On security vulnerabilities stemming from the usage of open-source dependencies

Alex Tompkins
Josh Bernasconi
Andrew Davidson
James Toohey
ato47@uclive.ac.nz
jbe113@uclive.ac.nz
ada130@uclive.ac.nz
jto59@uclive.ac.nz

ABSTRACT

Open-source packages are widely used across the software industry, as they give developers access to a wealth of existing functionality without the need to "reinvent the wheel". However, vulnerabilities can emerge when a piece of software depends on many public packages, any of which can introduce additional attack vectors or themselves act maliciously. The industry's attention has recently been drawn to this topic because of the 'event-stream' incident, where malicious code designed to steal cryptocurrency was inserted into the popular Javascript package and widely disseminated to the software that depended on it.

This paper studies the security issues that arise when using package management systems such as Javascript's NPM, Python's PIP and Java's Maven. Use of package managers is ubiquitous in modern software development, hence there can be significant ramifications for the industry if they supply packages containing malicious or insecure code.

The paper's main aim is to determine whether the current state of package managers for the Javascript, Python and Java languages provide a significant risk of installing third party software that is either malicious or insecure.

We hypothesised that these package managers make it too easy to install packages that either act maliciously or expose a system to vulnerability. Furthermore, we believe that the security risks involved with using numerous open-source dependencies are much greater than most software developers realize, even when just using popular packages.

To determine how prolific these dangerous packages are, the most popular packages from each platform were analysed for known vulnerabilities in their dependencies. This analysis showed that even when using just the most popular packages available on each platform, the risk of including security vulnerabilities was not insignificant. Furthermore, popular repositories using open source software (OSS) from PyPI were the most likely to contain unsafe packages, while the vast majority of the vulnerabilities from Maven and PyPI repositories were rated as high severity, unlike NPM.

A survey of software engineering students was also carried out to determine how developers perceive the risk of using open source packages. The results of this survey showed that while developers were aware of the risks with OSS, they were not actively searching for potential risks.

Finally, research was carried out into methods that the industry as a whole could adopt in order to mitigate these risks. Although three solutions were found, only one of these has been partially implemented within the three package managers analysed.

ACM Reference Format:
Alex Tompkins, Josh Bernasconi, Andrew Davidson, and James Toohey. 2019. On security vulnerabilities stemming from the usage of open-source dependencies. In *Proceedings of ACM Conference (SENG401 Conference 2019).* ACM, New York, NY, USA, 9 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Package managers allow developers to quickly and easily locate and install software packages for use in their projects. However, these platforms can create significant security risks when attackers find exploits in free to use packages. It is very common for packages to depend on other packages, to the point where common packages have hundreds of dependencies when the whole dependency tree is considered [8]. If any of these packages are compromised, the end user is at risk.

Today, open source software is commonly used in commercial systems as it lowers development costs while accelerating time to market [17]. As much as 80% of the code of the average commercial product comes from OSS, so vulnerabilities in these dependencies can be a direct threat to the security of such products.

The situation is further complicated because package managers currently offer few ways for end users to make safe decisions when downloading packages, enabling attackers to use techniques like typosquatting (uploading a malicious version of a popular package with a similar name). Package managers such as NPM provide an unfettered ability to install third-party software, typically through a simple command-line interface. While this is convenient, it encourages developers to skip the process of verifying a package is safe and free of vulnerabilities.

There are two categories of security issues related to open source packages: intentional and unintentional vulnerabilities. An intentional vulnerability is defined here as malicious code inserted into a public package by an author with ill intent, with the goal being that that code is executed via its dependent packages. Unintentional vulnerabilities are far more common, where issues in innocuous code such as improper input validation could provide a vector for attacks.

This paper provides a background of the current trends of vulnerabilities, alongside research on incidents that have occurred within the last two years and common attack techniques that are utilised to create intentional vulnerabilities. Examples of these common attack techniques include injecting hidden malicious code, publishing a different version than the source code, typosquatting, and PyPl setup.py exploits.

An analysis of open source dependencies was carried out using popular packages from three major platforms: the Node Package Manager (NPM), Python Package Index (PyPI) and Maven Central.

Furthermore, a study on developers' methods for evaluating the risks of external packages has been carried out to determine whether developers are cognizant about the security risks involved.

Finally, ways to mitigate these security issues has been examined. Some significant ways for package registries to reduce the number of exploited vulnerabilities were found, including registries forcing package releases to have reproducible builds from public repositories, package signing and implementation of the principle of least authority.

2 BACKGROUND

2.1 Current trends of vulnerabilities

Prior research on vulnerabilities in NPM packages [9] has shown that "the number of new vulnerabilities and affected packages is growing over time", and that "most of the reported vulnerabilities are of medium or high severity". This increasing trend in the number of vulnerabilities will, in turn, lead to an increase in security incidents, especially considering the research found that on average it took 32 months for 50% of all high severity vulnerabilities to get fixed.

A paper from 2014 analysing NPM packages found that "61508 of 99631 modules use at least one dependency, and around 3192 (3.2%) of those modules use at least one of the advisories as dependency", with advisories being packages publicly disclosed as vulnerable in the National Vulnerability Database or the Node Security Project [12]. This paper also looeds at the time taken to patch the vulnerabilities after disclosure, partitioning the vulnerable packages into three windows based on disclosure date. They found a positive indication that in recent advisories "modules are patched within in two weeks", whereas advisories published before 2014 took several months to update. This shows a substantial improvement in response time to advisories by developers.

The latest open source security report by Snyk also supports the increasing trend in vulnerabilities in package ecosystems, stating that "In 2018, new [vulnerability] disclosures for NPM grew by 47%, and Maven Central grew by 27%" [21], illustrated in Figure 1.

The OWASP Top 10 2017 [16] lists the most critical web application security risks. At number nine is 'Using Components with Known Vulnerabilities'. The inclusion in the top ten shows how serious of an issue dependency security is, with OWASP stating components run with the same privileges as the application so "Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts."

2.2 Recent Incidents

Over the past two years alone there have been a number of significant and public OSS security incidents:

2.2.1 ***9 March 2017***. *Web crawl finds 37% of sites depend on vulnerable JS libraries [14]*

Tobias Lauinger et al. conducted a comprehensive study of over 133,000 websites and found that 37% of them include at least one library with a known vulnerability. They also found that libraries included transitively, or via ad and tracking code, are more likely to be vulnerable.

2.2.2 ***6 June 2017***. *NPM force resets the passwords of more than 1000 users after credentials were scraped from open source projects [18]*

NPM themselves had to reset the passwords of more than a thousand users after an independent security researcher informed them of account credentials they were able to find available online. These credentials were accessible via Google search and were a result of security breaches on other sites. As some users had re-used passwords, this left their accounts open for misuse. NPM reset the passwords and revoked all auth tokens for the affected accounts.

2.2.3 ***1 August 2017***. *NPM removes 38 typosquatting attacks by a single author [19]*

Again NPM takes action after being notified by a user. The 'crossenv' package was typosquatting on the legitimate 'cross-env' package to steal environment variables. Environment variables are a common way of supplying auth tokens, email addresses and usernames to packages. NPM removed the package, and after further investigation removed all 38 of the 'crossenv' publisher's packages

2.2.4 ***5 September 2017***. *10 typosquatting packages found and removed from PyPi [15]*

NBU, the National Security Authority of Slovakia, identified malicious software libraries in the official Python package repository, PyPI, posing as well known libraries. These packages contained the exact same code as their legitimate counterparts, but had a modified installation script (setup.py). The malicious (but relatively benign) code would send the name and version of the fake package, user name of the installer and hostname of the computer installed on, to a remote server.

2.2.5 ***3 May 2018***. *"getcookies" malicious package snuck into popular package as dependency [5]*

Here the npm team responded to reports of a package containing a malicious backdoor, removing three packages and three versions of a fourth package from the registry. The 'getcookies' package allowed the hacker to input and run arbitrary code on a running server. The author managed to get it added as a dependency to the popular 'mailparser' package, however the code was not executed anywhere.

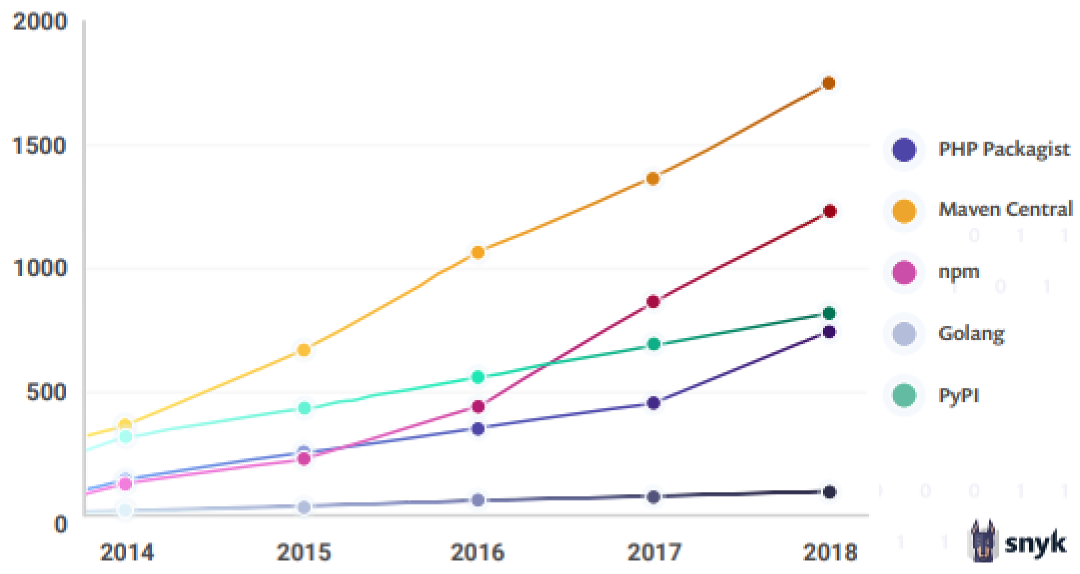

Figure 1: New vulnerabilities each year by ecosystem - Snyk

2.2.6 ***12 July 2018****. NPM publisher's credentials stolen, popular packaged modified to steal data [10]*

An attacker managed to compromise the npm account of an ESLint maintainer and use it to publish malicious versions of some packages. The malicious code would then attempt to steal further package publisher's account details on installation. Npm revoked all access tokens issued before 12:30 UTC, invalidating any tokens that may have been stolen.

2.2.7 ***26 November 2018****. "flatmap-stream" malicious package snuck into popular package as dependency [11]*

Here the attacker asked to take over a popular package that was not actively maintained, and as the developer thought it could be useful he allowed the transfer of ownership. The attacker then added a malicious dependency to 'event-stream' that targeted bitcoin wallets. Npm removed the malicious package 'flatmap-stream' and rolled 'event-stream' back to a safe version, but not before the malicious package gained over 8 million downloads.

It is clear that attackers are becoming more creative in finding ways to manipulate these package ecosystems to their advantage, such as the social engineering required to sneak their malicious packages in as new dependencies of popular packages. An example of this is the technique used in the 'event-stream' attack: the attacker first pushed malicious code to an initially unrelated package 'flatmap-stream', then posed as someone willing to maintain the target library to acquire publishing rights [4]. Once this had been achieved, the attacker added the compromised version of their package as a dependency of the target library, but afterwards updated the dependency, removing the malicious code so that it was not viewable in its latest version (but the compromised version was still used by the target package).

2.3 Common Attack Techniques

2.3.1 Injecting obfuscated malicious code. Commonly, malicious code is obfuscated and added around working code in packages. One obfuscation technique is simply to use misleading and short variable names, and structure function calls to look complicated and unreadable. Because it is situated around other working code, it is easy to overlook. Another commonly used technique for hiding malicious code is to encrypt and store the malicious code, and to then decrypt and run the hidden code at runtime. For example, the event-stream incident hid its malicious code in a string encrypted using AES-256. This was used to overwrite a function called Credentials.getKeys to send the user's bitcoin wallet details to the attacker's server. Because this code was cleverly obfuscated and also placed around working code, it was 2.5 months before the malicious code was discovered [11].

2.3.2 Publishing a different version from the public repository code. An author adding malicious code to an open source package's public repository (such as on GitHub) has a high likelihood of discovery by other contributors and users, who would notice the malicious code being added. However, if the malware author has the rights to publish new releases of a given package, they are capable of uploading arbitrary code to both the NPM and PyPI packages without updating the public repository that the package is linked to. Code may also be obfuscated in the published version using techniques such as minification. The aim of these strategies is to deceive those looking to audit the code themselves, delaying the discovery of the harmful code in the package registry's version.

2.3.3 Typosquatting. Typosquatting, in the context of a package manager, occurs when an attacker publishes a malicious package with a name very similar to a popular existing package in the hopes the target user mistypes the popular package name. An example of this was the 'crossenv' malware package hoping to trick developers trying to download the legitimate 'cross-env' package. While the package would still function as desired, it would also send the values of environment variables from the context to a third-party server. The malicious packages often behave exactly as their legitimate counterparts, whilst also carrying out whatever attack has been hidden inside.

2.3.4 PyPl setup.py. When using pip to install packages, it launches a customizable installation python script called setup.py. This file generally does not receive attention as its functionality is just to help users add the module to the existing python installation. However, the default mode of installation gives full privileges to setup.py when it runs, allowing it to run arbitrary code at administrator level. There is no way to check the contents of this script in between selecting the package for download and execution of the script. Therefore, malicious code be executed just from importing the module if it comes from the setup.py file [6].

3 OPEN-SOURCE PACKAGE ANALYSIS

3.1 Method

The 500 most popular packages available on each platform (Maven, NPM and PyPI) were chosen, with popularity being determined by their ‘SourceRank’ rating on libraries.io. The rating is calculated as the sum of several factors, with the primary ones being the number of contributors, subscribers and dependents a package has [3]. This was chosen as it is reasonable to expect that the most popular packages in each ecosystem have the highest likelihood of being included in a project, and therefore carry the greatest potential to introduce security vulnerabilities into it.

Once these packages had been selected, the locations of their public repositories were determined: the overwhelming majority of these were git repositories available via GitHub. These repositories were then analysed using the SNYK dependency-testing tool, which scans each repository to find its package manifest file [1] . For NPM packages, this is the ‘package.json’ file; for Maven packages, ‘pom.xml’; and for PyPI packages, ’requirements.txt’. The full dependency-chain for each package was then constructed, identifying both direct and transient dependencies.

The tool then searched for known vulnerabilities of each dependency within several public vulnerability databases, the foremost of these being the Common Vulnerabilities and Exposures (CVE) list and the U.S. National Vulnerability Database (NVD).

Vulnerabilities are categorised as either low, medium or high severity.

3.2 Results and Analysis

Table 1 and Table 2 display the findings from the repository analysis.

The results showed 8.34% of all the packages analysed had at least one vulnerable dependency. This demonstrates that even when developers use only the most popular (and actively maintained) repositories, there is still a significant risk of introducing vulnerabilities into their system through the use of third-party dependencies.

When gathering data, NPM repositories tended to give us a much higher rate of successful tests than Maven and PyPI. This was due to less of the popular repositories in Maven and PyPi having pom.xml and requirements.txt manifest files, whereas almost all NPM repositories had package.json files.

When looking at the differences between the mean and median number of dependencies per package on each platform, there is a clear difference between the numbers. From this, we can infer that there are repositories from each platform’s dataset that skew the means. Some repositories have a high number of dependencies, this brings the mean up to what may not be a representative number of dependents for an average representation of a repository. Therefore, we will concentrate on the median as opposed to the mean because we believe it is a better representation of data set as a whole.

PyPI had a significantly higher median of dependencies per repository than Maven and NPM, with an average of 7. This means based on this data that PyPI repositories on average were more likely to have a higher number of dependencies. It follows that when installing software from an average PyPI library, one is more likely to also install a wider range of transitive dependencies. This could correlate to the high ratio of unsafe packages in the popular PyPI repositories. As the number of dependencies increases, so does the chance of including dependencies with vulnerabilities. This relates back to our hypothesis such that Python’s package manager allows for a package to easily be installed without developers being aware of all the additional dependencies that they are getting with it.

NPM had a median of 4, which is not as high as the number we were expecting from our hypothesis. This shows a clearcut difference between the average number of dependencies of NPM and PyPI repositories. The reason for this is inconclusive, perhaps it relates to differences in package manager properties.

However, NPM’s lower rate of vulnerable packages could also relate to the lower branching factor of the average dependency trees. Since it is almost half that of PyPI’s, this may lead to a lower chance of install vulnerable dependencies.

Whereas Maven had a median of 0. This was surprising to us as this suggests many popular Maven repositories have no dependencies. Because of the properties of calculating medians, this means that over half of the dataset from Maven had 0 total dependencies. This obviously means that those repositories would also have no vulnerable packages either. Therefore, the ratio of vulnerable dependencies (8.75%) must come from the few repositories that have high numbers of dependencies.

The following Figure 2 illustrates the ratio of unsafe to total packages for each platform.

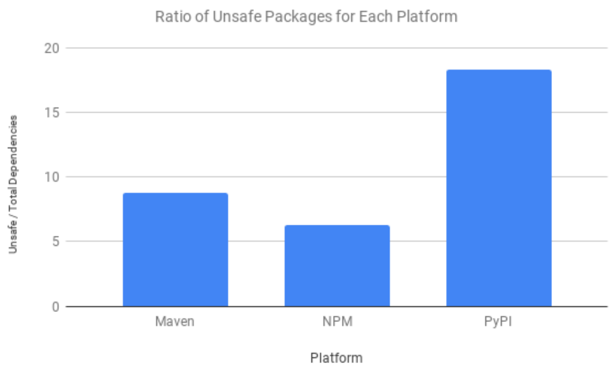

Figure 2: Percentage of unsafe packages to total packages

Table 1: Breakdown of Package Analysis

Platform	Unsafe Package Ratio	Mean dependencies per repository	Median dependencies per repository	Successful Tests
Maven	8.75%	9.33	0	263/500
npm	6.24%	41.78	4	487/500
PyPI	13.35%	17.85	7	139/500
Overall	8.34%	28.59	2	899/1500

Table 2: Breakdown of vulnerability types

Platform	High severity vulnerabilities	Medium severity vulnerabilities	Low everity vulnerabilities	Median days since publication date of vulnerability
Maven	76.09%	21.74%	2.17%	183 days
npm	32.5%	26.25%	41.25%	226 days
PyPI	63.37%	31.68%	4.95%	86 days
Overall	59.25%	26.42%	14.34%	183 days

Out of the three platforms, PyPI repositories were found to have a far higher rate of packages with vulnerable dependencies (15.1%). From this, it draws that packages from PyPI with vulnerabilities are commonly being used, in the popular repositories that this sample was taken from.

3.2.1 Vulnerability breakdown. To further analyse the vulnerabilities, we generated charts to display the differences between severity levels of the vulnerable dependencies in each repository.

Vulnerability severity was measured using the CVSSv3 score provided in the vulnerability disclosure [2] and ranked into three categories by the SNYK dependency-testing tool. The three categories binned scores as follows:

- Low - 0.0-3.9
- Medium - 4.0-6.9
- High - 7.0-10.0

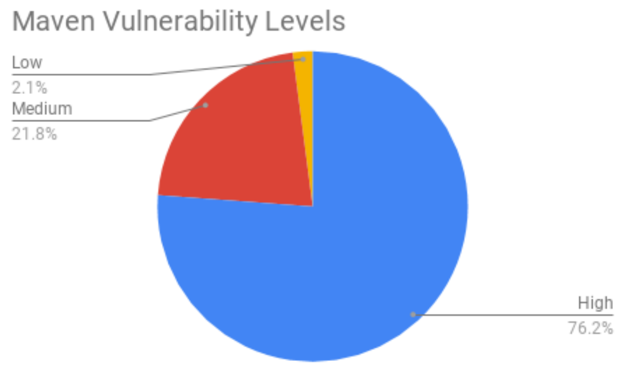

Figure 3: Maven vulnerability severity breakdown

76.2% and 63.4% of the vulnerabilities in Maven and PyPI package dependencies respectively were of high severity, substantially more than NPM's 32.5%. In contrast, NPM had a fairly even split between

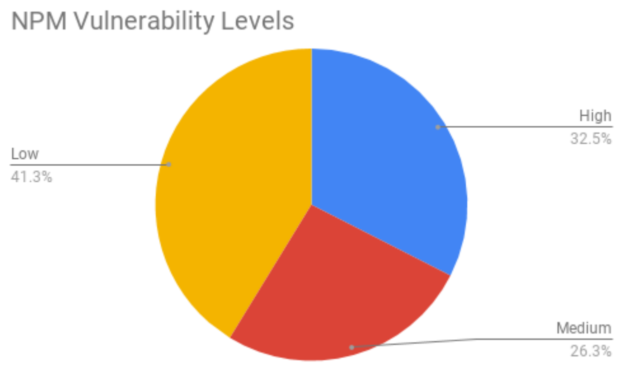

Figure 4: NPM vulnerability severity breakdown

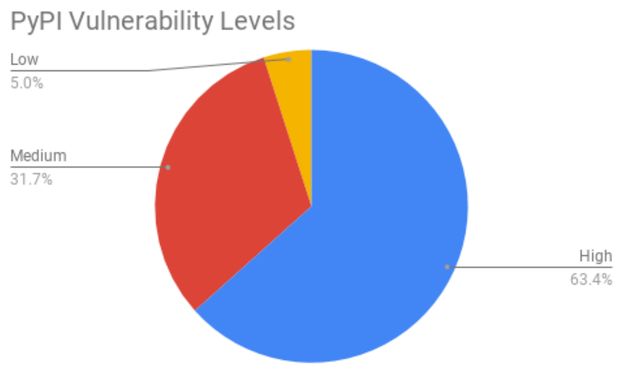

Figure 5: PyPI vulnerability severity breakdown

the three vulnerability levels. While 41.4% of the vulnerabilities were at the low level, this still left over half of the packages at a medium or high level.

We also wanted to investigate if there was a relationship between the number of dependencies and vulnerabilities in repositories, shown in Figure 6.

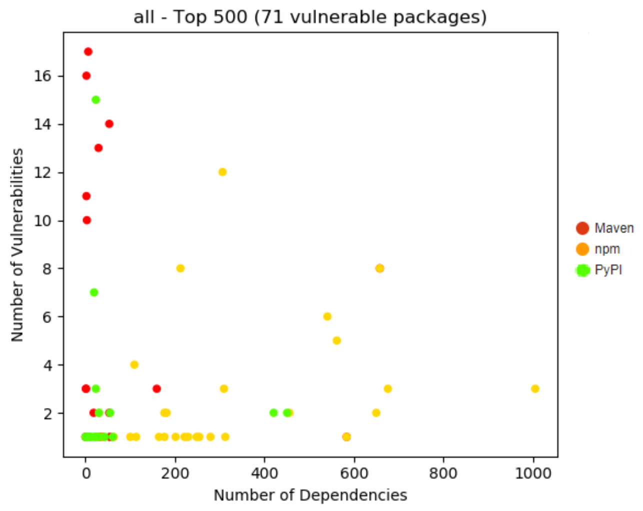

Figure 6: Vulnerabilities vs. dependencies for all platforms

There were no obvious relationships found between the number of packages and vulnerabilities in a repository, which was surprising as it was expected that a package with a larger number of dependencies had a far higher likelihood that one of those dependencies would contain a vulnerability.

We also scraped the last commit date of packages from GitHub, limited to the packages with a valid master branch. This gave us a measure of how active the development on these packages is and an indicator on whether they have had an opportunity to fix the vulnerable dependencies.

When analysing the packages with vulnerabilities with respect to their last commit date, Figure 7, we found that 83% of them had commits within the last 6 months on their master branch. This shows that a large portion of these packages were under active development, yet still vulnerable.

4 DEVELOPER RISK EVALUATION

4.1 Method

A survey was sent out to 4th-year software engineering students within New Zealand. The survey was designed to determine the developer's attitude about using open source dependencies, how often they used them, and how large they perceived the risk of security vulnerabilities stemming from them to be.

4.2 Results

A group of software engineering students were surveyed to assess how they evaluate the risks of importing packages. Initially, data was gathered to discover how many open source packages they used

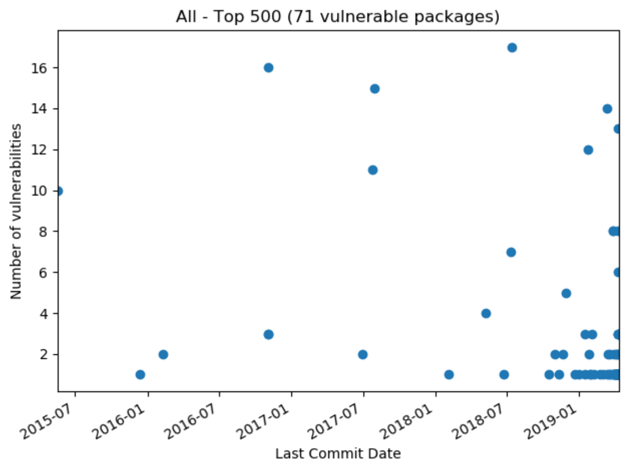

Figure 7: Vulnerabilities vs. last commit date for all platforms

on average — both for personal and group projects. As displayed in Figure 8 there was clearly proof that most developers' individual projects depend on external packages.

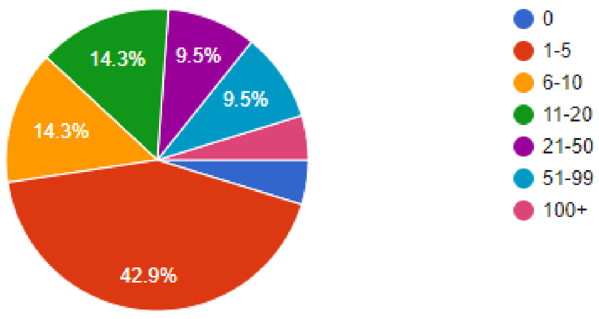

Figure 8: Average number of open source packages used per personal project

There also appears to be a relationship with team sizes and the number of packages depended on. 33% of participants said that group projects would have at least 51 external dependencies, and nearly 75% of participants agreed they would have more than 10. As code bases grow larger (as they tend to in group projects), so does the complexity and the apparent requirement for more dependencies. Therefore, developers are actively using package management systems, and potentially have a process in place for evaluating their risk.

To evaluate the risk of packages, developers tended to check the date of last publish, weekly download count, and what license it uses. This implies how developers have trust in using commonly downloaded packages, with the logic behind this being if other people are using it then it must be good.

Almost 50% of the sample data displayed that developers do not consider security when adding open source software to their system,

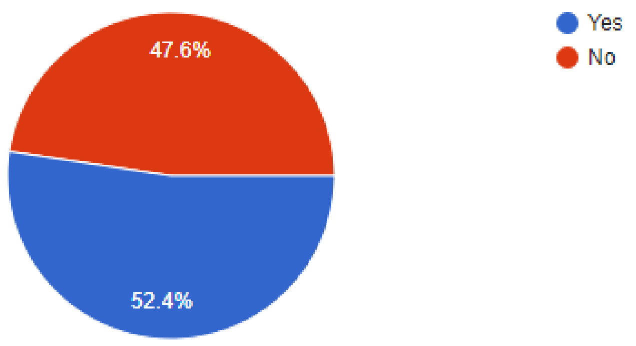

Figure 9: Whether developers consider security when adding open source dependencies to a project

seen in Figure 9. This is clearly an issue as it means that these projects are importing software that could potentially have security faults. The convenience of package managers could contribute to the negligent adding of new packages.

A contradicting point to this is that 47.7% of responses also thought that the security of open source packages was their responsibility to manage. Although developers are clearly aware that they are responsible for their own security, many of them still do not consider security when adding packages. From this, it can be inferred that the ease of use of package managers means that developers can lazily import new packages without looking into their risks.

A section of *In Dependencies We Trust: How vulnerable are dependencies in software modules?* [12] supports this idea. They looked at 85 modules and found "The majority of modules lack discussion about security issues. We identified only 21 modules with security discussions, where only 15 are directly related to the node security project. The findings suggest that there is a low rate of discussions and reports of the advisories."

A study from May 2107, Do developers update their library dependencies? [13], found even more developers weren't checking into the security of their dependencies. They said that "Surveying these developers, we find that 69% of the interviewees claimed to be unaware of their vulnerable dependencies." The study also found that if developers were aware, they may still not update a dependency stating "In the case of updating a vulnerable dependency, the study reveals that affected developers are not likely to respond to a security advisory". Reasons for this were cited as the estimated migration effort was too high and that it was "added responsibility and effort to be performed in their 'spare time"'. This mirrors the findings in our package analysis, Figure 7, that there a large number of vulnerable dependencies still in use in actively maintained packages.

5 MITIGATION TECHNIQUES RESEARCH

Research was carried out into mitigation techniques that could have prevented incidents such as those discussed previously, and increase resilience against open source dependency vulnerabilities in general. Numerous methods exist aimed at reducing the vulnerability of projects using open-source dependencies to the common attack techniques identified above, some still theoretical and some already put into practice by existing package managers.

5.1 Reproducible builds from a public repository

Situations can arise where the published version of a package contains malicious code while the package's source code hosted on a public repository (e.g. GitHub) does not. A potential solution is for the package manager to require publishing directly from a public repository, ensuring that a package's source code is always consistent with what is distributed. Secure NPM, a proof-of-concept for this solution on the NPM platform, works by the publisher providing only the new version number, repository URL and checksum. The registry itself then fetches the released version from the public repo, builds the package and compares the checksum to that provided by the publisher. This allows anyone to reproduce the build and have confidence that the version published on the registry hasn't been tampered with.

Although the aforementioned proof-of-concept exists, neither NPM nor any other package manager has implemented this system so far.

5.2 Package Signing

Another technique that commonly appeared in our research was the concept of cryptographically signed packages. The basic idea is that every release of an open source package should be signed with the author's private key. When the package is downloaded, the author's public key will be used to verify that the contents of the package have not been tampered with [7]. Although Maven, NPM and PyPI all implement some form of package signing, it is purely optional.

NPM was originally designed in such a way that packages could be unpublished from the NPM registry without reason, and thereafter others could take over the package name. This means that new code could be introduced to the package that is not from the original maintainer [23]. To address this, in April 2018 NPM introduced package signing. Each time a new package version is published, a signature would be required to ensure a verified user is releasing this version. The signature acts as validation for signing the version release of a <package>@<version>:<integrity> string. Based on this string, a detached signature is generated and stored in the npm-signature field of the registry's "packument". This allows developers to manually compare the hashes of each versions tarball and integrity string to ensure they match [20] . This solution was only originally available on newly-published package-versions but NPM has slowly been backfilling publishes prior to the package signing launch.

5.3 Principle of least authority

Some in the open source software community argue that the problem lies not with how easy it is to publish and distribute harmful packages, but with the unrestricted capabilities that these packages have once they have been imported. On the Node.js and Python platforms, imported dependencies have the same ambient authority as the code that calls them, i.e. they have the ability to execute arbitrary code. Proponents of the principle of least authority advocate that software components should instead be given only the minimum authority necessary to carry out their function [22]. For example, the platform could deny access to resources such as the file system and the network unless a package is explicitly granted those privileges. This would force developers to be more aware of the points of vulnerability in their projects.

The advantages of this approach are that unless a developer explicitly granted it access to the file system, a malicious release of a widely-used package would not be able to access confidential information such as SSH keys and cryptocurrency wallets, nor would it be able to send that information without network access, such as occurred in the event-stream incident.

6 LIMITATIONS AND FURTHER RESEARCH

This study was conducted over a short period of time and there are many ways to further the research conducted.

6.1 Non-standard package manifests

To analyse a package's dependencies, its open source repository must contain a valid package manifest file. While all NPM packages contain a "package.json" file, many libraries available via PyPI did not include a "requirements.txt". This was because Python packages can also specify their dependencies in other files e.g. the "setup.py" or "Pipfile", which were not analysed by the SNYK dependency testing tool we used. This greatly limited the selection of Python packages we could analyse. Future studies could improve this by adjusting the tool to allow for detecting dependencies present in these additional manifest files.

6.2 Survey representativity

Our developer survey was filled out by fourth year software engineering students undertaking the third professional year of study. While this gives some insights into what software developers think of security issues concerning package managers, it may not be representative of more experienced software developers who work in industry. Future studies could improve this aspect of the research by surveying more software engineers in industry.

6.3 Survey size

The survey only had 21 total respondents, which is a small number to draw conclusions from. A larger sample of people would yield more definite results. Because we thought that students in lower year groups would not have sufficient experience with package managers and the security risks that could be involved, we did not survey them. Again, future studies could improve on this research and find more accurate results by extending the survey to more software engineers in industry, and to a greater number in general.

6.4 Reputable sources

Unfortunately, we were unable to find journal papers that detailed the method and impact of the security incidents we investigated. The research we found focused on methods in an abstract context or trends with attacks, rather than on specific incidents. This meant that our most reputable sources for information on these incidents were blog posts from the package manager's official blogs, and articles written about the incidents. This information is likely to be more opinion based and less impartial, and it is often also missing crucial details about the impact that these incidents had. This made it harder for us to extract meaningful findings from the security incidents that have occured recently.

6.5 Investigation of mitigation techniques

Although this paper includes information about some potential mitigation techniques could be used to reduce the number of dependency security incidents, numerical evaluation of the effectiveness of these techniques was out of scope. Further research could determine how many of the incidents occur . Providing data on how feasible and effective these techniques are in practice would help to identify what preventative measures package managers should be taking.

6.6 Investigation of further package managers

Our study focused only on the Maven, NPM, and PyPI package managers. There are many other software development package managers that are used commonly, and many other types of package managers. Linux software package managers such as Apt and Yum are also commonly used in software development, and vulnerabilities in these operating system level package managers could be disastrous. Further research into these other package managers could highlight other attack techniques and potential vulnerabilities that should be addressed.

7 CONCLUSION

We have found that vulnerabilities due to package managers are widespread and poorly mitigated.We have found that vulnerabilities due to package managers are widespread and poorly mitigated. There have been several serious security incidents within the last two years, which utilize common and effective attack techniques. Some of these attack techniques could be largely invalidated by better package management architecture, namely preventing typosquatting and locking publically available package registry code to reproducible builds from public repositories. The number of vulnerabilities in packages published to these managers is consistently increasing, and many discovered vulnerabilities are not addressed for a significant period of time.

We analysed the 500 most popular Maven, NPM, and PyPI packages to find how many of them had dependencies that were vulnerable to attack using the SNYK dependency-testing tool. We found that PyPI packages had the highest median number of dependencies (7), followed by NPM (4). Maven surprisingly had a median of 0 dependencies. Our dataset showed that PyPI dependencies have higher branching factors in their dependency trees and NPM have the lowest. PyPI also had the highest rate of unsafe packages for their popular repositories, with 13.35% of the tested repositories having vulnerabilities.

Both Maven and PyPI repositories had significant amounts of their vulnerabilities classified as high, whereas NPM had a fairly even split between high, medium, and low.

When looking into whether there was a relationship between the number of dependencies a repository had and the number of vulnerabilities, there was no obvious relationship between the two variables. However it is possible that a larger dataset could prove otherwise.

The software engineers we surveyed were aware that there are security issues involved with using open source packages, but not how to deal with them. They trust recently published, popular packages, but these packages often have vulnerable dependencies. Another survey found that 69% of developers were unaware of their vulnerable dependencies. Clearly, there are issues with the security of package managers, and current awareness of these issues is low.

Some methods to mitigate the growing vulnerabilities are package signing, restricting package code permissions, and only allowing deployment from public repositories.

There have been several serious security incidents within the last two years, which utilize common and effective attack techniques. Some of these attack techniques could be largely invalidated by better package management architecture (typosquatting and uploading code not published public repositories). The number of vulnerabilities in these managers is consistently increasing, and many discovered vulnerabilities are not addressed for a significant period of time.

We analysed the 500 most popular Maven, NPM, and PyPI packages to find how many of them had dependencies that were vulnerable to attack using the SNYK dependency-testing tool. We found that PyPI packages had the most median dependencies (7), followed by NPM (4). Maven surprisingly had a 0 median of dependencies. Our dataset showed that PyPI dependencies have higher branching factors of their dependency trees and NPM have the lowest. PyPI also had the highest rate of unsafe packages for their popular repositories, with 13.35% of the tested repositories having vulnerabilities.

Both Maven and PyPI repositories had significant amounts of their vulnerabilities classified as high, whereas NPM had a fairly even split between high, medium, and low.

When looking into whether there was a relationship between the number of dependencies a repository had and the number of vulnerabilities, there was no obvious relationship between the two variables. However it is possible that a larger dataset could prove otherwise.

The software engineers we surveyed were aware that there are security issues involved with using open source packages, but not how to deal with them. They trust recently published, popular packages, but these packages often have vulnerable dependencies.

Another survey found that 69% of developers were unaware of their vulnerable dependencies. Clearly, there are issues with the security of package managers, and current awareness of these issues is low.

Some methods to mitigate the growing vulnerabilities are package signing, restricting package code permissions, and only allowing deployment from public repositories.

8 ACKNOWLEDGEMENTS

The authors are grateful to the survey respondents who spent their time to respond to our survey, providing us with valuable data.

REFERENCES

[1] [n. d.]. CLI - Test | Snyk. ([n. d.]). https://snyk.io/docs/cli-test/

[2] [n. d.]. Common Vulnerability Scoring System v3.0: Specification Document. ([n. d.]). https://www.first.org/cvss/specification-document

[3] [n. d.]. Libraries.io Documentation - SourceRank. ([n. d.]). https://docs.libraries.io/overview.html#sourcerank

[4] Adam Baldwin. 2018. The NPM Blog - details about the event stream incident. (November 2018). https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident

[5] Adam Baldwin. 2018. The NPM Blog - Reported malicious module: getcookies. (May 2018). https://blog.npmjs.org/post/173526807575/reported-malicious-module-getcookies

[6] Danny Bradbury. 2018. Snakes in the grass! Malicious code slithers into PyPI repository. (October 2018). https://nakedsecurity.sophos.com/2018/10/30/snakes-in-the-grass-malicious-code-slithers-into-python-pypi-repository/

[7] Justin Cappos, Justin Samuel, Scott Baker, and John H Hartman. 2008. Package management security. *University of Arizona Technical Report* (2008), 08–02.

[8] Alexandre Decan, Tom Mens, and Maëlick Claes. 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 20-24 Feb. 2017, Klagenfurt, Austria.* 3–4.

[9] Alexandre Decan, Tom Mens, and Eleni Constantinou. [n. d.]. On the impact of security vulnerabilities in the npm package dependency network. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), May 28-29, 2018, Gothenburg, Sweden.* 181–191. https://dl.acm.org/citation.cfm?id=3196401

[10] ESLint.org. 2018. Postmortem for Malicious Packages Published on July 12th, 2018. (July 2018). https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes

[11] Danny Grander. 2018. Malicious code found in npm package event-stream downloaded 8 million times in the past 2.5 months. (November 2018). https://snyk.io/blog/malicious-code-found-in-npm-package-event-stream/

[12] J I Hejderup. 2015. In dependencies we trust: How vulnerable are dependencies in software modules? (2015).

[13] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (2018), 384–417.

[14] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *Network and Distributed System Security Symposium.* 14. https://seclab.ccs.neu.edu/static/publications/ndss2017jslibver.pdf,https://www.ndss-symposium.org/wp-content/uploads/2017/09/NDSS2017_Proceedings_Front_Matter.pdf

[15] NBU. 2017. skcsirt-sa-20170909-pypi. (September 2017). https://www.nbu.gov.sk/skcsirt-sa-20170909-pypi/

[16] Top OWASP. 2017. Top 10-2017 The Ten Most Critical Web Application Security Risks. *URL: owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf* 29 (2017).

[17] Mike Pittenger. 2016. OPEN SOURCE SECURITY ANALYSIS - The State of Open Source Security in Commercial Applications. (2016). https://info.blackducksoftware.com/rs/872-OLS-526/images/OSSAReportFINAL.pdf

[18] C J Silverio. 2017. The NPM Blog - Credentials Reset. (June 2017). https://blog.npmjs.org/post/161515829950/credentials-resets

[19] C J Silverio. 2017. The NPM Blog - 'crossenv' malware on the npm registry. (August 2017). https://blog.npmjs.org/post/163723642530/crossenv-malware-on-the-npm-registry

[20] C J Silverio. 2018. The NPM Blog - new pgp machinery. (April 2018). https://blog.npmjs.org/post/141905368000/changes-to-npms-unpublish-policy

[21] Liran Tal. 2019. The State of Open Source Security Report 2019 | Snyk. (February 2019). https://snyk.io/opensourcesecurity-2019/

[22] A. Vega, P. Bose, and A. Buyuktosunoglu. 2017. Chapter 1 - Introduction. (2017), 8 pages. https://doi.org/10.1016/B978-0-12-802459-1.00001-4

[23] Ashley G Williams. 2016. The NPM Blog - changes to NPM's unpublish policy. (March 2016). https://blog.npmjs.org/post/141905368000/changes-to-npms-unpublish-policy