A One-Sided Tale: Investigating a few Code Dependency Risks

Funmilayo Olaiya
Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
folaiya@uwaterloo.ca

Anuradha Kulkarni
Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
av2kulka@uwaterloo.ca

***Abstract*—In this work, we investigated two different kinds of risks, namely vulnerabilities and license compatibility, which are associated with code dependencies within specific JavaScript projects. We wanted to look into and examine how carefully these risks came to be, what triggered them, why they exist, and what their actual overall impacts are. We thoroughly examined fifteen open-source JavaScript projects and discovered a total of about 75 dependency risks that posed specific vulnerabilities across all of them. We also observed that there were no license compatibility risks associated with these projects and their package dependencies. The most interesting finding amongst the many peculiar things we discovered is that the majority of these risks were not introduced into the project directly but rather indirectly because one project's package dependency might likely depend on another package dependency, which may ultimately result into something is known as-dependency hell. Future research can look into why so many packages and projects still rely on other package dependencies that pose certain risks, even though many of these dependencies have not been maintained for at least six months and more.**

***Index Terms*—software engineering, vulnerabilities, open-source, code dependencies, license compatibility**

I. Introduction

In software engineering practices today, it is safe to assume that no open-source (OS) project in software engineering does not depend on at least one type of dependency, whether it is through some form of third party code which can be classified as a project by itself that serves its own functionalities, and is freely available for re-use, or from one that a project's author develops internally and specifically for that project alone [1].

Majority of the dependencies used by many open-source projects are either from third-party or external projects or code. We can describe a *code dependency* as additional code that a programmer wants to call as this allows a developer to avoid repeating a piece of work that has already been done by another programmer somewhere, so as to save time, resources and other factors [2].

The inspiration for this project came from one of the MSR challenges for 2023 [3], which called for researchers to examine the advantages and disadvantages of three forms of software supply chains in World of Code which are; *author-code knowledge transfer, code dependencies, and code copying*. Because each software supply chain is a vast entity that may actually require multiple approaches to be able to answer some questions in regard to them, we opted to simplify it down to measuring risks only in one software supply chain, which is *code dependencies*.

The dataset we used as a starting point to determine which specific JavaScript projects we needed to investigate is called World of Code, aka WoC [4]. WoC can be characterised as a dataset that includes a large repository of open-source software, ranging from individual projects or classroom assignments to OS software from different organizations or individuals. We chose JavaScript as our language of choice for the projects we chose since we were very familiar with it, its *npm* or *yarn* package manager, and how *package.json* really works, as this will inform us on how to conduct evaluations in an expert manner. Furthermore, GitHut [5] reports that JavaScript is the most commonly utilised language for OS projects.

According to NIST [6], there has been an executive order (EO) to strengthen the country's cybersecurity in order to recognise and reduce security risks throughout the software supply chain. Practically everyday, new software vulnerabilities are found that have already done substantial damage [7] in regards to to various products, businesses and other many factors. And this became one of the main sources of inspiration for us to enable us evaluate risks within JavaScript projects.

Furthermore, we evaluated each JavaScript project using the Snyk tool [8], which produced reports for the vulnerabilities found. This tool was especially useful because it not only generates a report of the risks but also provides a detailed account of certain processes, such as identifying what dependencies probably caused the vulnerabilities, categorising the severity of the vulnerabilities, pulling the CVSS score [9] of each vulnerability, and obtaining its data facts from its own database called the *Snyk Vulnerability Database* [10]. Snyk also claims that their database contains *many non-CVE vulnerabilities* that are derived from numerous additional sources, significantly above the scope of CVE vulnerabilities [11]. To understand the effects of these discovered risks, we also calculated a form of risk analysis using the OWASP methodology [12], but we are aware that this form of risk analysis is debatable because there are many different approaches to risk analysis, and none of them can be categorised as a ground truth.

Our findings typically focus on direct and indirect dependencies, which is because the bulk of the risks we discovered in the projects that we investigated were introduced by the dependencies of the installed dependencies in a certain project rather than directly by a project's own installed dependencies.

In terms of licence compatibility, we discovered that the projects and their dependencies are all compatible. Given that this is primarily an investigative and exploratory study, we only address two significant research questions;

- **RQ 1**: What risks are involved with code dependencies and why do they exist?
- **RQ 2**: What are the likely impacts of these risks?

II. Background

Software dependencies have the advantage of enabling developers to produce software more quickly by using earlier efforts. Over the past two decades, software dependencies have revolutionized application development, but they also pose risks that are usually disregarded [13].

One of the primary goals of this article is to increase risk awareness and promote further research into potential solutions. In general, software dependencies are categorized into two types - **Direct Dependencies** (Direct calls from libraries or packages in your code) and **Indirect dependencies** (Libraries or packages are called by your dependencies. These are dependencies that depend on other dependencies) [13].

There are numerous causes of dependency issues, however the following are some of the most common:

1) Poorly Written Code:: In general, it's best to choose well-known libraries that other developers have recommended, but it doesn't mean the codes for these libraries are well-written.

2) Lack of Code Maintenance: When someone produces some kind of code and then entirely abandons it, never updates it, or stops receiving contributions from others, this occurs. Developers who employ these kinds of codes expose their projects to costly risks.

3) Poor Documentation: On rare occasions, a program's functionality may not be well-documented or described. And a developer using this type of code in a project may be hindered by these documentation gaps which might pose certain risks in the project.

Further, in this section, we first introduce two risks associated with the Software Dependencies - security vulnerabilities, and license compliance issues. Then, we discuss the existing work related to the identification and impact of these risks in software engineering.

A. Security Vulnerabilities

Software dependencies can result in a variety of different kinds of security vulnerabilities:

1) Outdated and Malicious Dependencies: Software dependencies could become security risks if they are not frequently maintained. Attackers might use these risks as a springboard for attacks, data theft, or unauthorized system access. Moreover, hackers have the capacity to infect legitimate software dependencies with malware. Attackers can get access to systems or networks by taking advantage of these dependencies that developers unintentionally include in their programs.

2) Supply Chain Attacks: Attackers can compromise the software supply chain by inserting harmful code into relied-upon software. This can be done by breaking the security of a third-party service or a software vendor.

3) Configuration Issues: Incorrectly configured dependencies can result in security vulnerabilities. An incorrect setup database dependency, for instance, could allow attackers access to confidential data.

B. License Incompatibility

Open-source software (OSS) licenses set forth the conditions that must be followed to reuse, distribute, and modify the software. In addition to generally used licenses like the MIT License, developers are also permitted to create their licenses (called custom licenses), whose specifications could be more flexible [14].

It is challenging to understand licenses and their compatibility because there are so many different types of licenses available. To avoid costs and legal issues, licensing compatibility must be ensured when integrating third-party packages or reusing code that has licenses [14].

License incompatibility describes conflicts between several licenses within the same project. To facilitate software development, developers frequently need to incorporate numerous third-party OSS into a single project. An open-source project may therefore need to abide by several licenses. However different rights and duties are governed by several permissions and exemptions [14].

III. Related Work

A paper written by Russ Cox [2] on software dependencies in general and its foundations was a valuable source of learning with respect to our project. The author examines the difficulties in handling dependencies in software development and provides some useful solutions. The study acknowledges from the start that dependencies are a fundamental component of modern software development, but it also points out that they may be a considerable cause of frustration and dissatisfaction. Furthermore, it illustrates a variety of common dependency issues, such as version conflicts, unexpected upgrades, and security flaws [2].

The paper [15] written by Barry W. Boehm examines the value of risk management in software development and offers a framework for identifying, assessing, and managing risks throughout the development process. Boehm begins by defining risk as the "possibility of loss or injury" before going on to state that risks can arise from a variety of sources in software development, including workers, technology, and external factors like market situations. Due to this, we developed a new evaluation metric to quantify the risks connected to software dependencies.

One of the most crucial aspects to take into account is the impact of these risks. The effectiveness of solutions for recovering software architecture is examined in research by Thibaud Lutellier et al. [16]. According to Lutellier et al., software architecture recovery is a crucial stage in software maintenance since it aids engineers in understanding the layout and structure of pre-existing software systems. The authors begin by discussing the importance of code dependencies in software systems and how they may impact the restoration of software architecture [16].

A framework is provided by Mark Keil et al., and in their work [17] for detecting and managing risks in software projects. The authors emphasise that effective risk management is essential for software project success since software development is a complex, dynamic process with many interrelated components. The four primary steps of Keil's methodology are risk identification, risk assessment, risk response planning, and risk monitoring and management.

The paper written by Yuxing Ma et al. [4] describes a mechanism for extracting data from open-source version control systems (VCS). The World of Code platform enables researchers to collect, assess, and analyse huge volumes of VCS data from multiple open-source projects. The authors draw attention to the fact that open-source VCS data provides significant information that can help with comprehending the evolution of software systems, the behaviour of software developers, and community dynamics.

IV. Data Collection

While we were only interested in OS projects with *package.json*, we used the WoC dataset to filter out only JavaScript projects. The projects must also be somewhat well-used and kind of meets or fulfils a commercial need. While this is one of the other sides of WoC, where it normally comprises of all OS projects regardless of what kinds of projects they were, one challenge we faced was having to sort through a plethora of diverse random projects.

Furthermore, in order for us to use the Snyk tool over each JavaScript project we discovered, we cloned each of them, and we entered every piece of information from the various reports we received from Snyk into an excel file.

We specifically noted *the dependencies posing these risks, the kind of vulnerabilities they introduced, and the severity rating Snyk gives to each risk*. In addition to the information Snyk provided, we recorded if the found risky dependencies are present in the *package.json* file of each project. We also noted the results of deeply inspecting each specific dependency that contributed to the risk as well as the risk's overall impact.

V. Methodology

A. Selecting the JavaScript Projects

We decided to investigate just 15 projects because of time constraints and the level of depth we would have to get through with each of these projects so as to be able to get substantial results. We used the *grep* function on the WoC ssh server to bring up JavaScript projects with package.json files. And out of the hundreds that came up, we decided to make our pick from them.

B. Finding Vulnerabilities using Snyk

One way to utilize the Snyk tool is to follow a public project on GitHub through the Snyk dashboard to get a summary of vulnerabilities that are currently present. We chose instead to clone each project, then install Snyk on our local terminal [18], and execute specific commands to generate a vulnerability report for us. We used the ***synk monitor*** command to create a link to a report for each project. Also, in order to generate a report for projects with most or all of their dependencies listed under *devdependencies* in package.json, we had to run ***snyk monitor –dev***.

C. Inspecting the Risky Dependencies

We thoroughly checked all of the risky dependencies found in Snyk's report, and it was good to find that Snyk identified several dependencies that were likely to cause the risks for each project. We examined these dependencies manually by looking for them in a project's package.json file.

Alternatively, if they are not present in a project's package.json file, but rather visible in the package-lock.json or yarn.lock file, we proceeded to clone some of the dependencies (that introduced the risk(s)) in order to inspect their respective package.json files to determine whether they had a detected risky dependency installed.

D. Analysing the Licenses for License Compatibility

We carefully and manually examined about 770 package dependencies among the 15 js projects so as to determine if their licenses are all compatible or not.

E. Calculating the Impact of the Risks

Finding vulnerabilities is crucial, but it's equally important to be able to calculate the risk to the organization. We have referred to a well-known OWASP Risk Rating Methodology [12].

When a potential risk is discovered and the tester wants to determine how significant it is, the first step is to calculate the "likelihood" [12]. The likelihood can be determined by a variety of factors.

For this study, we have just considered four generic factors as we created our own customization to the OWASP methodology. It should be noted that each component has a number of options, and each option is assigned a likelihood rating from 0 to 9. The four factors we considered are:

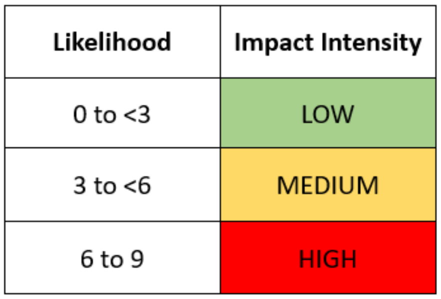
1) Ease of Discovery: How easy was it to discover these risks? Easy (0) and Difficult (9).

2) Severity of Risk: What is the Snyk or CVSS score on the severity of the risk? Low (2.25), Medium (4.5), High (6.75), and Critical (9).

3) Popularity Index: How frequently did a particular kind of risk show up on our evaluation? Less Popular (0), Mildly Popular (4.5), and Most Popular (9).

4) Business impact due to license incompatibility: Has this risk caused any violation? No Violation (0), License information not available (4.5), and Violation (9).

We simply average the results after we obtain the scores for each risk to get the overall likelihood. Figure 1 shows the mapping of the likelihood score and the level of impact.



Likelihood	Impact Intensity
0 to <3	LOW
3 to <6	MEDIUM
6 to 9	HIGH

Fig. 1. Likelihood and Impact Intensity Metric

VI. Results

A. RQ 1: What risks are involved with code dependencies and why do they exist?

1) Risky Dependencies and Vulnerabilities: For this RQ, we investigated all 15 JavaScript projects we gathered and discovered approximately 75 dependency risks that introduced specific vulnerabilities across all 15 JavaScript projects. Table I shows the number of dependency risks we found in each project, and we also noticed that certain risks were more common among the projects than others that appeared only once or twice, as shown in Figure 2.

jquery turned out to be the only project with the most risky dependencies as shown in Table I and also the project with the most unique kind of risks amongst all the projects we investigated, as shown in Table II.

The most prevalent vulnerability we discovered across all projects is known as ***Regular Expression Denial of Service-(ReDoS)***, and it was posed as a vulnerability by 33 out of 75 dependency risks. Some regex expressions that are widely used in all types of software cause the ReDos vulnerability, which could allow attackers to bypass security measures [19].

We discovered that only roughly 5 out of the 75 dependency risks we investigated are direct dependencies that are clearly obvious in the package.json files of the projects, as shown in Table III, while the rest of these risky dependencies were discovered in the package-lock.json or yarn.lock files, nine out of these risks were also not discovered in the package.json, package-lock.json, or yarn.lock files.

So to truly determine whether these risks were indirect dependencies, we used Snyk's suggestions to evaluate all currently installed dependencies in a project to see if these installed dependencies actually depend on these other risky dependencies and have them installed. Snyk gave some suggestions through the dashboard on the dependencies that likely have these risky dependencies installed as a dependency.

Furthermore, we discovered that the majority of the dependencies Snyk pointed out on the dashboard do not actually have these risky dependencies installed as direct dependencies, but we rather found them in the respective package-lock.json or yarn.lock files of the projects. For example; **ember-cli@4.10.0** is a direct dependency of the **ember.js** project, and we discovered **lodash.template**, which is a risk is actually installed as a dependency and visible in the package.json file of the **ember-cli** dependency.

We actually wanted to delve deeper into why these dependencies pose such a risk and why they exist at all. We used some of the guidelines presented in Russ Cox's paper to analyze the dependencies based on their maintenance and usage. We discovered that the bulk of these dependencies had their last commit to the master or main branch of the project's repository at least 6 months ago and more as shown in Table IV.

We simply were unable to locate the GitHub repository for the dependencies which are listed as *not found* in Table IV, therefore we made the assumption that the author had deleted them.

TABLE I
Number of Risks in each Project

Projects	No. of Found Risks
ember.js	5
tone.js	5
npm/cli	2
prettier	2
bootstrap	2
adonisjs/core	8
superstruct	3
googlechromelabs/jvsu	1
whatsapp-web-reveng	10
jquery	12
apollographql/apollo-client	1
freecodecamp	4
trekhleb/javascript-algorithms	1
Glider.js	9
pencil.js	10

TABLE II
JQuery Risks

Projects	Risks	Vulnerabilities
jquery	semver-regex	Regular Expression Denial of Service (ReDoS)
	debug	ReDoS
	request	Server-side Request Forgery (SSRF)
	growl	Arbitrary Code Injection
	ansi-regex	ReDos
	minimatch	ReDos
	uglify-js	ReDos
	word-wrap	ReDos
	ajv	Prototype Pollution
	underscore	Arbitrary Code Injection
	grunt-karma	Prototype Pollution
	mocha	ReDos

2) License Compatibility: We also examined around 770 package dependencies that these 15 JavaScript projects have

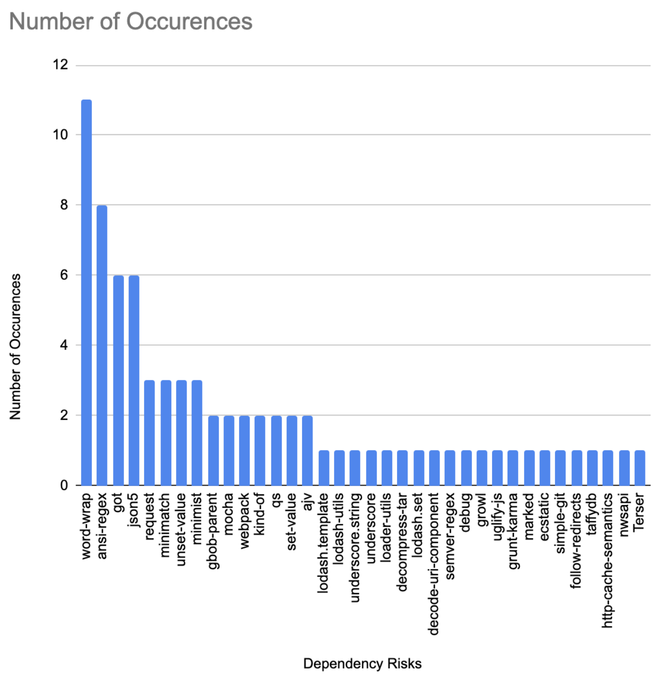


Fig. 2. Number of Risk Occurences

TABLE III
Risky Direct Dependencies

Projects	Risks	File
tone.js	webpack	package.json
	mocha	package.json
GoogleChromeLabs/jvsu	got	package.json
prettier	json5	package.json
jquery	uglify-js	package.json
	grunt-karma	package.json
pencil.js	webpack	package.json

TABLE IV
Risky Dependencies Maintenance and Usage

Risky Dependencies	Last Commit Date	No. of Dependents
word-wrap	May 4, 2018	1365
ansi-regex	October 28, 2021	1530
got	March 12, 2023	7,780
json5	January 5, 2023	3,625
request	February 11, 2020	54,988
minimatch	March 22, 2023	7,011
unset-value	February 11, 2022	703
minimist	February, 25, 2023	21,800
glob-parent	September 29, 2021	1,415
mocha	March 30, 2023	9,624
webpack	March 29, 2023	27,164
kind-of	January 16, 2020	1,312
qs	March 6, 2023	13,889
set-value	September 12, 2021	1, 016
ajv	January 8, 2023	9,799
lodash.template	August 18, 2015	965
lodash-utils	Not Found	Not Found
underscore.string	January 23, 2022	3,059
underscore	November 29, 2022	22,736
loader-utils	January 13, 2023	6,565
decompress-tar	July 27, 2017	53
lodash.set	April 23, 2021	1,702
decode-uri-component	December 19, 2022	808
semver-regex	July 8, 2022	243
debug	March 31, 2022	46, 478
growl	November 25, 2020	485
uglify-js	January 16, 2023	4,347
grunt-karma	May 19, 2021	69
marked	March 27, 2023	7,294
ecstatic	April 1, 2020	321
simple-git	March 27, 2023	3,500
follow-redirects	December 8, 2022	1,278
taffydb	January 14, 2021	134
http-cache-semantics	January 26, 2023	377
nwsapi	March 13, 2023	383
Terser	March 26, 2023	1,712

installed directly, as shown in each individual package.json file to determine whether or not all their licenses are compatible. Interestingly, none of the dependencies we evaluated turned out to be incompatible with their main projects, also, the MIT licence was the most common one.

The MIT licence was actually used by 57 of the risky dependencies, while the remaining 17 used the ISC license or one of the BSD licenses, and these were compatible since all 15 JavaScript projects were licensed under the MIT license.

B. RQ 2: What are the likely impacts of these risks?

Each risk gathered from all 15 JavaScript projects was evaluated for its potential implications for this RQ. To calculate the impact of this risk, we utilised the metric described in the methodology section. It was surprising to learn that we were unable to identify any risks with a high impact level. Yet, it was shown that out of roughly 75 risks, almost 30 had a medium impact. Last but not least, the final 45 had the lowest impact. These results are influenced by two crucial aspects. First off, no project—out of 15—had a licensing compatibility problem, and secondly, all the risks were simple to recognise since we used the Snyk tool. This evaluation may change if this kind of analysis is performed on more projects from diverse backgrounds.

VII. Discussion

1) Dependency Versions: Our investigation shows that majority of these risks have been introduced indirectly through a project. For instance, **tone.js** had a **webpack** risk with a **Sandbox Bypass** vulnerability. However, this risk was introduced in **webpack@5.74.0** as Snyk reports and webpack has since updated to **webpack@5.77.0**. We are not sure whether this vulnerability has been addressed in the newer version, but we are trying to point out is that, tone.js still uses the old version of webpack.

We feel that the fundamental issue behind a strict commitment to ensuring quick updates to a dependency is that many developers still don't consider two different versions of the same dependency as two dependencies but rather one. Dependency updates therefore need to be taken very seriously.

2) Sole Authors: Another intriguing finding was that a number of the risks we looked into were kind of started by an individual with a non-commercial perspective. These people rarely develop package dependencies that are appropriately maintained throughout time since a variety of circumstances could prevent them from maintaining the project.

We strongly advise developers to consider additional factors before using these kinds of dependencies, such as the length of the project's existence, its bug issue tracker, how quickly updates and changes are committed to the main branch, and so on.

3) Inspection of Dependencies: As developers follow instructions in documentation, tutorials, or the like to solve an issue or learn something new, they are frequently compelled to install dependencies. But, in our opinion, developers don't take the time to check to see if these dependencies are still in play. We strongly encourage every developer to adhere to the recommendations made by Russ Cox [2] in this paper to improve projects' risk-proofness.

VIII. Threats to Validity

A. External Validity

1) Diversity and Representativeness: Our study may not apply to projects built in other programming languages and, even if it did, it might produce different results because we only looked at JavaScript projects (as JavaScript is just one language population out of many others) with their respective package.json files.

2) Time Constraints: Due to time constraints, we were only able to conduct this investigation on 75 risks identified through 15 JavaScript projects. If we had conducted this analysis on several projects from various backgrounds, we might have discovered different results or unique risks.

3) Data's Validity in Relation to Time: The risky dependencies, vulnerabilities, and licence compatibilities that we identified are only valid as of the time they were discovered. If we perform another check on a particular project, we cannot guarantee that the data will be the same. It's possible that there are now more vulnerabilities, or lesser vulnerabilities, or newer license compatibility concerns.

IX. Conclusion

We examined over 700 package dependencies, 75 risks and vulnerabilities in JavaScript projects. We found that the majority of the risks that package dependencies carry into these projects comes not from them being installed directly but rather indirectly through these packages' own installed dependencies.

We discovered that, with the exception of one, that many of these risky dependencies haven't had a commit to the master or main branch on GitHub in many months or even years. And for this research - this is *only one side of a complex story*; further research can examine the beneficial aspects of code dependencies and conduct an empirical analysis of the reasons why some risky dependencies persist in projects, even over the years.

References

[1] A. Turgeman, "Software dependency risk," 2023. [Online]. Available: "https://riskfirst.org/risks/Software-Dependency-Risk"

[2] R. Cox, "Surviving software dependencies," *Communications of the ACM*, vol. 62, no. 9, pp. 36–43, 2019.

[3] M. 2023, "20th international conference on mining software repositories," 2023. [Online]. Available: "https://conf.researchr.org/track/msr-2023/msr-2023-mining-challengeCall-for-Mining-Challenge-Papers"

[4] Y. Ma, C. Bogart, S. Amreen, R. Zaretzki, and A. Mockus, "World of code: An infrastructure for mining the universe of open source vcs data," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 143–154.

[5] GitHut, "Githut," 2023. [Online]. Available: "https://githut.info/"

[6] NIST, "Software security in supply chains," 2023. [Online]. Available: "https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity/software-security-supply-chains"

[7] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 447–456.

[8] SNYK, 2023. [Online]. Available: "https://snyk.io/"

[9] N. CVSS, 2023. [Online]. Available: "https://nvd.nist.gov/vuln-metrics/cvss"

[10] S. V. Database, 2023. [Online]. Available: "https://security.snyk.io/"

[11] SNYK, 2023. [Online]. Available: "https://snyk.io/learn/what-is-cve-vulnerablity/"

[12] J. Williams, "Owasp risk rating methodology."

[13] Snyk, "Software dependencies: How to manage dependencies at scale," 2023. [Online]. Available: "https://snyk.io/series/open-source-security/software-dependencies/"

[14] S. Xu, Y. Gao, L. Fan, Z. Liu, Y. Liu, and H. Ji, "Lidetector: License incompatibility detection for open source software," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, feb 2023. [Online]. Available: https://doi.org/10.1145/3518994

[15] B. W. Boehm, "Software risk management: Principles and practices," vol. 8, no. 1, p. 32–41, jan 1991. [Online]. Available: https://doi.org/10.1109/52.62930

[16] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, and R. Kroeger, "Measuring the impact of code dependencies on software architecture recovery techniques," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 159–181, 2018.

[17] M. Keil, P. E. Cule, K. Lyytinen, and R. C. Schmidt, "A framework for identifying software project risks," *Commun. ACM*, vol. 41, no. 11, p. 76–83, nov 1998. [Online]. Available: https://doi.org/10.1145/287831.287843

[18] SNYK, "Install or update the snyk cli." [Online]. Available: "https://docs.snyk.io/snyk-cli/install-the-snyk-cli"

[19] C.-A. Staicu and M. Pradel, "Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 361–376. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/staicu