SCTE CABLE-TEC EXPO FAST FORWARD 2021
VIRTUAL EXPERIENCE
OCTOBER 11-14

2021 Fall Technical Forum
SCTE • NCTA • CABLELABS

Hidden Risk of Unpopularity in Open Source

A Technical Paper prepared for SCTE by

Chujiao Ma
Senior Security R&D Engineer
Comcast Cable Communications, LLC
Philadelphia, PA, USA
Chujiao_ma@comcast.com

Vaibhav Garg
Sr. Director Cybersecurity Research & Public Policy
Comcast Cable
Blacksburg VA
Vaibhav_garg@comcast.com

Table of Contents

Title **Page Number**

List of Figures

Title **Page Number**

List of Tables

Title **Page Number**

1. Abstract

Software development across the industry relies on the use of open source components (OSCs). Because these components are open-sourced, there is an assumption that these components are tested for security by third party researchers or open source communities. A vulnerability in a popular component can have ripple effects across the ecosystem. Consequently, more popular components are more likely to attract the attention of third-party researchers or the community. Less popular components are thus often left unexamined and potentially vulnerable. In this paper we propose a model to identify OSCs that create the greatest attack surface. Specifically, we propose a metric called relative popularity ratio and use it to risk-rank a set of JavaScript OSCs. We further refine the ranking using observable properties of code, such as number of lines of code. We then validate the efficacy of this metric by engaging third party university researchers to find vulnerabilities. Our results conclude that the hidden risk from unpopular OSCs is concentrated and can thus be addressed by small investments in the security analyses of OSCs.

2. Introduction

Black Duck's audit of commercial codebases in 2017 found that 96% of scanned applications contain open-source components [1]. According to Veracode software security report [2], 7 out of 10 applications contain flaws in their open source libraries on initial scan. 3 in every 10 applications have more flaws in their open source libraries than in the primary code base. Unlike commercial software, open source relies on voluntary contributors to identify vulnerabilities, build patches, and provide updates. Security researchers may spend more time finding vulnerabilities in popular components rather than those used less frequently. Thus, unpopular and consequently underexamined components may pose a hidden risk for their users.

A vulnerability in just one OSC can potentially impact software across multiple products and even across multiple companies. Consider "prototype pollution" [10]. It allows an attacker to modify an object in JavaScript, and its implications can range from injection attacks to denial of service. While fewer than 25 prototype pollution vulnerabilities were reported in 2019, according to a report from Synk, they impacted over 115,000 projects scanned [3]. For example, one high severity prototype pollution vulnerability was discovered in Lodash open source library. This library was used by 4.35 million projects on Github alone.

Once a vulnerability is found, the resulting exposure can take a long time to mitigate. Veracode's software security report notes that only 1 in 4 flaws are fixed in the first 32 days; 2 in 4 are still open after the first 6 months [2]. Simultaneously, a newly discovered vulnerability can turn into active exploits within days, leaving organizations with scant time to respond. A notable example is the 2017 Equifax data breach that affected 143 million customers. A vulnerability in the Apache Struts2 package made it possible for a remote attacker to send a malicious request that allowed them to execute arbitrary commands. 72 days after disclosure, the Equifax breach happened [4].

Thus, the hidden security risk of unpopular open source components cannot be ignored. One solution is for organizations to limit their software developers to an approved set of OSCs. Mature organizations may use tens of thousands of OSCs. Finding a secure and functionally equal OSC for each component is in itself a difficult, if not impossible, proposition. Another solution is to conduct continuous assessment of all the OSCs being adopted in an organization's software supply chain. Unfortunately, an exhaustive analysis would likely be cost prohibitive.

How, then, should organizations and development shops address the potential hidden security risk of the open source libraries? In this paper, we present a three-step approach to address this question:

1. Define a framework to risk rank OSCs based on factors such as popularity
2. Apply the framework to OSCs to produce a list of OSCs and identify areas of concentrated risk
3. Conduct security analysis of the high risk OSCs to identify vulnerabilities

Section 3 starts off by introducing different indicators of security risk for OSCs. In section 4 we describe how these factors can be combined to operationalize a *relative popularity ratio* and then used to determine areas of concentrated risk. Section 5 presents a case study of the security analysis of the OSCs identified in Section 4. Section 6 concludes with a discussion of key findings.

3. Risk Indicators

There are many direct and indirect indicators of the security risk posed by OSCs. They can be categorized as: 1) security status, 2) code characteristics, and 3) OSC popularity. Table 1 provides a list of potential measures for each category. Different language or package managers collect different information and rank OSCs differently. Even the metadata and statistics available also differs. Even if an organization has a complete inventory of all the OSC it uses, the details may be incomplete. Thus, there may be no one size fits all solution to using these OSC security risk indicators. Instead, each organization may select the indicators that suit their development environment. Here, we discuss the three categories of OSC security risk in detail.

Table 1. Possible risk indicators.

Security Status	Code Characteristics	Popularity
▪ Reported vulnerabilities from CVE and NVD ▪ Severity of vulnerabilities based on CVSS ▪ Security of the language ▪ Typical time to remediation ▪ Number of open issues	▪ Lines of code ▪ Complexity of the code ▪ Number of versions ▪ Time of creation ▪ Number of libraries they use ▪ Where/how it is used	▪ Number of contributors ▪ Number of Github stars ▪ Number of forks and pull requests ▪ Number of subscribers ▪ Number of downloads ▪ Number of dependencies

3.1. Security Status

Indicators of security status can include the number of reported vulnerabilities from CVE and NVD databases as well as severity of reported vulnerabilities based on CVSS scores. However, not all flaws have CVEs. The percentage of flaws or vulnerabilites that have a CVE vs those that don't can vary widely depending on the language. In Veracode's analysis of open source, 61.8% of Javascript vulnerabilities don't have a CVE while it is only 10.5% for PHP [5].

Furthermore, different repositories may have distinct level of security exposure depending on how security conscious the owner is and how many independent researchers have analyzed the OSC. They may be captured by the number of open issues against a repository as well as the mean time to remediate.

The language in which the code is written may provide additional indicators. Some languages provide more in-built protection against specific security threat, such as type safe languages. Each language is also more susceptible to specific attacks and less vulnerable to others [11]. For example, JavaScript unlike Ruby is more susceptible to deserialization.

3.2. Code Characteristics

Characteristics of the OSC code can also be indicative of the probability of risk. Prior research has found that code complexity and code size both may correlate with the probability of finding a vulnerability [6] [7]. The coding language used also matters, since almost half of PHP packages contain vulnerabilities while the number is much smaller for other languages [2]. The number of libraries used by the OSC also matters. A report by Snyk found that 86% Node.js vulnerabilities are from indirect dependencies, 81% for Ruby and 74% for Java [3].

In addition to the characteristics of OSC itself, characteristics of the application using the OSC also matters. Depending on where and how it's used, a vulnerability in the OSC can have widespread impact that's difficult to fix or be easily fixed with a compensating control.

3.3. Popularity

Another way to evaluate the risk of open source is by looking at its health or popularity. If a project has low contributor attraction or retention, there's a high chance of it dying out and used by attackers in typosquatting or social engineering. However, the total number of contributors does not correlate to whether a project survives, experience level of the contributors matter as well for the quality of the project [8].

With that said, there are many smaller libraries with just a handful of total contributors and 1-2 active maintainers, so it's important to look at activity level as well instead of just contributors. Node package manager (npm), for example, has a popularity rating that takes into account the number of stars, forks, subscribers, contributors, dependents, downloads. They also have a maintenance score looking at ratio of open/total issues, time takes to close issues, most recent commit, commit frequency, release frequency [9]. While the popularity of the project is important, it is also important to account for the stability of the project as well as distinguish between a huge pull request vs. multiple smaller ones.

4. Risk concentration

Current security solutions aim to identify known vulnerabilities and corresponding patches for OSCs used within an organization's software development lifecycle (SDL). Discovery of new vulnerabilities and verification through exploits necessitates manual analysis. If the organization uses thousands of OSCs across different applications, an exhaustive analysis would likely be cost prohibitive.

It might be difficult to get information on some OSC security risk indicators due to scale or incomplete inventory. Furthermore, it may be difficult to compare these indicators as distinct package managers collect different types of information. Thus, each organization must begin with a set of indicators for which they have the most complete information. These indicators can be combined into a relative popularity ratio to address the bias introduced by one single metric.

Based on this ratio organizations can identify areas of concentrated risk. This allows prioritized security analyses of OSCs that have a greater probability of containing hidden risks as well as for whom

vulnerabilities will result in greater organizational impact. In this section we discuss this process with an example from Comcast.

4.1. Relative Popularity ratio

The goal of the relative popularity ratio is to identify OSCs that are more popular within the organization (high impact) than externally (under evaluated) and is calculated as:

$$Relative\ Popularity\ ratio\ = \frac{Internal\ popularity}{External\ popularity}$$

By using a ratio of two measures, rather than a single measure, we are able to reduce biases specific to a single dataset. Furthermore, relative popularity captures both the under evaluation of an OSC by third party security researchers, and therefore the hidden risk, and the possible impact for the concerned organization. Table 1 notes the various indicators of popularity, e.g. number of forks.

For Comcast, internal popularity was measured by the number of dependents, as this information is easily assessable through Software Component Analysis or SCA tools. External popularity was measured as an average of: 1) dependents (impact), 2) weekly downloads (current popularity), and 3) Github stars (long term popularity). These were selected based on the specific information available for npm, the specific repository that hosts the specific OSCs under analysis.

$$Popularity\ ratio = \frac{\%\ of\ Comcast\ dependents}{(\%\ of\ npm\ dependents + \%\ of\ github\ stars + \%\ of\ weekly\ downloads)/3}$$

4.2. Risk Concentration

While ranking OSCs according to their relative popularity can help prioritize the list of OSCs, there are still have thousands or tens of thousands of OSCs for each language. The next step then is to determine areas of concentrated risk. For example, consider the main languages used for development at an organization. At Comcast most of the development is limited to three: 1) C, 2) Java, and 3) JavaScript.

For each language, developers may use hundreds of OSCs. An individual library may be used just once or in up to thousands of applications. Figure 1 plots the OSCs for each language by percentage of dependents. Observe that the top 50 OSCs covers 26.2% of dependents for C, 37.7% of dependents for JavaScript and 25.9% of dependents for Java. In fact, top 200 OSCs provide more than 50% coverage for each language, i.e. 52.2% for C, 59.6% for Java, and 59.5% for JavaScript.

Figure 2 shows the number of dependents or applications using each OSC for the top 1000 JavaScript for 2019, 2020 and 2021. The top 50 OSCs cover 37.7%, 23.7% and 37.9% of the risk for 2019, 2020 and 2021 respectively. The top 200 OSCs covers 63%, 45% and 60% of the risk for 2019, 2020 and 2021 respectively. Thus, as more development teams adopt SCA tools and more OSCs are inventoried, the general trend continues to hold true from year to year.

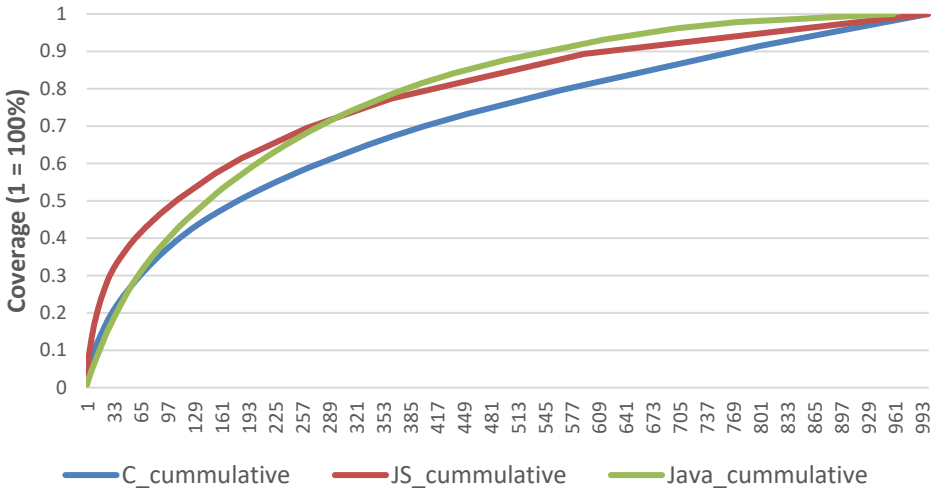

Figure 1. Percentage of dependents covered by the top 1000 OSC for C, JavaScript and Java

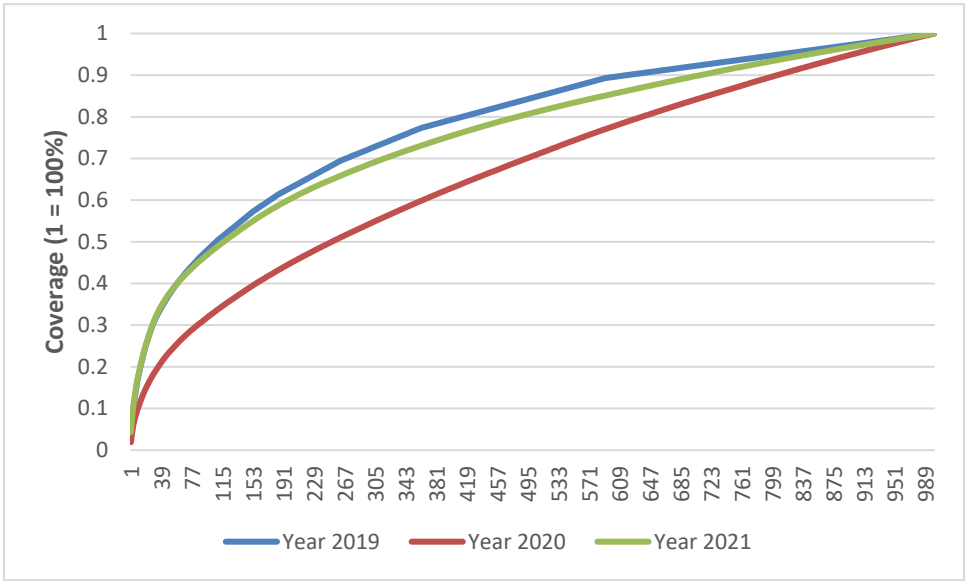

Figure 2. Number of Dependents for top 1000 JavaScript OSCs for 2019, 2020, and 2021.

Thus, the security risk from OSCs is concentrated. Analyzing the top 50-200 components will provide coverage for majority of the risk. There is then no need to conduct an exhaustive security assessment of all components. Instead, the organization can focus its resources on OSCs with greater number of dependents. Here we focus on the top 50 components in JavaScript, which covers approximately a third of the risk.

5. Case Study

As we note in Section 4, the security risk from OSCs is often concentrated. Thus, a detailed security assessment of a small set of OSCs provide coverage for much of the security risk. In this section we discuss an example of such an assessment. We began by selecting the OSCs. Based on the trajectory of usage of OSCs within Comcast, we decided to analyze top 50 JavaScript OSCs. We further narrowed our scope based on the relative popularity ratio.

In terms of manual analysis, there are many ways to attack and test the OSCs. OWASP top 10 and SANS top 25 are a good place to start. In addition, according to a report by Veracode [5], there are four categories of flaws that represent 75% of all flaws found in the open source libraries: access control, cross-site scripting, sensitive data exposure and injection. We decided to focus on the flaws within the source code rather than due to access control or other external factors, and those that are server side rather than client side.

5.1. Methodology

The first step of the analysis is to collect a list of OSCs. The information for OSCs used within Comcast are collected from WhiteSource, thus the list only includes OSCs used by applications of teams using WhiteSource. There is a limited amount of information available for all OSCs so the list only contains the name of the component, and the number of dependents (applications using it with all versions combined together). For this experiment we focused on JavaScript.

Once the internal list of popular OSC is collected, we ranked them according to the number of dependents, and external data is then collected for the top 50 OSCs that are used by most applications within Comcast. External data used for popularity ratio calculation such as number of dependents and weekly downloads are collected from npm repository. The number of stars is collected from the Github repository. The popularity ratio is then calculated for each OSC, where a higher number indicates it is more popular within Comcast than externally, while a lower number indicates that it is more popular outside of Comcast than internally.

The list of top 50 JavaScript OSCs was given to third party university researchers at Comcast Center of Excellence for Security Innovation at the University of Connecticut for security analysis. The analysis focused on identifying previously undiscovered code injection vulnerabilities and verifying that they are exploitable.

5.2. Results

Figures 3, 4, and 5 lists the 50 OSCs under analyses based on external popularity, internal popularity, and relative popularity ratio respectively. Security analyses found new code injection vulnerabilities in 3 of the 50 OSCs: jade, depot and prototype. These components are color coded red. The problems came from embedded dependencies; newer versions of the packages have fixed the vulnerabilities. Separately, one package was found to be deprecated, and colored grey.

When ranked according to either internal or external popularity, as seen in Figure 3 and 4, the vulnerable packages are scattered throughout the top 50 list. In Figure 5 however, when the top 50 OSCs are ranked according to the popularity ratio, where higher ratio indicates high popularity within Comcast but low popularity externally, we can see that the vulnerable components are concentrated in the top third of the list. It also makes sense that the deprecated OSC would rank high on the popularity ratio since there would

be almost no external usage (thus the external popularity is only from a few GitHub stars) while it was still used internally. The OSC has been phased out of use since then.

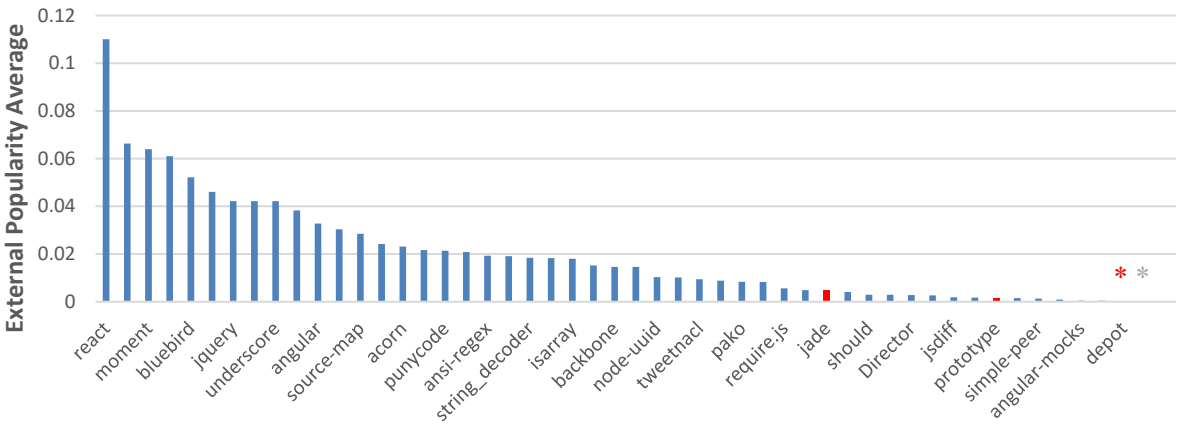

Figure 3. Top 50 OSCs ranked by external popularity. * denotes location of vulnerable/deprecated components difficult to see.

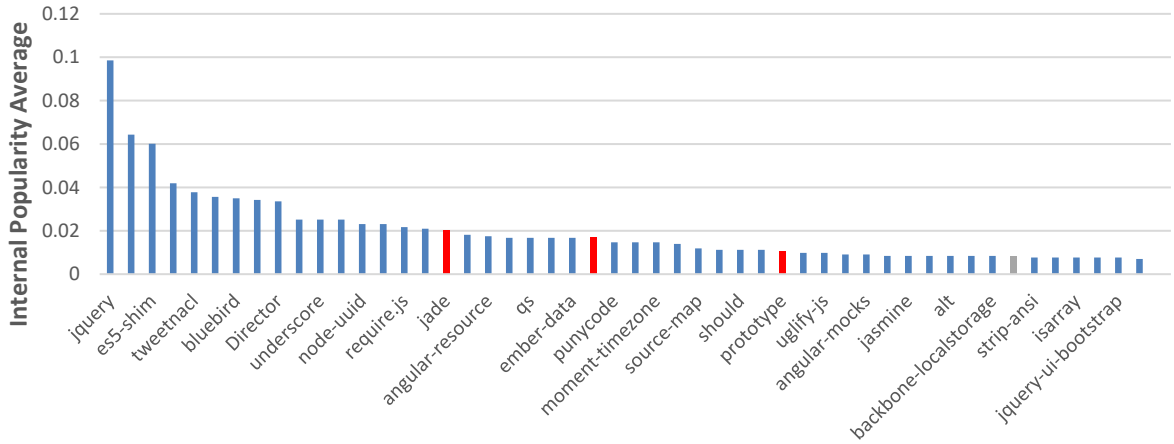

Figure 4. Top 50 OSCs ranked by internal popularity.

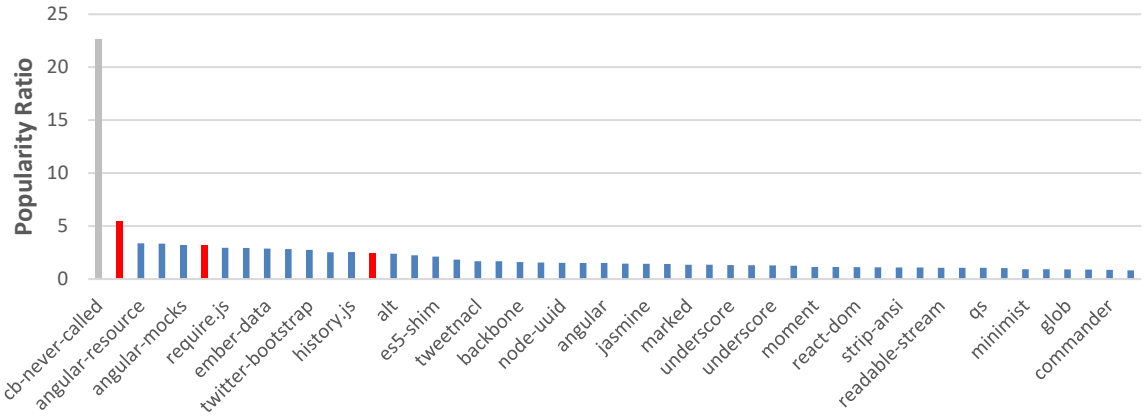

Figure 5. Top 50 OSCs ranked by popularity ratio.

The three vulnerable OSCs covers 1.78% of dependents or applications used by the teams in the list and the analysis was only done for code injection vulnerability. From this case study, we have verified our hypothesis that less popular OSCs contain hidden risk.

6. Conclusion

The first step toward managing the risk from open source is to identify the OSCs used across the organization. Using software composition analysis helps with both taking inventory of OSCs used and whether there are any existing vulnerabilities as well as corresponding patches available. It also automates the discovery of old or deprecated versions of an OSC being used. For many OSCs no significant vulnerabilities may be publicly known. This may be because these OSCs are secure. Alternatively, it is possible that these components are not popular enough for third party security researchers to analyze them. This creates a hidden security risk. As most organizations use hundreds, if not thousands, of OSCs an exhaustive security assessment would be both cost prohibitive and impractical.

In this paper we provide a roadmap to address the hidden security risk from such OSCs. First, we describe the various indicators that can be used to approximate the security status of an OSC. Then we provide a way to use these indicators to risk rank OSCs via a relative unpopularity ratio and identify areas of concentrated risk that can be prioritized for security assessments. We then present an example case study. Our results indicate that a majority of the hidden risk is concentrated in a minority of components. For us, the top 50-200 components cover 30-50% of the risk. Thus, small investments in security assessments of OSCs can address any potential unknown vulnerabilities.

This paper presents a specific instantiation of the relative popularity ratio. However, organizations may choose their own indicators based on their operational context, information sources, and data completeness. Furthermore, the relative unpopularity ratio is only intended to provide a way to prioritize security assessments. It is unlikely that there will be a quantitative correlation between the ratio and the number of vulnerabilities found. As the ratio can be constructed in different ways, the specific ranking of an OSC will differ in each distinct instantiation of the formula depending on the indicators used. Additionally, assessors (and assessments) themselves may be better at finding certain types of vulnerabilities over others. Regardless, as we show in this paper this qualitative approach is still effective at identifying areas of concentrated risk and addressing them. Thus, it is possible to shine a light on the hidden risk of unpopularity in OSCs.

Abbreviations

CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
npm	Node Package Manager
NVD	National Vulnerability Database
OSC	Open Source Component
OWASP	Open Web Application Security Project
PHP	Hypertext Preprocessor

Bibliography

[1] Z. Zorz, "The percentage of open source code in proprietary apps is rising," 2018. [Online]. Available: https://www.helpnetsecurity.com/2018/05/22/open-source-code-security-risk/.

[2] "State of Software Security v11," Veracode, 2020.

[3] A. M. a. S. Zitzman, "The State of Open Source Security Report 2020," Synk, 2020.

[4] C. Fearon, "Examining Apache Struts remote code execution vulnerabilities," 3 October 2017. [Online]. Available: https://www.synopsys.com/blogs/software-security/apache-struts-remote-code-execution-vulnerabilities/.

[5] "State of Software Security: Open Source Edition," Veracode, 2020.

[6] R. a. W. J. a. H. A. a. J. W. Scandariato, "Predicting Vulnerable Software Components via Text Mining," *IEEE Transactions on Software Engineering,* 2014.

[7] Y. a. M. A. a. W. L. a. O. J. A. Shin, "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities," *IEEE Transactions on Software Engineering,* 2011.

[8] N. Eghbal, "Methodologies for measuring project health," 2018. [Online]. Available: https://nadiaeghbal.com/project-health.

[9] A. C. a. A. Duarte, "npms - About," 2021. [Online]. Available: https://npms.io/about .

[10] Kim HY, Kim JH, Oh HK, Lee BJ, Mun SW, Shin JH, Kim K. DAPP: automatic detection and analysis of prototype pollution vulnerability in Node. js modules. International Journal of Information Security. 2021 Feb 13:1-23.

[11] "Most Secure Programming Languages," WhiteSource. 2021. https://www.whitesourcesoftware.com/most-secure-programming-languages/

