Finding Web Application Vulnerabilities with an Ensemble Fuzzing

João Caseirito, Ibéria Medeiros
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
jcaseirito@lasige.di.fc.ul.pt, imedeiros@di.fc.ul.pt

I. Introduction

Cyberattacks have been a constant on the Internet and their impact and cost have risen to billions of dollars. They are usually associated with the exploitation of vulnerabilities in web applications, the most common form of accessing services and organizations' data. These applications support a myriad of services for handling multiples daily needs of our lives (e.g., social media, e-commerce, bank account access). There are currently almost 2 billion of websites active [1] and the number of vulnerabilities disclosed and associated with applications they handle continues growing [2]. Cross Site Scripting (XSS) and SQL injection (SQLi) are the two most prevalent web vulnerabilities and they pose the first place in OWASP Top 10 [3]. Although these vulnerability classes are really well known, they continue appearing in recent applications used by millions of consumers [4].

Many techniques have been used for systematically detecting vulnerabilities, where fuzzing is the most precise and capable of exploiting them [5][6][7]. However, it does not identify them on the code, placing this task effort on the programmers' side. Contrarily, code inspection tools can localize vulnerabilities for programmers [8][9][10][11], but their results can be imprecise or (even when correct) unexploitable [12], requiring programmers to separate the true from false positives. Hence, empowering programmers with tools for the systematic and accurate identification of vulnerabilities is extremely beneficial for them.

This work introduces an *Ensemble Fuzzing* (`EF`) approach to identity real vulnerabilities existing in web applications written in PHP. The goal is to provide a combination of black-box fuzzers that use fuzzing to discover vulnerabilities by doing input injection and to identify the PHP code of the exploited vulnerabilities, i.e., the real vulnerabilities, by extracting the executed code traces that are exercised by the injected inputs.

II. Challenges

Doupé et al. [13] defined as being the biggest challenge of black-box fuzzing tools the determination of interactions that can change the state of the application, and thus, for getting and discovering the maximum of code coverage and vulnerabilities, respectively. For instance, using the same requests with distinct inputs or sending them in a different order can result in different executed paths. Hence, without considering this, the code coverage and vulnerabilities could be missed.

In addition to this challenge, we can derive two others for web vulnerability fuzzers. These fuzzers comprise two components – crawler and scanner[1]. The former inspects the attack surface of the web application under test to extract the URLs with their entry points (web requests), whereas the latter performs attacks over these URLs. However, there is no certainty that the crawler extracts all valid URLs the application contains, i.e., if it is capable of extracting all entry points from the attack surface, and composing valid URLs with them. Contrarily, for the scanner, there is no guarantee that it is capable of exploiting the possible vulnerabilities contained in the application, i.e., if it is able to exercise the URLs with the correct injected code capable of exploiting the vulnerabilities.

Finally, the last challenge is how to extract traces of the code executed when the scanner carries out the attacks, without accessing and inspecting the source code of the application. In addition, given all traces, how to identify which ones are associated with the exploited vulnerabilities.

III. The Ensemble Fuzzing Approach

The `EF` approach aims to improve the findings of web vulnerabilities, identifying their code in the application, and increase the code coverage by exploring the same set of URLs with different fuzzers. We propose an approach that combines different fuzzers and resorts from the best they have in order to improve the resolution of the challenges presented in Section II, by increasing the probability of exploiting a vulnerability by using several fuzzers that will exercise the same requests.

The approach encompasses three phases – *Crawling*, *Attacking* and *Vulnerability Identification*.

a) Crawling.: Based on the URL of the target application, the crawlers explore recursively the attack surface of the web application in order to discover all requests that the application receives. To do so, for the requests they sent, the content of their responses is analyzed and the entry points they contain are collected (i.e., the points of the application that allow the insertion of inputs). However, since distinct crawlers can extract the same request, a deduplication step takes place to remove the duplicated requests. The resulting requests are uniformized into an uniform format in order to be understandable by the different scanners in the next phase. The outcome of this phase is a list with distinct requests (uniformized), indicating for each one which fuzzers found

[1]We call scanner instead fuzzer to distinguish it from the fuzzer as a whole.

it. Despite the duplicated requests being eliminated, the list contains all fuzzers that found them.

b) Attacking.: The goal of this phase is to exploit vulnerabilities existing in the web application under test by exploring the resulting requests of the previous phase. For that, each request contained in the list is delivered to scanners after it being prepared according the format of each scanner. Scanners exercise the requests with malcraft inputs to attempt to find some vulnerability. At the end, for each scanner, a list of the found vulnerabilities is provided, as well as the request that exploited them.

c) Vulnerability Identification.: This phase aims to identify the code of the exploited vulnerabilities, without accessing and inspecting the source code of the application. For that, while the attacks are being preformed, the traces of the code generated by the execution of the injected inputs on the application are extracted by a monitor, and then correlated with the information provided by the previous phase in order to identify which of them match with the requests that exploited the vulnerabilities. At the end, the identified vulnerabilities and their exploits will be outputted.

We are currently developing a prototype of our EF. The first two phases are already implemented with three fuzzers: Wapiti 3.0.3 [14], w3af 2019.1.2 [15], and OWASP ZAP 2.9.0 [16] (ZAP for short). The third phase is under development. The fuzzers were configured to detect XSS (reflected (XSSed) and stored (XSSored)) and SQLi, and to make the login authentication and establish sessions automatically. The ensemble itself, the interactions between the two phases and the correlation of their results were implemented by us in Python.

IV. Preliminary Results

The objective of the experimental evaluation was to answer the following questions: (1) Can EF discover vulnerabilities that would be missed if the fuzzers used only the requests found by their crawlers? (2) Can EF improve the vulnerability findings and the code coverage?

We evaluated EF with three known vulnerable open-source web applications: Damn Vulnerable Web App (DVWA)[2], Mutillidae[3], and Buggy Web Application (bWAPP) [4].

All crawlers had different results and there is no crawler that is the best. The rate of common requests discovered by all crawlers was 28% (29 out of the 103) in DVWA, 19.5% (90 out of the 461) in Multilidae, and 13% (52 out of the 405) in bWAPP. However, the number of equal requests outputted by two crawlers was greater compared with the previous rate. Almost or even more than 50% of the requests found by each crawler were only discovered by it (called the unique requests). Such results denote the existing discrepancy in what crawlers can discover. Also, there are requests that are missed by some crawlers that may contain vulnerabilities and that would not be found by the fuzzers whose crawlers have lost.

[2]http://www.dvwa.co.uk/

[3]https://github.com/webpwnized/mutillidae

[4]http://www.itsecgames.com/

For scanners, the results vary with the complexity of each web application. Again, there is no scanner that is the best. EF improves the precision on finding vulnerabilities, denoting that scanners are able to explore requests that were found by other crawlers. Into the EF, w3af was the fuzzer that had a higher increase in its precision. ZAP exploited more vulnerabilities in DVWA, while Wapiti had better results in the Mutillidae and bWAPP applications. We compared the reported vulnerabilities between scanners, identifying the unique and common findings. The common findings ranges 27% – 50% between all scanners, and 14% – 49% between two scanners. The unique findings in average are 39%. Into the EF, Wapiti and ZAP had a greater number of unique findings, varying according to the application tested.

The preliminary results showed that it is beneficial in having an EF for finding web vulnerabilities, specially in those cases where it is able to detect vulnerabilities that would be missed if the fuzzers would run individually. They also showed that EF performs better than fuzzers individually.

Acknowledgments. This work was partially supported by the national funds through FCT with reference to SEAL project (PTDC/CCI-INF/29058/2017, LISBOA-01-0145-FEDER-029058, POCI-01-0145-FEDER-029058) and LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020).

References

[1] Internet live stats, "Total number of websites," Mar. 2021, https://www.internetlivestats.com/total-number-of-websites/.

[2] NVD, http://nvd.nist.org.

[3] J. Williams and D. Wichers, "OWASP Top 10 2017 – The Ten Most Critical Web Application Security Risks," 2017.

[4] K. Ryan, "Patched zoom exploit: Altering camera settings via remote sql injection," Jun 2020, https://medium.com/@keegan.ryan/patched-zoom-exploit-altering-camera-settings-via-remote-sql-injection-4fdf3de8a0d.

[5] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "Kameleonfuzz: evolutionary fuzzing for black-box xss detection," in *Proceedings of the ACM Conference on Data and Application Security and Privacy*, 2014, pp. 37–48.

[6] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *Proceedings of the 28th USENIX Security Symposium*, Aug. 2019, pp. 1967–1983.

[7] L. Demetrio, A. Valenza, G. Costa, and G. Lagorio, "WAF-A-MoLE," *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, Mar 2020.

[8] J. Dahse and T. Holz, "Simulation of built-in PHP features for precise static code analysis," in *Proceedings of the 21st Network and Distributed System Security Symposium*, Feb 2014.

[9] P. Nunes, J. Fonseca, and M. Vieira, "phpSAFE: A security analysis tool for OOP web application plugins," in *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Jun. 2015.

[10] I. Medeiros, N. F. Neves, and M. Correia, "Detecting and removing web application vulnerabilities with static analysis and data mining," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 54–69, March 2016.

[11] ——, "DEKANT: a static analysis tool that learns to detect web application vulnerabilities," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, Jul. 2016.

[12] I. Medeiros and N. Neves, "Effect of coding styles in detection of web application vulnerabilities," in *Proceedings of the 16th European Dependable Computing Conference*, 2020, pp. 111–118.

[13] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner," in *Proceedings of the USENIX Conference on Security Symposium*, Aug 2012, pp. 26–26.

[14] Wapiti, https://wapiti.sourceforge.io.

[15] w3af, http://www.w3af.org.
[16] ZAP Proxy, https://www.zaproxy.org.