On the impact of security vulnerabilities in the npm package dependency network

Alexandre Decan
Tom Mens
Eleni Constantinou
Software Engineering Lab, University of Mons
Belgium
firstname.lastname@umons.ac.be

ABSTRACT

Security vulnerabilities are among the most pressing problems in open source software package libraries. It may take a long time to discover and fix vulnerabilities in packages. In addition, vulnerabilities may propagate to dependent packages, making them vulnerable too. This paper presents an empirical study of nearly 400 security reports over a 6-year period in the npm dependency network containing over 610k JavaScript packages. Taking into account the severity of vulnerabilities, we analyse how and when these vulnerabilities are discovered and fixed, and to which extent they affect other packages in the packaging ecosystem in presence of dependency constraints. We report our findings and provide guidelines for package maintainers and tool developers to improve the process of dealing with security issues.

KEYWORDS

software repository mining, software ecosystem, dependency network, security vulnerability, semantic versioning

ACM Reference Format:
Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *MSR '18: 15th International Conference on Mining Software Repositories, May 28–29, 2018, Gothenburg, Sweden* . ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196398.3196401

1 INTRODUCTION

Security vulnerabilities in software are among the most pressing and important problems in our software-intensive society, given the increasing reliance on software libraries and the potentially harmful impact security vulnerabilities in such libraries may have [23]. According to a white paper by Contrast Security, a company providing products for software security, 80% of the code in today's applications comes from libraries and frameworks, and about one fourth of library downloads have known vulnerabilities [24]. A recent report by Snyk, one of the leading companies involved in analysing software vulnerabilities in Node.js and Ruby packages, summarises the current state of open source security [22]. Based on an analysis of 430,000 websites, they report that no less than 77% of them run at least one front-end library with a known security vulnerability. They also observe an increase in the number of newly published critical open source vulnerabilities.

A well-known example is the so-called Heartbleed Bug [12]. It represented a serious security leak in the OpenSSL cryptography library, allowing anyone on the Internet to read the memory of the software systems relying on the vulnerable versions of the OpenSSL software. The vulnerability was introduced in 2012 and remained lingering until it was discovered and traced in April 2014. Upon its discovery, half a million of servers certified by trusted authorities were believed to be affected by the security vulnerability, connected to a simple programming mistake.

The open source development community acknowledges the importance of security vulnerabilities, and is taking active measures to deal with them. For example, since October 2017, GitHub started to monitor the dependencies of hosted Ruby and JavaScript projects, and to trigger alerts when vulnerabilities are detected.[1] They also put into place a bounty hunting program to detect vulnerabilities more rapidly.[2] It is however too early to assess the positive impact of such a program.

Little is known about, and even less automated support is available for, assessing the ecosystem-wide impact of security vulnerabilities in package dependency networks. Vulnerabilities, as well as their fixes, may spread over the network through dependencies between software packages. Analysing the propagation of security vulnerabilities and their fixes is technically challenging and computationally intensive, due to the intricate interactions of the mechanisms of semantic versioning and dependency constraints.

This paper provides an empirical study of the propagation of security vulnerabilities and their fixes in the package release history of the npm distribution of Node.js packages. A replication package of our analysis is available on https://doi.org/10.5281/zenodo.1193577

Our study focuses on the following research questions:

- RQ_0 How many packages are known to be affected by vulnerabilities?
- RQ_1 How long do packages remain vulnerable?
- RQ_2 When are vulnerabilities discovered?
- RQ_3 When are vulnerabilities fixed?

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196401

[1] https://help.github.com/articles/about-security-alerts-for-vulnerable-dependencies/
[2] https://bounty.github.com

- RQ_4 When are vulnerabilities fixed in dependent packages?

By providing answers to these questions, we aim to get a better understanding of the extent and propagation of security vulnerabilities and their fixes ecosystem-wide. These insights may help in prioritising the most problematic software packages and vulnerabilities, as well as in proposing guidelines for dependent packages to reduce the risk of suffering from vulnerabilities.

The remainder of this paper is structured as follows. Section 2 motivates the choice of npm as a case study, presents the research methodology, and details the data extraction process. Section 3 to Section 7 answer the research questions and Section 8 puts these results into perspective. Section 9 presents the threats to validity of our research and Section 10 relates our research to other works. Section 11 discusses the future work and Section 12 concludes.

2 METHODOLOGY

2.1 Selected package manager

In October 2017, GitHub reported that JavaScript is by far the most popular language with 2.3M hosted open source projects.[3] This is more than twice the number of projects of the second-most popular language, Python. In addition, the npm package distribution of JavaScript packages was observed to have a higher distribution of package dependencies than other package distributions [9, 10]. This increases the potential impact of vulnerabilities. For these reasons we selected npm for our empirical study.

The metadata of each npm package (such as the name, version, and list of dependencies) are stored in a .json manifest file. Dependencies are used to specify other packages that are explicitly required by each package. The range of allowed versions can be restricted using *dependency constraints*.

npm package releases and dependency constraints rely on the mechanism of *semantic versioning*. A simple set of rules and requirements dictate how version numbers should be assigned to package releases and incremented based on the three-number version format Major.Minor.Patch. Package updates corresponding to bug fixes that do not affect the API should only increment the Patch version number, backward compatible updates should increment the Minor version number, and backward incompatible updates have to increment the Major version number.

Dependency constraints allow package maintainers to explicitly select the desirable or allowed versions of a dependency, and to explicitly exclude the undesirable ones, e.g., those that can contain backward incompatible changes. Dependency constraints are typically used to specify a minimal (e.g., >= 1.2.3) or a maximal version (e.g., < 1.3.0) of a dependency. Combinations of constraints can also be expressed using specific notations. The npm package manager relies on the semver tool[4] to identify which are the version numbers that satisfy a dependency constraint.

2.2 Identifying affected releases

In order to identify which releases of a package are affected by a vulnerability, we need to consider the vulnerability constraint expressed in the corresponding security report. Vulnerability constraints capture all the releases that are affected by the vulnerability. For example, the security report of Figure 1 defines the vulnerability constraint <1.1.4 and >=0.4.0 on package rendr and can be used to determine when a vulnerable package starts and ends being affected by the vulnerability.

Table 1: Releases for package B and its dependent package D. A cell with coloured background indicates that the corresponding release of B satisfies the vulnerability constraint >= 1.1.0 and < 3.0.1.

time	release of B	release of D	dependency constraint	latest installable release of B
T_0	1.0.0	none		
T_1		0.1.0	1.x.x	1.0.0
T_2	1.1.0			1.1.0
T_3	2.0.0			1.1.0
T_4		0.2.0	2.x.x	2.0.0
T_5	3.0.0			2.0.0
T_6		0.3.0	3.x.x	3.0.0
T_7	3.0.1			3.0.1

Let us illustrate this by means of an example, shown in Table 1. For readability purposes, we use the notation P@V to denote version V of package P. The second column in the table shows the different releases of a vulnerable package B corresponding to its release date represented by a timestamp T_i in the first column. Assume that a security report for B has vulnerability constraint >= 1.1.0 and < 3.0.1. Based on this constraint, we identify releases B@1.1.0, B@2.0.0 and B@3.0.0 as being affected by the vulnerability. According to the dates of these releases, B is considered as vulnerable from T_2 (date of the first affected release B@1.1.0) until just before T_7 (date of the first fixed release B@3.0.1).

As a vulnerable package can affect packages that have a dependency on it, we also compute the affected releases of those dependent packages. The identification of such affected releases is a bit more involved, since we need to take into account the dependency constraint specified in the manifest of the dependent releases, as well as the fact that it is the most recent release satisfying a dependency constraint that is by default installed by the npm package manager. This is, the release (of a required package) that is selected for installation may change depending on when the installation is performed, even if the dependency constraint remains the same.

Reconsidering the example shown in Table 1, assume that all releases of package D depend on B. The third and fourth columns show the different releases of D and their respective dependency constraints on B. The fifth column shows the release of B that will be automatically installed by the package manager when D is installed (i.e., the latest available release of B satisfying the dependency constraint).

Assume that D@0.1.0 has a dependency constraint 1.x.x on B. This constraint is satisfied by both B@1.0.0 and B@1.1.0. Depending on the moment D@0.1.0 is installed, either B@1.0.0 or B@1.1.0 will be selected. For instance, B@1.1.0 will be selected if D@0.1.0 is installed after T_2 because B@1.1.0 is the latest release of B satisfying the constraint at that time. On the other hand, B@1.0.0 will be

[3] https://octoverse.github.com/
[4] https://docs.npmjs.com/misc/semver

selected if the installation is performed before T_2 because B@1.1.0 was not yet available at that time. Notice that even if B@2.0.0 is the latest available release of B at T_3, it will never be selected during the installation of D@0.1.0 since it does not satisfy the dependency constraint 1.x.x.

In the example of Table 1, the cells with coloured background indicate the respective releases of B that are affected by a security vulnerability. Given that B@1.1.0 is affected by the vulnerability, D@0.1.0 is also affected from T_2 onward. Similarly, D@0.2.0 is affected by the vulnerability because it relies on the affected release B@2.0.0 as of T_4. Finally, D@0.3.0 either depends on B@3.0.0 (from T_6 until T_7) or on B@3.0.1 (starting from T_7). Given that B@3.0.0 is affected by the vulnerability while B@3.0.1 is not, D@0.3.0 is only affected from T_6 until T_7. To sum up, package D is identified as being vulnerable due to its dependency to B from T_2 until T_7.

2.3 Vulnerability Dataset

Snyk.io provides a continuous monitoring service aiming to help developers and organizations identify vulnerable packages on which they depend. It contains a detailed list of security reports for different package managers. For npm it includes information about the vulnerability, the name of the affected package, the vulnerability constraint specifying which releases of the package are concerned by the security report, the discovery date indicating when the vulnerability was discovered, and the date of *public* exposure of the vulnerability which is by definition greater than or equal to the discovery date. The large majority of vulnerabilities in the dataset have a publication date that is strictly later than the discovery date.

To each vulnerability a *severity* label (*low*, *medium* or *high*) is assigned. This severity is assigned manually based on the impact of the vulnerability and how easy it is to exploit it[5] and seems to be determined primarily by the vulnerability's CVSS score[6]. An example of such a report is provided in Figure 1.

Title:	Cross-site Scripting (XSS)
Severity:	Medium
Affected package:	rendr
Vulnerability constraint:	<1.1.4 and >=0.4.0
Discovery date:	10 March, 2016
Publication date:	8 May, 2017

Figure 1: Example of a security report for package rendr.

To identify the security vulnerabilities that affect npm packages, we manually gathered from Snyk.io all 700 security reports that were published before 2017-11-09. For each package identified in a security report, we retrieved the list of its releases from the open source discovery service libraries.io [19]. Their dataset, available under a CC-BY-SA 4.0 licence, contains metadata from the manifest of each package, based on the list of packages provided by the official registry of the npm package manager.

Based on this list of releases, we identified which ones were affected by the vulnerability following the process explained in Section 2.2. We filtered out those vulnerabilities for which the affected packages no longer exist in npm or for which none of the releases satisfy the vulnerability constraint (i.e., they have been deleted from npm). We also ignored vulnerability reports indicating a universal vulnerability constraint (i.e., "*") because such a constraint prevents us from identifying which releases are not (yet or anymore) affected by a vulnerability; either because there is no such "fixed" release, or because the security report was not updated when the vulnerability was fixed. We also removed vulnerabilities of type "Malicious Package". They correspond to typosquatting packages, i.e., packages that are introduced with names that are deliberately close to popular packages in order to trap inattentive users to mistakenly install a wrong and harmful package. We considered such cases as irrelevant to our study since they do not introduce vulnerabilities in existing packages. Because of the applied filters, our reported results should be regarded as a lower bound approximation of the impact of security vulnderabilities in npm. The filtered dataset contains 399 vulnerabilities affecting 269 distinct packages. These packages account for 14,931 distinct releases of which 6,752 are affected by a vulnerability.

As a vulnerable package can affect packages that make use of it, for each package affected by a vulnerability, we considered the packages that have direct dependencies towards the vulnerable ones. To do so, we analyzed the direct dependencies of all packages that were available on 2017-11-02 on libraries.io. These 610K packages account for more than 4M releases and for more than 20M runtime dependencies. Among these packages, we identified 133,602 packages that directly depend on a vulnerable package and 52% of these packages, i.e., 72,470 packages, have at least one release that relies on an affected release of a vulnerable package. Table 2 summarizes the descriptive statistics of the dataset.

Table 2: Descriptive summary of the npm dataset

610,097	npm packages
4,202,099	releases of npm packages
20,240,402	runtime dependencies
399	security vulnerability reports
269	packages affected by the vulnerability
14,931	releases of such vulnerable packages
6,752	releases affected by the vulnerability
133,602	packages depending on a vulnerable package
72,470	dependent packages affected by the vulnerability

3 RQ_0: HOW MANY PACKAGES ARE KNOWN TO BE AFFECTED BY VULNERABILITIES?

RQ_0 aims to provide some initial understanding about how many npm packages suffer from security vulnerabilities, and how this number increases over time.

Figure 2 shows the number of discovered vulnerabilities in the considered dataset. Straight and dotted lines correspond respectively to the number of discovered vulnerabilities and the number of corresponding packages. The results of Figure 2 suggest an increasing number of new vulnerabilities and of vulnerable packages over time. We observe that the large majority of vulnerabilities have a medium or high severity (respectively 235 and 139 out of 399), while there is a much lower number of low severity vulnerabilities

[5] See https://support.snyk.io/frequently-asked-questions/finding-vulnerabilities/how-do-you-determine-the-severity-of-a-vulnerability
[6] See https://nvd.nist.gov/vuln-metrics

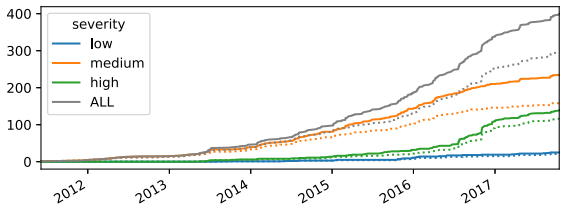

Figure 2: Evolution of the number of discovered vulnerabilities (straight lines) and corresponding distinct packages (dotted lines) per severity.

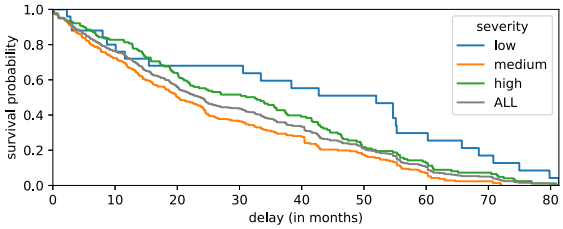

Figure 3: Survival probability for event "vulnerability is fixed" w.r.t. the date of first affected release.

(25 out of 399). This might be due to the fact that vulnerabilities that have a very limited impact in terms of security are considered as not worth having a vulnerability report by maintainers.

While the dataset contains a very low number of packages affected by vulnerabilities (269 out of a total of more than 610k), this does not mean that other packages are not vulnerable. It is very likely that many npm packages are not being monitored yet, and that many vulnerabilities that are already discovered have not been reported yet and therefore, are not included in our dataset. This can explain the change in trend that can be observed in 2017 as, on average, it takes nearly one year for a vulnerability to be published after it is discovered.

> ***Findings.*** The number of new vulnerabilities and affected packages is growing over time. Most of the reported vulnerabilities are of medium or high severity.

4 RQ_1: HOW LONG DO PACKAGES REMAIN VULNERABLE?

While RQ_0 revealed that the number of vulnerabilities is growing over time, RQ_1 focuses on how long packages are being affected by vulnerabilities. Such information can be quite relevant since the longer a package remains affected, the longer it will remain vulnerable to malicious users. For example, in the case of the OpenSSL cryptography library, the code corresponding to the Heartbleed security vulnerability[7] was *introduced* in December 2011, *released* in OpenSSL@1.0.1 in March 2012, *discovered* in March 2014 and *fixed* in April 2014, leaving OpenSSL open to attackers for more than two years [12].

Our empirical analysis of RQ_1 relies on the statistical technique of survival analysis (a.k.a. event history analysis) [1]. The technique 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 [14].

[7] https://nvd.nist.gov/vuln/detail/CVE-2014-0160

Figure 3 shows Kaplan-Meier survival curves for the event "vulnerability is fixed" w.r.t. the date of the first affected release. Regardless of the severity of the vulnerability, it takes a surprisingly long time before the vulnerability is fixed. One can observe that high severity vulnerabilities take longer to fix than medium severity ones, probably because it is more difficult to fix them. Low severity vulnerabilities take even longer to fix, perhaps because developers consider them as low priority. For example, it takes about 52 months before 50% of all low severity vulnerabilities get fixed, 20 months for 50% of all medium severity vulnerabilities, and 32 months for 50% of all high severity vulnerabilities. The admittedly long time for fixing vulnerabilities confirms the need for more and better support to find and deal with vulnerabilities, as will be discussed in Section 8.

We carried out log-rank tests to compare whether statistically significant differences could be found between the survival curves of time-to-fix by severity. The differences were statistically confirmed at $\alpha = 0.95$, i.e., the null hypotheses H_0 assuming that the survival curves are the same were rejected with p-values < 0.05. This confirms that there is a statistically significant difference in the time required to fix a vulnerability with respect to its severity.

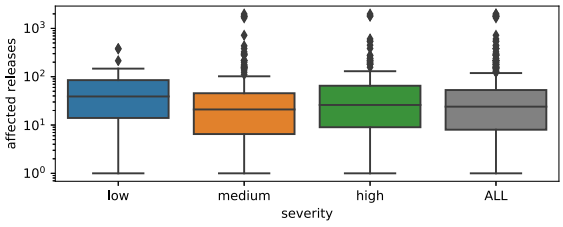

Figure 4: Distribution of the number of affected releases of vulnerable packages by severity.

Figure 4 illustrates how many releases of a vulnerable package are affected by each vulnerability, among all releases of the package that were available when the vulnerability was discovered.

One can observe that the number of affected releases per vulnerable package is quite high (median is 39 for low, 21 for medium and 26 for high severity). Even though these median values differ, Mann-Whitney U tests did not reveal any statistically significant difference. The affected releases represent a large proportion of all the releases that were available for the vulnerable packages at discovery time. Regardless of the severity, 75% of all vulnerable

packages have more than 90% of their releases affected (94% for low, 80% for medium and 93% for high severity vulnerabilities).

These observations indicate that vulnerabilities are not limited to only a few releases, but generally affect the vast majority of releases available at discovery time. Therefore, it is generally not feasible for users of a vulnerable package to avoid the vulnerability by downgrading to a previous release. Hence, users have to wait for the availability of a new release fixing the vulnerability.

> ***Findings.*** It takes a long time to fix vulnerabilities regardless of their severity. The required time to fix vulnerabilities varies depending on the severity. Three out of four vulnerable packages have more than 90% of their releases affected by the vulnerability at discovery time.

5 RQ_2: WHEN ARE VULNERABILITIES DISCOVERED?

RQ_1 revealed that it can take a long time before a vulnerability gets fixed since its introduction. RQ_2 aims to study the main reason why it takes such a long time. This could either be because it takes time to discover the vulnerability lurking in the package, or because it takes a long time to fix the vulnerability once it has been discovered. Reconsidering the example of the Heartbleed vulnerability, it took only a few days to fix the problem, while it took over 2 years to discover it.

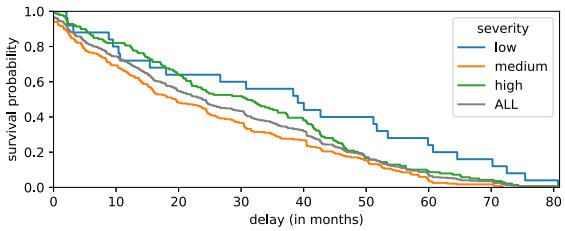

Figure 5: Survival probability for event "vulnerability is discovered" w.r.t. the date of first affected release.

Figure 5 presents the Kaplan-Meier survival curves for event "vulnerability is discovered" w.r.t. the date of first affected release. Although vulnerabilities should be discovered early, this does not seem to hold true in practice.

It takes more than 24 months to discover 50% of all vulnerabilities. The survival curves of Figure 5 also reveal that it takes more time to discover low severity vulnerabilities than medium or high severity vulnerabilities. For instance, it takes 39, 20 and 31 months to discover 50% of all vulnerabilities respectively for low, medium and high severity. We used log-rank tests to compare the time to discover vulnerabilities depending on their severity, and found a statistically significant difference ($p < 0.05$) when comparing low with medium severity, and medium with high severity vulnerabilities.

Given the long time required to discover vulnerabilities, we expect most of them to be found in old packages. Figure 6 shows the distribution of package age at discovery time by vulnerability severity, and confirms that most vulnerabilities, regardless of their severity, are discovered in old packages. For instance, 75%, 50% and

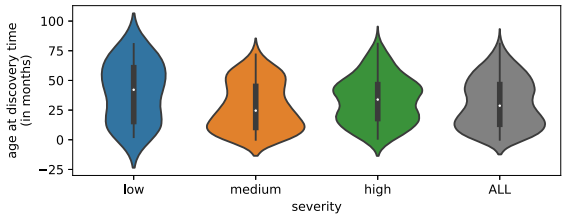

Figure 6: Violin plots of package age at discovery time by vulnerability severity.

25% of all vulnerabilities are discovered in packages respectively older than 13, 28 and 46 months. One possible explanation is that younger packages have not yet had time to reach a wide audience and to receive as much security-related attention as older packages.

Based on our observations, vulnerabilities of highest severity are observed more often in older packages. To statistically verify this, we compared the distributions between the different severity categories using a one-sided non-parametric Mann-Whitney U test. We found a statistically significant difference between medium and high severity at $p < 0.05$. However, we only found a small effect size for this difference using Cliff's delta, indicating that the frequency of older packages with high severity vulnerabilities is not greatly different from the one of packages with medium severity vulnerabilities.

> ***Findings.*** It takes a long time to discover vulnerabilities, especially for those with low severity. Most vulnerabilities are found in packages older than 28 months. Vulnerabilities of high severity are observed more often in older packages.

6 RQ_3: WHEN ARE VULNERABILITIES FIXED?

Not only one can expect a vulnerable package to be fixed, but also to quickly release the fix in order to limit the exposure and exploitation time of the vulnerability. This is particularly important if the vulnerability has been made public. RQ_3 investigates when vulnerabilities are fixed with respect to the discovery and publication dates.

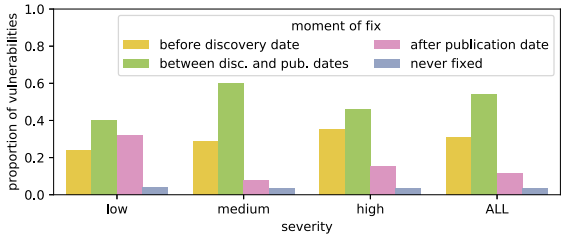

Figure 7: Proportion of vulnerabilities by severity according to the moment of fixing the vulnerability.

Figure 7 shows the proportion of vulnerabilities, by severity, according to their moment of fix: before the vulnerability has been discovered ("before discovery date" in Figure 7), between discovery and publication date, after the vulnerability has been made public

("after publication date"), or "never fixed". We observe that the vast majority of vulnerabilities are eventually fixed, regardless of their severity. Indeed, only 3.5% of the vulnerabilities from our dataset have not (yet) been fixed (resp. 4%, 3.4% and 3.6% for low, medium and high severity).

Surprisingly, 30.9% of all vulnerabilities (resp. 24%, 28.9% and 35.3% for low, medium and high severity) were already fixed at discovery time. One possible explanation is that the higher the severity of a vulnerability is, the less likely the maintainers of an affected package are willing to disclose the vulnerability and to report its discovery while working on a fix. Finally, of the 65.7% remaining vulnerabilities (i.e., those that were fixed after the discovery date), a large majority (82% of them) are fixed before being publicly announced. About 12% of all vulnerabilities are fixed only after their official publication date (resp. 32.0%, 7.7% and 15.1% for low, medium and high severity). This makes them particularly vulnerable for being exploited by malevolent users. These results indicate that maintainers seek to fix vulnerabilities before their publication, strengthening our hypothesis that they want to minimise the chances of their exploitation.

Let us focus now on only those vulnerabilities that get fixed after their discovery, in order to understand how long it takes to fix discovered vulnerabilities. To achieve this, we ignored all vulnerabilities that were fixed *before* the reported discovery date (31% of all vulnerabilities, as shown in Figure 7). For the remaining vulnerabilities, Figure 8 presents the survival curves of the probability for event "vulnerability is fixed" w.r.t. their discovery date.

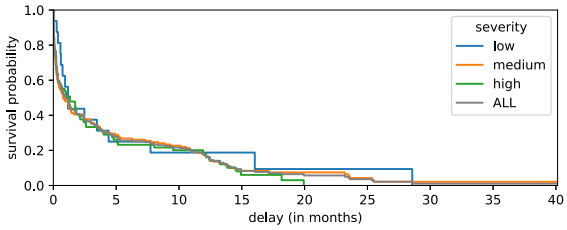

Figure 8: Survival probability for event "vulnerability is fixed" w.r.t. vulnerability discovery time.

We observe that most vulnerabilities, regardless of their severity, are fixed within a few months after their discovery. The probability that a vulnerability is fixed within the month following its discovery is 50%, while it is 74% after only 6 months. There is, however, a non-negligible proportion of vulnerabilities that take a long time to be fixed after their discovery. For instance, the probability that a discovered vulnerability is not fixed after 12 months is still 17.4%.

> ***Findings.*** Most vulnerabilities are fixed after the reported discovery date but before they become public. Most of the vulnerabilities are quickly fixed after their discovery, but there is still a non-negligible proportion of vulnerabilities that take a long time to be fixed.

7 RQ_4: WHEN ARE VULNERABILITIES FIXED IN DEPENDENT PACKAGES?

In the context of package dependency networks such as npm, packages cannot be considered in isolation. As highlighted in [2, 10, 13], npm packages can have a high number of dependents, and these dependencies may cause vulnerabilities to propagate to dependent packages. This phenomenon can be clearly observed in our setting. While our dataset contains 399 security vulnerabilities for 269 distinct npm packages, we identified 133,602 packages that directly depended on these potentially vulnerable packages, and more than half of them (72,470 i.e., 54%) have at least one release that relies on an affected release of a vulnerable package.[8] These 72,470 vulnerable dependents account for 920,661 releases, of which 411,169 are affected by a vulnerability because of one of their dependencies. This advocates the need for better tools to monitor dependencies, combined with support for indicating vulnerable releases.

In the light of the high impact of vulnerabilities in required packages on their dependents, it is useful to study how quickly a dependent package stops being affected by a vulnerability in a required package. It is important to stress that, when a vulnerable package fixes its vulnerability by introducing a new release, dependent packages are not necessarily fixed as well, since dependency constraints may cause these packages to continue to depend on vulnerable releases even if fixed releases are already available.

Dependent packages can explicitly fix vulnerabilities in three different ways: (1) by rolling back to an earlier version of the dependency that is not affected by the vulnerability; (2) by updating to a more recent version of the dependency that has fixed the vulnerability; or (3) by removing the dependency to the vulnerable package.

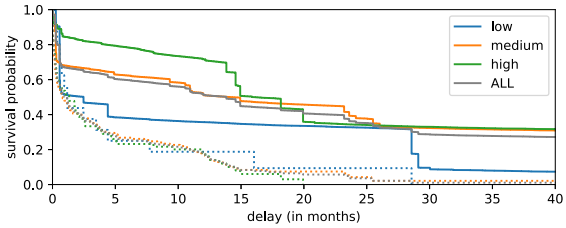

Figure 9: Survival probability for event "package is fixed" w.r.t. vulnerability discovery time. Dependent packages are shown as straight lines and upstream packages as dotted lines.

Figure 9 presents the Kaplan-Meier survival curves of the probability for event "package is fixed", w.r.t. discovery date of the vulnerability. Straight and dotted lines correspond to the time it takes for a package to be fixed for the dependent and upstream packages, respectively. Dotted lines thus correspond to the survival curves of Figure 8.

We observe that dependent packages need considerably more time to be freed from vulnerabilities than their upstream packages.

[8] This is an over-approximation, as we cannot be sure that the (affected release of a) dependent is truly concerned by the vulnerability in its dependency.

We confirmed this using log-rank tests, and found a statistically significant difference between dependent packages and their upstream packages in the time required to fix a vulnerability for medium and high severity vulnerabilities. While 50% of the upstream packages are fixed within the month, only 33.1% of the dependent packages are fixed within this time frame, and it takes nearly 14 months to fix 50% of them.

These results illustrate the importance of having continuous monitoring of dependencies, so that maintainers of dependent packages can be notified quickly of the presence of a vulnerability in one of their dependencies, as well as the presence of a fix.

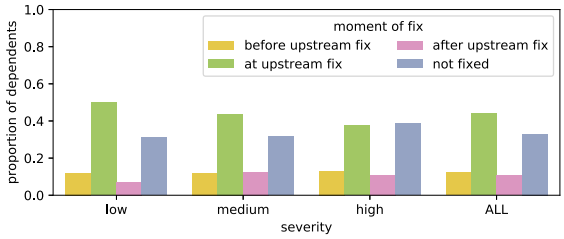

Figure 10: Moment of fix for affected dependents.

We analysed in more depth the moment that affected dependents are freed from a vulnerability in their upstream packages. Figure 10 presents the proportion of affected dependents that are fixed before, simultaneously with, or after the fix of the upstream package. This figure also shows the proportion of dependents that are never fixed.

The results indicate that only a small fraction (12.2%) of all dependents are fixed before the upstream fix, either by removing the dependency to the vulnerable package or by rolling back to a non-affected release. Most dependents (44%) are fixed at the same time as the upstream fix. This implies that the dependent package automatically benefits from the fix, because the specified dependency constraint allows to update to releases containing the fix. We also observe that 10.8% of dependents are fixed after the upstream fix, suggesting that "manually" managing dependencies and changing dependency constraints requires effort and leads to delays in benefiting from vulnerability fixes.

More importantly, Figure 10 reveals that more than 33% of all dependents affected by an upstream vulnerability are not (yet) fixed. Zooming in on those dependent packages that have not (yet) been fixed from a vulnerability in an upstream package, Figure 11 presents the proportion of these affected dependents according to the status of the upstream fix.

We observe that only 4.8% of them are not fixed because the vulnerable upstream package has not yet published a fix ("no upstream fix" in Figure 11). For affected dependent packages for which an upstream fix is available, we consider their latest update date. The packages that had some more recent update yet did not integrate the upstream fix ("updated since upstream fix") represent 21.3% of the dependents that are not fixed. For dependent packages that were not updated since an upstream fix was available we consider two cases, to address the fact that recent upstream fixes may not have had sufficient time to be taken into account in affected dependent packages. Affected dependent packages for which the upstream

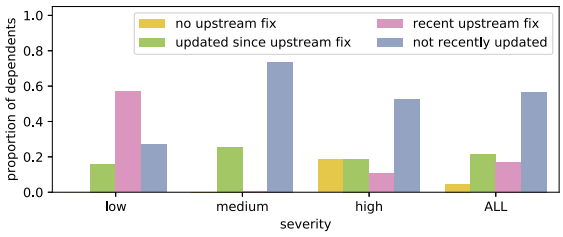

Figure 11: Status of affected dependents that are not fixed.

fix was published during the last 6 months ("recent upstream fix") represent 17.2% of the unfixed dependents.[9] Affected dependent packages that have not been updated during the last 6 months ("not recently updated") represent 56.6% of the unfixed dependents. The reason why such a large majority of affected dependents continue to remain vulnerable even after the availability of a fix appears to be that these dependent packages are no longer actively maintained.

> ***Findings.*** More than half of all dependent packages are affected by vulnerabilities in upstream packages. While 44% of these affected dependents are automatically fixed together with the vulnerable upstream package, about the same percentage need considerably more time than their upstream packages to eliminate vulnerabilities. A large fraction of affected dependent packages are not updated, even if an upstream fix is available. Improper or too restrictive use of dependency constraints and unmaintained packages are the main causes of dependent packages remaining vulnerable even if fixes are available.

8 DISCUSSION

The npm package dependency network is witnessing an exponential growth over time, both in terms of number of packages and of package dependencies [9, 10]. It is therefore not surprising that we found an increasing number of newly discovered vulnerabilities over time (RQ_0). Most of these vulnerabilities are of medium or high severity, possibly because developers consider low severity vulnerabilities as less important or fix them immediately, and hence disclose them less frequently.

We also found that the impact of vulnerabilities on the package release history was considerable (RQ_1). For those packages that were reported to suffer from a vulnerability, in 75% of the cases more than 90% of all package releases were affected by the vulnerability. This suggests that it can take a long time to fix vulnerabilities.

To avoid attackers abusing vulnerabilities present in a package, it is not only important to fix vulnerabilities rapidly once they are discovered, but it is equally important to discover them rapidly. Our findings for RQ_2 revealed that this is not the case in practice. Regardless of the severity, it takes more than 24 months to discover 50% of all vulnerabilities. For high severity vulnerabilities this is even 31 months.

RQ_2 also revealed that most vulnerabilities are found in packages older than 28 months, probably because younger packages have not yet had time to reach a wide audience and to receive as much

[9] We also considered other thresholds (3, 9 and 12 months) and obtained similar results.

security related attention as older packages. We also found that vulnerabilities of high severity are observed more often in older packages. This finding might be explained by the fact that older packages are more complex or more widely used than younger packages, making vulnerabilities more intricate to fix and more impactful, two factors increasing the computed severity.

> **Actionable result:** Package maintainers should strive to discover vulnerabilities sooner, to avoid packages being vulnerable to attackers without its maintainers even being aware of it. More and better tool support is needed to help discovering these vulnerabilities.

Only about one third of all medium and high severity vulnerabilities are already fixed when their discovery is reported (RQ_3). The remaining two thirds are fixed later, or never at all. More than half of all vulnerabilities get fixed *after* their discovery date, but *before* official publication of the vulnerability, most likely because package maintainers strive to fix discovered vulnerabilities before they become public. Nevertheless, there is still a non-negligible amount of vulnerable packages that have a high risk of being abused, either because the vulnerability is only fixed after its publication date (11.8% of all packages) or never at all (3.5% of all packages).

> **Actionable result:** Package maintainers should fix their vulnerable packages, ideally before the vulnerability is publicly announced. Package managers should strive to detect and deprecate packages that continue to suffer from vulnerabilities, and should warn users when a package known to be vulnerable is installed or becomes the target of a dependency.

Vulnerable package releases may also affect releases of dependent packages. One of the developers of the libraries.io package manager monitor confirmed the importance of package dependencies in the context of security vulnerabilities (private communication, 11 September 2017): *"it is fundamental to understand security implications of *all* dependencies"* and *"to understand the security problems of a product, they need to understand the security problems of all its dependencies."*

Among the 133,602 packages depending on potentially vulnerable packages, 72,470 dependent packages were affected by a vulnerability in an upstream package, accounting for 411,169 affected releases. This implies that the presence of explicit package dependencies increases the "impact" of security vulnerabilities by more than an order of magnitude.

Not only the number of affected dependents is problematic, but also the time required to free them from vulnerabilities in upstream releases. RQ_4 revealed some surprising and worrisome findings: a large number of dependents remain affected for a long time after the upstream package on which they depend has already eliminated the vulnerability. This may be caused by dependency constraints preventing dependents from automatically benefiting from upstream fixes.

> **Actionable result:** From the perspective of a package user, there is no difference between a package containing a vulnerability and a package being exposed to a vulnerability through a dependency. Package maintainers should strive to reduce their packages being at risk due to vulnerabilities in required packages, by monitoring the security reports issued for these packages, and by adapting their dependency constraints to quickly benefit from security fixes.

Automated tool support can be of great help to achieve these goals. For this purpose, in October 2017 GitHub introduced support for tracking reported vulnerabilities in dependencies, and for alerting affected dependents of potential security problems. Since our dataset was gathered on 2017-11-09, it was too early to observe any beneficial effects of such support.

Besides GitHub's vulnerability tracking mechanism, many other tools monitor vulnerable dependencies.[10] Tools supporting npm and JavaScript include commercial services (Snyk, Gemnasium, SRC:CLR), as well as free/open-source solutions (RetireJS, Node Security Project, OSSIndex, Dependency-check). While all these tools search existing databases for vulnerability information, some of them also recover information from their own private databases and from mailing lists, bug-tracking systems and blogs. This diversity of tools and data sources when reporting security issues makes it impossible to get a complete historic view on the ecosystem's vulnerabilities.

> **Actionable result:** Package maintainers and tool developers should maintain and contribute to a single and common vulnerabilitiy database reporting all vulnerabilities related to the package dependency network.

9 THREATS TO VALIDITY

Our empirical findings are restricted by the relatively limited number (700) of security reports for npm available in the dataset. Our results therefore only represent a "lower bound" of the actual number of vulnerabilities, as it is very likely that many vulnerabilities that have been found and fixed are not (yet) reported. This threat is associated with the quality and completeness of the vulnerability database used for the analysis [18].

When analysing the vulnerability constraints (i.e., the version constraints specifying the range of package releases that are considered vulnerable), we observed that the large majority of these constraints specify only an upper bound on the versions. It is difficult to say whether this effectively means that all previous releases are affected, or whether no lower bound is specified out of ignorance. This may affect the reported "first affected release" and the computed duration of a vulnerability. There is no easy way to mitigate this limitation.

Our analysis combined security reports from Snyk.io with historical data of package releases and package dependencies from libraries.io. For one package and a few releases referenced in the security reports, we found no match in libraries.io, probably because they were removed from the npm registry before libraries.io started collecting the data. While this could slightly affect our analyses, the impact remains limited and does not affect our main findings.

With respect to the analysis of affected dependent packages (RQ_4), we only considered runtime dependencies between packages (i.e., dependencies that are required to install and run the package). While packages can also specify development dependencies (i.e.,

[10] https://techbeacon.com/13-tools-checking-security-risk-open-source-dependencies-0

dependencies needed during development of the package, e.g., test frameworks), we ignored them in our analysis because they are unlikely to affect the production environment.

Another threat concerns whether the vulnerable functionality of an upstream package actually affects its dependent packages. A recent report by SAFECode [3] suggests that developers must also evaluate whether a product uses the specific functionality of vulnerable third-party components. However, it is not feasible to perform such a task in an automated way for hundreds or thousands of modules and therefore, our work provides an upper limit on the number of dependent packages that can be affected by dependencies affected by security vulnerabilities. We prefer to stay on the safe side, by always considering an affected dependent to be vulnerable, even if it does not use any functionality involved in the vulnerability. We believe it is good practice for maintainers to avoid depending on versions of upstream packages that are known to be vulnerable, regardless of whether the vulnerability will manifest itself.

A final threat relates to the generalisability of our findings. With some amount of effort, the approach we followed could be replicated on other package managers and package dependency networks. However, the results may be quite different from those we obtained for npm, due to different policies, practices and culture that may be specific to the package manager under study [4, 9].

10 RELATED WORK

Package dependency networks. The research community of software repository mining has carried out quite some research on understanding the evolution of large networks of software package dependencies. We only focus on the npm package dependency network since it was the subject of our empirical analysis.

Wittern et al. [25] analysed a subset of npm packages, focusing on the evolution of characteristics such as their dependencies, update frequency, popularity, versioning policy and so on. They observed that package maintainers use different versioning conventions, despite the prescribed usage of semantic versioning, resulting in different version adoption ratios. Abdalkareem et al. [2] empirically analysed "trivial" npm packages and the risk of depending on such packages. The results were inconclusive, in the sense that depending on trivial packages can be useful and risk-free if they are well implemented and tested.

Decan et al. [8, 9] quantitatively compared npm with three other ecosystems (CRAN, PyPI and RubyGems), focusing on the topology of their package dependency networks. A follow-up study [10] expanded the comparison to 7 different packaging networks: Cargo, CRAN, CPAN, npm, NuGet, Packagist and RubyGems. They observed important differences between the considered ecosystems, which could be explained by ecosystem specific factors. Similarly, through a qualitative analysis based on developer interviews, Bogart et al. [4] compared npm with two other ecosystems (CRAN and Eclipse) in order to understand the impact of community values, tools and policies on breaking changes. Specifically related to package dependencies, they identified two main types of mitigation strategies adopted by package developers to reduce their exposure to changes in other packages: limiting the number of dependencies; and depending only on "trusted" packages.

Security vulnerabilities. Software repository mining research has focused a great deal on empirically studying defects or bugs in evolving software systems, and to a lesser extent on security vulnerabilities. Camillo et al. [6] empirically analysed the development history of the Chromium project to examine the relationship between bugs and vulnerabilities. Their statistical analysis revealed that bugs and vulnerabilities are empirically dissimilar groups, warranting the need for research targeting vulnerabilities specifically.

Perhaps the most closely related to our work is Hejderup's master thesis that focused on security vulnerabilities in npm [13]. He investigated how long it takes for a developer to publish a fix after the publication of a security report involving one of his packages, and how many packages are affected by a vulnerability. The set of vulnerabilities he studied was very limited compared to our own dataset: it consisted of only 19 vulnerable packages and 1,029 vulnerable dependent packages. In addition, the way he identified dependent package releases affected by vulnerabilities undermine the validity of his results. More concretely, he considered the latest release (among all releases) satisfying the dependency constraint, ignoring that not all releases are available at installation time and that the release selected for installation depends on the installation date. In doing so, he either omitted affected packages in the case where the vulnerability has been fixed later on, or included packages that became vulnerable only in some future release. For these reasons, the reported findings differ from our own findings that are based on a larger dataset with a more precise way of measuring releases affected by vulnerabilities.

Several works investigate the impact of relying on vulnerable dependencies, without specifically focusing on package dependency networks. Lauinger et al. [16] examined the security implications of relying on client-side JavaScript library usage. They examined 133K websites and found that 37% of these websites use at least one library with a known vulnerability. Kula et al. [15] investigated 4,659 GitHub projects and more than 850K library dependency migrations to find developer responsiveness to existing security awareness mechanisms. They found that developers do not tend to update third-party libraries, especially to fix vulnerabilities, while 81.5% of the studied systems remain with outdated dependencies. By interviewing developers, they also found that 69% of them were unaware of their vulnerable dependencies and their decision to update their dependencies was based on project-specific priorities.

Cox et al. [7] acknowledged the increased security and backward incompatibility risk when depending on outdated components. Based on an industry benchmark, they evaluated a system-level metric of "dependency freshness" and investigated the relationship between outdated dependencies and security vulnerabilities. They revealed that systems using outdated dependencies are four times more likely to have security issues than systems that are up-to-date. Derr et al. [11] investigated how outdated libraries are in the Android ecosystem by conducting a survey with more than 200 app developers. They reported that most apps use outdated libraries and that almost 98% of 17K actively used library versions with a known security vulnerability could be easily fixed by updating the library to a fixed version. Massacci et al. [17] analyzed the evolution of Firefox and found that a large fraction of vulnerabilities apply to code that is no longer maintained in older versions, thus leaving users that slowly upgrade to newer versions exposed to attacks.

Cadariu et al. [5] presented an automated Vulnerability Alert Service (VAS) to track known vulnerabilities in industrial software systems throughout their life cycle. They evaluated the usefulness of this tool in the context of external software product quality monitoring, and found that depending on external components with known security vulnerabilities is a commonplace. Pham et al. [21] conducted an empirical study and found that software vulnerabilities are often recurring due to software reuse. They developed SecureSync, a tool to automatically detect recurring software vulnerabilities on the systems that reuse source code or libraries. The tool leads to high accuracy in the vulnerability detection task and also identifies vulnerable code that has not been reported or fixed yet. Di Penta et al. [20] conducted an empirical study to analyze the evolution of source code vulnerabilities using three static analysis tools. They report that they didn't find a "silver bullet" detection tool since there is almost no overlap between the results of different tools.

11 FUTURE WORK

Our dataset was restricted to the 610K packages within npm. In practice, the JavaScript ecosystem is much larger, including packages on public repositories like GitHub. npm packages may have external dependencies to such packages and vice versa. A recent striking example was a vulnerability affecting Electron on the Windows platform.[11] While Electron is distributed through the npm package manager, the impact of the vulnerability reached far beyond this ecosystem, affecting many popular apps including Skype and Slack. Considering such an "extended" package dependency network will reveal a further increase in the potential impact of vulnerable packages, and requires monitoring tools capable of combining and integrating data from different sources.

Similar in spirit to the work of Decan et al. [10], we aim to replicate our analysis on other open source package dependency networks. This will enable us to study whether and why some dependency networks are more fragile to security vulnerabilities than others, and how these vulnerabilities propagate across the network. Community-specific policies and practices, such as the way in which version constraints are being used, may affect an ecosystem's resilience to vulnerabilities.

The empirical findings of this paper were reported using the vulnerability severity type (i.e., high, medium or low). An alternative way to study vulnerabilities and their impact would be to rely on their CWE type (Common Weakness Enumeration), a community-developed list of common software security weaknesses.[12] The CWE type could be used to assess if certain vulnerabilities behave differently (e.g., they take longer to detect or fix or they are more prominent). While there are more than 700 CWE types, we only found 40 distinct CWE types in the 399 security reports we analysed for npm. The three most frequently observed types were CWE-79 "Cross-site Scripting" (105 occurrences), CWE-400 "Resource Exhaustion" (47 occ.) and CWE-94 "Code Injection" (18 occ.). We did not use this information in our empirical analysis mainly because the number of security reports per CWE type is low (median value is 3), hence any analysis broken down by CWE type is likely to lead to statistically insignificant results.

This paper only considered vulnerable package releases and their *direct* dependents, mainly because of the size of the npm package dependency network and the computation time required to identify and process all dependency constraints. It would be very desirable to take into account transitive dependencies as well, because package maintainers typically have a less clear idea on which packages they depend indirectly. Taking into account transitive dependencies is very likely to increase the impact of vulnerabilities by another order of magnitude.

12 CONCLUSION

This paper presented an empirical study of the evolution of security vulnerabilities in the npm package dependency network for JavaScript packages. Based on survival analysis, we studied how fast vulnerabilities are discovered, how long packages remain vulnerable, when vulnerabilities are fixed, and how vulnerable packages affect other packages that rely on them.

Our approach aimed at identifying packages releases affected by a vulnerability in a precise way. We did so by taking into account the version constraints specifying the security vulnerability, the version constraints specified for package dependencies as well as the exact release that is selected for installation depending on when the installation is performed.

We studied 399 security reports, affecting 269 distinct packages and 6,752 releases of these packages. Considering package dependencies and taking into account dependency constraints, 72,470 other packages are affected by these vulnerable releases.

We observed that it often takes a long time to discover vulnerabilities since their introduction. One third of the vulnerabilities is fixed at (or before) their discovery date, half of the vulnerabilities take longer but are fixed before their publication date. The remaining 15% or so are considered high risk since they are either fixed after public announcement of the vulnerability, or not fixed at all.

The presence of dependency constraints plays an important role in fixing vulnerabilities. More than 40% of all package releases depending on a vulnerable package release cannot be fixed automatically by depending on a more recent release, because the imposed dependency constraints do not allow them to be installed.

These findings scream for a higher awareness among npm package maintainers of the risks incurred by security vulnerabilities, not only at the level of individual packages, but also at a wider ecosystem level by considering package dependencies. Package maintainers should also rely on better use of policies and automated tools to detect and fix vulnerabilities faster, and to reduce the impact of vulnerabilities on dependent packages.

ACKNOWLEDGMENTS

This research was carried out in the context of FRQ-FNRS collaborative research project R.60.04.18.F "SECOHealth", FNRS Research Credit J.0023.16 "Analysis of Software Project Survival" and Excellence of Science project 30446992 SECO-Assist financed by FWO - Vlaanderen and F.R.S.-FNRS.

[11] https://www.theregister.co.uk/2018/01/24/skype_signal_slack_nherit_electron_vuln
[12] see https://cwe.mitre.org

REFERENCES

[1] O. Aalen, O. Borgan, and H. Gjessing. 2008. *Survival and Event History Analysis: A Process Point of View.* Springer. https://doi.org/10.1007/978-0-387-68560-1

[2] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab. 2017. Why do developers use trivial packages? An empirical case study on npm. In *Joint Meeting on Foundations of Software Engineering (ESEC/FSE).* 385–395. https://doi.org/10.1145/3106237.3106267

[3] P. Bisht, M. Heim, M. Ifland, M. Scovetta, and T. Skinner. 2017. Managing Security Risks Inherent in the Use of Third-party Components. (2017). Executive Information Systems, Inc., White Paper No. Eleven.

[4] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung. 2016. How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems. In *Int'l Symp. Foundations of Software Engineering.* https://doi.org/10.1145/2950290.2950325

[5] M. Cadariu, E. Bouwers, J. Visser, and A. van Deursen. 2015. Tracking known security vulnerabilities in proprietary software systems. In *Int'l Conf. Software Analysis, Evolution, and Reengineering.* 516–519. https://doi.org/10.1109/SANER.2015.7081868

[6] F. Camilo, A. Meneely, and M. Nagappan. 2015. Do Bugs Foreshadow Vulnerabilities? A Study of the Chromium Project. In *Working Conf. Mining Software Repositories.* 269–279. https://doi.org/10.1109/MSR.2015.32

[7] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser. 2015. Measuring Dependency Freshness in Software Systems. In *Int'l Conf. Software Engineering.* IEEE Press, 109–118. https://doi.org/10.1109/ICSE.2015.140

[8] A. Decan, T. Mens, and M. Claes. 2016. On the Topology of Package Dependency Networks — A Comparison of Three Programming Language Ecosystems. In *European Conf. Software Architecture Workshops.* ACM . https://doi.org/10.1145/2993412.3003382

[9] A. Decan, T. Mens, and M. Claes. 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In *Int'l Conf. Software Analysis, Evolution, and Reengineering.* 2–12. https://doi.org/10.1109/SANER.2017.7884604

[10] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2018. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* (10 Feb 2018). https://doi.org/10.1007/s10664-017-9589-y

[11] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes. 2017. Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android. In *ACM Conf. on Computer and Communications Security.* https://doi.org/10.1145/3133956.3134059

[12] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman. 2014. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference (IMC '14).* ACM, New York, NY, USA, 475–488. https://doi.org/10.1145/2663716.2663755

[13] J.I. Hejderup. 2015. *In Dependencies We Trust: How vulnerable are dependencies in software modules?* Master's thesis. Delft University of Technology.

[14] E. L. Kaplan and P. Meier. 2012. Nonparametric Estimation from Incomplete Observations. *J. American Statistical Association* 53, 282 (2012), 457–481. https://doi.org/10.23072281868

[15] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue. 2017. Do developers update their library dependencies? *Empirical Software Engineering* (11 May 2017). https://doi.org/10.1007/s10664-017-9521-5

[16] T. Lauinger, A. Chaabane, W. Robertson, C. Wilson, and E. Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *ISOC Network and Distributed System Security Symposium.* https://doi.org/10.14722/ndss.2017.23414

[17] F. Massacci, S. Neuhaus, and V. H. Nguyen. 2011. After-life Vulnerabilities: A Study on Firefox Evolution, Its Vulnerabilities, and Fixes. In *Proceedings of the Third International Conference on Engineering Secure Software and Systems (ESSoS'11).* Springer-Verlag, Berlin, Heidelberg, 195–208. http://dl.acm.org/citation.cfm?id=1946341.1946361

[18] F. Massacci and V. H. Nguyen. 2010. Which is the Right Source for Vulnerability Studies?: An Empirical Analysis on Mozilla Firefox. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics (MetriSec '10).* ACM. https://doi.org/10.1145/1853919.1853925

[19] A. Nesbitt and B. Nickolls. 2017. Libraries.io Open Source Repository and Dependency Metadata. (June 2017). https://doi.org/10.5281/zenodo.808273

[20] M. Di Penta, L. Cerulo, and L. Aversano. 2009. The life and death of statically detected vulnerabilities: An empirical study. *Information and Software Technology* 51, 10 (2009), 1469 – 1484. https://doi.org/10.1016/j.infsof.2009.04.013

[21] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. 2010. Detection of Recurring Software Vulnerabilities. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE '10).* ACM, New York, NY, USA, 447–456. https://doi.org/10.1145/1858996.1859089

[22] snyk. 2017. The State of Open Source Security. https://snyk.io/stateofossecurity/. (November 2017).

[23] H. H. Thompson. 2003. Why security testing is hard. *IEEE Security Privacy* 1, 4 (July 2003), 83–86. https://doi.org/10.1109/MSECP.2003.1219078

[24] J. Williams and A. Dabirsiaghi. 2014. *The Unfortunate Reality of Insecure Libraries.* White Paper. Contrast Security.

[25] E. Wittern, P. Suter, and S. Rajagopalan. 2016. A Look at the Dynamics of the JavaScript Package Ecosystem. In *Int'l Conf. Mining Software Repositories.* ACM, 351–361. https://doi.org/10.1145/2901739.2901743