VulinOSS: A Dataset of Security Vulnerabilities in Open-source Systems

Antonios Gkortzis, Dimitris Mitropoulos and Diomidis Spinellis
Department of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
{antoniosgkortzis,dimitro,dds}@aueb.gr

ABSTRACT

Examining the different characteristics of open-source software in relation to security vulnerabilities, can provide the research community with findings that can lead to the development of more secure systems. We present a dataset where the reported vulnerabilities of 8694 open-source project versions, can be correlated with the corresponding source code and a number of software metrics. The metrics were obtained by analyzing the project's source code via well-established tools. Apart from commonly used metrics (e.g. LOC), we also provide data related to modern development trends such as continuous integration and testing. We outline motivational examples based on the dataset we describe.

CCS CONCEPTS

• **Security and privacy → Software and application security**; • **Software and its engineering → Open source model**; **Software defect analysis**;

KEYWORDS

Security Vulnerabilities, Open-source Software, Continuous Integration, Testing

ACM Reference Format:
Antonios Gkortzis, Dimitris Mitropoulos and Diomidis Spinellis. 2018. VulinOSS: A Dataset of Security Vulnerabilities in Open-source Systems. In *MSR '18: MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3196398.3196454

1 INTRODUCTION

The examination of diverse software characteristics (e.g. code size) in relation to security vulnerabilities has been a constant topic of interest to the research community [2, 6, 8]. However, current software development trends, such as Continuous Integration (CI), haven't been studied from the software security perspective. Due to its nature, open-source software provides an opportunity for such a study.

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196454

We have created a dataset that correlates diverse *software metrics* derived from thousands open-source components with their (known) *security bugs*. To construct our dataset we examined the reports of the National Vulnerability Database[1] (NVD), to search for vulnerable open-source project versions. Notably, the NVD provides many details regarding the *severity* of a defect and the *impact* it may have on integrity, confidentiality and more. Then, we cloned the corresponding project repositories and analyzed the source of the vulnerable versions to retrieve specific metrics. Our analysis was based on well-established tools and techniques. Apart from a number of standard metrics (e.g. lines of code), we have collected elements related to *testing* and *continuous integration*. This provides a way to examine if such methods indeed lead to more secure software. We have made some initial measurements to demonstrate how researchers can use our dataset and produce interesting results.

The contributions of our work are a) the construction process of a dataset that correlates software metrics derived from 8694 open source components with their security vulnerabilities, b) the dataset and, c) how it can be used to produce research results.

2 DATASET CONSTRUCTION

Figure 1 illustrates the architecture of our dataset construction process. The construction was done in four steps. First we collected and processed (**1**) the various vulnerability reports from the National Vulnerability Database (NVD) to produce a set of open-source projects that contain software defects for further analysis. Then we cloned (**2**) the repositories of the selected projects. With the repositories at hand we mapped (**3**) version references (i.e. commit tags and branches) to the project versions retrieved during the first step. Based on the mapping, we checked-out (**4**) specific versions from the repository. Finally, by analyzing the code base of these versions, we retrieved code metrics and continuous integration traces. The results were then stored in a database, which is described in Section 3. In the following, we discuss each step in detail.

Refine The goal of this step was to examine all vulnerabilities contained in the NVD and store them along with the projects that they affect. First, we parsed the JSON feed files of the NVD, and retrieved the CVE[2] (Common Vulnerabilities and Exposures) entries, to capture reported defects. Notably, NVD enriches the information of a CVE entry with severity scores,[3] technical information regarding the vulnerability, its category (as defined by MITRE's Common Weaknesses and Enumeration[4] list), and the project it affects. If

[1] https://nvd.nist.gov/
[2] https://cve.mitre.org/
[3] https://nvd.nist.gov/vuln-metrics/cvss
[4] https://cwe.mitre.org/

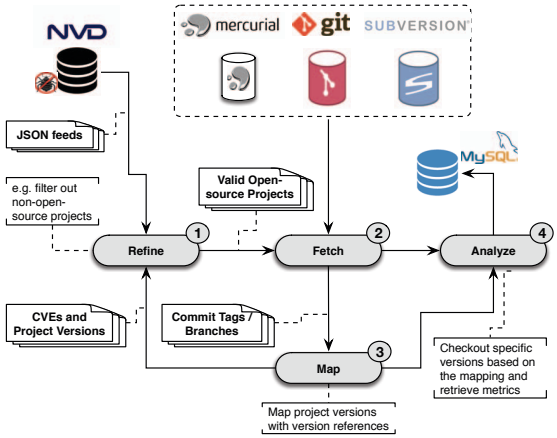

Figure 1: The dataset construction architecture.

multiple versions of a project are affected then the entry includes all these versions too. For every reported defect he have collected all the aforementioned details. CVEs marked as REJECT were ignored in order to avoid including false positives or duplicated entries in our dataset. Specifically, we have obtained 94010 CVEs for 7627 projects and their corresponding affected versions. We then ordered the list of 7627 project in descending order with respect to the number of vulnerabilities.

Then, we manually inspected the first six-hundred projects (those who had the most reported defects) and selected those that satisfy the following criteria: (1) they are open-source, and (2) there is a publicly available repository that hosts their code-base. One hundred and fifty three (153) projects satisfied these criteria. Note also that we excluded cases where it was not clear if the source code matched the project that was registered in the NVD. For each of these 153 cases we manually specified the following data: their repository URL, their web-site URL, and their software category. Our categorization was based on the FreeBSD port classification.[5]

A challenge during the first step was the existence of projects reported with different names due to simple misspellings (e.g. `nodejs:node.js` and `nodejs:nodejs`), the change of the software vendor (e.g. `rob_flynn:gaim` changed to `pidgin:pidgin`), or just variations of the names (e.g. `xorg:x11` and `x:x.org_x11`). In order to have a unique entry for each project we examined the history of each project. Then we scanned the list of the 7627 projects and found 102 cases that had more than one names. For these cases we created a list that maps the product vendor and the name variations to one entry.

Fetch With the aforementioned list at hand, we cloned locally the 153 project repositories. As Figure 1 indicates, the various repositories were based on three different source control management systems, namely: Git,[6] Mercurial,[7] and Subversion.[8]

Map In this step we associated the project versions (found in the first step) with the version references (commit tags and branches) found in the corresponding project repositories. To achieve this we queried each repository with the appropriate commands for each case (`git tag`, `hg tags`, `svn ls -v ^/tags` and, their branch listing commands respectively) and found that ninety-nine repositories had version references. For these projects we manually inspected the format of their version references by looking at each repository's tags. We then checked the NVD reported versions to find similarities between the two and link them. For example, an NVD entry involved *Xen* hypervisor's version `4.7.2`. A commit tag that involves this version however, has the following ID: `RELEASE-4.7.2`. Hence, for each project we created a set of regular expressions that could map such cases. For the aforementioned example, we created the following replacement regular expression (in Python): `re.sub(r'(RELEASE-)','',tag)`.

Analyze Having the repositories and the project versions at hand, we created Python workers that operated in the following manner: Initially, the worker checks out a specific version. Then it scans the source code to identify files that contain testing code. The scan is based on the heuristics of *reaper*, a well-established tool by Munaiah et al [7]. In its current state, the worker can identify testing code written in C, C++, C#, Java, Javascript, Objective-C, PHP, Python and Ruby. Finally, the worker invokes CLOC[9] (Count Lines Of Code) which in turn, runs on both the source code of the repository and the testing code that was retrieved by the worker. CLOC is commonly used in research to collect similar metrics [5]. The produced metrics are then stored in a MySQL database.

Apart from the results of CLOC, the worker also searches the project's directories for specific configuration files that indicate the use of CI technologies. If it finds such instances, it also records that the developers employ CI for this project along with the type of the CI technology that was used.

3 DATASET DESCRIPTION

Table 1, lists some descriptive statistics regarding our dataset including the number of projects it contains, project versions based on CI and more. All the collected data are stored in a relational database that includes nine tables. All tables and their relations can be seen in Figure 2. Below we provide details for each table.

CVE contains vulnerabilities, their description, and metrics that involve their severity. The severity metrics were based on the the second version of the Common Vulnerability Scoring System[10] (version 2 was selected due to its better backwards compatibility compared to version 3). The CVE table is related to the CWE table.

CWE stores the vulnerability classification list which consists of the 716 categories and their corresponding descriptions. Each entry of the previous table is related to one of the categories of this one.

Software Categories stores the list of software categories namely, *operating systems*, *end-user applications*, *system and administration utilities*, *programming languages & development frameworks*, *web and network utilities*, *science and engineering*. We created this list by grouping the 92 FreeBSD software port classification categories.

[5] https://www.freebsd.org/ports/categories-grouped.html
[6] https://git-scm.com/
[7] https://www.mercurial-scm.org/
[8] https://subversion.apache.org/
[9] https://github.com/AlDanial/cloc
[10] https://www.first.org/cvss/

Table 1: Descriptive statistics measurements for our dataset.

Measurement	Value
Projects	153
Project Versions	23884
Mapped Versions	8694
Number of Vulnerabilities	17738
Project Versions with Testing Code	38650
Project Versions employing CI	1538

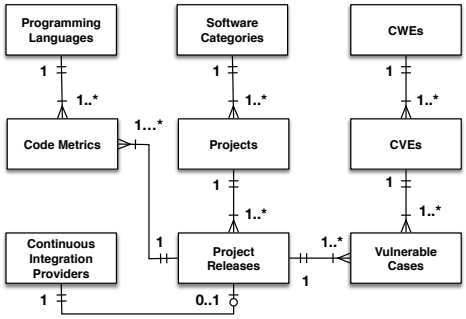

Figure 2: Vulinoss database schema.

This list can be customized and|or extended by researchers to fit their research needs.

Projects holds the data related to the 153 projects, selected during the first step of the collection process, and it is related to the Software Categories table. It also includes data about project vendors, websites, a URL that points to the project repository, the project's version control system type and a boolean field that shows if the repository has version references.

Project Releases entries represent a project version, as reported in the NVD JSON feeds. This table stores 23884 different project releases, and each release refers to a project found in the "Projects" table. In addition, we keep the version reference, (commit tag or branch). There are 8694 mapped versions, which account to the 36.4% of the total project releases. Finally, we store a reference to the "Continuous Integration Providers" table, which we describe later on.

Vulnerable Cases contains mappings between vulnerabilities and specific project versions.

Continuous Integration Providers stores a list of 9 continuous integration service providers, including *Travis*,[11] *AppVeyor*,[12] *Circle*,[13] and more.

Programming Languages contains records of a programming languages that the CLOC tool can identify as valid. Specifically, 219 programming languages are stored here including Java, C, C++, PHP, Python, and JavaScript.

Code Metrics contains the collected metrics for every mapped version of a project during the analysis step (**4**) of the dataset construction. If the project is written in more than one programming languages, there are metrics for each case. Specifically, for each case the table contains the number of files (containing code written in this language), the lines of code, the numbers of blank lines and finally, the number of comment lines. Additionally, if testing code exists the corresponding metrics are repeated that code.

4 APPLICATIONS

We provide a number of examples that illustrate how researchers can harness our database.

Vulnerability Density with respect to testing ratio and continuous integration. The vulnerability density is the number of vulnerabilities per unit of code size. In our first measurement, we set the code size as 1000 lines. We consider as testing ratio the lines of code related to testing divided by the total lines of code for a specific version.

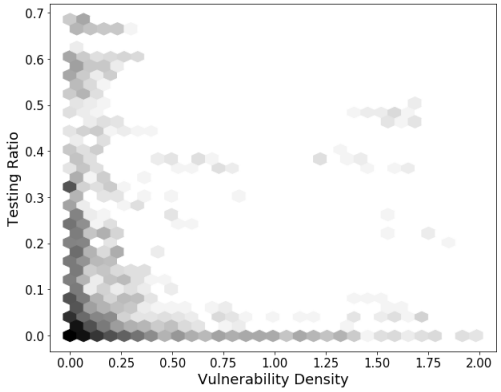

Figure 3: Vulnerability density with respect to testing ratio.

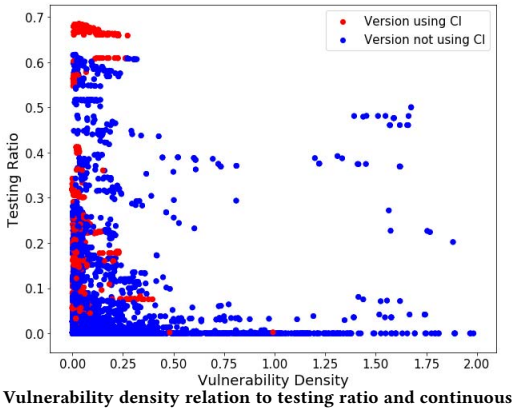

Figure 4: Vulnerability density relation to testing ratio and continuous integration.

First, we wanted to observe how the vulnerability density measure is related to the testing ratio. Figure 3 illustrates this relation. Darker polygons represent more project releases corresponding to that chart area. The Figure indicates that there are significantly more vulnerable project releases when the testing ratio is lower. This observation may indicate that projects accompanied by testing code can be more secure. We also attempted to correlate the two aforementioned measures with continuous integration. Figure 4 presents a similar plot but adds two colors to the various points of the graph. Red points represent project versions that make use of continuous integration while the blue points are versions that do not. The Figure indicates that CI has a positive effect on the code reducing the number of vulnerabilities that it contains.

Testing ratio and vulnerability severity. The severity of a vulnerability (low, medium, high) is an element that is of high importance in terms of security. We wanted to correlate the severity of the various defects of our dataset with the testing ratio of a project release. Figure 5 illustrates the average severity of the defects of a project version in relation to its testing ratio. A general observation is that the majority of the versions have a testing ratio lower than 40%. In addition, versions with a very low testing ratio appear to contain more severe vulnerabilities.

Vulnerability severity in non-bounds checking language. Programming languages such as C, C++ and assembly are more prone to security bugs because they lack a protection scheme

[11] https://travis-ci.org/
[12] https://www.appveyor.com/
[13] https://circleci.com/

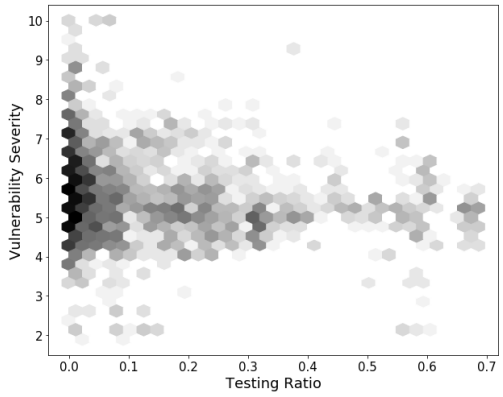

Figure 5: Vulnerability severity with respect to testing ratio.

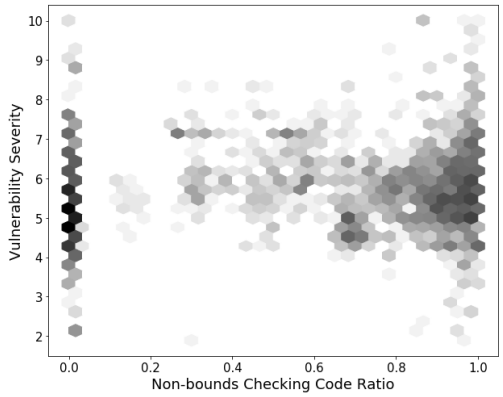

Figure 6: Vulnerability severity with respect to non-bounds checking code ratio.

against overwriting data in arbitrary parts of their memory space. For each project version written in such languages, we calculated the correlation of the corresponding code ratio, with the severity of this project's vulnerabilities. Figure 6 illustrates the results of this measurement. In the hexagonal binning, we observe that releases with a high ratio of non-bounds checking languages seem to have slightly more severe vulnerabilities, compared to other releases.

5 LIMITATIONS

Limitations concerning our dataset involve the completeness of some project repositories. In particular, there are old project versions that do not exist in the project's active repository. Hence, if the NVD has entries regarding these versions the mapping step will fail. A potential solution would be to retrieve legacy repositories, if they indeed exist. A threat to the internal validity of our dataset construction could be the limited analysis that our worker processes perform to identify test code. However, our scripts can be easily extended with additional modules that search for test code written in more languages.

6 RELATED WORK

Massacci et al. [4] examined the evolution of security vulnerabilities by examining six major versions of Firefox. To do so they created a database that contained information coming from the "Mozilla Firefox-related Security Advisories" (MFSA) list,[14] Bugzilla entries and more. Mitropoulos et. al [6] have presented a dataset with the software bugs (including security bugs) of the Maven repository. To produce this dataset, authors scanned all the JAR files of the repository with the Findbugs static analyzer.[15] On the CI front, Vasilescu et. al. [9] have analyzed the historical data of GitHub and reported that continuous integration has a positive effect on bug reporting when it comes to core-developers (up to 48%).

7 CONCLUSIONS

We have presented a dataset that contains vulnerable open-source project versions, details about their defects (e.g. severity), and a number of metrics related to their development process (e.g. if testing code exists). We have also shown how our data can be used to extract results about the relation between security bugs and software development techniques such as testing and continuous integration. Our dataset can be used to complement existing ones such as GHTorrent [3], RepoReaper [7], and TravisTorrent [1], to investigate the security aspects of a software artifact with respect to its software development team and the characteristics of its development process such as, the number of contributors and the number of issues. The dataset, together with the scripts are publicly available on a GitHub repository[16] under a Creative Commons License.

ACKNOWLEDGMENTS

The research presented in this paper is supported by the European Union within the H2020 MSCA-ITN-EID Project "SENECA".

REFERENCES

[1] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *Proceedings of the 14th working conference on mining software repositories*.

[2] Nigel Edwards and Liqun Chen. 2012. An historical examination of open source releases and their vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS '12)*. ACM, New York, NY, USA, 183–194.

[3] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. 233–236. /pub/ghtorrent-dataset-toolsuite.pdf Best data showcase paper award.

[4] Fabio Massacci, Stephan Neuhaus, and Viet Hung Nguyen. 2011. After-life vulnerabilities: a study on firefox evolution, its vulnerabilities, and fixes. In *Proceedings of the Third international conference on Engineering secure software and systems (ESSoS'11)*. Springer-Verlag, Berlin, Heidelberg, 195–208.

[5] Andrew Meneely, Alberto C. Rodriguez Tejeda, Brian Spates, Shannon Trudeau, Danielle Neuberger, Katherine Whitlock, Christopher Ketant, and Kayla Davis. 2014. An Empirical Investigation of Socio-technical Code Review Metrics and Security Vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering (SSE 2014)*. ACM, New York, NY, USA, 37–44.

[6] Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. 2014. The Bug Catalog of the Maven Ecosystem. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, New York, NY, USA, 372–375.

[7] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22, 6 (Dec. 2017), 3219–3253. https://doi.org/10.1007/s10664-017-9512-6

[8] Andy Ozment and Stuart E. Schechter. 2006. Milk or wine: does software security improve with age?. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15 (USENIX-SS'06)*. USENIX Association, Berkeley, CA, USA.

[9] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, 805–816.

[14] http://www.mozilla.org/projects/security/known-vulnerabilities.html

[15] http://findbugs.sourceforge.net/

[16] https://github.com/AUEB-BALab/VulinOSS