Vulnerability Discovery Strategies Used in Software Projects

Farzana Ahamed Bhuiyan
Akond Rahman
fbhuiyan42@tntech.edu
arahman@tntech.edu
Department of Computer Science
Cookeville, Tennessee, USA

Patrick Morrison
IBM
Durham, North Carolina, USA
pjmorris@us.ibm.com

ABSTRACT

Malicious users can exploit undiscovered software vulnerabilities i.e., undiscovered weaknesses in software, to cause serious consequences, such as large-scale data breaches. A systematic approach that synthesizes strategies used by security testers can aid practitioners to identify latent vulnerabilities. *The goal of this paper is to help practitioners identify software vulnerabilities by categorizing vulnerability discovery strategies using open source software bug reports.* We categorize vulnerability discovery strategies by applying qualitative analysis on 312 OSS bug reports. Next, we quantify the frequency and evolution of the identified strategies by analyzing 1,632 OSS bug reports collected from five software projects spanning across 2009 to 2019. The five software projects are Chrome, Eclipse, Mozilla, OpenStack, and PHP.

We identify four vulnerability discovery strategies: diagnostics, malicious payload construction, misconfiguration, and pernicious execution. For Eclipse and OpenStack, the most frequently used strategy is diagnostics, where security testers inspect source code and build/debug logs. For three web-related software projects namely, Chrome, Mozilla, and PHP, the most frequently occurring strategy is malicious payload construction i.e., creating malicious files, such as malicious certificates and malicious videos.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*.

KEYWORDS

bug report, empirical study, strategy, taxonomy, vulnerability

ACM Reference Format:
Farzana Ahamed Bhuiyan, Akond Rahman, and Patrick Morrison. 2020. Vulnerability Discovery Strategies Used in Software Projects. In *35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3417113.3422153

HCSE&CS-2020, September, 2020,

ACM ISBN 978-1-4503-8128-4/20/09…$15.00
https://doi.org/10.1145/3417113.3422153

1 INTRODUCTION

Discovering latent vulnerabilities is vital for secure software development. Vulnerability discovery is a human-centric activity, where the security tester applies security knowledge to identify vulnerabilities. Unfortunately, information technology (IT) organizations may not have the expertise of security testers, potentially prohibiting them to identify latent vulnerabilities in their software. A systematic synthesis of the vulnerability discovery strategies can help practitioners in identifying latent software vulnerabilities. Practitioners have acknowledged the importance of synthesizing knowledge related to vulnerability discovery and advocated for inculcating such knowledge into the software development process [14, 25]. On online forums, such as Reddit, we observe practitioners ask about strategies that security testers perform: "*In our team we are planning to pro-actively find vulnerabilities in our legacy software products. We don't have expert security testers in our team who have the knowledge to discover vulnerabilities. What strategies are typically used to find software vulnerabilities? Are there categories?*" [23]. Researchers [30, 42] also have advocated for synthesizing vulnerability discovery strategies, so that inexperienced practitioners are well-equipped to identify undiscovered vulnerabilities.

We hypothesize that by systematically analyzing OSS bug reports, we can derive a list of vulnerability discovery strategies used by security testers. OSS bug reports contain information on how bugs are discovered in OSS development [31]. Practitioners can learn from a synthesis of OSS bug reports that include information on how a vulnerability was discovered.

The goal of this paper is to help practitioners identify software vulnerabilities by categorizing vulnerability discovery strategies using open source software bug reports.

We answer the following research questions:

- ***RQ***$_1$ *[Categorization]*: What vulnerability discovery strategies are used by security testers in open source software?
- ***RQ***$_2$ *[Frequency]*: How frequently are the identified vulnerability discovery strategies used in open source software?

We derive vulnerability discovery strategies using open coding [43] on 312 OSS bug reports collected from the Mozilla organization. Next, we apply closed coding [12] to determine the frequency of the identified strategies in 1,632 bug reports collected from five OSS projects.

We list our contributions as follows:

- A categorization of vulnerability discovery strategies used in OSS; and
- An empirical study that quantifies the frequency and evolution of the identified vulnerability discovery strategies in OSS.

2 RELATED WORK

Researchers have used OSS bug reports to characterize and identify software vulnerabilities. Prediction model-based vulnerability detection research is prevalent. Shin and Williams [44] mined Mozilla bug reports to construct fault prediction models and reported that the models can be used to automatically predict vulnerable source code files. Wijayasekara et al. [49] mined text features from Linux bug reports to automatically detect vulnerabilities that appear in Linux. In another work, Zimmermann et al. [55] used bug reports from Windows Vista and constructed vulnerability prediction models (VPMs) with code metrics, such as churn, along with organizational hierarchy metrics. They [55] observed the median precision and recall to respectively be 0.6 and 0.4. Zhe et al. [54] mined Mozilla bug reports to construct vulnerability datasets, and applied active learning to improve vulnerability detection efficiency. Using active learning they identified 99% of the vulnerabilities by inspecting 34% of 28,750 source code files.

We notice a lack of research related to vulnerability discovery strategies as prior work has focused on vulnerability metrics, prediction, and resolution. We address this research gap by conducting a qualitative analysis of OSS bug reports to identify vulnerability discovery strategies.

3 RQ_1: CATEGORIZATION OF VULNERABILITY DISCOVERY STRATEGIES

In this section, we answer **RQ_1**:*What vulnerability discovery strategies are used by security testers in open source software?* First, we provide the methodology to answer RQ_1 in Section 3.1. Then, we provide our findings in Section 3.2.

3.1 Methodology to Answer RQ_1

Our methodology involves two steps: bug report collection and qualitative analysis. We describe these two steps as following:

3.1.1 Bug Report Collection. We answer RQ_1 by applying a qualitative analysis on the text content of OSS bug reports available from the Mozilla organization. We select Mozilla as the Mozilla organization has a wide range of OSS projects namely Firefox, SeaMonkey, Bugzilla, and Thunderbird [1]. We use the Mozilla Bugzilla API [34] to download necessary bug reports.

We apply filtering criteria to identify bug reports that describe what strategies security testers applied to discover the vulnerability:

- *Criterion-1:* We search for the keyword 'cve' in the title, description, and comments for each collected bug report. We use 'cve', as vulnerabilities in the National Vulnerability Database (NVD) are indexed using the 'CVE' identifier [2]. Practitioners mislabel security bugs [24, 27]. Label-based filtering of bug reports for bug/vulnerability identification can bias research results [26, 28], which motivated us to apply CVE-based filtering.
- *Criterion-2:* We manually examine if a bug report actually is related to a vulnerability indexed by NVD.
- *Criterion-3:* The bug report describes steps on how the vulnerability was discovered.

[1] http://kb.mozillazine.org/Summary_of_Mozilla_products
[2] https://nvd.nist.gov/vuln

The first author manually examined if all three criteria are satisfied for the collected bug reports. Altogether, we obtain 312 bug reports as of March 2019.

3.1.2 Qualitative Analysis. For qualitative analysis, we use open coding, where a rater observes and synthesizes patterns within unstructured text [43]. We apply open coding on the content of bug reports, which includes comments, attachments, and descriptions. We select open coding because we can obtain detailed information on what strategies security testers used to discover vulnerabilities by inspecting bug report content.

The first and second authors, individually, conduct an open coding process on the bug reports collected from Section 3.1.1. The first and second author respectively has three and eight years of experience in software security.

3.1.3 Rater Agreement. : We record the agreement rate between the first and second author for the identified vulnerability discovery strategies. We record Cohen's Kappa [11], and interpret Cohen's Kappa using Landis and Koch's interpretation [29].

3.2 Answer to RQ_1

First, we describe the vulnerability discovery strategies. *Second*, we provide details on rater agreement.

Our categorization includes four vulnerability discovery strategies. We describe each strategy with definitions, examples, and related MITRE ATT&CK techniques in the following subsections.

3.2.1 Diagnostics. The strategy of inspecting software source code or software logs to discover vulnerabilities. During the inspection process, the security testers apply their knowledge and inspect for patterns that are indicative of security weaknesses in the software. This strategy includes two sub-categories namely, diagnosis of software source code and diagnosis of software logs:

Software source code: We observe security testers to diagnose source code to discover vulnerabilities. The diagnosis of source code can be done manually or automatically by using static analysis tools. When diagnosing source code security testers search for security weaknesses, such as insecure coding patterns and incorrect implementations of cryptography algorithms.

Example: A security tester discovered a vulnerability (CVE-2015-2730) in Mozilla Firefox by inspecting source code [48]. The vulnerability was related to the computation of a cryptography algorithm called 'Elliptic Curve Digital Signature Algorithm (ECDSA)' [41]. When inspecting the implementation of ECDSA the security tester observed that the implementation does not handle corner cases, such as division by zero. The vulnerability can be exploited to generate invalid digital signatures when used for cryptography.

Software logs: Security testers perform manual examination of logs to discover a vulnerability. From our qualitative analysis, we observe security testers to collect logs from the build and debugging tools. In the logs, security testers inspect evidence of vulnerabilities, such as information leakage and uninitialized memory access.

Example: A security tester discovered a Mozilla Firefox vulnerability (CVE-2013-5595) [13] by inspecting logs generated from a debugging tool called 'Valgrind' [46]. The vulnerability is related to a buffer overflow. The source code of interest resides in the 'JavaScript Engine' of Mozilla Firefox and uses 'malloc', a C/C++ function used

to allocate memory. The implementation does not check for conditions that may cause buffer overflow [13]. Inspection of the Valgrind logs also helped the security tester reveal other source code files where buffer overflow conditions are not checked:"*I found a few other places that were performing unchecked multiplications to compute buffer sizes*" [13].

3.2.2 Malicious Payload Construction. The strategy of constructing a malicious software artifact to discover a vulnerability. For this strategy, security testers construct the payload with code snippets or binary files. Upon construction, the payload is passed as input to the software, which leads to an undesired consequence, such as software crashes or leakage of data. We observe two sub-categories for this strategy namely, constructing malicious code snippets and constructing malicious binary files:

Code snippets: Security testers construct payload using code snippets with programming languages, such as JavaScript, HTML, and Python. Code snippets can be created manually by the security testers themselves where they apply their knowledge to construct source code snippets. Code snippets can also be created using automated tools, such as fuzzing tools. Fuzzing is used to generate erroneous input to software so that the software can be monitored for exceptions [1].

Example: A security tester constructed a payload manually using the 'marquee' element in HTML to discover a Mozilla Firefox vulnerability (CVE-2016-5262) [37]. The marquee element is used to create a scrolling text or an image [47]. The vulnerability occurred due to the incorrect event handling for the marquee element and could have been exploited to conduct cross-site scripting attacks [16].

As an example of fuzzing tool usage, a practitioner discovered a vulnerability in Thunderbird (CVE-2016-5824) [6] that can allow a malicious user to cause a denial of service attack [17].

Binary files: Security testers construct binary files to discover vulnerabilities in the software. We observe four sub-categories of malicious binary construction: (i) certificates i.e. data files that digitally bind a cryptography key; (ii) documents, such as PDF files; (iii) multimedia files, such as image and video files, and (iv) executables, such as .exe files.

Example: We provide examples of vulnerabilities for the four sub-categories below:
Certificate: A digital certificate holds information that confirms the identity of an information technology (IT) organization on the Internet [5]. Each digital certificate contains a field called an object identifier (OID). OID is a numeric value that identifies the IT organization for which the certificate is being used. A security tester constructed a digital certificate file to discover a Mozilla Firefox vulnerability (CVE-2017-7792) [45]. The security tester created a certificate with an OID with 399 digits [3]. The vulnerability occurred for not checking buffer size when processing certificates with long OIDs [45] and could have been exploited to cause a crash [19].
Document: A security tester created a malicious PDF file to discover a vulnerability (CVE-2018-5158) [20, 52]. The constructed malicious PDF file contained array of strings. The PDF Viewer implementation of Mozilla Firefox did not check if strings are included in a PDF file, which allows malicious users to use strings as a method to inject JavaScript snippets [52].
Multimedia: A malicious MP4 file was used to discover a vulnerability in Mozilla Firefox (CVE-2015-0829) [39]. The malicious MP4 file revealed that an adequate amount of buffer was not allocated when writing MP4 files [39]. The vulnerability could have been exploited to execute arbitrary code [15].
Executable: A malicious executable was used to discover a vulnerability that affects Mozilla Firefox for the Windows operating system (CVE-2017-7761) [18, 51]. The security tester created an executable that creates a directory in 'C:/Windows/Temp'. The vulnerability could have been used to delete arbitrary files.

3.2.3 Misconfiguration. The strategy of identifying and altering one or multiple configurations of the software to discover vulnerabilities. As part of this strategy security testers first identify the software's configurations and their corresponding values. Then security testers alter configuration values.

Example: A security tester discovered a vulnerability in Mozilla Firefox (CVE-2015-0832) [33] by changing values for a configuration 'security.cert_pinning.enforcement_level' [36]. The configuration is related to public key pinning, a mechanism that allows a website to explicitly specify certificate authorities [7].

The configuration has four possible values: 0, 1, 2, and 3. The default value is 1, which indicates that public key pinning is enabled but not enforced [50]. To discover the vulnerability, public key pinning was enforced by assigning a value of 2. The vulnerability could have been used to conduct man-in-the-middle attacks [33].

3.2.4 Pernicious Execution. The strategy of identifying a sequence of software interactions that expose a vulnerability. While an end-user interacts with a software to accomplish a certain task, a security tester interacts with the software to discover vulnerabilities.

The difference between pernicious execution to the other strategies are: (i) unlike diagnostics, the practitioner does not inspect source code and build/debug logs; (ii) unlike malicious payload construction, the practitioner does not create malicious payload and pass it as input to the software project of interest; and (iii) unlike misconfiguration, the software's configurations are not altered.

Example: A security tester discovered a vulnerability (CVE-2014-1499) in Mozilla Firefox by identifying a sequence of interactions that involve visiting websites [22, 32]. According to the bug report [22], the tester *first* visited a website called 'Hacker News' [4], a modern news aggregator website [2], by opening a tab in the browser. *Second*, the tester visited a website [5] that uses WebRTC in the same tab of the browser. WebRTC is a technology that enables websites to capture audio/video media from a web camera [35]. *Finally*, when the tester pressed the back button of the Firefox browser to go back to the Hacker News website, a prompt appeared.

The vulnerability occurs due to a race condition, which a malicious user can exploit by convincing the user that a harmless website is trying to access the user's web camera [22]. The vulnerability is an example of how sequences of interactions can be identified to discover vulnerabilities in software.

3.2.5 Rater Agreement. : The first and second authors respectively, determines four and six vulnerability discovery strategies. The open coding process took 107 and 114 hours respectively, for the first

[3]https://bugzilla.mozilla.org/attachment.cgi?id=8872576

[4]https://news.ycombinator.com/

[5]http://hughsk.github.io/post-process/

and second authors. No new strategies generated after the analysis of 215 and 221 bug reports respectively, for the first and second authors. The Cohen's Kappa is 0.58, which is 'moderate' agreement according to the interpretation of Landis and Koch [29].

4 RQ_2: FREQUENCY OF VULNERABILITY DISCOVERY STRATEGIES

In this section, we answer: ***RQ_2**: How frequently are the identified vulnerability discovery strategies used in open source software?* Our identified strategies are derived from one OSS organization. Answer to RQ2 can provide evidence if our identified strategies are generalizable across other types of software projects, such as IDEs.

4.1 Methodology to Answer RQ_2

We answer RQ_2 using three steps: dataset construction, applying closed coding [12], and conducting frequency analysis, to determine the frequency of strategies in the datasets.

4.1.1 Dataset construction. We use five datasets to answer RQ_2. We construct these datasets from OSS projects that (i) have made their bug reports available, and (ii) have bug reports that include information of CVEs indexed by the NVD. We briefly describe each dataset below:
Chrome: The Chrome dataset consists of bug reports for Google Chrome, a web browser [10].
Eclipse: The Eclipse dataset consists of bug reports for the Eclipse IDE [21].
Mozilla: The Mozilla dataset includes bug reports for the following applications: Bugzilla, Firefox, SeaMonkey and Thunderbird.
OpenStack: The OpenStack dataset includes bug reports from OpenStack software i.e. software developed and maintained by OpenStack, which is used to create computational, storage, and networking resources [38].
PHP: The PHP dataset includes bug reports for PHP, a programming language used in web development [40].

After collecting the bug reports for all the five projects, a rater who is not involved in the open coding process manually examined if the three criteria mentioned in Section 3.1.1 is satisfied for each of the collected bug reports.

4.1.2 Closed Coding to Determine Frequency of Strategies. Closed coding is the process of mapping an entry to a pre-defined category [12]. For RQ_2, we use a rater who is not involved in deriving the vulnerability discovery strategies to mitigate bias in the results. The rater involved in closed coding first determines if a bug report includes information related to a vulnerability discovery. Next, if the bug reports include vulnerability discovery information, then the rater determines if the bug report content can be mapped to any of the four identified vulnerability discovery strategies listed in Section 3.2. A vulnerability can be discovered using one or more strategies, and the rater can map a bug report to one or multiple strategies. Before performing the examination, the rater is provided a reference document, which has the name, definition, and examples of each strategy. The reference document is included in our verifiability package [4]. Upon completion of closed coding, we obtain a mapping between each bug report and one or multiple strategies.

A graduate student in the university conducts the closed coding process. The rater is an M.Sc. student in Computer Science, with four years of professional software development experience, and two years of security testing experience. The student has also completed three cybersecurity-related courses offered at the graduate level. The rater was not involved in the open coding process and is not an author of the paper. The rater manually examined each bug report and determined which of the identified strategies is applicable for each of the bug reports.

4.1.3 Frequency analysis. We answer RQ_2 using two metrics. First, using the 'PropVuln(x)' metric we quantify the proportion of vulnerabilities that are identified using each strategy. Second, using the 'Strategy/Year' metric we quantify the temporal frequency for each identified strategy. We use Equations 1 and 2 respectively, to calculate 'PropVuln' and 'Strategy/Year'. The 'PropVuln' metric provides a summary of how frequently the identified strategies are reported in bug reports. The 'Strategy/Year' metric shows how the use of each strategy has evolved over time.

$$\text{PropVuln}(x) = \frac{\text{number of vulnerabilities discovered with strategy } x}{\text{number of vulnerabilities}} \quad (1)$$

$$\text{Strategy/Year}(x, y) = \frac{\text{number of vulnerabilities discovered with strategy } x \text{ in year } y}{\text{number of vulnerabilities reported in year } y} \quad (2)$$

4.2 Answer to RQ_2

In Table 1 we provide a breakdown of how many bug reports we filtered using the criteria mentioned in Section 3.1.1. As shown in Table 1, we identify 943, 36, 410, 119, and 124 bug reports respectively, for the Chrome, Eclipse, Mozilla, OpenStack, and PHP datasets. We report attributes of the five datasets in Table 2. From Table 2, we observe except for Eclipse and OpenStack all datasets are related to web-based software projects. The set of 410 Mozilla bug reports includes the 312 bug reports used in Section 3.1.2.

Results: We answer RQ_2 by first reporting the summary of the five datasets and then reporting the 'PropVuln' and 'Strategy/Year' values. We report the 'PropVuln' values for the four strategies in Table 3. The columns 'Diagnostics', 'Execution', 'Misconfig.', and 'Payload' respectively, presents the four strategies namely, diagnostics, pernicious execution, misconfiguration, and malicious payload construction. Except for Eclipse and OpenStack, we observe malicious payload construction to be the most frequent vulnerability discovery strategy. For Eclipse and OpenStack, the most frequently occurring strategy is diagnostics. For Chrome, Firefox, and PHP, 83% or more of the total vulnerabilities are discovered using malicious payload construction. For Eclipse and OpenStack respectively, 62.9% and 52.1% of the vulnerabilities are discovered using diagnostics. We also observe one strategy not to be comprehensive in discovering all reported vulnerabilities in the five software projects, which is congruent with prior research [3].

The strategies malicious payload construction and diagnostics have sub-categories. We present the 'PropVuln' values for each sub-category for malicious payload construction and diagnostics

Table 1: Bug Reports Satisfying Filtering Criteria

Criteria	Chrome	Eclipse	Mozilla	OpenStack	PHP
Initial	62,777	10,425	52,154	18,290	13,505
Criterion-1	3,241	147	427	368	182
Criterion-2	960	37	410	120	125
Criterion-3	943	36	410	119	124
Total	943	36	410	119	124

Table 2: Attributes of Bug Reports

Dataset	Type	Bug Reports	CVEs	Duration
Chrome	Web browser	943	943	02/2010-06/2019
Eclipse	IDE	36	36	03/2017-06/2019
Mozilla	Web apps	410	410	07/2011-06/2019
OpenStack	Computing	119	119	07/2012-06/2019
PHP	Web progr.	124	124	09/2009-06/2019
All		1,632	1,632	09/2009-06/2019

Table 3: Proportion of Vulnerabilities Discovered by Each Strategy. Highlighted Cells in Green Indicate the Most Frequently Occurring Strategy for a Dataset.

Dataset	Diagnostics	Execution	Misconfig.	Payload
Chrome	10.0%	2.7%	0.8%	86.5%
Eclipse	62.9%	5.7%	8.5%	25.7%
Mozilla	13.4%	6.8%	2.7%	83.2%
OpenStack	52.1%	4.2%	29.4%	16.8%
PHP	15.3%	1.6%	17.7%	89.5%

Table 4: Proportion of Sub-categories for Malicious Payload Construction and Diagnostics. Highlighted Cells in Green Indicate the Most Frequently Occurring Sub-category.

Dataset	Diagnostics		Payload	
	Binary	Snippet	Code	Build/Debug
Chrome	12.0%	88.0%	18.5%	81.5%
Eclipse	6.7%	93.3%	5.9%	94.1%
Mozilla	7.3%	92.7%	26.7%	73.3%
OpenStack	21.1%	78.9%	25.0%	75.0%
PHP	30.0%	70.0%	11.1%	88.9%

in Table 4. We observe 'Snippet' i.e., payload construction with code snippet, to be the most frequently occurring sub-category for malicious payload construction, whereas, 'Build/Debug' i.e., performing diagnostics through the build and debugging logs, is the most frequently occurring sub-category for diagnostics.

5 DISCUSSION

In this section we discuss implications:

Implications for Practitioners. In Section 1 we mentioned a Reddit post [23], where the forum participants suggested the use of static analysis and fuzzing. Our categorization validates the provided suggestions in the Reddit post [23]: we observe code inspection with static analysis tools, and payload construction using fuzzing tools are strategies that security testers use to discover vulnerabilities. Furthermore, we observe security testers to not only inspect source code but also inspect build and debug logs to discover vulnerabilities.

Our categorization also includes pernicious execution and misconfiguration. Practitioners can use our dataset to learn from vulnerabilities that were discovered using pernicious execution. For misconfiguration, practitioners would be exploring available configurations of a software. We acknowledge exploring configurations can be non-trivial, as software projects can have as many as 412 configurations [53]. Practitioners may find Xu et al. [53]'s findings helpful: Xu et al. [53] proposed natural language processing techniques to explore configurations efficiently.

Implications for Researchers. For future work, researchers can investigate if security testers prioritize any strategies to discover vulnerabilities. Closed coding used in Section 4.1 requires manual effort and might not scale for a large number of datasets. Researchers can build upon our paper, and use methodologies proposed by prior research [8, 9] to automatically identify vulnerability discovery strategies from bug reports.

Limitations: Our analysis only includes vulnerabilities that are indexed by the NVD, which could be limiting. Our categorization is susceptible to conclusion validity as answers to RQ_1 and RQ_2 might be influenced by rater bias. Our empirical study is limited to the datasets that we analyzed. Our identified strategies are not comprehensive. Investigating bug reports from other types of software from the OSS domain and the proprietary domain might reveal strategies not reported in our paper. We mitigate this limitation by using OSS bug reports from a variety of sources including web-based programming languages, computing services, and IDEs.

6 CONCLUSION

A categorization of software vulnerability discovery strategies can help practitioners identify undiscovered vulnerabilities. We construct a categorization by applying qualitative analysis on 312 bug reports collected from the Mozilla organization. Next, we quantify the frequency of the identified strategies with 1,632 bug reports.

We identify four strategies namely, diagnostics, malicious payload construction, misconfiguration, and pernicious execution. Malicious payload construction is the most frequently occurring strategy for web-related software projects: Chrome, Mozilla, and PHP. For Eclipse and OpenStack, diagnostics is the most frequent strategy.

Our findings have implications for practitioners and researchers. Practitioners can use our categorization to discover new vulnerabilities before deployment of software to end-users. Our categorization can also be used as educational materials in cybersecurity-related courses. Our paper lays the groundwork for future research related to automated characterization of strategy prevalence.

ACKNOWLEDGMENTS

We thank the PASER group at Tennessee Tech. University for their valuable feedback. The research was partially funded by the Cybersecurity Education, Research and Outreach Center (CEROC) at Tennessee Tech. University.

REFERENCES

[1] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing.* Cambridge University Press.
[2] Mauricio Aniche, Christoph Treude, Igor Steinmacher, Igor Wiese, Gustavo Pinto, Margaret-Anne Storey, and Marco Aurelio Gerosa. 2018. How Modern News

Aggregators Help Development Communities Shape and Share Knowledge. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 499–510. https://doi.org/10.1145/3180155.3180180

[3] A. Austin and L. Williams. 2011. One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques. In *2011 International Symposium on Empirical Software Engineering and Measurement*. 97–106. https://doi.org/10.1109/ESEM.2011.18

[4] Farzana Ahamed Bhuiyan, Akond Rahman, and Patrick Morrison. 2020. Verifiability Package for Paper. https://figshare.com/s/eaac5aeea283239f2c56. [Online; accessed 05-Sep-2020].

[5] Matthew A. Bishop. 2002. *The Art and Science of Computer Security*. Addison-Wesley Longman Publishing Co., Inc., USA.

[6] Brandon Perry. 2016. Bug 1275400 (CVE-2016-5824). https://bugzilla.mozilla.org/show_bug.cgi?id=1275400. [Online; accessed 16-May-2020].

[7] C. Evans and C. Palmer. 2011. Public Key Pinning Extension for HTTP. https://tools.ietf.org/html/draft-ietf-websec-key-pinning-01. [Online; accessed 15-Feb-2020].

[8] Oscar Chaparro, Carlos Bernal-Cardenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. 2019. Assessing the Quality of the Steps to Reproduce in Bug Reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 86–96. https://doi.org/10.1145/3338906.3338947

[9] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting Missing Information in Bug Descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 396–407. https://doi.org/10.1145/3106237.3106285

[10] Chrome. 2020. Chrome infrastructure. https://chromium.googlesource.com/infra/infra/+/master/appengine/monorail/doc/api.md. [Online; accessed 26-Jan-2020].

[11] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. https://doi.org/10.1177/001316446002000104

[12] Benjamin F Crabtree and William L Miller. 1999. *Doing qualitative research*. sage publications.

[13] Dan Gohman. 2013. Bug 916580 (CVE-2013-5595). https://bugzilla.mozilla.org/show_bug.cgi?id=916580. [Online; accessed 09-Feb-2020].

[14] Danelle Au. 2015. The Importance of Learning From Hackers. https://www.securityweek.com/importance-learning-hackers. [Online; accessed 09-Feb-2020].

[15] National Vulnerability Database. 2015. NVD-CVE-2015-0829. https://nvd.nist.gov/vuln/detail/CVE-2015-0829. [Online; accessed 19-Feb-2020].

[16] National Vulnerability Database. 2016. NVD-CVE-2016-5262. https://nvd.nist.gov/vuln/detail/CVE-2016-5262. [Online; accessed 27-Feb-2020].

[17] National Vulnerability Database. 2016. NVD-CVE-2016-5824. https://nvd.nist.gov/vuln/detail/CVE-2016-5824. [Online; accessed 19-May-2020].

[18] National Vulnerability Database. 2017. NVD-CVE-2017-7761. https://nvd.nist.gov/vuln/detail/CVE-2017-7761. [Online; accessed 17-Feb-2020].

[19] National Vulnerability Database. 2017. NVD-CVE-2017-7792. https://nvd.nist.gov/vuln/detail/CVE-2017-7792. [Online; accessed 18-Jan-2020].

[20] National Vulnerability Database. 2018. NVD-CVE-2018-5158. https://nvd.nist.gov/vuln/detail/CVE-2018-5158. [Online; accessed 30-Jan-2020].

[21] Eclipse. 2020. The Platform for Open Innovation and Collaboration. https://www.eclipse.org/. [Online; accessed 27-Jan-2020].

[22] Ehsan Akhgari. 2014. Bug 961512 (CVE-2014-1499). https://bugzilla.mozilla.org/show_bug.cgi?id=961512. [Online; accessed 01-Feb-2020].

[23] eusian. 2020. Strategies to find software vulnerabilities: what are the categories? https://www.reddit.com/r/cybersecurity/comments/egflzj/strategies_to_find_software_vulnerabilities_what/. [Online; accessed 23-Jan-2020].

[24] M. Gegick, P. Rotella, and T. Xie. 2010. Identifying security bug reports via text mining: An industrial case study. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 11–20.

[25] Giovanni Vigna. 2019. How to Think Like a Hacker. https://www.darkreading.com/vulnerabilities---threats/how-to-think-like-a-hacker/a/d-id/1335989. [Online; accessed 10-Feb-2020].

[26] K. Herzig, S. Just, and A. Zeller. 2013. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *2013 35th International Conference on Software Engineering (ICSE)*. 392–401.

[27] Yuan Jiang, Pengcheng Lu, Xiaohong Su, and Tiantian Wang. 2020. LTRWES: A new framework for security bug report detection. *Information and Software Technology* 124 (2020), 106314. https://doi.org/10.1016/j.infsof.2020.106314

[28] Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. 2019. The Importance of Accounting for Real-World Labelling When Predicting Software Vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 695–705. https://doi.org/10.1145/3338906.3338941

[29] J. Richard Landis and Gary G. Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *Biometrics* 33, 1 (1977), 159–174. http://www.jstor.org/stable/2529310

[30] Gary McGraw. 2018. *Software Security: Building Security In*. Addison-Wesley Professional.

[31] L. Moreno, W. Bandara, S. Haiduc, and A. Marcus. 2013. On the Relationship between the Vocabulary of Bug Reports and Source Code. In *2013 IEEE International Conference on Software Maintenance*. 452–455. https://doi.org/10.1109/ICSM.2013.70

[32] Mozilla. 2014. Mozilla Foundation Security Advisory 2014-19. https://www.mozilla.org/en-US/security/advisories/mfsa2014-19/. [Online; accessed 11-Jan-2020].

[33] Mozilla. 2015. Mozilla Foundation Security Advisory 2015-13. https://www.mozilla.org/en-US/security/advisories/mfsa2015-13/. [Online; accessed 17-Jan-2020].

[34] Mozilla. 2020. Bugzilla Main Page. https://bugzilla.mozilla.org/home. [Online; accessed 03-Feb-2020].

[35] Mozilla Developer Network. 2020. WebRTC API. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API. [Online; accessed 16-Feb-2020].

[36] Muneaki Nishimura. 2014. Bug 1065909 (CVE-2015-0832). https://bugzilla.mozilla.org/show_bug.cgi?id=1065909. [Online; accessed 22-Feb-2020].

[37] Nikita. 2016. Bug 1277475 (CVE-2016-5262). https://bugzilla.mozilla.org/show_bug.cgi?id=1277475. [Online; accessed 13-Feb-2020].

[38] Openstack. 2020. Openstack - Build the future of Open Infrastructure. https://www.openstack.org/. [Online; accessed 21-Feb-2020].

[39] pantrombka. 2015. Bug 1128939 (CVE-2015-0829). https://bugzilla.mozilla.org/show_bug.cgi?id=1128939. [Online; accessed 10-Feb-2020].

[40] PHP. 2020. PHP Bug Tracking System. https://bugs.php.net/. [Online; accessed 28-Jan-2020].

[41] T. Pornin. 2013. Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). https://tools.ietf.org/html/rfc6979. [Online; accessed 28-Jan-2020].

[42] Akond Rahman and Laurie Williams. 2019. A Bird's Eye View of Knowledge Needs Related to Penetration Testing. In *Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security* (Nashville, Tennessee, USA) *(HotSoS '19)*. Association for Computing Machinery, New York, NY, USA, Article 9, 2 pages. https://doi.org/10.1145/3314058.3317294

[43] Johnny Saldana. 2015. *The coding manual for qualitative researchers*. Sage.

[44] Yonghee Shin and Laurie Williams. 2013. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering* 18, 1 (01 Feb 2013), 25–59. https://doi.org/10.1007/s10664-011-9190-8

[45] Tyson Smith. 2017. Bug 1368652 (CVE-2017-7792). https://bugzilla.mozilla.org/show_bug.cgi?id=1368652. [Online; accessed 14-Feb-2020].

[46] Valgrind. 2019. Valgrind Home. http://www.valgrind.org/. [Online; accessed 22-Feb-2020].

[47] w3docs. 2020. HTML <marquee> Tag. https://www.w3docs.com/learn-html/html-marquee-tag.html. [Online; accessed 13-Feb-2020].

[48] watsonbladd. 2015. Bug 1125025 (CVE-2015-2730). https://bugzilla.mozilla.org/show_bug.cgi?id=1125025. [Online; accessed 10-Feb-2020].

[49] D. Wijayasekara, M. Manic, and M. McQueen. 2014. Vulnerability identification and classification via text mining bug databases. In *IECON 2014 - 40th Annual Conference of the IEEE Industrial Electronics Society*. 3612–3618. https://doi.org/10.1109/IECON.2014.7049035

[50] Mozilla Wiki. 2019. SecurityEngineering/Public Key Pinning. https://wiki.mozilla.org/SecurityEngineering/Public_Key_Pinning. [Online; accessed 29-Jan-2020].

[51] Wladimir Palant. 2017. Bug 1215648 (CVE-2017-7761). https://bugzilla.mozilla.org/show_bug.cgi?id=1215648. [Online; accessed 10-Feb-2020].

[52] Wladimir Palant. 2018. Bug 1452075 (CVE-2018-5158). https://bugzilla.mozilla.org/show_bug.cgi?id=1452075. [Online; accessed 10-Feb-2020].

[53] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, You Have given Me Too Many Knobs!: Understanding and Dealing with over-Designed Configuration in System Software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 307–319. https://doi.org/10.1145/2786805.2786852

[54] Z. Yu, C. Theisen, L. Williams, and T. Menzies. 2019. Improving Vulnerability Inspection Efficiency Using Active Learning. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2949275

[55] T. Zimmermann, N. Nagappan, and L. Williams. 2010. Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista. In *2010 Third International Conference on Software Testing, Verification and Validation*. 421–428. https://doi.org/10.1109/ICST.2010.32