Article

Not peer-reviewed version

Assessing Security Vulnerabilities in Large Code Bases Using Open Source Software

Dr. Kristen Walcott [*] , Mark Vaszary [*] , Dr. Thomas Hastings

Posted Date: 13 December 2024

doi: 10.20944/preprints202412.1154.v1

Keywords: open-source software; vulnerabilities; dependencies; software supply chain attack

Article

Assessing Security Vulnerabilities in Large Code Bases Using Open Source Software

Kristen Walcott [*,†,‡], Mark Vaszary [†,‡] and Thomas Hastings [†,‡]

University of Colorado, Colorado Springs
[*] Correspondence: kjustice@uccs.edu
[†] University of Colorado, Colorado Springs address: 1420 Austin Bluffs Pkwy, Colorado Springs, CO 80918.
[‡] These authors contributed equally to this work.

Abstract: Large software applications typically use open source software (OSS) to implement key pieces of functionality. Often only the functionality is considered, and the non-functional requirements, such as security, are ignored or not fully taken into account prior to the release of the software application. This leads to issues in two different areas: 1) evaluating the OSS used in our current software, 2) evaluating OSS that our developers may be considering using. This research focuses on providing a mechanism to examine third-party components, focusing on security vulnerabilities. Both current and future software face similar security issues. We develop an approach to minimize risk when adding in OSS to an application, and we provide an assessment of how exploitable new/existing software applications may be if not secured with other layers of defense. We have analyzed two case studies of known compromised open source components. The developed controls identified high levels of risk immediately.

Keywords: open-source software; vulnerabilities; dependencies; software supply chain attack

1. Introduction

Usage of Open Source Software (OSS) is rampant in many commercial software applications. This is due to the importance of meeting tight release schedules and software developers heavily relying upon OSS to make their release dates. It is much easier and less expensive to integrate existing libraries with the functionality needed, instead of custom coding those same capabilities using native code. Large code bases are built up using many kinds of OSS, third-party libraries, and frameworks. Some are heavily researched and commonly used. Others are picked ad-hoc for use in projects. This affects both new and older code. The heavy usage of external libraries and other third party dependencies leads to the application under development to inherit any of the vulnerabilities of underlying libraries or OSS components. This includes direct dependencies as well as transitive dependencies.

There have been many high profile vulnerabilities found with OSS used in commercial software applications that were exploited by attackers. Amongst this list are Log4j, Spring4Shell and Kaseya [1]. According to TechMonitor, attacks like these against Open Source Software have increased 650% in 2021 alone [2] As this problem will only increase, there is an urgency to better understand how vulnerable a commercial software application/OSS is and what underlying vulnerabilities exist within its code base. Opensource software is utilized in 97% of software applications today [3].

There are over 5,000 open-source security advisories on GitHub today [4]. The zero trust model assumes that every project will have some security findings even without targeted supply chain attacks. Therefore, we need better methods of vetting and monitoring open source components throughout the component's life-cycle. We can no longer trust packages straight off the internet, and we need to verify that the packages are safe.

Current state of the art includes using tools like Dependency graphs and Dependabot, of which both are available in GitHub [5]. There are also tools like Snyk [6], that operate similarly to the GitHub tools and monitor open source code. The tools have limited coverage and success for finding transitive dependencies. Other state of the art tools involve heavy lifting and require software developers to manually review their OSS libraries used in their application builds and/or dynamic run-time executions. This is typically accomplished through scanning of the source code base and tracking the usage of which OSS components/libraries are used in their application. After the list of components, or manifest, is created, they cross check known vulnerability databases, such as the National Vulnerability Database (NVD) that tracks Common Vulnerabilities and Exposures (CVE), to determine if their application is affected.

In this research, we intend to perform and develop the technique and associated tools for OSS security assessment. The framework will be applicable to existing code or to up-in-coming applications. The work is divided into two significant focuses within security analysis: 1) Providing a realistic vulnerability exploit-ability score and 2) Incorporating continuous verification of security analysis of OSS components. Together, the main contributions of this work are the following:

- Description of a technique to examine used/potential components for vulnerabilities
- Creation of a static code analysis technique to determine a dependency tree of components used
- Development of a continuous verification system to enable organizations to make data-driven decisions based on component analysis
- An automated systems architecture for vetting open-source applications before or after use

2. Materials and Methods

Understanding all the third party components that make up a large commercial software application is not a trivial task. As most commercial software is highly built upon the integration of Open Source Software (OSS), software developers inherit all associated vulnerabilities that come with the OSS libraries that the applications use for functionality. Usage of open source software can account for as much as 80% of the total software base of a large application, based on industry averages [7]. As a result of this, vulnerabilities that exist in open source libraries has a wide impact across most industry available applications. Speed of release delivery can come at the price of security.

As source code for OSS is publicly available and can be reviewed and analyzed by attackers, any existing vulnerabilities can be discovered and attempts to exploit those vulnerabilities can be implemented. By the very nature of how software applications release security patches and updates to remediate those vulnerabilities, attackers can learn where those vulnerabilities exist. In the event that the security patches are not applied in a timely manner, the attackers have that window of opportunity for their attacks. There are many widely known exploits that have taken advantage of these security patch delays, to wreck havoc and exploit vulnerable applications.

Software supply chain attacks targeting open-source components have increased 742% in the 2022 year [8]. These exploits continue to grow in frequency and magnitude. Over the previous two years, this topic has become a priority among organizational leaders and researchers. As a result, the software community has identified weak links in packages [9], created standards to highlight best practices for component vetting [10], automated vulnerability look-ups for dependencies [11], and widened the aperture for reporting vulnerabilities and malicious packages [12].

The Open Source Security Foundation has been doing much research into the topic of software supply chain protection. One of the projects we use extensively in our study is their Security Scorecards for Open Source Projects. Although we did not use all of the metrics, the scorecards provided and added a couple of metics that heavily influenced our research. According to the researchers from Google, “The goal of Scorecards is to auto-generate a ‘security score‘ for open source projects to help users decide the trust, risk, and security posture for their use case. This data can also be used to augment any decision making in an automated fashion when new open source dependencies are introduced inside projects or at organizations." [10]

Understanding the package ecosystem and the community support around a package is essential, but a more holistic view is required to understand the actual risks before incorporating open source packages. The MITRE Corporation has been a faithful steward of maintaining two critical databases used for static code analysis. The first database is the Common Vulnerabilities and Exposures (CVE) database. This database contains a list of known vulnerabilities in packages [13]. The second database they steward is the Common Weakness Enumeration database. This database contains ”weakness types for software and hardware and is used as a baseline for weakness identification, mitigation, and prevention" [14]. There are a number of other Open Source vulnerability sources, which supplement the NVD database [15]. Combining these vulnerability sources enables the vulnerability assessment to be more complete and to limit the number of false positive and false negative events.

In learning the “new“ software development lifecycle, now sometimes called the “software development lifecycle in the cloud age“ [16], we need to question how the traditional software development lifecycle changes with these “outside“ components in mind, especially in terms of security vulnerabilities. How security vulnerability checks fall into CI/CD is also a concern.

2.1. Dealing with Vulnerabilities in the Supply Chain

The supply chain is now a vital consideration in software engineering. With any large code base, at any point in time, a key importance is understanding the dependencies and what vulnerabilities might affect the code. This needs to be built into the traditional software supply chain methodology. However, we first need to learn more about the dependencies with which we are working and discover ways to delve deeply into those dependencies. We then determine what vulnerabilities affect the most commonly used components and classify them. Next, we will examine how vulnerability scanners could be used in a secure development lifecycle. Lastly, we will discuss how our preliminary work has been used to accomplish these goals in incorporating past vulnerability information into operations.

2.1.1. Potential and Current Vulnerabilities

Most code relies on OSS or other components. We frequently use and trust these components without much thought. Methods of determining what third party applications are readily available today and may vary across different programming languages. Source scanning can provide a better understanding of the underlying dependencies. This is often done by building dependency trees that document the program flow and clearly identify which dependencies are used [17]. There are many scanning tools that can help build a list of dependencies through scanning of the source code.

Beyond source code scanning, there are industry supported initiatives to build machine readable software bill of materials (SBOM) [18] lists, which would potentially replace or supplement the need to build dependency trees through source code. The integrity of the SBOM would be ensured by the provider or maintainer of the dependent third party component, whether that be a commercial vendor or open source. However, source code scanning is still prudent as a check and balance compared to what is being provided by the maintainers of the dependent code.

The challenge comes in prioritizing the “discovered“ vulnerabilities. For large code bases, the number of identified vulnerabilities can be in the thousands or more. Most teams that support large code bases do not have unlimited resources available to jump into and review thousands of identified and unaudited vulnerabilities. Furthermore, after going through a review of thousands of unaudited vulnerabilites, there is a high percentage chance that many of those identified vulnerabilities will not be relevant or end up being false positives. “Not relevant“ vulnerabilities are ignored and essentially marked as “don't care“. What is key is being able to filter down to the most urgent and critical vulnerabilities that are hidden amongst the thousands of identified vulnerabilities. A third party component version may show as vulnerable, but that does not necessarily mean that it can be exploited, but this analysis of potential and current vulnerabilities can give a warning to developers.

In this work, we will use open-source and industry applications as tests to identify components in underlying programs that are likely to be vulnerable, thus making our applications vulnerable.

2.1.2. Code Analysis for OSS Dependencies

Software product vendors determine their development strategies and need to balance those with security considerations that come with using OSS components. Commercial software products on average have a significant amount of open source third party components. Of key importance is understanding what the product uses when it comes to open source third party components. Identifying which OSS components are used and who maintains them can help determine if the product is properly securing its application against attacks. In addition, if the product is heavily dependent upon an OSS component but is not involved in maintaining it or ensuring its continued development, this will lead to higher risks of vulnerabilities arising.

Identification of the components used needs to be followed by linking to vulnerabilities to paint the attack surface [19] picture. There are deficiencies with using CVEs from the NVD database. Enhancing the relevancy can be accomplished by using Common Platform Enumeration (CPE) information, in addition to CVE information. Linking to open-source communities and scorecards, such as the Open Source Security Foundation (OSSF), can also improve the accuracy of the vulnerability analysis.

Higher number of vulnerabilities affecting the same OSS component may highlight certain components to pay more attention to and to ensure that continued usage of them is warranted. By analyzing the components being used through OSS community metrics and supplemental vulnerability sources, a component wellness score can be associated with each component. Using our algorithms, we plan to formulate a simpler and more useable ranking of the components sourced from the software supply chain. Ranking those components that have minimal usage of its code base, following the principles around minification, will also be included.

2.1.3. Continuous Integration System

Based on the results from Section 2.1.2 and 2.1.1, we will incorporate these tools and analyses into our framework. We leverage the CVEs and the CWEs in our methods to identify known vulnerabilities in the open-source component and its dependencies. Then we use the CWEs to check for known code signatures that allow the package to be compromised if measures are not implemented to prevent malicious attacks. With speed of delivery in cloud environments on the rise, we are also researching how our assessment can be applied in real time for continuously changing cloud environments.

2.1.4. Incorporating Operations

There is a new way to think about the operations life-cycle, Instead of following a traditional software development lifecycle, the premise is that organizations can manage their IT operations using day 0, day 1, and day 2 nomenclatures. We leverage day 0, day 1, and day 2 in our evaluation. Day 0 is the design phase where an organization considers how or if it will incorporate a new open source component into a software project. Once the decision had been made to use the open-source component, we move into day 1 operations. Day 1 is what goes into actually incorporating the component. This may include adding the component to the package or vendorizing the component to make it available to the organization [20]. Day 2 operations occur after the package is included and running in production, this is when maintenance and monitoring becomes a priority.

The challenge is in determining how vulnerabilities become evident over the operations lifecycle with the goal of early detection of potential issues.

3. Results

Our preliminary work includes an assessment of the gaps in the currently available vulnerability assessments, as well as an proposing an exploit-ability score to prioritize the assessed vulnerabilities. We examine the trade offs that contribute towards a practical exploit-ability score that can be used to provide resource limited product teams that capability to focus on the more impactful and critical vulnerabilities for securing their products sooner than later.

3.1. Vulnerabiity Assessments

With a better understanding of the effectiveness of the vulnerability scans, the next step is to identify the gaps and contribute with methods to fill those gaps. If vulnerabilities are getting through to deployment, why are they making it past vulnerability scanning? Reasons could be due to improper vulnerability scanning tools to lack of resources to perform the unaudited reviews. Many potentially bad vulnerabilities could be making it to deployment, because there is a lack of resources to review them all in time.

The holy grail for prioritizing vulnerabilities involves pulling the previous vulnerability leanings together, in such a manner that you can derive an exploitability score that would help quickly assess all the unaudited vulnerabilities and highlight those of the upmost for further review. Innovating a solution here will need to rely upon cutting through the vast amount of vulnerability assessment results and making actionable sense from it.

Evaluation
Using the vulnerability assessment, we then apply an exploitable score to rank the severity of the vulnerabilities introduced through the dependent libraries. Our exploit-ability score is based on the attributes of the vulnerabilities, as learned from the NVD and supplementary vulnerability databases.
Our techniques are proven using commercial software that is sold globally. In addition, we've applied our techniques to the open source software application, OpenPilot, which is being championed by Comma.AI. Comma.AI sells the hardware product, Comma.

3.2. Continuous Verification in Operations

In addition to static vulnerability assessment, we can learn much about the current state of a project by looking at the package's dependencies, source code, and community. These six controls are used to understand the current state of a package and the package's community and can inform the decision of whether or not to use the package.

- C1: checks the package for known vulnerabilities in a package's dependencies and in the package itself.
- C2: checks the source code for known weaknesses in the code base using static code analysis, which leverages CWE information.
- C3: looks at the package's community to understand the makeup of the project's maintainers.
- C4: looks at the hygiene of the package.
- C5: is a policy that dictates that no open-source artifact will be included, which is not built by a trusted source.
- C6: is a network perimeter defense around the development and production environment of the software project.

Evaluation We applied the methods to two case studies that were compromised. The first case study, UAParser.js, came from the NPM ecosystem and the second, rest-client, came from the RubyGems ecosystem. These OSSs were prime candidates because they averaged hundreds of downloads a week. Our controls are repeatable and capable of identifying risk during day 0, day 1, and day 2 of the opensource component's operational life. These six continuous verification controls enable organizations to make data-driven decisions and mitigate breaches, such as analyzing community metrics and project hygiene using scorecards and monitoring the boundary of the software in production. In one case study, UAParser.js, the controls identified high levels of risk immediately even though the package is widely used and has over 7 million downloads a week. In both case studies we found that the controls could have prevented malicious actions despite the project breaches [21].

4. Discussion

Other works involve improving the known vulnerability databases and better understanding the inter-dependencies within the libraries used in the source code base. The work done by Wang et al. [22] is representative of the efforts to build up a database of known vulnerabilities and supplement the NVD CVE database of vulnerabilities. By infusing machine learning techniques, their system helps to identify vulnerabilities that exist in OSS. It has been shown through research, that security vulnerabilities exist through many of the commonly used OSS tools available today, when using state of the art tools and threat models like Stride [23], even when the vulnerabilities are known and documented in vulnerability databases. The number of security vulnerabilities also increases when the number of dependent libraries increases [24]. Lastly, there are concerns that the current state of the art over-inflates the number of vulnerabilities found [7].

Other works involve improving the software development life-cycle itself or the security scanning done. One approach is to do more continuous security monitoring of code, using an artifactory based approach that is driven by a constant state of change done by different people in different organizations [7]. Another involves using static and dynamic analyses to detect the reach-ability of vulnerable code in libraries. Eclipse Steady [25] is a tool that improves the percentage findings of true positives for finding vulnerabilities, as compared to OWASP Dependency Checker (ODC). ODC has been shown to provide a false positive rate of up to 88.8% for code concentric scans. Other works, such as Vuln4Real, also provide vulnerability assessments with improve falso positive alerts [26].

There is also work around why developers are either slow or decide never to address security vulnerabilities in their code base. When it comes to developers keeping their library dependencies up to date, studies have shown that 81.5% will not update their outdated dependencies. A study by Raula Kula et. al. highlighted that developers were unlikely to prioritize a library update, even when vulnerable dependencies are known. They cite that extra workload and responsibility is beyond what they are willing to take on, as well as risks to the feature enhancements they are releasing [7]. It is highly important that the number of dependencies are managed and timely updates are done for dependent libraries [27], whether it be by notification of vulnerability discovery or ease of library update.

5. Conclusions

Vulnerability assessment is a complicated and tedious process, which current state of the art could evolve with innovation and new ways to help reduce the noise and provide focus on the most urgent vulnerabilities that need immediate attention. This is drastically impacted with the usage of OSS components, that provide attackers the means of reviewing source code that goes into your large code base. Attackers may not be able to get access to your custom code, but they can potentially exploit any vulnerabilities that you bring in with OSS components. In our work, we propose our techniques to improve and further automate the vulnerability assessment and to minimize the attack surface when using OSS components. These techniques offer insights into how products can improve their vulneability scanning capabilities and ensure their limited resources are used in a manner most efficient to the product.

We provide a way to prioritize the vulnerabilities found and areas to look for potential gaps, based on what we learned from studying our product and its usage of OSS components. By automating and addressing the gaps, not only from missed vulnerabilities but also from enhancing the vulnerability mapping itself, our work shows that the OSS community can gain benefit by continuing our efforts with developing the exploit-ability factor and using it to measure the level of risk that an application may have if continuing usage of OSS components.

Author Contributions: 'Conceptualization, K.W., M.V., T.H.; methodology, K.W., M.V., T.H.; software, M.V., T.H.; validation, K.W., M.V., T.H.; formal analysis, K.W., M.V., T.H.; investigation, M.V.; resources, K.W.; data curation, T.H.; writing—original draft preparation, K.W., M.V., T.H.; writing—review and editing, K.W., M.V., T.H.; supervision, K.W.; All authors have read and agreed to the published version of the manuscript.'

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Townsend, K. Cyber Insights 2023 | Supply Chain Security. https://www.securityweek.com/cyber-insights-2023-supply-chain-security/, 2023.
2. Fitri, A. Supply chain attacks on open source software grew 650% in 2021. https://techmonitor.ai/technology/cybersecurity/supply-chain-attacks-open-source-software-grew-650-percent-2021, 2021.
3. Plumb, T. GitHub's Octoverse report finds 97% of apps use open source software. https://venturebeat.com/programming-development/github-releases-open-source-report-octoverse-2022-says-97-of-apps-use-oss/, 2022.
4. Microsoft. Github advisory database. url = https://github.com/advisories, 2023.
5. Blog. Enable Dependabot, dependency graph, and other security features across your organization. https://github.blog/changelog/2020-07-13-enable-dependabot-dependency-graph-and-other-security-features-across-your-organization/, 2020.
6. Snyk. Enable Dependabot, dependency graph, and other security features across your organization. url = https://docs.dependencytrack.org/datasources/snyk/, 2023.
7. Paschchenko, I.; Vu, D.L.; Massacci, F. A Qualitative Study of Dependency Management and Its Security Implications. Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security; Association for Computing Machinery: New York, NY, USA, 2020; CCS '20, pp. 1513–1531. doi:10.1145/3372297.3417232.
8. Vailshery, L. Year-over-year (YoY) increase in open source software (OSS) supply chain attacks worldwide from 2020 to 2022. url = https://www.statista.com/statistics/1268934/worldwide-open-source-supply-chain-attacks/, 2023.
9. Zahan, N.; Zimmermann, T.; Godefroid, P.; Murphy, B.; Maddila, C.; Williams, L. What are Weak Links in the npm Supply Chain? Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice, 2022, pp. 331–340. arXiv:2112.10165 [cs], doi:10.1145/3510457.3513044.
10. lmays. Security Scorecards for Open Source Projects, 2020.
11. Malicious code found in npm package event-stream downloaded 8 million times in the past 2.5 months, 2018.
12. Mitre. CNAs | CVE, 2023.
13. Mitre. Overview | CVE, 2023.
14. Mitre. CWE - About - CWE Overview, 2023.
15. Swinhoe, D. 7 places to find threat intel beyond vulnerability databases. https://www.csoonline.com/article/3315619/7-places-to-find-threat-intel-beyond-vulnerability-databases.html, 2018.
16. Urbanski, W. Day 0/Day 1/Day 2 operations & meaning - software lifecycle in the cloud age, 2021.
17. vipul1501. Dependency Graph in Compiler Design, 2022. Section: Compiler Design.
18. Vaszary, M. The EO and SBOMs: What your security team can do to prepare, 2023.
19. NIST. attack surface - Glossary | CSRC, 2023.
20. npm. vendorize, 2019.
21. Hastings, T.; Walcott, K.R. Continuous Verification of Open Source Components in a World of Weak Links. 2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, 2022, pp. 201–207.
22. Wang, X.; Sun, K.; Batcheller, A.; Jajodia, S. Detecting "0-Day" Vulnerability: An Empirical Study of Secret Security Patch in OSS. 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2019, pp. 485–492. ISSN: 1530-0889, doi:10.1109/DSN.2019.00056.
23. Idris Khan, F.; Javed, Y.; Alenezi, M. Security assessment of four open source software systems. *Indonesian Journal of Electrical Engineering and Computer Science* **2019**, *16*, 860. doi:10.11591/ijeecs.v16.i2.pp860-881.
24. Gkortzis, A.; Feitosa, D.; Spinellis, D. Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. *Journal of Systems and Software* **2021**, *172*, 110653. doi:10.1016/j.jss.2020.110653.
25. Ponta, S.E.; Plate, H.; Sabetta, A. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering* **2020**, *25*, 3175–3215. doi:10.1007/s10664-020-09830-x.
26. Pashchenko, I.; Plate, H.; Ponta, S.E.; Sabetta, A.; Massacci, F. Vuln4Real: A Methodology for Counting Actually Vulnerable Dependencies. *IEEE Transactions on Software Engineering* **2022**, *48*, 1592–1609. Conference Name: IEEE Transactions on Software Engineering, doi:10.1109/TSE.2020.3025443.

27. Prana, G.A.A.; Sharma, A.; Shar, L.K.; Foo, D.; Santosa, A.E.; Sharma, A.; Lo, D. Out of sight, out of mind? How vulnerable dependencies affect open-source projects. *Empirical Software Engineering* **2021**, *26*, 59. doi:10.1007/s10664-021-09959-3.