Journal of Statistics Applications& Probability
An International Journal

http://dx.doi.org/10.18576/isl/130208

Exploring Code Vulnerabilities through Code Reviews: An Empirical Study on OpenStack Nova

Taleb Fahmawi[1], *Ahmad Nabot*[1,*], *Issam Jebreen*[1] *and Ahmad Al-Qerem*[2]

[1]Department of Software Engineering, Faculty of Information Technology, Zarqa University, Zarqa, Jordan
[2]Department of Computer Science, Faculty of Information Technology, Zarqa University, Zarqa, Jordan

Received: 7 Aug. 2023, Revised: 21 Sep. 2023, Accepted: 23 Oct. 2023
Published online: 1 Mar. 2024

Abstract: Effective code review is a critical aspect of software quality assurance, requiring a meticulous examination of code snippets to identify weaknesses and other quality issues. Unfortunately, the biggest threat to software quality is developers' disregard for code-writing standards, which leads to code smells. Despite their importance, code smells are not always identified during code review, creating a need for an empirical study to uncover vulnerabilities in code reviews. This study aimed to explore vulnerabilities in code reviews by examining the OpenStack project, Nova. After analyzing 4873 review comments, we identified 187 comments related to possible vulnerabilities, and a pilot study confirmed 151 of them as vulnerabilities. Our findings revealed that injection vulnerability flaws were the most prevalent, while insecure deserialization was the least common. Our study also identified three primary reasons for vulnerabilities: developers' knowledge of secure coding practices, unfamiliarity with existing code, and unintentional errors. In response to these vulnerabilities, reviewers suggested that developers fix the issues, and developers generally followed their recommendations. We recommend that developers receive training in secure coding practices to improve software quality, and those code review procedures include specific checks for common vulnerabilities. Additionally, it is essential to ensure that reviewers and developers communicate effectively to address vulnerabilities efficiently and effectively.

Keywords: Code Review, Security, Vulnerability, Code Review Comments, OpenStack

1 Introduction

According to McGrath [1], software security is a crucial aspect of software design to ensure that it continues to function appropriately even in the face of malicious attacks. Fu (2022) [23] also emphasizes that security is a vital quality attribute for all software systems. With the widespread use of software in various areas such as health, economy, and social applications, a high degree of security is required. Unfortunately, software development teams may lack the necessary mindset, expertise, and knowledge to establish security controls [2, 3]. Engineers may be unaware of vulnerabilities in essential software components and lack the skills to write secure code. A software vulnerability may allow an attacker to rob or tamper with sensitive information, link a system to a botnet, set up a backdoor, or implant other malware. Even if the design is flawless, a flaw in the program source code might result in a vulnerability. Code review is a systematic process where developers convene to verify each other's code for failures, bugs, and other issues, both functional and non-functional. Properly executed reviews can save time by simplifying the development cycle and significantly reducing the work required later by QA players. Reviews can save money by capturing bugs that might go undiscovered throughout validation, production, and even users' devices. Several studies have focused on code smell detection and removal techniques using tools such as PMD, SonarQube, and Designite [12, 14, 15]. However, according to Yamashita and Tahir [16, 17], software context and domain are crucial for vulnerability detection, which makes it difficult for automatic detection tools to be reliable. Therefore, this study used manual code smell detection, considered more reliable than automatic detection tools. The study used an OpenStack project called Nova to analyze the comments for the eight most prevalent vulnerabilities, including Flaw Injection, overflow, Exposure of sensitive data, Broken access control, Broken/Missing Authentication, Security misconfiguration, Insecure deserialization, and Insufficient Logging

* Corresponding author e-mail: anabot@zu.edu.jo

and Monitoring. The second section presents related work, the third section details the methodology, the fourth section presents the study results and discussion, and the fifth section presents the conclusions and future work.

2 Related Work

2.1 Software Vulnerabilities

According to Huang et al., [4], various factors and criteria influence software vulnerability and analysis equations through fuzzy synthetic decision-making. Ozment (2007) [5] defines software vulnerability as a fault in the software specification, development, or configuration that violates the explicit or implicit security policy during execution [32]. Ozment characterizes vulnerabilities based on a hypothetical life cycle and their status. Software vulnerabilities have many types, such as plugin-based web system vulnerabilities, including cross-site scripting, SQL injection, cross-site request forgery, path traversal, permissions management, improper input validation, code injection, information exposure, and unrestricted upload [6, 29–31]. Santos et al. (2019) [7] define vulnerability types from a tactical point of view, including improper input validation, functionality implication from untrusted environments, improper access control, cross-site scripting, origin validation error, control of generation of code, handling of insufficient privileges, certificate validation, privilege management, authentication, untrusted search path, handling of insufficient permissions or privileges, incorrect privilege assignment, foreign control of files, execution without necessary privileges, missing authorization, link following, command injection, and SQL injection. Bosu et al. (2014) [8] found that race condition vulnerability was the most identified type through peer code review. They worked on several projects such as Android, Chromium OS, Gerrit, ITK/VTK, MediaWiki, OmapZoom, OpenAFS, o Virt, Qt, and Typo3. They identified vulnerabilities such as buffer overflow, integer overflow, improper access, XSS, dos/crash, deadlock, SQL injection, format string, etc. Code review is known as code inspection or walkthroughs and is considered one of the first formalized processes for code review [9, 18]. Antunes et al. (2014) [3] define code reviews as an initial stage of code inspection, considered an informal code review. Code walkthroughs are an informal approach that manually analyzes the code by following its paths determined using predefined input conditions. Gerrit facilitates as a tool for code changes review, which allows authors to upload their initial version of code changes for review. Then, all changes are checked based on predefined conditions, such as verification bots. After that, others can review these changes and provide their comments and feedback to the authors [24, 25].

2.2 Security Code Review

According to Goodwin (2003), a security code review is a process that analyzes the code base of software to identify vulnerabilities that could potentially compromise the software's integrity, confidentiality, and availability [10]. When a security code review report is received, the developer must understand the security analyst's process to identify vulnerabilities. This can help the developer better understand how to fix the identified issues [10]. Code reviews are an effective technique for improving developers' security skills [26]. Weir et al. (2016) found that developers who participated in code reviews were better equipped to detect security vulnerabilities in code than those who did not [26]. Similarly, Braz et al. (2021) investigated how developers detect software vulnerabilities during code reviews and found that developers often found it easier to detect vulnerabilities after a code review [28]. The ability to identify security code issues during code reviews can depend on the developers' knowledge, perceptions, and mental attitudes towards detecting code vulnerabilities [28]. Three denominations of code review approaches were discovered as follows [10]:

–Manual Source Code Analysis, which entails scanning each source code file, line by line, manually.
–Semi-Automated Analysis employs software tools to aid the review process.
–Fully Automated Analysis uses the program to detect and report all known code vulnerabilities.

These different approaches have different levels of formality, rigor, and intensity, and they can be used depending on the specific needs of the software project and its development team.

2.3 OpenStack and Gerrit Review – Nova Project

OpenStack is a platform that comprises various software tools for managing and creating cloud computing environments [18]. It is one of the most active open-source projects in the world and has a large community [19]. The reason for choosing OpenStack for this study is its rich investment history in its code review process. Gerrit is a web-based code

review platform this project uses to offer online code reviews for projects, enabling developers to propose changes [20]. Other developers can review and suggest improvements to the changes. If changes need improvement, they can be updated with a new comment. Once the suggested modifications have been approved, they can be merged into the target branch of the GitHub repository using the Gerrit user interface. A reviewer can add comments to each modified line. This study focuses on mining reviewers' code review comments using Gerrit. The mining process begins by manually identifying search keywords for potential code vulnerabilities. After that, the collected comments are analyzed and checked for explicit or implicit comments related to software vulnerabilities within the identified scope.

3 Methodology

This study employs an empirical approach, widely recognized as the most appropriate method for collecting and analyzing comments related to code vulnerabilities. The data collection procedure and analysis are explained in detail in the following sections. Additionally, the comments related to vulnerabilities were manually classified according to the specific type of vulnerability within the scope of this study.

3.1 Research Questions

The research questions for this study aimed to investigate the code vulnerabilities identified during code reviews and the actions proposed by reviewers and developers to address these vulnerabilities. This aim is accomplished by developing three research questions (RQs) as follows:

- **RQ1: What are the most common software vulnerabilities discovered by code reviewers?**
  This RQ seeks to ascertain how frequently code reviewers identify vulnerabilities and which code vulnerabilities are commonly detected by reviewers.
- **RQ2: What are the most common causes of code vulnerabilities discovered during code reviews?**
  This RQ delves into the primary causes of the discovered vulnerabilities, as illustrated by the reviewers or developers. When reviewing a code, reviewers can illustrate why they suppose that any code under review may be vulnerable.
- **RQ3: How are the discovered vulnerabilities dealt with?**
  This RQ investigates how reviewers and developers deal with the discovered vulnerabilities.

3.2 Procedure

In the vulnerability identification phase, several steps are followed to identify work boundaries within software vulnerabilities. Second, a set of keywords is identified that includes both in-scope and out-of-scope vulnerabilities, as presented in Table 1. These vulnerabilities are selected based on the most prevalent vulnerabilities mentioned by specialized organizations such as OWSAP [**?**].

3.3 Building the Keywords Set

To construct the set of keywords, comments were screened for implicit or explicit references to software code vulnerabilities and filtered to retain only comments related to code vulnerabilities. Table 1 illustrates the keywords for searching and analyzing comments about the most common vulnerabilities.

Table 1: Software vulnerabilities and search keywords

Vulnerability Type	Keywords Set
Flaws Injection	SQL Injection, Flow Injection, XML Injection, HTML Injection, OS Command Injection, LDAP Injection, Malicious Code, Cross-Site Scripting, Input Validation, Code Injection, Injection Attack
Buffer Overflow	Stack Overflow Attack, Integer Overflow, Unicode Overflow, Overflow, Overflow Attack
Exposure To Sensitive Data	Encrypting Sensitive Data, Personal Data, Insecure Data Transmission, Sensitive Information, and Exposure Of Sensitive Data.
Broken Access Control	Access Control, Broken Access Control, Unprotected Admin Functionality, User Role Broken Access, URL-Based Broken Access Control, User ID Broken Access Control, Unauthorized
Broken/Missing Authentication	Bypass The Authentication, Application Session Timeouts, Passwords, Session Manager, Authentication
Security Misconfiguration	Unpatched Systems, Default Account Credentials, Unused Web Page, Poorly Configured Network, Documentation
Insecure Deserialization	Deserialization, Dos, Loads, Dump, Serialization
Insufficient Logging and Monitoring.	Logging, Monitoring, Unlogged, Missing Log, Obscure Error Logging
Common Keywords	Vulnerability, Attack, By-Pass, Vulnerable, Threat, Insecure

3.4 Pilot Study

To ensure the high reliability of the results, a pilot study was conducted in this study. The pilot study was carried out by three security professionals with over ten years of experience in the software development industry. After the review process, they reviewed 187 possible vulnerability comments and identified 151 comments as vulnerabilities-related.

3.5 Data Classification

The collected comments were analyzed and classified based on the type of code vulnerabilities in the analysis section. The classification included flaws in injection, buffer overflow, exposure of sensitive data, broken access control, broken/missing authentication, security misconfiguration, insecure deserialization, insufficient logging, and monitoring. To extract vulnerability-related comments, the comments provided by Gerrit were analyzed even if they did not explicitly mention vulnerabilities. Finally, the extracted comments were linked to their corresponding vulnerability type for classification.

4 Results and Discussion

This section presents the analysis results of the study RQs and the collected data.
RQ1: What are the most common software vulnerabilities discovered by code reviewers?
After conducting our search, we found that the term ”software vulnerability” and its types were not mentioned in most of the comments, which may be due to a lack of knowledge among developers about software security vulnerabilities and code security, as well as not considering security vulnerabilities during the review process. Despite this, we identified 151 approved vulnerability-related comments using the search keywords in Table 1. These vulnerabilities are not frequently identified during code reviews. However, flaws injection was the most frequently identified vulnerability with 41 comments, followed by broken/missing authentication with 23 comments and broken access control with 20 comments. Additionally, insufficient logging and monitoring were mentioned in 19 comments, buffer overflow in 15 comments, security misconfiguration in 13 comments, exposure of sensitive data in 12 comments, and insecure deserialization in 8 comments. The distribution of the identified vulnerabilities is presented in Figure 1. Figure 1

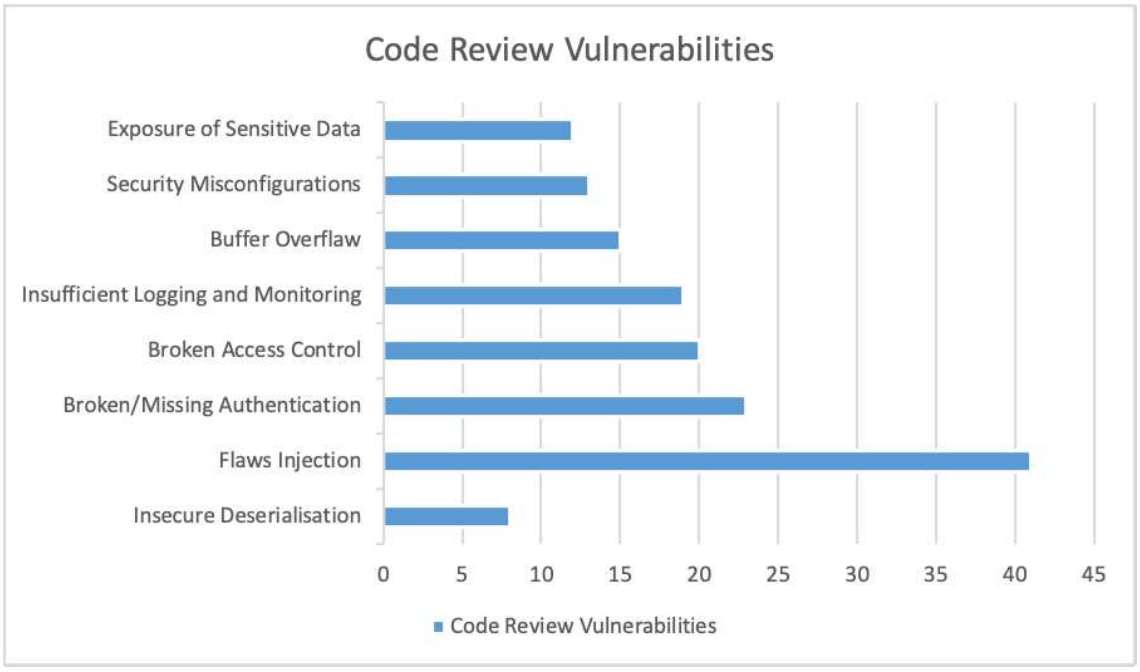

Fig. 1: Distribution of code comment vulnerabilities

illustrates that flaws injection was the most prevalent vulnerability in the analyzed code review comments, which allows attackers to inject malicious code into the system. Flaws injection can manifest in various forms, such as OS Command Injection, SQL Injection, and Cross-Site Scripting (XSS). For example, attackers can use SQL injection to exploit software and web application databases. Attackers typically identify a parameter the web application passes to a database interaction to exploit an SQL injection flaw. Therefore, to prevent these types of vulnerabilities, input and return values from users should be validated. Figure 2 depicts a reviewer's query asking if the value returned to the database is secure. In this scenario, attackers could use this code for SQL injection to inject the database with malicious code. A buffer

Fig. 2: SQL Injection Vulnerability

overflow vulnerability arises when the data volume exceeds the memory buffer's storage capacity. This could enable attackers to overwrite an application's memory, potentially inserting extra code and gaining access to IT systems. For instance, attackers may send new instructions to the application to exploit this vulnerability. Figure 3 presents an example of a reviewer's recommendation to set a constraint on the string input's maximum length and value type to prevent buffer overflow. After conducting the analysis, it was found that software vulnerabilities are not frequently

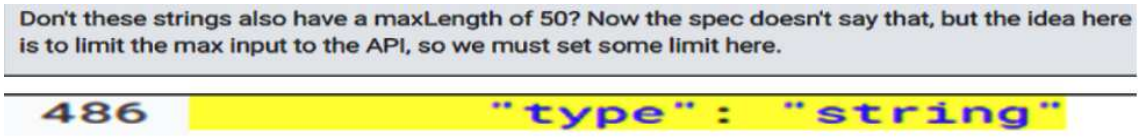

Fig. 3: Buffer overflow Vulnerability

identified during code reviews. Based on the results of RQ1, flaws injection and Broken/Missing authentication were the most common types of software vulnerabilities identified during the code review process. However, the exposure of sensitive data and insecure deserialization were not frequently mentioned in the reviewers' comments.

RQ2: What are the most common causes of code vulnerabilities discovered during code reviews?

To address RQ2, we analyzed the reviewers' comments to identify the common causes of code vulnerabilities, which are as follows:

–Cause 1: Lack of knowledge about secure coding practices.

Insufficient knowledge and experience in writing code while considering security practices lead to vulnerabilities. Figure 4 illustrates an instance of the lack of knowledge of secure coding practices where the developer failed to log a password without scrubbing, a critical security concern. This mistake results in a broken authentication vulnerability that enables attackers to access sensitive information or the system.

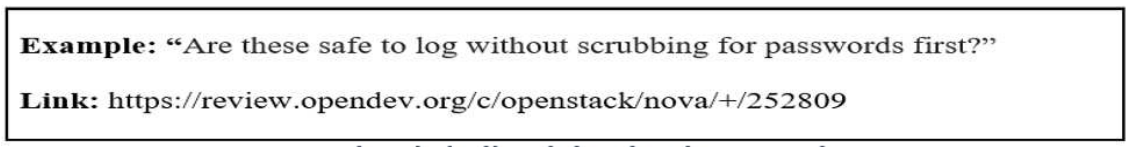

Fig. 4: Example of Lack of Knowledge of good security coding practices

–Cause 2: Unfamiliarity with existing code.
Unfamiliarity with code functionality or structure can lead to code vulnerabilities. An example of this is shown in Figure 5, where a developer's lack of familiarity with the existing code results in vulnerabilities in the software code.

Example: "Setting or re-setting password is not supported for the %s protocol."

Link: https://review.opendev.org/c/openstack/nova/+/622336

Fig. 5: Example of unfamiliarity with the existing code

–Cause 3: Unintentional mistakes of developers
Developers may inadvertently forget to fix code vulnerabilities or expose them accidentally. Figure 6 illustrates an example of how a developer can cause a vulnerability unintentionally. As seen in the figure, the developer admitted to not modifying the code that should have been changed due to an unintentional mistake [27]. These types of vulnerabilities are considered accidental mistakes caused by developers.

Example: "The change is verified...I must be missing the point."

Link: https://review.opendev.org/c/openstack/nova/+/93787

Fig. 6: Example of unfamiliarity with the existing code

This study identified three common causes of code vulnerabilities during the code review phase. First, according to reviewers' comments, the Lack of knowledge and skills for using good practices to write secure code was the most commonly identified cause of vulnerabilities. This highlights the importance of developers having appropriate training in secure code practices and standards to avoid potential vulnerabilities during development, which can increase software maintenance costs. Second, unfamiliarity with existing code was found to increase the likelihood of code vulnerabilities, which a lack of knowledge of secure code practices may also cause. Therefore, developers should prioritize understanding existing code before making modifications to reduce the cost of software maintenance. Finally, unintentional mistakes made by developers during the code-writing stage can also increase the chance of code vulnerabilities, bugs, and smells. It is crucial to avoid such mistakes through careful code review and testing. Overall, understanding and addressing these common causes of code vulnerabilities can improve the security and maintainability of software systems [22].

RQ3: How do developers deal with the discovered code vulnerabilities?
The focus of this RQ is to investigate the actions taken by code reviewers when identifying vulnerability issues and their suggested solutions for fixing these issues. Additionally, it explores the possible suggestions others provide to address these vulnerability issues. To address this RQ, the responses from code reviewers and the possible actions to fix these vulnerabilities are presented in Table 2. Most reviews, representing 80.7%, suggested a fix to address the identified code vulnerabilities. These fixes could be general instructions or specific actions to remove the vulnerabilities in the code. The reviewers may also provide a sample code snippet or a solution to assist the developer in fixing the vulnerability issue. However, 19.2% of the reviews just noted the presence of the vulnerabilities without providing any recommendations for refactoring.

Table 2: Suggested action by reviewers to resolve vulnerabilities

Reviewer's Suggestion	Count
Fix (without certain suggestion):	75
Fix (provided certain suggestion):	47
Just noted	29

Figure 7 shows an example of a solution suggested by a reviewer. In this example, the reviewer suggests validating the input, and then he provides a code example on GitHub. Table 2 outlines the recommended actions to address code

Fig. 7: Example of unfamiliarity with the existing code

vulnerabilities based on feedback provided by code reviewers. Table 3, on the other hand, presents the review number that corresponds to the code vulnerability and the suggested fixes for each vulnerability. After the review, the study found that the developers fixed or refactored 118 (78.1%) of the identified code vulnerabilities. The most commonly identified vulnerabilities by reviewers, flaw injection and broken/missing authentication, were widely refactored by developers, with over 82.9% of flaws injection and 82.6% of broken/missing authentication vulnerabilities addressed after being identified in the reviews. Figure 8 shows an example of a reviewer suggesting adding a validation schema to the code

Table 3: Developers' actions to code vulnerabilities identified during reviews

Code Vulnerability	Reviews Comment	Fixed By Developers	Percentage of fixes
Flaws injection	41	34	82.9%
Broken/Missing authentication	23	19	82.6%
Broken access control	20	15	75%
Insufficient logging and monitoring	19	14	73%
Buffer overflow	15	11	73.3%
Security misconfiguration	13	11	84%
Exposure to sensitive data	12	8	66.6%
Insecure deserialization	8	6	75%
Total	151	118	78.1%

to line 76; the developer then agrees to the reviewer's recommendation and adds the validation schema. Regarding the

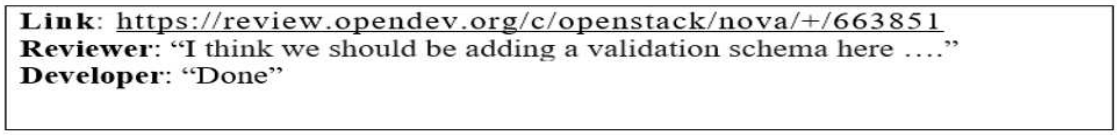

Fig. 8: Example of a fixed vulnerability comment

outcomes of RQ3 and RQ4, it was found that when code reviewers identify vulnerabilities, they typically provide practical recommendations, sometimes in the form of code snippets, and developers tend to follow these recommendations. In the context of sustainable reviews, developers generally agree with the vulnerability detection mechanism based on reviews and implement the suggested actions to address the issues. However, there were instances where reviewers' suggestions for modifications were ignored or rejected, which could have several reasons. These reasons may include the programmer's lack of understanding of the reviewer's comment, lack of awareness of potential bugs, or a belief that they are correct in rejecting the suggestion and that the reviewer is wrong and there is no bug present.

5 CONCLUSIONS

Code review is a widely recognized process for identifying and addressing software vulnerabilities. It is an effective way to improve the software's quality and reduce vulnerabilities [13]. However, despite its importance, code review is often overlooked by software developers, resulting in a lack of published research studies in this area. To fill this gap, an empirical study of code vulnerabilities during code review was conducted using the reviews from the OpenStack Nova projects. The study found that the most common vulnerabilities were flaws injection and broken/missing authentication, often caused by developers lacking the necessary knowledge and skills for writing secure code [21]. To address these vulnerabilities, the study suggests several actions: 1) developers should be more experienced with good security coding practices; 2) reviewers should consider code vulnerabilities during the review process; 3) developers should adhere to coding conventions to decrease code vulnerabilities; and 4) code vulnerability discovery via code reviews is a trustworthy approach [21]. Future work includes investigating code reviews in various projects from different societies, exploring the refactoring actions developers took to remove specific vulnerabilities, and identifying why programmers reject reviewers' suggestions or disregard the suggested modifications [21].

Acknowledgement

This research is funded by the Deanship of Research and Graduate Studies at Zarqa University /Jordan.
The authors are grateful to the anonymous referee for a careful checking of the details and for helpful comments that improved this paper.

References

[1] G. McGraw, Software security, IEEE Security & Privacy, **2**, 80-83 (2004).
[2] S. Millar, Vulnerability Detection in Open Source Software: The Cure and the Cause, Queen's University Belfast, (2017).
[3] N. M. D. S Antunes, Software Vulnerability Detection in Service-Based Infrastructures: Techniques and Tools, Doctoral dissertation, University of Coimbra, Portuguese, (2014).
[4] C. C Huang, F. Y Lin, S Lin, & Y. S Sun, A novel approach to evaluate software vulnerability prioritization, Journal of Systems and Software, **86**, 2822–2840, (2013).
[5] J. A Ozment, Vulnerability discovery & software security, Doctoral dissertation, University of Cambridge, UK, (2007).
[6] O. Mesa, R. Vieira, M. Viana, V. H Durelli, E. Cirilo, M. Kalinowski, & C. Lucena, Understanding vulnerabilities in plugin-based web systems: an exploratory study of wordpress, Proceedings of the 22nd International Systems and Software Product Line Conference, 149-15, (2018).
[7] J. C Santos, K. Tarrit, A. Sejfia, M. Mirakhorli, & M. Galster, An empirical study of tactical vulnerabilities, Journal of Systems and Software, **149**, 263-284, (2019).
[8] A. Bosu, J. C Carver, M. Hafiz, P. Hilley, & D. Janni, Identifying the characteristics of vulnerable code changes: An empirical study, Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, 257-268, (2014).
[9] C. Sadowski, E. Söderberg, L. Church, M. Sipko and A. Bacchelli, Modern code review: a case study at google, Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, 181-190, (2018).
[10] A. M Goodwin, causes and recommendations to resolve vulnerabilities that can arise in security code reviews, Citeseer, (2003).
[11] L. Braz, C. Aeberhard, G. Çalikli and A. Bacchelli, Less is more: supporting developers in vulnerability detection during code review, Proceedings of the 44th International Conference on Software Engineering, 1317-1329, (2022).
[12] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto and De A. Lucia, On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation, Proceedings of the 40th International Conference on Software Engineering, 482-482, (2018).
[13] E. Fregnan, J. Fröhlich, D. Spadini and A. Bacchelli, Graph-based visualization of merge requests for code review. Journal of Systems and Software, 195, p.111506, (2023).
[14] P. Becker, M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, Refactoring: improving the design of existing code, Addison-Wesley Professional, (1999).
[15] M. Abbes, F. Khomh, Y. G Gueheneuc and G. Antoniol, An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension, 15Th european conference on software maintenance and reengineering, 181-190, (2011).
[16] A. Yamashita and L. Moonen, Do developers care about code smells? An exploratory survey, 20th working conference on reverse engineering (WCRE), 242-251, (2013).
[17] A. Tahir, J. Dietrich, S. Counsell, S. Licorish and A. Yamashita, A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites, Information and Software Technology, **125**, 106333, (2020).
[18] R. Li, M. Soliman, P. Liang and P. Avgeriou, Symptoms of architecture erosion in code reviews: a study of two OpenStack projects, IEEE 19th International Conference on Software Architecture (ICSA), 24-35, (2022).

[19] A. Foundjem, E. Eghan and B. Adams, Onboarding vs. Diversity, Productivity and Quality—Empirical Study of the OpenStack Ecosystem, IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 1033-1045, (2021).
[20] E. A AlOmar, M. Chouchen, M. W Mkaouer and A. Ouni, Code Review Practices for Refactoring Changes: An Empirical Study on OpenStack, Proceedings of the 19th International Conference on Mining Software Repositories (MSR '22), New York, 689–701, (2022).
[21] L. Fu, P. Liang and B. Zhang, Understanding Code Snippets in Code Reviews: A Preliminary Study of the OpenStack Community, IEEE/ACM 30th International Conference on Program Comprehension (ICPC), 152-156, (2022).
[22] D. Cotroneo, L. D Simone, A.K Iannillo, R. Natella, S. Rosiello, & N. Bidokhti, Analyzing the Context of Bug-Fixing Changes in the OpenStack Cloud Computing Platform, IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), 334-345, (2019).
[23] L. Fu, P. Liang, Z. Rasheed, Z. Li, A. Tahir, and X. Han, Potential Technical Debt and Its Resolution in Code Reviews: An Exploratory Study of the OpenStack and Qt Communities, Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 216-226, (2022).
[24] F. E Zanaty, T. Hirao, S. McIntosh, A. Ihara and K. Matsumoto, An empirical study of design discussions in code review, Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 1-10, (2018).
[25] G. Kudrjavets, A. Kumar, N. Nagappan and A. Rastogi, Mining Code Review Data to Understand Waiting Times Between Acceptance and Merging: An Empirical Analysis, Proceedings of the 19th International Conference on Mining Software Repositories (MSR 2022), 579-590, NY, (2022).
[26] C. Weir, A. Rashid and J. Noble, How to improve the security skills of mobile app developers: comparing and contrasting expert views, Proceedings of the 2016 ACM Workshop on Security Information Workers, 810-821, USA (2016).
[27] Senanayake, Janaka, et al. Android source code vulnerability detection: a systematic literature review. ACM Computing Surveys, **55**, 1-37, (2023).
[28] L. Braz, E. Fregnan, G. Çalikli and A. Bacchelli, Why Don't Developers Detect Improper Input Validation?, IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 499–511,Madrid, (2021).
[29] A. Edmundson, B. Holtkamp, E. Rivera, M. Finifter, A. Mettler and D. Wagner, An empirical study on the effectiveness of security code review, International Symposium on Engineering Secure Software and Systems, 197-212, Berlin, (2013).
[30] N. Bhatt, A. Anand and V. S Yadavalli, Exploitability prediction of software vulnerabilities, Quality and Reliability Engineering International, **37**, 648-663, (2021).
[31] N. M Mohammed, M. Niazi, M. Alshayeb and S. Mahmood, Exploring software security approaches in software development lifecycle: A systematic mapping study, Computer Standards & Interfaces, **50**, 107-115, (2017).
[32] Rahman, Akond, et al., Security Misconfigurations in Open Source Kubernetes Manifests: An Empirical Study. ACM Transactions on Software Engineering and Methodology, (2023).