The Phantom Menace: Unmasking Security Issues in Evolving Software

Emanuele Iannone, Fabio Palomba
eiannone@unisa.it, fpalomba@unisa.it
Software Engineering (SeSa) Lab — University of Salerno, Fisciano, Italy

***Abstract*—Software security concerns the creation of secure software starting from its initial development phases, i.e., software that can withstand malicious attacks. To this end, several automated and not-automated solutions have been developed that support developers in identifying and assessing security issues, e.g., software vulnerabilities. However, most solutions were not meant to cooperate synergically or continuously run in the context of evolving software, i.e., software subject to frequent maintenance and evolution activities. In this scenario, developers have trouble setting up an effective *defensive line* against security issues arising in their projects. This research fills this gap by investigating how vulnerabilities affect evolving software projects and by proposing novel solutions to improve and simplify the security verification and validation process. The paper concludes by presenting the open challenges in the field of software security we framed while conducting our research.**

***Index Terms*—Software Security, Automated Software Engineering, Software Vulnerabilities.**

I. Context

Software security is commonly defined as the idea of *"engineering software so that it continues to function correctly under malicious attacks"* [1]. It encompasses all the phases of the software development lifecycle, from requirements to testing, having the ultimate goal to produce software that withstands attacks from malicious users, like denial of service or data theft. The core idea is to build an inherently secure design, adhering to the so-called *"security-by-design"* principle [1]. Software security includes methods and tools to detect and diagnose security issues—i.e., flaws in the design or implementation that might expose the system to attacks—that could arise in the developed system at any stage. For example, the absence of a proper mechanism to validate and/or sanitize externally-supplied data or the use of an outdated cryptography algorithm to send confidential data over an insecure channel [2], [3]. Most research efforts have been invested in defining new tools and techniques that *identify* vulnerabilities in source code or binaries. In this respect, there exists a wide range of automated detection tools [4], [5]. Most rely on static code analysis to spot recurring weak code patterns or detect unusual data flows. Some notable examples are Fortify [6] and Flawfinder [7]. Another relevant portion of analysis tools leverages concrete executions of the system to identify weird behavior hinting at the presence of vulnerabilities. For instance, OWASP ZAP [8] stimulates the tested system with specific inputs and interprets the response to understand whether there might be a vulnerability. American Fuzzy Lop [9] seeds the system with random inputs that are continuously mutated to maximize the chance of detecting crashes and buffer overflows. To a lesser extent, researchers have also been searching for new ways to *assess* the risk associated with security issues [10], [11]. Such solutions can either rely on experts' judgment—e.g., through the Common Vulnerability Scoring System (CVSS) [12]—or on fully-automated solutions, such as Eclipse Steady [10], [13]. What is more, over the last decades, even the development processes have been updated to address software security challenges. Microsoft defined the so-called *Security Development Lifecycle* (SDL),[1] a collection of best practices and tools to introduce security and privacy concerns since the first phases of software creation—a.k.a. the "shift left" principle. Similar concepts can be observed in the *Building Security In Maturity Model* (BSIMM),[2] which presents a large set of activities to put in action to address security threats at various levels, starting from adopting new governance mechanisms to managing configurations. What is more, all these concepts have been recently integrated into the tactics of *DevOps*, giving rise to *DevSecOps*, whose goal is to foster a close collaboration among the development, security, and operation teams [14].

Nevertheless, such methodologies and paradigms for secure development (and operation) appear not to be extensively adopted in practice [15], [16]. Indeed, developers find it difficult to strictly adhere to all the recommended practices for building secure software, mainly because they are deemed hard to comprehend and time-consuming—sometimes not even considered essential. What is more, the available automated instruments for *identifying* and *assessing* security issues are not designed to cooperate, i.e., they have overlapping capabilities, and the output of one cannot be easily used as input for another. In this respect, developers are forced to adopt ad-hoc solutions for setting up an effective and automated *defensive line* that contrasts security issues as soon as they emerge in the codebase. These obstacles become particularly relevant in the context of software that must evolve continuously and rapidly to meet the users' needs [17], causing an ever-growing increment in both its size and complexity. In other words, as more and more functionalities are added, the overall security level becomes more challenging to control.

[1] https://www.microsoft.com/en-us/securityengineering/sdl/practices
[2] https://www.bsimm.com/framework.html

II. Problem & Research Statements

In this context, we believe that the existing solutions for managing the rise of security issues need to be **re-engineered to make them work together synergically** in development contexts where changes are continuously committed. In other words, we aim to provide actionable contributions to *verification and validation* (V&V) of security issues affecting the source code in the context of continuously-evolving software. Hence, our research pursues this major goal:

> ◎ **The Ultimate Goal.** Envision an approach that supports the combination of existing security V&V solutions to maximize the chance of discovering a vulnerability and minimize the time required to react to them while upholding any project's constraint.

The current research has made limited progress in reaching this goal. Several studies have concluded that there is no silver bullet: the best way to maximize the vulnerability detection capabilities is to merge the results obtained by existing solutions [18], [19]. Nevertheless, the indiscriminate combination of multiple equivalent tools does not always lead to good results, as observed by Nunes et al. [20]. Neither machine learning can help solve this problem: using the tools' outputs as features of prediction models does not provide good enough results [21]. These preliminary findings strengthen the need for a smarter approach that efficiently combines existing solutions for improving the quality of the security verification & validation activities. Such an approach should not simply limit to running existing analysis tools and aggregating their findings in a standard format, but should (1) select the most suitable set of tools having the highest probability of finding real vulnerabilities in a target project, (2) configure them based on the project's constraints, e.g., time budget, (3) run them, and (4) harmonize the various outputs. We believe this research is a further step toward the straightforward creation of an automated *defensive line* against security threats that (1) continuously monitors new changes in a project's repository, (2) decides the security analysis to run, and (3) employs the best possible reacting actions in case of positive discovery. Such an approach should also consider all the socio-technical context of the project, e.g., the contribution flow, the allotted budget for running the security verification & validation pipeline, etc. To reach this goal, we need to conduct preliminary investigations and experiment with novel solutions for detecting and assessing software vulnerabilities. Only after gaining a solid knowledge of the research gap in security verification & validation can we focus on experimenting with practical solutions that will lead us to reach our main goal.

The first and most important step is to **gather additional knowledge** on how vulnerabilities appear in evolving software systems, studying how they are introduced, managed, and removed. We think this aspects is crucial for the success of our research as the literature lacks of empirical studies analyzing software vulnerabilities while taking into account the projects' histories—i.e., analyzing the flow of contributions and code changes. Addressing these aspects would help us discovering the challenges the developers face when dealing with vulnerabilities. In particular, we aim to understand the possible causes and circumstances in which developers commit changes that contribute to the insertion of vulnerable code. This sub-goal forms our first major research question:

> 🔍 **RQ$_1$.** *How do software vulnerabilities affect evolving software projects?*

Once obtained sufficient information on how vulnerabilities appear as a consequence of maintenance and evolution activities, we can **experiment with new solutions** that accelerate the detection and assessment of vulnerabilities. In particular, we envision novel detection and assessment approaches that can be run without delay after a new commit is made to a project. We think that the first layer of the defensive line should intercept vulnerabilities whenever they are being added to the code, reducing at their minimum their lifetime.

> 🔍 **RQ$_{2.1}$.** *To what extent can we detect software vulnerabilities when new changes are made into a project's repository?*

> 🔍 **RQ$_{2.2}$.** *To what extent can we assess software vulnerabilities when new changes are made into a project's repository?*

While working on **RQ$_{2.1}$** and **RQ$_{2.2}$**, we **compare** the effectiveness—in terms of detection performance—of existing vulnerability discovery tools, encompassing any kind of technique—i.e., static code analysis, dynamic analysis, and prediction modeling—and how they behave in the context of continuously-evolving projects. We also question how these tools achieve complementary findings to comprehend how they could be combined. We do not limit to automated tools but also **analyze the support** given by established practices for building secure code to developers during code review or other activities requiring human supervision.

> 🔍 **RQ$_{3.1}$.** *What is the effectiveness and complementarity of currently available solutions to identify vulnerabilities in evolving software?*

> 🔍 **RQ$_{3.2}$.** *What is the support given by secure development best practices and methodologies to identify vulnerabilities in evolving software?*

All these preliminary studies are instrumental in achieving the main goal. After answering all the described research questions, we ask ourselves whether it is possible to **combine existing solutions** into an adaptive automated approach, as previously described.

> 🔍 **RQ$_4$.** *Is it possible to combine any security V&V solution to maximize the discovery of vulnerabilities and minimize the reaction time while suiting the needs of the specific project?*

III. Methodological Approach

Our research employs several quantitative and qualitative research methods. We mainly carry out empirical studies that aim to find correlations among variables, such as understanding the socio-technical factors linked to an increase or decrease in the occurrence of vulnerabilities. We also present novel tools and techniques to address the challenges that emerge from the results observed in the empirical studies. Any solution we propose is validated following a rigorous evaluation design, e.g., comparing the performance with other existing solutions.

A. Data Collection

Most of our quantitative studies collect data through large-scale mining of software repositories employing state-of-the-art tools and techniques. In particular, we gather information on the change history of software projects and how developers make their contributions. In practice, the software projects are selected from GitHub using frameworks like PyDriller [22] to build custom scripts to run the mining and the data analyses. The data relating to known vulnerabilities observed in the past are collected from well-established public databases, such as the National Vulnerability Database (NVD), which is a comprehensive source of known vulnerabilities described by means of CVE (Common Vulnerabilities and Exposure) records. Their mining is enabled by calling web APIs (if available), downloading dumps, or scraping their web pages when other means fail. Whenever we need to hear the opinion of developers, we carry out surveys to collect many answers. For more in-depth insights, we rely on interviews and focus groups. We plan to reach out to the participants via GitHub or other platforms like Prolific[3] or Reddit[4] to quickly reach the largest number of developers.

B. Data Analysis

Any kind of data collected will be analyzed using traditional descriptive statistics. The statistical instruments can be different depending on the specific research question to answer. For instance, if the goal is to find correlations among two variables, we rely on correlation tests and/or the interpretation of regression models. We also enrich the analyses by computing effect size measures to give additional insights into the magnitude of the observed effects. On the other hand, if the goal is to evaluate a novel solution, we compare it with existing state-of-the-art solutions using standard evaluation metrics. The results are shown utilizing tables, box plots, and other graphs deemed instrumental in explaining the results obtained.

IV. State of Work

A. $\boldsymbol{RQ}_1$ – *Software Vulnerabilities in Evolving Software*

Achieved Results. We investigated appearance trend of software vulnerabilities in open-source systems [2]. Specifically, we focused on *how*, *when*, and *under which circumstances* developers contribute to introducing vulnerabilities in the codebase. Moreover, we analyzed *how long* they remain in the code and *which actions* are put in place to remove them. This large-scale investigation involved 3,663 known vulnerabilities—mined from the National Vulnerability Database—and 1,096 distinct projects hosted on GitHub. As part of our research, we mined the commits that (likely) contributed to the introduction of a vulnerability—known as Vulnerability Contributing Commits (VCCs)—by applying a modified version of the SZZ algorithm [23]–[25] starting from its public fixing commits. We also manually validated such an approach with two independent raters, observing 68% precision, which we deemed enough for our purposes. The key results showed that developers mainly contribute to vulnerabilities while doing maintenance activities (in 60.93% of the cases), more than half having the goal of fixing bugs. Vulnerabilities are generally not introduced within a single contribution: an average of four VCCs are required to introduce a vulnerability fully. In over 60% of the cases, the VCCs are spanned over four years, so they appear to be introduced "slowly". Once introduced, the vulnerabilities commonly affect a limited amount of files (1.43 on average). The experience does not matter: newcomer and expert developers can introduce vulnerable code—though the trend can differ according to the specific vulnerability types. The vast majority of vulnerabilities (83.99%) start appearing the year after the project's creation and are generally issued at least 30 days before a release. They remain in the code for a long time: half of them for over one year and a half. However, they are generally removed with simple code changes, such as escaping HTML entities or adding missing checks on the boundaries of a buffer. The results observed were deemed enough to answer our first research question.

Lesson Learned from $\mathbf{RQ}_1$

Vulnerabilities are a constant threat in evolving software. Even the most experienced developers need further support to improve their awareness of security issues and facilitate the detection of vulnerabilities promptly.

B. $\boldsymbol{RQ}_{2.X}$ – *Novel Security V&V Solutions*

Achieved Results. The findings observed when answering $\mathbf{RQ}_1$ led us to reason about how developers can intercept vulnerabilities before they are inserted into the source code, so that reducing the exposure window in which they can be exploited to attack the system. Hence, we experimented with how machine learning performs when detecting software vulnerabilities at the commit level, i.e., classifying potential vulnerability-contributing commits from "safe" commits [26]. We evaluated several learning algorithms using three sets of features extracted in different ways, i.e., (1) source code metrics extracted from the syntactic structure of the files changed in the commit, (2) process metrics describing the commit's characteristics within the change history, and (3) the counting of tokens extracted from the source code directly modified by the patch. We ran our experimentation on nine Java projects with public fixing commits available from NVD, so we could mine the VCCs using the same mining technique

[3] https://www.prolific.co [4] https://www.reddit.com

used in the previous study [2]. The study brought out two main findings: (1) ensemble learning algorithms perform way better than basic models, though without achieving remarkable results, and (2) combining different types of metrics does not always improve the classification performance.

However, even the most effective model cannot completely intercept all possible vulnerabilities. In our research, we did not limit to dealing with the problem of detecting new vulnerabilities but also evaluating how easily they can be exploited to carry out an attack. Thus, we developed SIEGE [27], an automated generator of exploit test cases that reach and execute any target piece of vulnerable code. Such an approach leverages a genetic algorithm that evolves a population of candidate exploits, having the goal of minimizing the "distance" between their execution traces and the target vulnerable components. SIEGE targets any code construct appearing in the project's classpath, including external libraries. Consequently, SIEGE can assess how risky it is to include a third-party dependency known to be affected by a vulnerability. The preliminary evaluation shows promising performance: SIEGE can produce valid test cases for 11 real-world vulnerabilities reachable from artificially-crafted projects.

Lesson Learned from $\mathbf{RQ}_{2.X}$

Commit-level detection of vulnerabilities requires further research and experimenting with state-of-the-art methods to improve the quality of the predictions. On the other side, SIEGE is the first step toward setting up a "reaction" pipeline that mitigates the threat of external vulnerabilities.

Ongoing Work. SIEGE runs a combination of static and dynamic analyses on third-party vulnerabilities to assess their exploitability. Yet, we believe that such an approach could fail in certain circumstances. If SIEGE fails the generation of test cases, then we cannot conclude that the target vulnerability is unexploitable. For this reason, we have been experimenting with how machine learning can detect likely-exploitable vulnerabilities leveraging only the textual information available at the time of their disclosure. We extract features from the unstructured text in online discussions mentioning such vulnerabilities and evaluate several learning configurations.

Planned Work. SIEGE is a research prototype for which we have planned several extensions. For instance, we plan to improve its test generation engine in terms of efficiency and explainability—e.g., describing the methods the test cases invoked to reach the vulnerability. SIEGE will also be evaluated with real-world projects affected by real vulnerabilities.

C. $\boldsymbol{RQ}_{3.X}$ – *Effectiveness of Existing Security V&V Solutions*

Ongoing Work. Currently, we have been investigating on the extent to which developers *know*, *understand*, and *adopt* best practices to implement secure software. In this study, we ask which challenges the developers face when they consult security guidelines—e.g., SEI CERT Coding Standards—or adopt secure development methodologies—e.g., Microsoft's SDL. The data will be collected through surveys spread online on channels like PROLIFIC or REDDIT. Our goal is to identify the set of possible actions that guidelines designers can put in place to make them more comprehensible and actionable even to inexperienced developers.

Planned Work. There is no clear sign in the literature on how the different vulnerability detection techniques (i.e., static analysis, dynamic analysis, code review, etc.) perform. We plan to fill this knowledge gap by providing an empirical comparison of a wide range of tools, techniques, and methods to discover vulnerabilities. This investigation aims to profile the effectiveness of the various existing techniques to understand whether they achieve complementary findings, motivating the need to create an engineered approach that combines them.

D. $\boldsymbol{RQ}_4$ – *Combining Security V&V Solutions*

Planned Work. As soon as we answer all the previous research questions, we are ready to define the approach that combines all the existing solutions and draws the best from them. Such an idea can be designed in several ways. For instance, we could implement the use of a *meta-classifier* that suggests the best set of solutions (e.g., tools) to be adopted, having the highest chances of finding vulnerabilities. Such a meta-classifier would leverage the specific project's characteristics—i.e., product and process metrics—to make its decisions. Then, a *software agent* would configure and instantiate the solutions recommended by the meta-classifier while fulfilling any project's constraints, e.g., the available time budget. We plan to validate the effectiveness and usefulness of this solution with real developers—e.g., in a controlled experiment—and real-world projects in the wild—e.g., collecting usage statistics from projects having the agent installed.

V. OPEN CHALLENGES

Software security has been attracting the attention of software engineering researchers, who has been striving to develop novel solutions for increasing the dependability of software systems. Our research aims to advance the state of the art in terms of verification & validation of security issues. During our research, we identified additional challenges that the research community should consider.

C1 – Security V&V should be more automated. Experienced developers rely on personal knowledge to design secure systems, while newcomers have trouble comprehending the recommended practices. Novel automated tools and recommendation systems are required to simplify and hasten the adoption of secure development practices and processes.

C2 – Security V&V should be more "Continuous". Most existing automated tools are meant to be run in a standalone fashion and to process only a single snapshot of a project, without considering its evolution over time. Indeed, researchers commonly experiment with "throw-away" solutions that are either not released at all or difficult to be employed in practice. Hence, research works should not limit experimenting with a yet-another security tool but also releasing actionable solutions for the software engineering community.

REFERENCES

[1] G. McGraw, "Software security," *IEEE Security & Privacy*, vol. 2, no. 2, pp. 80–83, 2004.

[2] E. Iannone, R. Guadagni, F. Ferrucci, A. De Lucia, and F. Palomba, "The secret life of software vulnerabilities: A large-scale empirical study," *IEEE Transactions on Software Engineering*, pp. 1–1, 2022.

[3] A. Austin and L. Williams, "One technique is not enough: A comparison of vulnerability discovery techniques," in *2011 International Symposium on Empirical Software Engineering and Measurement*, 2011, pp. 97–106.

[4] H. Shahriar and M. Zulkernine, "Mitigating program security vulnerabilities: Approaches and challenges," *ACM Comput. Surv.*, vol. 44, no. 3, jun 2012. [Online]. Available: https://doi.org/10.1145/2187671.2187673

[5] A. Kaur and R. Nayyar, "A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code," *Procedia Computer Science*, vol. 171, pp. 2023–2029, 2020, third International Conference on Computing and Network Communications (CoCoNet'19). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050920312023

[6] M. F. CyberRes, "Fortify static code analyzer." [Online]. Available: https://www.microfocus.com/en-us/cyberres/application-security/static-code-analyzer

[7] D. A. Wheeler, "Flawfinder." [Online]. Available: https://dwheeler.com/flawfinder/

[8] OWASP, "Zed attack proxy." [Online]. Available: https://www.zaproxy.org/

[9] M. Zalewski, "American fuzzy lop." [Online]. Available: https://lcamtuf.coredump.cx/afl/

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

[11] J. Jacobs, S. Romanosky, B. Edwards, I. Adjerid, and M. Roytman, "Exploit prediction scoring system (epss)," *Digital Threats*, vol. 2, no. 3, jul 2021. [Online]. Available: https://doi.org/10.1145/3436242

[12] C. SIG, "Common vulnerability scoring system." [Online]. Available: https://www.first.org/cvss/

[13] E. Foundation, "Eclipse steady." [Online]. Available: https://projects.eclipse.org/proposals/eclipse-steady

[14] R. N. Rajapakse, M. Zahedi, M. A. Babar, and H. Shen, "Challenges and solutions when adopting devsecops: A systematic review," *Information and Software Technology*, vol. 141, p. 106700, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584921001543

[15] D. Geer, "Are companies actually using secure development life cycles?" *Computer*, vol. 43, no. 6, pp. 12–16, 2010.

[16] E. Venson, R. Alfayez, M. M. F. Gomes, R. M. C. Figueiredo, and B. Boehm, "The impact of software security practices on development effort: An initial survey," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019, pp. 1–12.

[17] M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[18] N. Imtiaz, S. Thorn, and L. Williams, "A comparative study of vulnerability reporting by software composition analysis tools," 2021. [Online]. Available: https://doi.org/10.1145/3475716.3475769

[19] A. Algaith, P. Nunes, F. Jose, I. Gashi, and M. Vieira, "Finding sql injection and cross site scripting vulnerabilities with diverse static analysis tools," in *2018 14th European Dependable Computing Conference (EDCC)*, 2018, pp. 57–64.

[20] P. Nunes, I. Medeiros, J. Fonseca, N. Neves, M. Correia, and M. Vieira, "An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios," *Computing*, vol. 101, no. 2, pp. 161–185, Feb. 2019.

[21] J. D. Pereira, J. R. Campos, and M. Vieira, "Machine learning to combine static analysis alerts with software metrics to detect security vulnerabilities: An empirical study," in *2021 17th European Dependable Computing Conference (EDCC)*, 2021, pp. 1–8.

[22] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," 2018.

[23] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 1–5. [Online]. Available: https://doi.org/10.1145/1083142.1083147

[24] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 426–437. [Online]. Available: https://doi.org/10.1145/2810103.2813604

[25] L. Yang, X. Li, and Y. Yu, "Vuldigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes," in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, 2017, pp. 1–7.

[26] F. Lomio, E. Iannone, A. De Lucia, F. Palomba, and V. Lenarduzzi, "Just-in-time software vulnerability detection: Are we there yet?" *Journal of Systems and Software*, p. 111283, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121222000437

[27] E. Iannone, D. D. Nucci, A. Sabetta, and A. De Lucia, "Toward automated exploit generation for known vulnerabilities in open-source libraries," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, 2021, pp. 396–400.