On the Discoverability of npm Vulnerabilities in Node.js Projects

MAHMOUD ALFADEL, DIEGO ELIAS COSTA, and EMAD SHIHAB, Concordia University, Canada
BRAM ADAMS, Queen's University, Canada

The reliance on vulnerable dependencies is a major threat to software systems. Dependency vulnerabilities are common and remain undisclosed for years. However, once the vulnerability is discovered and publicly known to the community, the risk of exploitation reaches its peak, and developers have to work fast to remediate the problem. While there has been a lot of research to characterize vulnerabilities in software ecosystems, none have explored the problem taking the discoverability into account.

Therefore, we perform a large-scale empirical study examining 6,546 Node.js applications. We define three discoverability levels based on vulnerabilities lifecycle (undisclosed, reported, and public). We find that although the majority of the affected applications (99.42%) depend on undisclosed vulnerable packages, 206 (4.63%) applications were exposed to dependencies with public vulnerabilities. The major culprit for the applications being affected by public vulnerabilities is the lack of dependency updates; in 90.8% of the cases, a fix is available but not patched by application maintainers. Moreover, we find that applications remain affected by public vulnerabilities for a long time (103 days). Finally, we devise DepReveal, a tool that supports our discoverability analysis approach, to help developers better understand vulnerabilities in their application dependencies and plan their project maintenance.

CCS Concepts: • **Software and its engineering → Software packages and repositories**.

Additional Key Words and Phrases: Open source software, Software packages, Software ecosystems, Dependency vulnerabilities

1 INTRODUCTION

Modern software systems are developed with increasingly more reliance on open source software packages (dependencies). This dependence on open source packages is highly beneficial to software development, as it speeds up development tasks and improves software quality [24, 49], but has implications on the security of those systems [22, 34]. Dependencies with security vulnerabilities have the potential to expose hundreds of applications to security breaches, causing huge financial and reputation damages. One such example is the Equifax incident [38], where a vulnerability on a single dependency of Equifax (the Apache Struts package) led to unauthorized access to hundreds of millions of consumers' personal information and credit card numbers.

The recent popularity of software packages has only magnified the problem. For example, npm (the main package manager used by Node.js applications) hosts more than 1.73M npm packages available for the JavaScript community. Prior studies (e.g. [70]) showed that a significant proportion (up to 40%) of all npm packages use code with at least one publicly known vulnerability, which increases the risk of a vulnerable package in a software application.

In fact, an essential factor to evaluate the impact of vulnerable packages in an application is the *discoverability* of vulnerabilities, i.e, how publicly known is the package vulnerability [55]. As an example, the vulnerability that caused the Heartbleed incident was not discovered in the OpenSSL package for years [7], but once published, more than 4 thousand exploit attempts were registered by researchers [36]. While undisclosed (unknown) vulnerabilities

Authors' addresses: Mahmoud Alfadel; Diego Elias Costa; Emad Shihab, Concordia University, Montreal, Canada; Bram Adams, Queen's University, Kingston, Canada.

1049-331X/2022/11-ART $15.00
https://doi.org/10.1145/3571848

can be exploited by attackers who are aware of the breach, once vulnerabilities become public, the chances of exploitation reach their peak and developers need to act fast to mitigate the security risks.

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

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

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

Finally, it is critical that applications patch public dependency vulnerabilities as soon as possible to avoid potential exploits. Hence, to understand how fast vulnerable dependencies are patched in the applications, we examine **(RQ3)** how long it takes for public dependency vulnerabilities to be removed from the applications. We find that the applications take a substantially long time (103 days) before *public* dependency vulnerabilities are fixed in the applications.

In summary, this paper makes the following main contributions:

- To the best of our knowledge, we conduct the first empirical study on 6,546 open-source Node.js applications to determine the prevalence of affected applications that rely on vulnerable dependencies taking into consideration the discoverability levels. We also examine why these applications end up depending on vulnerable versions of the package in order to better understand how we can mitigate such issues.
- We develop DepReveal, a prototype tool that generates historical analytical reports of npm packages used in a GitHub Node.js project. The tool (DepReveal) is only a proof of concept that has the potential to help developers when dealing with post-mortem analysis of security vulnerabilities.
- We provide a replication package comprising the scripts and the applications dataset that we used in this study as a means to bootstrap other studies in the area.

Paper organization. The rest of the paper is organized as follows. Section 2 describes how npm manages dependencies in Node.js applications. Section 3 introduces our vulnerability classification used in this study. Section 4 describes our study design. Section 5 explains how we identify and classify vulnerable dependencies in Node.js applications. Section 6 presents our results. Section 7 discusses our results further. Section 8 presents our tool. Section 9 discusses the implications of our findings. Section 10 discusses the related work. Section 11 presents the threats to validity. Section 12 concludes our paper.

2 NPM DEPENDENCY MANAGEMENT

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

Node Package Manager (npm) is the de-facto package manager used by Node.js applications to handle their dependencies [54]. npm has a registry where packages are published and maintained. To date, npm registry hosts more than 2M packages [11], and has had the highest growth rate in terms of packages amongst all known programming languages [12].

To determine the discoverability of vulnerable dependencies in Node.js applications, we need to understand two important mechanisms of the npm ecosystem: 1) how Node.js applications specify their npm dependencies and 2) how npm resolves a dependency version, i.e., find the dependency version to install in a Node.js application. Node.js applications specify their dependencies in a JSON-format file, called package.json, which lists the dependencies and their versioning constraints. The versioning constraint is a convention to specify the dependency version(s) of the package that an application is willing to depend upon. The version constraints can be static, requiring a specific version of the dependency (e.g., "P:1.0.0"), or dynamic specifying a range of versions of the dependency (e.g., "P:>1.0.0"). Typically, developers use dynamic versioning constraints if they want to install the latest version of a dependency, allowing them to get the latest updates/security fixes of the package. When a dynamic version is used, the resolved version (i.e., the actual version) corresponds to the latest installable version that satisfies the constraint [31].

Node.js applications can specify two sets of dependencies in their package.json file: development and production dependencies. Development dependencies are installed only on development environments, and consequently, issues that may arise from them (e.g., vulnerabilities and bugs) have no impact on production environments. On the other hand, production dependencies (also called runtime dependencies) are installed on both production and development environments. In our work, we only consider direct production dependencies since they are the ones that impact the production environment [35].

3 ABOUT DISCOVERABILITY

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

3.1 Vulnerability Lifecycle

Typically, a vulnerability goes through a number of different stages [1, 13]:

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

- **Publication with a fix.** Another (and more common) way that a vulnerability can be published is when a fix is provided by the package maintainers. If a fix is provided (before 45 days), then npm publishes the vulnerability along with the version of the package that fixes the vulnerability.

3.2 Discoverability Levels

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

(1) ***Undisclosed*: before report.** Since very little is known about a vulnerability before it is reported, i.e., dependency vulnerabilities in the application are not disclosed yet, we believe that the chances of being exploited are low. We classify all dependency vulnerabilities in the application at this stage as *undisclosed* dependency vulnerabilities.
(2) ***Reported*: after report & before publication.** Once a vulnerability has been discovered and reported, the general public is not yet aware of the vulnerability, as the process is conducted internally by the npm team. Still, there is a chance that others may know about the vulnerability and has the capability to exploit, so we consider the chances of exploit to be at a medium level. We classify dependency vulnerabilities in the application at this stage as *reported* dependency vulnerabilities.
(3) ***Public*: after publication.** After publication the chance of exploitability is at its highest. A proof-of-concept is often published [10] alongside the vulnerability report, explaining how the vulnerability could be exploited. The threat of this vulnerability can only be mitigated once package maintainers release another version fixing the vulnerability and the application developers update their dependency accordingly. Failing to perform both these tasks in a timely fashion may put the application at higher security risk. We classify dependency vulnerabilities in the application at this stage as *public* dependency vulnerabilities.

Note that our discoverability levels are not based on a heuristic, but rather on the existing typical vulnerability disclosure process. Such a disclosure process has been discussed and mentioned in previous studies [2, 34]. For example, Decan et al. [34] analyzed different aspects related to vulnerability lifecycle, e.g., how long it takes for packages vulnerabilities before being disclosed. Also, they analyzed how long a vulnerability report takes before being publicly disclosed (public level). Therefore, our levels are defined based on existing practices of reporting security vulnerabilities.

4 STUDY DESIGN

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

4.1 Research Questions

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

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

4.2 Data Collection

Our study examines vulnerable dependencies in Node.js applications, particularly applications that use the Node Packages Manager (npm) as dependency management [54]. We opt to focus on Node.js applications due to its popularity and importance in the current development landscape. JavaScript is currently the most popular

Table 1. Statistics of the 6,546 studied Node.js applications.

Metric	Min.	Median($\tilde{x}$)	Mean(μ)	Max.
Commits	100	326	1035.47	77,271
Dependencies	3	23	27.93	134
Developers	3	5	6.33	62
Age (in years)	5	7.24	7.53	12.81
Stars	1	11	405.73	56,661
Forks	1	7	123.34	16,841

programming language in the world [62] with a vibrant and fast growing ecosystem of reusable software packages [12].

To perform our study, we leverage two datasets: (1) Node.js applications that use npm to manage their dependencies, and (2) Security vulnerabilities that affect npm packages. To do so, we **(i)** obtain the Node.js applications from GitHub, **(ii)** extract their dependencies, and **(iii)** obtain the security vulnerabilities for npm packages from npm advisories [53].

(i) Applications Dataset. To analyse a large number of open source Node.js applications that depend on npm packages, we mine the GHTorrent dataset [42] and extract information about all Node.js applications hosted on GitHub. The GHTorrent dataset contains a total of 7,863,361 JavaScript projects hosted on GitHub, of which 2,289,130 use npm as their package management platform (i.e., these projects contain a file called package.json). Moreover, since both Node.js *packages* and *applications* can use GitHub as their development repository, and our applications dataset should only contain Node.js *applications*, we filter out the GitHub projects that are actually npm *packages* by checking their GitHub URL on the npm registry. The main reason that we focused on applications and not packages is that packages become exploitable when used and deployed in an application. This filtering excludes 328,343 projects from our list of GitHub projects as they are identified as packages and not Node.js applications.

Inspired by previous studies [43, 45, 50], projects in GitHub are not always representative of mature software projects we aim to investigate. Hence, we refined the dataset to focus on projects that are active and more likely to be mature software projects, by including applications that satisfy the following criteria:

- Non-forked applications, as we do not want to have duplicated project history to bias our analysis.
- Applications that depend on more than two dependencies.
- Applications that have at least 100 commits by more than two contributors, which indicates a minimal level of commit activity.
- Applications that have had their creation date (first commit) before January 1st 2017. Since vulnerabilities take on median 3 years to be discovered [34], applications in our dataset need to have a development history long enough to have had a chance for their vulnerabilities to be discovered.
- Applications that have at least one commit after January 1st 2020, as we want to analyze applications that had some level of development activity recently.

After applying these refinement criteria, we end up with 6,546 Node.js applications that make use of npm packages. Table 1 shows the descriptive statistics on the selected Node.js applications in our dataset. Overall, the applications in our dataset have a rich development history (a median of 326 commits made by 5 developers and 7.24 years of development lifespan) and make ample use of external dependencies (a median of 23 dependencies). Inspired by prior work (e.g., [3]), we purposely did not want to restrict our project dataset to only the most

active projects because many projects are updated infrequently, but are actively used by project users. For example, Strider-CD/strider project is an open-source continuous deployment platform. Although the project is infrequently updated, it seems to be commonly used by users (more than 445 forks and 4.6K stars).

To further understand the characteristics of the projects in our dataset, we collected more statistical information. Inspired by prior work (e.g., [30, 50],), we calculated two main metrics: number of Stars & number of Forks. We found that the projects in our dataset have a community interest in them (the median number of stars = 11), and the projects are attracting several users and contributors (the median number of forks = 7). We have incorporated such analysis in Table 1.

Moreover, to shed light on the domain of the examined projects, we manually classified a sample of projects in our dataset. We randomly selected a statistically significant sample (95% confidence level) of the projects (i.e., 354 projects). Then, for each project, the first author of the paper inspected the description of the projects (utilizing the Github page and the project page of the repository) to provide a brief description of the application. After that, the author assigns a domain label for each project. The labels were also discussed by the first and second author to reach a consensus all of them. The projects are classified into the following four domains:

- **Software tools (173, 48.87%):** repositories that support software development tasks, like IDEs, package managers, deployment frameworks, and compilers (e.g., twitter/hogan.js).
- **Web applications (125, 35.31%):** repositories that provide functionalities to end-users, like browsers and text editors (e.g., atom/atom).
- **Educational projects (47, 13.28%):** repositories with documentation, tutorials, source code examples, etc. (e.g., angular/angular-phonecat).
- **System software (9, 2.54%):** repositories that provide services and infrastructure to other systems, like operating systems, middleware, servers, and databases (e.g., Strider-CD/strider).

Our classification shows that the majority of the manually analyzed projects are of interest to both software developers and project users.

(ii) Application Dependencies. After obtaining the applications dataset, we extract the history of dependency changes of all applications. This is necessary to identify the exact dependency versions that would be installed by the Node.js application at any specific point-in-time. Node.js applications specify their dependencies in a JSON-format file, called package.json, which contains the dependency list, a list of the depended upon packages and their respective version constraints. A version constraint is a configuration that specifies the dependency version(s) of the package that an application is willing to depend upon [17]. Hence, we extract all changes that touched the package.json file and associate each commit hash and commit date to their respective package.json dependency list, creating a history of dependency changes for all applications. Note that these dependencies are not yet resolved, that is, we only have the version constraints (not the versions) for the dependencies of each application.

(iii) npm Advisories Dataset. To identify the Node.js applications that depend on vulnerable packages, we need to collect information on npm vulnerable packages. We resort to the *npm advisories* registry to obtain the required information about all npm vulnerable packages [53]. The npm advisories dataset is the official registry for all vulnerability reports related to Node.js packages. This dataset provides some key information on vulnerable packages, such as the affected package, the affected package versions, and the first version in which the vulnerability has been fixed (safe version), if available. This dataset also contains the vulnerability discovery (report) time and publication time, which we use in our approach for identifying and classifying vulnerabilities (Section 5).

Our initial dataset contains 1,456 security reports that cover 1,234 vulnerable packages. Following the criteria filtration process applied by Decan et al. [34], we removed 312 vulnerable packages of the type “Malicious Package", because they do not actually introduce vulnerable code. These vulnerabilities are packages with names

Table 2. Descriptive statistics on the npm advisories dataset.

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

close to popular packages (a.k.a. typo-squatting) in an attempt to deceive users at installing harmful packages. The 312 vulnerable packages account for 312 vulnerability reports. At the end of this filtering process, we are left with 1,144 security vulnerabilities reports affecting 925 distinct vulnerable packages. These packages have combined 38,562 distinct package versions of which 20,206 are affected by vulnerabilities from our report. The collected advisories dataset covers vulnerability reports created between October 2015 and May 2020. Table 2 shows the summary statistics for vulnerability reports on npm packages.

5 IDENTIFYING AND CLASSIFYING VULNERABLE DEPENDENCIES IN NODE.JS APPLICATIONS

In this section, we explain how we classify the discoverability levels of vulnerable dependencies and how we use these levels to classify Node.js applications.

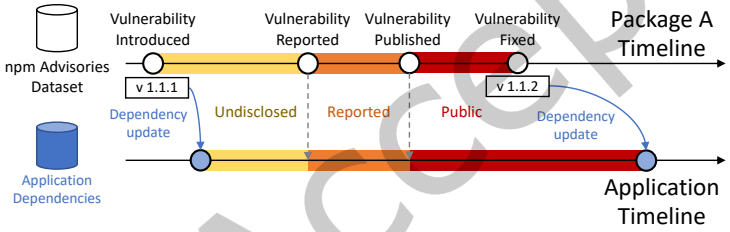

Fig. 1. Illustration of the methodology for classifying the discoverability level of a single vulnerable dependency (Package A) for an application.

We illustrate our methodology in Figure 1, on an example of an application with a single vulnerable dependency. As we can observe, the timelines of the discoverability levels of both the vulnerable package and the application are different. In the example of Figure 1, the application is only affected by a vulnerable dependency once it starts depending on the first vulnerable version (v 1.1.1). Similarly, even if the package latest release contains a fix to the vulnerability, the application can only benefit from it once it updates to the fixed version (v 1.1.2). This is different for the changes of discoverability levels once the vulnerability is made public. Due to the open nature of open source software, as soon as a vulnerability is published, any attacker in potential can identify that the application depends on the vulnerable version of Package A.

The goal of our study is to investigate how often Node.js applications depend on vulnerabilities that are hidden, reported and public. To make our analysis feasible, we focus on classifying applications at one specific point in time of the application development history, which we call the *analyzed snapshot time*. We accomplish this by leveraging a 3-step approach. Figure 2 provides an overview of our approach, which we detail below:

Step 1. Extract dependencies and resolve versions. The goal of this step is to extract the application dependencies and find the dependency version that would be installed at the analyzed snapshot time. For each application, we extract the dependency list (with the versioning constraints) at that snapshot time from the history of dependency changes. After that, to find the actual version of each dependency at the analyzed

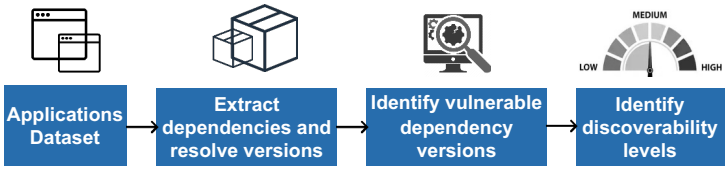

Fig. 2. Approach for identifying and classifying Node.js applications affected by vulnerable dependencies.

snapshot, we utilize the *semver* tool [61]. This tool is used by npm to resolve versioning constraint in Node.js applications, and it provides several modules and methods that support versioning schemes [15]. For example, we use the module *maxSatisfying(versions, range)* which returns the highest version in the list that satisfies the range.

We included one additional restriction to *semver*, that the satisfying version should have been released (in the npm registry) before the analyzed snapshot time. For example, an application can specify a versioning constraint ("P:>1.0.0") at the snapshot May 1st 2016. Hence, the actual installed version is the latest version that is greater than 1.0.0 and also has been released in the npm registry before May 1st 2016. This step allows us to find the installed version of the dependency at the analyzed snapshot time.

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

Step 3. Identify discoverability levels of vulnerable versions. Once we identify the vulnerable dependency versions at the analyzed snapshot time, we classify each vulnerable dependency version using one of the discoverability levels we defined in Section 3.2. To that aim, for each vulnerable version, we compare its vulnerability *discovery (report)* and *publication* time to the analyzed snapshot time. As we stated previously (in Section 3.2), if the vulnerability was made public before the snapshot time, we mark the dependency version as having a *public* vulnerability. If the vulnerability of the dependency was not published but only discovered (reported) before the application's snapshot time, the vulnerable dependency version is considered to have a *reported* vulnerability. And finally, if the vulnerability was neither published nor discovered (reported) before the analyzed snapshot time, then we classify the dependency version as a *hidden* vulnerability. In cases where more than one vulnerability affects the vulnerable dependency version, we label the vulnerable dependency version with the highest level. For example, if we find that the vulnerable version of the dependency is affected by two vulnerabilities, one classified as hidden and the other classified as public, we label the dependency version as having a public vulnerability, at that snapshot time.

Replication Package. While the proposed approach may seem simplistic in its principle, it comprises several technical challenges of processing data from npm registry API, GitHub API, and vulnerability advisories reports, especially when considering the entire application development history. To facilitate the reproduction and foment further research in the field, we make a well-documented replication package publicly available [16].

6 STUDY RESULTS

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

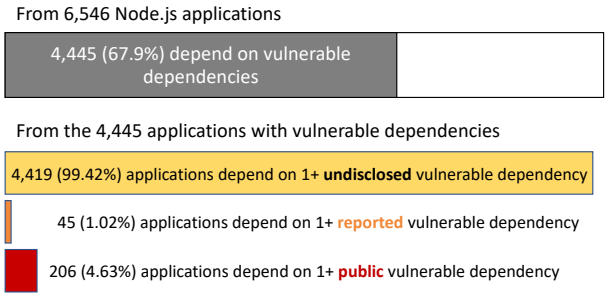

Fig. 3. Bar-plots showing the share of the examined applications with one or more (1+) vulnerable dependency, overall and per discoverability levels.

RQ_1: How often Node.js applications depend on vulnerable dependencies? How discoverable are their vulnerable dependencies?

Motivation: Previous studies have shown that security vulnerabilities are very common in the npm ecosystem, with nearly 40% of all npm packages relying on code with known vulnerabilities [70]. However, vulnerable dependencies can only be exploited once deployed in applications: how many of our studied applications depend on vulnerable dependencies? Moreover, given that the discoverability is essential in assessing the threat of a security vulnerability [55], we want to quantify how many studied Node.js applications depend on undisclosed (low risk), reported (medium risk), and public vulnerabilities (high risk), at the analyzed time. Answering these questions will give us a better assessment on the exposure of Node.js applications to dependency vulnerabilities.

Approach: To reduce the biases in our analysis, we need to account for the time it takes to discover a vulnerability. Prior work showed that vulnerabilities in npm packages take on median 3 years to be discovered and publicly announced [34]. Consequently, selecting snapshots of our applications in 2021 will paint an incomplete picture, as most vulnerabilities recently introduced in the package's code could remain undisclosed for a median of 3 years. Since our collected applications contain their latest commits between Jan 2020 and May 2020, we chose to evaluate our applications as of May 1st 2016 (more than 3 years prior), which ensures that at least half the dependency vulnerabilities introduced in the applications are reported in the current npm advisories dataset.

Then, we answer our RQ in two steps. In the first step, we examine if the *selected snapshot* of the application had at least one dependency that contains a vulnerability (irrespective of its discoverability level). In the second step, we analyze only the applications containing at least one vulnerable dependency and use the methodology described in Section 5 to classify the discoverability levels of all vulnerable dependencies. In addition, since some applications have more than one vulnerable dependency, we further analyze the distribution of vulnerable dependencies in the applications under each discoverability level.

Results: As shown in Figure 3, we found that of the 6,546 studied applications **4,445 (67.90%) applications depend on at least one vulnerable dependency**. From the 4,445 affected applications, we break down the dependency vulnerabilities by the discoverability levels and evaluate how many applications contain one or more undisclosed, reported and public dependency vulnerabilities. We show this break down also in Figure 3. Note that the total percentage of undisclosed, reported and public surpasses 100%, as one application might contain dependency vulnerabilities on different discoverability levels. We observe that the majority of the affected applications, 4,419 (99.42%), depend on one or more dependency vulnerabilities that were undisclosed at the analyzed snapshot time.

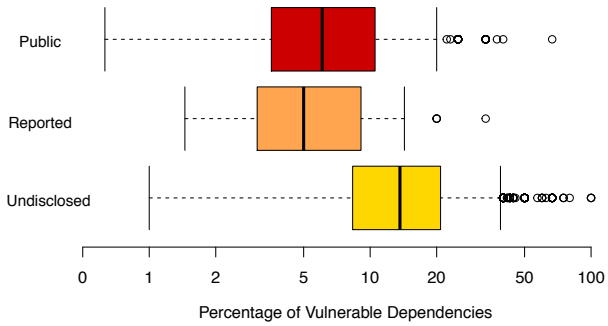

Fig. 4. Box-plots showing the distributions of the percentages of vulnerable dependencies in the applications, per discoverability level.

In fact, on 94.26% of the cases (4,190 applications), the applications were affected only by undisclosed vulnerabilities. Still, **206 (4.63%) applications depended on at least one package with a public vulnerability** and 45 (1.02%) applications depended on packages with a vulnerability reported to package maintainers.

Given that applications may have multiple vulnerable dependencies, we analyze proportion of vulnerable dependencies in each application. Figure 4 shows the distribution of the percentage of vulnerable dependencies per application in each discoverability level (public, reported, undisclosed). For instance, if an application has 10 dependencies, of which only 2 contained public vulnerabilities, this application would have 20% of its dependencies affected by public vulnerabilities.

In terms of **public** vulnerabilities, the 206 applications with at least one public dependency vulnerability had, on median, 6.25% of their dependencies affected by a public vulnerability, or, 1 out of 16 dependencies. The majority (80.1%) of the 206 applications depend on a single vulnerable dependency with a public vulnerability. For example, one of the applications affected by a public dependency vulnerability is the project *Atom*, a popular text editor, which has more than 40 dependencies but it was affected by a public vulnerability on a package called *marked* [3].

Upon closer inspection we found that, while the 206 applications depended on a total of 2,438 different packages, the public dependency vulnerabilities were found in only 17 packages. That is, the public dependency vulnerabilities occurred in less than 1% of total dependencies, but could, nevertheless represent the highest threat of exploitation on those applications. Table 3 shows a list of the 17 packages along with some meta-data to help us better understand the packages' use case, e.g., their domain/functionality, frequency, and popularity. From the Table, we can observe that the 17 packages are popular packages (have millions of weekly downloads) and quite common in most of the affected applications. For example, the vulnerable packages lodash and morgan are used in 80% of the affected projects. Such packages are common because they provide basic but essential functionalities that support projects from different domains. For instance, the lodash package provides methods for iterating arrays, objects, & strings. The morgan package simplifies the process of logging API requests in the application.

Such results may indicate that **most projects (regardless of the project type/domain) are subject to being affected by some popular vulnerable dependencies.** In other words, the packages with the most vulnerabilities affecting the applications are utility packages, which are used by many different types of applications. Therefore, application developers need to pay higher attention to specific packages 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 applications from being impacted. Interestingly, we notice that some of the 17 packages (e.g., node-uuid, request) have been deprecated, but are still used in some applications

Table 3. List of 17 packages with public dependency vulnerabilities, stating their domain, frequency, and popularity (# weekly downloads).

Vulnerable Package	Domain	% Affected Applications	# Downloads
lodash	Modular utilities.	82	40M
morgan	Request logger middleware.	79	2M
moment	Parse and manipulate dates.	60	15M
request	Simplified HTTP client.	54	16M
method-override	Override HTTP verbs such as PUT or DELETE.	50	472,927
mongoose	MongoDB object modeling tool.	45	1M
debug	Debugging utility.	34	148M
sequelize	ORM tool for databases, e.g., Postgres, MySQL.	28	1M
pg	PostgreSQL client for Node.js applications.	28	2M
mysql	Node.js driver for mysql.	24	689,626
jsonwebtoken	An implementation of JSON web tokens.	23	8M
superagent	Client-side HTTP request library.	17	5M
node-uuid	Used for the creation of RFC4122 UUIDs for distributed computing environment.	17	849,855
mime	A library for MIME type mapping.	16	41M
jquery	A library for DOM operations.	16	3M
jwt-simple	JWT (JSON Web Token) encode and decode module.	8	192,007
handlebars	A templating languages that keep the view and the code separated.	8	8M

with a vulnerable version. Application developers need to be careful about using such packages as they may be unmaintained and will no longer be updated by package maintainers.

Our previous results show that a set of popular dependencies are the main cause of affecting the projects with public discoverability vulnerabilities. However, it is still unclear the risk types associated with such vulnerabilities in the dependencies. Therefore, in this analysis, we investigate the types of dependency vulnerabilities that affect the projects. Such analysis is important to answer the following question: What are the most common types of vulnerabilities that affect the project dependencies? Each vulnerability report (in the npm advisories dataset) is associated with a Common Weakness Enumeration (CWE), aiming at categorizing vulnerabilities based on the explored software weaknesses (e.g. XSS, Buffer Overflow). We examine the frequency of vulnerability types to establish a profile of the vulnerabilities in the affected project. Doing so is important to understand the distribution of threat types in the projects.

To perform our analysis, we identify the vulnerability type associated with each vulnerable dependency that affects each project (at the latest snapshot). Then, we count the total number of affected projects by each type.

While we found that the projects are affected by 149 distinct CWEs, **5 vulnerability types (CWEs) are affecting the majority (77.98%) of the projects.** Table 4 lists the top five vulnerability types that affect most of the projects in our dataset. As we can see, 3 types of them (i.e., SQL Injection, Remote Memory Exposure, XSS) are among the top 10 security vulnerabilities as ranked by NVD [5]. However, 2 other types (i.e., Prototype Pollution and ReDoS) are not among the top-ranked type by NVD, yet they frequently affect the projects in our dataset,

Table 4. Ranking of the 5 most commonly found vulnerability types.

Vulnerability Type	# Affected Applications
Prototype Pollution	1,482
SQL Injection	974
Regular Expression Denial of Service (ReDoS)	771
Remote Memory Exposure	473
Cross-Site-Scripting (XSS)	466

which can expose the projects to a large threat in practice. For example, Prototype Pollution is the most common type, with 1,482 projects affected by the vulnerability. Also, ReDoS affects more than 771 projects in the dataset.

Upon a close investigation, we find that such common vulnerability types come from specific packages. For example, the package lodash is the reason for the projects being affected by the Prototype Pollution vulnerabilities. Similarly, only four packages are the source of ReDoS vulnerabilities (i.e., moment, method-override, debug, mime).

Interestingly, some recent research work has proposed techniques to effectively detect such vulnerability types (Prototype Pollution and ReDoS). For example, Li et al. [48] proposed a static taint analysis tool to detect prototype pollution vulnerabilities in Node.js packages. They found 61 previously-unknown vulnerabilities. Also, Davis et al. [32] studied the impact of ReDoS vulnerabilities in npm and PyPi and found that thousands of regexes are affecting over 10,000 modules across diverse application domains. Therefore, our results suggest that researchers should direct their efforts to improve practices and tools that tackle such vulnerability types, which would bring significant benefits to a wide range of software projects. Moreover, package maintainers are encouraged to widely adopt such research tools to constantly detect their vulnerabilities and fix them as soon as possible.

In terms of **reported** vulnerabilities, we can observe from Figure 4 that reported vulnerabilities are present in only 45 applications (1% of the affected applications). The median rate of dependencies with reported vulnerabilities in these 45 applications is 5.5% (1 out of 18 dependencies). It is notable that we find such a small share of applications that depend on reported dependency vulnerabilities. This is attributed to the npm policy for managing vulnerabilities: the policy states that the reported period of a vulnerability lasts at most 45 days, i.e., the vulnerability is published after 45 days of being reported to maintainers [14]. This limits how long a vulnerability can remain reported, thus, explaining the small occurrence of vulnerabilities at this stage.

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

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

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

Motivation: In RQ_1, we observe that a sizeable number of the affected applications (206 applications = 4.63%) depend on packages with public vulnerabilities. In such cases, the developers of the applications could know about the presence of the vulnerability in the affected dependency, and, hence, should avoid using that vulnerable version, if a fix is available. Much prior work (e.g., [45, 57])) focuses on studying whether application developers

Table 5. The percentage of vulnerabilities caused by the lack of available fix patch (Package-to-blame) vs. caused by the lack of dependencies update (Application-to-blame).

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

update their vulnerable dependencies or not. That said, they do not consider studying why the applications end up depending on package versions with public vulnerabilities. Given that public discoverability vulnerabilities are critical and have a high chance to be exploited from the point of adoption on, RQ_2 aims to investigate who is mostly responsible for the discoverability problem in order to understand how we can mitigate such issues. For example, if package maintainers are not providing a fix before the vulnerability publication time, this means that there is a need for designing a better disclosure process that gives maintainers more time to fix the vulnerability and release it to package users. Inspired by the git-blame command which is used to examine who is responsible for the modifications, we want to know who is responsible (to "blame") for not fixing vulnerable packages - the package maintainers for not providing a version that fixes a public vulnerability - or the application maintainers for not keeping their applications up-to-date.

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

- **Package-to-blame:** if at the analyzed snapshot, no safe version has been provided by the package maintainers for the public vulnerability. As the publication of a vulnerability comes after a period of 45 days, we consider the package maintainers the responsible for the dependency public vulnerability in applications.
- **Application-to-blame:** if there is already a released safe version of the vulnerable package but the application continues to rely on an (old) version with a public vulnerability. Application developers should monitor their dependencies and update to releases without public vulnerabilities, hence, we consider the application maintainers responsible for depending on a vulnerable package version.

Note that we check (for each dependency vulnerability) whether there was a fixed version of the vulnerable package at the time of analysis. Therefore, we had to do the checks at the vulnerability level, given that some projects were affected by more than one vulnerability.

Results: Table 5 shows the percentage of public vulnerabilities based on our responsibility analysis. We observe that **for public dependency vulnerabilities, the application is to blame in 90.76% of the cases**. That means that in 9 out of 10 cases the public vulnerability had an available fix, but developers did not update their application dependencies accordingly to receive the latest fix patch.

Therefore, and perhaps counter-intuitively, applications are not exposed to public vulnerabilities because packages have unfixed vulnerabilities. Instead, the real cause is the fact that application developers fail to keep up or at least to inform themselves well enough about a given dependency version. Hence, a major implication of our study is that application developers struggle with keeping their dependencies up-to-date, which may have serious effects in the security of their systems. In fact, there are some factors that play a role in deciding about the package update. For example, gaps in continuous integration (e.g., breaking changes and compatibility issues) can lead to ignoring the package update. Mirhosseini and Parnin studied the impact of using automated tools to update packages and found that such tools can lead to a higher rate of notification fatigue, issues in continuous integration, and tool design issues that can interfere with a developer's productivity [50]. Such factors indicate

that software projects might not address a vulnerability-related update, though developers could be aware of the update.

To have a better understanding of our results, we investigate how much effort would developers need to migrate to a safe version of their packages. npm adopts a semantic version scheme [61] where package maintainers are encouraged to specify the extent of their updates in three different levels: 1) patch release, which indicates backward compatible bug fixes, 2) minor release, which indicates backward compatible updates and 3) major release, which informs developers of backwards incompatible changes in the package release. Hence, patch and minor updates are deemed backwards compatible and may be performed at a lower migration cost, while major release updates incur on a high migration cost, as developers have to adapt their code to the new package API.

Once we take the update levels into consideration, we found that, **in 43.07% of the public vulnerabilities, the fix is only available in another major release of the package**. For instance, an application depends on P:1.0.0, and the fix patch was only released for a major version 2.0.0. Hence, to benefit from a fix patch in such a case, developers are required to adapt their code, imposing significant migration costs, especially for large projects that depend on dozens of packages. Furthermore, this shows that relying on automatic updates at the level of patch and minor releases (as recommended by npm [18]) does not completely prevent public vulnerabilities for affecting Node.js applications.

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

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

Motivation: Previous RQs show that a small but significant number of applications are exposed to public discoverability vulnerabilities, mostly due to a lack of dependency updates. In particular, the result of RQ2 shows that mostly the applications are to blame for being affected by a vulnerable package (since they do not update to a safer version at analysis time). Still, the picture might not be as bad as it seems because some applications could adopt the fixed version rapidly after adoption of a vulnerable dependency (within a few days). Hence, to better understand how bad the lagging situation is, RQ3 examines how long it takes for the application to fix dependency vulnerabilities with the public discoverability level. Once again, we focus on public discoverability vulnerabilities as they pose a higher chance of being exploited. Public discoverability vulnerabilities that affect the dependent application for a long time can leave an open channel for successful attacks, as shown in cases such as the Heartbleed incident [36]. Prior studies have examined the time to discover or fix vulnerabilities, e.g., the study by Decan et al. [34] examines the vulnerabilities at the package level in npm, investigating how long it takes to discover and publish new npm vulnerabilities. Our RQ_3 complements these studies by analyzing the risks of vulnerable dependencies in the Node.js applications, aggregating the vulnerability lifecycle through the discoverability level metric. Hence, we investigate how long applications remain dependent on a package version affected by a public vulnerability. Answering this question will give us insights into the prioritization of patching public dependency vulnerabilities that affect an application.

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

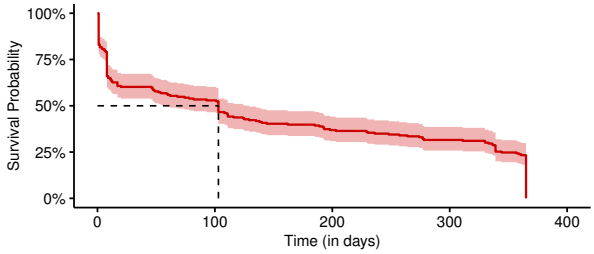

Fig. 5. Kaplan-Meier survival probability for affected applications with a publicly known vulnerability.

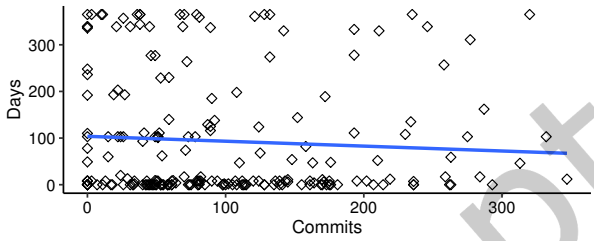

Fig. 6. Scatter plot showing the correlation analysis of number of commits vs. number of days.

June 2016 and then from 1st Sept. 2016 to 1st December 2016. In such cases, we sum all such periods (i.e., add up the number of days).

To present this analysis, we conduct a survival analysis method (a.k.a. event history analysis) [20]. The survival analysis is a non-parametric statistic method used to measure the survival function from lifetime data where the outcome variable is the "time until the occurrence of an event of interest". In the context of our study, we are interested in the time period that an application remains (survives) depending on a public dependency vulnerability. We use the non-parametric Kaplan-Meier estimator [44] to conduct the survival analysis, as used in previous studies [22, 33, 34].

Results: Figure 5 presents the survival probability for the applications depending on a public dependency vulnerability in the year of 2016. As we can observe, **half the applications remain exposed to a public dependency vulnerability for at least 103 days**. Such a long exposure of applications is concerning, as it gives a considerable time window for attackers' exploitation.

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

Figure 6 shows the scatter plot of both variables number of commits vs. number of days (affected by a public vulnerability). We draw a trendline in Figure 6 in order to study the relationship between the variables. We can observe that there is no clear pattern of the dots; indicating no correlation between the application activity and

Table 6. The share of applications with one or more (1+) public dependency vulnerabilities per severity levels.

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

the duration of which an application had been affected by a public dependency vulnerability (Pearson corr = -0.066).

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

7 DISCUSSION

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

7.1 Severity vs. Discoverability

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

The npm advisories associates each package vulnerability report with its severity level [52]. Severity level has four possible levels, Low, Medium, High, and Critical, which are assigned manually by the npm team. Vulnerabilities clasified as High or Critical are considered of high impact and need to be addressed immediately by software maintainers [52]. By cross-referencing our dataset with the severity reports, we report in Table 6 the distribution of the severity levels of the 206 application with public dependency vulnerabilities. Once again, the total percentage of Low, Moderate, High and Critical surpasses 100%, as some applications contain multiple dependency vulnerabilities on different severity levels. As shown in Table 6, of the 206 applications that are affected by public dependency vulnerabilities, 172 (83.49%) applications are affected by at least one vulnerable dependency of *low severity*. Still, a majority of 140 (67.96%) applications are affected by public vulnerabilities classified as high severity. In 59 (28.64%) applications, the public vulnerability was classified as critical, given their potential for exploitation. These results dismiss the idea that applications only depend on public dependency vulnerability with low impact of exploitability. More than 140 applications had dependencies with public vulnerabilities where analysts classified them as of high and critical impact, a dangerous combination for the health of those software projects.

7.2 Project evolution vs. Discoverability

Our study thus far has been conducted on one snapshot of the examined applications (RQ_1). However, our results might change if the study would be performed at different stages of a project's development cycle. We would like to determine whether our results generalize to different historical snapshots of the application development. Hence, we investigate the evolution of discoverability levels across different snapshots of applications' development.

Table 7. The percentage of vulnerable applications at different historical snapshots, per discoverability level.

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

Since each application has different lifespans, we want to find a measure that makes comparing them feasible. To do so, we normalize the applications by segmenting the lifetime of each application into five equal intervals (each containing 20% of an application's lifetime by time in days). Then, we perform the same anlaysis conducted in RQ_1 on the last snapshot of these five intervals. For this analysis, we only consider the 4,445 applications with at least one vulnerable dependency, as identified in RQ_1.

Table 7 shows the percentage of applications that have at least one vulnerable dependency for the 5 analyzed snapshots across their lifetime, along with the distribution of their discoverability levels. We recall that the snapshot of 100% represents applications analyzed on May 2016, the same snapshot analyzed in RQ1. Overall, we found that the proportion of applications with one vulnerable package remain steady between 64 to 67% of the analyzed applications. The major findings in RQ1 holds for all snapshots, there is a predominance of applications with undisclosed vulnerabilities, followed by a small share with public and even smaller share with reported dependency vulnerabilities. While the number of affected applications have increased as the applications evolve, it is noteworthy that the number of applications exposed to public vulnerabilities more than doubled since the snapshot 20% (from 101 to 206 applications). To conclude, our complementary analysis shows that the trends observed in RQ_1 hold at different stages of the projects.

8 TOOL SUPPORT: DEP-REVEAL

A major problem of vulnerabilities in software dependencies is the lack of developers awareness to security vulnerabilities in their dependencies [45]. Developers need better tools to help them identify the occurrence of vulnerabilities and how timely package maintainers respond to reported vulnerabilities, which affects the discoverability we studied in this paper. To address this problem, we build a tool called DepReveal, which uses the approach described in Section 5 to generate analytical reports of dependency discoverability levels for a GitHub Node.js project. DepReveal is open-source, publicly available, and can be easily integrated in any GitHub npm project.

The tool (DepReveal) works as a prototype or proof of concept that generates analytical reports of npm packages used in a GitHub Node.js project, which could have the potential to help developers in many ways, as follows:

- Modern applications rely on many dependencies and the number of dependencies is growing over time. Therefore, application developers struggle to track all these dependencies. In fact, developers should give more priority to dependencies that are frequently affected by vulnerabilities during the development lifetime. To help with this, our tool prototype can provide developers with the frequency in which certain dependencies have become vulnerable in the past, in order to grab the risks of depending on such packages and better plan their project maintenance in the future.

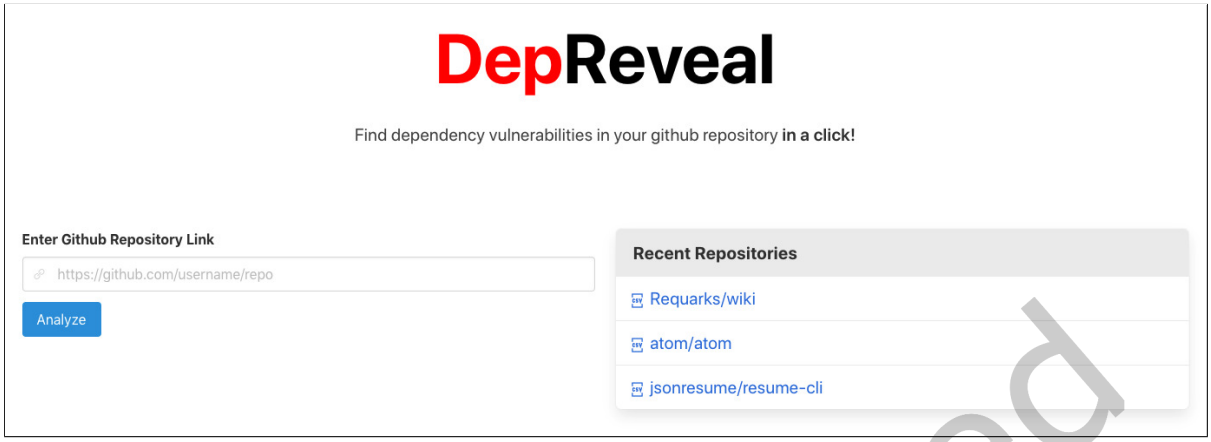

Fig. 7. Screenshot of the DepReveal website showing its interface and the recently analysed repositories.

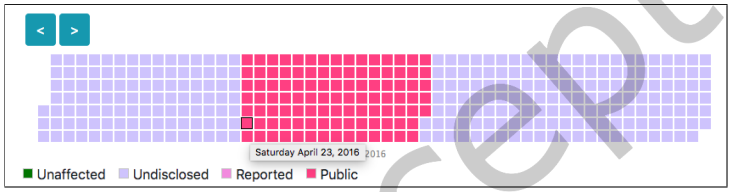

Fig. 8. Screenshot of the generated Dependency Discoverability Graph for the atom application using DepReveal.

- Moreover, the tool can provide developers with the history of all vulnerable dependencies of the application in order to understand the duration in which the application became at the risk of a public discoverability vulnerability in the past. Packages that do not update their code to address reported vulnerabilities incur a high risk for applications that use them and should be avoided by critical applications. We believe that such information is important for developers to build a more fine-grained picture of the risk of application dependencies, not only using automated tools (e.g., Dependabot) to update each and every package in the application.

Our tool generates 4 different reports to help developers understand: 1) the discoverability level of dependency vulnerabilities, 2) the frequency of dependency vulnerabilities per discoverability level, 3) the period of package exposition to discoverability levels, and 4) what package versions account for those dependency vulnerabilities. Figure 7 shows a screen-shot of the DepReveal's interface.

Next, we explain 2 of the most insightful reports generated by our tool. Inspired by the Github's Contributions Activity Graph, DepReveal generates a **Dependency Discoverability Graph**, which shows the historical exposure of the application to dependency vulnerabilities. We show in Figure 8 a screenshot of this report generated for the atom application [4]. Each cell represents a day in the history of the application during a year and the colors represent the discoverability level, with dark red indicating exposure to public vulnerabilities. In the example, it is easy to see that atom was exposed for 14 weeks to public dependency vulnerabilities in 2016, by seeing how many columns show the darker red color. Users can get more information about the date by hovering the mouse over the cell.

Period of Package Discoverability is another report generated by DepReveal to show the time period (in days) in which a vulnerable package affected the application, per discoverability level. Figure 9 shows a screenshot

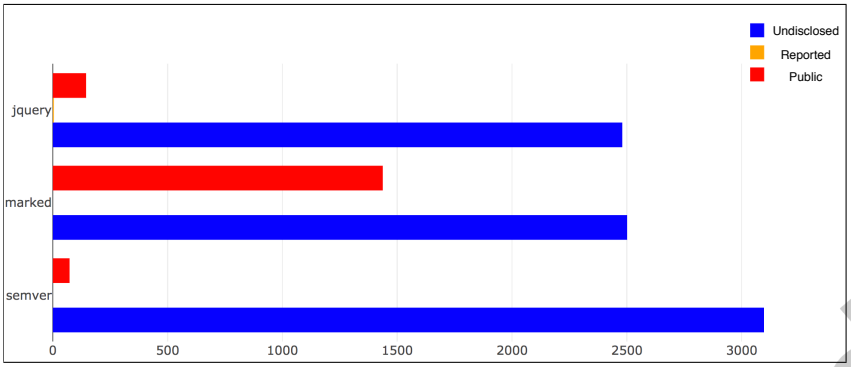

Fig. 9. Screenshot of the generated report Period of Discoverability for the atom application using DepReveal.

of this report, generated also for the atom application. From the Figure, we can observe, for example, that the package *jquery* was affected by a public, reported and undisclosed vulnerabilities through the project lifetime. Hovering the mouse over the tip of the red bar for the *jquery* package, it is possible to notice that the application remained depending on a public vulnerability in the jquery for 145 days through the entire application lifetime. Users can also enable/disable one of the discoverability levels by clicking on the legends at the right-side of the report plot.

Furthermore, the tool generates a CSV file that contains the analysis details for the entire application lifetime to help a further investigation. Finally, note that we provide a command-line version of the DepReveal, which is available from our open-source GitHub repository [9]. We also provide a web user-interface for the tool to facilitate using and interacting with it [19].

Comparison to Dependabot. Several dependency management tools have been proposed to help developers better track and update their outdated and vulnerable dependencies. For example, Dependabot is a bot the issues pull-requests (PRs) to help developers automatically update their vulnerable dependencies through PRs [23]. While related to dependency management, DepReveal and Dependabot have different goals:

- Dependabot's goal is to help developers update their dependencies. It submits pull requests to projects once their dependencies have new versions available, either to keep the dependencies up-to-date or to respond to a publicly known vulnerability.
- DepReveal, on the other hand, helps developers at analyzing the past exposure of their dependencies to public vulnerabilities, i.e., how often their dependencies have exposed their project to public discoverability vulnerabilities. Hence, our tool performs a post-mortem analysis and informs developers about how often their projects have been exposed to vulnerable dependencies in the past considering our discoverability levels.

9 IMPLICATIONS

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

9.1 Implications to researchers

Researchers should account for discoverability to better understand practices of security and dependency management. Discoverability is key to distinguish when vulnerabilities require immediate action from package and application maintainers. Undisclosed vulnerabilities are prevalent, present in the majority of studied applications (RQ1), and they tend to remain undisclosed for many years [22]. However, the presence of

undisclosed vulnerabilities does not denote lack of dependency maintenance from application maintainers or the lack of security maintenance from package maintainers. The way public vulnerabilities are handled by the community, on the other hand, shows a more accurate picture of the good and bad practices related to security and dependency management, as both package and application maintainers have to coordinate to reduce the risks of exploits. Researchers should include discoverability to better understand the practices related to dealing with dependency vulnerabilities, and can rely on the approach we propose in the study to implement this analysis. Our approach can also be applied in studies that aim to contrast our findings on different ecosystems (Python, Go, Java) to provide a more complete picture of the problem of vulnerable dependencies.

The lack of dependency updates remains is the main responsible for public exposure to vulnerabilities. Our results (RQ_2) show that in 9 out of 10 cases, the lack of dependency maintenance is the main reason applications are affected by public vulnerabilities. Furthermore, we found that it takes a long time to resolve to a fixed version (RQ_3). Such persistence to vulnerable dependencies indicates that developers do not perform a timely update to their dependencies. One reason for that is related to the potential risks of dependency updates (e.g., breaking changes) and the effort required to resolve them [45]. Thus, there is a need for approaches that provide developers with more confidence about the suggested dependency update. One line of work that needs to be explored further is to investigate techniques that automatically allow client code in the dependent application to catch up with the latest dependency updates. Unfortunately, the accuracy of existing works (e.g., [47, 64]) tends to be limited to a particular set of APIs (e.g., Android APIs). Also, researchers are encouraged to propose techniques that detect breaking changes in package updates, particularly those that are generalizable. Current works (e.g., [51]) are limited to specific ecosystems (e.g., Java, NodeJS) but still can be improved to work at scale and help developers detect breakages in practice. Such automated tools can motivate developers to perform a timely update of dependencies and catch security patches of their dependencies.

Certain vulnerability types are more frequently found in packages and can impact the ecosystem at large. Our results (RQ_1) show that certain types of dependency vulnerabilities that affect packages are prevalent across the affected projects. For example, we found that Prototype Pollution is the most common type, with 1,482 projects affected by the vulnerability. Also, ReDoS affects more than 770 projects in the dataset. This suggests a potential widespread benefit from research into the resolution or mitigation of such vulnerability types. Researchers should direct their effort to find effective techniques that discover such types. It will also be beneficial to conduct future research into common root causes of such vulnerability types to prevent or detect such issues in package code.

9.2 Implications to practitioners

Developers can use DepReveal to keep track of their vulnerability exposure in the past and better tailor management practices. The tool prototype we propose operationalize the Discoverability analysis and showcases to practitioners how often their dependencies have exposed their project with vulnerabilities. As discussed in Section 8, this information may help developers in two major ways. First, practitioners can prioritize maintenance tasks before release on dependencies that are more frequently flagged as been affected by vulnerabilities. That means, developers can actively monitor the project repository and its related security advisory database to identify when a vulnerability is published as soon as possible. Second, practitioners should reassess dependencies that expose them to public vulnerabilities without an immediate fix patch. While our results show that this is rare, publishing a vulnerability without a fix patch is a sign of inefficient security policies as it puts dependent projects at risk. Whenever possible, practitioners should select better alternative packages that prioritize submitting a fix patch before a vulnerability is made public.

Relying on SemVer does not guarantee that application projects receive all security updates in dependencies. Our results show that using semantic versioning for automatic updates at the level of patch and minor releases is not sufficient to prevent public vulnerabilities of dependencies in Node.js applications. Our manual inspection (RQ_2) revealed that in many cases the vulnerability fix is only available in a major release of the package, which comes at the cost of breaking changes. Tools such as Dependabot [40] help overcome this issue by helping developers migrate even when the fix is on another major version. Dependabot also includes a method to estimate the migration cost of security updates for dependencies, which is calculated based on the outcome of similar updates that were already done by other projects. Hence, developers that rely on SemVer (as recommended) should make use of other more robust mechanism, i.e., Dependabot, to verify if they are receiving the latest security fix patches as part of their automated dependency update policy.

10 RELATED WORK

The work most related to our study falls into security vulnerabilities in software ecosystems. In the following, we discuss the related work and reflect on how the work compares with ours.

10.1 Software Ecosystems

A plethora of recent work focused on software ecosystems. Several works compare different ecosystems. For example, Decan et al. [35] empirically compared the evolution of 7 popular package ecosystem using different aspects, e.g., growth, changeability, resuability, and fragility. They observed that the number of packages in those ecosystems is growing over time, showing their increasing importance.

Other work focused specifically on npm [39, 46, 66]. For example, Fard et al. [39] examined the evolution of dependencies within an npm project, and showed that there is a heavily interdependence, with the average number of dependencies being 6 and growing over time. Wittern et al. [66] investigated the evolution of npm using metrics such as dependencies between packages, download count, and usage count in JavaScript applications. They found that packages in the npm ecosystem are steadily growing. Such amounts of packages make the discovery of vulnerabilities much difficult, given the heavy dependence on such packages and the potential security problems in those packages.

Other studies pointed out the fragility of software ecosystems and provided insights on the challenges application developers face. For example, Bogart et al. [25, 26] examined the Eclipse, CRAN, and npm ecosystems, focusing on what practices cause API breakages. They found that a main reason for breaking changes are the updates of a dependency. This finding may explain why application developers are hesitant to update and explain why we see public vulnerabilities impacting applications that do not update in time. The authors extended the study by including a big survey on 18 ecosystems and repository mining methods [27]. They conducted a survey with more than 950 participants and report that Node.js is the ecosystem with higher frequency of breaking changes while Lua, Go are one of the least frequent.

Our study differs from the prior work since we focus on the discoverability levels of dependency vulnerabilities in Node.js applications. Moreover, we examine the reason that public dependency vulnerabilities exist. We also examine the time that an application stays depending on a public dependency vulnerabilities. That said, much of the aforementioned work motivated us to study npm and focus on examining vulnerabilities in application dependencies.

10.2 Package Vulnerabilities

Vulnerabilities in ecosystems have been studied broadly [21, 22, 45, 57]. For example, Kula et al. [45] explored how developers respond to security awareness mechanisms such as library migration, and found that developers were unaware of most vulnerabilities in dependencies. Pashchenko et al. [57] indicated (based on interviews with developers) a high demand for high-level metrics to assess the maintainability and security of software packages. Our proposed tool DepReveal partially fulfils such a demand since it generates analytical reports to inform

developers how vulnerable their dependences are, considering the discoverability levels. Enck and Williams [37] proposed the top five challenges in software supply chain security. For example, one of the study participants mentioned the challenge of being the first or last to update a dependency. Participants in the study mentioned that there is a need to develop a policy that strikes this balance.

More specifically, several studies focused on analyzing the impact of security vulnerabilities in the npm ecosystem [29, 34, 70]. Decan et al. [34] found that npm vulnerabilities take more than 2 years to be discovered. Zimmermann et al. [70] analysed the maintainers role for npm vulnerable packages, and found that a small number of maintainers' accounts could be used to inject malicious code into thousands of npm dependent packages, a problem that has been increasing over time. Zerouali et al. [69] studied npm vulnerable packages in Docker containers, and found that they are common in the containers, suggesting that Docker containers should keep their npm dependencies updated. Bodin et al. [28] 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. Zahan et al. [67] defined some signals that could indicate malicious npm package, such as the presence of install scripts. Their study shows that 2,818 maintainer accounts associated with an expired domain, allowing an attacker to hijack 8,494 packages by taking over the npm accounts. Similar to npm packages, Wang et al. [65] found that Java packages contained dependencies which lag for a long time and never been updated. Our study complements previous studies by analysing npm vulnerable dependencies throughout the Node.js application lifetime, aggregating the vulnerability lifecycle through the discoverability level metric.

Other studies perform a code-based analysis to assess the danger of dependency vulnerabilities [56, 58, 60, 68]. A study by Zapata et al. [68] manually analysed 60 projects that depend on vulnerable npm packages, and found that 73.3% of them were actually safe because they did not make use of the vulnerable functionality, showing that there is an overestimation on previous reports. Our study includes another aspect that impacts vulnerable dependencies in applications, by including the discoverability levels.

There were several efforts to assess the impact of vulnerable dependencies in dependent applications [58, 60]. Plate et al. [58, 60] proposed a code-centric tool that determines whether or not a Java application executes the fragment of the dependency where the vulnerable code is located. Their proposed approach is implemented in a tool called, Eclipse Steady (aka VULAS), which is an official software used by SAP to scan its Java code. Furthermore, Ponta et al. [59, 60] built upon their previous approach in [58] to generalize their vulnerability detection approach by using static and dynamic analysis to determine whether the vulnerable code in the library is reachable through the application call paths. Bodin et al. [29] implemented an extension of the Eclipse-Steady tool to support JavaScript. They analysed analysed 42 applications to find their vulnerable constructs, showing that a code-centric approach is viable, although there are challenges given the dynamic nature of the JavaScript and the complexity of the npm dependencies [29]. Our tool (DepReveal) complements these tools by looking at vulnerable dependencies through the history of a Node.js application. DepReveal aims to increase developers awareness on how often their application project is exposed to vulnerable dependencies.

Our tool could be extended to include a code-centric analysis and report the vulnerable constructs per discoverability analysis. However, the analysis at this level is indeed problematic due to execution costs needed to analyse the code. Other automated code analysis tools work on vetting the changes in the releases of packages to analyse their lines of code. Recently, there were several efforts for auditing npm security vulnerabilities, both from academia [41, 63] and from industry practitioners [6, 8].

11 THREATS TO VALIDITY

Internal Validity considers the relationship between theory and observation. Our dataset contains 925 vulnerability report available in the npm advisories dataset. There might be other vulnerable packages that have

been discovered but not yet reported. However, we leveraged up-to-date dataset from npm advisories, which we believe contains the most accurate information about the vulnerable packages reported to them.

The paper only considered direct dependencies. Direct dependencies are managed (directly) by the software project, while indirect dependencies are usually out of the control of the application developer, as they are dependencies of a dependency, which makes it more challenging for updating them. Moreover, vulnerabilities of direct dependencies are more likely to impact the software project, as they are directly used in the project codebase. Our technique can be extended to analyse indirect dependencies considering the discoverability levels.

We did not consider whether the vulnerable functionality in the package actually affects the application, i.e., whether the applications use the vulnerable code of the package. Considering this would be challenging, since our dataset is composed of thousands of applications. That said, our analysis is in line with prior work in the area of software ecosystems, which also examine dependencies in the package.json file to associate packages to applications.

Also, our paper did not consider the stage at which a fix was released by package maintainers or adopted by the studied application. In fact, this has been studied in prior work [22] at the package level, e.g., the paper [22] examines the stage at which a fixed version was released. Still, a similar analysis at the application level could be considered in future work, however, we found only nine vulnerabilities (affecting nine applications) that were published with a fix available from day 1. That is, the vast majority of vulnerabilities are made public without any fix. Unfortunately, the amount of vulnerabilities that had a fix available from day 1 is too low to derive any meaningful comparison or insights to the community.

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

External Validity is related to the generalizability of our findings. Our study is based on Node.js applications that use npm. Hence our results may not generalize to applications written in other languages. However, the key concepts and design of our study can be applied to other package dependency networks to expand the investigation on vulnerable dependencies. Our dataset contains 6,546 JavaScript applications that use npm packages. Our dataset might be considered small when it is compared to the whole population of JavaScript applications. However, our dataset is of high quality, since we filtered out applications that are immature and have less development history, by using the filtering criteria used by Kalliamvakou et al. [43]. Also, to our knowledge, our dataset is considered to be among the largest number of Node.js applications analyzed.

12 CONCLUSION

Our study examines vulnerable dependencies in Node.js applications based on their discoverability lifecycle. First, we define three discoverability levels for dependency vulnerabilities in Node.js applications. Then, we perform an empirical study on 6,546 Node.js applications to assess how discoverable vulnerable dependencies are. Our findings show that 67.9% of the examined applications depend on at least one vulnerable package. 99.42% of the affected applications depend on undisclosed dependency vulnerabilities. Still, 206 (4.63%) applications were still affected by a public discoverability vulnerability, and they often remain affected for a substantially long time (103 days) during the application lifetime. Moreover, we examined why these applications end up depending on public dependency vulnerabilities. We observed that the application developers are mostly to blame, i.e., a fix for the vulnerable dependency is available but not patched in the application. Furthermore, we examine the relationship between the occurrence of public discoverability vulnerabilities and the underlying project factors. We find that such metrics do not strongly indicate better handling of the vulnerable dependencies.

Our findings imply that accounting for discoverability analysis can help researchers better understand practices of security and dependency management. Also, researchers are encouraged to explore approaches that consider confidence measures of dependency updates to help developers catch up with their critical vulnerable

dependencies. We developed a tool prototype that supports our analysis approach for npm projects, which have the potential to help developers better understand and characterize package vulnerabilities that affect their applications. Finally, our result of the relationship between project dependencies and the number of dependency vulnerabilities shows that managing specific packages in the project will be more effective than reducing the number of project dependencies. Project maintainers can try to reduce relying on some specific popular packages by prioritizing updates of those packages or replacing them with a single package that covers their functionalities and has an active security track record.

Our paper outlines directions for future work. For example, our discoverability analysis can be extended to consider other data sources to enhance vulnerability risk assessment, e.g., severity, exploitability, etc. Such enhancement can also be integrated into our prototype tool to help developers better analyze the risk of security vulnerabilities that affect their dependencies. In the future, we plan to evaluate the usefulness of our prototype tool (i.e., DepReveal). Finally, we plan to examine if our findings hold for applications written in different programming languages (e.g., Python and Java).

REFERENCES

[1] [n.d.]. About coordinated disclosure of security vulnerabilities - GitHub Docs. https://docs.github.com/en/code-security/security-advisories/about-coordinated-disclosure-of-security-vulnerabilities#about-disclosing-vulnerabilities-in-the-industry. (Accessed on 07/17/2021).

[2] [n.d.]. About coordinated disclosure of security vulnerabilities - GitHub Docs. https://docs.github.com/en/code-security/repository-security-advisories/about-coordinated-disclosure-of-security-vulnerabilities. (Accessed on 04/09/2022).

[3] [n.d.]. Add package marked@0.3.4. https://github.com/atom/atom/commit/41b799aebc7f40201219d8ec435d1520cf057285. (Accessed on 07/17/2021).

[4] [n.d.]. atom/atom: The hackable text editor. https://github.com/atom/atom. (Accessed on 07/17/2021).

[5] [n.d.]. CWE - 2022 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html. (Accessed on 07/18/2022).

[6] n.d.. Fast static analysis for the DevSecOps workflow — by r2c. https://r2c.dev/. (Accessed on 07/17/2021).

[7] n.d.. Heartbleed Bug. https://heartbleed.com/#:~:text=The%20Heartbleed%20Bug%20is%20a,used%20to%20secure%20the%20Internet.. (Accessed on 07/17/2021).

[8] [n.d.]. How to ensure JavaScript code quality | DeepScan. https://deepscan.io/. (Accessed on 07/17/2021).

[9] [n.d.]. mahmoud-alfadel/DepReveal: A lightweight tool to analyze npm vulnerabilities in the history of GitHub repositories. https://github.com/mahmoud-alfadel/DepReveal. (Accessed on 07/17/2021).

[10] [n.d.]. npm. https://www.npmjs.com/advisories/33. (Accessed on 07/17/2021).

[11] [n.d.]. npm - Libraries.io. https://libraries.io/NPM. (Accessed on 01/24/2022).

[12] [n.d.]. npm - Libraries.io. https://libraries.io/NPM. (Accessed on 07/17/2021).

[13] [n.d.]. npm Blog Archive: A Day in the Life of npm Security. https://blog.npmjs.org/post/190665497245/a-day-in-the-life-of-npm-security.html. (Accessed on 07/17/2021).

[14] [n.d.]. npm Blog Archive: AppSec POV on Dependency Management. https://blog.npmjs.org/post/187496869845/appsec-pov-on-dependency-management. (Accessed on 07/17/2021).

[15] [n.d.]. npm/node-semver: The semver parser for node (the one npm uses). https://github.com/npm/node-semver. (Accessed on 04/02/2022).

[16] [n.d.]. On the Discoverability of npm Vulnerabilities in Node.js Projects | Zenodo. https://zenodo.org/record/5153319#.YQfhBFP0nUI. (Accessed on 08/02/2021).

[17] [n.d.]. Semantic Versioning 2.0.0 | Semantic Versioning. https://semver.org/. (Accessed on 07/17/2021).

[18] [n.d.]. Semantic Versioning 2.0.0 | Semantic Versioning. https://semver.org/. (Accessed on07/17/2021).

[19] 2021. DepReveal. https://bit.ly/3emg5w3. (Accessed on 07/17/2021).

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

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

[22] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2021. Empirical Analysis of Security Vulnerabilities in Python Packages. In *Proceedings of the 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 446–457.

[23] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Mouafak Mkhallalati. 2021. On the Use of Dependabot Security Pull Requests. *Proceedings of the 2021 International Conference on Mining Software Repositories (MSR '21)* (2021).
[24] Victor R Basili, Lionel C Briand, and Walcélio L Melo. 1996. How reuse influences productivity in object-oriented systems. *Commun. ACM* 39, 10 (1996), 104–116.
[25] Christopher Bogart, Christian Kästner, and James Herbsleb. 2015. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), 2015*. IEEE, 86–89.
[26] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 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, 109–120.
[27] Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2021. When and How to Make Breaking Changes: Policies and Practices in 18 Open Source Software Ecosystems. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 4 (2021), 1–56.
[28] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. 2021. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering* 26, 3 (2021), 1–28.
[29] Bodin Chinthanet, Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. 2020. Code-based Vulnerability Detection in Node. js Applications: How far are we?. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1199–1203.
[30] Md Atique Reza Chowdhury, Rabe Abdalkareem, Emad Shihab, and Bram Adams. 2021. On the Untriviality of Trivial Packages: An Empirical Study of npm JavaScript Packages. *IEEE Transactions on Software Engineering* (2021).
[31] Filipe Roseiro Cogo, Gustavo Ansaldi Oliva, and Ahmed E Hassan. 2019. An empirical study of dependency downgrades in the npm ecosystem. *IEEE Transactions on Software Engineering* (2019).
[32] James C Davis, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 246–256.
[33] Alexandre Decan and Tom Mens. 2019. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering* (2019).
[34] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *International Conference on Mining Software Repositories*.
[35] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2018. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* (2018), 1–36.
[36] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. 2014. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*. 475–488.
[37] William Enck and Laurie Williams. 2022. Top Five Challenges in Software Supply Chain Security: Observations From 30 Industry and Government Organizations. *IEEE Security & Privacy* 20, 2 (2022), 96–100.
[38] Equifax. 2017. Equifax Releases Details on Cybersecurity Incident, Announces Personnel Changes. https://investor.equifax.com/news-and-events/news/2017/09-15-2017-224018832 Accessed on 07/17/2021.
[39] Amin Milani Fard and Ali Mesbah. 2017. JavaScript: The (un) covered parts. In *IEEE International Conference on Software Testing, Verification and Validation (ICST), 2017*. IEEE, 230–240.
[40] Github. [n.d.]. Dependabot. https://dependabot.com/. (Accessed on 07/17/2021).
[41] Liang Gong. 2018. *Dynamic Analysis for JavaScript Code*. Ph. D. Dissertation. UC Berkeley.
[42] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (San Francisco, CA, USA) *(MSR '13)*. IEEE Press, Piscataway, NJ, USA, 233–236.
[43] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proceedings of the 11th working conference on mining software repositories*. ACM, 92–101.
[44] Edward L Kaplan and Paul Meier. 1958. Nonparametric estimation from incomplete observations. *Journal of the American statistical association* 53, 282 (1958), 457–481.
[45] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (2018), 384–417.
[46] Raula Gaikovina Kula, Ali Ouni, Daniel M German, and Katsuro Inoue. 2017. On the Impact of Micro-Packages: An Empirical Study of the npm JavaScript Ecosystem. *arXiv preprint arXiv:1709.04638* (2017).
[47] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. 2020. A3: Assisting android api migrations using code examples. *IEEE Transactions on Software Engineering* (2020).
[48] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2021. Detecting node. js prototype pollution vulnerabilities via object lookup analysis. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations*

of Software Engineering. 268–279.
[49] Wayne C Lim. 1994. Effects of reuse on quality, productivity, and economics. *IEEE software* 5 (1994), 23–30.
[50] Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 84–94.
[51] Anders Møller and Martin Toldam Torp. 2019. Model-based testing of breaking changes in Node. js libraries. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 409–419.
[52] npm. [n.d.]. About audit reports | npm Docs. https://bit.ly/3uqn0uv. (Accessed on 07/17/2021).
[53] npm. [n.d.]. npm advisories. https://www.npmjs.com/advisories. (Accessed on 07/17/2021).
[54] npm. [n.d.]. npm registry. https://docs.npmjs.com/misc/registry. (Accessed on 07/17/2021).
[55] OWASP. [n.d.]. OWASP Risk Assessment Framework. https://owasp.org/www-project-risk-assessment-framework/. (Accessed on 07/17/2021).
[56] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.
[57] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. 2020. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1513–1531.
[58] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. 2015. Impact assessment for vulnerabilities in open-source software libraries. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 411–420.
[59] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 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, 449–460.
[60] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2020. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering* 25, 5 (2020), 3175–3215.
[61] semver. [n.d.]. semver - npm. https://www.npmjs.com/package/semver. (Accessed on 07/17/2021).
[62] SOF. [n.d.]. Stack Overflow Developer Survey 2019. https://insights.stackoverflow.com/survey/2019. (Accessed on 07/17/2021).
[63] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2018. SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE. JS.. In *NDSS*.
[64] Ferdian Thung, Stefanus A Haryono, Lucas Serrano, Gilles Muller, Julia Lawall, David Lo, and Lingxiao Jiang. 2020. Automated deprecated-api usage update for android apps: How far are we?. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 602–611.
[65] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 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, 35–45.
[66] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A look at the dynamics of the JavaScript package ecosystem. In *IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), 2016*. IEEE, 351–361.
[67] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. 2022. What are weak links in the NPM supply chain?. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 331–340.
[68] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. 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, 559–563.
[69] Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles, and Jesus M Gonzalez-Barahona. 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, 619–623.
[70] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 995–1010.