TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Kirill Lõssenko IVSB192930

Risk assessment for 3rd party dependencies in software development projects

Bachelor’s thesis

Supervisor: Aleksei Talisainen
MSc.

Tallinn 2022

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Kirill Lõssenko IVSB192930

Kolmandate osapoolte sõltuvuste riskihindamine tarkvaraarendusprojektides

Bakalaureusetöö

Juhendaja: Aleksei Talisainen
MSc.

Tallinn 2022

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Kirill Lõssenko

25.04.2022

Abstract

Risk assessment for 3rd party dependencies in software development projects

The use 3rd party software dependencies poses a great risk despite its benefits of time efficiency and code reusability. With the ease of access to 3rd party dependencies in form of packages and the ease of their installation using package managers, the practice of 3rd party package use is ever-increasing, with almost 2 million packages published in the NPM ecosystem alone. This system introduces risks due to untrustworthiness of publishers whose packages a developer might use. This thesis goes over the risks that arise from the use of 3rd party packages and attempts to offer potential solutions to the growing problem of 3rd party package auditing, by developing a software prototype that would offer a summary of risk factors present in Node based projects, which would help with their subsequent risk assessment. The software design was based off answers to a survey aimed towards JavaScript developers which aimed to determine relevant security practices of JavaScript developers. The thesis attempts to act as a base for future research done on 3rd party dependency risk assessment and plans to expand the developed prototype in the future.

This thesis is written in English and is 44 pages long, including 4 chapters, 3 figures and 2 tables.

Annotatsioon
Kolmandate osapoolte sõltuvuste riskihindamine tarkvaraarendusprojektides

Kolmanda osapoole tarkvara sõltuvuste kasutamine kujutab endast suurt ohtu, hoolimata selle eelistest, mis seisnevad ajalises tõhususes ja koodi taaskasutatavuses. Kuna kolmandate osapoolte sõltuvused on pakettide kujul kergesti kättesaadavad ja nende paigaldamine paketihaldurite abil on lihtne, suureneb kolmandate osapoolte pakettide kasutamine üha enam, ainuüksi NPM-ökosüsteemis on avaldatud peaaegu 2 miljonit paketti. See süsteem toob kaasa riskid, mis tulenevad nende avaldajate ebausaldusväärsusest, kelle pakette arendaja võib kasutada. Käesolevas lõputöös käsitletakse riske, mis tulenevad kolmandate osapoolte pakettide kasutamisest, ning püütakse pakkuda võimalikke lahendusi kolmandate osapoolte pakettide auditeerimise kasvavale probleemile, arendades välja tarkvara prototüübi, mis pakuks kokkuvõtet Node'i-põhistes projektides esinevatest riskiteguritest, mis aitaks nende hilisemat riskihindamist. Tarkvara disain põhines JavaScript-arendajatele suunatud küsitluse vastustel, mille eesmärk oli määrata kindlaks JavaScript-arendajate asjakohased turvatavad. Lõputöö püüab toimida alusena tulevastele uuringutele, mis on tehtud 3. osapoole sõltuvuse riskihindamise kohta, ja plaanib arendada välja töötatud prototüüpi tulevikus.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 44 leheküljel, 4 peatükki, 3 joonist, 2 tabelit.

List of abbreviations and terms

Code libraries	Libraries in programming languages are collections of prewritten code that users can use to optimize tasks.
Pull request	In the context of git, a pull request is a method by which a developer can notify their team that a code feature has been finished. Also sometimes referred to as “PRs” [26].
Node	An asynchronous event-driven JavaScript runtime, Node is designed to build scalable network applications [25].
SAST	Stands for Static Application Security Testing, which includes analysis of the code that has not yet been executed.
SQL	Stands for Structured Query Language, SQL is a standard language for accessing and manipulating databases [27].
CLI	Stands for command-line interface, it is a text-based user interface used to run programs, manage computer files and interact with the computer [28].
Typosquatting	Package typosquatting is a type of software supply chain attack where the attacker tries to mimic the name of an existing package on a public registry in hopes that users or developers will accidentally download the malicious package instead of the legitimate one [29].
Dependency Confusion Attack	A type of attack that attempts to trick the software installer script into pulling a malicious code file from a public repository instead of the intended file of the same name from an internal repository [30].
CVE	Sands for Common Vulnerabilities and Exposures, it is a list of publicly disclosed information security vulnerabilities and exposures [31].
XSS	Stands for Cross-Site Scripting, which is an injection type of attack where malefactor injects malicious script into a website that would target other users of the site [32].
CSRF	Stands for Cross-Site Request Forgery, is an attack that forces an end user to execute unwanted actions on a web application in which they’re currently authenticated [33].
API	API stands for application programming interface, which is a set of definitions and protocols for building and integrating application software [8].

PGP	Pretty Good Privacy, it is an encryption system used for both sending encrypted emails and encrypting sensitive files [34].
LGTM	A code analysis platform for finding zero-days and preventing critical vulnerabilities. Utilizes the CodeQL query engine [35].
CodeQL	CodeQL is an analysis engine that is used by developers to automate security checks, and by security researchers to perform variant analysis [36].
JSON	JavaScript Object Notation is a lightweight format for storing and transporting data [37].
NPM	Is a package manager for JavaScript programming language maintained by the NPM, Inc. [38].

Table of contents

List of figures

List of tables

1 Introduction

This thesis will go over security risks that arise from the use of 3rd party dependencies in the context of the Node ecosystem and the NPM package manager, what problems arise from those risks and how these problems are addressed today to see if there are any additions that could be made to the current pool of solutions. The specifications for the attempted solution will be based on a survey targeted at JavaScript developers. The survey will attempt to find problems that are relevant to modern developers that are not yet addressed by the current state of the art and develop a solution for them. The NPM package manager was chosen as a focus for this thesis as it is currently a widely used package manager with over a million packages registered and millions of daily package downloads, with the field of web development being in high demand [21].

1.1 Background

A dependency in software development context is an external code library of variable size that provides the developed code with functionality that allows it to perform a specific task. When the code includes and utilizes a library, it becomes dependant on it, hence the name “dependency”. These code libraries are designed and developed in a way that tries to ensure maximal independency from the code that they are used in, because that way, they could be reused across multiple projects, therefore cutting out the need to repeatedly develop the same functionality all over again whenever a different project requires it. This fits under the paradigm of code reusability and is of utmost benefit to software developers [1, 2]. Of course, not all dependencies are entirely independent of the code environment they are used in, and because of that, they might rely on other dependencies either in form of programming language standard libraries or other libraries made by the initial developer or a 3rd party. In a software development projects dependencies that are relied upon by other dependencies are referred to as transitive, while a dependency that is first in a chain of potentially multiple dependencies, is called direct. In a software development project, the used dependencies form a branching structure that is often referred to as a dependency tree.

To further maximise the benefits that come with the use of code dependencies the developers are encouraged to share their libraries with each other in form of code packages. This means that developers can not only reuse their own code, but also the code of other people, which cuts the development time even more. On top of that, developers use package managers which allows them to save resources on costs that have to do with package storage and distribution by offering centralised repositories where their users can upload their packages for others to download. If a package is updated with a new functionality or a security fix, a new version will be uploaded by the developer to the repository, the integrity of which is usually reinforced cryptographically, but it mainly depends on a package manager. For example, the NPM package manager adds a PGP signature to package metadata and publicize it in its public PGP key on a service called Keybase, which maps data to encryption keys, in a publicly auditable manner.

Despite the benefits that the use of 3rd party dependencies can provide, there are also security risks associated with it. The fact that the code packages are developed by a 3rd party means that there is a risk for unknown content that must be checked for manually by the user or a trusted auditor, otherwise the users will be at risk of utilizing vulnerable packages. For example, the recent log4j2 CVE-2021-44832 vulnerability that allowed to remotely execute arbitrary code under certain conditions goes to show how a single vulnerable component can cause great damage to a project that is dependent on it if its vulnerability assessment is neglected. As projects grow the problem only gets worse with the addition of new direct packages into the dependency tree. On top of that, each time that one of these packages receives a hotfix or a security update, it must be updated, otherwise the user will be at the risk of leaving their project vulnerable [3]. This applies for both direct and transitive dependencies. Because of that, the auditing process for a single package is not a onetime occurrence, but a reoccurring procedure that must be repeated regularly.

The problem goes deeper when we consider that packages can also contain intentionally malicious code [4]. This means that a malefactor can attempt to either hijack already existing trusted packages by gaining access to a publisher’s credentials or proceed to disseminate their own malicious packages by masking them as packages with legitimate functionality or as already existing often used packages [6]. The activity of the latter approach recently increased as reported by software companies like Checkmarx, Sonatype and JFrog. They noted a significant increase in the number of malicious packages that were relying on dependency confusion and typosquatting to trick developers into installing those packages into their projects [11] [12] [13]. The packages themselves were primarily targeting the @azure scope by creating new malicious packages with the same name as already existing @azure scope package, i.e., instead of "@azure/core-tracing" they would name a package "core-tracing", omitting the scope name. The malicious code in said packages was mostly the same containing a payload which attempts to extract basic system information and environment variables via a Telegram API.

This thesis goes over packages with both intentionally and unintentionally placed vulnerabilities, as well as packages with other malicious content. The reason being that they pose a security risk, first and foremost, for the virtue of being a part of a code dependency supply chain whose effects can only be manifested if added into a dependency tree of a project [16].

1.2 State of the art

Today, a big contributor to 3rd party package safety is due to open source. By creating a platform where developers can openly access the source code of a given package and leave feedback, allows for swift vulnerability detection and remediation, and the probability of that increases as a package gets more popular. Developers can notice a vulnerable package and quickly notify other users by leaving feedback, and in cases when the content of a package is deliberately malicious, notify the moderator staff to take it down. And if a vulnerability is found in a package that is not maintained anymore, another user can clone and fix the code and publish it if it complies with any license that applied to the original package. The NPM registry also keeps track of additional package information like weekly downloads, number of issues, number of published versions, time of the latest publish and number of packages that are dependent on this package.

Developers have also come up with other solutions that attempt to mitigate the dangers that are associated with the use of 3rd party packages. For example, there is a tool called Dependabot that helps developers keep their dependencies up to date by automatically checking the dependency files in their git projects, and if any of the versions are not up to date, it will open a pull request so that the developer will be able to review the changes that come with a new version, in accordance with the specified configuration. This is sometimes referred to as version bumping. It can also temporarily disable updates for specific packages if the need arises. There is also an ability to automatically merge pull requests that are created by Dependabot for the sake of automation. This helps developers make sure their dependencies are up to date without spending resources on manually checking every dependency in the dependency tree for available and desired updates. But this also means that if no precautions are put in place, Dependabot might automatically update to a vulnerable or a compromised version, in turn compromising the entire project.

NPM has an in-built command called Npm-audit which goes over the project’s dependency tree and checks them against a list of known and reported vulnerabilities [5]. This sort of behaviour is not unique to NPM with other services providing an ability to access databases of documented vulnerabilities some of which often contain exclusive content [9]. If any vulnerabilities are found by Npm-audit, then the impact and appropriate remediation will be calculated. The remediations can be applied instantly if a -fix flag is provided to the command call. Otherwise, if no vulnerabilities were detected, the command will finish with the exit code 0. It relies on two audit endpoints to fetch the vulnerability information: Bulk Advisory and Quick Audit. For the Bulk Advisory end point NPM will generate a JSON payload with every dependency and its version from the project’s dependency tree and send it via POST to the default configured registry. Also, an omit config can be created to make sure that certain packages won’t be sent for vulnerability assessment. If any vulnerabilities are detected the response will contain advisory objects which are used to calculate the vulnerabilities of the dependencies. If NPM will not be able to connect to the Bulk Advisory endpoint, it will attempt to get the advisory data from the Quick audit endpoint, where the contents of the package-lock.json file will be sent with some additional metadata. So, in essence Npm-audit provides supply chain security by referencing a database of known vulnerable packages, but this means that this command will not be of use if a package contains a vulnerability that has not been discovered and documented yet.

There is also Snyk, which is an opens source application security and testing platform that is used for finding and remediating vulnerabilities in open-source libraries, codes, and containers. It has similar functionality to Dependabot that allows developers to version bump their project dependencies with auto-generated pull requests. It has an ability to generate and scan a project’s dependency tree for vulnerabilities and offer potential solutions in accordance with the severity of a vulnerability, based on databases of commonly used exploits and known vulnerabilities, but it is still prone to false positives when it comes to code analysis.

JavaScript frameworks like Angular and React also offer means by which the developed code can be exempt of commonly known vulnerabilities by attempting to handle cases that result in XSS (Cross-Site Scripting) or Injection vulnerabilities, by automatically sanitising inputs.

1.3 Problem to be solved

With all of this in mind, it can be derived that vulnerability assessment for 3rd party dependencies as it currently stands is heavily reliant on manual code auditing, either by the community review or by the developers who are directly connected to a given project. As automatic version bumping must open pull requests that need to be manually checked before merging to retain its effectiveness, or that the databases of known vulnerable package versions only hold information that was acquired from incident response or code review means that it is reasonable to focus on streamlining the process of 3rd party package auditing as it has proven to be time consuming yet most relevant, especially so in large projects. To achieve this, it is first necessary to determine relevant security practices of JavaScript developers so that the solution will prove to be beneficial rather than detrimental in saving time when checking for package safety. To determine the relevance of security practices, a survey will be issued to JavaScript developers. Based on the results of the survey, a software prototype will be developed.

2 Methodology

From everything that has been gathered in the previous chapter, in order to solve the problem that is stated by this thesis, being the problem of time consumption when it comes to manual 3rd party package auditing in a projects that use the NPM package manager, it has been concluded that a possible solution that could alleviate some of the time costs is a software prototype that will help developers to more efficiently pick out dependencies whose code is in need of a review to assess their safety. To develop this prototype, it is first important to determine what security concerns are currently relevant for JavaScript developers. To do that, the decided approach is to collect data on key security practices that modern JavaScript developers follow. The collected data will be interpreted and applied to the prototype's design. The data is going the be collected via open ended survey questions and then analysed to determine how it can be applied to the design phase of the prototype. The survey is designed and shared with survey takers via Google Forms. The software will be developed using node and JavaScript and will be published as an NPM package.

2.1 Survey

As stated previously, the main objective of the survey is to collect data on security practices that are currently relevant to JavaScript developers, analyse the data and extract information that will be used in the design stage of the software prototype. The extracted information will be used to determine what are primary concerns of developers and what can be done to address these concerns in an automated fashion. The survey consists of 6 questions in total, and they will be referred to as Q1 – Q6. Questions Q1 – Q5 aim to determine what developers focus on in a specific security context, while having Q6 as an open-ended question where the survey takers can leave additional thoughts on the topic and leave feedback, this question was not made mandatory, yet it managed to provide useful information regardless. The contexts which the rest of the questions were built around go as follows:

- Code safety practices
- Vulnerability awareness
- 3rd party package safety
- Use of code analysis tools

With Q6 covering additional thoughts and feedback and not strictly falling under any of the listed contexts. These contexts were chosen to make sure that the gathered data would cover different aspects of the developers’ security practices. Q1 which falls under “Code safety practices” tried to establish if any strict set of practices was followed to assure that the code is safe. With this data it will be easier to determine a strategy that the developed prototype would use to establish the safety of a given package. Q2 falls under “Vulnerability awareness” and tried to establish what vulnerabilities if any are of note for JavaScript developers. This data will help determine what aspect of any given package the developed prototype will be focusing on when determining its safety. Q3 falls under “3rd party package safety” and tried to establish that in an event of a decision to use a 3rd party dependency, how would it be determined that a package in question is safe to use. This data will help establish what the developed prototype can use as a reference to determine the safety of a package (Similar to Q2). Q4 – Q5 fall under “Use of code analysis tools”, with Q4 trying to determine if the developers used any code analysis tools, and with Q5 what drawbacks were noticed with their use. With this data it will be possible to determine what design choices should be made to assure that the developed prototype will be desirable to developers from the standpoint of setup and execution.

The survey questions were created primarily with the design of the software prototype in mind, and because of that the information inferred from the answers might hold less value if taken out of this context, which might call for their revaluation if used by any subsequent research on the topic of vulnerability assessment for 3rd party dependencies. The validity of the inferred information will be determined to an extent by the performance of the developed prototype. As of April 23rd, 2022, the survey has received 7 responses all of which were manually looked over, with varying levels of insight given by the responders. Some of the answers were short and minimalistic, while others were provided with additional elaborations, with 5 people answering Q6, providing additional thoughts. With 7 people who answered the survey in total, each person will be referred to as P1 – P7.

2.2 Summary of survey answers

By manually going over the answers we can already see some patterns emerging, like a preference to use frameworks as means to assure a degree of freedom from certain vulnerabilities (P1, P3, P4, P6), or wide preference to use Dependabot (P1, P2, P3, P6) as well as Snyk (P1, P5, P7) to check for vulnerable packages. Although some survey takers pointed out clear disregard for the safety of the 3rd party dependencies that they use in their projects by not checking for any indications of their potential vulnerability (P4, P5, P7). This section will go over every question, summarize the answers and describe the main conclusions made from them. The questions themselves are formulated as follows:

- Q1 “Do you follow any particular set of practices to assure that your JavaScript code is safe? If you do, please describe them. (Choice of frameworks, Linters, Data safety, etc.)”
- Q2 “What vulnerabilities do you keep an eye out for when developing software in JavaScript? (XSS, SQL Injections, etc.)”
- Q3 “When you decide to use a 3rd party package for your project, what do you check for to make sure that it is free of vulnerabilities and/or malicious content?”
- Q4 “Do you use any code analysis tools to assure that your code and its dependencies are free of vulnerabilities? If you do, please name them.”
- Q5 “If you answered the previous question, can you point out any flaws that those tools might have?”
- Q6 “Additional thoughts and feedback.”

2.2.1 Q1 Summary

Answers to the first question showed that survey takers take a diverse set of actions to affirm the safety of their code. It was mentioned that tools like Prettier (P1) and ESLint (P1, P2, P5, P6) were used, with the latter being an automated code analysis tool that scans code for bad programming practices like deep if statement nesting or unsafe function usage, while the former is an opinionated code style enforcing tool that was designed to settle debates regarding preferences towards tabulation, line breaks and variable declaration order [22]. The use of JavaScript frameworks like Angular (P4) and Meteor (P3) were mentioned as means to assure code safety, with Angular having in-built protections against vulnerabilities and attacks such as XSS, although it doesn’t cover application-level security issues like authentication and authorization, as well as Meteor having a need to pay extra attention as to not expose any server-side functionality, by checking for user permissions. Snyk (P1, P7) and Veracode (P2) were mentioned which are tools that provide automated source code analysis to detect vulnerabilities. Snyk is also used to automatically fetch urgent security updates for used dependencies. Dependabot (P1) which was also part of the answers to this question solves a similar task, by automatically updating versions of used packages. And finally, one of the survey takers (P6) said that they try to keep the number of dependencies in their project at its absolute minimum.

2.2.2 Q2 Summary

Answers to the second question showed that survey takers were wary of XSS (P1, P3, P4) and CSRF (P1, P3, P4) (Cross-Site Request Forgery) or other types of code injections and arbitrary code executions (P6) vulnerability. Cross-Site Scripting being a vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable application. It allows an attacker to circumvent the same origin policy, which is designed to segregate different websites from each other and execute arbitrary JavaScript, while Cross-Site Request Forgery is vulnerability that allows perpetrators to induce a user to make an action they do not intend to do. For example, issue an http request, but not necessarily pass any data to the malefactor. Also, while not a vulnerability in code itself, one survey taker (P1) showed concern for legitimacy of the NPM packages themselves, alluding to the fact that package managers by their design leave its users vulnerable to malicious packages and said that they exercise caution when picking new packages. Another survey taker (P5) mentioned keeping sensitive information out of source control in the answer to this question.

2.2.3 Q3 Summary

When asked about what they check for when assessing whether a used package has any vulnerabilities or malicious content, survey takers showed that for the most part they don’t look out for anything (P3 P4, P5 P7), with one of them (P3) mentioning that they might pay attention to NPM download counts rather than the content of the package and said that they pretty much use packages made only by trusted organizations. The same survey taker mentioned that they rely on Dependabot which warns them if they deploy a vulnerable version of a package. Number of downloads was also mentioned by another survey taker (P1) who on top of that said that they look out for the date of last update and any spelling mistakes to make sure they won't pull a typosquatted package. Other than using tools like Dependabot, Veracode and Npm-audit, general online feedback was mentioned (P2).

2.2.4 Q4 Summary

When referring to analysis tools, some survey takers mentioned Snyk (P1, P5, P7), Npm-audit (P3). There is also a mention of LGTM (P6), a code analysis platform for finding zero-days and preventing critical vulnerabilities, that utilises CodeQL which is an analysis engine used by developers to automate security checks [10]. The same person mentioned Prettier and ESLint. Most of the survey takers (P1, P2, P3, P6) said that they use Dependabot as a tool to ensure that their dependencies are free of vulnerabilities.

2.2.5 Q5 Summary

From the little feedback that the survey takers provided on the automated tools that they use (P1. P3, P7), first a survey taker (P1) mentioned that solutions like automated dependency vulnerability version checks from solutions like Snyk, rely on databases of known vulnerabilities, which means it's aware of the vulnerabilities that were found and patched, although other survey taker (P7) was satisfied with Snyk as a whole. Also, it does not check for obfuscated code or typos in package names. Another survey taker P3 saw it as an issue that even after Npm-audit finds a sever issue with a package, it still allows the developer to ignore the warning and incorporate the vulnerable package into their code. And lastly a survey taker (P5) was completely unaware of any shortcomings when it came to Snyk.

2.2.6 Q6 Summary

Significant feedback was provided from four of the survey takers (P1, P2, P3, P4) with the first one mentioning that if a tool for improving risk assessment should be created, it should consider the number of weekly downloads, frequency of updates, uglification or obfuscation of the code, and typos in the package names. Second survey taker expressed a concern with their lack of overall experience with JavaScript and their potential inability to express answers in a detailed manner. Third survey taker showed a general dissatisfaction with the security of JavaScript and how a potential reason for it might be the fact that the use of native JavaScript by the presence of different frameworks that all have their own security issues which leads to overall JavaScript security be fragmented. Fourth survey taker said that their company focuses more on securing the server side of their projects so that if an attacker manipulates the client side, they will not be able to access the data on the server.

2.2.7 Assessment of the answers

It is worth to note that the use of code formatting tools like Prettier and ESLint, which is indicative of the fact that code readability and avoidance of bad coding practices are of importance to developers and are even considered as a factor when it comes to code security. Use of CVE databases is a common factor in the use of Snyk, Dependabot and Npm-audit, but tools like Veracode that also rely on a database of known vulnerabilities has its own unique database that is constantly finding vulnerabilities in open-source libraries and are separate from CVE [9]. Snyk offers Static Application Security Testing or SAST for short, which allows to analyse the source code of an application for vulnerabilities in an automated fashion so that the developers wouldn’t have to do it themselves. It helps with things like saving passwords in clear texts or sending data over unencrypted connection. It provides different types of analyses that focus on finding different issues like compliance of configuration files with security policies, vulnerabilities to SQL injections, insecure data sources like files The use of frameworks is another point of interest as for multiple survey takers alluded to the fact that frameworks allow developers to be less concerned by certain vulnerabilities as an example how React handles unsanitized inputs. Although the response of another survey taker offers some more insight when it comes to framework use, as different frameworks might have different exploits and vulnerabilities which play as a detriment for security of JavaScript projects due to security being inconsistent across different frameworks, in addition to that another survey a taker pointed out to the fact that client-side code is in full control of a potential exploiter and because of this it is more important to secure the server-side, which was also a concern shown in the survey response that pointed out exposure of server-side functionality is worth of attention. Lack of opinionated action from the side of code analysis tools in a case where a severe vulnerability is found was also pointed out as a drawback. Opinionated action means that if for example a tool finds a use of a severely vulnerable package it will forbid the build of the code until the issue is addressed. This however might cause issues when new vulnerabilities are found, and the build block congests the development process. When it came to concrete vulnerabilities, XSS CSRF, SSRF, arbitrary code execution and remote code injections were of note. Code injection refers to an ability to remotely inject code into an application often because the application in question is written in a language that allows dynamic code evaluation at runtime. With legitimacy of the NPM packages also being a concern, Survey takers that reported looking out for specific tells of a vulnerable package mentioned things such as number of weekly downloads and the time of last updated can be indications of a secure package, as well as the identity of the package publisher. Things like uglification of code is also a concern for package security, although it is practiced by developers who want to retain their intellectual property by obfuscating the initial logic of their package, it also means that the actual functionality is unknown and might contain vulnerable or straight-out malicious code. Another concern regarding malicious content of a package is the names of the packages themselves as it is a well-documented occurrence that malefactors are resorting to typosquatting to spread malicious code by publishing packages with names resembling already existing popular packages in hopes that its users misspell the name when installing them.

As stated previously, the analysis of the survey answers was done with the design of the software prototype in mind, therefore it will reflect aspects that could be used by a software whose main purpose is to fix the problem of inefficient security assessment process for project dependencies. The results will be assessed to find realistically applicable attributes for the design process. The results of the analysis were represented as a list of underlying themes and is split into 3 categories which reflect a partial thematical overlap among their representatives, all of which are depicted on Table 1.

Table 1. Table of resulting themes separated into 3 categories

Project risk indicators	**Assessment and remediation approaches**	**Code vulnerabilities**
Used package information	Vulnerability databases reference	Cross-Site Scripting
Package publisher information	Automated source code analysis	Cross-Site Request Forgery
Used frameworks	Automatic security updates	Server-Side Request Forgery
Code linting	Opinionated software action	Remote code injection
Client-side input sanitization		Arbitrary code execution
Dependency code obfuscation		Credential exposure

2.3 Assessment of extracted themes

The assessment of the themes will go in the order of the columns of Table 1, starting with the headings representing categories, which can be looked at as overarching themes that encompass into themselves several subthemes which will be assessed for relevance and applicability to a developed software prototype. The first category “Project risk indicators” describes what aspects of a project in its entirety can be indicative of risks that it is subjected to, they will be applied as metrics to the developed software or disregarded if they are deemed unreliable or irrelevant. Next, the category “Assessment and remediation approaches” describe means by which dangerous elements of a project can be pinpointed and addressed, the subthemes from this category are to be used as guidelines for the behaviour of designed prototype if they are deemed feasible. And finally, “Code vulnerabilities”, is a list of extracted and implied to be relevant in the context of JavaScript development vulnerabilities that might be a focus of the developed prototype. The reason why this category was chosen to be separate from the “Project risk indicators” category because in the context of creating a software prototype that can automatically assess those vulnerabilities the logic of such software will primarily rely on static code analysis, the problems of which will be described in the Automated source code analysis subtheme.

2.3.1 Project risk indicators

It has been noted that the assurance of standardised code styling using **Code linters** is of noticeable concern when it comes to code development, as there is a claim that it ensures code readability and lessens the likelihood of mistakes from the side of developers that work on the code, especially so if the code is worked on by multiple people. Yet studies that were made on the topic remain inconclusive about the implications of code readability regarding code quality [14] [15], because of this, the use of code linting is deemed as an unreliable metric for assessment of project risk.

The **Use of frameworks** was a common theme in multiple survey responses, as it was stated that certain frameworks can address problems like input sanitization by doing it automatically on render. But a possibility of framework usage as vector that introduces more vulnerabilities was also presented. The research done on the topic confirms that frameworks that have built in security controls result in a decreased rate of vulnerability, but the research was only done with the XSS vulnerability in mind and covered specific scenario and hence cannot be used to make a broad claim about the improvement of security in a broader context [7]. This means that there needs to be more research done on the topic which fall outside of this thesis' scope, so framework use will not be a metric in the developed prototype.

Client-side input sanitization is mentioned in the survey responses and at first it might sound like a highly important indication of a project's security, since distrust for user input is generally agreed as a good coding practice, but what is important to note is that client-side code can be accessed and altered by the client as mentioned by one of the survey takers, and to address this point server-side validation is employed to assess the input data and determine if its valid and is safe for work. This means that input sanitization is virtually irrelevant when it comes to application safety, although it has its value in assuring quality user experience. This means that sanitization of user inputs will not be a metric in developed prototype.

Used package information implies information like weekly downloads, total download count, date of the last update and frequency of updates, number of versions published and number of packages that are dependent on it, all of which can be used to identify a level of trust that the community has towards a particular package, plus the exposure to the community makes it more likely that even if a vulnerability is present in a package, the vulnerability going to be promptly detected and reported. Package names can also be used to check them against popular packages with similar names to detect typosquatting, but it might be difficult to implement. Also checking package versions can be crucial for preventing a potential dependency confusion attack. In general, this means that used package information can be relied on to assure a degree of security and can be used as metrics for the developed prototype. Although information related to number of downloads can be considered unreliable due to how easy it is to manually increase a package's download count by using scripts or bots [24].

Package publisher information refers to information like date of registration, number of packages published and the reputation of published packages, number of stars that the publisher has on Github. Publisher information goes hand in hand with other contributors' information, like number of contributors and their activity, or even presence of trusted auditors.

While **Code obfuscation** is widely used to protect intellectual property of the developers [19], in the context of open-source dependency security assessment it impedes the ability of an auditor to accurately determine the business logic of a package and therefore makes the contents unknown for the developers that use them. To accurately ascertain the behaviour of the code, it first must be deobfuscated [23]. This means that code obfuscation of an NPM package can be treated as red flag that could indicate unaccounted for vulnerable logic or even malicious content, which means that code obfuscation can be used as a metric for the developed prototype.

2.3.2 Assessment and remediation approaches

Vulnerability database reference has shown to be a strong asset in the arsenal of code analysis tools, it allows an efficient lookup of known vulnerabilities to let the developer know if a used code is reliant on vulnerable dependencies as shown by tools like Dependabot and Snyk. The fact that there are already reliable tools in place to address a demand for automated security updates means that there is no need to steer the developed prototype into this direction.

Automatic security updates, in the same vein as previous subtheme it has already established quality tools that covers this need and will not be used to direct the design course of the developed prototype.

Automated source code analysis is a very powerful approach when it comes to vulnerability assessment and could be used to address the metrics of credential exposure and code obfuscation. It also can be utilized to look for all the vulnerabilities that were extracted by the earlier survey analysis Although it is important to note that reliable automated source code analysis is a very difficult thing to implement and as such might not be a directing force for the developed prototype because it might require a scope much larger than that of this thesis.

Opinionated software action refers to restrictive software functionality that is executed in accordance with an opinionated set of directives which can’t be defined by the user of a software in question. This behaviour was pointed out by one of the survey takers who was discontent by the fact that the Npm-audit command allowed to ignore severe security vulnerabilities in used packages despite its warnings. The survey taker was much rather making it so that it would outright block the building process of the code before the detected vulnerabilities are addressed. Similar functionality is also present in one of the tools that was mentioned by another survey taker who mentioned prettier. Prettier is an opinionated code formatting tool that automatically formats code in accordance with a predefined style which the user has very limited control over. Such software behaviour is beneficial for enforcing a specific standard for the sake of consistency and to avoid prolonged discourse on what style to use [22]. But in the context of a security assessment tool, the lack of flexibility might become a detriment to the development process and because of that opinionated software action will not affect the direction of the developed prototype.

2.3.3 Code vulnerabilities

As stated previously, the approach that is required to assess these vulnerabilities requires a high level of software design which does fall under the scope of this thesis and because of that, subthemes that fall under the theme of code vulnerabilities will not be covered by the business logic of the developed prototype. Although, credential exposure is a noteworthy theme due to a lack of found solutions when it comes to assessing project's dependencies for exposed credentials.

Credential exposure like embedded into the code API credentials or uploaded credentials to version control systems like Github is a sure way to compromise the security of an application and can be used as a reliable metric that reflect the overall security of an application. If a dependency that relies on the use of an API experiences a credential leak, this means that the credential could be discovered and used to compromise the security of the data that interacts with that dependency [17] [20]. Credential exposure can be a metric for the developed prototype. Also, it is important to be wary of packages that utilise untrusted APIs or other resources. If sensitive information is passed through these packages, it means that there is a potential for the confidentiality of the data to be compromised.

2.3.4 Assessment results

After assessing the themes received from the survey responses, it became apparent that some of the initial metric candidates were either too difficult to implement, or completely irrelevant to the process of assessing the potential risk factor of an NPM package, leaving only 6 metrics that made it to the prototype's implementation. The prototype itself is to be designed as a tool that would warn the user of certain flags that are associated with a specific metric to let the user know that the package needs closer inspection. The metrics that are deemed too difficult to implement into the current rendition of the developed prototype are still relevant and there is a possibility to add them to subsequent versions of the developed prototype in the future.

During the assessment of the results only metrics direct connected package information was deemed feasible for implementation to the developed prototype, those include frequency of package updates, which are supposed to be an indication of whether the package is receiving regular security updates. Then there is the number of dependencies a package uses; this metric was deemed important because the probability of one package being at risk in a project's dependency tree increases with the overall number of packages in it. Also, whether a package was declared deprecated by its publisher is another important thing that a package user should be notified of if they decide to use it, as it means that the package stopes receiving security updates. With the date of first publish, it can be determined whether a package is too new and lacks exposure which would be beneficial in finding faults in its security. Then there is the number of dependent packages, which indicates a level of exposure through how many people decided to add the package in question into their published NPM package, this establishes a chain of dependency which makes it more likely for its content to be looked at by a user. And lastly, a packages version can be indicative of a fact that it is being used to conduct a dependency confusion attack by having an unnaturally high patch or minor version which is needed to confuse automated update scripts to pull a compromised package.

The metrics that were too difficult to implement and subsequently excluded from the roster of the metrics that the developed prototype used mainly related to the packages source code, and therefore required the use of automated static code analysis to be detected. For example, to determine what is considered sensitive information would be a difficult endeavour by itself, and then there is additional problem of detecting if this information is passed through an untrusted API or any other resource, whose trustworthiness assessment is an additional problem, and therefore it was decided to concentrate on more package information. The problem of static code analysis also applies to determining code obfuscation and ability to execute system commands. Detection of prior incident and vulnerabilities associated with the package is doable by refereeing to the CVE database but was ultimately decided to be too big of a project to implement into the developed prototype.

When it comes to metric that were not deemed relevant, it was concluded that metrics that are connected to contributors and publishers like their activity, connection to trust corporations or existence of reliable auditors are ultimately too unreliable, because firstly, it is very difficult to establish whether an arbitrary NPM package has an auditor, and then whether that auditor is trusted enough for it to matter for a package user. Number of contributors is too much of an arbitrary metric which does not accurately reflect the merits of their contributions. Tracking contributor activity creates a need to create a profile for every contributor, whose activity then must be constantly evaluated, which is not cost effective. It has also been deemed not necessary to pay attention whether a package is connected to any trusted corporation because a problem similar to contributor profiling arises and therefore not feasible. Package licenses were not deemed relevant for the development of the prototype because they only reflect limitations to their consumption and distribution, and not their actual performance. And lastly, it was decided not to focus on ways to determine presence of tests and their quality for packages, because while it could be a good indication of low risk, ultimately developers might not add their test suites to their git repositories, and even if they did, assessment of test quality of every package in a dependency tree is too big of a task for the scope of this thesis. Table 2. shows how the final list of metrics was formulated.

Table 2. Table of final metrics separated by feasibility

Feasible	**Need more work**	**Not feasible**
Update frequency	Putting sensitive information through a foreign API.	Number of contributors
Number of dependencies	Passing sensitive information through an unproven http endpoint.	Contributor's activity
Deprecation of a package	Connecting to unproven resources	Connection to a trusted corporation
First publish date of the package	Ability to execute code from a string in the code, like using eval.	Existence of reliable auditors
Number of dependent projects	Use of obfuscation	Licences that apply to the package
Suspicious semantic versions	Prior incidents or vulnerabilities connected to the package	Misspelled package names that resemble popular package names
	System command execution	Download count
		Presence of unit and integration tests

All in all, for the development process of the prototype, only the metrics associated with the package information will be used as their procurement does not call for the use of static code analysis nor difficult trust assessment. There will be an attempt to utilize static code analysis for future versions of the developed prototype to expand its list of metrics.

It is important to note that extensive research must be done in the future to determine optimal default values for all the metrics e.g., how many dependencies must a package have to warrant a warning of its potential risk. To address this point, the developed software will be made with configurability in mind. Its users will be able to determine what thresholds the metrics need to exceed to issue a warning and have an ability to customize it for a specific package. This feature of the prototype will beneficial if a user will have a reason to exclude certain packages from the risk assessment of their project.

2.3.5 Detriments to the validity of the analysis

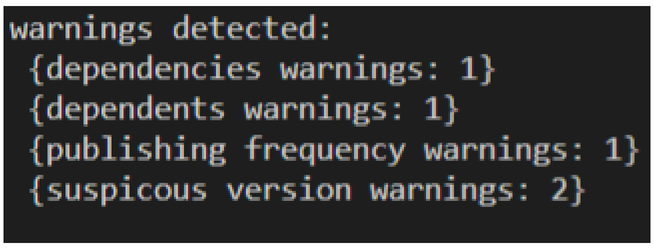
A low number of survey takers means that the collected data does not accurately represent the opinions of the JavaScript developers as a whole and because of that, the final analysis might reflect this inaccuracy.

3 Developed prototype

The final design of the prototype takes shape of a CLI (Command Line Interface) tool that presents its users with a list of warning that reflect presence of risk factors in the dependency tree which the user can subsequently use in their risk assessment, for example the user does not need to manually check for whether a given package is deprecated, the tool will automatically collect the status on package deprecation for every package in the extracted dependency tree and then warn the user which ones are deprecated.

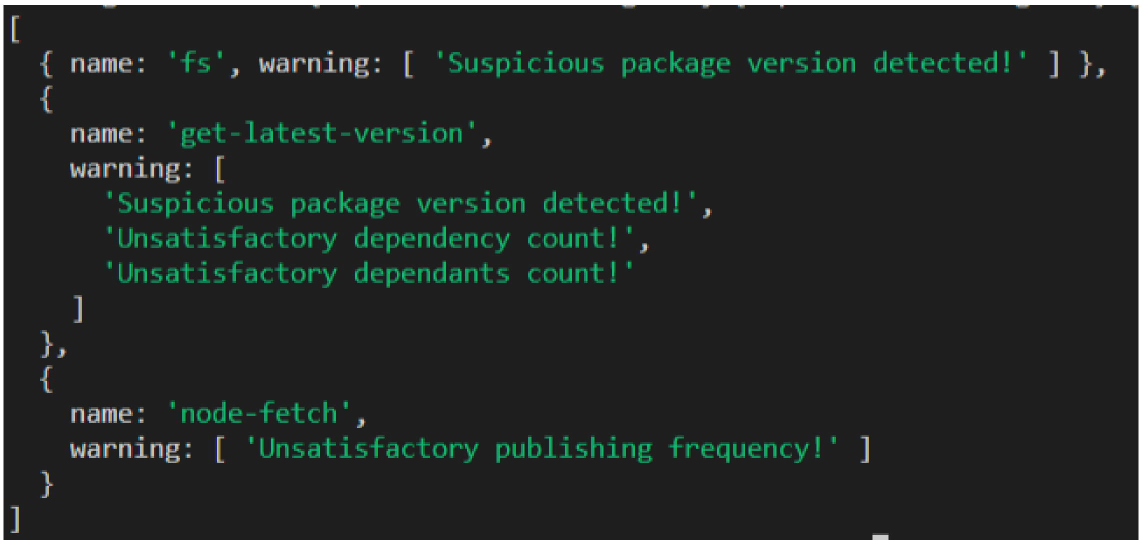
To extract the dependency, the tool goes over the **package-lock.json** which is normally automatically generated by NPM when the command **Npm install**, which by itself might be risky since some packages might run preinstall scripts during the installation process, which creates a risk for execution of malicious commands. For this reason, it is recommended to run the tool before the packages are installed. The tool can create a **package-lock.json** without installing the packages, but the process for procuring the list of versions for the used packages will take longer. If the user is certain that the installation of the packages will not cause harm, they can run the software with preinstalled packages, to safe time.

To run the tool, one must input the command **seersec,** while located in the root directory of the project, the output will be the number of warnings each metric had triggered as seen on Figure 1., and if the tool will receive a **–verbose** or **-v** flags the list of all warnings will be displayed as seen on Figure 2.



```
warnings detected:
 {dependencies warnings: 1}
 {dependents warnings: 1}
 {publishing frequency warnings: 1}
 {suspicous version warnings: 2}
```

Figure 1. Default output of the developed prototype



```
[
  { name: 'fs', warning: [ 'Suspicious package version detected!' ] },
  {
    name: 'get-latest-version',
    warning: [
      'Suspicious package version detected!',
      'Unsatisfactory dependency count!',
      'Unsatisfactory dependants count!'
    ]
  },
  {
    name: 'node-fetch',
    warning: [ 'Unsatisfactory publishing frequency!' ]
  }
]
```

Figure 2. Verbose output of the developed prototype

3.1 Tool configuration

The behaviour of the tool can be configured with a JSON file. The changes will affect the sensitivity of the tool to defined metrics. The JSON file itself represents an object with 3 fields.

- Default
- Excluded
- Custom

With **default** being a configuration object that defines what thresholds each metric needs to exceed for the tool to notice an undesirable result, this configuration will apply to every package that is not covered by the **excluded** or the **custom** fields. **Excluded** being an array that represents names of packages that will be ignored by the tool during the assessment of the metrics. And **custom** being an array of configuration objects for each individual package that the tool user deems necessary to specify, by default it is empty. The configuration objects in the **custom** array are structured similarly to the **default** configuration object, but they also have an additional field to represent a name that is used to determine what package is associated with a given custom configuration. If the configuration file is not present it will be auto generated with predefined default values, with **custom** and **excluded** fields empty. If one of the configuration object fields is defined as **null**, then that means it is going to be ignored completely during the runtime of the tool, and therefore be omitted from the final output. In the final rendition of the tool, the configuration file must be called **seersec-config.json**.

```
{
    "default": {
        "dependencyThreshold": "10",
        "dependentsThreshold": "1",
        "publishFrequencyThreshold": "14",
        "versionTrustThresholds": {
            "majorVersionThreshold": null,
            "minorVersionThreshold": null,
            "patchVersionThreshold": null
        },
        "firstPublishedThreshold": "3",
        "assessDeprication": "false"
    },
    "excluded": ["eslint", "get-latest-version"],
    "custom": [
        {
            "name": "angular",
            "dependencyThreshold": "10",
            "dependentsThreshold": "10",
            "publishFrequencyThreshold": "20",
            "versionTrustThresholds": {
                "majorVersionThreshold": "9",
                "minorVersionThreshold": "99",
                "patchVersionThreshold": "999"
            },
            "firstPublishedThreshold": "22",
            "assessDeprecation": "false"
        }
    ]
}
```

Figure 3. Structure of the prototype’s configuration JSON file

The values for *firstPublicationThershold* and *publishFrequencyThreshold* are taken in days.

3.2 Acquisition of metrics

To retrieve most of the metrics it was sufficient to use the Npm-view functionality which allows to pull basic package information from the NPM registry via the CLI. However, to get the information about the number of dependent packages that is only displayed on the official NPM package web page, it was necessary to scrape the page as it is not offered by the Npm-view functionality.

3.3 Assessment of the developed prototype

The tool managed to consistently procure NPM package information and check it against a set of metrics using a configuration file as a reference point. If a configuration file is not present it will be automatically generated. Logic that is responsible for excluding packages from the assessment and logic that determines custom thresholds for specific packages are working well and did not show any problems. A noteworthy observation is that the process of procuring metrics in accordance with the specified dependency list takes a significant amount of time. On average for a project with 21 dependencies it took the developed tool approximately 1 minute and 50 seconds to finish its execution, meaning it takes around 5 seconds for the tool to assess whether a single dependency contains any risk encompassing factors. Currently the tool only assesses few risk factors, and because of that covers relatively little ground, meaning the scope of the tool as it is now is limited.

The developed tool has room for improvement, primarily when it comes to the number of risk factors that it is currently covering, in the future when this software will be able to cover static code analysis and profile databases it will be able to expand its scope significantly, by procuring more information associated with the package, like peculiarities of its source code or profiles of its contributors. On top of that, the execution time of the tool can be improved through optimisation of its business logic and implementation of a database that could hold pre-collected package information for quicker retrieval speeds.

4 Summary and conclusions

During this thesis a survey of security practices targeted at JavaScript developers was conducted based on the information extracted from the answers an attempt to develop a software prototype was made. The software prototype ended up being a CLI tool which aimed to decrease the amount of time it takes a developer to assess a presence of certain risk factors in their project's dependency tree. The software managed to reliably issue warnings of risk factor presence in accordance with a given configuration, making it flexible although narrow in scope. In the future, the developed tool can see improvements in its scope by introducing an ability to statically analyse the source code of the packages to detect things like code obfuscation and system command execution, as well as improvement in performance through hosting a database of package metrics for quicker reference. A noteworthy observation that happened during the process of making this thesis was a lack of automated solutions found for the problem of exposed credentials in 3rd party dependencies, the ramifications of which are yet to be fully assessed and the topic itself requires more research in and of itself. In general, this thesis provides minimal insight on the topic of risk assessment for 3rd party packages, but still can be used as basis for future research as this topic is of high relevance in the current development landscape. The result of this thesis is a software that can check a dependency tree of any NPM project against a list of metrics to determine whether a particular dependency is in need of auditing by issuing warnings of presence of risk factors that coincide with a user's configuration.

References

[1] Enrique Larios-Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, Georgios Gousios. "Selecting third-party libraries: The practitioners' perspective", 2020, doi: arXiv:2005.12574

[2] Manuel Sojer, Joachim Henkel, "Code Reuse in Open Source Software Development: Quantitative Evidence, Drivers, and Impediments". 2010, doi: 10.17705/1jais.00248

[3] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yang Liu, Yijian Wu, "An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects", 2020, doi: arXiv:2002.11028

[4] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, Laurie Williams, "What are Weak Links in the npm Supply Chain?", 2022, doi: arXiv:2112.10165

[5] npm, Inc., "Documentation for npm-audit", Accessed: May 16 2022, Available: https://docs.npmjs.com/cli/v8/commands/npm-audit

[6] Gabriel Ferreira, Limin Jia, Joshua Sunshine, Christian Kästner, "Containing Malicious Package Updates in npm with a Lightweight Permission System", 2021, doi: arXiv:2103.05769

[7] Ksenia Peguero, Nan Zhang, Xiuzhen Cheng, "An Empirical Study of the Framework Impact on the Security of JavaScript Web Applications", 2018, doi: 10.1145/3184558.3188736

[8] Red Hat, Inc., "What is an API?" (October 31, 2017), Accessed: May 16 2022, Available: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces

[9] Veracode, "Explore the Veracode Vulnerability Database", Accessed: May 16 2022, Available: https://docs.veracode.com/r/Explore_the_Veracode_Vulnerability_Database

[10] CodeQL, "About CodeQL", Accessed: May 16 2022, Available: https://codeql.github.com/docs/codeql-overview/about-codeql/

[11] Jossef Harush, "A Beautiful Factory For Malicious Packages" (March 28 2022), Accessed: May 16 2022, Available: https://checkmarx.com/blog/a-beautiful-factory-for-malicious-packages/

[12] Andrey Polkovnychenko, Shachar Menashe, "Large-scale npm attack targets Azure developers with malicious packages" (March 23, 2022), Accessed: May 16 2022, Available: https://jfrog.com/blog/large-scale-npm-attack-targets-azure-developers-with-malicious-packages/

[13] Ax Sharma, "This week in malware—400+ npm packages target Azure, Uber, Airbnb developers" (March 25, 2022), Accessed May 16 2022, Available: https://blog.sonatype.com/this-week-in-malware-400-npm-packages-target-azure-uber-airbnb-developers

[14] Andrea Capiluppi, Cornelia Boldyreff, Karl Beecher, Paul J. Adams, "Quality Factors and Coding Standards – a Comparison Between Open Source Forges", 2009, doi: 10.1016/j.entcs.2009.02.063

[15] Talita Vieira Ribeiro, Guilherme Horta Travassos, "Who is Right? Evaluating Empirical Contradictions in the Readability and Comprehensibility of Source Code", 2017, Accessed: May 16 2022 [Online], Available: https://www.researchgate.net/publication/321085627_Who_is_Right_Evaluating_Empirical_Contradictions_in_the_Readability_and_Comprehensibility_of_Source_Code

[16] Marc Ohm, Henrik Plate, Arnold Sykosch, Michael Meier, "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks", 2020, doi: arXiv:2005.09535

[17] Nikita Skovoroda (ChALkeR), "Do not underestimate credentials leaks" (May 12, 2018), Accessed May 16 2022, Available: https://github.com/ChALkeR/notes/blob/master/Do-not-underestimate-credentials-leaks.md

[18] Liran Tal, "Alert: peacenotwar module sabotages npm developers in the node-ipc package to protest the invasion of Ukraine", March 16, 2022, Accessed: May 16 2022, Available: https://snyk.io/blog/peacenotwar-malicious-npm-node-ipc-package-vulnerability/

[19] Chandan Kumar Behera, D. Lalitha Bhaskari, "Different Obfuscation Techniques for Code Protection", 2015, doi: 10.1016/j.procs.2015.10.114

[20] Guy Podjarny, "Keeping your Open Source credentials closed"(December 14 2015), Accessed: 16 2022, Available: https://snyk.io/blog/leaked-credentials-in-packages/

[21] Erik DeBill, "Modulecounts", Accessed: May 16 2022, Available: http://www.modulecounts.com/

[22] Prettier, "Why prettier", Accessed: May 16 2022, Available:https://prettier.io/docs/en/why-prettier.html

[23] Mister-Hope, Vuejs issue #7054 on github, Accessed: May 16 2022, Available: https://github.com/vuejs/vue-cli/issues/7054

[24] Andy Richardson, "How I exploited NPM downloads... and why you shouldn't trust them" (March 16, 2021), Accessed: May 16 2022, Available: https://dev.to/andyrichardsonn/how-i-exploited-npm-downloads-and-why-you-shouldn-t-trust-them-4bme

[25] OpenJS Foundation, "About Node.js", Accessed: May 16 2022, Available: https://nodejs.org/en/about/

[26] GitHub, Inc., "About pull requests", Accessed: May 16, Available: https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests

[27] Refsnes Data, "Introduction to SQL", Accessed: May 16 2022, Available: https://www.w3schools.com/sql/sql_intro.asp

[28] Peter Loshin, Alexander S. Gillis, "command-line interface (CLI)", Accessed May 16 2022, Available: https://www.techtarget.com/searchwindowsserver/definition/command-line-interface-CLI

[29] Kim Lewandowski, Bentz Tozer, "Beware the Package Typosquatting Supply Chain Attack", Accessed: May 16 2022, Availalble: https://www.darkreading.com/vulnerabilities-threats/beware-the-package-typosquatting-supply-chain-attack

[30] Dhiyaneshwaran, "Dependency Confusion", Accessed: May 16 2022, Available: https://dhiyaneshgeek.github.io/web/security/2021/09/04/dependency-confusion/

[31] Abi Tyas Tunggal, "What is a CVE? Common Vulnerabilities and Exposures Explained", Accessed: May 16 2022, Available: https://www.upguard.com/blog/cve

[32] PortSwigger Ltd., "Cross-site scripting", Accessed: May 16 2022, Available: https://portswigger.net/web-security/cross-site-scripting

[33] PortSwigger Ltd., "Cross-site request forgery (CSRF)", Accessed: May 16 2022, Available: https://portswigger.net/web-security/csrf

[34] Jeff Petters, "What is PGP Encryption and How Does It Work?" (April 6 2020), Accessed: May 16 2022, Available: https://www.varonis.com/blog/pgp-encryption

[35] Semmle Inc., LGTM website, Accessed: May 16 2022, Available: https://lgtm.com/

[36] GitHub, Inc., "About CodeQL", Accessed: May 16 2022, Available: https://codeql.github.com/docs/codeql-overview/about-codeql/

[37] Refsnes Datah, "What is JSON?", Accessed: May 16 2022, Available: https://www.w3schools.com/whatis/whatis_json.asp

[38] Npm Inc., “About npm”, Accessed May 16 2022, Available: https://docs.npmjs.com/about-npm

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Survey answers

Question 1. Do you follow any particular set of practices to assure that your JavaScript code is safe? If you do, please describe them. (Choice of frameworks, Linters, Data safety, etc.)

- I use code formatting tools (Prettier) to keep the code readable, ESLint to detect code smell (there are some rules which forbid unsafe usage of functions, etc), as well as stuff like dependabot and snyk to automatically get security fixes.
- Eslint and Veracode
- Not really - other than basic practices like sanitizing all user input, not exposing server-side functions (might be a framework specific potential vulnerability, we use Meteor) are ones that come to mind. Always check calling users permissions for whatever is being requested.
- Angular as a framework, always encrypt the password
- Eslint
- proven frameworks, limit dependencies to absolute minimum, linters
- Only snyk

Question 2. What vulnerabilities do you keep an eye out for when developing software in JavaScript? (XSS, SQL Injections, etc.)

- I guess XSS would be the only relevant one. It is usually handled by the framework in use, though. For example, React automatically sanitizes all input upon render. The only thing I really keep an eye for would probably be legitimacy (security) of NPM packages. I usually am careful whenever installing new ones.
- Not experienced enough to answer
- XSS, user input, parameter "spoofing" (I'm unsure of the term but manually calling server methods from an external client, with bad data, like to fetch another users data from a DB call if the method naively takes on a userId or email as a parameter)
- XSS, CSRF

- keeping secrets out of source control
- Remote code injections, Arbitrary code executions and others
- None

Question 3. When you decide to use a 3rd party package for your project, what do you check for to make sure that it is free of vulnerabilities and/or malicious content?

- I look at:
    1. Date of the latest update. If a package's last update was over a year ago, then it's a red flag as it could mean that the package could include a security vulnerability that's only recently been discovered.
    2. Number of weekly downloads is a good metric by which one could judge the overall trust in the package.
    3. Also it's important to make sure that I spelled the package name correctly (i.e., one could write `eslitn` instead of `eslint`, which hackers could abuse to inject malicious code into the project by deliberately registering the package with a typo)
- Veracode/Online Feedback
- We mainly use official ones created by orgs, like aws-sdk. "3rd-party" packages we use aren't really checked, except the download count on npm. We do have dependabot I believe it's called on Github, that warns if we deploy a vulnerable package or version of a package.
- No
- Nothing
- Npm audit, CVE databases
- Nothing

Question 4. Do you use any code analysis tools to assure that your code and its dependencies are free of vulnerabilities? If you do, please name them.

- Snyk and dependabot for my GitHub repositories.
- Veracode/Dependabot
- Dependabot security on Github which automatically issues, and NPM audit.
- No
- I think our org uses Snyk

- ESLint, LGTM, Prettier, Dependabot
- Snyk

Question 5. If you answered the previous question, can you point out any flaws that those tools might have?

- Snyk uses a database of known vulnerabilities, which means that it only knows about vulnerabilities/fixes which were already known and patched. As far as I know (citation needed), it doesn't do static analysis of the code of any of the packages to weigh stuff like uglification, weekly downloads (to measure trust that other devs have for this package), typos, etc...
- Unsure
- NPM audit is too "soft" - it should block building of a project if there's any severe issues with a used package. As it stands, it can be and sometimes is ignored.
- -
- no idea
- -
- It is fairly ok

Question 6. Additional thoughts and feedback.

- I think if one would want to create a tool that does surface analysis of the package to provide a score, the following points should be considered (the list is not exhaustive!!):
    - 1. Weekly downloads of the package
    - 2. Frequency of updates (security and feature updates)
    - 3. Uglification
    - 4. A check for typos

  * Maybe check against a db of most frequently used packages and apply an algorithm which would notify the developer when they attempt to install a package that is super similar to the a commonly used one. (e.g., if the developer tries to install tslitn, the software could notify him about the typo)
- Still a relatively new JS/TS dev so not able to give highly detailed answers.
- Security for JS in general is god-awful. I'm not sure why, maybe it's the fact that no one does native JS, it's all one of a dozen different massive frameworks, all

with their own vulnerabilities and exploits, leading security analysis to maybe be fragmented?

- For my company we make sure the server side is more secure so that if the attacker manipulates the client side they still can't access the data from the server
- your questions make me uncomfortable
- -
- -