Assessing Security Vulnerabilities in Large Code Bases Using Open Source Software

Kristen Walcott *, Mark Vaszary *, Thomas Hastings

Posted Date: 25 February 2025

doi: 10.20944/preprints202412.1154.v2

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

Article

Assessing Security Vulnerabilities in Open Source Software Based on Exploit Risk

Kristen Walcott *,†, Mark Vaszary † and Thomas Hastings †

University of Colorado, Colorado Springs Address, 1420 Austin Bluffs Pkwy, Colorado Springs, CO 80918, USA
* Correspondence: kjustice@uccs.edu
† 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 [1]. The developed controls identified high levels of risk immediately.

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

1. Introduction

Usage of software that is ppen-source, also known as Open Source Software or OSS, is highly used in many a vast number of 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 [2]. According to TechMonitor, attacks like these against Open Source Software have increased 650% in 2021 alone [3] 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. OSS is leveraged and included in up to 97% of software applications currently in use today [4].

There exists way over five thousand OSS advisories related to security in GitHub today [5]. 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 [1].

Current state of the art includes using tools like Dependency graphs and Dependabot, of which both are available in GitHub [6]. There are also tools like Snyk [7], that operate similarly to the GitHub tools and monitor open source code. The tools have limited coverage and success for finding transitive

 doi:10.20944/preprints202412.1154.v2

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 risk score and (2) Incorporating continuous verification of security analysis of OSS components. Together, our primary contributions of work are comprised of the following:

* Description of a technique to examine used/potential components for vulnerabilities.
* Leveraging of the Open Source Security Foundation (OSSF) scorecard health metrics.
* Analyzing the component dependencies and the software bill of materials (SBOM) 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.
* Automated systems architecture for reviewing and sanitizing OSS components before and 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 [8]. 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 adversaries, any inherent 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.

There have been a 742% increase of software supply chain security attacks targeting OSS components in the 2022 calendar year [9]. Exploits of this nature continue to grow in frequency and occurrence. In the last two years, this topic has become of critical importance among both researchers and leaders of organizations. As a result of this, software practitioners have identified weaknesses in leveraged OSS packages [10], created new security practices and standards to highlight best practices for OSS component vetting [11], automated vulnerability look-ups for dependencies [12], and have widened the requirements and practices for reporting vulnerabilities and malicious packages [1,13].

The Open Source Security Foundation (OSSF) has completed a lot of great research and contributions in the domain of software supply chain protection. An OSSF project that we use extensively in our study is their Security Scorecards for Open Source Projects. Although not all of their available metrics were leveraged, that were available from their scorecards, we did add several metrics that heavily influenced our research. Researchers from Google [1] stated the following, “The goal of Scorecards is to auto-generate a ‘security score‘ for open source projects to help users decide the trust, risk, and security

 doi:10.20944/preprints202412.1154.v2

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 [11]".

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 [14]. 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" [15]. There are a number of other Open Source vulnerability sources, which supplement the NVD database [16]. 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" [17], 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 [18]. 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) [19] 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.

 doi:10.20944/preprints202412.1154.v2

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. We apply our exploit risk score to them, which is used to derive an exploit risk score for including the component as a dependency in an overall software application.

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 [20] 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 Sections 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 another way to think about the operations life-cycle. Instead of following a traditional software development life-cycle, the premise is that organizations can manage their IT operations using Shift Left 0, Shift Left 1, and Shift Left 2 nomenclatures [21]. Shift Left follows the practice of moving testing earlier in the development process, where in our case, we are moving security of the OSS components used to earlier in the development process. This better aligns with further incorporating the vulnerability management of OSS components into the overall Continuous Integration/Deployment (CI/CD) pipeline.

We leverage Shift Left 0, Shift Left 1, and Shift Left 2 in our evaluation. Shift Left 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 Shift Left 1 operations. Figure 1 shows the Shift Left 1 process is and 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 [22]. Shift Left 2 operations occur after the package is included and running in production. This is when maintenance and monitoring becomes a priority, as zero day vulnerabilities become a potential issue that could destroy the organization affected by them. The

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 25 February 2025 doi:10.20944/preprints202412.1154.v2

challenge is in determining how vulnerabilities become evident over the operations life-cycle with the goal of early detection of potential issues.

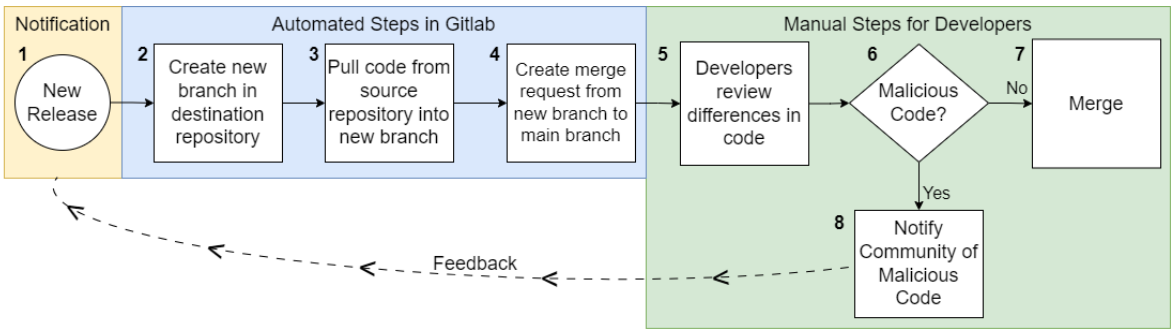

Figure 1. Shift Left 1 Process [1].

3. Results

Our work includes an assessment of the gaps in the currently available vulnerability assessments, as well as an proposing an exploit risk score to prioritize the assessed vulnerabilities. We examine the trade offs that contribute towards a practical exploit risk 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. Vulnerability 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 exploit risk 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.

Using the vulnerability assessment, we then apply an exploit risk score to rank the severity of the vulnerabilities introduced through the dependent libraries. Our exploit risk score is based on the attributes of the vulnerabilities, as learned from the NVD and supplementary vulnerability databases.

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. By rebuilding and expanding on the original six controls with additional controls (total of ten controls), we can more thoroughly understand the current state of a package and the package's community [21]. This helps to better inform the decision of whether or not to use the package. Our work revamps and adds to those original six controls for more in depth analysis, towards producing an overall exploit risk score.

- C1: Improving the checks on the package for known vulnerabilities in a package's dependencies and in the package itself, analyzing the history of vulnerabilities and the weaknesses causing the vulnerability.
- C2: Leveraging available static code analysis tools to review and analysis the source code for known weaknesses, with emphasis on known CWE information.

 doi:10.20944/preprints202412.1154.v2

- C3: Deeper dive into the package's community to understand the security posture of the community's maintainers and contributors.
- C4: Code and ecosystem complexity for the OSS component.
- C5: Architecture and threat modeling of the package has been completed.
- C6: Security documentation to make software practitioners more aware of security configurations and concerns when including the OSS component as a dependency.
- C7: Confirm there is an authorized and available Software Bill of Materials (SBOM) available for the OSS component.
- C8: Completing dynamic scanning of the overall application that is using the OSS component as a dependency.
- C9: Assessing the application before it is deployed into production with an application level penetration test.
- C10: Improving the hardening of the network perimeter defense around the development and production environment of the software project.

3.2.1. Evaluation

In our previous work we applied our first generation methods to two case studies that were compromised [1,21]. The first case study, UAParser.js, came from the NPM ecosystem and the second, rest-client, came from the RubyGems ecosystem. These OSS components were prime candidates because they are widely used and have many hundreds of millions of downloads. UAParser.js averages over two million downloads a week with over 610 million downloads in 2024 alone [**?**]. rest-client ranks 90 in RubyGems.org, with weekly downloads over one million [**?**]. Our deeper dive revisits these two case studies, applying our new exploit risk score to both.

Our controls are repeatable and capable of identifying exploit risk during Shift Left 0, Shift Left 1, and Shift Left 2 of the opensource component's operational life. These ten continuous verification controls enable organizations to make data-driven decisions and mitigate breaches resulting from known vulnerabilities, such as analyzing OSS community metrics and project hygiene using scorecards and monitoring the boundary of the software in production.

Case Study: UAParser.js

In our first case study, UAParser.js, our controls identified high levels of exploit risk immediately and continue to place this package as a High risk. Lots of software practitioners are actively using and integrating it within their applications, as it has over 7 million downloads a week. Our controls identified many potential risks caused by lack of proper security controls in place, beginning with Shift Left 0. Culminating with unexpected behavior identified with Shift Left 2, our Shift Left 0 control for dependents would have made a compelling case to implement mitigating controls or to look into alternative components/implementation options. Comparing our findings to that of the OpenSSF's Concise Guide for Evaluating Open Source Software, in Table 1 we can see their recommended controls have identified the Shift Left 0 risks as not being significant, as the OSSF scorecard presents an overall score of 8.1, as of January 30, 2025, which would make software practitioners believe this package is not a problem and has low risk of exploit. And in support of that, software practitioners continues to highly use the component with over 4 million packages. In addition, OpenSSF's controls would not have identified the malicious behavior on the network. Their framework does not advocate or use continuous network monitoring like our Shift Left 2 controls which is a core tenant of NIST's Risk Management Process.

Table 1. UAParser.js, Framework Comparison.

Operations	OSSF Framework	Our Framework
Shift Left 0	Low Risk	High Risk
Shift Left 1	Not in Scope	Effective
Shift Left 2	Not in Scope	Effective

 doi:10.20944/preprints202412.1154.v2

Shift Left 0. While performing controls 1–7, we identified indicators that make the component a likely future target for adversaries. First, there is complexity in the code base, along with a complicated ecosystem that is used when deployed. There is a significant number of lines of source code, spread across a small number of source files. Historically, there have been five reported vulnerabilities in the National Vulnerability Database (NVD), that all were identified either High or Critical in severity. Of those vulnerabilities, four of the five are similar, in regards to related to the categorized weaknesses. These are the known vulnerabilities, which are the ones reported by ethical security researchers. In addition, the project's community does not protect the main or release branches from maintainers directly committing to those branches and did not enforce signed commits or signed packages. Opportunity also presents itself for accidental weaknesses, as the project is highly maintained with a high number of commits and contributors from many different organizations and no architecture or threat modeling (C5), security documentation (C6) or official SBOM (C7). These controls alone identify enough risk for an organization to choose not to use the package, in spite of a high OSSF score. Our exploit risk analysis produced a score of 7.1, which following the NVD scale is a High risk.

Shift Left 1. Bringing the package under internal control could be an option, but any vulnerabilities discovered would need to be brought back to the OSS project community and fixed. Instead, controls C8 and C9 are preferred and recommended. Testing APIs with C8 would help reveal any issues, as related to improper access controls or handling of data through the API. C9 would help determine how exploitable the vulnerability truly is as architected, assuming there are other defenses in place to help protect against improper access control or manipulation of the regular expression inputs. These controls, if completed before October 23, 2021, would have helped to protect the organization before deploying it to production. These controls may have identified the malicious code and raised a "red flag" about trusting this component. Depending on the architecture of the overall application using this component, the exploit risk score can be decreased to a number lower than 7.1.

Shift Left 2. By monitoring the production environment, C10 can provide insights into abnormal behavior that could provide another opportunity for the organization to identify the malicious behavior of the package. If the malicious behavior was detected, the organization could have taken measures to protect itself, had the reviews in both Shift Left 0 and Shift Left 1 missed it. As the malicious code went outside of the network to download its malicious tooling, C10 would have identified the request, flagged it, and created a notification through the production monitoring system. Additional perimeter controls, such as a web application firewall (WAF), proxy server, etc., should have prevented this from occurring and potentially reduced the exploit risk score down to a low risk level near zero.

Overall, the methods identified many risks within the UAParser.js package, even with an acceptable OSSF scorecard score of 8.1. Our exploit risk analysis shows that it remains a High exploit risk at 7.1 for adversaries, due to several factors:

- The larger size of its source code base and usage of JavaScript.
- The high number of dependencies it has, as well as the large number of repositories and packages that are dependent upon it.
- The overall security posture is lacking, due to no official software bill of materials, lack of security related documentation, and no evidence of threat modeling completed.
- Repetitive vulnerabilities that have been High or Critical severity.

The fact that it is such a popular package, used by over 4 million packages, makes it a more attractive target for adversaries. But ultimately, each organization must chose to accept the risk when they include the package as a dependency in their application. However, our controls provide additional reviews to highlight risks for future exploit, for those organizations accepting the risk of using the UAParser.js.

Case Study: rest-client

Our exploit risk methods identified with Shift Left 0, a number of issues that make this component attractive to adversaries. As we saw in the UA Parser.js use case, our Shift Left 1 and 2 controls provide

 doi:10.20944/preprints202412.1154.v2

the most protection in this case study as well, for software practitioners that decide to leverage this component in their software application. When comparing our findings to that of the OpenSSF, in Table 2 we can see their recommended controls have identified the Shift Left 0 risks as High, as the OSSF scorecard presents an overall score of 3.3, as of January 30, 2025. However, OSSF does not provide any insights into the exploit risk of using the component.

Table 2. Rest-Client, Framework Comparison.

Operations	OSSF Framework	Our Framework
Shift Left 0	High Risk	High Risk
Shift Left 1	Not in Scope	Effective
Shift Left 2	Not in Scope	Effective

Shift Left 0. While performing controls 1-7, we continue to see that the package has not been maintained or modified since August 2019, and with the gaps observed with our other indicators, this component is a likely future target for adversaries. There is also complexity in the code base and the Ruby ecosystem that is used when deployed. This is further worsened, due to a lack of sufficient test code to support the complex code base. Analyzing the historical vulnerabilities in the National Vulnerability Database (NVD), there are three Critical CVEs that have occurred. Of those vulnerabilities, two of the three are similar, in regards to related to the categorized weaknesses and are related to remote code execution. As with US Parser.js, the project's community does not protect the main or release branches from maintainers directly committing to those branches; nor do they enforce signed commits or signed packages. There also is no architecture or threat modeling (C5), security documentation (C6) or official SBOM (C7). The lack of these controls show a lack of concern for security, which correlates with the low OSSF score as well. Our exploit risk analysis produced a score of 7.3, which following the NVD scale, is a High risk.

Shift Left 1. Applying our controls C8 and C9, would help to reveal the vulnerabilities around remote code execution and session fixation, which are the known vulnerabilities with this component. These controls would have protected the organization before the component was deployed to production. Depending on the architecture of the overall application using this component, the exploit risk score can be decreased to a number lower than 7.3.

Shift Left 2. Monitoring the production environment is of key importance, to identify and block exploits from actively occurring. C10, through the usage of additional perimeter controls (as outlined in the UA Parser.js use case), could have picked up the remote call to Pastebin.com and blocked it from occurring. This would have saved the organization from being compromised for five days. If C10 was in place, it could have reduced the exploit risk score down to a low risk level near zero. Having compensating controls in place in your production environment, is key if you are deciding to accept the exploit risk of using components that do not follow safe security practices.

In both case studies we found that the exploit risk is high and that our C8-C10 controls could have prevented malicious actions from adveraries [23].

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. [24] 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 [25], 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 [26]. Lastly, there are concerns that the current state of the art over-inflates the number of vulnerabilities found [8].

 doi:10.20944/preprints202412.1154.v2

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 [8]. Another involves using static and dynamic analyses to detect the reach-ability of vulnerable code in libraries. Eclipse Steady [27] 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 [28].

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 [8]. It is highly important that the number of dependencies are managed and timely updates are done for dependent libraries [29], whether it be by notification of vulnerability discovery or ease of library update.

5. Conclusions

Vulnerability assessment is a complicated and tedious process, which in the 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 can potentially exploit any vulnerabilities that you bring in with OSS components. In our work, we propose our techniques to improve and further automate vulnerability assessment and minimize the attack surface when using OSS components. By highlighting components that have a high exploit risk, software practitioners can make decisions on whether to implement risky components and/or to implement compensating controls to reduce the exploit risk in production. These techniques offer insights into how software applications can improve their vulnerability scanning capabilities and ensure that their limited resources are used in a manner most efficient to the product.

We provide a way to prioritize the vulnerabilities found, based on exploit risk, and areas to look for mitigating those risks, based on what we learned from studying usage of OSS components. By automating and addressing the gaps, not only from missed weaknesses/vulnerabilities but also from better understanding the risk itself, our work shows that the OSS community can gain benefit by continuing our efforts in further developing the exploit risk score and using it to measure the level of risk that an application may have if it continues to use the OSS components.

For future work, we would like to scale from our existing risk framework so that our controls can better evaluate the OSS components and look deeper into frameworks with many underlying or transitive dependencies. In addition, we would like to implement our controls and exploit risk score into an organizational setting with other software practitioners to see how it fits into their practices and development life cycle. Specifically on the impact the exploit risk score has with their threat modeling and architecture design for production deployment. The usefulness and efficiency of our framework can be assessed by surveying the software practitioners and obtaining their feedback. Lastly, training and using an artificial intelligence model on our framework would allow the automation of the evaluation of each OSS package and speed up the assessment provided. A much wider view of a package could be obtained through an automated recursive check of the underlying transitive dependencies.

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.

 doi:10.20944/preprints202412.1154.v2

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. Hastings, T.G. Combating Source Poisoning and Next-Generation Software Supply Chain Attacks Using Education, Tools, and Techniques. Ph.D., University of Colorado Colorado Springs, United States – Colorado, 2024. ISBN: 9798382262970.
2. Townsend, K. Cyber Insights 2023 | Supply Chain Security. url = https://www.securityweek.com/cyber-insights-2023-supply-chain-security/, 2023.
3. Fitri, A. Supply chain attacks on open source software grew 650% in 2021. url = https://techmonitor.ai/technology/cybersecurity/supply-chain-attacks-open-source-software-grew-650-percent-2021, 2021.
4. Plumb, T. GitHub's Octoverse report finds 97% of apps use open source software. url = https://venturebeat.com/programming-development/github-releases-open-source-report-octoverse-2022-says-97-of-apps-use-oss/, 2022.
5. Microsoft. Github advisory database. url = https://github.com/advisories, 2023.
6. Blog. Enable Dependabot, dependency graph, and other security features across your organization. url = https://github.blog/changelog/2020-07-13-enable-dependabot-dependency-graph-and-other-security-features-across-your-organization/, 2020.
7. Snyk. Enable Dependabot, dependency graph, and other security features across your organization. url = https://docs.dependencytrack.org/datasources/snyk/, 2023.
8. Pashchenko, I.; Vu, D.L.; Massacci, F. A Qualitative Study of Dependency Management and Its Security Implications. In Proceedings of the Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, New York, NY, USA, 2020; CCS '20, pp. 1513–1531. https://doi.org/10.1145/3372297.3417232.
9. 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.
10. Zahan, N.; Zimmermann, T.; Godefroid, P.; Murphy, B.; Maddila, C.; Williams, L. What are Weak Links in the npm Supply Chain? In Proceedings of the Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice, 2022, pp. 331–340. arXiv:2112.10165 [cs], https://doi.org/10.1145/3510457.3513044.
11. lmays. Security Scorecards for Open Source Projects, 2020.
12. Malicious code found in npm package event-stream downloaded 8 million times in the past 2.5 months, 2018.
13. Mitre. CNAs | CVE, 2023.
14. Mitre. Overview | CVE, 2023.
15. Mitre. CWE - About - CWE Overview, 2023.
16. Swinhoe, D. 7 places to find threat intel beyond vulnerability databases. url = https://www.csoonline.com/article/3315619/7-places-to-find-threat-intel-beyond-vulnerability-databases.html, 2018.
17. Urbanski, W. Day 0/Day 1/Day 2 operations & meaning - software lifecycle in the cloud age, 2021.
18. vipul1501. Dependency Graph in Compiler Design, 2022. Section: Compiler Design.
19. Vaszary, M. The EO and SBOMs: What your security team can do to prepare, 2023.
20. NIST. attack surface - Glossary | CSRC, 2023.
21. Hastings, T.; Walcott, K.R. Continuous Verification of Open Source Components in a World of Weak Links. In Proceedings of the 2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2022, pp. 201–207. https://doi.org/10.1109/ISSREW55968.2022.00068.
22. npm. vendorize, 2019.
23. Hastings, T.; Walcott, K.R. Continuous Verification of Open Source Components in a World of Weak Links. In Proceedings of the 2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, 2022, pp. 201–207.

 doi:10.20944/preprints202412.1154.v2

24. Wang, X.; Sun, K.; Batcheller, A.; Jajodia, S. Detecting "0-Day" Vulnerability: An Empirical Study of Secret Security Patch in OSS. In Proceedings of the 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2019, pp. 485–492. ISSN: 1530-0889, https://doi.org/10.1109/DSN.2019.00056.
25. 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. https://doi.org/10.11591/ijeecs.v16.i2.pp860-881.
26. 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. https://doi.org/10.1016/j.jss.2020.110653.
27. Ponta, S.E.; Plate, H.; Sabetta, A. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering* **2020**, *25*, 3175–3215. https://doi.org/10.1007/s10664-020-09830-x.
28. 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, https://doi.org/10.1109/TSE.2020.3025443.
29. 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. https://doi.org/10.1007/s10664-021-09959-3.