Using Package Manager and Repository Metrics to Determine the Security of NPM Packages.

by

Carly Pereira

A Major Qualifying Project

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

in

Computer Science

by

____

May 2021

APPROVED:

____________________________________ Professor Lorenzo DeCarli

Abstract

Package managers are tools used to find, install, maintain, and uninstall software packages. Anyone can publish packages to package managers, allowing developers to install and use their code. While this is a revolutionary innovation for programmers, it is also the perfect platform to enable threat actors to execute attacks. However, malicious code is not the only threat that comes from downloading packages. It is also possible that uploaded packages do not employ secure coding techniques and therefore contain security vulnerabilities. If a developer were to download and use an unknowingly vulnerable package in their project, this would make their project vulnerable to attacks. Currently there are no tools available to determine the likelihood that a package may contain an unknown vulnerability before downloading it. Therefore, the goal of this project was to determine whether there are any package or repository metrics that reliably correlate with the security of packages. We explored this idea specifically with packages from Node Package Manager (NPM), an online repository for publishing open-source Node.js projects. The metrics of NPM packages that we explored are number of monthly downloads, number of dependents, number of open issues, number of closed issues, and each of these were compared to the number of known vulnerabilities. The data for this project was sourced from package libraries, the NPM website, GitHub's website, and the Snyk known vulnerability database. This data was then analyzed, and the metrics were found to have a very weak correlation to known vulnerabilities. Future work and testing is necessary to determine whether these metrics do correlate to security for certain.

Contents

1 Introduction **1**

1.1 Approach 4

1.2 Obstacles 4

1.3 Results 5

2 Background **6**

2.1 Packages and Package Managers 6

2.2 GitHub 7

2.2.1 Issues 7

2.3 Security Concerns 8

2.4 Motivation 8

3 Related Works **9**

4 Methodology **11**

4.1 Metrics 11

4.2 Popularity Baseline 13

4.3 Gathering the Data 14

4.4 Data Analysis 17

5 Results and Discussion **19**
5.1 Monthly Downloads . . . 21
5.2 Dependents . . . 22
5.3 Open Issues . . . 24
5.4 Closed Issues . . . 25
5.5 Data Restrictions and Limitations . . . 26
6 Future Work **27**
6.1 Package Manager and Repository Metrics . . . 27
6.1.1 Open and Closed Issues . . . 27
6.1.2 Number of Dependencies . . . 28
6.2 Software Metrics . . . 28
6.3 Metric Manipulation . . . 29
6.4 Version Testing . . . 29
6.5 Vulnerability Score . . . 30
7 Conclusion **31**

List of Figures

5.1 The relationship between the number of monthly downloads and the number of known vulnerabilities for NPM packages. 21

5.2 The relationship between the number of dependents and the number of known vulnerabilities for NPM packages. 22

5.3 The relationship between the number of open issues and the number of known vulnerabilities for NPM packages. 24

5.4 The relationship between the number of closed issues and the number of known vulnerabilities for NPM packages. 25

Chapter 1

Introduction

Package managers are tools used to find, install, maintain, and uninstall software packages. Anyone can publish packages to package managers, allowing developers to install and use their code. While this is a revolutionary innovation for programmers, it is also the perfect platform to enable threat actors to execute attacks. There are many types of attacks that threat actors can utilize package managers to perform, such as general supply chain attacks [10], dependency confusion [7], and typosquatting [11].

With supply chain attacks the goal is to access source codes, build processes, or update mechanisms by infecting legitimate apps to distribute malware. Attackers hunt for unsecure network protocols, unprotected server infrastructures, and unsafe coding practices. They break in, change source codes, and hide malware in build and update processes. Because software is built and released by trusted vendors, these apps and updates are signed and certified. In software supply chain attacks, vendors are likely unaware that their apps or updates are infected with malicious code when they're released to the public. The malicious code then runs with the same trust and permissions as the app [14].

Package managers are a tool that threat actors have been able to exploit to perform supply chain attacks, such as dependency confusion. Dependency confusion is a relatively new type of supply chain attack. It takes advantage of internal packages, i.e. packages that are used within an organization and not publicly shared. Internal package.json files become embedded in public script files during their build process, and therefore leak the names of these internal packages. These names are then used by threat actors as names for their malicious packages, and are uploaded to public package managers. Then, the malicious package would accidentally be downloaded through public package managers and infiltrate large organizations. Dependency confusion was detected inside more than 35 organizations to date, across three tested programming languages. The vast majority of the affected companies fall into the 1000+ employees category, and almost 75% of all the logged callbacks came from NPM packages [2].

Typosquatting was initially thought of only as domain typosquatting or URL hijacking, where an actor would register a domain under a name very similar to that of a more popular site (i.e. using goggle.com to typosquatt google.com). It relies on mistakes such as typos or translation errors made by Internet users to direct traffic to the hijacked URL. The purpose of this could have been to drive traffic away from competitors, to make money through advertisements on the site, or even steal information from unsuspecting users through fake login portals [24]. However, this issue of typosquatting also appears in package managers. Packages are often downloaded by users through the terminal by typing commands such as `npm install is-even`. If a user misspells a package name, but the misspelling is an existing package on the package manager, then it will still be installed. This leaves room for threat actors to register malicious packages under names similar to those of very popular packages, and rely on the fact that users may misspell the

intended package's name and download the malicious package by accident.

In 2018 alone, research indicated more than 100 malicious packages had more than a cumulative 600 million downloads [9]. A study by Symantec showed that supply chain attacks increased by 78% in 2019 [25]. With malicious attacks through package managers increasing at an alarming rate, it is important to take countermeasures to protect against them. There are some tools used to detect and mitigate these attacks, including NPM Audit and Spellbound. NPM Audit is a security feature that is built into NPM. It checks the current version of the installed packages in a project (aka, dependencies) against known vulnerabilities reported on the public NPM registry. If it discovers a security issue, it reports it. Notably, the report contains the level of severity of the identified vulnerability. The extent of severity is determined by the impact and exploitability of the issue [16]. Spellbound is used when downloading a package, and warns the user if the package they are downloading is likely typosquatting a similar, more popular package. This is accomplished by defining string transformation patterns, such as repeated characters, omitted characters, swapped characters, common typos, etc. If the given package name matches at least one package from a list of popular packages after a set of allowed transformations, then it is considered to be a typosquatting suspect and the user is alerted [26].

However, intentionally malicious code is not the only threat that comes from downloading packages. It is also possible that uploaded packages do not employ secure coding techniques and therefore contain security vulnerabilities. If a developer were to download and use an unknowingly vulnerable package in their project, this would make their project vulnerable to attacks. Currently there are no tools available to determine the likelihood that a package may contain an unknown vulnerability before downloading it. Therefore, the goal is to determine whether there

are any package or repository metrics that reliably correlate with the security of packages.

1.1 Approach

The approach taken when evaluating this idea was to determine what package and repository metrics are available and common across all packages, and to interpret what their values could represent in relation to security. These metrics then needed to be gathered from a package manager and code repository using an automated program. Information about the number of known vulnerabilities per package would also needed to be identified and collected.

1.2 Obstacles

There were many roadblocks when creating the program to gather the metrics, such as `GET` request timeouts when trying to use the GitHub API to get metric information. We had initially thought that using an API to do some of the work for us would be the most time efficient method of gathering repository information. However, the GitHub API limits requests to 5,000 per hour and so it was not feasible to continue using this for data collection. Initially the program took many hours to run due to the number of packages that were being used and the number of requests that had to be made per package. After employing multi-threading techniques and scraping the GitHub repository pages instead of using the GitHub API, the program was able to run much more quickly and avoided `GET` request timeouts.

1.3 Results

After the data was collected, correlation to known vulnerabilities and other general statistics needed to be calculated. The Pearson Correlation Coefficient (PCC) was calculated to measure the linear correlation, r, between each of the sets of data, with x representing a package metric and y as the number of known vulnerabilities. This coefficient is known as the best method of measuring the association between variables of interest because it is based on the method of covariance, which measures the total variation of two random variables from their expected values. Once the PCC was calculated it was interpreted with the standard thresholds for correlation. Metrics averages were also calculated, as well as the maximum, minimum, and standard deviation of each metric. This was done to gain insight and compare packages with known vulnerabilities to those that have no known vulnerabilities.

Chapter 2

Background

2.1 Packages and Package Managers

A software package is a file or files bundled together that perform dedicated functions. Packages can be created by anyone and uploaded to package managers. Software packages can be used to perform simple calculations such as `is-even`, which takes an input and determines if it is even or not, or for providing complex libraries, such as `react` to manage DOM elements. Developers can download and include packages in their projects to add functionality without having to re-create it themselves, saving valuable time. Using packages also allows developers to offload code maintenance to the package's owners. Packages managers automate the installation process, upgrades, maintenance, and uninstallation process for packages. They also collect and display download data, dependency and dependent data, as well as version information, a corresponding GitHub repository URL, and collaborator information. We decided to work with the Node Package Manager (NPM) because we have previously worked with packages from this manager, and were most familiar with it.

2.2 GitHub

GitHub is a code hosting platform for version control and collaboration amongst developers [12]. Most packages are open-source, meaning their source code is available to the public, and is usually published to a GitHub repository or a similar service. Members of the open source community can suggest changes to the code through pull requests, which can be accepted or denied by the maintainers of the repository. For repository metrics, we chose to search for the package's repositories on GitHub. GitHub is the largest collaborative version control platform in the world, reporting having over 40 million users and more than 190 million repositories as of 2020 [12]. This makes GitHub the ideal candidate for gathering reliable and up-to-date metrics, as it is likely that a package will be hosted using this service. In fact, of the roughly 16,000 packages used for this project only about 3.5% had no GitHub repository information.

2.2.1 Issues

GitHub issues are suggestions to the code base, but do not supply the necessary code to make the change. They are often bug reports or suggestions for improvement. Issues are marked as open until they are resolved with a pull request or manually closed, marking them as closed. It was our belief that GitHub issues could be used as a metric to determine the security of a package due to the nature of why issues are submitted and what they represent. Issues can be used by the community to report potential vulnerabilities, and therefore a greater number of issues could represent a larger number of vulnerabilities being reported.

2.3 Security Concerns

The benefits of using packages are very clear, however, there are some downfalls when considering security. If the developer of a package does not employ secure coding practices, then their software could contain vulnerabilities. Threat actors could also take advantage of this, and include malicious or vulnerable code into a package. Unfortunately, discovering vulnerabilities in code is not an easy task. Most of the time, vulnerabilities remain undisclosed until they are exposed, for instance, by an attack during the software operational phase. There are many studies on the use of software and/or program metrics for determining if a program contains potentially vulnerable code, and some have been able to reliably do this [6] [8] [13] [22]. However, these methods are language specific and are mostly designed to be used in the development process of the program.

2.4 Motivation

The goal of this project is to determine if there are any package or repository metrics that correlate with the security of a package, providing a foundation for a larger project to develop vulnerability scanning tools for end users to use when downloading packages from package managers. Since package and repository metrics are not language specific, such a tool could be used with any language that has a package manager. We explored this idea specifically with packages from Node Package Manager (NPM), an online repository for publishing open-source Node.js projects [15].

Chapter 3

Related Works

Software Ecosystem Security: Past research has focused on the security of package managers themselves, performed and tested attacks against them, and have recommendations for building secure package managers [1] [4]. These works discuss attacks directly to package managers and the consequences of these attacks, but do not discuss how package managers could be used to perform attacks on others.

Discovering Vulnerabilities in Dependencies: The Snyk vulnerability scanner is an open source tool for developers. It scans open source dependencies for known vulnerabilities including SQL injections, cross-site scripting (XSS), insecure direct object references (IDOR), cross-site request forgery (CSRF), and security misconfiguration [21]. NPM Audit is a similar tool for reporting known vulnerabilities in dependencies [16]. In fact, there are other open source tools available that perform this task [3] [17] [18]. However, these tools do not provide information about the potential unknown vulnerabilities within dependencies or packages.

Using Software Metrics to Determine Security of Packages: In past research software metrics have been used to determine whether code may be vulnerable. Most of the research focused on creating vulnerability prediction models using complexity

metrics. According to their findings, traditional metrics including code churn, complexity, and fault history exhibit similar performance in vulnerability prediction as they exhibit in fault prediction models [19] [20] [5]. Recent research has tested software metrics at different granularity levels to predict vulnerable code components (i.e., vulnerable classes and methods), and have found success with this method [22] [23] [13]. This research has shown promising results using software metrics to discover potentially vulnerable code, however it is often language specific. There is no universal method to calculate software metrics, and not all metrics are appropriate for all languages (ex. metrics related to Objects are not relevant to non object oriented programming languages such as C). This was a large motivation of using package and repository metrics as indicators of vulnerability, because they are universal.

Chapter 4

Methodology

4.1 Metrics

To discover what metrics could reliably determine the security of an NPM package, we first had to decide what metrics to calculate and why they might correlate with security. The metrics of NPM packages that we explored are number of monthly downloads, number of dependents, number of open issues, number of closed issues, and each of these were compared to the number of known vulnerabilities. These metrics were chosen because almost all packages have data for them on NPM and GitHub, with only 3.5% of packages used missing GitHub information, allowing for a large data set to be analyzed. We used the Pearson Correlation Coefficient (PCC) to measure the linear correlation, r, between each of the sets of data, with x representing a package metric and y as the number of known vulnerabilities. This coefficient is known as the best method of measuring the association between variables of interest because it is based on the method of covariance, which measures the total variation of two random variables from their expected values. The PCC ranges from -1 to +1, with a positive coefficient meaning that as x increases, y increases and a

negative coefficient meaning that as x increases, y decreases. Correlation coefficient absolute values below 0.3 are considered to be very weak; 0.3-0.5 are weak; 0.5-0.7 are moderate; >0.7 are strong.

Pearson Correlation Coefficient (r)

$$r = \frac{\sum(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum(x_i - \overline{x})^2 \sum(y_i - \overline{y})^2}}$$

It is important to calculate and interpret this coefficient to determine whether these package metrics are related to known vulnerabilities. We hypothesize that more downloads or dependents a package has, the less likely it is to have a known vulnerability. The reasoning for this is that more downloads/dependents means more users rely on that package to be secure, and would likely have noticed a vulnerability if one existed. This hypothesis can be tested by calculating the Pearson coefficient between downloads and known vulnerabilities; in this case the hypothesis would be proven by a negative coefficient, showing that as downloads increase known vulnerabilities decrease. We also hypothesize that if a package has many open issues, it may be more vulnerable. Likewise, if a package has many closed issues, it may be more secure. GitHub issues are used to request or report feature changes, new features, bugs/bug fixes, as well as vulnerabilities/vulnerability patches. Therefore, if a package has many open issues it may be true that the open issues include vulnerability reports and the package is vulnerable. Similarly, if a package has many closed issues, this may mean that many vulnerabilities have been patched and the package is more secure. This may not be a useful metric on its own because GitHub issues are not solely used for reporting vulnerabilities. However, We thought it was important to explore this.

4.2 Popularity Baseline

There are over 1.4 million packages on NPM, and while the information from each of these packages may be useful there are many cases where packages lack documentation of vulnerability history. The assumption was made that popular packages have a greater expectation to have clear and complete documentation of code changes. We also assumed that popular packages are more likely to have vulnerabilities identified due to the number of users exposed to the code base. To ensure that the packages analyzed would have the data necessary we used a popularity baseline to decide what packages would be included in the data set. This baseline is inspired by Spellbound; Taylor et al. determined that the number of downloads for a package was the most indicative measure of popularity, and considered packages with 100,000 weekly downloads as "unquestionably popular"[26]. However, the authors also mention an important piece of information, that download counts represent more than the number of people that have downloaded the package. The creators of NPM estimated in 2014 that a package can be downloaded up to 50 times per day without ever being installed by an actual developer[26]. Therefore, ensuring packages had community involvement and a history of documentation also meant it was important to set a baseline that would eliminate packages that were not be downloaded by humans. For these reasons, the popularity baseline set for this project includes packages with 50,000 or more monthly downloads. This would eliminate any packages with counts that are inflated by bots or automated services, and include packages that clearly have strong community usage and involvement.

4.3 Gathering the Data

After determining what metrics to calculate and the data needed, we created a program that would scrape various websites and use APIs to gather the information needed. To get a list of package names, it queries the official NPM registry which returns a JSON array of the packages and some extra data that was discarded. Then the package names are used to query the NPM registry for the number of downloads per package in the past month. Once the package names and download counts were determined, packages with less than 50,000 downloads were filtered out, following the popularity baseline. This left me with about 15,949 out of 1.4+ million packages. This new list of packages is then used to get the number of dependents from the Libraries.io API, as well as the GitHub repository URL associated with the package.

```
async function getDependendentsAndUrl(packageName) {
    const librariesData = await fetch_retry_json(`https://libraries.io/api/NPM/${packageName}?api_key=${librariesioApiKey}`)

    if (librariesData == null) {
        // return[dependentCount, URL], -1 is used to represent no data found
        return [-1, null]
    }
    return [librariesData.dependents_count, librariesData.repository_url]
}
```

The GitHub repository web page is then scraped for the number of open and closed issues.

```
async function getIssues(repo) {
    // issuesCache is used to reduce the amount of redundant queries
```

```
3    if (repo in issuesCache) return issuesCache[repo]
4    const res = await fetch_retry(`https://github.com/${repo}/
   issues`)
5
6    // 404 is a page not found error
7    if (res.status === 404) {
8         console.log(`https://github.com/${repo}/issues`)
9         return ["Not On Github", "Repo Removed"]
10   }
11   const text = await res.text()
12   const parsed = parse(text);
13
14   // This is where the GitHub page is scraped for the issue
   counts
15   let issues = parsed.querySelectorAll('#js-issues-toolbar a').
   slice(0, 2)
16   if (issues.length === 0) {
17        return [0, 0]
18   }
19   try {
20        let open = parseInt(issues[0].removeWhitespace().text.split
   (' ')[0].replace(/,/g, ''))
21        let closed = parseInt(issues[1].removeWhitespace().text.
   split(' ')[0].replace(/,/g, ''))
22        issuesCache[repo] = [open, closed]
23        console.log(`${repo}: ${open}, ${closed}`)
24        return [open, closed]
25   } catch (err) {
26        console.error(`Error getting issues for ${repo}: ${err}`)
27        issuesCache[repo] = [-1, -1]
28        return [-1, -1]
```

```
29     }
30 }
```

For information on known vulnerabilities associated with each package, we utilized Snyk's online vulnerability database [21]. This database provides detailed information for known vulnerabilities associated with all versions of a package, including their classification (high risk, medium risk, low risk), a description of the vulnerability, and what version resolved the vulnerability if applicable. For the purpose of this project, only vulneravility data about the latest version of the package was collected, and excluded vulnerabilities associated with the dependencies of the package. The purpose for this was to keep the collected data as relevant and related as possible. The data for downloads, dependents, and issues were collected in real time, and therefore are related to the current version of the package and should be compared to the current number of known vulnerabilities in the package. Vulnerabilities associated to dependents of a package were not included because they are not directly in the package's code and therefore the other data collected is not related to it. There is also a significant amount of research and existing tools that can search packages dependencies for known vulnerabilities, so working on this further would have been redundant. Snyk's database also excludes dependency vulnerabilities by default. So, the Snyk database page associated with the package is scraped for the number of known vulnerabilities in the latest version.

```
1 async function getVulns(packageName) {
2     let vulnCount = 0;
3     // getting the number of vulnerabilities from the Snyk.io DB
4     let res = await fetch_retry(`https://snyk.io/vuln/npm:${
    packageName}`)
5     let snykBody = await res.text();
6     let $ = cheerio.load(snykBody)
```

```
7     if (!snykBody.includes('No known vulnerabilities have been
    found for this package in Snyk\'s vulnerability database.') &&
8         !snykBody.includes('404: Page not found')) {
9         // The known vulnerabilities in the current version are in
    the first 3 elements with this class on the page
10        $('.severity__item-count').each((i, e) => {
11            if (i < 3) {
12                vulnCount += parseInt($(e).html());
13            }
14        });
15        console.log('vuln count ' + vulnCount);
16        return vulnCount;
17    }
18    return -1
19 }
```

The data is then written to a CSV file which can then be exported and analyzed.

4.4 Data Analysis

Using the data collected from the program, we calculated the Pearson Correlation Coefficient and graphed the relationships between the number of known vulnerabilities and each of the other metrics, number of monthly downloads, number of dependents, number of open issues, and number of closed issues. Correlation coefficient absolute values below 0.3 are considered to be very weak; 0.3-0.5 are weak; 0.5-0.7 are moderate; ¿0.7 are strong. There was a notable outlier in the data, a package called node-sass. We investigated node-sass and found that it is a Node.js bindings package for libsass, a python package, and its vulnerabilities are directly caused by its bundled usage of libsass. Because of this, node-sass and its corresponding data

was excluded from the correlation calculations and graphs.

Chapter 5

Results and Discussion

From the data collected, we would like to know if any of the package or repository metrics (monthly downloads, dependents, open issues or closed issues) correlate with the number of known vulnerabilities for a package. We hypothesized that the number of monthly downloads, the number of dependents, and the number of closed issues would have negative Pearson Correlation Coefficients. This would mean that as these metrics increase, known vulnerabilities decreases. Therefore, packages with many downloads or dependents or closed issues can be said to be more secure. We also hypothesized that the Pearson Correlation Coefficient for open issues would be positive, meaning that as it increases, known vulnerabilities increases. Therefore, packages with many open issues can be said to be more vulnerable.

The Pearson Correlation Coefficient (PCC) for downloads or dependents vs. known vulnerabilities were both negative, which confirms the hypothesis that as downloads or dependents increase, known vulnerabilities decreases, but the absolute value of r in each case was less than 0.3 and therefore the correlation is very weak. However, the public dependent count only includes the number of packages that directly depend on a given package, while number of downloads includes direct and

indirect downloads through chains of dependencies [26]. Therefore the data for the number of dependents per package is not complete, and it may not be the best metric to use as an indicator of vulnerability. The PCC for open or closed issues vs. known vulnerabilities were both positive which means that as open or closed issues increase, vulnerabilities increase, but the absolute value of r in each case was less than 0.3 and therefore the correlation is very weak. This proves the hypothesis about open issues, but is not what was predicted for closed issues.

Pearson Correlation Coefficients (r)	
$Downloads(x) and Vulnerabilities(y)$	$r = -0.010$
$Dependents(x) and Vulnerabilities(y)$	$r = -0.004$
$OpenIssues(x) and Vulnerabilities(y)$	$r = 0.019$
$ClosedIssues(x) and Vulnerabilities(y)$	$r = 0.009$

However, there are many cases where packages share repository information (i.e. react and react-addons-update) which means it is not possible to accurately determine issue counts for each package using this method of collecting data. Therefore, these metrics cannot yet be said to correlate in any way with package security. The correlations between these metrics and known vulnerabilities are not strong, and therefore are not certain. More data is necessary to determine whether these metrics correlate to security in any way.

5.1 Monthly Downloads

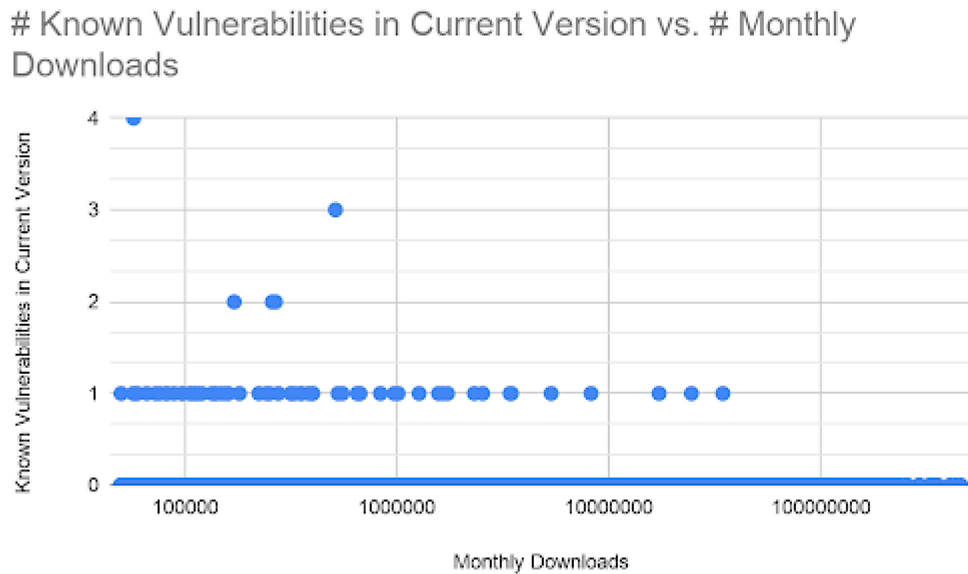

Figure 5.1: The relationship between the number of monthly downloads and the number of known vulnerabilities for NPM packages.

The figure above displays the number of monthly downloads vs. known vulnerabilities for each package in the data set. Notice that packages with 1 known vulnerability are heavily concentrated on the lower end of the x-axis.

Table 5.1: Monthly Download Statistics

Average Number of Downloads for All Packages	6,519,233
Minimum Number of Downloads with Known Vulnerability	50,129
Maximum Number of Downloads with Known Vulnerability	34,386,901
Average Number of Downloads with Known Vulnerability	1,923,944.46
Standard Deviation of Downloads with Known Vulnerability	5,668,886.66

Also, packages with 1 or more known vulnerability have an average of about

1,923,944 monthly downloads compared to the overall average of about 6,712,474 monthly downloads for packages with no known vulnerabilities. Therefore, it can be said that generally, packages with known vulnerabilities are downloaded less than packages with no known vulnerabilities.

5.2 Dependents

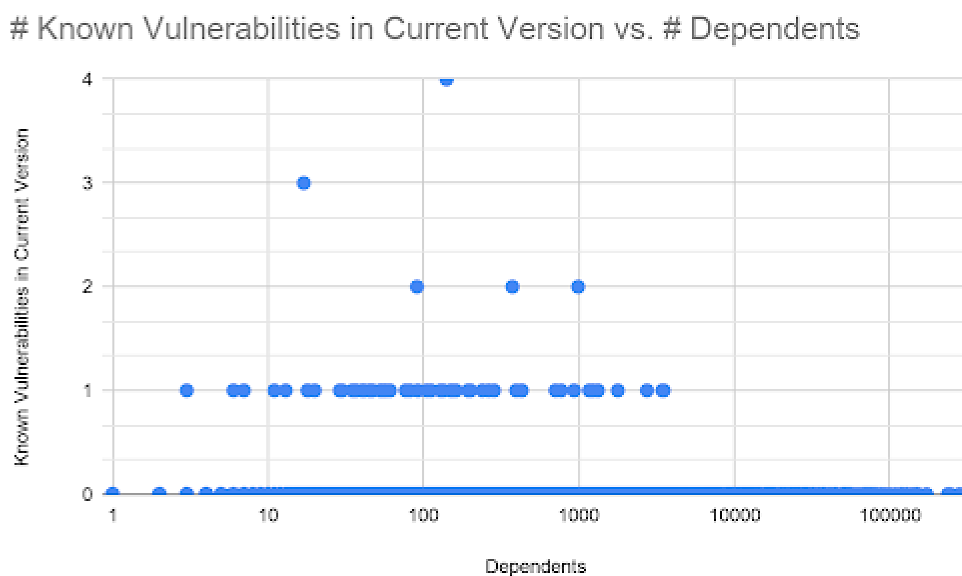

Figure 5.2: The relationship between the number of dependents and the number of known vulnerabilities for NPM packages.

The figure above displays the number of dependents vs. known vulnerabilities for each package in the data set. Notice that similar to figure 4.1, packages with 1 known vulnerability are heavily concentrated on the lower half of the x-axis.

Table 5.2: Dependent Statistics

Average Number of Dependents for All Packages	802.78
Minimum Number of Dependents with Known Vulnerability	3
Maximum Number of Downloads with Known Vulnerability	3513
Average Number of Dependents with Known Vulnerability	384
Standard Deviation of Dependents with Known Vulnerability	744.61

Also, packages with 1 or more known vulnerability have an average of about 384 dependents compared to the overall average of about 828 dependents for packages with no known vulnerabilities. Therefore, it can be said that generally, packages with known vulnerabilities have less dependents than packages with no known vulnerabilities.

5.3 Open Issues

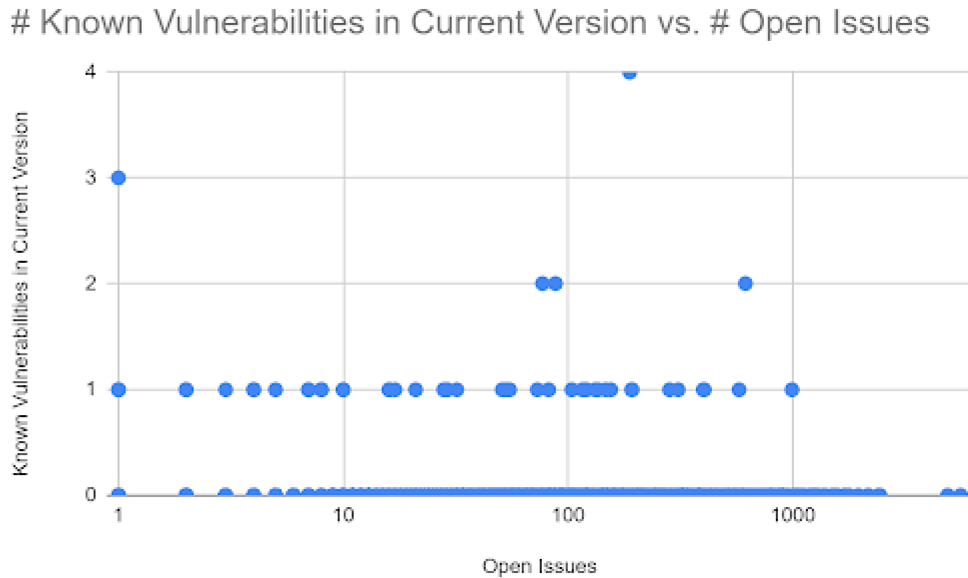

Figure 5.3: The relationship between the number of open issues and the number of known vulnerabilities for NPM packages.

The figure above displays the number of open issues vs. known vulnerabilities for each package in the data set. Unlike the previous figures, packages with known vulnerabilities are more distributed along the x-axis but seem to be concentrated more towards the middle.

Table 5.3: Open Issue Statistics

Average Number of Open Issues for All Packages	49.85
Minimum Number of Open Issues with Known Vulnerability	0
Maximum Number of Open Issues with Known Vulnerability	992
Average Number of Open Issues with Known Vulnerability	95.10
Standard Deviation of Open Issues with Known Vulnerability	175.49

Also, packages with 1 or more known vulnerability have an average of about 95 open issues compared to the overall average of about 49 open issues for packages with no known vulnerabilities. Therefore, it can be said that generally, packages with known vulnerabilities have more open issues than packages with no known vulnerabilities.

5.4 Closed Issues

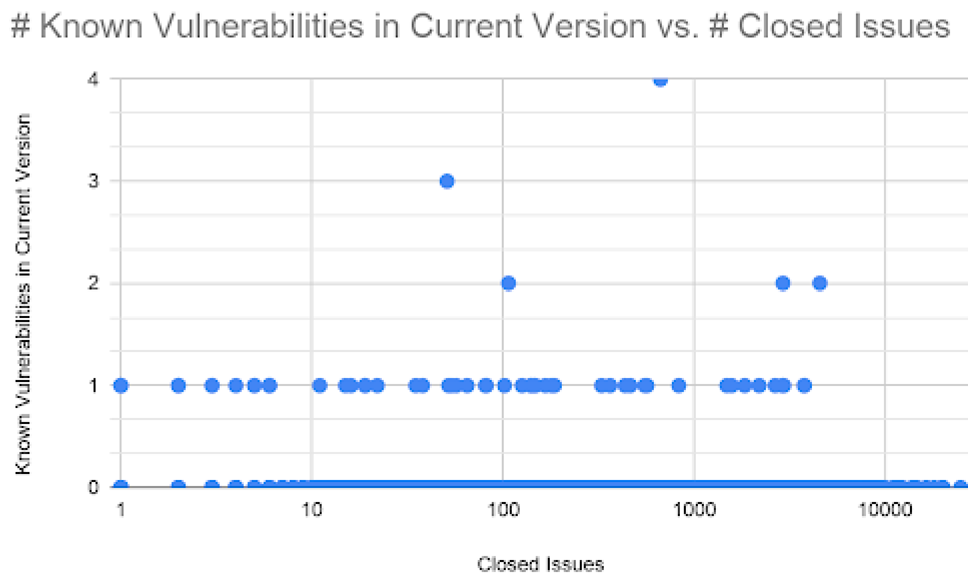

Figure 5.4: The relationship between the number of closed issues and the number of known vulnerabilities for NPM packages.

The figure below displays the number of closed issues vs. known vulnerabilities for each package in the data set. Similar to figure 4.3, packages with known vulnerabilities are more distributed along the x-axis.

Table 5.4: Closed Issue Statistics

Average Number of Closed Issues for All Packages	427.65
Minimum Number of Closed Issues with Known Vulnerability	0
Maximum Number of Closed Issues with Known Vulnerability	4568
Average Number of Closed Issues with Known Vulnerability	587.59
Standard Deviation of Closed Issues with Known Vulnerability	1108.40

Also, packages with 1 or more known vulnerability have an average of about 587 open issues compared to the overall average of about 426 open issues for packages with no known vulnerabilities. Therefore, it can be said that generally, packages with known vulnerabilities have more closed issues than packages with no known vulnerabilities.

5.5 Data Restrictions and Limitations

Out of the 15,949 packages that passed the popularity baseline only 64 packages had at least one known vulnerability. This resulted in a very small amount of data, something that we were not expecting. In this preliminary investigation the level of documentation for a package is irrelevant, and so we believe that the popularity baseline may not be necessary for this project. It may not be feasible to collect this data for all 1.4+ million packages on NPM, but the popularity baseline put in place was certainly too strict. However, we believe the idea of a popularity baseline will be useful for future work.

Chapter 6

Future Work

6.1 Package Manager and Repository Metrics

6.1.1 Open and Closed Issues

We believe there is still more work needed to determine if the number of open or closed issues correlates with security. Packages that share repositories have their own download count and dependent count, but share the number of open and closed issues with other packages. This creates ambiguity, because the package may only be a small portion of a repository but be associated with the issue counts for a larger, more popular package. Therefore, the current issue counts and known vulnerability correlation is likely inaccurate. Future work could filter packages from the data set that share repositories, therefore only including packages that have independent data. This would give a much more accurate and informative result.

6.1.2 Number of Dependencies

The number of dependencies a package had was originally a metric we considered, but decided was too complex to use for this project, because the interpretations for what the number of dependencies could mean is not clear. There are two assumptions that can be made about the number of dependencies a package has: (1) The more dependencies a package has, the more likely it is to be vulnerable because it is depending upon potentially vulnerable code. (2) Assuming all dependencies are completely secure, then the more dependencies a package has, the more secure it is, because less of the code is written and maintained by the main developer(s).

However, there is a flaw with assumption (2): For example, assume a package has 20 dependencies. Assuming these are all secure (2) would then mean that request is more likely to be secure. However, if the 20 dependencies each have 0 dependencies, then it would mean that they are more likely to be vulnerable. This makes me believe that assumption (1) would be the best avenue to pursue.

6.2 Software Metrics

Research has shown that there are project-level software metrics that strongly correlate to the number of vulnerabilities in a project [13]. These metrics include Coupling Between Objects (CBO) of a project (calculated by counting the number of functions/methods of a file/class that are coupled with other files/classes) and SumEssential complexity metric (calculated by count-ing the cyclomatic complexity after iteratively replacing all structured programming primitives with a single statement)[13]. However, these metrics have not been calculated or tested on JavaScript packages and were only tested on 5 projects. Calculating software metrics such as these could be used with package and repository metrics to determine the security

of NPM packages. This could be a tedious and expensive investigation to carry out on all packages in NPM. It could be implemented on a per-package basis, calculating the metrics and determining its likelihood to contain a vulnerability before it is downloaded by a user.

6.3 Metric Manipulation

While trying to determine what metrics reliably correlate to the security of a package, it is also important to think about the ways these metrics could be manipulated if they were to be used in the field. Malicious actors could fabricate large download counts for a malicious package to make it seem secure. Similarly, while a large number of closed issues could imply that a package is well maintained and therefore secure, the contributors of the package could easily create "false" issues to inflate their closed issue count. It is necessary to consider how these metrics will be implemented so as to avoid metric manipulation. To prevent this it may be necessary to include a metric that is not easily manipulated when evaluating a package, such as software metrics like cyclomatic complexity. Software metrics cannot be manipulated through the package manager or repository directly, and therefore would have to be manipulated within the package code itself. It still may be possible to manipulate software metrics, but disguising the fact that there is code in the package specifically for manipulating these metrics may be difficult for threat actors.

6.4 Version Testing

This project was a preliminary investigation into the use of software and security metrics to determine the security of NPM packages. For further context surrounding these metrics they could be compared across different package versions. This is

where we believe the popularity baseline would be best utilized to find packages with extensive documentation. The Snyk database provides information on known vulnerabilities across versions, and could be used to find versions of a package that are vulnerable as well as the version that resolves the vulnerability. This was the original goal of this project, but finding the number of downloads, dependents, open and closed issues of past versions was not something we were able to do. This may be more feasible with other metrics, such as package complexity.

6.5 Vulnerability Score

After determining what individual metrics may correlate with security, an investigation on the combination of metrics and how they correlate with security is necessary. This combination of metrics could be used to determine a vulnerability score for a given package. Software metrics have had success with determining whether code may contain vulnerabilities, so we would combine these metrics with package or repository metrics that correlate to security.

Chapter 7

Conclusion

Package managers are great tools for developers, but currently lack safeguards to prevent their users from malicious attacks or unintentional vulnerabilities. The goal of this project was to determine if there are any package or repository metrics that reliably correlate to the security of a package on NPM. If there are any metrics that correlate to security, a tool could be created to evaluate a package before it is downloaded to warn users of potentially vulnerabilities. The data for this project was collected from package libraries, the NPM website, GitHub's website, and the Snyk known vulnerability database using a program we created. This data was then analyzed, and the metrics were found to have a very weak correlation to known vulnerabilities. From the data analyzed, it was found that generally packages with known vulnerabilities have less downloads or dependents compared to those with no known vulnerabilities. Packages with known vulnerabilities were also found to have generally more open and closed issues than those with no known vulnerabilities. However, there were data restrictions and limitations that may delegitimize these findings. For example, packages with shared repositories have independent package metrics, but shared repository metrics. Also, the dependent counts for packages is

not complete, as it is only the count of direct dependents. Therefore, future work and testing is necessary to determine whether these metrics do correlate to security for certain. A larger data set should be analyzed, as well as removing packages with shared repository information. There is also potential for future work to focus on the combination of metrics to determine a vulnerability score and develop a tool for scanning packages before they are downloaded by a user.

Bibliography

[1] Anish Athalye et al. “Package Manager Security”. In: (2014).

[2] Alex Birsan. “Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies”. In: (2021).

[3] *bundler-audit*. URL: https://github.com/rubysec/bundler-audit (visited on 05/04/2021).

[4] Justin Cappos et al. “A look in the mirror: Attacks on package managers”. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS’08. 2008.

[5] Istehad Chowdbury, Brian Chan, and Mohammad Zulkernine. “Security metrics for source code structures”. In: Fourth International Workshop on Software Engineering for Secure Systems, SESS. 2008.

[6] Istehad Chowdhury, Brian Chan, and Mohammad Zulkernine. *Security Metrics for Source Code Structures*. 2008.

[7] Lucian Constantin. “Dependency confusion explained: Another risk when using open-source repositories”. In: (2021).

[8] Xiaoning Du et al. “LEOPARD: Identifying Vulnerable Code for Vulnerability Assessment through Program Metrics”. In: (2020).

[9] Ruian Duan. “TOWARD SOLVING THE SECURITY RISKS OF OPEN-SOURCE SOFTWARE USE”. PhD thesis. 2019.

[10] Ruian Duan et al. “Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages”. In: Network and Distributed Systems Security (NDSS) Symposium 2021. 2021.

[11] Jake Edge. “Typosquatting in package repositories”. In: (2016).

[12] *GitHub*. URL: https://github.com (visited on 11/01/2020).

[13] Nadia Medeiros et al. “Software Metrics as Indicators of Security Vulnerabilities”. In: (2017).

[14] Microsoft. “Supply Chain Attacks”. In: (2021).

[15] *npm*. URL: https://www.npmjs.com (visited on 11/01/2020).

[16] *npm-audit*. URL: https://docs.npmjs.com/cli/v7/commands/npm-audit (visited on 04/01/2021).

[17] *OWASP Dependency-Check*. URL: https://owasp.org/www-project-dependency-check/ (visited on 05/04/2021).

[18] *Retire.js: What you require you must also retire.* URL: http://retirejs.github.io/retire.js/ (visited on 05/04/2021).

[19] Y Shin and L Williams. "An empirical model to predict security vulnerabilities using code complexity metrics." In: Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '08). 2008, pp. 315–317.

[20] Y Shin et al. "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities." In: IEEE Trans Softw Eng. 2011, pp. 772–787.

[21] *Snyk Vulnerability Database*. URL: https://snyk.io/vuln/?type=npm (visited on 11/01/2020).

[22] Kazi Zakia Sultana. "Towards a Software Vulnerability Prediction Model using Traceable Code Patterns and Software Metrics". In: (2017).

[23] Kazi Zakia Sultana, Vaibhav Anu, and Tai-Yin Chong. "Using software metrics for predicting vulnerable classes and methods in Java projects: A machine learning approach". In: (2020).

[24] Dan Swinhoe. "What is typosquatting? A simple but effective attack technique". In: (2020).

[25] *Symantec Security Center*. URL: https://www.broadcom.com/support/security-center (visited on 05/04/2021).

[26] Matthew Taylor et al. "SpellBound: Defending Against Package Typosquatting". In: *arXiv preprint arXiv:2003.03471* (2020).