Improving Web Application Vulnerability Detection Leveraging 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

Keywords: Fuzzing, Web Applications, Vulnerability Discovery

Abstract: The vast majority of online services we use nowadays provide their web application to the users. The correctness of the source code of these applications is crucial to prevent attackers from exploiting its vulnerabilities, leading to severe consequences like the disclosure of sensitive information or the degradation of the availability of the application. Currently, multiple existent solutions analyse and detect vulnerabilities in the source code. Attackers, however, do not usually have access to the source code and must work with the information that is made public. Their goals are clear – exploit vulnerabilities without accessing the code –, and they resort of black-box fuzzing tools to achieve such. In this paper, we propose an ensemble fuzzing approach to check the correctness of the web applications from the point of view of an attacker and, in a posterior phase, analyse the source code to correlate with the collected information. The approach focuses first on the quality of fuzzers' crawlers and afterwards on fuzzers capabilities of exploiting the results of all crawlers between them, in order to provide better coverage and precision in the detection of web vulnerabilities. Our preliminary results show that the ensemble performs better than fuzzers individually.

1 INTRODUCTION

The Internet is currently part of our daily life, and so web applications are built upon it. There are over 1.8 billion websites active in the wild, and every month this number increases by hundreds of thousands as new web services and applications emerge online[1].

Companies are constantly faced with the decision to choose between functionality and security for their web services, as the time for security assessment may exceed the planned service launch time and often companies choose to release applications with incomplete security testing. Sohoel et al. (Sohoel et al., 2018) studied how startups consider software security in their applications. The applications tested had significant security breaches, and those who had the lowest awareness about good practices and well-known vulnerabilities had the most critical security holes. Also, none of the companies had done prior security testing and relied on secure code from third parties.

Web applications' development is getting easier, even for those who lack programming knowledge. WordPress[2] (WP) is a content management system (CMS) used to create blogs and web applications easily and intuitively, making it very appealing to inexperienced users. It is written in PHP, the server-side language mainly used to manage back-end data in web applications. Almost 80% of websites use PHP as their server-side language, and almost 40% of them use WP to manage their systems[3]. Despite their usage, PHP has not a specification language, turning it poor and limited for data validation, requiring the programmers to use the right functions and have a stable set of good practices. Pairing this lack of knowledge and bad practices with the existing vulnerabilities and limitations of the language, web applications are left with security bugs that are in the wild, vulnerable and open for anyone willing to explore their flaws.

Cross Site Scripting (XSS)[4] and SQL injection (SQLi)[5] are injection web vulnerabilities that continue to be very prevalent, specially in applications with legacy code. In fact, they are in first place in the OWASP (Open Web Application Security Project)[6] Top 10 2017, a report of the top vulnerabilities found

[1] https://www.internetlivestats.com/total-number-of-websites/

[2] https://wordpress.com/

[3] https://w3techs.com/

[4] https://owasp.org/www-community/attacks/xss/

[5] https://owasp.org/www-community/attacks/SQL_Injection

[6] https://owasp.org/

in the wild (Williams and Wichers, 2017). Its impact is huge and can result in information disclosure, data loss, denial of service, between others. XSS vulnerabilities, despite having a more moderate impact, were found in around two thirds of the applications in the OWASP study, revealing their big prevalence. Although this vulnerabilities are really well known, there are recent cases with some of them in applications used by millions (Ryan, 2020).

Static analysis examines the source code and identifies vulnerabilities without executing the code (Chess and McGraw, 2004; Jovanovic et al., 2006; Medeiros et al., 2014). Although it finds a big percentage of the vulnerabilities, it has a considerable amount of false positives. Fuzzing, on the other hand, is an (semi)automatic software testing technique to find vulnerabilities without having access to the source code, which involves providing invalid and unexpected inputs to the application and monitoring for exceptions (Sutton et al., 2007). It has an inferior false positives' rate and is able to detect issues caused by the code's interactions with other components and vulnerabilities that may be too complicated to static analysis to find (Duchene et al., 2014). However, one of the biggest challenges in black-box fuzzing tools is to determine the interactions that can change the application's state. For example, sending the same requests in a different order or with different inputs can result in different paths explored. Without taking this into account, a big portion of the application code could be missed and multiple vulnerabilities overlooked (Doupé et al., 2012).

This paper presents an *ensemble fuzzing* approach for discovering vulnerabilities in web applications written in PHP, and, in a posterior phase, to identify the code that contain the vulnerabilities exploited. To the best of our knowledge, it is the first ensemble approach of fuzzing for web applications. The approach focuses first on the quality of fuzzers' crawlers and then on fuzzers' capabilities to exploit the results of all crawlers between them, in order to provide better code coverage and precision in detecting vulnerabilities. The ensemble fuzzing is composed of three open-source fuzzers, namely, OWASP ZAP[7], w3af[8], and Wapiti[9], and aims to address the issue stated above. In a second phase, the approach will be extended to identify the vulnerabilities in the applications' code by correlating information on how to characterize and exploit vulnerabilities with data provided by fuzzers' attacks and the monitoring of target applications. Our preliminary results show that it performs better than fuzzers individually and the fuzzers' performance vary with the complexity of applications.

The contributions of the paper are: (1) an ensemble fuzzing approach to find flaws in web applications, with better code coverage and accuracy; (2) an implementation of the approach with three fuzzers; (3) an experimental evaluation providing distinct results with fuzzers in the ensemble and individually.

2 WEB APPLICATION VULNERABILITIES

The difficulty of protecting the code of web application relies on treating the user inputs unduly (e.g., $_GET), leaving applications vulnerable and an easy target for attackers. Attackers inject malicious data through the application attack surface and check if they exploit some bug existent in the target application. Vulnerabilities associated with user inputs are called *input validation vulnerabilities*, because user inputs are improperly validated or sanitized, or *surface vulnerabilities*, because they are exploited through the attack surface of the application.

SQLi and reflected XSS are two vulnerability classes of this kind. SQLi is associated with malcrafted user inputs (e.g., ', OR) used in SQL statements without any sanitization, and then executed over the database through an appropriated function (e.g., *mysqli_query* in PHP). XSS injects malicious scripts (e.g., JavaScript) which are used in output functions (e.g., *echo*), allowing the exploitation of vulnerabilities that reflect data from the browser's victim to the attacker. Another type of XSS is the stored XSS, which is made in two steps. First, the malicious script is stored in some place in the target application (e.g., database, blog file), and later it is loaded and used in an output function.

3 RELATED WORK

Black-box fuzzing is a software testing technique that, without accessing the source code of the program, injects random inputs to find bugs in it (Miller et al., 1990), which involves monitoring for exceptions, such as crashes, memory leaks, and information disclosure (Sutton et al., 2007). The injected inputs are generated based on two approaches: generation and mutation. In the former, the fuzzer generates inputs without relying on previous inputs or existing seeds, usually learning input models and generating new inputs based on the learned models. In the latter, the

[7]https://www.zaproxy.org/

[8]http://w3af.org/

[9]https://wapiti.sourceforge.io/

fuzzer modifies inputs based on defined patterns and existing seeds (Sutton et al., 2007).

Chen et al. (Chen et al., 2019) proposed an ensemble fuzzing approach to increase the performance in bug discovery. First, they selected fuzzers based on their input generation strategies, seed selection and mutation processes, and the diversity of coverage information granularity. Secondly, they implemented a system global asynchronous and local synchronous that periodically synchronizes fuzzers attacking the same application, sharing between these fuzzers interesting seeds that cover new paths or generate new crashes. WAF-A-Mole (Demetrio et al., 2020) is a tool that models the presence of an adversary based on a guided mutation fuzz approach. The tool explores failing tests, which are repeatedly mutated with mutation operators. Sargsyan et al (Sargsyan et al., 2019) and Araujo et al (Araújo et al., 2020) presented a directed fuzzing approach with the goal of executing interesting code fragments as fast as possible. All these approaches are for C/C++ code, and not applicable for web applications. Our approach aims to explore the latter's flaws and uses the ensemble fuzzing concept by sharing all the different requests between the ensemble and then using each fuzzer's capability to explore those requests.

KameleonFuzz (Duchene et al., 2014) is an extension of LigRE (Doupé et al., 2012; Duchène et al., 2013), a fuzzer that performs control flow analysis to detect reflected and stored XSS vulnerabilities. LangFuzz (Holler et al., 2012) is a blackbox fuzzing tool for script interpreters. Vimpari et al (Vimpari, 2015) presented a study of open-source fuzzers, analysing the differences between them from the perspective of someone with basic knowledge in software testing and their usability. They ended up selecting 6 fuzzers: Radamsa, MiniFuzz, Burp Suite, JBroFuzz, w3af and OWASP ZAP, being the last four web application fuzzers. However, they did not explored the fuzzers' ability to exercise the results of the different crawlers between them, which is what we make.

4 ENSEMBLE FUZZING APPROACH

The ensemble fuzzing aims to improve the detection of web vulnerabilities and increase the code coverage. Also, it intents to minimize the problem of interactions that can change the application's state, presented previously (Doupé et al., 2012), by increasing the probability of exploiting a vulnerability by using several fuzzers that will exercise the same requests.

Web application fuzzers before attacking the application under test, need to inspect its attack surface to extract the URLs (web requests) they need to perform the attacks. These fuzzers contain two components – crawler and scanner[10] – to perform, respectively, the inspection of the attack surface and the attacks. In addition, the latter depends on the results from the former. However, there is no guarantee that the crawler extracts all entry points from the attack surface and compose with them all valid URLs the application contains. On the other hand, there is also no guarantee that the scanner has the ability to exercise the URLs with the correct injected code, capable of exploiting the existing vulnerabilities in the application through these URLs.

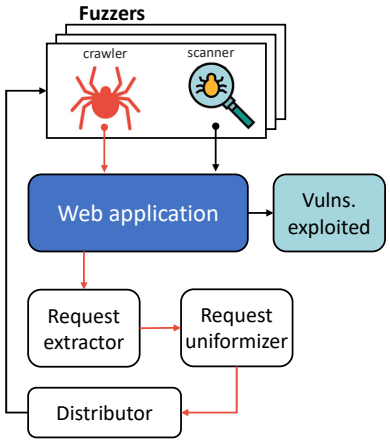


Figure 1: Overview of the ensemble fuzzing approach.

Therefore, we propose an approach that combines different fuzzers and resorts from the best they have in order to improve the resolution of these issues. The approach comprises two phases – *Crawling* and *Scanning* – and their interactions are illustrated in Figure 1, respectively, by the orange and the black arrows.

Crawling phase. This phase starts with the initial URL of the target application. Based on this URL, the crawlers explore recursively the attack surface of the web application in order to discover all requests that the application receives. The *Request extractor* module extracts the requests they made, analyses the content of their responses and collects the entry points they contain, i.e., the application points that receive user inputs. Next, the *Request Uniformizer* module compares the requests in order to remove the duplicated ones, and then converts the distinct ones into an uniform format that will allow to be used by different fuzzers in the attack phase. At the end of this phase, a list of distinct requests (uniformized) is obtained, in-

[10]We denominate scanner instead fuzzer to distinguish it from the fuzzer as a whole.

dicating for each one which fuzzers found it. Despite the duplicated requests being eliminated, the list contains all fuzzers that found them. Also, in order to get the best results of each crawler and to ensure that the results obtained from a crawler are not being influenced by other crawler's execution, the application's data is reset after each execution.

Attack phase. This phase aims to attack the web application to exploit existing vulnerabilities in its source code, using the requests resulting from the previous phase. The *Distributor* module receives the list of request and distributes it to the fuzzers. However, as each fuzzer has a different format to store the requests found by its crawler in its database, the distributor, before distributing the requests, must prepare each one according to the format of each fuzzer. Next, each fuzzer delivers the requests to its scanner for this latter to exercise them with malcraft inputs and exploit some vulnerability. At the end, for each fuzzer, a list of the vulnerabilities found is provided, as well as the request that exploited them.

Assessment. We evaluated the fuzzers based on the coverage and precision of their crawler and scanner. We compared the crawlers through the number of URL's (requests) and the type of endpoints explored. The scanners were assessed for their ability to exploit vulnerabilities, given the same list of requests for all fuzzers and their unique vulnerability findings. Note that, since all scanners use the same requests, we are able to detect limitations in the capability of exploring vulnerabilities by comparing the results of each fuzzer, as the results are not crawling dependent. Also, for the results found by each scanner, we manually inspected them to verify what vulnerability each scanner (fuzzer) found (see Section 6).

5 IMPLEMENTATION AND CONFIGURATION

The proposed approach was implemented with three fuzzers, namely the Wapiti 3.0.3, with the *Web_Spider* plugin for crawling, the w3af 2019.1.2, with the *Web_Spider* plugin for crawling, and the OWASP ZAP 2.9.0 (ZAP for short) with the *Spider* tool for crawling and the *active-scan* tool for attacking. For attacking, all tools were configured for *sqli* and *xss*. ZAP was also configured with *xss_persistent* and *xss_reflected* modules. The plugin *xss* of the first two fuzzers also allows the discovery of persistent XSS.

Fuzzers must be correctly configured to achieve a better performing and vulnerability exploitation. One important configuration they contain, besides the URL of the target application, is the user authentication on the application, as nowadays several web applications have a user authentication process for preventing unauthorized users from gaining access to sensitive information. Hence, next we describe the authentication forms each fuzzer offers, which we opted and how we configured it. In addition, to avoid bias analysis conclusions, the crawlers' depth level was set to the same level and they were run only once.

Wapiti authentication is based on session cookies. The utility *wapiti-getcookie*, given the login URL, can fetch the session cookies from the application that are later imported to the wapiti scanner. However, this utility, in some cases, does not detect the hidden values that are sent in the authentication process. To circumvent this issue, we created a script that generates the cookie with the mandatory values.

w3af offers two authentication modes: *autocomplete* and *cookie-based*. For the former, the user provides the login URL, the authentication parameters and some information about a successful login. It works well most of times, however, when it does not, we use the second mode which its configuration is very similar to Wapiti and where the cookie needs to be converted to the w3af-readable format.

ZAP has multiple ways of dealing with authentication. We explored two of them: *form-based* and *script-based*. In both, we must supply some application context, such as the login and logout regex, and the username and password. The form-based mode automatically detects the data and format required for authentication, given the login page, and anti-CSRF tokens if the name of the tokens used by the website is in the ZAP Anti-CSRF list. Although this automation seems useful, in some cases it is unable to find the correct data format, resulting in failed authentication attempts. The script-based mode resolves this issue. In this mode, the authentication data is provided in a script that, when executed, will perform the actions required for authentication. This mode achieves better results than the first, and so we opted by it.

6 EXPERIMENTS AND EVALUATION

The objective of the experimental evaluation was to answer the following questions: (1) Can the ensemble fuzzing lead to the discovery of vulnerabilities that would be missed if the fuzzers used only the requests found by their crawler? (2) Can the ensemble fuzzing improve the overall coverage and precision of the vul-

nerability detection?

6.1 Tested Web Application

The ensemble fuzzing was evaluated with three known vulnerable open-source web applications.

DVWA. Damn Vulnerable Web App (DVWA[11]) is a PHP/MySQL web application designed to be vulnerable to SQL injections, XSS and other vulnerability classes. The application requires user authentication, few pages can be accessed without an active session, and the different pages can be accessed via simple hyperlink tags inside of list items. It has also the option of changing the level of security used. For our testing purposes we used the level "low" (no security).

Mutillidae. Mutillidae[12] is a deliberately vulnerable web application that contains at least one vulnerability type belonging to OWASP Top 10 (Williams and Wichers, 2017). To access its different pages, nested HTML unordered lists, that change with "onmouseover" events, are used. There is a base URL and most of the pages are accessible by changing the value of a variable in the query string.

bWAPP. Buggy Web Application (bWAPP[13]) is a vulnerable web application with over one hundred vulnerabilities, including those described in OWASP Top 10. Such as DVWA, it also requires authentication and the pages accessible without it are limited and vulnerability free.

6.2 Crawlers Evaluation

This section has the goal of assessing the capabilities of the crawlers and understanding their discrepancies.

Before crawling the web applications, a manual analysis was made to identify how is performed the authentication process, and the entry points that would modify the application's state in an unwanted way, as for example the logout endpoint. Although this last identification requires some manual work, it leads to more constant results, and so to a more fair and complete crawler comparison.

The evaluation of the crawlers was based on the results provided by the Request Uniformizer module, where we compared the requests executed by each crawler pursuing the following criteria: (i) requests with the same method (eg., GET); (ii) requests with the same base URL; (iii) requests with the same variables and same values in the query string, despite order. Initially we considered similar requests that had the same variables in the query string, despite it's value. Although the majority of the requests with the same variables lead to the same page, in some cases the value determined the page to be presented. To avoid missing this endpoints, we compared the values as well. (iv) requests with the same post data. This criteria was defined for simplistic reasons, as comparing the data parameters with all the types that are possible to send is not trivial.

Based on these four criteria, we were able to automatically compare the requests made by all crawlers and verify which ones represent the unique and similar requests between crawlers. Figure 2 presents the results of this comparison for each web application.

As we can observe, there is no crawler that is the best. All crawlers had different results. The rate of common requests discovered by all crawlers was 28% (29) in DVWA, 20% (90) in Multilidae, and 13% (52) in bWAPP. However, the number of equal requests outputted by two crawlers was grater compared with the previous rate. For example, for Multilidae and bWAPP it was, respectively, 35% (161 out of 461) and 45% (182 out of 405). Almost or even more than 50% of the requests found by each crawler were only discovered by it (called the unique requests).

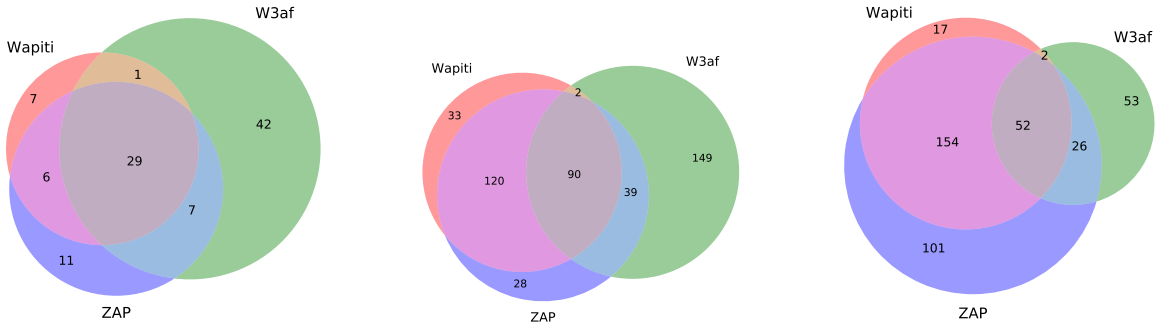
Some interesting unique requests were found in DVWA and Mutillidae. For DVWA, ZAP unique URL's are related to the login page and pages that are accessible from there, in which w3af and Wapiti excluded from their results. Of the few unique requests found by Wapiti, one has an entry point only accessible through a redirect Javascript function. ZAP misses this completely and w3af reaches the function but excludes the URL due to an erroneous redirect. As we will see in Section 6.3 this request will allow the exploitation of vulnerabilities. w3af unique requests are mainly images that are missed or ignored by the other fuzzers. For Multilidae, the unique URLs for ZAP and Wapiti are related to a public *phpmyadmin* service that they were able to detect, while for w3af are related to folders that have common names, such as "/javascript" or "/webservices".

These results denote the existing discrepancy in what crawlers can discover. Also, in the first instance, we can say that there are advantages in crawling applications with an ensemble fuzzing, because 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.

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

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

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



(a) DVWA (b) Mutillidae (c) bWAPP

Figure 2: Number of URLs found by crawlers of each fuzzer when inspect each web application.

Table 1: Number of successfully explored vulnerabilities in the tested web applications.

Web App	Vulnerability	Wapiti	Wapiti-EF	W3af	W3af-EF	ZAP	ZAP-EF	False Positives
DVWA	SQLi	2	2	3	3	3	3	1
	Reflected XSS	5	5	1	5	7	8	1
	Stored XSS	0	0	0	2	2	2	0
Mutillidae	SQLi	17	17	4	12	9	9	0
	Reflected XSS	52	54	26	43	22	24	1
	Stored XSS	17	17	1	11	3	3	1
bWAPP	SQLi	1	1	0	1	0	0	0
	Reflected XSS	9	21	1	13	19	20	11
	Stored XSS	1	4	0	2	1	1	0

6.3 Scanners Evaluation

In this section it is assessed the capabilities of the fuzzers' scanners, individually and in the ensemble. Also, the section intents to find out if scanners have the ability to explore requests found by other crawlers than their own fuzzer and, hence, exploit vulnerabilities from them.

For the individual evaluation, each scanner was run as standalone with the requests that its crawler discovered. On the other hand, in the ensemble, each scanner run with the requests found by the ensemble fuzzing and provided by the distributor, after they being formatted according to the fuzzer format. Hence, having a list of requests gotten by the three fuzzers' crawlers, each fuzzer was fed with the list, and each fuzzer's scanner used the requests, exercised them with diverse inputs and carried out attacks on the web applications to try to exploit SQLi and XSS.

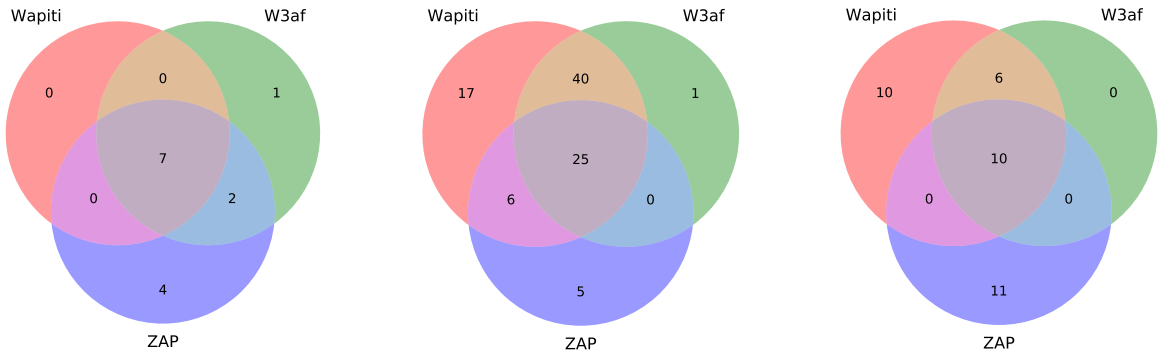
Table 1 summarize the results obtained from the scanners on both evaluations, where columns 3, 5 and 7 regard individual fuzzers and columns 4, 6, and 8 to fuzzers in the ensemble (EF). The results vary with the complexity of each web application. Again, there is no scanner that is the best. It is visible that fuzzers within the ensemble improves their precision on discovering vulnerabilities, denoting thus that they are able to explore request that were found by other crawlers. Also, w3af-EF was the fuzzer that had its precision more increased. ZAP-EF exploited more vulnerabilities in DVWA, while in the Mutillidae and bWAPP applications, Wapiti-EF had better results.

In order to assess the quality of the scanners, we compared the reported vulnerabilities between each other, identifying the unique and common findings. Figure 3 displays the vulnerability distribution for each application. The common findings ranges 27% – 50% between all scanners, and 14% – 49% between two scanners. The unique findings in average are 39%. Wapiti-EF and ZAP-EF had a greater number of unique findings, varying according to the application tested. w3af-EF, on the other hand, had a low rate of unique findings (1 or 0).

During the analysis we found interesting cases when we compared the results of the scanners and the requests of the crawlers. Also, we tried all outputted results on the tested web applications (i.e., we performed the attacks manually) and we found that some of them were false positives. The next two sections we present some of these cases.

6.3.1 Identifying Interesting Cases

We considered an attack interesting if it fits in one of the following cases: (i) the crawler of the fuzzer F_x missed the URL_i and the remaining crawlers found it but were unable to explore it. In the ensemble, only



(a) DVWA (b) Mutillidae (c) bWAPP

Figure 3: Vulnerability distribution by each fuzzer when attacking each web application.

F_x was able to explore the URL_i and found the vulnerability; (ii) only the crawler of the fuzzer F_x was able to find the URL_i and explored it. In the ensemble, all fuzzers were able to explore URL_i and found the vulnerability. (iii) both fuzzers F_x and F_y found the same base URL, however F_y was not able to retrieve a valid request from it because F_y missed a variable or sent an erroneous value. F_x discovered the correct request (URL_i) and was able to explore it. In the ensemble, F_y also was able to explore URL_i.

We found 18 interesting cases, namely 6 from the first case, 11 from the second and 1 from the last. Table 2 presents an interesting XSS vulnerability we found of each case. We used a binary representation to identify if the vulnerability was explored or not. The "–EF" columns represent the attacks performed in the ensemble, and the others columns represent the attacks performed by the fuzzer individually.

Table 2: An example of XSS of each case.

URL	Wapiti	Wapiti-EF	W3af	W3af-EF	ZAP	ZAP-EF
/bWAPP/csrf_3.php	0	1	0	0	0	0
/DVWA/vulnerabilities/csrf/test_credentials.php	1	1	0	1	0	1
/bWAPP/xss_php_self.php	0	1	0	1	1	1

In the first row, it is represented a vulnerability where the request was missed by the Wapiti's crawler but caught by other one's crawlers, in which these were unable to exploit that vulnerability. Observing the application code, there is a submit button that sends some values, including one that is in a hidden input Wapiti was able to manage this hidden input, unlike the other fuzzers.

The second row is the case mentioned in Section 6.2. Wapiti's crawler is the only one that found the URL, being able to explore it. When the URL is shared between all fuzzers, all of them exploited the vulnerability too.

In the last row, the w3af's crawler missed the URL, unlike the other crawlers. But, ZAP and Wapiti returned different parameters for this URL. Wapiti only found the parameters from a form that is used to update information, missing a parameter that is needed to send the request; hence, its exploration failed. ZAP found the correct URL and explored it successfully. When the correct URL (from ZAP) was explored by all fuzzers, they all were able to explore the vulnerability. The code that validates this request is presented in Listing 1. The vulnerable code can only be accessible if all the parameters *form*, *firstname* and *lastname* are set, making the parameter *form*, discovered by ZAP, crucial in the vulnerability exploitation.

Listing 1: Source code that validates the requests sent to /bWAPP/xss_php_self.php

```
if(isset($_GET["form"]) && isset($_GET["firstname"]) && isset($_GET["lastname"])) {
  $firstname = $_GET["firstname"];
  $lastname = $_GET["lastname"];
  if($firstname == "" or $lastname == "")
    echo "<font color=\"red\">Please enter both fields...</font>";
  else
      echo "Welcome ".xss($firstname)." ".xss($lastname);
}
```

6.3.2 Identifying False Positives

By manually performing all the attacks reported by each fuzzer's scanner, we confirmed a total of 15 false positives: 13 from ZAP and 2 from w3af. The last column of the Table 1 presents them on the three applications. Most of the ZAP false positives were XSS attacks in the application *bWAPP*, as the values of the variables instantiated by ZAP were not used, since it was only checked if the variable was set. One example of a w3af false positive was in DVWA, as it reported a SQLi attack in "*/DVWA/instructions.php?doc=*", where the attacked parameter was "*doc*". Listing 2 presents a fragment of code that

deals with this parameter. Here, the value of the variable is only used to compare with values of an existing array, and if it does not match any of it's values, a default one is used. The value is not used anywhere else in the application, and so no SQLi attack is possible.

Listing 2: Source code of the page attacked by the w3af fuzzer that resulted in a false positive.

```
$docs = array(
        'readme'=> array('file' => 'README.md'),
        'PDF'=> array('file' => 'docs/pdf.html'),
);
$selectedDocId = isset($_GET['doc']) ? $_GET['doc'
    ] : '';
if( !array_key_exists($selectedDocId, $docs) ) {
        $selectedDocId = 'readme';
}
$readFile = $docs[$selectedDocId]['file'];
```

7 CONCLUSIONS

The paper presented an ensemble fuzzing approach for web applications to improve the detection of vulnerabilities by exploring all returned requests of all fuzzers' crawlers and increase the code coverage of such applications. The approach was implemented with three open-source web fuzzers and evaluated with three well known vulnerable applications. The preliminary results are promising and showed that there are advantages to have such ensemble, specially in those cases where it is able to detect vulnerabilities that would be missed if the fuzzers would run individually. As a further step, we want to identify in the code of the applications the vulnerabilities exploited by inspecting the code traces resulting from fuzzers.

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

Araújo, F., Medeiros, I., and Neves, N. (2020). Generating tests for the discovery of security flaws in product variants. In *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation Workshops*, pages 133–142.

Chen, Y., Jiang, Y., Ma, F., Liang, J., Wang, M., Zhou, C., Jiao, X., and Su, Z. (2019). Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *Proceedings of the 28th USENIX Security Symposium*, pages 1967–1983.

Chess, B. and McGraw, G. (2004). Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79.

Demetrio, L., Valenza, A., Costa, G., and Lagorio, G. (2020). WAF-A-MoLE. *Proceedings of the 35th Annual ACM Symposium on Applied Computing*.

Doupé, A., Cavedon, L., Kruegel, C., and Vigna, G. (2012). Enemy of the state: A state-aware black-box web vulnerability scanner. In *Proceedings of the USENIX Conference on Security Symposium*, pages 26–26.

Duchène, F., Rawat, S., Richier, J., and Groz, R. (2013). Ligre: Reverse-engineering of control and data flow models for black-box XSS detection. In *Proceedings of the Working Conference on Reverse Engineering*, pages 252–261.

Duchene, F., Rawat, S., Richier, J.-L., and Groz, R. (2014). Kameleonfuzz: evolutionary fuzzing for black-box xss detection. In *Proceedings of the ACM Conference on Data and Application Security and Privacy*, pages 37–48.

Holler, C., Herzig, K., and Zeller, A. (2012). Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium*, pages 445–458.

Jovanovic, N., Kruegel, C., and Kirda, E. (2006). Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the workshop on Programming languages and analysis for security*, pages 27–36.

Medeiros, I., Neves, N. F., and Correia, M. (2014). Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *Proceedings of the International Conference on World Wide Web*, pages 63–74.

Miller, B. P., Fredriksen, L., and So, B. (1990). An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44.

Ryan, K. (2020). Patched zoom exploit: Altering camera settings via remote sql injection.

Sargsyan, S., Kurmangaleev, S., Hakobyan, J., Mehrabyan, M., Asryan, S., and Movsisyan, H. (2019). Directed fuzzing based on program dynamic instrumentation. In *Proceedings of the International Conference on Engineering Technologies and Computer Science*, pages 30–33.

Sohoel, H., Jaatun, M., and Boyd, C. (2018). Owasp top 10 - do startups care? pages 1–8.

Sutton, M., Greene, A., and Amini, P. (2007). *Fuzzing: brute force vulnerability discovery*. Addison-Wesley.

Vimpari, M. (2015). An evaluation of free fuzzing tools.

Williams, J. and Wichers, D. (2017). OWASP Top 10 - 2017 rcl - the ten most critical web application security risks. Technical report, OWASP Foundation.