A Synopsis of Static Analysis Alerts On Open Source Software

Nasif Imtiaz
simtiaz@ncsu.edu
North Carolina State University

Laurie Williams
lawilli3@ncsu.edu
North Carolina State University

ACM Reference Format:
Nasif Imtiaz and Laurie Williams. 2019. A Synopsis of Static Analysis Alerts On Open Source Software. In *Hot Topics in the Science of Security Symposium (HotSoS), April 1–3, 2019, Nashville, TN, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3314058.3317295

1 INTRODUCTION

Static application security testing (SAST) tools detect potential code defects (alerts) without having to execute the code. SASTs are now widely used in practice by both commercial and open source software (OSS). Prior work found that half of the state-of-the-art OSS projects have already employed automated static analysis [1]. However, little public information is available regarding the *actionability* (important to developers to act upon) of SAST alerts.

In prior work, researchers investigated what type of static analysis alerts are enabled or disabled by OSS projects [1] and what type of alerts are most likely to cause build failure when SASTs are used within a continuous integration (CI) environment [4]. The goal of this paper is *to aid researchers in improving the usability of static application security testing tools by looking at what type of static analysis alerts are most likely to be acted on by OSS developers.*

We investigate five OSS projects that use Coverity Scan, an SAST service. Based on the alerts generated by Coverity Scan for these five projects and developer responses to those alerts, we ask the following questions:

- **RQ1:** What are the alert types that are most often introduced, triaged, and eliminated?
- **RQ2:** What are the alert types that are most likely to be unactionable?
- **RQ3:** What is the median lifespan of each alert type?
- **RQ4:** Are security alerts more likely to be triaged and eliminated than non-security alerts?

2 METHODOLOGY

We investigate five OSS projects that use Coverity Scan as an SAST service. In this section, we briefly describe Coverity Scan and the selected OSS projects. We then describe the data analysis methods of our empirical study.

Coverity Scan: Coverity Scan is a free SAST service for open source projects provided by Synopsys[1]. As per the 2017 annual report, Coverity Scan hosts 4,600 active projects and claims to have detected 1.1 million defects (as alerts) with 600,000 of those defects being fixed afterwards [3]. Coverity Scan also maintains a defect database on its own cloud server for each project that it hosts. The defect database contains information regarding all the alerts and their respective triage history.

Project Selection: We choose five large and popular open source projects that 1) are written in C/C++ language; and 2) have at least 100 analysis reports (historical data) on Coverity Scan. We gather information of all the alerts that are available on Coverity Scan database till January, 2019. Table 1 lists information on these five projects.

Table 1: Analyzed Projects

Project	Analyzed Code Lines	Defect Reports Since	# of Defect Reports
Linux	14,455,530	May 17, 2012	592
Firefox	8,223,984	February 22, 2006	662
Qt	5,988,259	July 12, 2014	196
Samba	2,925,363	February 23, 2006	714
Kodi	111,054,632	August 28, 2012	388

Analysis: Our dataset contains 24 high-level alert types (marked as "Category" by Coverity Scan) along with respective impact priority (high, medium, or low), where a single alert type consists of multiple low-level alert types (rule checkers). Developers can triage alerts on Coverity Scan by manually updating any information on the alert (e.g. severity, action) and try to determine the veracity of the alert [2]. If an alert is not triaged, we can not determine if and how any developer has responded to that alert from our dataset. For each project, we count how many alerts were detected (Median Count) and how many of those alerts were triaged by developers (Triage rate) for an alert type. We then count how many of the triaged alerts were eliminated (Fix rate). We also count how many alerts were explicitly marked by developers as false positive (FP rate), or intentional (Intentional rate), or as a bug (Bug rate). For each alert type, we then report the median for each metric that we examine. For the eliminated alerts, we count lifespan by taking the time difference between the last and the first time they were detected. Furthermore, Coverity Scan marks each alert as a quality issue, a security issue, or both (marked as "various"). We count an alert as security if it is marked as security or various, and non-security if otherwise. For each of the 5 projects in this study, we measure if security alerts are more likely to be triaged and if triaged, if security alerts are more likely to be fixed than non-security alerts

[1] https://scan.coverity.com/

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
HotSoS, April 1–3, 2019, Nashville, TN, USA
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7147-6/19/04…$15.00
https://doi.org/10.1145/3314058.3317295

Table 2: Findings - RQ1, RQ2, RQ3 (Ordered by Triage Rate)

Alert Category	Impact	Median Count	Triage Rate	Fix Rate	FP Rate	Intentional Rate	Bug Rate	Lifespan (weeks)
Memory - Illegal Access	High	154	65.6%	56.3%	31.3%	12.5%	18.8%	5.0
Memory - Corruptions	High	146	60.3%	39.2%	24.3%	7.5%	14.9%	31.9
Security Best Practices Violations	Low	132	50.0%	72.9%	0.5%	8.3%	0.0%	15.9
Null Pointer Dereferences	Medium	813	39.7%	64.6%	17.6%	2.9%	41.0%	3.4
Resource Leaks	High	831	33.9%	67.9%	14.4%	3.3%	32.5%	17.7
Parse Warning	Low	237	33.3%	93.2%	5.7%	1.1%	5.7%	17.4
Uninitialized Variables	High	357	29.1%	63.6%	11.7%	14.3%	37.9%	3.4
Insecure Data Handling	Medium	477	25.2%	37.7%	9.5%	33.3%	9.4%	137.1
Control Flow Issues	Medium	1795	22.3%	33.8%	12.0%	50.0%	30.0%	15.9
Error Handling Issues	Medium	486	20.3%	42.1%	1.2%	28.6%	27.6%	17.7
Code Maintainability Issues	Low	154	18.8%	33.3%	2.1%	33.3%	27.3%	27.7
API Usage Errors	Medium	118	15.6%	38.7%	20.0%	33.3%	11.3%	8.5
Incorrect Expression	Medium	428	15.0%	43.4%	7.7%	26.6%	17.3%	6.8
Integer Handling Issues	Medium	400	13.6%	43.1%	25.9%	19.2%	28.0%	10.9
Performance Inefficiencies	Low	163	12.5%	98.4%	0.0%	0.0%	7.5%	2.6
Uninitialized Members	High	526	5.3%	26.6%	1.1%	17.7%	23.1%	26.9

using Chi-squared test. We also measure if there is a significant difference between the lifespan of security and non-security alerts for each project using Mann-Whitney U test.

3 FINDINGS

Taking the median from the detection counts of an alert type for each individual projects, we find that 8 alert types have a median count under 50 while the rest of the 16 alert types have a median count over 100. We report our findings for RQ1, RQ2, and RQ3 (Table 2) for the top 16 alert types considering they have the highest introduction rates. We answer RQ4 based on all the alerts from a project dividing them into two classes: security, and non-security.

Answer to RQ1: Table 2 lists the median count, triage rate, and fix rate for the top 16 alert types. We find that control flow issues generate the highest number of alerts. These alerts mostly point towards dead code and missing break in switch operations. However, low triage rate and the highest intentional rate suggest that developers may often find these alerts less important. We find that developers are most likely to triage alerts regarding memory related defects. These alerts mostly point towards "use after free" and "out-of-bounds access" type defects. Among the alerts that are triaged, we find that alerts from performance inefficiency (e.g. large stack use) and parse warnings have the highest fix rates. Furthermore, we find that defects from null pointer dereferences, uninitialized variables, and resource leak alert types are most likely to be explicitly marked as a bug when triaged.

Answer to RQ2: When triaging an alert, developers can mark the alert as false positive or intentional on Coverity Scan defect database. We find that alerts from illegal memory access (e.g. use after free) and integer handling issues (e.g. division by zero) are most likely to be marked as false positives, while control flow related alerts (e.g. missing break in switch) are most likely to be intentional coding by the developers. We also find that alerts regarding uninitialized members have the lowest triage and fix rate despite being marked as high impact by the tool.

Answer to RQ3: We find that alerts regarding performance inefficiency get eliminated in less than three weeks. We find that alerts from null pointer dereferences (e.g. dereference after null check) and uninitialized variables get eliminated within a month alongside being most likely to be marked as a bug. Conversely, we see alerts from insecure data handling have the longest lifespan (137 weeks) with a low triage rate.

Answer to RQ4: For Firefox, Samba, and Linux, we find that security alerts are significantly more likely to be triaged than non-security alerts while for Kodi, the difference is in the opposite direction ($p < .001$ for all). We find no such difference for Qt. While looking at the alerts that are triaged, we find that security alerts are significantly more likely to be eliminated than non-security alerts ($p < .001$) only for Firefox. For other projects, there is no significant difference. Similarly, for Firefox, we see that security alerts (111 days) have significantly shorter lifespan than non-security alerts (124 days), while for Linux, the difference (434 and 190.5 days) is in the opposite direction ($p < .001$ for both). For the other three projects, there is no significant difference between lifespan of security and non-security alerts.

From our findings, we make following observations:

- Alerts related to control flow issues appear most often, but they are also most likely to be marked as intentional coding by the developers.
- Developers are most likely to triage alerts that point towards memory related defects.
- Our cross-project comparison indicates that an alert being marked as a security issue by the tool does not affect its likelihood of getting fixed or its lifespan.

REFERENCES

[1] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 470–481.

[2] Philip J Guo and Dawson R Engler. 2009. Linux Kernel Developer Responses to Static Analysis Bug Reports.. In *USENIX Annual Technical Conference*. 285–292.
[3] Synopsys Mel Llaguno, Open Source Solution Manager. [n. d.]. 2017 Coverity Scan Report. Open Source SoftwareâĂŤThe Road Ahead. https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/SCAN-Report-2017.pdf.
[4] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. How open source projects use static code analysis tools in continuous integration pipelines. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 334–344.