Npm package manager security for developers using metadata analysis

Mohammad Zaid Khaishagi
mkhaishagi3@gatech.edu
Georgia Institute of Technology

May 2022

Abstract

The node.js package manager, npm, is widely used by developers who use this programming language in their applications. As such, the supply chain attacks that target developers using attack vectors like typosquatting, account compromise, and malicious publish are also increasing. Most of the proposed solutions for addressing this threat focus on suggesting increased security controls on the part of registry maintainers and package maintainers. While there are some tools which are aimed at developers, there are shortcomings for these. So, there is a need for security tools to be implemented which help the developers to mitigate the risk from downloading unsafe packages. This project aims to provide such a tool while addressing two specific attack vectors: typosquatting and malicious publish, which are among the most prevalent attack vectors on package manager ecosystems. The approach used in this project uses the metadata information in order to perform various tests that aim to detect if the package being installed by the developer could be a potential typosquatting or malicious publish package. The evaluation results for these tests show that such an approach has good effectiveness for detecting potential typosquatting, but has limited use for detecting malicious publishes. The tool developed in this project can be used by developers to analyse packages before installation which allows developers to make an informed decision about whether to continue. The analysis takes roughly 4 seconds to complete for a package and does not adversely affect usability. Using this approach of analysis, this tool provides risk mitigation against supply chain attacks.

1 Introduction

Node.js is a very widely used interpreted programming language and npm, the package manager for it, is used to install packages from the npm repository. The npm package manager provides useful utility by allowing developers to easily initialise their projects and then install a variety of packages for many different

purposes for their projects. These can include things such as template engines [1], web application frameworks [2], resource routing [2], coding utilities [3], IoT [4], and many more. The npm repository is easy to use for uploading repositories and downloading them for use while working on a project. The npm repository has been growing rapidly [5] and this makes it important to provide effective security tools for the developers who rely upon it.

There are various attacks that are launched against developers by uploading malicious packages, and these have been rising over recent years [6]. The attacker goals include things like stealing credentials [7], installing backdoors [8], crypto mining [9], and extracting information about environment variables [10]. Such attacks have an increased level of danger because of their downstream effects in the software supply chain. Developers use packages from npm to build their application and supply chain attacks can amplify the damage from a vulnerable package. Victims can range from normal users who visit websites or use web applications to companies that use npm to build their products. As an example, gathering information about the environment variables [10] on a developer's device can be used as part of reconnaissance done by an attacker on the development environment used inside a company and then use this information for other attacks such as phishing. The importance of securing the supply chain has become more pressing in recent times in light of high impact attacks such as Solarwinds and log4j vulnerability [11].

There have been efforts to assess the trends of attack vectors [6] [12] [13] and propose solutions to defend against exploiting package repositories [14] [15] [16]. Zimmerman et al. [14] studied the npm ecosystem and showed a large attack surface in it. Ruian et al. [6] conducted an analysis of multiple package repositories for interpreted languages and evaluated the trends in registry abuse with different attack vectors.

The solutions proposed in these works focus on increasing the security for the registry maintainers who manage and host registries, the package maintainers who develop and upload packages to these registries, and to have tighter security around the package upload process [6] [12]. Npm has a very open policy regarding publishing packages to its registry; it can be done using the npm-publish tool [17]. There are also no code reviews required when publishing packages. Since it is easy for anyone to publish and there is no review, the onus falls on the developers to make sure that the packages they download are secure [18]. Some other tools provide security notification and information to developers [15] [16] about the packages and its vulnerabilities but these do not address install scripts and also have other issues which are discussed later. There is also the issue of usability when a registry implements security controls which make it difficult for independent or small scale developers to upload their packages freely to the registry, even if it would improve the security of the registry.

There is a lack of solutions which focus on the developer side with a more active approach. The solutions use either a cross-checking approach with a database or advisory [15] [16], or have some blindspots like in BreakApp which does not consider install scripts [14]. Bug finding tools exist which can be used to find bugs in packages but full code coverage in dynamic analysis testing is

difficult. So, developers do not have many options in the way of an easy tool to mitigate the risk from packages when they are installed on their systems for use in their projects.

The problem statement, therefore, is that developers can be exploited when they install npm packages by various attack vectors and this can be improved by providing an additional layer of security using metadata analysis during package installation without significantly adversely affecting usability. This project aims at addressing the typosquatting and malicious publish attack vectors. The malicious publish attack vector is also referred to as 'malicious package' in this project.

The solution approach in this project is to use metadata from the packages—metadata analysis—in order to perform various tests before installation which would help to detect characteristics of typosquatting or malicious packages. The results from these tests are then presented to the user who then has the opportunity to decide whether to continue with the installation of the package. The main benefits of this approach are that it (1) mitigates risk for developers by providing information about potential typosquatting or malicious package; (2) does not install the package and so does not run any installation scripts which might themselves be malicious, which is important because performing program analysis after downloading and installing the package means that the scripts have already been executed; and (3) keeps the tests lightweight by avoiding rigorous static and dynamic testing of all package code and thus provides better usability to the developer.

In implementing this approach, there are a total of 13 tests collectively for typosquatting and malicious publish packages. These tests were evaluated for effectiveness against three data sets of benign, typo and malicious samples. For the typos, 5 tests showed greater number of positives as compared to benign packages, with some of them being very effective. For the malicious samples, 6 tests (different from typos) showed greater rate of positives than the benign sample set, but most of them had limited effectiveness. As an observation, tests that used package popularity performed better than the others.

This tool can be used to give information to the developer about potentially malicious packages, in the typosquatting or malicious publish sense, before installing it. It provides a means of package analysis at installation time and reduces the risk posed to developers by malicious attackers using typosquatting and malicious publish as attack vectors. It also does not adversely affect usability since it takes only a few seconds to run.

2 Background

2.1 Stakeholders

There are 4 primary stakeholders in the ecosystem [6]. In summary, these are:

1. Registry maintainers: These are responsible for the package repositories

that host the packages which get published on it. 'Registries' and 'repositories' are used interchangeably.

2. Package maintainers: These are responsible for maintaining the package itself which they publish to the repository. They work on the project, publish new versions, maintain the codebase, etc.

3. Developers: These are the consumers of the packages which are available for download from the repositories. They download the packages and install them in order to use the utilities they provide in their applications and projects that they are working on. It should be noted that developers can also be package maintainers because these developers may download and install packages in order to work on their project which they later publish as a package to the repository.

4. Users: These make use of the end-user applications that are developed using different packages. Such applications include websites, web applications, software, etc.

The focus of this project is to develop a tool that mitigates the risk posed for the developers when they install packages available on the npm repository.

2.2 Package capabilities

In npm, packages can be installed using the npm tool which fetches the package files and installs them in the project directory automatically. This is a convenient feature of the npm package manager. However, there are some key points about packages which allow a potential attacker to run malicious code on the developer's system. Firstly, when a package is imported in node.js using `require`, the node.js interpreter will parse and execute the code within the files of the package. This means that if there is a line of malicious code in the package, then when that package is imported, this malicious code will be executed regardless of whether the importing code ever calls any of the functions or uses the utilities exported by this package. Second, node.js packages can specify various installation time scripts for the package. These scripts are automatically run by the npm when it installs the package in the project directory. Npm does have some safeguards in place to contain the effect of malicious scripts. One of these is that it downgrades the privilege of the script to the current user even if the install command was run with `sudo` privilege. This mitigates some damage, but is not sufficient and leaves room for damage still. To give one example, it is possible to gain access to the `/etc/passwd` file with these scripts.

So, there are capabilities available to an attacker who is able to get a victim to install a package with malicious code using various attack vectors.

2.3 Trends in registry abuse

According to Ruian et al. [6], the attack vectors for registry abuse can be categorised as follows, ordered by prevalence:

1. Typosquatting: An attacker publishes a new package with a similar name and then relies on developers making a mistake while typing the name of the package so that the incorrect packages gets installed and used. This incorrect package can then contain malicious code.

2. Account compromise: An attacker is able to compromise the account of a package maintainer and can then use this account to make malicious updates to the packages. This can allow the attacker to poison the projects of developers who then download the new version of this package.

3. Malicious publish: An attacker publishes a standalone package that has malicious code in it. It does not necessarily have to be a typo or a dependency of another package.

4. Malicious contributor: An attacker gets to contribute code to a package. This allows for contributing code to a package which could potentially contain malicious logic which is obfuscated in order to avoid detection.

5. Ownership transfer: An attacker convinces a package owner to transfer ownership, for example, by using social engineering.

6. Disgruntled insider: Someone who already has access to the codebase for the package makes some malicious updates. An attacker can use social engineering to convince someone to do such an action.

For this project, the two attack vectors that are being considered are typosquatting and malicious publish.

2.4 Existing tools

There do exist some tools which help the developers to mitigate risk when they install packages. However, these have some drawbacks which are addressed here. These tools can also be used in conjunction with this project in order to get greater security, but this may be at the cost of usability. Such improvements are discussed in the future works section later.

BreakApp. BreakApp is a tool which allows for the imported code from a package to be compartmentalised when working on a project. This prevents the vulnerabilities from one package affecting other components of the project. It is a good approach as it isolates each package and restricts supply chain attacks. [14]

Shortcoming. The shortcoming with BreakApp is that it only does the compartmentalisation of packages when they are imported into a node.js file or code. This leaves room for an attacker to target a developer using install scripts. So, even though it is a good mitigation step, it leaves holes in the defence.

Npm-audit. The npm registry maintains an advisory [16] [19] for the packages that it hosts. When the npm-audit functionality of the npm tool is used for a package, it checks against this advisory and reports the vulnerabilities or alerts for it. The npm tool runs the npm-audit functionality by default whenever

a package is installed. This gives the developer important information about the package.

Shortcoming. The npm tool's default behaviour is that it will show the alerts and vulnerabilities using npm-audit but then it will also install the package which was requested without prompting for confirmation to continue. This is a minor point but relevant nonetheless. The other thing is that it references an advisory database which is updated when bugs or vulnerabilities are found—known vulnerabilities. So, it does not try to detect potential malicious activity or code.

Snyk.io. It is similar to npm-audit in that it references a vulnerability database to check for problems with any packages, utilities or dependencies that the project is using [15] [20] [21].

Shortcoming. Similar to npm-audit, referencing a database of known vulnerabilities does not allow for detecting potential malicious behaviour.

Synode. Synode checks for code injection vulnerabilities in packages, specifically with 'eval' and 'exec'. It is easy to deploy, is fast, protects against vulnerable modules, and has a low false positive rate. [22]

Shortcoming. Although it is effective, it is restricted to checking for code injection, which is only a single type of vulnerability. An attacker can execute malicious logic without using code injection with something like typosquatting, where a developer downloads a typo package and this leads to the malicious logic being executed either in the install scripts, when the package is imported, or when some functions are called.

Semgrep and other static analysis tools. Semgrep is a tool that can be used to perform static analysis on npm packages and is also effective [23].

Shortcoming. Even though static analysis using this tool is an effective approach, it adds additional work with having to download the source code for the project and then perform the analysis. The approach taken in this project aims to give the developer information which would prevent downloading a suspicious package so that the static analysis is not necessary. The static analysis can also be done as an additional step if the package raises some alerts and the developer wants to continue with package installation.

The solution proposed in this project looks at indicators of potential malicious activity, which is not addressed by the existing tools discussed above. The benefit that this provides is that it gives the developer an idea of the risk associated with installing the package before it is installed, which the developer can then use to make a decision on whether to continue with the installation. So, even with fast static analysis tools which would require downloading the package and then analysing it, this approach can inform a developer with preliminary information about the risk associated with installing the package before it is downloaded for installation.

3 Solution

3.1 Core idea

The core idea is motivated by the framework of Ruia et al. [6] where they perform three types of testing to identify malicious packages on different repositories. In order, they perform metadata analysis, static analysis and dynamic analysis, followed by a manual review of the packages and then adjust the heuristics used by their automated detection framework. In their work, they aim to identify the trends in registry abuse by attackers using malicious packages. This provides useful information for registry maintainers and package maintainers to implement security controls, but developers are still at risk.

This work builds on that by using a similar approach in order to mitigate the risk for developers that comes from installing packages. With a lightweight testing mechanism for the package, using metadata analysis, it would mitigate risk by providing alerts to the developer about characteristics seen with potential typosquatting or malicious packages. Performing these lightweight tests on the developer's client-side also prevents any loss of usability that would come from restricting the upload process with things like code review, vetting, etc. For comparison, the `apt-get` package manager for Debian and Ubuntu has implemented such controls which has made it difficult for independent or small-scale developers to upload [24] [25] [26]; whereas on npm, a developer can upload a package with minimal effort using npm-publish [17].

This solution uses only metadata analysis in its tests. Static and dynamic analysis are discounted for the sake of keeping the tests lightweight as well as for the sake of limiting the scope of the project. The possibility of using static analysis to improve the effectiveness is discussed in the future works section.

3.2 Implementation

This section gives the implementation details of the project. It outlines the details of the modules, scripts, lists used, structure of the tests and how the thresholds were chosen.

3.2.1 Typo generation

The typo generation script is based on the work of Nikolai Tschacher [18]. The script takes an input name and then generates a list of typos for it. It generates typos within one edit distance from the input. The name is modified using three operations for the characters present in the name: deletion, insertion and replacement. These operations are done using each of the characters in the name and every distinct output generated is saved to the list of typos which it outputs in the end. The deletion operation deletes one character from the input. The insertion operation inserts each character in each possible position in the name. The replacement operation swaps positions for each pair of character positions in the name.

3.2.2 Top 1000 packages

A list of the top 1000 packages, based on the number of dependents [27], is used in this project. This list is used in the evaluation of the effectiveness of this tool and is not required if a developer uses the tool for install time package checking.

The purposes that this list is used for includes looking at the top 100 packages for deciding popularity thresholds and looking at 2nd top 100 (ranks 101-200) for evaluation of effectiveness of the tests.

3.2.3 Blacklist

A blacklist of users was compiled from a set of malware samples for npm packages obtained from the work of Ruian et al. [6]. The users involved as either authors or maintainers in these packages were added to this blacklist.

3.2.4 Structure of tests

The tests performed as part of the install time package analysis return results in the form of two values: `positive` and `msg`. The `positive` value is a boolean value that indicates whether this particular test showed a positive for the package that was tested. The `msg` value is empty if `positive` is false, otherwise it is populated with a string that has information on what test was performed as well as useful information about the context. The context can be different based on which specific test it is; for example, testing for the age of a package will include information on which packages are older in publish date. This helps the developer with deciding on whether to continue with package installation or to abort. The `positive` value is useful for internal logic, especially with the evaluation.

3.2.5 Caching

The tool implements a caching feature for the metadata of the packages that it retrieves. It stores the metadata in a separate file for each package name, and also stores a list of names which do not exist. This improves performance because it sidesteps the need to repeatedly make network requests to get information about a package. This feature can also easily be turned off in the code if needed.

3.2.6 Modules

There are two modules which perform different sets of tests of the input package: Typo module, and Anly module ('anly' is short for self-analysis). There is another module, Collective Analysis which makes use of both of these for the purpose of performing tests on the input package. Besides these, an Evaluator module is also implemented which helps with bulk analysis of a list of packages, using the Collective Analysis module.

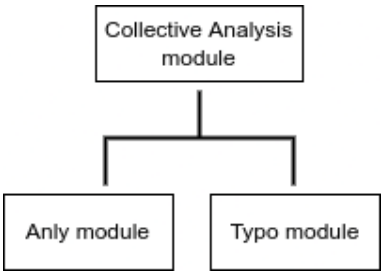

Figure 1: Modules

3.2.6.1 Typo module

This module is responsible for analysing the package in the context of the multiple packages that are surrounding it within 1 edit distance of package name. It looks at the attributes of packages which might indicate typosquatting and whether the input package name is a typo that the developer made while trying to install a package. It is implemented as a class, `Typo_Framework`. It has methods which implement each of the tests that are a part of this module. It also has utility functions which allow for retrieval of valid typo package names, their metadata, results, etc.

In the initialisation, it generates all the possible typos around a given input package name. It attempts to retrieve the metadata for each of these and checks whether it is valid or not. Then, it stores information about valid typo packages and discards the others. Checking for validity includes checking for existence as well as checking if the package is a security holding [28]. Once it has a list of typo packages, it stores this information to be used later when it performs the tests for this module across these packages.

When this class performs the tests, it stores the results from each test so that it can be easily accessed later. It also allows for retrieving the list of valid typo packages, the results and the metadata for these packages.

3.2.6.2 Anly module

This module is responsible for analysing a package by itself, without contextual information from other similarly named packages. It is implemented in the class `Analysis_Framework`. When initialised, it retrieves the metadata for the package and parses it to get information about the attributes that it will be using for its tests. After the tests are performed, it stores the results for easy retrieval.

3.2.6.3 Collective Analysis

This module makes use of the classes which implement the two modules described above. It provides the input package name to them, triggers their tests,

allows for retrieving results from either one, tallies the positives that were shown for each test for each of the modules, and allows retrieval of these tally results.

3.2.6.4 Evaluator

This module helps with performing analysis on a list of packages which can be given as either a file or as an array of package names. It also allows for storing of the evaluation results to file and tallying the positives from each of the packages that were analysed. The main use for this is in the evaluation of the effectiveness of this tool.

3.2.7 Tests for each module

The two main modules each have a separate list of tests that they perform. The description of each of these is given here along with an explanation of the rationale for these and what they were expected to find.

Typo module:

1. Popularity comparison: This test compares the popularity across each of the valid typo packages. It does this by using the weekly download numbers (downloads in last week) available using the npm API [29]. Once it has these numbers, it checks for the package with the highest number of downloads and then classifies them based on two thresholds: relative threshold and absolute threshold. The relative threshold classifies as unpopular if it falls below 0.02% of the most popular package. The absolute threshold classifies as unpopular if it has less than 5000 downloads. The test results in a positive if the input package is not the only popular package.

   Rationale: A package which is legitimate in the sense that it is well-reputed and commonly used by developers would have a large number of downloads as compared to others in the typo radius. The requirement that it is the only popular package implies that its popularity is not ambiguous. The thresholds are used to quantify what is considered 'popular'. The way that these values were obtained is discussed separately later.

2. Age comparison: This test looks at the publishing time of the first version of the packages. It raises a positive if the input package is not the oldest package in the typo radius.

   Rationale: If an attacker is using typosquatting to target some package then the packages that this attacker publishes with similar names are going to be published at a later date than the original one.

3. Same Author: This test looks at whether there are multiple packages that have the same author in the typo radius. It raises a positive if it finds multiple packages with the same author.

   Rationale: An attacker doing typosquatting is free to register multiple names in order to increase the likelihood of a malicious package being

installed by accident when a developer mistypes a name while trying to install it. This test was designed to catch such behaviour, however, a different behaviour is observed in the evaluation which is discussed in the evaluation results section.

Anly module:

Note: Illegitimate package means a package that does not have legitimate uses, this would include malwares but is not restricted to it.

1. Version skipping: It checks the published versions of the package and looks for whether any major version number was skipped. For example, if the package has versions 1.x.x and versions 3.x.x, but does not have any version 2.x.x. It raises a positive if a major version number was skipped.

   Rationale: This is mostly a good-practices test. An illegitimate package would be focused on injecting malicious code rather than developing a legitimate project. This could mean that the package would not have proper version ordering.

2. Immature package: It looks at the latest version of the package and checks if it is below 1.0.0. If it is, then it raises a positive.

   Rationale: An illegitimate package would not see much publishing activity and the package can be expected to never go past initial development versions.

3. Strictly increasing versions: Checks whether the package version numbers do not decrease over time, i.e., a lower version number is not published at a later date. If such behaviour is seen, it raises a positive.

   Rationale: It is expected to catch if a package is trying to publish an older version at a later time. This kind of behaviour could be if an attacker wants to publish a vulnerable version at a later time to get developers to download it.

4. Dist-tag latest: The dist-tag latest version number is an attribute in the package metadata which defines the version number downloaded by default when a package is downloaded just by name. So, this test raises a positive when the dist-tag latest version is not the same as the latest version by timestamp.

   Rationale: Similar to the strictly increasing versions test, this aims to catch behaviour in a package where an older version is set to be the default download. An attacker can use this approach after compromising a maintainer's account and then publishing a vulnerable version in order to cause damage to dependent packages.

5. First version: This checks if the first version to be published was greater than 1.0.0. It raises a positive if that is the case.

Rationale: This is also more of a good-practices test. It is intended to catch behaviour where the package development did not follow normal versioning conventions.

6. Maintainer changes: This checks for changes in the maintainers for the package. It includes addition, removal, or replacement of users who are added in the maintainers list for the package. It raises a positive if there are such changes. It also gives information on the versions where these changes happened and what changes happened.

   Rationale: One of the attack vectors outlined in [6] is that of a malicious contributor. This test does not detect a malicious contributor itself but provides information about where a malicious contributor (or maintainer, in this case) could have been introduced to the package.

7. Author changes: This checks if there were any changes in the author of a package across its versions. If there were authorship changes, then it raises a positive.

   Rationale: The attack vectors in [6] include ownership transfer. This test is intended to detect possible places where such an ownership transfer to a malicious owner could have happened. This does not inherently mean malicious activity, however.

8. Package popularity: This checks if the download numbers (last week) are less than 5000 downloads for the package. It raises a positive if this is true.

   Rationale: An unpopular package with few downloads is not used by many people and so it would not have had much review. This means that this package may have unsafe or malicious code.

9. Malicious authors involved: If the collection of authors for a package across all of its versions have a blacklisted user in it, then it raises a positive.

   Rationale: A user who had been involved with a known malicious package before (blacklisted), then a package authored by such a user is suspect for malicious activity.

10. Malicious maintainers involved: If the collection of all maintainers across all versions of the package include a user who was blacklisted, then it raises a positive.

    Rationale: The involvement of a blacklisted user in a package as a maintainer gives cause for concern regarding the malicious nature of the package.

3.2.8 Popularity thresholds

Some of the tests in the modules make use of a threshold for deciding whether a package is popular or not. There are two types of thresholds: relative and

absolute. These thresholds were obtained after doing an analysis of the download numbers from a sample of packages and their typos. The broad outline for how this analysis was performed is as follows. First, a sample of popular packages was chosen from the top 1000 list. Next, a list of typo packages for each of these was generated which were then ordered according to their download numbers. After this, a manual review was done for each of these lists of typo packages to identify which of them were typos in the sense that most of their downloads would be from mistyped install commands. Using this information, a cumulative graph was plotted to visualise the information. This graph showed that there were knee-points, and these were then used as the thresholds. These steps are described in more detail below.

In the first step which involved gathering a sample of popular packages, the top 1000 list was used and the top 100 from these were then chosen as the list of popular packages (top 100 list). The packages from this list are called the 'main package' when discussing the typo lists for each of them. Additional filtering was done for packages that did not have any valid typo packages which resulted in a set of 65 packages.

In the second step, the typo generation script was used to generate a list of all possible typo names for each sample in the top 100 list. These were then filtered so that only the package which actually existed were included. After this, the npm API [29] was used to gather the download numbers for each of them. The download period that was looked at was 'last-week'. This was a design choice that was made based on the download numbers that npm displays on their website for each package. The list of typos for each of the packages in the top 100 list was then ordered according to the number of downloads.

The third step involved manually reviewing the typo packages. For each of the typo package lists, the following approach was taken for the manual review. The packages would be reviewed in order of their popularity, starting with the most popular typo. Each package would be reviewed until a package was found whose downloads were mostly from developers mistyping a name. Once the first such package was found, it would be flagged and the position in the typo list was noted, later referred to as the n-th package. The packages that were less popular than this one would then be skipped from being reviewed. Some of the packages did not have any valid typo packages so these were not considered.

The manual review included looking at the npm page, the github repository (if available), activity for the package, the utility and purpose of the package, etc. After looking at these things, an informed decision was made about considering the package to be an actual typo.

The fourth step involved graphing various things from this manual review stage. In addition to the n-th package, the characteristics of the 2nd typo (the most popular typo after the main package from the top 100 list) were also plotted for reference. The first thing that was plotted for both the n-th and the 2nd typo was the relative downloads, shown in Figure 2 (a) and (b). This is the fraction of downloads that the n-th or 2nd typo had as compared to the main package, as a percentage value. For example, if the main package 'lodash' has 47636300 downloads, and the n-th or 2nd typo has 43511 downloads, then the

percentage difference would be 43511/47636300 = 0.091%. The second thing that was graphed was the absolute downloads for each of the n-th or 2nd typos, shown in Figure 2 (c) and (d). This was used to get the absolute download count threshold. The graphs for these are shown in Figure 2. All of the graphs are cumulative.

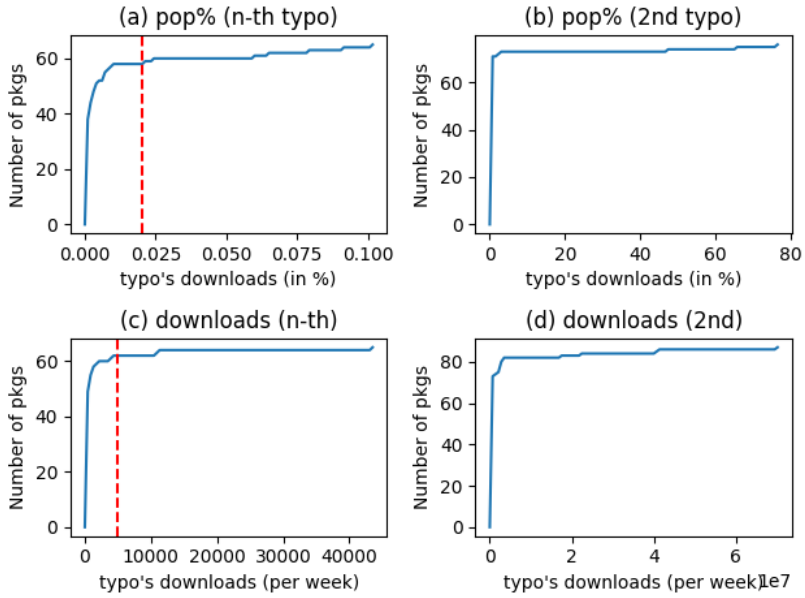

Figure 2: Popularity thresholds

The final step was to choose a suitable threshold. The graphs show knee-points for both the relative and absolute downloads. So, these were chosen as suitable thresholds. The values for these were 0.02% and 5000 downloads.

4 Evaluation

This section describes the evaluation process for the effectiveness of each of the tests that are performed by the modules using an input package. It also describes what results were found for these tests. After the results are described, a discussion on the interpretation of these results is also given.

4.1 Evaluation of effectiveness

The evaluation of each of the tests included in this tool was performed using 3 sets of package samples. The first set of samples is a set of benign packages, called 'ben' or 'benign' later. The second set of samples is a set of typo packages, which are packages that a developer could install by mistyping the name of a

popular package. The third set contains packages that are known malware or have some malicious versions; 'maloss' refers to this set. These are described in more detail below.

The first set of benign packages is obtained by taking the 2nd top 100 packages from the top 1000 list, meaning packages of rank 101 to 200. These were then further filtered to get a final sample set of 63 packages. The filtering process excluded packages which did not have any valid typos or were problematic for other reasons relating to ease of interfacing with the evaluation scripts.

The second set of typo packages was generated by taking the benign samples and generating a list of valid typos for each of them. From these candidates, one package was chosen at random. This formed the set of typo packages. Since it is derived from the first set, it also has 63 packages.

The third set of samples, which are known to be either malicious or have a malicious version, is from the work of Ruian et al. [6]. These samples were provided by them. It contains 31 packages.

Once the sample sets were obtained, the tool was used on each of the packages in these sample sets. This was done separately for each of the sample sets and the results were correspondingly also stored for the sets. The results from each test that the tool performs for a package were recorded for whether they showed a positive or not. This was useful in collecting information about how many packages from the sets showed a positive for the tests.

4.2 Results and findings

The results obtained from the evaluation done using the three sample sets were plotted as bar graphs showing the rate of positives for each set. Figures 3 and 4 show the graphed results for each of the three sets. The results for the benign set indicate the false positives. The results for typo and maloss show the true positive rate when analysing candidates for typosquatting and malicious packages (malicious publish) respectively—the two attack vectors being addressed in this project.

4.3 Interpretation of results

This section discusses the results obtained in light of the positivity rates shown by the 3 sample sets and what these results indicate. The Typo tests are discussed first, followed by the Anly tests. The true positives rate is discussed for both the typo and maloss sets and also compared with the false positives rate from the benign sample set. Any additional observations or comments about the findings are also discussed where suitable.

Typo tests:

1. Age comparison: The true positives rate for the typo sample set shows as 85.71%. This is significantly greater than the false positive rate of 31.74% and shows that this is an effective test to use for detecting possible typosquatting packages.

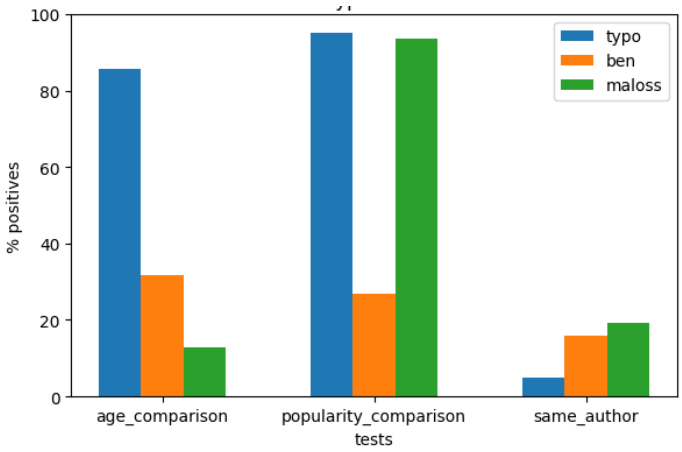

Figure 3: Results for Typo tests

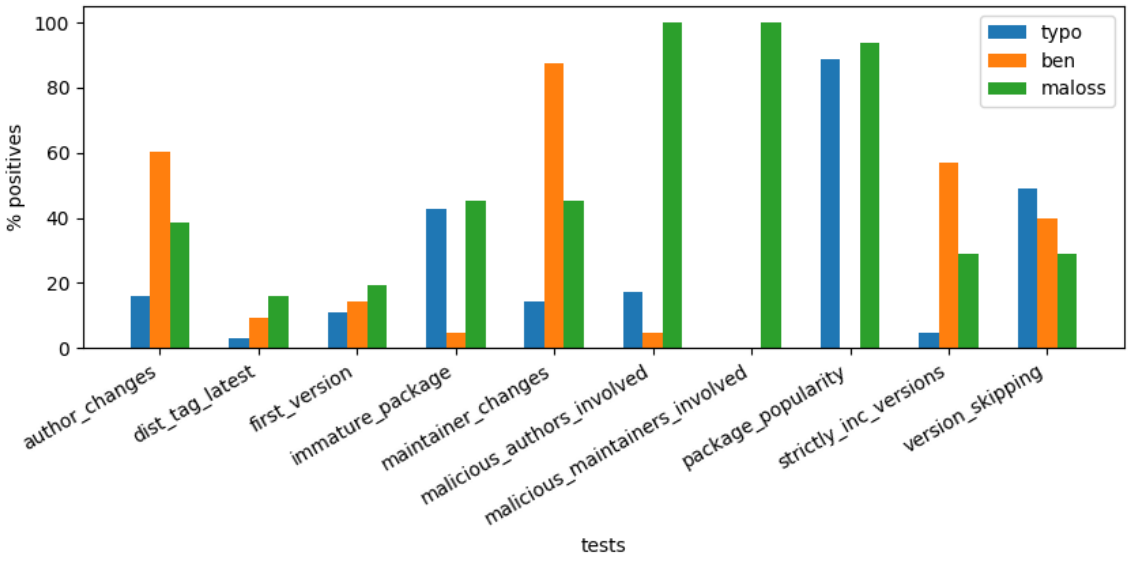

Figure 4: Results for Anly tests

The true positives from the maloss samples show 12.9% positives. This is a low value and is less than the rate for false positives. So, this test is not useful for detecting malicious packages.

2. Popularity comparison: The true positive rate for typo samples is 95.23%, while the false positive rate is 26.98%. This is extremely effective and shows that packages that are typos have low popularity according to the chosen thresholds.

   The true positive rate for maloss samples is 93.54%. This shows that this test is effective when used for malicious samples as well.

3. Same Author:The true positives rate for typo samples as well as maloss samples is low; and the false positives rate is also low. Relatively, however, the test is not effective for the typo samples since the positivity rate is lower than the false positives; and is only marginally effective for maloss samples with a slightly higher positivity rate than the false positives.

   During the development of this project, it was observed that some packages make name reservations where the author registers similar names to their original package in order to prevent typosquatting. This may imply that the Same Author test is a kind of 'inverse' positive where a positive is actually indicative of name reservation and thus benign activity rather than an attacker registering multiple typosquatting names.

Anly tests:

1. Author changes: The true positives rate for both the typo and maloss samples is lower than the false positives rate from the benign samples. This means that this test is not useful for either set. However, it does show that maloss samples have a greater positivity rate than typo samples.

2. Dist-tag latest: The positivity rate of all samples is low, but the rate for maloss samples at 16.12% is greater than the false positives rate of 9.52%. This means that it is possible to use this for malicious package detection with some marginal effectiveness. It also shows that malicious samples are more likely to publish an older version as the default version to be downloaded, ostensibly to provide a version with vulnerabilities.

3. First version: The positivity rate for all sample sets is low. This test shows that the maloss samples with a rate of 19.35% are more likely to have a first version greater than 1.0.0, as compared to benign samples with a rate of 14.28%. This means that this test can be used with limited effectiveness.

4. Immature packages: The false positives rate of 4.76% is low, showing that most benign packages have matured development with the latest version being greater than 1.0.0.

The true positives rate for typo samples is 42.85% which is much greater than the false positives rate. This means that this test can be used effectively to detect possible typosquatting packages.

Similarly, the true positives rate for maloss samples is 45.16%, showing that it is also effective in detecting possible malicious packages.

These results also show that typo samples and malicious samples do not have much development activity and remain at versions less than 1.0.0. One reason for this could be abandoned projects, or perhapsthat there is no need to have any development when the purpose is only to inject malicious dependencies or malicious logic.

5. Maintainer changes: The positivity rate for benign packages is 87.3%, which is a high rate. In comparison, typo packages have only 14.28% and maloss packages have 45.16%. This shows that benign packages are more likely to have maintainer changes. This makes sense because benign packages would have more development activity and so there is involvement of more maintainers. However, the test as it is now is not effective for detecting either typo or malicious packages. Instead, if this test is used as an indicator for benign behaviour as opposed to malicious behaviour as an 'inverse' positive, then it can still be made useful.

6. Malicious authors involved: The result for the maloss samples is not valid here because the blacklist was compiled using these samples.

   It has a low positivity rate for both benign and typo packages, but the typo packages still have a higher rate. So, it can be used with limited effectiveness. As an improvement, if the blacklist is expanded, it could be made more effective.

7. Malicious maintainers involved: The result for maloss samples is invalid here. Neither the benign or typo packages have any positives, so this test has no usefulness here.

8. Package popularity: The false positive rate is 0% here.

   The true positives for typo packages are 88.88% and for maloss samples it is 93.54%. This shows that this test, using the popularity thresholds, is very effective for detection of both potential typosquatting and malicious packages.

9. Strictly increasing versions: The false positive rate is 57.14%, which is quite high. The true positive rates for both the typo and maloss samples is also low at 4.76% and 29.03% respectively. This shows that this test is not effective for detecting either typosquatting or malicious packages.

10. Version skipping: The false positive rate is 39.68%. The true positive rate for maloss samples is less than this at 29.03%, so it is not useful for detecting malicious packages. The true positive rate for typo packages is 49.2%, which is slightly higher than the false positive rate. So, it can be

used for detection of potential typosquatting packages but with limited effectiveness.

In all, there are some tests which are not effective, some which are useful as 'inverse' positives indicating benign instead of malicious behaviour, some that have limited effectiveness, and some which are highly effective. The effectiveness of these tests can be used to inform which positives a developer should pay more attention to and which ones can be ignored.

The tests which are very effective for potential typosquatting detection are: Age comparison, Popularity comparison, Immature package, and Package popularity. The test which has a slightly lower level of effectiveness, but is still useful, is Malicious Authors Involved. The following tests have an 'inverse' positive effectiveness: Same Author and Maintainer Changes.

Similarly, the tests which are very effective for detecting potential malicious packages are: Popularity comparison, Immature package, and Package popularity. The ones with limited effectiveness are: Same author, Dist-tag latest, and First version.

5 Deployment

The tool can be used by running a bash script which takes the package name as its argument. It first runs the tests on the requested package and computes all the positives raised for it as well as the alert messages associated with those positives. It then prints these alert messages to the terminal for the developer, sectioned by the module that the test is for. The developer is then prompted to review these alerts before proceeding with the `npm install` command for the package. This prompt is in the form of a 'yes or no' input. If the developer chooses to abort, then the script terminates. If the developer chooses to continue, then `npm install` is run with the package name which will install the package.

The formatting of each alert message includes information on the test name, a short description of the test, and additional information about the context. The context can be things like versions numbers, author names, publish date, etc. as relevant for the particular test.

The tool can be run in only a few seconds. So, it does not cause any significant usability issues. Running a timing test on the script, it takes roughly 4 seconds to perform the tests on the package `passports`.

6 Discussion

This section has discussions on various aspects of the tool developed in this project. The things discussed here are the impact and usefulness of the tool, the limitations, interesting findings during the development of this project, what future improvements can be made and other future works that can be done using the findings presented here.

6.1 Impact

The original problem that was being addressed in this project was that there was a risk to developers during package installation and that there was a lack of tools which provided information on the potential malicious nature of a package before installing it. As shown above, this tool has some tests that it performs which are quite effective in detecting both potential typosquatting as well as malicious packages. This can be used by developers during the development of projects and while installing packages using npm, to mitigate the risk from typosquatting and malicious packages. It also does not exclude malicious behaviour in the installation scripts of the package, which would go undetected when using some of the existing tools as already discussed above. It also has good usability, taking only a few seconds to complete performing the tests for a given package.

6.2 Limitations

One of the more significant limitations of this tool is that some of the tests are not effective or have limited effectiveness if detecting potential typosquatting and malicious packages. Particularly, only few tests are effective at detecting characteristics of malicious packages. This means that the tool is generally better at detecting potential typosquatting than malicious packages. The effectiveness of the tests could be improved by including some form of program analysis in these tests.

Another limitation is the readability of the alert messages. The formatting of these messages can be improved upon to make the alerts more readable and easy to parse.

It also does not check cached metadata for freshness. This means that a developer would need to manually delete this cache in order for the tool to use fresh metadata. It does, however, use fresh download numbers.

The blacklist can be made more robust and also expanded to include npm users from more malicious packages that have been reported in the past. This information could help to improve the effectiveness of the tests that make use of this blacklist.

The absolute threshold used for popularity checks is 5000 downloads per week. It is possible that this threshold would exclude benign packages that are not popular. This can be improved, as discussed later.

The packages which have very short names, e.g., one-letter names, are problematic for some of the Typo module's tests because the typo generation script has a very small set of characters to generate typos with. It is also problematic because there can be many legitimate packages that add just one more character to their name., for example, the packages 'q' and 'qs', 'a' and 'aa', etc.

6.3 Future work

There are certain improvements that can be made to the tool which would make the tests more effective at detecting malicious packages. The first is that static

analysis similar to Synode [22] can be used to detect malicious logic within a package. There are also node.js APIs that have been identified as being suspicious [6]. These can also be incorporated into the tests to further increase the effectiveness of detecting malicious packages.

It is also possible for the developer to use existing tools to further investigate the package if it seems suspicious after reviewing the alerts displayed by this tool. The developer could perform manual review or automated program analysis. Such actions would also help with finding packages that the open source community might consider for deeper inspection. Additionally, a crowd-sources database can be implemented to which developers can add package names that should be reviewed further.

The tool can be expanded to analyse the entire dependency graph of the package that the developer has requested to install. Currently, it only analyses the main package and does not consider its dependencies for analysis. This might increase the time it takes to perform the analysis but it could potentially give more information that is useful to the developer.

Another thing which can be explored in future works is to perform a study on the distribution of download numbers of all packages available on npm. This would be useful to make a more informed decision about what absolute threshold to use for popularity checks, and also about whether an absolute limit should be used.

The typo generation script can be improved to include typos with an edit distance of greater than 1. This can then also be evaluated to see how this affects the effectiveness of the tests.

During the development of this project, it was noticed that the npm install scripts are automatically downgraded to the user level before being executed even if the command was run with ‘sudo‘ privilege. This can be explored further to investigate if it’s possible to bypass this with careful manipulation of `suid`, `ruid` and `euid` values.

7 Conclusion

To mitigate the risk of potential typosquatting and malicious packages when installing packages using the npm package manager, a tool was developed which performs various tests using the metadata for the package. These tests identify characteristics about the package which are observed in typosquatting and malicious packages. The tests are shown to be generally effective for detecting potential typosquatting, whereas they only have limited effectiveness for detecting malicious publish packages. The tool displays results from these tests to the developer who is then prompted for confirmation of package installation. This helps by informing the developer of the risk associated with installing packages from the npm repository.

The tests which are useful for detecting potential typosquatting are: Age comparison, Popularity comparison, Immature package, Package popularity, and Malicious authors involved. The ones with an ‘inverse’ positive characteris-

tic are: Same author and Maintainer changes. The tests for detecting potential malicious packages are: Popularity comparison, Immature package, Package popularity, Same author, Dist-tag latest, and First version.

The results show that detection of potential typosquatting is more suitable for metadata analysis, while it is not so suitable for detecting malicious packages when some form of program analysis is not used along with it. So, the typosquatting attack vector aiming to execute malicious logic can be mitigated effectively using the approach used in this project.

8 Acknowledgement

I would like to thank Professor Mustaque Ahamad and Professor Nadiya Kostyuk for their feedback during the course of this project. I would also like to thank Jeffrey Ho and Tucker Hembree for their help and feedback with designing and implementing this project as well as their advice with some of the challenges faced while working on this project. I also thank Professor Brendan Saltaformaggio for his advice on the project; and also thank him and Dr. Ruian Duan for providing the samples that were used in the evaluation of this project. I extend my thanks to my peers, classmates and friends for their advice and feedback; and my family for their support and encouragement.

References

[1] J. Kasun, *Guide to handlebars: Templating engine for node/javascript*, Mar. 2020. [Online]. Available: `https://stackabuse.com/guide-to-handlebars-templating-engine-for-node/`.

[2] *Express - node.js web application framework*. [Online]. Available: `https://expressjs.com/`.

[3] *Lodash*. [Online]. Available: `https://lodash.com/`.

[4] J. Saring, *10 javascript iot libraries to use in your next project*, Apr. 2018. [Online]. Available: `https://blog.bitsrc.io/10-javascript-iot-libraries-to-use-in-your-next-projects-bef5f9136f83?gi=f56bb43b11b4`.

[5] A. Nassri, *So long, and thanks for all the packages!* Apr. 2020. [Online]. Available: `https://blog.npmjs.org/post/615388323067854848/so-long-and-thanks-for-all-the-packages.html`.

[6] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*, The Internet Society, 2021. [Online]. Available: `https://www.ndss-symposium.org/ndss-paper/towards-measuring-supply-chain-attacks-on-package-managers-for-interpreted-languages/`.

[7] J. Foundation and other contributors, *Postmortem for malicious packages published on july 12th, 2018*, Jul. 2018. [Online]. Available: https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes.

[8] J. Koljonen, *[cve-2019-15224] version 1.6.13 published with malicious backdoor. · issue 713 · rest-client/rest-client*, Aug. 2019. [Online]. Available: https://github.com/rest-client/rest-client/issues/713.

[9] S. S. R. Team, *Newly found npm malware mines cryptocurrency on windows, linux, macos devices*, Oct. 2021. [Online]. Available: https://blog.sonatype.com/newly-found-npm-malware-mines-cryptocurrency-on-windows-linux-macos-devices.

[10] A. Polkovnychenko and S. Menashe, *Malicious npm packages are after your discord tokens – 17 new packages disclosed*, Dec. 2021. [Online]. Available: https://jfrog.com/blog/malicious-npm-packages-are-after-your-discord-tokens-17-new-packages-disclosed/.

[11] E. Orzel, *Lessons learned from 2021 software supply chain attacks*, Feb. 2022. [Online]. Available: https://thenewstack.io/lessons-learned-from-2021-software-supply-chain-attacks/.

[12] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 995–1010.

[13] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2020, pp. 23–43.

[14] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith, "Breakapp: Automated, flexible application compartmentalization.," in *NDSS*, 2018.

[15] *Snyk | developer security: Develop fast. stay secure.* [Online]. Available: https://snyk.io/.

[16] *Npm-audit.* [Online]. Available: https://docs.npmjs.com/cli/v8/commands/npm-audit.

[17] *Npm-publish.* [Online]. Available: https://docs.npmjs.com/cli/v8/commands/npm-publish.

[18] N. P. Tschacher, "Typosquatting in programming language package managers," Ph.D. dissertation, Universität Hamburg, Fachbereich Informatik, 2016.

[19] *Github advisory database.* [Online]. Available: https://github.com/advisories.

[20] *Open source vulnerability database.* [Online]. Available: https://security.snyk.io/.

[21] *What is snyk?* [Online]. Available: https://snyk.io/what-is-snyk/.

[22] C.-A. Staicu, M. Pradel, and B. Livshits, "Synode: Understanding and automatically preventing injection attacks on node. js.," in *NDSS*, 2018.

[23] *Shift left with fast static analysis.* [Online]. Available: https://r2c.dev/#semgrep.

[24] *Ubuntu packaging guide¶.* [Online]. Available: https://packaging.ubuntu.com/html/.

[25] *Packaging new software.* [Online]. Available: https://packaging.ubuntu.com/html/packaging-new-software.html.

[26] [Online]. Available: https://mentors.debian.net/.

[27] A. Kashcha, *Npm rank*, Aug. 2019. [Online]. Available: https://gist.github.com/anvaka/8e8fa57c7ee1350e3491.

[28] Npm, *Npm/security-holder: An npm package that holds a spot.* [Online]. Available: https://github.com/npm/security-holder.

[29] Npm, *Npm/registry: Npm registry documentation.* [Online]. Available: https://github.com/npm/registry.