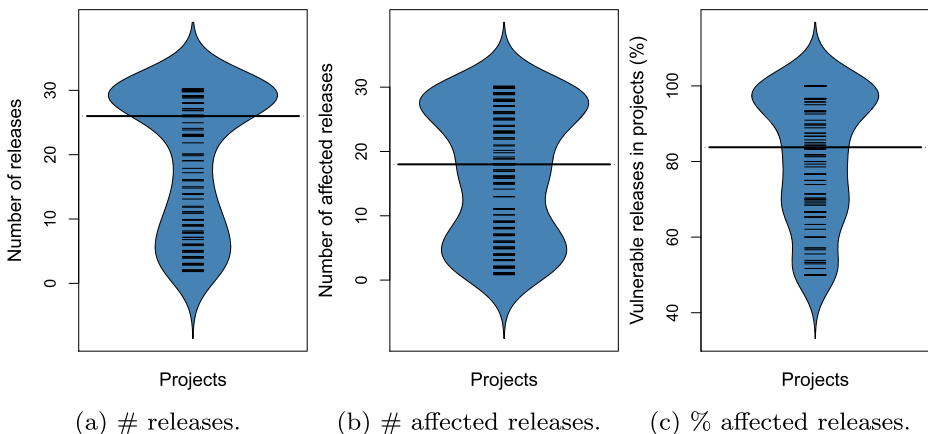


Fig. 10 Distribution of releases and affected releases of the vulnerable dependent projects in our dataset. On median, projects have 26 releases and 18 affected releases

of affected project releases in such vulnerable projects (Fig. 7b), and the percentages of vulnerable releases in the projects (Fig. 7c). We observe that most projects have dozens of releases (median of 26 releases), and tend to have, on median, 18 vulnerable releases. The affected releases represent an alarmingly high proportion of all releases in a project. Figure 7c shows that the percentage of releases affected by a dependency vulnerability in a project is 83.77%, on median.

While most releases in affected projects contain vulnerable dependencies, the affected releases may have been released in the initial development phase of the projects, where the security concern was not at its peak. To investigate this, we analyse how many of the 1,237 projects have a vulnerable release as its most recent (latest) release. We found that, out of the 1,237 projects, 58,72% are affected by at least one vulnerable dependency in the latest project release. This indicates that the problem of vulnerable dependencies is not relegated to the initial phases of the project. Rather, the majority of projects are exposed to vulnerable dependencies in the most recent project release.

Within every project, there are dozens of dependencies, however, not all of them are exposing the project to security vulnerabilities. Hence, we conduct an analysis to quantify the share of vulnerable packages over all project dependencies in the affected 1,237 projects. To that aim, we compute the number/share of vulnerable dependencies in each project release and then aggregate this information across project releases, by calculating the median number of vulnerable dependencies. Figure 11 shows three distributions: the median number of project dependencies taken with respect to the the project releases (Fig. 8a), the median number of affected project dependencies in such vulnerable projects (Fig. 8b), and the percentages of vulnerable dependencies in the projects (Fig. 8c). We observe that most projects have a few dependencies (median number of dependencies is 16), and tend to have, on median, only a single vulnerable dependency. The affected dependencies represent only a very small portion of all dependencies in a project. Figure 8c shows that the percentage of vulnerable dependencies is 9.52%, on median. Finally, when considering the severity level of dependency vulnerabilities that affect the dependent projects, we find that the majority of vulnerabilities are of low (46.75%) and medium severity (44.034%).

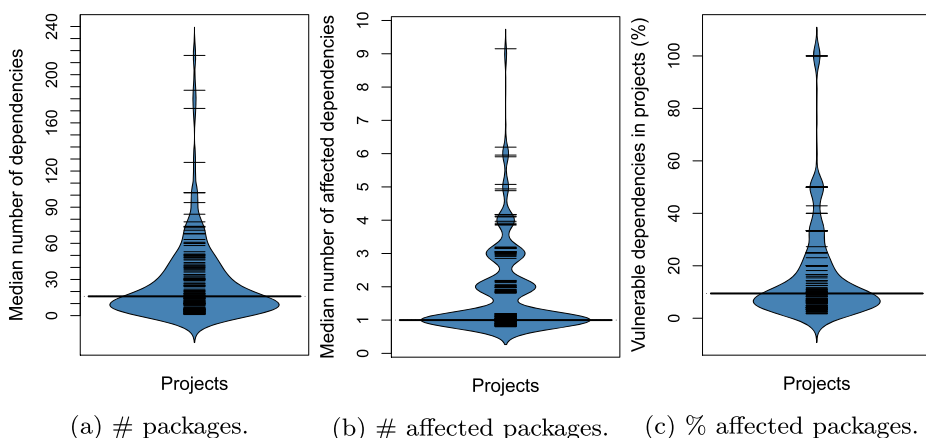


Fig. 11 Distribution of packages and affected packages of the vulnerable dependent projects in our dataset. On median, project releases have 16 packages and only 1 vulnerable package

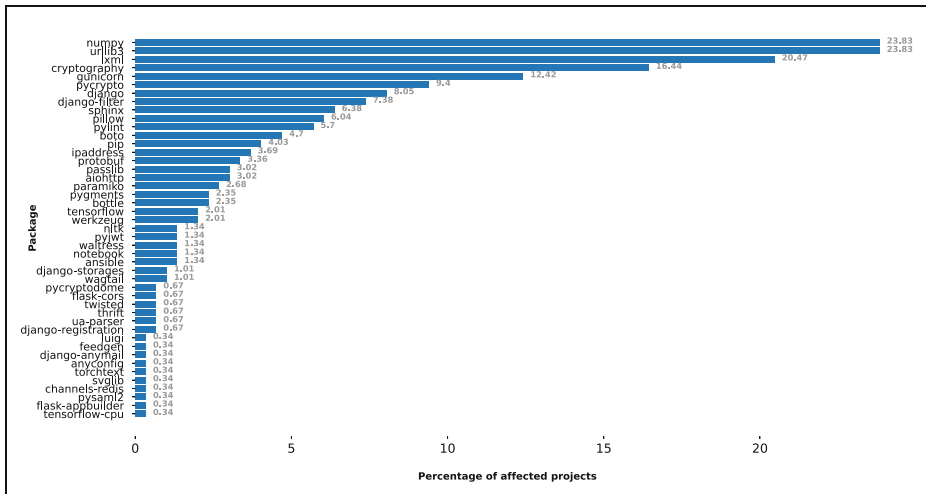


Fig. 12 Percentage of dependent projects affected by vulnerable packages

To gain more insights into our results, we consider two main analyses. First, we look at vulnerable packages dependent projects depend on, which allows us to assess how popular packages may affect an ecosystem of dependent projects. To do this, we first obtain the list of packages affecting the dependent project in our dataset. Then, for each package, we count how many dependent projects are affected by each package. This analysis benefits both package maintainers and project developers, e.g., by increasing their awareness of the most vulnerable packages affecting dependent projects. Second, we study the domain of packages affecting dependent projects. Such analysis will help package maintainers and project developers understand if certain types of packages offer more risk of vulnerabilities given their domain and functionality.

While more than 600 packages are reported to be vulnerable, less than 7% of the packages tend to affect most of the dependent projects in our dataset. Figure 12 shows the percentage of dependent projects affected by each vulnerable package. From the figure, we observe that only 45 packages (6.4% out of 698) affect the dependent projects in our dataset. Moreover, we can observe that the majority (35 out of 45) of packages affect between 1% - 5% of the dependent projects. However, only 10 packages affect > 5% of the projects (i.e., between 6% - 24% of the dependent projects). We looked at the domain of the top 10 packages affecting dependent projects. Table 5 shows a list of the 10 packages and some information to help us better understand the packages' use case, e.g., their domain/functionality and frequency in the dependent project. From the Table, we observe that packages of different domains are common in the affected projects. For example, the vulnerable packages numpy and urllib3 are used in 23.83% of the affected projects. Such packages are common because they provide basic but essential functionalities that support projects from different domains. For instance, the numpy package supports large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions. The urllib3 package is an HTTP client for Python connection pooling, and it helps with client-side SSL/TLS verification, and file uploads with multipart encoding.

Table 5 List of 10 packages with their domain and percentage of dependent projects affected by the packages

Vulnerable package	Domain	% Affected projects
numpy	Support large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions.	23.83
urllib3	HTTP client with thread-safe connection pooling.	23.83
lxml	XML processing library combining libxml2/libxslt with the ElementTree API.	20.47
cryptography	Cryptographic recipes and primitives to Python developers.	16.44
gunicorn	Python HTTP Server for UNIX.	12.42
pycrypto	Cryptographic modules for Python (e.g., SHA256) and various encryption algorithms (AES, DES, RSA, ElGamal, etc.)	9.4
django	Web framework that follows the model–template–views architectural pattern.	8.05
django-filter	Reusable Django application for allowing users to filter querysets dynamically.	7.38
sphinx	Python documentation generator.	6.38
pillow	Python image processing library.	6.04

Comparison to the npm Ecosystem Similar to the npm study, our study shows that the majority of dependent projects are affected by vulnerabilities in their dependencies. Our findings show that more than half of Python dependent projects are affected by vulnerable dependencies. In npm, 79% of npm dependent projects have at least one vulnerable dependency. In both ecosystems, one single vulnerable package could be responsible for exposing most of dependent projects to dependency vulnerabilities.

The majority (55.62%) of the dependent projects depend on vulnerable dependencies. Affected projects have, on median, 83.77% of their releases affected. In most cases, projects are affected by just a single vulnerable dependency.

5.6 RQ₆: How Long Do Dependent Projects Take to Update their Vulnerable Dependencies?

Motivation A dependent project should keep its dependencies updated, especially for security updates. When a vulnerable package publishes a new version that includes a security fix, dependent projects should pull the fixed version. Unfortunately, this is not always the case since dependency constraints may limit the range of versions that a project can depend upon. Hence, in this RQ, we examine how long dependent projects remain depending on a vulnerable package version. Doing so is important to provide insights about the prioritization of fixing vulnerable dependencies of a dependent project.

Approach To analyse the time it takes to fix a vulnerable package in a dependent project, we focus on the affected 1,237 dependent projects in our dataset. For those projects, we analyse the vulnerable releases, i.e., for each vulnerable release, we extract the affected

dependencies in the release (dependencies with vulnerable versions). Then, for each vulnerable dependency, we search for the earliest project release that contains a non-vulnerable version of the corresponding dependency. Finally, we calculate the time difference between both releases. For example, if a project release defines a vulnerable version of package A at release 5, and it only updates to a fixed version at release 7, we then calculate the time difference between release 5 and release 7.

Findings Figure 13 presents the Kaplan-Meier survival curves of the probability for vulnerable packages in dependent projects to get fixed. From the figure, we observe that dependent projects take a considerably long time to update their dependency vulnerabilities to fixed versions (median = 7 months), regardless of the severity level. In fact, this is more than the time needed to fix vulnerabilities in the corresponding upstream packages, which is two months (see RQ4). This indicates that dependent projects still need to constantly monitor their dependencies, especially for security fixes.

As outlined in our approach, our analysis of dependent projects is based on their releases. Hence, the release cycle may have an impact on the time to update a vulnerable package in the dependent project. To shed light on the release cycle impact, we analyse how many releases are produced before a vulnerable package version become updated to a non-vulnerable version in the newer release, i.e., how many releases are produced between the release that contain a vulnerable package version and the release with the corresponding non-vulnerable version of the package.

On median, we found that it takes only 2 releases of the project to update to a non-vulnerable version, i.e., the vulnerable dependency is only updated in the second release produced after the vulnerable release. Nevertheless, we found that it may take long time for the project to produce a new release (median = 3 months). This indicates that the release cycle may cause a lag in the vulnerable package updates, especially if the project has a release cycle.

Comparison to the npm Ecosystem Our findings show that Python dependent projects take, regardless of severity levels, a median of 7 months to update their vulnerable dependencies. The analysis related to the time required to fix a vulnerable dependency in a dependent project was not conducted by the npm study, and hence, it is not applicable to make such a comparison.

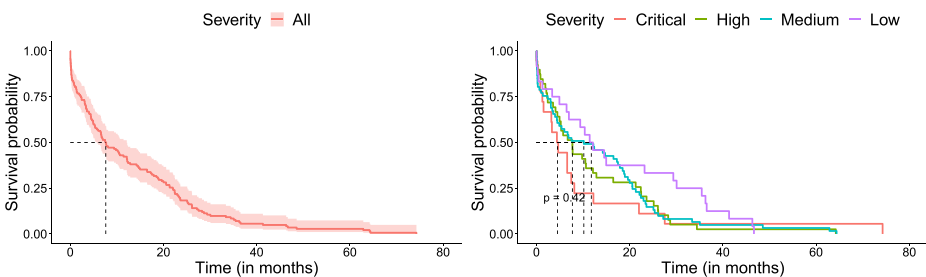


Fig. 13 Kaplan-Meier survival probability for vulnerable packages in dependent projects to get fixed

Half of the dependent Python projects take a long time, a median of 7 months, to update their vulnerable dependencies to a fixed version. The project release cycle may cause such projects to remain vulnerable even if fixed versions are available; it takes 2 releases before the vulnerable package is updated.

6 Discussion and Implications

In this section, we discuss more details about our results with comparison to the *npm* ecosystem (Section 6.1). Then, we highlight the implications of our study to researchers and practitioners (Section 6.2).

6.1 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 and 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 et al. 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 (Thomé et al. 2018; Johari and Sharma 2012); and 2) the community efforts in taming this well-known vulnerability, as identifying XSS has been a top concern by 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

et al. 2018a). Contrasting to these findings, we found 40.86% 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 (NPM 2020a), 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 (Python 2020). In this issue, the vulnerability could cause serious problems (high-severity level) through a Use-After-Free (UAF) (cwe.mitre 2020) 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 (Snyk 2020b).

6.2 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 (RQ₂). 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 (Google 2020; Lu et al. 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 et al. (2020a, 2020b) for *PyPi*, and Synode (Staicu et al. 2016), NoRegrets (Mezzetti et al. 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* (RQ₁).

***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 (RQ₃), and took a couple of months to be fixed and released (RQ₄). 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 (ISC 2020). 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 (RQ₄). A relatively new idea introduced by 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 et al. 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* (Lodash 2020) and *PyPi* (Pillow 2020) 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 (RQ₄). This metric will give package users insights about the prioritization of fixing vulnerabilities of the package. We developed a tool called DEPHEALTH (Section 7), which utilizes our analysis approach in this study to generate analytical metrics of security vulnerabilities that affect Python packages.

***PyPi* dependent projects should employ tools to audit vulnerabilities that affect their dependencies.** Our findings show that the Python dependent projects take a significant long time before the vulnerable dependency gets fixed, and they are affected by just a single vulnerable dependency (RQ₅ & RQ₆). This result illustrates that maintainers of dependent projects should continuously monitor their dependencies by integrating automated tools to quickly notify them of vulnerabilities in their project dependencies. Recently, GitHub acquired Dependabot tool (Dependabot 2020), a tool that tracks vulnerabilities in several ecosystems. Maintainers of dependent projects can adopt Dependabot, and utilize its features, e.g., prioritize the notifications for specific set of affected dependencies in the project, given that only 9.52% of all dependencies are responsible for vulnerabilities that affect the dependent projects. Moreover, 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 2020b), 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.

While projects depend on thousands of packages, project maintainers should pay attention only to a few specific packages that are responsible for most affected projects. Our results (RQ₅) show that the packages with the most vulnerabilities affecting the dependent projects are utility packages, which are used by many different types of projects. Therefore, project developers need to pay higher attention to specific packages (e.g., *numpy*, *urllib3*, *django*) to track their updates and security issues. Also, maintainers of those packages need to be responsive and seriously consider finding and fixing security issues as fast as possible to prevent dependent projects from being impacted. Finally, project maintainers can try to prioritize updates of those packages to prevent their projects from potential risks.

Project maintainers should provide better strategies for producing releases to consider a timely update of vulnerable dependencies. Our result (RQ₆) shows that vulnerable dependencies in the dependent projects are fixed within two project releases produced after the vulnerable release, however, such releases take almost 6 months to be published. Hence, the project release cycle may cause lags for updating vulnerable dependencies. Maintainers of the dependent projects are recommended to publish a release as soon as they have applied the dependency fix. While this is ideal, one possible way to achieve that is to evaluate the impact of the dependency fix and consequently release a patch only for critical fixes. Such cases should also be highlighted in the project release in a form of GitHub issues, or pull requests, to help the project users be aware of the critical fixes. The project users can then download a version of the project considering the history from the commit or the pull-request that considers the highlighted fixes, as this feature is supported by the Git version control system.

7 Tool Support

A critical issue of vulnerabilities that affect software packages is the lack of developers awareness (Kula et al. 2018). Developers need help to better 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 et al. 2020a).

To address this problem, we build a tool called **Dependency Health** (DEPHEALTH), which uses the approach described in Section 4 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 give package users insights about the prioritization of fixing vulnerabilities of the package.

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 14 shows a screen-shot of the DepHealth's interface. As shown in the figure, each row represents the relevant vulnerability data of a vulnerable

DepHealth
Find the discovery and fix timespan of vulnerabilities in Python packages

CSV Search:

Package Name	Number of Vulnerability Reports	Severity Level	Avg Time to Discover (in days)	Avg Time to Fix (in days)
tensorflow-cpu	197	C: 16 H: 57 M: 112 L: 12	Critical: 42 High: 50 Medium: 44 Low: 44	Critical: 0 High: 0 Medium: 0 Low: 0
tensorflow-gpu	190	C: 15 H: 55 M: 109 L: 11	Critical: 28 High: 44 Medium: 43 Low: 46	Critical: 0 High: 0 Medium: 0 Low: 0
tensorflow	187	C: 18 H: 52 M: 106 L: 11	Critical: 42 High: 44 Medium: 45 Low: 35	Critical: 0 High: 0 Medium: 0 Low: 0
django	94	C: 1 H: 16 M: 67 L: 10	Critical: 35 High: 40 Medium: 35 Low: 47	Critical: 0 High: 0 Medium: 0 Low: 0
plone	82	C: 0 H: 10 M: 69 L: 13	Critical: 0 High: 58 Medium: 52 Low: 63	Critical: 0 High: 0 Medium: 0 Low: 0
ansible	52	C: 4 H: 14 M: 28 L: 9	Critical: 65 High: 32 Medium: 44 Low: 44	Critical: 0 High: 0 Medium: 0 Low: 0
salt	49	C: 4 H: 22 M: 17 L: 6	Critical: 77 High: 42 Medium: 27 Low: 14	Critical: 0 High: 0 Medium: 0 Low: 0
pillow	43	C: 1 H: 18 M: 23 L: 1	Critical: 86 High: 51 Medium: 50 Low: 28	Critical: 0 High: 0 Medium: 0 Low: 0
apache-airflow	31	C: 1 H: 13 M: 17 L: 0	Critical: 7 High: 49 Medium: 46 Low: 0	Critical: 0 High: 0 Medium: 0 Low: 0
mojin	24	C: 0 H: 4 M: 16 L: 2	Critical: 0 High: 36 Medium: 37 Low: 15	Critical: 0 High: 0 Medium: 0 Low: 0

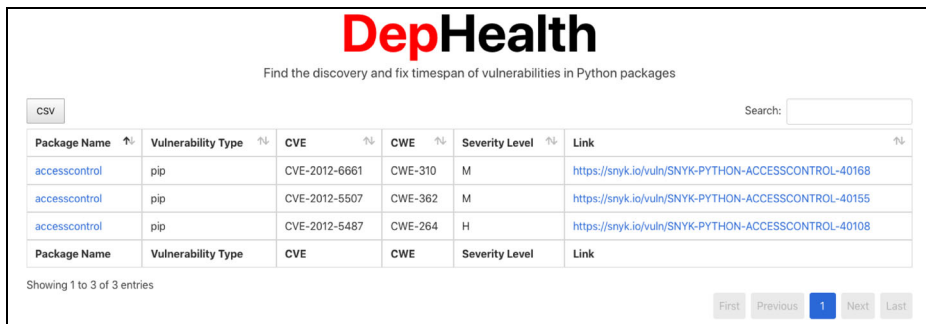
Showing 1 to 10 of 1,156 entries

First Previous 1 2 3 4 5 ... 116 Next Last

Fig. 14 Screen-shot of the DEPHEALTH website showing its main interface (Dephealth 2021). The columns' names that appeared inside the red-colored borders are the main metrics calculated for each vulnerable package

package. Our tool (DepHealth) has the potential to help developers in two major ways as follows:

- Modern applications rely on many third-party packages, and the number of packages adopted is growing over time. Therefore, project developers struggle to check the security health of a particular package. In fact, a recent survey with developers indicated a high demand for high-level metrics to assess the maintainability and security of software packages (Pashchenko et al. 2020b). To help with this, our tool prototype provides developers with the frequency of vulnerabilities that affected packages in the past, i.e., the columns “**Number of Vulnerability Reports**” and “**Severity Level**” presented in the tool website can help developers to grab the risks of depending on such packages and better understand the security health of the package.
- Moreover, the tool can work as a dashboard to provide developers with the average time a vulnerability takes to be discovered and fixed in the package (i.e., “Avg Time to Discover & Avg Time to Fix” in the tool website). Such analytical reports can help developers understand the degree to which a library would be safe to use, i.e., if the time to discover and fix vulnerabilities in the library is short, it is highly likely that the library is mature in terms of security. That said, packages that do not discover and fix vulnerability fast incur a higher risk for projects that use them and should be avoided by



DepHealth

Find the discovery and fix timespan of vulnerabilities in Python packages

CSV Search:

Package Name ↑	Vulnerability Type ↑	CVE ↑	CWE ↑	Severity Level ↑	Link ↑
accesscontrol	pip	CVE-2012-6661	CWE-310	M	https://snyk.io/vuln/SNYK-PYTHON-ACCESSCONTROL-40168
accesscontrol	pip	CVE-2012-5507	CWE-362	M	https://snyk.io/vuln/SNYK-PYTHON-ACCESSCONTROL-40155
accesscontrol	pip	CVE-2012-5487	CWE-264	H	https://snyk.io/vuln/SNYK-PYTHON-ACCESSCONTROL-40108
Package Name	Vulnerability Type	CVE	CWE	Severity Level	Link

Showing 1 to 3 of 3 entries

First Previous 1 Next Last

Fig. 15 A Screen-shot showing some meta-data for vulnerabilities in the *Accesscontrol* package

critical projects. We believe that such information is important for developers to build a valuable picture of the risk of adopted packages, not only using automated tools (e.g., Dependabot) to update each and every package in the project but also high-level metrics to assess the maintainability and security of software packages.

To facilitate using the tool, DEPHEALTH provides the user with an option to search for a specific package to view its data, by using the search box in the interface (the top-right of Fig. 14). Also, the user can download (using the CSV button) a complete version of the data presented in the website. It is also possible to download the data for selective packages, by pressing a command key + 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 report can be shown, e.g., a reference to the report, as shown in Fig. 15.

Moreover, the prototype provides an analysis of the trend of vulnerabilities in each package. For each package, we plot the distribution of published vulnerabilities over time. Figure 16 shows an example plot for vulnerability trends in the *ansible* package. The figure shows that the number of vulnerabilities affecting *ansible* ranges between 1 and 4. The number of vulnerabilities ranges between one to two vulnerabilities before 2021. However, the number has increased to 4 vulnerabilities in 2021 and 2022. We believe such plots can help package users build a better understanding of the security health of the package and how active and mature the project is in terms of discovering and reporting vulnerabilities. In fact, prior work (e.g., Walden 2020) found that a small number of vulnerabilities may indicate that a project does not expend much effort to find vulnerabilities. We incorporated these plots in our prototype tool for packages with more than 10 vulnerabilities. The user can find

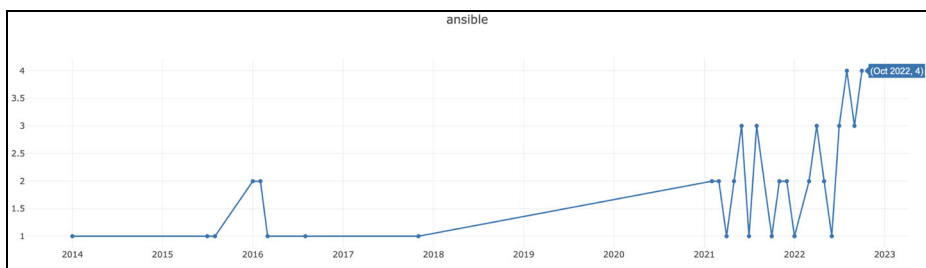


Fig. 16 Distribution of reported vulnerabilities in the *Ansible* package

a plot icon under the column “Number of Vulnerability reports.” Clicking on the icon opens a new window that views the plot of the package.

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. The tool is publicly available through this website (Dephealth 2021). Moreover, to make the prototype tool active and provide the user with up-to-date data, we schedule a cron job that runs monthly to collect updated data from the Snyk vulnerability database and perform the analysis to incorporate it into the tool website. The website currently contains 3,324 vulnerability reports that correspond to 1,156 vulnerable python packages, i.e., 458 packages have been added to the data as new vulnerable packages since we performed our study.

8 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.

8.1 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., 1,396 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 (Snyk 2020c), we expect our results to be representative of high-quality Python packages. Furthermore, our vulnerability dataset contains more than 1,300 reports that cover 16 years of *PyPi* reported vulnerabilities, and many of these reports are related to a popular and most used Python packages (e.g., Django, Flask, Numpy, Requests).

With respect to the cascading impact of vulnerabilities on the dependent projects, we only take into consideration runtime dependencies (i.e. dependencies that are required to install and run packages). While another type of dependencies could be considered (such as development dependencies), we decide to not include them as they tend to have no effect on the production environment. For that same reason, libraries.io considers only runtime dependencies in their dataset.

Another concern is whether the vulnerability that inhabits a dependency actually affect its dependent project. An industrial report by SAFECode (Bisht et al. 2017) recommends package users to evaluate if a vulnerability in the packages is really being used by the dependent. However, checking thousands of packages and dependencies automatically is a hard task, where there is little type information at the source-code level. This was also confirmed by Hejderup et al. (2018), where authors stated that analysing the usage of dependencies is very difficult, as building such a large-scale call-graph is time consuming and prone to false-positives.

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 6.1, *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 (Decan et al. 2018a; Zapata et al. 2018a; Chinthanet et al. 2019)

8.2 Conclusion Validity

Our study uses the Kaplan-Meier test to analyze to estimate the survival functions of several time events. Our results may be biased because the test is purely a significance test and cannot provide an estimate of the size of the difference between the groups. Another limitation of the Kaplan-Meier test method is that it only provides survival probabilities, i.e., it may be needed to adjust the probability for potential confounders that may impact the test results. Although this is a threat, we would like to state that the Kaplan Meier test can still provide an average overview related to the event. Also, the test does not require too many features, i.e., time to the survival analysis event is only required for our evaluation. Furthermore, several similar studies have adopted the test to perform “time to event” analysis (Decan et al. 2018a; Zerouali et al. 2019; Decan and Mens 2019; Constantinou and Mens 2017).

8.3 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.

9 Related Work

In this section, we discuss the related literature by focusing on studies that investigate software ecosystems in general, and studies that approach the link of security-related issues and software ecosystems.

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 and Mesbah 2017; Wittern et al. 2016). Fard and 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. (2016) compared 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.

A few studies conducted a comparison across software ecosystems. Decan et al. (2016, 2019) empirically compared the dependency network 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 study is fundamentally different. Our work can be used as to complement previous work by providing a view on another important quality metric of software ecosystems: security vulnerabilities.

Security Vulnerabilities in Ecosystems The potential fragility of the ecosystems shown in previous studies (e.g., Bogart et al. 2015, 2016) has motivated researchers to examine security vulnerabilities, as vulnerabilities are one of the most problematic aspects of software ecosystems (Thompson 2003). Camilo et al. (2015) empirically studied the correlation between vulnerabilities and bugs (non-security bugs). They did analysis only based on Chromium Project. They found no correlation between bugs and vulnerabilities, and files of high bugs density are not overlapped with files of high vulnerability density. A study by Pham et al. (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.

Studies by Derr et al. (2017) and Massacci et al. (2011) showed that vulnerabilities appeared commonly in unmaintained code and old versions, and could be fixed by just an update to a newer version. Acknowledging that the use of external components, especially outdated ones, can increase risks, e.g., vulnerabilities and comparability issues, Cox et al. (2015) evaluated the so-called “dependency freshness”, to understand the relationship between outdated dependencies and vulnerabilities using an industry benchmark. The authors found that vulnerabilities were four times more likely to occur in outdated systems than in updated systems.

Other studies focused on analysing vulnerabilities in software ecosystems. 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 (2009) analysed vulnerabilities that affect RedHat packages and their dependencies and build a model to predict packages with vulnerabilities. Zimmermann et al. (2019) 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. Zerouali et al. (2019) identified that vulnerabilities that affect *npm* packages are common in official Docker containers. A study by Zapata et al. (2018b) assessed the danger of having vulnerabilities in dependent 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. (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. 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. Prana et al. (2021) studied several project characteristics and their impact on the project being affected by vulnerable packages. They focused on project from three

languages, Java, Python, and Ruby. They found that project activity level, popularity, and developer experience have no correlation with the project being affected by dependency vulnerabilities, which highlights the importance of managing dependencies and their security updates, regardless of the project characteristics.

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. (2020) found that Java packages contained dependencies which lag for a long time and never been updated. Alfadel et al. (2021) examined the use of Dependabot security pull-requests (PRs) for tackling vulnerable packages in 2,904 JavaScript projects, and found that a large proportion (65.42%) of Dependabot security PRs are merged, often in one day, indicating that Dependabot can be of great help to increase awareness of developers to dependency vulnerabilities in their projects. Also, Pashchenko et al. (2020b) 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 is a package. Our study methodology (as suggested in Section 6.2) can be employed to provide developers with such metrics for package selection process.

Ponta et al. (2018, 2020) proposed a code-centric approach to detect and mitigate open source vulnerabilities for Java and Python industry grade applications. Pashchenko et al. (2018, 2020) addressed the over-estimation problem of approaches that report vulnerable dependencies in the Java ecosystem. They introduced the concept of *halted dependencies* to describe the libraries that are no longer maintained, and studied 200 popular Java packages used by SAP in its own software. They found that 20% of the known vulnerable dependencies are not deployed, and hence, have less chance to be exploited. Also, they found that 82% of the vulnerable dependencies deployed in the software could be fixed by simply updating to the patch version. Di Penta et al. (2009) assessed three static analysis tools for detecting source code vulnerabilities in three open-source network systems, accounting for the evolution of code vulnerabilities. The authors did not find much overlap between the tools' results, suggesting no "silver bullet" vulnerability detection tool.

Novelty of our study We conducted our study to examine security vulnerabilities in the *PyPi* ecosystem. We studied several aspects related to vulnerability propagation, their discovery and fix timeliness. By comparing our findings with the ones reported by Zerouali et al. (2022), we identified some particularities of the *PyPi* ecosystem and devise important recommendations to improve the safety of *PyPi*. Moreover, we extend our study by examining the impact of dependency vulnerabilities on Python dependent projects, considering their production releases. Finally, we developed a prototype tool that supports our study analysis. We believe that such a tool can be further improved and be very beneficial to projects maintainers, to fulfill the need for understanding the health of their project dependencies.

10 Conclusion

In this paper, we conduct an empirical study of security vulnerabilities in the *PyPi* ecosystem, evaluating over than 1,396 package vulnerabilities that affect 698 packages. In particular, we explore vulnerabilities propagation, discovery, and fixes. Also, we examine

the impact of vulnerable package on a set of 2,224 Python dependent projects. Finally, we compare our findings with the *npm* ecosystem (Zerouali et al. 2022).

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 a large portion of vulnerabilities (40.86%) were patched only after public disclosure. When analysing the impact on the dependent projects, we find that the majority (55.62%) of the dependent projects depend on vulnerable dependencies. Affected projects have, on median, 83.77% of their releases affected. In most cases, projects are affected by just a single vulnerable dependency. Such project also take a long time (7 months) to update their dependency vulnerabilities to a fixed versions. Finally, our comparison to *npm* vulnerabilities reveals in some aspects a departure from the *npm*'s findings, which can be attributed to ecosystems polices.

Future work should focus on broadening our study to other ecosystems and work on the development of package security tools that help practitioners at selecting and using secure software packages. In addition, we plan to evaluate the usefulness of our prototype tool (i.e., DepHealth) to evaluate the usefulness of the tool for developers.

Data Availability The dataset we curated and analyzed during the current study is available in a public repository (Alfadel et al. 2020).

Declarations

Conflict of Interests The authors have no conflict of interests.

Competing interests The authors declared that they have no conflict of interest/competing interests.

References

- Aalen O, Borgan O, Gjessing H (2008) Survival and event history analysis: a process point of view. Springer Science & Business Media, Berlin
- Abdalkareem R, Noury 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
- Abdalkareem R, Oda V, Mujahid S, Shihab E (2020) On the impact of using trivial packages: an empirical case study on npm and PyPI. *Empir Softw Eng* 25(2):1168–1204
- Alfadel M, Costa DE, Shihab E (2020) Dataset: Empirical analysis of security vulnerabilities in Python packages — zenodo. <https://zenodo.org/record/4158611>. Accessed 29 Oct 2020
- Alfadel M, Costa DE, Shihab E (2021) Empirical analysis of security vulnerabilities in python packages. In: 2021 IEEE international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 446–457
- Alfadel M, Costa DE, Shihab E, Mkhallalati M (2021) On the use of dependabot security pull requests. In: 2021 IEEE/ACM 18th international conference on mining software repositories (MSR). IEEE, pp 254–265
- Allodi L, Massacci F (2014) Comparing vulnerability severity and exploits using case-control studies. *ACM Trans Inf Syst Secur (TISSEC)* 17(1):1–20
- Bewick V, Cheek L, Ball J (2004) Statistics review 12: survival analysis. *Crit Care* 8(5)
- Bisht P, Heim M, Ifland M, Scovetta M, Skinner T (2017) Managing security risks inherent in the use of third-party components. (2017). executive information systems, Inc., White Paper No Eleven

- Bogart C, Kästner C, Herbsleb J (2015) 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
- 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. ACM, pp 109–120
- Borges H, Valente MT (2018) What's in a Github star? understanding repository starring practices in a social coding platform. *J Syst Softw* 146:112–129
- Camilo F, Meneely A, Nagappan M (2015) Do bugs foreshadow vulnerabilities?: a study of the chromium project. In: Proceedings of the 12th working conference on mining software repositories. IEEE Press, pp 269–279
- Chinthanet B, Kula RG, 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 RG, McIntosh S, Ishio T, Ihara A, Matsumoto K (2021) Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empir Softw Eng* 26(3):1–28
- Chowdhury MAR, Abdalkareem R, Shihab E, Adams B (2021) On the untriviality of trivial packages: An empirical study of npm javascript packages. *IEEE Trans Softw Eng*
- Constantinou E, Mens T (2017) An empirical comparison of developer retention in the rubygems and npm software ecosystems. *Innov Syst Softw Eng* 13(2):101–115
- Cox J, Bouwers E, van Eekelen M, Visser J (2015) Measuring dependency freshness in software systems. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering (ICSE), vol 2. IEEE, pp 109–118
- cwe.mitre (2020) Cwe - cwe-416: Use after free (3.3). <https://cwe.mitre.org/data/definitions/416.html>. Accessed 10 Oct 2020
- Dabic O, Aghajani E, Bavota G (2021) Sampling projects in Github for MSR studies. In: 2021 IEEE/ACM 18th international conference on mining software repositories (MSR). IEEE, pp 560–564
- Decan A, Mens T (2019) What do package dependencies tell us about semantic versioning? *IEEE Trans Softw Eng*
- Decan A, Mens T, Claes M (2016) On the topology of package dependency networks: A comparison of three programming language ecosystems. In: Proceedings of the 10th european conference on software architecture workshops, pp 1–4
- 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). IEEE, pp 2–12
- Decan A, Mens T, Constantinou E (2018a) On the impact of security vulnerabilities in the npm package dependency network. In: 2018 IEEE/ACM 15th international conference on mining software repositories (MSR). IEEE, pp 181–191
- Decan A, Mens T, Constantinou E (2018b) On the evolution of technical lag in the npm package dependency network. In: 2018 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 404–414
- Decan A, Mens T, Grosjean P (2019) An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empir Softw Eng* 24(1):381–416
- Dependabot (2020) <https://github.com/dependabot>. Accessed 28 Oct 2020
- Dephealth (2021) Home. <http://104.237.154.205:8443/?fbclid=IwAR3qdZPNXISqK7VvkPNXYQaEhtdxKR8nBEbmGJI7Z-nHw9f6.oSNAjLc.dI>. Accessed 2021
- Derr E, Bugiel S, Fahl S, Acar Y, Backes M (2017) Keep me updated: An empirical study of third-party library updatability on android. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security. ACM, pp 2187–2200
- Di Penta M, Cerulo L, Aversano L (2009) The life and death of statically detected vulnerabilities: An empirical study. *Inf Softw Technol* 51(10):1469–1484
- Durumeric Z, Li F, Kasten J, Amann J, Beekman J, Payer M, Weaver N, Adrian D, Paxson V, Bailey M et al (2014) The matter of heartbleed. In: Proceedings of the 2014 conference on internet measurement conference, pp 475–488
- Fard AM, Mesbah A (2017) Javascript: The (un) covered parts. In: 2017 IEEE international conference on software testing, verification and validation (ICST). IEEE, pp 230–240
- Github (2022) Transparency report: January to June — the Github blog. <https://github.blog/2022-08-16-2022-transparency-report-january-to-june/>. Accessed 31 Oct 2022
- Godefroid P, Levin MY, Molnar D (2012) SAGE: whitebox fuzzing for security testing. *Commun ACM* 55(3):40–44

- Google (2020) Android – google play protect. https://www.android.com/intl/en_ca/play-protect/. Accessed 27 Oct 2020
- Hejderup J (2015) In dependencies we trust: How vulnerable are dependencies in software modules?
- Hejderup J, van Deursen A, Gousios G (2018) Software ecosystem call graph for dependency management. In: 2018 IEEE/ACM 40th international conference on software engineering: new ideas and emerging technologies results (ICSE-NIER). IEEE, pp 101–104
- ISC (2020) Internet systems consortium. <https://www.isc.org/#>. Accessed 10 Oct 2020
- 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. IEEE, pp 453–458
- Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining Github. In: Proceedings of the 11th working conference on mining software repositories, MSR '14. ACM, pp 92–101
- Kaplan EL, Meier P (1958) Nonparametric estimation from incomplete observations. *J Am Stat Assoc* 53(282):457–481
- Kula RG, German DM, Ouni A, Ishio T, Inoue K (2018) Do developers update their library dependencies? *Empir Softw Eng* 23(1):384–417
- Larios-Vargas E, Aniche M, Treude C, Bruntink M, Gousios G (2020) Selecting third-party libraries: The practitioners' perspective. [arXiv:2005.12574](https://arxiv.org/abs/2005.12574)
- 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. ACM, pp 2201–2215
- Libraries.io (2021) Libraries - the open source discovery service. Accessed 10 Jan 2021
- Lodash (2020) lodash - npm. <https://www.npmjs.com/package/lodash>. Accessed 10 Oct 2020
- 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
- MITRE (2020) Cwe. <https://cwe.mitre.org/about/index.html>. Accessed 10 Oct 2020
- Massacci F, Neuhaus S, Nguyen VH (2011) After-life vulnerabilities: a study on firefox evolution, its vulnerabilities, and fixes. In: International symposium on engineering secure software and systems. Springer, pp 195–208
- Metha N (2022) Heartbleed and shellshock: The new norm in vulnerabilities. <https://securityintelligence.com/heartbleed-and-shellshock-the-new-norm-in-vulnerabilities/>. Accessed 31 Oct 2022
- Mezzetti G, Möller A, Torp MT (2018) Type regression testing to detect breaking changes in node.js libraries. In: 32nd european conference on object-oriented programming (ECOOP 2018), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik
- NPM (2020a) Reporting a vulnerability in an npm package — npm documentation. <https://docs.npmjs.com/reporting-a-vulnerability-in-an-npm-package>. Accessed 10 Oct 2020
- NPM (2020b) Auditing package dependencies for security vulnerabilities — npm documentation. <https://docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities>. Accessed 10 Oct 2020
- Nesbitt A, Nickolls B (2018) Libraries.io open source repository and dependency metadata. v1.2.0. <https://doi.org/10.5281/zenodo.808273>. Accessed 10 Oct 2020
- Neuhaus S, Zimmermann T (2009) The beauty and the beast: Vulnerabilities in red hat's packages. In: USENIX annual technical conference
- OWASP (2019) Owasp. https://www.owasp.org/index.php/Main_Page. Accessed 10 Oct 2020
- Pashchenko I, Plate H, Ponta SE, 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 SE, Sabetta A, Massacci F (2020) Vuln4Real: A methodology for counting actually vulnerable dependencies. *IEEE Trans Softw Eng*
- 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
- Pashchenko I, Vu D-L, Massacci F (2020) A qualitative study of dependency management and its security implications. *Proc of CCS'20*
- Pham NH, Nguyen TT, Nguyen HA, Nguyen TN (2010) Detection of recurring software vulnerabilities. In: Proceedings of the IEEE/ACM international conference on Automated software engineering. ACM, pp 447–456
- Pillow (2020) Pillow · pypi. <https://pypi.org/project/Pillow/>. Accessed 10 Oct 2020

- Ponta SE, 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). IEEE, pp 449–460
- Ponta SE, Plate H, Sabetta A (2020) Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empir Softw Eng* 25(5):3175–3215
- Prana GAA, Sharma A, Shar LK, Foo D, Santosa AE, Sharma A, Lo D (2021) Out of sight, out of mind? how vulnerable dependencies affect open-source projects. *Empir Softw Eng* 26(4):1–34
- PyPi (2018) Security · pypi. <https://pypi.org/security/>. Accessed 10 Oct 2020
- Python (2020) Issue 27863: multiple issues in _elementtree module - python tracker. <https://bugs.python.org/issue27863>. Accessed 10 Oct 2020
- 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 (IWESPE). IEEE, pp 25–30
- Sabotkke C, Suciú O, Dumitras 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
- Semver (2020) semver · pypi. <https://pypi.org/project/semver/>. Accessed 10 Oct 2020
- Snyk (2020a) Vulnerability db — Snyk. <https://snyk.io/vuln>. Accessed 10 Oct 2020
- Snyk (2020b) Scoring security vulnerabilities 101: Introducing cvss for cves — snyk. <https://snyk.io/blog/scoring-security-vulnerabilities-101-introducing-cvss-for-cve/>. Accessed 10 Oct 2020
- Snyk (2020c) How Snyk finds out about new vulnerabilities – knowledge center — snyk. <https://support.snyk.io/hc/en-us/articles/360003923877-How-Snyk-finds-out-about-new-vulnerabilities>. Accessed 24 Oct 2020
- Snyk.io (2017) The state of open-source security. <https://snyk.io/>
- StackOverflow (2020) Stack overflow developer survey. <https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages-all-respondents>. Accessed 10 Jan 2021
- 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
- Thomé J, Shar LK, Bianculli D, Briand L (2018) Security slicing for auditing common injection vulnerabilities. *J Syst Softw* 137:766–783
- Thompson HH (2003) Why security testing is hard. *IEEE Secur Priv* 1(4):83–86
- Vu D-L, Pashchenko I, Massacci F, Plate H, Sabetta A (2020) Typosquatting and combosquatting attacks on the python ecosystem. In: 2020 IEEE european symposium on security and privacy workshops (EuroS&PW). IEEE, pp 509–514
- Vu D-L, Pashchenko I, Massacci F, Plate H, Sabetta A (2020) Poster: Towards using source code repositories to identify software supply chain attacks. In: CCS '20
- 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, Wu Y, 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). IEEE, pp 35–45
- Williams J, Dabirsiaghi A (2012) The unfortunate reality of insecure libraries. *Asp. Secur. Inc*, 1–26
- Wittern E, Suter P, Rajagopalan S (2016) A look at the dynamics of the javascript package ecosystem. In: 2016 IEEE/ACM 13th working conference on mining software repositories (MSR). IEEE, pp 351–361
- Zapata RE, Kula RG, 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). IEEE, pp 559–563
- Zapata RE, Kula RG, 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). IEEE, pp 559–563
- Zerouali A, Cosentino V, Mens T, Robles G, Gonzalez-Barahona JM (2019) On the impact of outdated and vulnerable javascript packages in docker images. In: 2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 619–623
- Zerouali A, Mens T, Decan A, De Roover C (2022) On the impact of security vulnerabilities in the npm and rubygems dependency networks. *Empir Softw Eng* 27(5):1–45
- Zerouali A, Mens T, Robles G, Gonzalez-Barahona JM (2019) On the relation between outdated docker containers, severity vulnerabilities, and bugs. In: 2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 491–501
- Zimmermann M, Staicu C-A, Tenny C, Pradel M (2019) Small world with high risks: A study of security threats in the npm ecosystem. In: 28th USENIX security symposium (USENIX security 19), pp 995–1010

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Affiliations

Mahmoud Alfadel¹  · Diego Elias Costa² · Emad Shihab¹

Diego Elias Costa
costa.diego@uqam.ca

Emad Shihab
emad.shihab@concordia.ca

¹ Department of Computer Science and Software Engineering, Data-Driven Analysis of Software (DAS) Lab, Concordia University, Montreal, QC Canada

² LATECE Lab, Université du Québec à Montréal, Montreal, Canada