Trust, but Verify: Evaluating Developer Behavior in Mitigating Security Vulnerabilities in Open-Source Software Projects

Janislley Oliveira de Sousa[1,2], **Bruno Carvalho de Farias**[3], **Eddie Batista de Lima Filho**[2,4], **Lucas Carvalho Cordeiro**[2,3]

[1]Sidia Institute of Science and Technology, Manaus, Brazil

[2]Federal University of Amazonas (UFAM), Manaus, Brazil

[3]University of Manchester, Manchester, United Kingdom

[4]TPV Technology, Manaus, Brazil

janislley.sousa@sidia.com, bruno.farias@manchester.ac.uk,

eddie.filho@tpv-tech.com, lucascordeiro@ufam.edu.br

Abstract. *This study investigates vulnerabilities in dependencies of sampled open-source software (OSS) projects, the relationship between these and overall project security, and how developers' behaviors and practices influence their mitigation. Through analysis of OSS projects, we have identified common issues in outdated or unmaintained dependencies, including pointer dereferences and array bounds violations, that pose significant security risks. We have also examined developer responses to formal verifier reports, noting a tendency to dismiss potential issues as false positives, which can lead to overlooked vulnerabilities. Our results suggest that reducing the number of direct dependencies and prioritizing well-established libraries with strong security records are effective strategies for enhancing the software security landscape. Notably, four vulnerabilities were fixed as a result of this study, demonstrating the effectiveness of our mitigation strategies.*

1. Introduction

As widely known, modern software development often employs extensive third-party code from external libraries to save time, which usually comes from open-source software projects and offers numerous advantages, such as transparency, flexibility, and cost-effectiveness. Further analysis also reveals that developers frequently rely on these libraries even when carrying out simple tasks, instead of writing their code [Tang et al. 2022], and their centralized repositories make download and integration tasks easier, boosting productivity. However, they also present significant risks due to potential problems that can directly impact users [Plate et al. 2015].

Indeed, open-source third-party libraries may contain security vulnerabilities [Kula et al. 2018,Pashchenko et al. 2020]. While developers usually review their code for bugs and security issues using specialized tools, e.g., Coverity [Smith et al. 2020], CPPCheck [Marjamäki 2013], IKOS [Brat et al. 2014], and ESBMC v7.4 [Menezes et al. 2024], they often skip checking third-party libraries due to the extra effort involved in

their evaluation [Kula et al. 2018]. Going deeper, since a software project may depend on several open-source libraries, which may, in turn, depend on many other libraries in a complex package dependency network, analysis of a software project's entire dependency tree can become very complex.

In addition, the C programming language, which is widely used to develop critical open-source projects (e.g., operating systems, device drivers, and encryption libraries), lacks protection mechanisms such as bound checking and memory safety [Lipp et al. 2022], leaving developers responsible for memory and resource management [Berger et al. 2019]. This way, any lapse in this regard may result in undefined behavior, exposing a program to security vulnerabilities. Consequently, developers must be aware of these risks and ensure that pointers in operations such as subtraction, addition, and comparison belong to the same memory segment to prevent adverse outcomes.

Toward this need, the National Cybersecurity Federally Funded Research and Development Center (NCF), operated by the MITRE Corporation, oversees the common vulnerabilities and exposures (CVE) system [CVE], which regularly publishes newly identified open-source vulnerabilities. These vulnerabilities are documented in a comprehensive database with over $237,725$ entries, spanning across different languages, project types, and technologies.

Additionally, although security vulnerabilities, in an isolated manner, already constitute a significant challenge when creating applications with open-source code, they may also be boosted by other factors. Xiao *et al.* [Xiao et al. 2014] investigated several social factors impacting developers' adoption decisions, based on a multidisciplinary field of study called diffusion of innovations [Wermke 2023]. Their results indicate that security tools can compel developers to build more secure software by aiding in detecting and resolving vulnerabilities during the implementation and code review phases. However, it is essential to emphasize the importance of integrating security considerations throughout the entire software development lifecycle to ensure comprehensive protection. Moreover, conditions such as concerning behavior and lack of understanding regarding the consequences of security failures were identified in those whose primary activity is code writing [Assal and Chiasson 2018]. furthermore, while most open-source software projects have large communities contributing to their growth, some are not regularly maintained, which favors security issues [Wermke et al. 2022].

Regarding third-party libraries implemented in C language during open-source projects, it is crucial to adopt a critical perspective: developers should thoroughly examine open-source software to identify any vulnerabilities and potential backdoors [Zou et al. 2019]. Even when a project does not use specific vulnerable components directly, an element bundled in some linked package (e.g., third-party library or module) may cause problems and affect others by cascading effects defined as transitive dependency. In other words, examining source code and its documentation is essential to finding software vulnerabilities [Almarimi et al. 2020]. However, although many developers are mindful of secure-code best practices, there is no guarantee that they will follow all guidelines during development phases or integrate them into software processes. Moreover, some problems may still exist in the available code as it is challenging to detect security risks before software deployment [Gueye et al. 2021].

As initial and general perceptions, integrating third-party libraries in open-source projects has become a standard practice to expedite development and leverage existing solutions [Massacci and Pashchenko 2021]. However, this approach introduces significant security challenges [Tang et al. 2022]. On the one hand, while static analyzers often produce false positives, they can also identify genuine issues that will likely be overlooked during manual code reviews. On the other hand, formal verifiers, supported by mathematical proofs, produce significantly fewer false positives compared to static analysis tools [Švejda et al. 2020]. This ensures a higher level of accuracy in identifying vulnerabilities. Therefore, developers should balance dismissing all analyzer reports and addressing every single one.

Ultimately, the primary goal of software quality phases is to ensure software integrity and protect project results from potential threats, regardless of their origin, which includes avoiding risks related to bad practices and common assumptions. We examined various aspects related to the characteristics of the discovered vulnerabilities in the sampled projects' open-source dependencies. Our analysis revealed that developers' behaviors and practices significantly influence the mitigation of security vulnerabilities in third-party libraries within OSS projects. Consequently, this study aims to answer the following research questions:

- **RQ1:** *What are the Common Types and Prevalence of Dependency Vulnerabilities in Open-Source Software Projects?*
- **RQ2:** *How do developers' behaviors and practices influence the mitigation of security vulnerabilities?*
- **RQ3:** *What is the most effective strategy for mitigating risks from dependency vulnerabilities in open-source software projects?*

The remainder of this article is organized as follows: Section 2 describes the methodology used in this study, including the tools and techniques employed for vulnerability detection and analysis. Section 3 provides a detailed analysis of the identified vulnerabilities in various OSS projects, discusses how these vulnerabilities are managed by developers, and presents the outcomes of the remediation efforts, including the specific fixes applied. Lastly, Section 4 summarizes our findings, discusses the implications of developer behaviors on security practices, and offers recommendations for mitigating security vulnerabilities in open-source software projects.

2. Methodology

As already mentioned, software developers frequently use open-source libraries to accelerate development, but these libraries can contain security vulnerabilities, leading to high-profile incidents. Besides, as the use of open-source libraries grows, understanding, managing, and mitigating these dependency vulnerabilities becomes increasingly important [Prana et al. 2021].

In that sense, testing is inevitable. However, it is important to understand that software quality protocols are not simple evaluation sessions. Indeed, the complete process for software verification usually includes vulnerability identification, confirmation, code analysis, and code repair (e.g., patch application and merge requests to a repository), which may even be extended. Moreover, the entire chain begins with the vulnerability

identification step, which undoubtedly employs specialized tools, given that manual evaluation is impracticable for large projects.

This section presents the key concepts and technologies related to LSVerifier, an automated approach for software project evaluation. We focus on its structure and implementation for analyzing security vulnerabilities in open-source codebases.

2.1. Bounded Model Checking

BMC is a verification technique that detects errors up to a specified depth k, using Boolean Satisfiability (SAT) or Satisfiability Modulo Theories (SMT). Consequently, without a known upper bound for k, BMC cannot guarantee complete system correctness. In addition, as it only explores a limited state space by unwinding loops and recursive functions to a maximum depth, the state-explosion problem is inherently alleviated. In summary, this bounded nature makes BMC effective for uncovering fundamental errors in applications [Clarke et al. 2004, Gadelha et al. 2019]. Properties under verification are defined by

$$\text{BM}C_{\Phi}(k) = I(s_1) \wedge \left(\bigwedge_{i=1}^{k-1} T(s_i, s_{i+1}) \right) \wedge \left(\bigvee_{i=1}^{k} \neg\phi(s_i) \right), \tag{1}$$

where $I(s_1)$ is the set of initial states for a system, $\bigwedge_{i=1}^{k-1} T(s_i, s_{i+1})$ is the transition relation between time steps i and $i+1$, encompassing the evolution of the system over k steps, and $\bigvee_{i=1}^{k} \neg\phi(s_i)$ represents the negation the property ϕ at state s_i, indicating its violation within a bound k. Together, these components formulate a problem that is satisfiable if and only if a counterexample of length k or less exists, which includes the necessary information for its reproducibility.

2.2. LSVerifier

The LSVerifier tool [de Sousa et al. 2023a, de Sousa et al. 2023b] provides comprehensive support for the entire C11 standard, the current version of the C programming language. Unlike other tools, it can handle entire software projects and not only main entry functions, presenting high flexibility and coverage. It identifies software vulnerabilities by simulating a finite program execution prefix that includes all possible defined inputs, explicitly generating one symbolic execution per interleaving [Cordeiro and Fischer 2011]. By default, LSVerifier checks for buffer overflow, arithmetic overflow, invalid pointer access, improper buffer access, null pointer dereference, double free, division by zero, array bounds violations, pointer arithmetic violations, and user-specified assertions. Its verification process is shown in Figure 1.

LSVerifier conducts a comprehensive verification process by specifying the target source-code directory and the required configuration, including solver, encoding, and verification methods. Subsequently, all `.c` files inside the input directory are listed and examined using the Efficient SMT-based Context-Bounded Model Checker (ESBMC) [Gadelha et al. 2021], leading to the creation of a spreadsheet summarizing the obtained results.

The core BMC methodology employed by ESBMC involves unfolding a target system for a limited number of iterations and formulating a verification condition (VC). If the latter is satisfiable, it indicates a counterexample for a given property at a specific depth. ESBMC, in turn, is a robust and publicly available formal software verifier selected

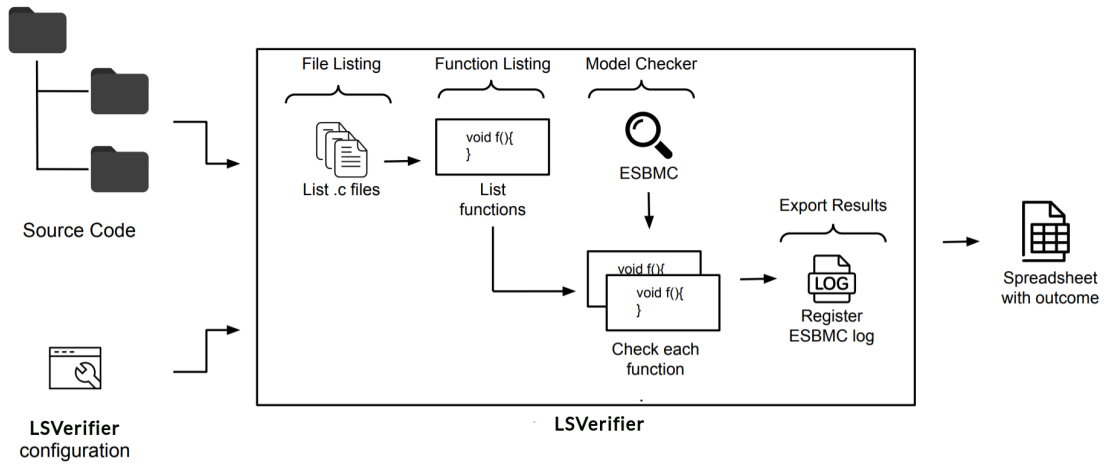

Figure 1. The LSVerifier's verification process involves specifying source-code directory and configurations, including solver, encoding, and verification methods. Violations are categorized and reported, with a detailed summary output in a spreadsheet.

as our BMC module for formal verification. ESBMC employs state-of-the-art incremental BMC techniques and *k*-induction proof-rule algorithms based on abstract interpretation, constraint programming (CP), and SMT solvers, whose effectiveness has already been demonstrated in various contexts [Beyer 2024]. The ESBMC's architecture is illustrated in Figure 2.

ESBMC employs several key components during its verification process. The Control-flow Graph (CFG) Generator handles C++ programs by including type-checking and static analysis, creating an Intermediate Representation (IR) for GOTO program generation, while for ANSI-C, it converts Abstract Syntax Trees (AST) into GOTO programs with additional checks and simplifications. The Symbolic Execution Engine symbolically executes the GOTO program, unrolling loops, generating Static Single Assignments (SSA) forms, and deriving safety properties for SMT solvers, including pointer safety checks. The SMT Back-end supports multiple solvers, encoding the SSA form into a formula to check satisfiability and generate counterexamples if a bug is detected.

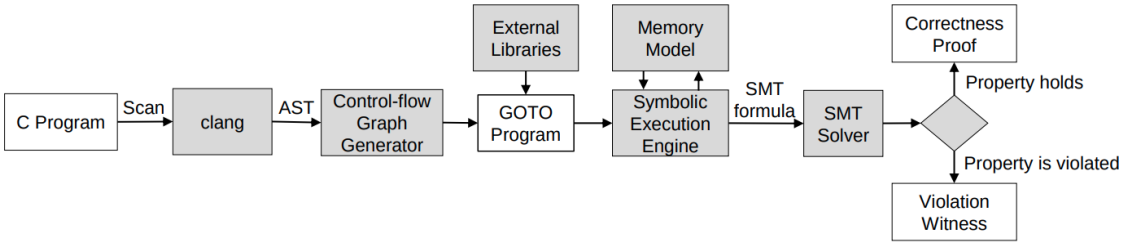

Figure 2. ESBMC verifier approach. White rectangles represent input and output and gray rectangles represent the verification steps. ESBMC uses several key components during its verification process [Gadelha et al. 2021].

Any property violations found during a verification procedure are communicated and categorized by LSVerifier, using a detailed report in a spreadsheet, which includes a

counterexample. The latter is a detailed path that demonstrates the violation of a specified property within a system. It includes a sequence of states and transitions, showing how a system evolves from an initial state to a condition where a property is violated. Consequently, this trace provides critical information for debugging as it pinpoints the exact sequence of operations leading to an error. By analyzing a counterexample, developers can understand the root cause of a violation and take corrective actions to fix the underlying issue.

2.3. Experiment Setup

All experiments described in this study were conducted on a system equipped with an Intel(R) Core(TM) i7-9750H computer processing unit (CPU) operating at 2.60 GHz, using 32 GB of RAM, and running Ubuntu 22.04. For benchmarking purposes, we curated a dataset comprising ten widely used software modules written in C: VideoLAN Client (VLC) in version 3.0.18, VI improved (VIM) in version 9.0.1672, terminal multiplexer (Tmux) in version 3.3a, reliable USB formatting utility (RUFUS) in version 4.1, OpenBSD secure shell (OpenSSH) in version 9.3, cross-platform make (CMake) in version 3.27.0-rc4, network data (Netdata) in version 1.40.1, Wireshark in version 4.0.6, Open Secure Sockets Layer (OpenSSL) in version 3.1.1, PuTTY in version 0.78, structured query language lightweight (SQLite) in version 3.42.0, and remote dictionary server (Redis) in version 7.0.11. All open-source software utilized in this research was distributed under open-source licenses, including GNU GPL, Apache, and MIT.

The following command was used to run LSVerifier on the entire set of OSS projects to analyze the codebase and identify potential vulnerabilities:

```
"$ lsverifier -r -f -l dep.txt".
```

The parameter `-l dep.txt` specifies a file containing paths for including header files from dependencies, ensuring that all necessary resources are considered. The parameter `-f` enables function verification, verifying individual functions within a codebase. Finally, the parameter `-r` enables recursive verification, ensuring that the verification process includes all nested functions and dependencies. These parameters together ensure a comprehensive analysis of the source code and its dependencies.

3. Empirical Study Results

In this section, we discuss the investigation results related to vulnerability aspects in OSS project dependencies. It also includes the relationship between such vulnerabilities and overall project security and how developers' behaviors and practices influence security vulnerability mitigation.

3.1. OSS Project Exploitation

The issues reported in this study were based on the counterexample traces provided by LSVerifier, during its analysis procedures. In this context, OSS projects were assessed according to the methodology outlined in Section 2.

Table 1 provides an overview of the issues reported, analyzed, and fixed in the chosen OSS projects. It is worth noticing that such issues were discussed with the respective developers and maintainers of the chosen OSS projects, which enabled us to evaluate and confirm many of them.

Table 1. Issues reported to the open-source software project repositories.

OSS project	Issues reported	Issues fixed
VLC	1 [Lhomme]	1
VIM	1 [Oliveira n]	0
RUFUS	2 [Oliveira l, Oliveira j]	1
OpenSSH	2 [Oliveira b, Oliveira a]	0
CMake	1 [Oliveira i]	1
Netdata	2 [Oliveira c, Oliveira h]	0
Wireshark	1 [Oliveira m]	1
OpenSSL	1 [Oliveira k]	0
SQLite	2 [Oliveira d, Oliveira e]	0
Redis	2 [Oliveira g, Oliveira f]	0

3.2. RQ1: What are the Common Types and Prevalence of Dependency Vulnerabilities in Open-Source Software Projects?

Our evaluation regarding developer behavior when mitigating security vulnerabilities, in open-source C projects, revealed critical insights into current practices and their implications.

In the VLC project, a pointer dereference issue was identified in the framebuffer third-party library [Lhomme]. A double-free error was identified as its cause, which is a type of vulnerability related to CWE-415 [MITRE]. Consequently, the respective maintainers decided to remove the Linux *fbdev* subsystem, which has been deprecated for over a decade, as superior alternatives are now available. This proactive approach led to an immediate impact on mitigating such vulnerabilities.

In our analysis of RUFUS, we identified property violations such as array bounds, division by zero, and invalid pointers [Oliveira l,Oliveira j]. Each issue highlights specific code errors and their implications, providing insights into the root causes and potential fixes for the identified vulnerabilities in RUFUS's software structure. However, when writing the present paper, we received only one bug fix for the library tiny-regex-c to address an out-of-bounds violation related to CWE-787 [MITRE]. Indeed, such behavior implies careless maintenance, which is a key aspect that can cause higher future impacts on system availability and reliability.

In the case of CMake [Oliveira i], developers promptly addressed a pointer dereferencing issue in the source code by adding a verification step before pointer usage, which was caused by an invalid pointer related to CWE-824 [MITRE]. This result highlights the importance of developers being aware of potential memory management issues and adopting defensive programming practices, such as boundary-checking on memory access operations. By prioritizing secure memory management practices, developers can mitigate serious security vulnerabilities in their projects.

In our investigation of Wireshark [Oliveira m], we uncovered common types of dependency vulnerabilities, including array access violations, related to CWE-125 [MITRE], and invalid and null pointers, related to CWE-824 and CWE-476 [MITRE], respectively. These vulnerabilities were identified in the CMake and network programming language (NPL) libraries, which are critical project dependencies. The issues stemmed

from dereference failures, caused by out-of-bounds access, and null pointer occurrences. Notably, the library NPL has not been actively maintained as its last significant update occurred approximately nine years ago. Besides, the most recent commit log reference dates back eight years. This lack of maintenance highlights the prevalence and risk of dependency vulnerabilities in OSS projects. To mitigate such problems and ensure the robustness and security of Wireshark, the development team decided to remove the library NPL, which is a significant result.

Finding 1

Based on the vulnerabilities identified and mitigated in this study, common types of dependency vulnerabilities in open-source software projects include pointer dereference issues, such as the double-free errors (CWE-415) found in VLC; array access violations, including out-of-bounds violations (CWE-787) in RUFUS; invalid pointers detected in CMake and Wireshark (CWE-824); and null pointer dereferences identified in Wireshark (CWE-476). These vulnerabilities were prevalent across multiple OSS projects, highlighting the importance of robust dependency management and proactive vulnerability mitigation strategies.

Finding 2

Such revelations underscore the importance of continuous monitoring and proactive management of third-party libraries to mitigate security vulnerabilities. The developers' responsive actions, such as removing deprecated subsystems and implementing verification steps, highlight the need for robust security practices in maintaining the integrity of open-source software projects.

3.3. RQ2: How do developers' behaviors and practices influence the mitigation of security vulnerabilities?

Understanding how developers' behaviors and practices influence the mitigation of security vulnerabilities is crucial for enhancing the security of OSS projects. Developer responses to the identified vulnerabilities, their approach to maintaining dependencies, and their willingness to adopt proactive security measures play a significant role in mitigating risks.

In OSS projects, diligent maintenance is crucial as it ensures the timely addressing of vulnerabilities, bug fixes, and compatibility updates. Neglecting these responsibilities can lead to unaddressed security flaws, reduced system performance, and increased risk of system failures. Consistent and careful maintenance practices are essential to sustain OSS projects' health, security, and reliability, ensuring they remain robust and dependable for users.

The SQLite project's response to identified violations underscores a prevalent challenge in the software development community: the inclination to dismiss static analyzer or formal verifiers results as "false positives" [Oliveira d, Oliveira e]. This practice stems from the belief that static analyzers often produce inaccurate results, causing unnecessary alarms and potentially wasting development resources. The SQLite team highlighted that they usually disregard these reports without concrete evidence, such as

an SQL script or specific code reproducing the issue. While pragmatic and aimed at preventing undue alarm, this stance carries significant risk. By dismissing these warnings, the team may overlook potential vulnerabilities that have not yet manifested and could be avoided. Indeed, it reveals the usual paradigm: corrections only arrive after a real problem, which shows a lack of proactivity and leads to higher losses.

The reliance on historical codebase performance further exacerbates this issue. During our discussions, the SQLite team noted their confidence in their codebase's historical stability, which they believe confuses static analyzers. This over-reliance can lead to complacency, resulting in missed opportunities to address latent issues before they become significant security threats. By not investigating potential false positives, in their opinion, developers may inadvertently leave their software susceptible to vulnerabilities that are initially difficult to detect but could have severe implications if exploited.

Besides, such an approach highlights a critical gap in development processes: the need for a balanced perspective on static analysis results. On the one hand, while it is true that not all warnings require immediate action, completely disregarding them without thorough investigation can undermine the overall security of a given software system. On the other hand, a more nuanced approach, where static analysis results are carefully evaluated and verified, can help identify genuine issues early, enhancing software security and robustness. Thus, developers must balance skepticism and due diligence to mitigate security vulnerabilities effectively.

Finding 3

There is a critical gap in the development process: the need for a balanced perspective on static analysis results. While static analyzers might produce false positives, they can identify genuine issues that might be overlooked during manual code reviews. In contrast, formal verifiers, supported by mathematical proofs, ensure a higher level of accuracy. Thus, developers must integrate both tools, balancing skepticism and due diligence, to enhance the overall security and reliability of software systems.

Similarly, an issue reported in OpenSSL involving an invalid pointer dereference, related to CWE-476 [MITRE], was acknowledged by the developers but not classified as a vulnerability. It happened because many OpenSSL APIs do crash if a null pointer is passed [Oliveira k]. However, this perspective reveals a problematic practice: developers frequently assume that certain conditions will never occur, dismissing potential vulnerabilities, which can be dangerous. If an attacker manipulates parameters or code to create these conditions, even using regular code contribution tools, the identified problem could lead to severe consequences, including system crashes or security breaches.

This example highlights the importance of changing the usual behavior of developers to make them address all identified issues, regardless of associated perceived likelihood. By not considering these scenarios as potential vulnerabilities, it is clear that developers leave their code open to exploitation. Addressing seemingly unlikely issues can prevent attackers from leveraging them to compromise the system. Besides, encouraging a proactive approach to vulnerability management, where all identified issues are investigated and resolved, is essential for maintaining robust security.

Moreover, Suc a practice highlights the need for developers to adopt conduct that anticipates and mitigates even the rarest of scenarios. This shift in behavior involves recognizing that assumptions about the improbability of certain conditions can lead to significant security gaps. A comprehensive approach to security should include evaluating and addressing all potential issues, ensuring that a software structure is resilient against a wide range of attacks. In conclusion, fostering a culture of thorough investigation and resolution of all identified vulnerabilities is crucial for the security and integrity of software projects.

Finding 4

Dismissing potential issues (e.g., buffer overflow and dereference failures) identified by static analysis or model-checking tools without thorough investigation can leave software susceptible to real threats. Embracing a balanced approach that considers manual testing and static analysis findings is essential for maintaining robust security in open-source C projects.

3.4. RQ3: What is the most effective strategy for mitigating risks from dependency vulnerabilities in open-source software projects?

As indicated by our analysis, the most effective strategy for mitigating risks from dependency vulnerabilities, in open-source software projects, is to reduce the number of direct dependencies. This can be achieved by carefully selecting and substituting multiple smaller libraries with a single, well-established library known for its strong security track record. This approach simplifies dependency management and leverages widely used and reputable open-source libraries' security practices and community support. Although it does not dismiss thorough analysis and careful evaluation, as previously suggested here, it may reduce both risk and revision workload.

The analysis of Redis revealed multiple violations [Oliveira g, Oliveira f], including array bound violations, related to CWE-787 [Oliveira g, Oliveira f], invalid pointer dereferences, related to CWE-476 [MITRE], null pointer dereferences, related to CWE-476 [MITRE], and out-of-bounds object access, related to CWE-119 [MITRE]. While some of these were confirmed as false positives, a significant oversight was identified: inadequate null pointer checks. Indeed, this oversight could lead to undefined behavior if a function is called with a null pointer. Even so, the Redis developers dismissed this issue, claiming that the function or method would never be invoked in a problematic way. In summary, this is a dangerous assumption as attackers could potentially exploit such scenarios. In addition, it is worth noticing that these issues were not false positives but instead, problems dismissed by wrong assumptions.

Finding 5

Functions from dependency libraries that are called in software modules should always be carefully verified by developers as they can be dangerous. Given that C programs often use pointers to access arrays and these are usually passed as arguments to functions, such a condition can bring serious security issues.

Overall, our results indicate that managing dependency vulnerabilities in open-source software is more effective when reducing direct dependencies rather than expanding development teams. This can be achieved, in open-source projects, by carefully selecting and replacing multiple smaller libraries with a single and well-established library known for its robust security track record. This approach simplifies dependency management and leverages widely used, reputable open-source libraries' security practices, and community support. By focusing on integrating libraries with a proven history of security and reliability, in addition to active maintenance and proactive coding, developers can minimize potential vulnerabilities and enhance the overall security posture of projects.

Finding 6

To mitigate risks from dependency vulnerabilities in open-source software, library management is more effective than increasing the number of contributors, project activity level, or project overall size. These results indicate that reducing the number of direct dependencies, for example, by substituting multiple smaller libraries with a single and well-established one known for its robust security record, is a key strategy.

4. Conclusion

Our findings emphasize the need for developers to adopt a more rigorous approach to security, particularly regarding third-party libraries. Despite their potential for false positives, static analyzers play a crucial role in identifying genuine issues that may be missed during manual reviews. Developers must also balance addressing these reports with a collaborative approach, working with security researchers and tool developers to validate and fix potential vulnerabilities. By fostering a culture that prioritizes security and encourages thorough examination of all potential risks, the open-source community can enhance the overall integrity and robustness of software projects.

This study demonstrated the effectiveness of these mitigation strategies, with four vulnerabilities being fixed in OSS projects VLC, RUFUS, CMake, and Wireshark. Such results highlight the importance of proactive dependency management and the need for integrating static analysis and formal verification tools into development processes.

Maintaining robust security in OSS projects requires a balanced approach that involves static analysis, formal verification, and collaborative efforts between developers and security experts. By reducing the number of direct dependencies and focusing on well-established libraries with strong security records, developers can significantly mitigate risks and improve the security posture of their projects. The open-source community must continue to prioritize security, adopt best practices, and remain vigilant in monitoring and managing dependencies to protect against emerging threats.

In conclusion, our analysis emphasizes the critical importance of diligent maintenance in OSS projects. Adopting these integrated approaches may lead to the evolution of new development methodologies where testing, evaluation, and analysis are as extensive as development activities. This balance ensures that OSS projects are not only rapidly developed but also rigorously vetted for security and reliability, resulting in a more resilient and trustworthy software ecosystem. Neglecting regular updates and vulnerability

management can lead to severe consequences. Moreover, our study suggests that integrating agile strategies with extensive testing, evaluation, and analysis phases can significantly enhance the development process. By treating these phases as equally important as development itself, we can foster new approaches that improve the robustness and dependability of OSS projects. This proactive stance is crucial for sustaining the health and longevity of open-source software.

Acknowledgment

The authors are grateful for the support offered by the SIDIA R&D Institute in the SEICO project. This work was partially supported by Samsung, using resources of Informatics Law for Western Amazon (Federal Law No. 8.387/1991). Therefore, the present work disclosure is in accordance as foreseen in article No. 39 of number decree 10.521/2020.

References

[Almarimi et al. 2020] Almarimi, N., Ouni, A., and Mkaouer, M. W. (2020). Learning to detect community smells in open source software projects. Knowledge-Based Systems, 204:106201.

[Assal and Chiasson 2018] Assal, H. and Chiasson, S. (2018). Security in the software development lifecycle. In Fourteenth symposium on usable privacy and security (SOUPS 2018), pages 281–296.

[Berger et al. 2019] Berger, E. D., Hollenbeck, C., Maj, P., Vitek, O., and Vitek, J. (2019). On the impact of programming languages on code quality: A reproduction study. ACM Transactions on Programming Languages and Systems (TOPLAS), 41(4):1–24.

[Beyer 2024] Beyer, D. (2024). State of the art in software verification and witness validation: Sv-comp 2024. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 299–329. Springer.

[Brat et al. 2014] Brat, G., Navas, J. A., Shi, N., and Venet, A. (2014). Ikos: A framework for static analysis based on abstract interpretation. In Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings 12, pages 271–277. Springer.

[Clarke et al. 2004] Clarke, E., Kroening, D., and Lerda, F. (2004). A tool for checking ansi-c programs. Lecture Notes in Computer Science, 2988:168–176.

[Cordeiro and Fischer 2011] Cordeiro, L. and Fischer, B. (2011). Verifying multi-threaded software using smt-based context-bounded model checking. In Proceedings of the 33rd International Conference on Software Engineering, pages 331–340.

[CVE] CVE, M. Cve list. `https://cve.org/`. Accessed 16 June 2024.

[de Sousa et al. 2023a] de Sousa, J. O., de Farias, B. C., da Silva, T. A., Cordeiro, L. C., et al. (2023a). Finding software vulnerabilities in open-source c projects via bounded model checking. arXiv preprint arXiv:2311.05281.

[de Sousa et al. 2023b] de Sousa, J. O., de Farias, B. C., da Silva, T. A., de Lima Filho, E. B., and Cordeiro, L. C. (2023b). Lsverifier: A bmc approach to identify security vulnerabilities in c open-source software projects. In Anais Estendidos do XXIII Simpósio

Brasileiro em Segurança da Informação e de Sistemas Computacionais, pages 17–24. SBC.

[Gadelha et al. 2019] Gadelha, M., Monteiro, F., Cordeiro, L., and Nicole, D. (2019). ESBMC v6.0: Verifying C Programs Using k-Induction and Invariant Inference. In Tools and Algorithms for the Construction and Analysis of Systems.

[Gadelha et al. 2021] Gadelha, M. R., Menezes, R. S., and Cordeiro, L. C. (2021). Esbmc 6.1: automated test case generation using bounded model checking. International Journal on Software Tools for Technology Transfer, 23(6):857–861.

[Gueye et al. 2021] Gueye, A., Galhardo, C. E., Bojanova, I., and Mell, P. (2021). A decade of reoccurring software weaknesses. IEEE Security & Privacy, 19(6):74–82.

[Kula et al. 2018] Kula, R. G., German, D. M., Ouni, A., Ishio, T., and Inoue, K. (2018). Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration. Empirical Software Engineering, 23:384–417.

[Lhomme] Lhomme, S. fb: remove support. `https://code.videolan.org/videolan/vlc/-/pipelines/227531`. Accessed 16 June 2024.

[Lipp et al. 2022] Lipp, S., Banescu, S., and Pretschner, A. (2022). An empirical study on the effectiveness of static c code analyzers for vulnerability detection. In Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 544–555.

[Marjamäki 2013] Marjamäki, D. (2013). Cppcheck: a tool for static c/c++ code analysis. URL: https://cppcheck. sourceforge. io.

[Massacci and Pashchenko 2021] Massacci, F. and Pashchenko, I. (2021). Technical leverage in a software ecosystem: Development opportunities and security risks. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pages 1386–1397. IEEE.

[Menezes et al. 2024] Menezes, R. S., Aldughaim, M., Farias, B., Li, X., Manino, E., Shmarov, F., Song, K., Brauße, F., Gadelha, M. R., Tihanyi, N., et al. (2024). Esbmc v7. 4: Harnessing the power of intervals: (competition contribution). In International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 376–380. Springer.

[MITRE] MITRE. Common weakness enumeration (cwe). Accessed 16 June 2024.

[Oliveira a] Oliveira, J. Bug 3382 - software vulnerabilities detected using esbmc-wr tool. `https://bugzilla.mindrot.org/show_bug.cgi?id=3382`. Accessed 16 June 2024.

[Oliveira b] Oliveira, J. Bug 3452 - potential software vulnerabilities detected using esbmc-wr tool. `https://bugzilla.mindrot.org/show_bug.cgi?id=3452`. Accessed 16 June 2024.

[Oliveira c] Oliveira, J. [bug]: Code properties violations found - dereference failure: invalid pointer #13219. `https://github.com/netdata/netdata/issues/13219`. Accessed 16 June 2024.

[Oliveira d] Oliveira, J. Code properties violations during software vulnerabilities investigation - bug report. https://sqlite.org/forum/forumpost/ac645ab114. Accessed 16 June 2024.

[Oliveira e] Oliveira, J. Code properties violations during software vulnerabilities investigation - bug report 2. https://www.sqlite.org/forum/forumpost/a2d232d413. Accessed 16 June 2024.

[Oliveira f] Oliveira, J. Linenoise and lua issues. https://github.com/janislley/lsverifier_final_results/blob/main/redis-7.0.11/issue%20report/Advisory_02.pdf. Accessed 16 June 2024.

[Oliveira g] Oliveira, J. Redis issue. https://github.com/janislley/lsverifier_final_results/blob/main/redis-7.0.11/issue%20report/Advisory_01.pdf. Accessed 16 June 2024.

[Oliveira h] Oliveira, J. Security vulnerabilities found in sqlite3.c. https://www.sqlite.org/forum/forumpost/3ffffb11d0. Accessed 16 June 2024.

[Oliveira i] Oliveira, J. Software vulnerabilities detected during code analysis with esbmc-wr tool. https://gitlab.kitware.com/cmake/cmake/-/issues/23132. Accessed 16 June 2024.

[Oliveira j] Oliveira, J. Software vulnerabilities detected during code analysis with esbmc-wr tool #103. https://github.com/tytso/e2fsprogs/issues/103. Accessed 16 June 2024.

[Oliveira k] Oliveira, J. Software vulnerabilities detected during code analysis with esbmc-wr tool #17560. https://github.com/openssl/openssl/issues/17560. Accessed 16 June 2024.

[Oliveira l] Oliveira, J. Software vulnerabilities detected during code analysis with esbmc-wr tool #76. https://github.com/kokke/tiny-regex-c/issues/76. Accessed 16 June 2024.

[Oliveira m] Oliveira, J. Software vulnerabilities detected in development tools. https://gitlab.com/wireshark/wireshark/-/issues/17897. Accessed 16 June 2024.

[Oliveira n] Oliveira, J. Vim issue. https://github.com/vim/vim/issues/9571. Accessed 16 June 2024.

[Pashchenko et al. 2020] Pashchenko, I., Vu, D.-L., and Massacci, F. (2020). A qualitative study of dependency management and its security implications. In Proceedings of the 2020 ACM SIGSAC conference on computer and communications security, pages 1513–1531.

[Plate et al. 2015] Plate, H., Ponta, S. E., and Sabetta, A. (2015). Impact assessment for vulnerabilities in open-source software libraries. In 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 411–420. IEEE.

[Prana et al. 2021] Prana, G. A. A., Sharma, A., Shar, L. K., Foo, D., Santosa, A. E., Sharma, A., and Lo, D. (2021). Out of sight, out of mind? how vulnerable dependencies affect open-source projects. Empirical Software Engineering, 26:1–34.

[Smith et al. 2020] Smith, J., Do, L. N. Q., and Murphy-Hill, E. (2020). Why can't johnny fix vulnerabilities: A usability evaluation of static analysis tools for security. In Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020), pages 221–238.

[Švejda et al. 2020] Švejda, J., Berger, P., and Katoen, J.-P. (2020). Interpretation-based violation witness validation for c: Nitwit. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 40–57. Springer.

[Tang et al. 2022] Tang, W., Xu, Z., Liu, C., Wu, J., Yang, S., Li, Y., Luo, P., and Liu, Y. (2022). Towards understanding third-party library dependency in c/c++ ecosystem. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pages 1–12.

[Wermke 2023] Wermke, D. (2023). Security considerations in the open source software ecosystem.

[Wermke et al. 2022] Wermke, D., Wöhler, N., Klemmer, J. H., Fourné, M., Acar, Y., and Fahl, S. (2022). Committed to trust: A qualitative study on security & trust in open source software projects. In 2022 IEEE Symposium on Security and Privacy (SP), pages 1880–1896. IEEE.

[Xiao et al. 2014] Xiao, S., Witschey, J., and Murphy-Hill, E. (2014). Social influences on secure development tool adoption: why security tools spread. In Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing, pages 1095–1106.

[Zou et al. 2019] Zou, J., Zeng, W., Zhao, Y., Liang, R., Cai, L., and Zhao, Y. (2019). Research on secure stereoscopic self-checking scheme for open source software. In Proceedings of the 2019 International Conference on Artificial Intelligence and Computer Science, pages 158–162.