A Dataset of Vulnerable Code Changes of the Chromium OS project

Rajshakhar Paul, Asif Kamal Turzo, Amiangshu Bosu
Department of Computer Science
Wayne State University
Detroit, Michigan, USA
{r.paul, asifkamal, amiangshu.bosu}@wayne.edu

***Abstract*—This paper presents a an empirically built and validated dataset of code reviews from the Chromium OS project that either identified or missed security vulnerabilities. The dataset includes total 890 vulnerable code changes categorized based on the CWE specification and is publicly available at: https://zenodo.org/record/4539891**

***Index Terms*—security, code review, dataset, vulnerability**

I. Introduction

Peer code review is an effective and well-recommended practice to identify security vulnerabilities during the pre-release stages [2]. However, despite practicing mandatory code reviews, many Open Source Software (OSS) projects are still encountering a large number of post-release security vulnerabilities, as some security defects remain undetected during code reviews. Therefore, a project manager may wonder if there was any weakness or inconsistency during a code review that missed a security vulnerability. Knowledge about the types of security defects that are escaping and the characteristics of code reviews that failed to identify security defects can help practitioners to improve the effectiveness of code reviews in identifying security defects.

On this context, we conducted a case-control study of the Chromium OS project to identify the factors which differentiate code reviews that successfully identified security defects from those that missed such defects [5]. We selected the cases and the controls based on our outcome of interest, namely whether a security defect was identified or escaped during the code review of a vulnerability contributing commit (VCC). Using a keyword-based mining approach followed by manual validations on a dataset of 404,878 Chromium OS code reviews, we identified 516 code reviews that successfully identified security defects. In addition, from the Chromium OS bug repository, we identified 239 security defects that escaped code reviews. Using a modified version of the SZZ algorithm followed by manual validations, we identified 374 VCCs and corresponding code reviews that approved those changes.

II. Research Method

As we have detailed our dataset creation steps with examples in another paper [5], this section provides a brief overview of that process. To build the dataset of vulnerabilities that were identified during code reviews, we adopted a keyword-based mining approach similar to Bosu *et* al. [2]. To build the dataset of vulnerabilities that were escaped during code reviews, we use the Monorail-based bug tracking system hosted at https://bugs.chromium.org/. We searched in the bug tracking system to identify a list of security defects for the Chromium OS project[1]. We used the following five-step approach to build this dataset.

(Step I) Custom search: We used a custom search (i.e., (`Type=Bug-Security status:Fixed OS=Chrome`), to filter security defects for the Chromium OS projects with the status as 'Fixed'. Our search result identified total 591 security defects. We exported the list of defects as a comma-separated values(i.e., CSV) file, where each issue is associated with a unique ID.

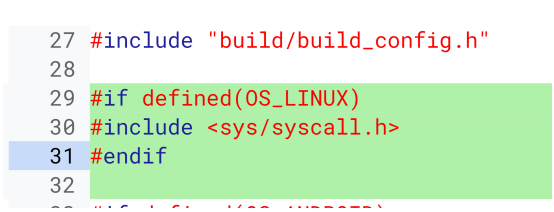
(Step II) Identifying vulnerability fixing commit: The Monorail page for each 'Fixed' issue includes detailed information (e.g., commit ID, owner, reviewer, review URL, and list of modified files) regarding its fix. We wrote a Python script to automate the extraction of the review URLs and commit IDs for each security defect identified in Step I. Finally, we excluded the security fixes that were not reviewed on Chromium OS's Gerrit repository (e.g., third-party libraries). At the end of this step, we were left with 239 security defects and its' corresponding fixes.

(Step III) Identifying vulnerability contributing commit(s): We adopted the modified version of the SZZ algorithm [1] to identify the vulnerability introducing commits from the vulnerability fixing commits identified in Step II. Our modified SZZ algorithm uses the `git blame` and `git bisect` subcommands and is adopted based on the approaches followed in two prior studies [4], [7] on VCCs. For each line in a given file, the `git blame` subcommand names the commit that last commit_id that modified it.

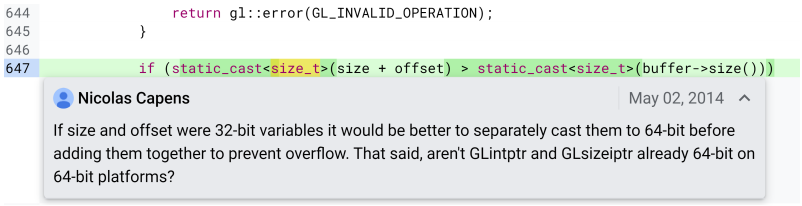
We manually inspect each of the VCCs identified by our modified SZZ algorithm as well as corresponding vulnerability fixing commits to exclude unrelated commits or include additional relevant commits. At the end of this step, we identified total 374 VCCs.

(Step IV) Identifying code reviews that approved VCCs: A git repository mirror for the Chromium OS project is hosted at https://chromium.googlesource.com/ with a gitiles[2] based

[1]https://bugs.chromium.org/p/chromium/issues/list
[2]https://gerrit.googlesource.com/gitiles/



(a) Improper access[a] (CWE-284)



(b) Integer overflow[a] (CWE-190)

[a]https://chromium-review.googlesource.com/c/chromium/src/+/781160/22/base/memory/shared_memory_posix.cc

[a]https://chromium-review.googlesource.com/c/angle/angle/+/197819/1/src/libGLESv2/libGLESv2.cpp

Fig. 1: Examples of vulnerable code changes from our dataset

frontend. We used the REST API of gitiles to query this repository to download commit logs for each VCC identified in the previous step. Using a REGEX parser, we extracted the URLs of the code review requests that approved VCCs identified in Step III. At the end of this step, we identified total 374 code reviews that approved our list of VCCs.

(Step V) CWE classification of the VCCs: 124 out of the 374 VCCs in our dataset had a CVE reported in the NIST NVD database[3]. For the remaining 250 VCCs, two of the authors independently inspected each VCC as well as its fixing commits to understand the coding mistake and classify it according to the CWE specification. Conflicting labels were resolved through discussions. The 890 VCCs (i.e., both identified and escaped cases) in our dataset represented 86 categories of CWEs. However, for the simplicity of our analysis, we decreased the number of distinct CWE categories by combining similar categories of CWEs into a higher level category using the hierarchical CWE specification[4].

For each of the 890 identified VCCs, we computed 25 different attributes that may influence the identification of a vulnerability during code reviews. Our dataset includes locations of these 890 VCCs as well as the 25 attributes for each VCC. Using these attributes, we developed a Logistic Regression model following the guidelines suggested by Harrell Jr [3]. The model, which achieved an AUC of 0.91, found nine code review metrics that distinguish code reviews that missed a vulnerability from the ones that did not. We have made this dataset and the R script to reproduce our results publicly available on Zenodo [6]: https://dx.doi.org/10.5281/zenodo.4539891.

III. Potential Future Usage

Our dataset includes 890 real world vulnerable code changes. Each of those vulnerabilities is classified using the CWE specification. We envision several ways researchers may benefit from this artefact.

1) Developers of static analysis tools may use this dataset to test the effectiveness of their solution in identifying various CWEs.
2) This dataset can be useful for training and evaluating automated machine learning models to identify security vulnerabilities.
3) This dataset may be also used to conduct further empirical studies to identify which other factors (e.g., code smells, or nano-patterns) may influence the identification of security defects during code reviews.
4) Our R script can be reused by researchers to develop Logistic Regression models following the guidelines of Harrell Jr.

URLs to access vulnerable files are located at the 'file_url' column of the `identified_vs_escaped.csv` file. For example, the 889th row of the dataset includes the link to a vulnerable file that includes an improper access (CWE) that escaped code review. Figure 1a shows the URL and the faulty code segment. Figure 1b shows an integer overflow that was identified during code reviews.

[3]https://nvd.nist.gov/

[4]https://cwe.mitre.org/data/graphs/1000.html

References

[1] M. Borg, O. Svensson, K. Berg, and D. Hansson, "Szz unleashed: an open implementation of the szz algorithm - featuring example usage in a study of just-in-time bug prediction for the jenkins project," *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation - MaLTeSQuE 2019*, 2019. [Online]. Available: http://dx.doi.org/10.1145/3340482.3342742

[2] A. Bosu, J. C. Carver, H. Munawar, P. Hilley, and D. Janni, "Identifying the characteristics of vulnerable code changes: an empirical study," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 257–268.

[3] F. E. Harrell Jr, *Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis*. Springer, 2015.

[4] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates, "When a patch goes bad: Exploring the properties of vulnerability-contributing commits," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 65–74.

[5] R. Paul, A. K. Turzo, and A. Bosu, "Why security defects go unnoticed during code reviews? a case-control study of the chromium os project," p. TBD, 2021.

[6] ——, "A dataset of Vulnerable Code Changes of the Chormium OS project," Feb. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4539891

[7] 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*, 2015, pp. 426–437.