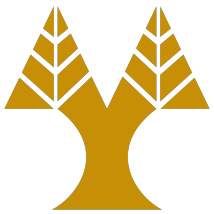
Thesis Dissertation

WEBFUZZ: IMPLEMENTATION OF A GREY-BOX FUZZER FOR WEB APPLICATIONS

Marcos Antonios Charalambous

UNIVERSITY OF CYPRUS



COMPUTER SCIENCE DEPARTMENT

December 2020

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

webFuzz: Implementation of a Grey-box Fuzzer for Web Applications

Marcos Antonios Charalambous

Supervisor

Dr. Elias Athanasopoulos

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

December 2020

Acknowledgments

I would like to express my sincere gratitude to my thesis supervisor Assistant Professor Dr. Elias Athanasopoulos for his crucial guidance, encouragement, support and advice he provided to help me complete and accomplish this dissertation. During the past year, Dr. Athanasopoulos' interest, enthusiasm and expertise in the field of Cybersecurity has undoubtedly been a source of inspiration. He sparked my interest in computer security which has led to this thesis. All his positive input made this endeavour an exciting experience.

Also, I would like to thank my fellow students Demetris Kaizer and Orpheas Van Rooy for their excellent teamwork and contribution in a bigger project that combines each of our thesis.

Moreover, I want to thank all my professors from whom I received invaluable knowledge enabling me to become a Computer Scientist after my four years of study at the Department of Computer Science of the University of Cyprus.

Finally, I would like to thank my family and friends for being with me during my trials and tribulations, supporting me at every step of the way.

Abstract

Testing software is a common practice for exposing unknown vulnerabilities in security-critical programs that can be exploited with malicious intent. A bug-hunting method that has proven to be very effective is a technique called fuzzing.

Specifically, this type of software testing is frequently in the form of fuzzing of native code, which includes subjecting the program to enormous amounts of unexpected or malformed inputs in an automated fashion.

This is done to get a view of their overall robustness to detect and fix critical bugs or possible security loopholes. For instance, a program crash when processing a given input may be a signal of memory-corruption vulnerability.

Although fuzzing has significantly evolved in analysing native code, web applications, invariably, have received limited attention until now.

This thesis explores the technique of grey-box fuzzing of web applications and the construction of a fuzzing tool that automates the process of discovering bugs in web applications. We design, implement and evaluate webFuzz, which is a prototype grey-box fuzzer for web applications.

Our fuzzing tool was successful in leveraging instrumentation for detecting manually-injected Reflective Cross-Site Scripting (RXSS) vulnerabilities and covering more code faster than black-box fuzzers. The functionality of webFuzz is demonstrated using popular open-source web applications written in PHP.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

In the introductory chapter to this thesis, we analyse what motivated us to explore grey-box fuzzing to begin a research project in this area. We further discuss related studies in the field of fuzzing, the contribution that our thesis makes to better understanding it while outlining the eight chapter topics that are addressed.

This thesis is focused on the core functionalities of the fuzzing tool so we do not delve into details about the *instrumentation* or the *automated bug injection* tool (Centaur) used during the evaluation in Chapter 5, as they are beyond the scope of our research.

1.1 Motivation

Despite much research done in web applications security flaws in recent years, vulnerabilities are still commonplace. The overall number of new vulnerabilities in 2019 (20,362) increased by *17.6%* compared to 2018 (17,308) and by *44.5%* compared to 2017 (14,086) [16,60]. A major cause of this phenomenon in web-apps comes from the source code itself

which more often than not is written in unsafe languages, such as PHP or JavaScript [16] opening the door to vulnerabilities. Many programmers do not have adequate expertise in secure coding, so they leave applications with vulnerabilities.

As the occurrence of security vulnerabilities in web applications has increased to an all-time high with damages inflicted costly, an automated approach is required to keep up with the number of web-apps that need vetting for vulnerabilities. DARPA funded the Cyber Grand Challenge competition in 2016 [17], with millions of dollars in prize money, to further research on automated vulnerability finding and patching. There are many bug bounty programs and capture the flag competitions dedicated to securing applications that power large organizations [34]. Our research hopes to take advantage of this heightened interest and renewed awareness.

Several approaches exist for protecting web applications by detecting and removing vulnerabilities. They usually fall into three main categories: static, concolic, and dynamic analysis systems. Static analysis tools detect vulnerabilities in source code by examining the code without executing the program [6, 38, 39, 47, 48]. Concolic execution combines both concrete execution (*i.e.,* normal code execution) and symbolic execution. Tools that utilize symbolic execution, instead of feeding the program with expected inputs, one supplies symbols representing arbitrary values [32,42]. Dynamic analysis, including fuzzing, where the web-app is monitored while exercised with malformed data in an automated fashion [24, 27, 31, 52].

Fuzzing is now widely recognised as an essential process for discovering hidden bugs in computer software. Automated software testing or fuzzing is a tried and tested method of generating or mutating inputs and passing them to programs in search of bugs. The spark in the fuzzing 'revolution' to discover bugs in software through an automated process has precipitated with the introduction of American Fuzzy Lop (AFL) [87], a state-of-the-art fuzzer that produces feedback during fuzzing by leveraging *instrumentation* of the analysed program.

In creating this *feedback loop*, fuzzers can significantly improve their performance by determining whether an input is interesting, namely, it triggers a new code path, and uses

that input to produce other test cases. That is called coverage guided fuzzing.

Software testing plays a vital role in the software development cycle because when vulnerabilities are present, they can cause severe or irreparable consequences. By exploiting software bugs, adversaries can perform data breaches, install malicious malware or even take complete control of a device [29, 44].

Detecting bugs before they get exploited is a viable but demanding task. Mainly because bugs are triggered when an unexpected input is given to the program, something that is difficult to fully simulate through statically written unit tests. That is down to the fact unit tests usually revolve around expected inputs to test the intended functionality of code [12].

Although automated software testing has become a burgeoning field of research, it still has a long way to go, especially for web applications [25]. As the Internet infrastructure expands, much more of the software written in *native code* (pre-compiled program in the CPU's *machine language*) is migrating to web applications. This process attracts many more malicious attacks on web applications. This predicates a strong need for the development of automated vulnerabilities scanners that target web applications.

1.2 Related Work

Fuzzers recently developed try to optimize the fuzzing process by proposing different methodologies [12, 13, 31, 35, 57, 72, 81]. For example, most fuzzers take advantage of *instrumentation* at the binary or source level. This is done by inserting code in the program to receive feedback when a code block is triggered so the fuzzer can adjust the generated inputs to improve code coverage. Other fuzzing methodologies utilize symbolic/concolic execution for extracting useful information about the program, using that information to improve the input generation process [30–32, 81]. However, all these fuzzers currently target finding vulnerabilities in *native code* while web applications - which do not run on *native code* - have more-or-less been neglected. A more detailed analysis is given in Chapter 7.

Traditionally, fuzzers come under three categories; black-, white- or grey-box which are clearly defined for native applications. When it comes to web applications grey-box approaches have not been defined, so our mission was to produce a prototype process inspired by work done on native applications, more precisely AFL.

1.3 Contributions

In this thesis, webFuzz is proposed. It is a prototype grey-box fuzzing tool for web applications. Today, the only fuzzers available for web applications are developed to behave in a black-box fashion [25]. That is to say, they use brute force to bombard their targets with URLs that embed known web-attack payloads. There have been breakthroughs with white-box fuzzing also [3, 9], that combine static analysis and concolic testing with fuzzing.

Unlike the black-/white-box fuzzing approach, webFuzz initially instruments the targeted web application by adding code that tracks all control flows triggered by an input and notifies the fuzzer, accordingly. Notifications can be embedded in the web application's HTTP response using custom headers or outputted to a shared log file or memory region. Subsequently, the fuzzer sends requests to the target and analyses the responses to detect any interesting requests that would later help to improve the code coverage and as a result, trigger vulnerabilities embedded deep in the web application's code.

The following contributions are made in this thesis:

1. We design, implement and evaluate webFuzz, a prototype grey-box fuzzer created for discovering vulnerabilities in web applications. webFuzz applies *instrumentation* on the target web application for guiding the entire fuzzing process. *Instrumentation* can be applied to the Abstract Syntax Tree (AST) level of PHP-based web applications for establishing a *feedback loop* and utilizing it to increase code coverage.

2. We thoroughly evaluate webFuzz in terms of efficiency in finding unknown bugs, of code coverage and throughput. For a better understanding of the measurement ca-

pabilities of webFuzz, we compared our results with three existing black-box web-application vulnerability scanners. Indicatively, webFuzz can cover about 21.5% of WordPress, which has a codebase of approximately half a million lines of PHP code, in 4.2 days (6000 minutes) of fuzzing. As expected, webFuzz is slower, in terms of throughput, due to the involved *instrumentation*. Another popular fuzzer, Wfuzz [49] is three times faster when fuzzing Drupal. This is to be expected since the reduction of the throughput from the *instrumentation* pays off in increased coverage in the long run. Finally, webFuzz, compared to the other three fuzzers, detects more injected vulnerabilities (30 with the second one being Wfuzz with 28) during a fuzzing session lasting 65 hours. The evaluation of webFuzz is detailed in Chapter 5.

3. To foster further research in the field webFuzz will be released as open-source.

1.4 Thesis Outline

This thesis has eight chapters. In the first chapter, we presented our inspiration for undertaking this research, related work on the topic and the contributions made in this thesis. In the second chapter, we state any relevant background information required to grasp the perspective of this work.

Continuing to the third chapter, the architecture of the tool is discussed on a higher level without delving into too much implementation detail. The fourth chapter is dedicated to discussing the technical aspects of the fuzzing tool developed. The fifth chapter evaluates how well webFuzz performs in finding bugs, code coverage and throughput against other fuzzers.

In the sixth chapter, we review the limitations faced during the research process while unfurling what future plans we have for our tool. In the seventh chapter, we elaborate on the related work made in the area of fuzzing in recent years. In the eighth and final chapter, we summarise and reflect on the research done, concluding on the evaluation of our approach.

Chapter 2

Background

Contents

In this chapter, we provide background information giving a detailed understanding of several key points about this thesis. First, we define what a Cross-Site Scripting bug is in web applications by giving specific examples of how this vulnerability may occur. Then, we briefly discuss what fuzzing is and the various categories that constitute it, showing how *instrumentation* helps when used during grey-box fuzzing. Towards the end, this chapter discusses the concept of concurrency in Python and concludes with the containerization of services using Docker.

2.1 Web Application Bugs

The internet has been growing exponentially since its commercial inception in 1969 with the creation of ARPANET. Although there are over 1 billion pages currently online, writing a web application secure from any vulnerability can be extremely difficult. Every significant web application, especially large-scale ones that are composed of thousands

of Lines of Code (LoC), have dangerous bugs in them. Popular social-networking site Facebook had bugs in its 100 million LoC that resulted in 50 million users having their personal data exposed [44, 55]. Such problems are not exclusive to complex web applications. Even the simplest web-apps can be the root of irreparable damage when they are exploited by attackers with ulterior motives.

In fact, web application vulnerabilities are among the most frequent vulnerabilities reported in the Common Vulnerabilities and Exposures database (CVE). According to CVE 2019 data, Denial of Service (DoS) (19.2%) is ranked second and Cross-Site Scripting (XSS) (12.5%) is fourth among the top Cybersecurity vulnerabilities [15].

The Open Web Application Security Project (OWASP) Top 10 represents a broad consensus on the most critical security risks to web applications [60]. One of the most pressing security issues on the Internet, according to the OWASP list, is Cross-Site Scripting (ranked 7).

XSS flaws occur whenever an application includes untrusted data in its web page responses without validating or escaping them first. In other words, the web application accepts input from the user and then attempts to display it without filtering for HTML tags or script code, such as JavaScript. JavaScript is an essential part of web applications as it is used during both frontend and backend development with all major web browsers having a dedicated engine to execute `.js` code. So, allowing such untrusted code to be executed can hijack the browser, deface the website, redirect the user to dangerous sites and many other attacks. Some XSS types include Reflected (Non-Persistent or Type II), Stored (Persistent or Type I) and DOM-based (Type-0).

Reflected XSS [63] vulnerabilities arise when arbitrary data is copied from a request and echoed into the application's immediate response. By not filtering the data input, scripting language code included within a request can be executed, whatever its content. In the case of Stored XSS vulnerabilities, the malicious payload is permanently stored in storage such as a database residing on a server and is only later outputted by an unsuspecting query. Locations, where Stored XSS may occur, include Web forums or blog comments.

webFuzz focuses on detecting bugs that can lead to both Reflected or Stored Cross-Site

Scripting, that are among the most common of XSS attacks. A step-by-step illustration of the latter can be seen in Figure 2.1 and the former in Figure 2.2. In both illustrations, the attacker and victim are represented by webFuzz.

It is imperative that we understand what an RXSS (Reflected XSS) bug typically looks like, in order to grasp the thesis' perspective on such vulnerabilities. Usually, RXSS is caused due to a failure to sanitise user input. For instance, let us assume that we have a simple login page with two input fields: the username and password. The login page also displays the appropriate error messages back to the user if the login fails. An implementation of this in PHP could look something like Listing 2.1.

```
1  <?php
2  $username=$_POST['username'];
3  $pwd=$_POST['password'];
4  if (search_username($username)) {
5      if (match_username_password($username, $pwd)) {
6          // do normal login procedures
7      } else {
8          echo 'Wrong Password';
9      }
10 } else {
11         echo 'Error' . $username . 'was not found.';
12 }
13 ?>
```

Listing 2.1: *Vulnerable login form*

The above code is faulty for two reasons. First, knowing a username exists offers clues for an attacker to guess a set of correct credentials much faster since only the password is left to find. But this design choice is not linked with Cross-Site Scripting. The source of the bug is on line 11 where the error message "`the $username was not found`" is displayed. Because `$username` is a variable that has not been sanitized, an attacker can inject malicious payload in this field that will be interpreted by the HTML parser according to whatever its content is.

`Exploit`: A victim is tricked into submitting a `form` located in an attacker-controlled

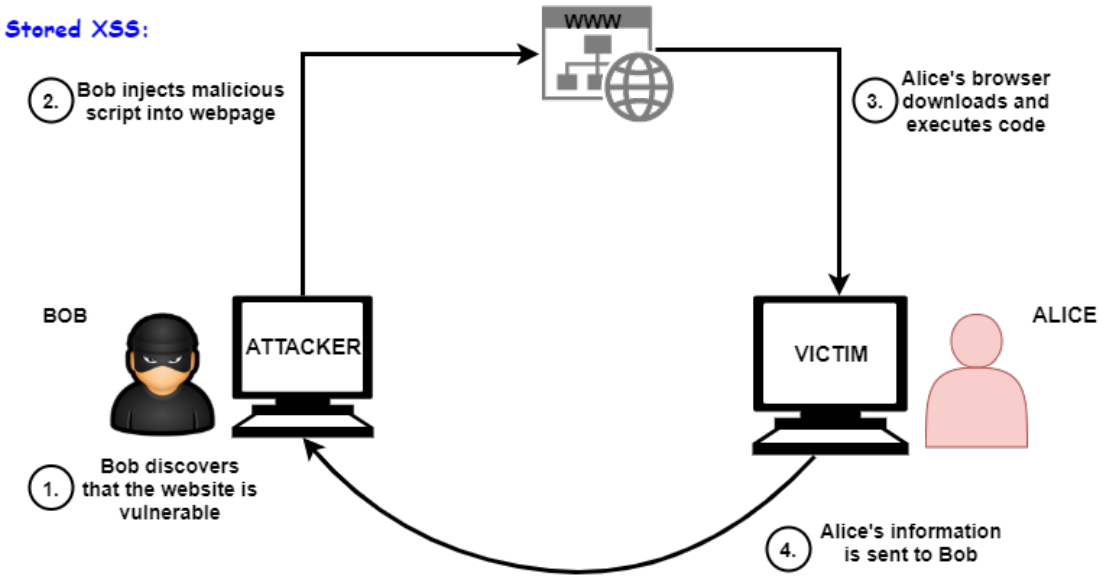


Figure 2.1: *How Stored Cross-Site Scripting can be exploited by an attacker*

website. The malicious payload is designed to trigger the vulnerability found in the above login `form` (Listing 2.1). As soon as the `form` is submitted, the vulnerable login page is opened with the XSS script executed in it. When the victim tries to login, the XSS script can easily send the credentials to the attacker as well.

Another example of a vulnerable page is highlighted in Appendix C. This presents a more complex scenario than the one shown in Listing 2.1, since a *magic number* must be guessed first before entering a conditional branch to reach the XSS vulnerability.

Defeating XSS attacks is not dissimilar to defending against other types of code injection. The input must be sanitized. User input containing HTTP code must be escaped or encoded to avoid its execution. System-wide measures such as Content Security Policy (CSP) [46] may be enabled to eliminate or mitigate XSS attacks. Nevertheless, flaws such as Buffer Overflows (CVE ranked 3 [15]) or Cross-Site Scripting issues comprise a majority of security incidents that malicious hackers exploit daily.

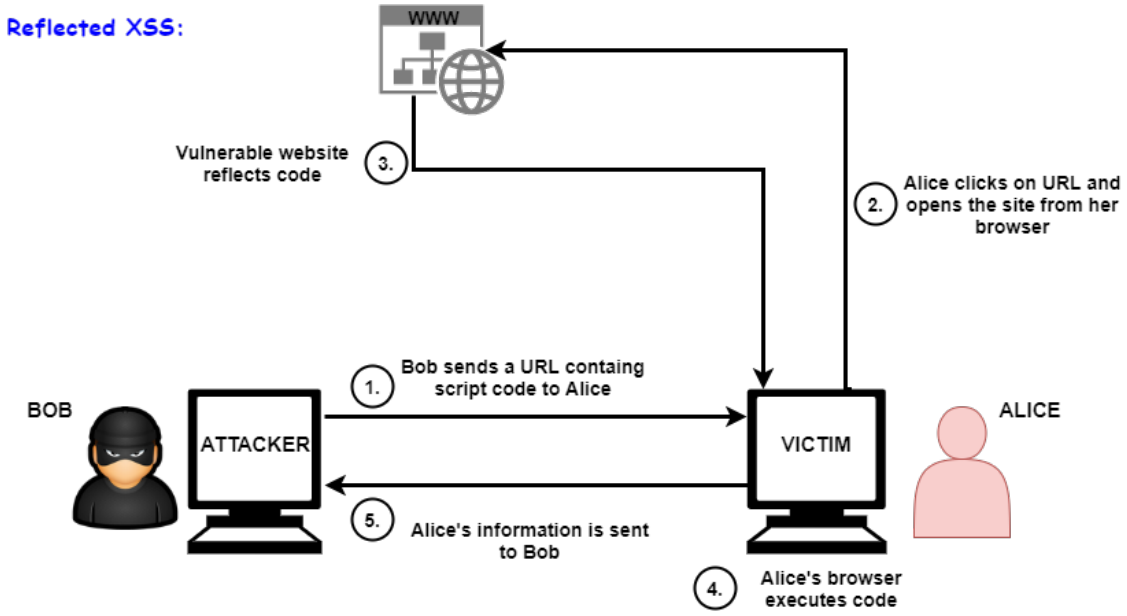


Figure 2.2: *How Reflected Cross-Site Scripting can be exploited by an attacker*

2.2 Fuzzing

A promising method for discovering unknown vulnerabilities in programs and web applications proven to be very effective, is a technique called fuzzing (or fuzz testing) [51]. Fuzzing was invented by Barton Miller at the University of Wisconsin, as one of several tools to test UNIX utilities [52]. With this quality assurance technique, the software is exercised using a vast number of anomalous inputs for inferring if any of them introduce security-related side-effects. A fuzzer, the tool that automates the aforementioned stress-testing process can be categorized in relation to its awareness of the program structure as black-, white-, or grey-box [83].

A black-box fuzzer treats the program as a 'black box' and is unaware of internal structures. It conducts its test on the target through external interfaces and produces random inputs using no information about the target's underlying structure. More often than not, black-box fuzzers are only able to scratch the surface and expose "shallow" bugs [58]. For example, the branch of the conditional statement `"if x==5:"` has only one in 2^{32} chance of being executed if `x` is a randomly chosen 32-bit input value (*i.e.,* an integer). That intuitively explains why black-box testing usually provides low code coverage and

is unable to find bugs nestled deep in the program [32].

A white-box fuzzer infers source code knowledge, such as source code auditing, to reveal flaws in the software. It leverages program analysis to systematically increase code coverage or to reach certain critical program locations otherwise unreachable. Program analysis can be based on either static or dynamic analysis, or their combination [61]. They may also leverage symbolic execution to derive what inputs cause each part of a program to execute [42]. It makes them effective at exposing bugs that hide deep in the program. By studying the application code, you can detect optional or proprietary features, which should be tested as well.

A fuzzer is considered grey-box when it leverages *instrumentation* rather than program analysis to glean information about the coverage of a generated input from the program it tries to fuzz [18, 87]. Adopting this process significantly reduces the 'guesswork' that occupies black-box fuzzers. This thesis explores in detail grey-box fuzzing, which combines elements of the white-box and black-box approaches since it uses the internals of the software, to a minimal extent, to help generate better test cases without needing full access to the code.

We also explored the feasibility of constructing a fuzzing tool that will automate the process of discovering bugs in web applications. This was done by providing randomized invalid inputs to an under-analysis instrumented web application, mutating these inputs according to the feedback received and finding test cases that cause a systems crash or make them act inappropriately to prevent exploitable vulnerabilities.

2.3 Instrumentation

Typically, a fuzzer is considered more effective if it achieves a higher degree of code coverage. To be able to trigger any given bug, the fuzzer must first execute the code where the bug lies. So, widening code coverage increases the chances of executing unsafe pieces of code where bugs may reside. As mentioned in the previous section, using *instrumentation* may be the key to achieving a higher code-coverage percentage.

However, some studies have failed to reach a consensus on the correlation between code coverage and the number of bugs found [37,43]. Increasing global code coverage may be less effective in finding new bugs than, for instance, focusing on widening code coverage in targeted error-prone code areas as AFLGo [8] does. Therefore, code coverage should be considered a secondary metric and the number of bugs found as the primary [43]. Nevertheless, measuring coverage is crucial for any fuzzer.

Available fuzzers for web applications act in a black-box fashion [25]; by applying brute force to the target with URLs that embed known web-attack payloads with little or no information about the underlying structure of the target. In contrast, webFuzz firstly instruments a web application by adding code that tracks all control flows triggered by an input and notifies the fuzzer, accordingly. Notifications can be embedded in the web application's HTTP response using custom headers or it can be outputted to a shared file or memory region.

Consequently, the fuzzer sends requests to the target and analyses the responses to detect any requests of interest that would later help to improve the code coverage and as a result, trigger vulnerabilities nested deep in the web application's code. To measure code coverage we calculate the ratio of how many basic blocks were visited in respect to the total number of basic blocks instrumented. It gives us a good idea of the coverage but omits information such as sequences of basic blocks that were visited.

We instrumented web applications for delivering feedback once being fuzzed. As opposed to native applications, where several options exist for instrumenting their source or binary representation. We decided to instrument web applications by modifying the AST of PHP files and then reverting it to source code form. This provided us with crucial feedback on the basic blocks that are visited during the analysis. *Instrumentation* performed by webFuzz on our targeted web application is similar to how AFL instruments binaries, but adapted to work in web applications.

2.4 Concurrency

Concurrency is defined as working on multiple tasks at the same time [75]. However, in Python this does not mean that they work in parallel, since only one core of the CPU is active at any given time. Instead, each task takes turns in occupying the core and executing their code. When a task is interrupted, its state is stored so it can restart from the point where it left off.

Concurrency aims to speed up the overall performance of input/output (I/O) bound programs, whose performance can be slowed dramatically when they are obliged to frequently wait for I/O operation from an external resource. An example of such resources are requests over the internet or any type of network traffic that takes several orders of magnitude longer than CPU instructions. An illustration of the above can be seen in Figure 2.3:

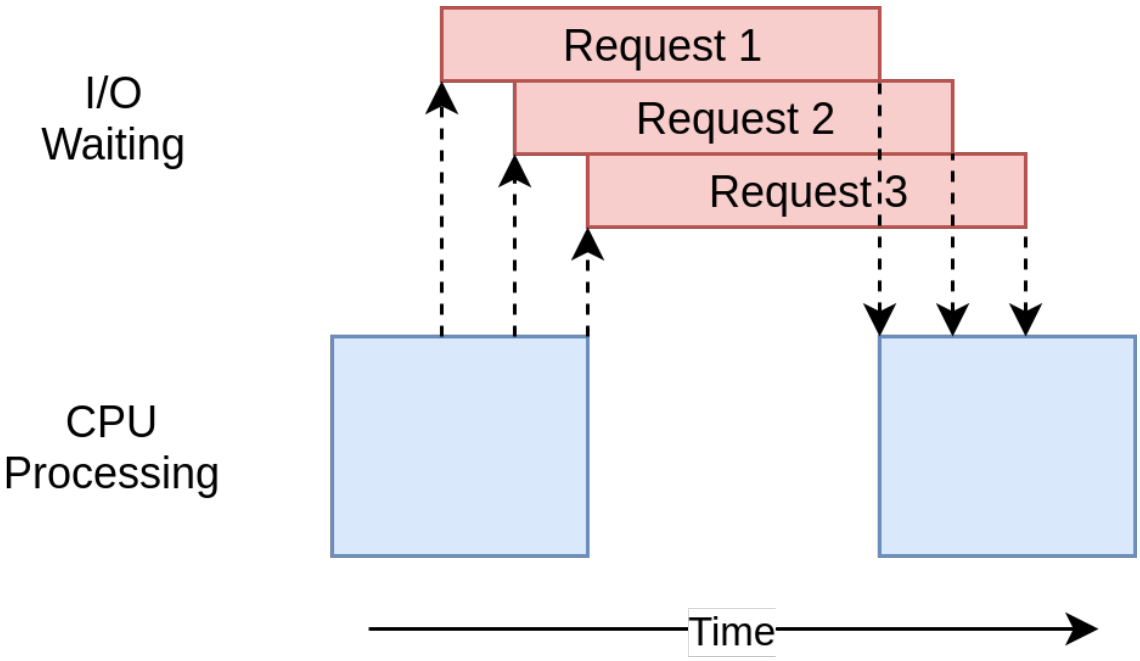


Figure 2.3: *Requests over the internet processed concurrently* [75]

In Python, concurrency is expressed either through the Threading or AsyncIO (short for Asynchronous Input Output) [73] modules. Due to the infamous *Global Interpreter Lock* (GIL) [76] Python has, both AsyncIO and Threading, they are single-threaded, single-process design. There was no clear advantage in using the latter so, AsyncIO was

opted for instead, although initial work was done with `threading` it was shelved. Not to mention the added complexity of using threads and making the program thread-safe.

Briefly, GIL ensures there is only one thread running at any given time, thus making the use of multiple cores/processors with threads infeasible. In the Python community there is a general rule of thumb when it comes to I/O-bound problems; "Use asyncio when you can, threading when you must". More information on the AsyncIO module and its use in the webFuzz implementation can be found in Chapter 4.

2.5 Docker

Docker containers [22] provide developers the commodity for creating software locally with the knowledge that it will run identically regardless of the host environment [53]. Containers are an encapsulation of an application's dependencies that share resources with the host OS, unlike frequently used *Virtual Machines*. During the evaluation, detailed in Chapter 5, a docker-compose `YAML` file was created to allow multiple containers to be initiated and managed at the same time with a set of pre-defined configurations.

Services are deployed with containers through the use of Docker images. A Docker image consists of a collection of files that bundle together all the essentials, such as installations, application code and dependencies required to configure a fully operational container environment. Official Docker images can be found at Docker Hub [21].

Chapter 3

Architecture

Contents

This chapter illustrates the general design of the fuzzer without going into too much technical detail. The in-depth breakdown of the fuzzer's components is thoroughly described in Chapter 4. The key components elaborated on in this chapter are the high-level working view of webFuzz, the mutations made to the requests, and the different vulnerabilities in web applications that webFuzz is designed to detect.

3.1 Fuzzing Session

webFuzz constitutes two intertwined components that work together in providing a guided fuzzing approach to find web application vulnerabilities. The first component is the *instrumentation* of the target web application that provides feedback to the fuzzer on which basic blocks were visited to deduce if new control paths have been discovered. For the *instrumentation* process, webFuzz adopts similar techniques to how AFL instruments binaries but in our case, we adapted them to work in web applications.

The second component is the fuzzing application with its core functionalities responsible

for sending requests from a dynamic request queue, reading their respective responses, parsing them to provide an informed decision on what the next request should be and displaying various statistics about the fuzzing session to the user. The fuzzer also features an inbuilt *crawler* that scans the HTML responses to detect `anchor` and `form` elements that can give new, unseen paths for the web application to explore.

A regular fuzzing session using webFuzz is seen in Figure 3.1. It displays the process from the point the request is sent, up to the stage where a response is received. A request can be produced in one of two ways; it can be in a mutated form of a previously made request which turned out to be interesting or as a new link discovered by the inbuilt *crawler* but has not been visited yet. When the response is received, it is parsed in order to extract the execution time, vulnerabilities it may have triggered, coverage score, and to record newly discovered links.

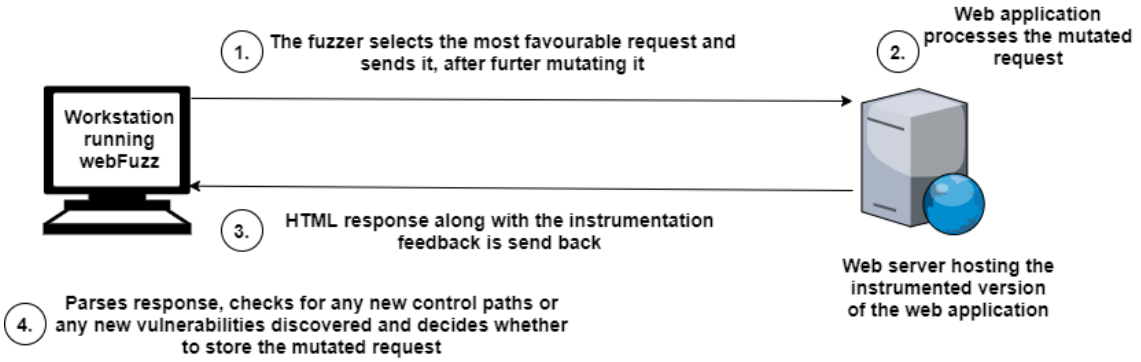


Figure 3.1: *High-level overview of a webFuzz fuzzing session*

3.2 Mutations

In most cases, sending randomly generated inputs are quickly rejected by the target program as the data is syntactically invalid. One way to increase our chances of obtaining valid input is through mutational fuzzing where small modifications are made to existing inputs that may still keep the input valid, yet exercise new behaviour. Mutation-based fuzzers such as EFS [18] and AFL [87] actively see the code paths executed on the target for each input they send and make adjustments accordingly.

For creating fuzz test cases, mutation is a core part of the fuzzing process. It is vital be-

cause we need it to maintain diversity in our test cases to avoid stagnation on a suboptimal plateau in the search space [78]. Choosing which mutation function to detect the most vulnerabilities is both a challenging and empirical task.

If changes made to the input are too conservative, only limited code coverage will be achieved as there may not be enough to trigger new control flows whereas over-aggressive tweaks can destroy much of the input data structure leading to the test cases failing at a premature stage of the execution [86].

webFuzz currently supports five kinds of mutation functions, although the tool can be easily extended to support custom GET or POST parameter mutations. The mutation functions employed are; injection of known XSS payloads, mixing the parameters from other requests (cross-over), insertion of a randomly generated payload, insertion of syntax aware payloads and altering the parameter types. Some parameters may get randomly opted out from the mutation process.

That is useful in cases where certain parameters need to remain unchanged for designated areas of the program to execute. Unlike many fuzzers that employ malicious payload generation via the use of genetic algorithms, guided by an attack grammar [27], webFuzz chooses randomly from a corpus that consists of real-life known XSS payloads. The corpus was created with payloads found scattered across the internet, mainly in open-source repositories [50, 82, 84].

A small sample of XSS payloads contained in the corpus can be viewed in Table 3.1. Such payloads can further mutate by prepending or appending to them random strings or specific HTML, JavaScript and PHP syntax tokens. Generating payloads from scratch using complex algorithms may have zero false positives but, nevertheless, time-consuming.

Although *arrays* in URL strings are not clearly defined in RFCs with their format more framework-specific, some web applications rely on them or are oblivious to their existence. Therefore, an input type altering mutation was added where an input parameter expected to be parsed as a string in the web application transforms into an array or vice versa. Web applications not equipped to process unexpected types of input can be prone to glitches and bugs.

Using evolutionary algorithms in the test case creation process is widely practised in fuzzers to optimize solution searching [78], webFuzz will also mix GET or POST parameters from various favourable requests to generate new inputs. Contrary to how evolutionary algorithms work, crossing over of input is not defined as a necessary step in each new input creation but can happen with a medium probability.

3.3 Detecting Vulnerabilities

webFuzz can detect Reflected and Stored Cross-Site Scripting vulnerabilities, and subsequently, web applications that can be exploited for Distributed Denial of Service (DDoS) attacks. To detect such vulnerabilities we conducted a string-matching process for the injected, possibly malicious, payload in the returned HTML response. This method is more efficient in terms of speed, however, it can result in a high ratio of false positives, as the location of the payload in the response is unaccounted for. False positives did arise when the tool reported that an XSS was detected when in fact there was none. One example was when the XSS payload returned enclosed with double quotes inside an HTML element's attribute. If the web application correctly escapes any double-quotes found in the XSS payload it will no be executable. The plans to improve the efficiency of our XSS detection method are discussed in Chapter 6.

	Corpus of known Cross-Site Scripting payloads
1	<form onsubmit=alert(1)><input type=submit>
2	test
3	<abbr id=x tabindex=1 onbeforedeactivate=alert(1)></abbr><input autofocus>
4	<body onscroll=alert(1)>
5	<canvas onbeforepaste="alert(1)" contenteditable>test</canvas>
6	<nav onmouseover="alert(1)">test</nav>
7	<style onreadystatechange=alert(1)></style>
...	**.....**

Table 3.1: *Randomly selected XSS payloads from the corpus webFuzz uses during a fuzzing session. The corpus consists of thousands of payloads*

Chapter 4

Implementation

Contents

This chapter is dedicated to the technical aspects involved while also exploring some of the key characteristics that constitute webFuzz. In-depth, we look at the coding standards used when developing this fuzzing technique, exploiting Asynchronous I/O to achieve concurrency in Python, the parsing procedure of a response and a user-friendly interface displaying statistics. Additionally, we refer to other functionalities of webFuzz accompanied by useful information on how to operate our fuzzing tool.

4.1 Coding Standards

Guido van Rossum (creator of the Python programming language) said; "Code is read much more often than it is written". For this reason, throughout this thesis, we aimed to write clean, readable and eye-catching code by following best practice that leading professional tools adhere to. In so doing, we applied the latest conventions, as recommended

by the Python community to enforce maintainability, clarity, consistency, and generally, a foundation for best programming habits and standards.

More specifically, our fuzzing tool is entirely written in Python 3.8 using the PEP 8 [71] coding style standard and, regarding documentation, the PEP 257 [70] and Sphinx [80] docstring conventions were adopted for it to be clear and easy to read for programmers. Pylint [65] was also used to check for errors in Python code and to implement the aforementioned coding standards and search for *code smells*.

To enhance best practise, unit tests were created through which individual modules of the tool's source code were put under different tests to determine a particular unit's correctness and whether it is fit for purpose. More precisely, parts of the application's code are validated by using test cases that stress-test the tool and ascertain the quality of the code by checking it against the expected response. For this part, popular python test frameworks were used such as pytest [66], unittest [67] and mock [68].

In Listing 4.1, an example of unit testing for the parser module can be found. It tests the functionality of `set_default_hostname()` in the aforementioned module, to check if it is operating as intended. This function's purpose is to transform any URL to its absolute path so that various kinds of URLs are given with their correct absolute value to check if all tests return a `true` value.

```
1  . . .
2  @pytest.mark.parametrize('url_test, url_correct',
3                           [
4                               ("/action/logout.php", "http://localhost/action/logout.php"),
5                               ("", "http://localhost"),
6                               ("/action", "http://localhost/action")
7                           ]
8                           )
9  def test_set_default_hostname(parser_setup, url_test, url_correct):
10     test = parser.set_default_hostname(urlparse(parser_setup.node.url), urlparse(url_test))
11     assert urlunparse(test) == url_correct
```

Listing 4.1: *Unit test for method set_default_hostname()*

4.2 Asynchronous I/O

webFuzz utilises concurrent programming (see Section 2) with the help of the `asyncio` [73] Python module. In our case, `asyncio` has made it possible to send, continuously, HTTP requests to the target website while at the same time statistics on the fuzzing session are printed on the user's screen and a respective log file is updated. With assistance from the module, potential speed-bumps that we might otherwise encounter; such as logging request information to a file or waiting idly for a response for each request, were overcome, since any I/O operation caused by a *blocking* function does not forbid others from running. Conversely, it allows other functionalities to run from the time that it starts until the time that it returns.

Multiple asynchronous tasks (also known as routines) cooperate to let each one take turns running using the `await` keyword, to yield optimal performance. This keyword enables tasks to pause while they wait for their results and let other tasks run in the meantime. This process is called *cooperative multitasking* and although it involves doing extra work up front, the benefit is that you always know when your task will be swapped out, thus optimising to yield better performance.

To summarise, the concept of `asyncio` is that a single-threaded Python object, called the event loop, controls how and when each task is run. Each task can either be in a ready state, which means that the task has work to do and is ready to be run while the waiting state means the task is waiting for some external thing to finish, such as a network operation. The event loop is aware of each task and knows what state it is in and maintains two lists of tasks, one for each of these states.

It selects one of the ready tasks and then returns it back to running. That task is in complete control until it cooperatively hands the control back to the event loop, which in turn places that task into either the ready or waiting list and chooses another task to run. It is important to note, that the tasks never give up control without intentionally doing so using `await`, hence, they are never interrupted in the middle of an operation. A detailed depiction of the asynchronous process executed by `asyncio` can be viewed in Figure 4.1.

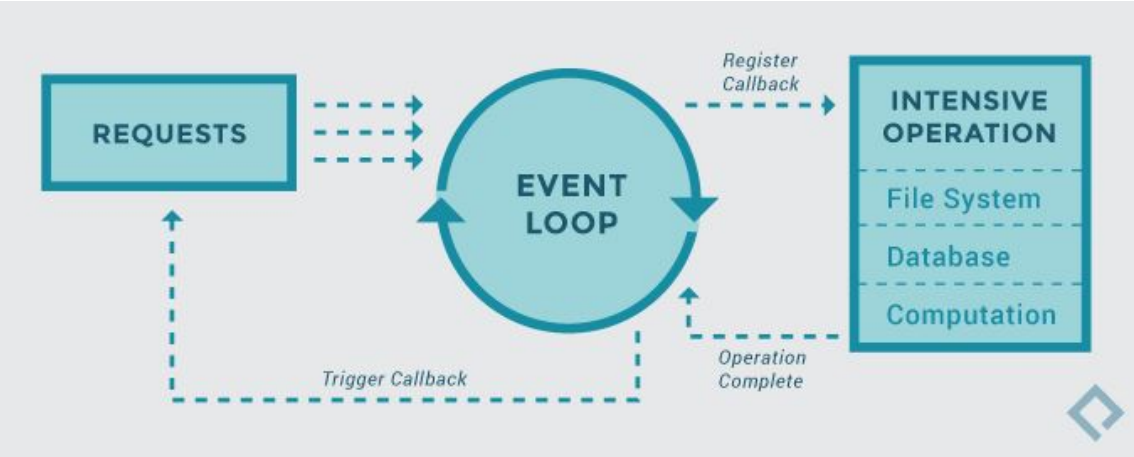


Figure 4.1: *AsyncIO mechanism; it provides a high-performance asynchronous frameworks for making our fuzzing requests* [28]

Communication with the target's website is achieved with a rapidly fast asynchronous HTTP client/server framework named `aiohttp` [2]. The `aiohttp` module creates a reusable Session object per web application through which all requests are performed. Since our fuzzer works with one web application per execution, a single session is created, shared across all tasks, and reused for the entire execution of the program. The re-usability of the session is feasible because all tasks are running on the same thread. Pairing `aiohttp` with `asyncio` evidently speeds things up.

It is important to note here that not all available Python modules are compatible with `asyncio`. For our requests, we could not use the default and recommended Python `requests` package, since it is built on top of `urllib3`, which uses Python's `http` and `socket` modules. Socket operations are *blocking* and not `awaitable` which signals that Python will not like the `await` statement. It is recommended to avoid modules that are incompatible with `asyncio` as the asynchronous framework will not operate at full capacity. However, more modules are becoming compatible with `asyncio` [2].

An example of an asynchronous method is at Listing 4.2. The `run_simple()` method, is one of the most important in our toolbox as it constitutes the entry point to our fuzzer's execution. This provides a simple print interface and not the heavy interface implemented with `curses` shown later in this chapter. It specifies asynchronous tasks to run concurrently.

```
. . .
async def run_simple( self ,  printToFile :  bool)  -> None:
            self .  register_signal_handlers  ()
            fuzzer  =  self
            if  printToFile :
                f  = open("/tmp/ fuzzer_stats ",  "w+")
                def  printer ( line ):
                    f . write ( line  + "\n")
                def  refresh ():
                    f . truncate (0)
                    f . seek (0)
                sm = Simple_menu(fuzzer,  printer ,  refresh ,  f . flush )
            else :
                sm = Simple_menu(fuzzer)
             print_stats_task  = asyncio . create_task (sm. print_stats  ())
            fuzzer_loop_task  = asyncio . create_task ( self . fuzzer_loop ())
            await  print_stats_task
            exit_code  = await  fuzzer_loop_task
            self . _grace_exit ( exit_code )
. . .
```

Listing 4.2: *Starting point for the fuzzer's execution using a simple print interface*

4.3 Parser

The fuzzer's parsing module is responsible for extracting vital information during the fuzzing process from each response received, after, of course, the respective request is made. Each response contains the HTML document which is parsed using the `Beautiful Soup` [14] module to extract the `form` and `anchor` elements from it. These elements are useful as they can provide us with new URLs which translate into potentially new code paths and bugs to further explore and locate.

When new URLs are found, they are added to the *crawler*'s pending request list, if they are interesting they will be fuzzed in the future. At this stage, the HTML document is also

checked for XSS vulnerabilities. The metadata stored for each request can tell us which XSS payloads were injected into it and led to the vulnerability. If they happen to reside in the HTML document, which signals an RXSS vulnerability, a warning is triggered, incrementing the total number of XSS found and logging the related information. The document is checked for Stored XSS vulnerabilities by scanning the document for all the XSS payloads that were injected in all the requests. A high-level pseudocode for the parsing process can be seen in Algorithm 1.

As the pseudocode shows clearly, parsing relies heavily on the `urllib.parse` [69] Python module. To be exact, the urlparse method is used for breaking the Uniform Resource Locator (URL) string up into components; such as the addressing scheme, network location, path *etc.* An object is returned that contains a 6-item tuple with all the URL subfields. The reverse can also be achieved through the urlunparse method; a URL object can be converted into a string.

4.4 Curses Interface

A Textual User Interface (TUI) for webFuzz was created using the `curses` module containing information and essential statistics, gathered while our grey-box fuzzer is running. The `curses` library supplies a terminal-independent screen-painting and keyboard-handling facility for text-based terminals [19], such as the Linux console. The text editor *nano* is a good example of a `curses` application.

Unfortunately, this functionality is not available for Windows, as the Windows version of Python does not include the `curses` module. By running our fuzzing tool on a Windows-based machine, regardless of the Command Line Interface (CLI), you opt to use, it will result in a crash.

There are of course ways to run webFuzz without this interface which will be explained in the next section. Although many may think this is obsolete technology, it is valuable for Unix-based operating systems that do not provide any graphical support. The Python module, which is the one we utilised, is a fairly simple wrapper over the *C* functions

Algorithm 1 *Parsing HTML documents pseudocode*

lookForXSS(HTML) {Increments global XSS counter if one is found.}

$links \leftarrow set()$

for every form found in the HTML document **do**

if form does **not** contain an action field **then**

$urlObject \leftarrow urllib.parse(callingNodeUrl)$

else

$urlObject \leftarrow urllib.parse(relativeToAbsolute(form.action))$

end if

$parameters \leftarrow parseQueryString(urlObject.query)$

$urlString \leftarrow urllib.unparse(urlObject)$

$inputs \leftarrow dictionary()$

for every < input > element found in form **do**

$value \leftarrow input.get(value)$

$name \leftarrow input.get(name)$

$inputs[name] \leftarrow append(value)$

end for

$method \leftarrow form.get(method)$

$Node \leftarrow createNode(parameters, urlString, inputs, method)$

$links \leftarrow add(Node)$

for every < a > element found in form **do**

$anchor \leftarrow a.get(href)$

end for

$Node \leftarrow createNode(parameters, urlString, inputs, method)$

$links \leftarrow add(Node)$

end for

return links

provided by the first and original `curses`.

A snapshot of the webFuzz interface is seen in Figure 4.2. As illustrated, the statistics are divided into three categories; namely the process statistics, the overall progress and the examining node details. As the fuzzing tool expands more valuable information is added to the interface.

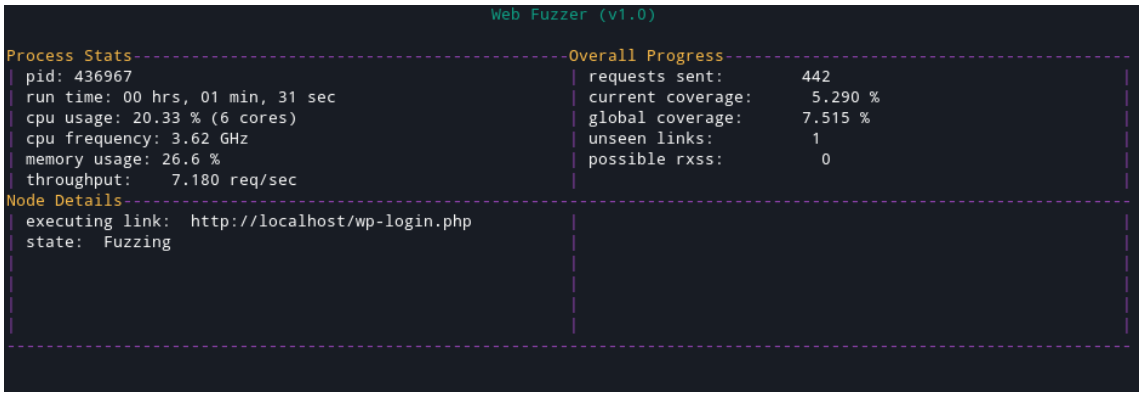


Figure 4.2: *Interface of webFuzz is implemented using the Curses module*

4.5 Interactive and Black-Box Functionalities

Our fuzzing tool also provides manual functionality. It allows the user to engage through an interactive session with our fuzzer. After the user provides the target to be fuzzed, a connection is established and the session begins. Before a request is made, forms are extracted from the target and all input fields are presented to the user. The user can choose from a menu of options on which fields to fuzz and how. More specifically, the options consist of filling the fields with manually inserted data, XSS payloads from the corpus seen in Table 3.1 or choosing to mutate data using a basic mutating function.

Mutating functions for a given input provided include deleting random characters, inserting characters at random places and flipping random characters. The user must insert manually any different link desired to be fuzzed, as no crawling process is provided. At Listing 4.3, is a code snippet of the interactive mode.

The user can switch at any time to a more automated, brute-forcing type session where input fields of the forms at the given fuzz target are filled with XSS payloads with no

need of interruption from the user. This brings us to the black-box fuzzing capabilities also provided by our fuzzing tool. When choosing to fuzz in a black-box fashion, everything that we have mentioned still applies, as the core functionalities are shared between the two. The only aspect not taken into account with this approach, is, of course, the *instrumentation*.

This thesis only aims to demonstrate the capabilities of webFuzz as a grey-box fuzzer, therefore no rigorous testing was conducted to prove its efficacy. Thus, no further discussion is made on its interactive and black-box functionalities.

```
1  def fuzz():
2      . . .
3      print("Please choose one of the following fuzzing methods:")
4      print("(1) Select a random XSS payload from corpus")
5      print("(2) Mutate previous inputs")
6      print("(3) Manually insert data")
7      print("(4) Switch to Black-box fuzzing")
8      print("(5) Enter new link to fuzz")
9      ans = input()
10
11     if ans == "1":
12         data[str(i)] = str(xss_fuzzing_payload())
13     elif ans == "2":
14         choose_mutating_function(data[str(i)])
15     elif ans == "3":
16         value = input("Insert data for {}: ".format(i))
17         data[str(i)] = value
18     elif ans == "4":
19         black_box_fuzzing()
20     elif ans == "5":
21         link = input("Insert new link")
22     . . .
```

Listing 4.3: *Options menu and their processing during interactive mode fuzzing*

4.6 Running webFuzz

Running webFuzz is straightforward. The necessary modules in their exact version are listed to be installed for webFuzz to operate smoothly. These are listed in a "requirements" text file which is in Appendix B. Other executing and installing dependency instructions are at the README.md file in the tool's repository.

A help menu that shows all available arguments in which webFuzz can run in are shown in Figure 4.3. As depicted, arguments are separated into three categories; namely *Optional*, *Required* and *Positional*. *Optional* arguments are extra functionalities that you do not have to include when running the tool, whereas *Required* and *Positional* are the arguments that must be included. For the creation of the usage menu and parsing the arguments, the `argparse` [64] Python module was used.

Also, throughout the execution, `logging` is used for tracking events that happen when the fuzzer runs. `Logging` is a module in the Python standard library that provides a richly-formatted log.

```
marcos@marcos-virtual-machine:~/PycharmProjects/hhvm-fuzzing/web_fuzzer$ ./webFuzz_runner.py -h
usage: webFuzz_runner.py [options] -r/--run <mode> <URL>

webFuzz is a grey-box fuzzer for web applications.

Optional Arguments:
  -h, --help            show this help message and exit
  -v, --verbose         Increase verbosity
  -s, --session         Login through the browser and get cookies
  --ignore_404          Do not fuzz links that return 404 code
  --ignore_4xx          Do not fuzz links that return 4xx code
  -m META, --meta META  Specify the location of instrumentation meta file (instr.meta)
  -b BLOCK, --block BLOCK
                        Specify a link to block the fuzzer from using, Form = 'url|parameter|value'
  -w WORKER, --worker WORKER
                        Specify the number of workers to spawn that will concurrently send requests
  --anchor_unique       Treat urls with different anchors as different urls
  --driver DRIVER       Specify the location of the web driver (used in -s flag)
  -t TIMEOUT, --timeout TIMEOUT
                        Set timeout value in seconds
  --version             Prints webFuzz latest version

Required Arguments:
  -r RUN, --run RUN     Choose mode in which you want the fuzzer to run. Select one of the following: auto, manual, simple, file

Positional Arguments:
  URL                   Specify a URL to fuzz
```

Figure 4.3: *webFuzz usage menu includes all available arguments needed to run it*

Chapter 5

Evaluation

Contents

This chapter examines the evaluation of webFuzz against other black-box vulnerability scanners. We begin by describing in detail the methodology used to evaluate our tool and explain the *automated bug injection* process on the fuzz targets. Then, we discuss the details of the evaluation such as the metrics to be deployed. Finally, we proceed in reviewing the results of each metric used.

5.1 Methodology

In the evaluation of our tool, for convenience, we opted to use Docker [20], which was discussed in Chapter 2. Docker is software that can package your application, its dependencies, system tools, system libraries and settings in a single comprehensive virtual

container. This is because Docker is lightweight, portable and can considerably improve application development and deployment.

As mentioned in Chapter 3, webFuzz is limited to web applications written in PHP. The most popular and widely deployed language for Web applications is undoubtedly PHP, powering more than 80% of the top ten million websites and contributing to almost 140,000 open-source projects on GitHub [88]. Also, in the server-side technologies category, PHP – the most prevalent server-side language – was associated with the highest number of vulnerabilities in 2019 [16]. For these reasons, we decided to evaluate our tool on web-apps developed in PHP.

The first web application we tested our tool on was WordPress. The WordPress CMS (Content Management System) [20] is among the most popular open-source web application for managing and publishing content on the web with nearly half of the top 1 million sites on the internet using it [10]. While WordPress powers more than a third of the web, what was more important for us, is that it is written in PHP and widely used for building a variety of websites, ranging from simple blog spots to professional news sites. This means the results of our experiment represent a wider cross-section of websites. WordPress was not only the most popular platform but also dominated the number of new vulnerabilities in 2019 [16]. Unsurprisingly, 97.2% of WordPress vulnerabilities were related to plugins. The most common WordPress vulnerability by far was XSS with 44.6%.

We tested our tool on a second web application, Drupal CMS [26]. Drupal is a free and open-source content-management framework written in PHP and distributed under the GNU General Public License. It is used as a backend framework for at least 2.1% of all Web sites worldwide ranging from personal blogs to corporate, political, and government sites. To generate additional evidence-based data another two web applications were used - Firefly-III, Mautic - for the code coverage experiment (Table 5.2).

Using Docker and its docker-compose functionality, we were able to achieve a multi-container deployment through a single docker-compose YAML file for the following services:

- *NGINX*: An open-source, high-performance HTTP server that handles all the HTTP request made by webFuzz and forwarded to our running web applications [56].
- *WordPress, Drupal, Firefly-III and Mautic*: Open-source CMS web applications. Having access to their code, we began examining the existing systems in terms of injecting bugs and performing our *instrumentation*.
- *MariaDB*: A popular open-source relational databases we used to store and manipulate the WordPress data [45].

The official images for the above services are free at Docker Hub. An illustration of the above infrastructure for WordPress is in Figure 5.1. Files and instructions for replicating this process are in the fuzzer's repository. The respective docker-compose is in Appendix A.

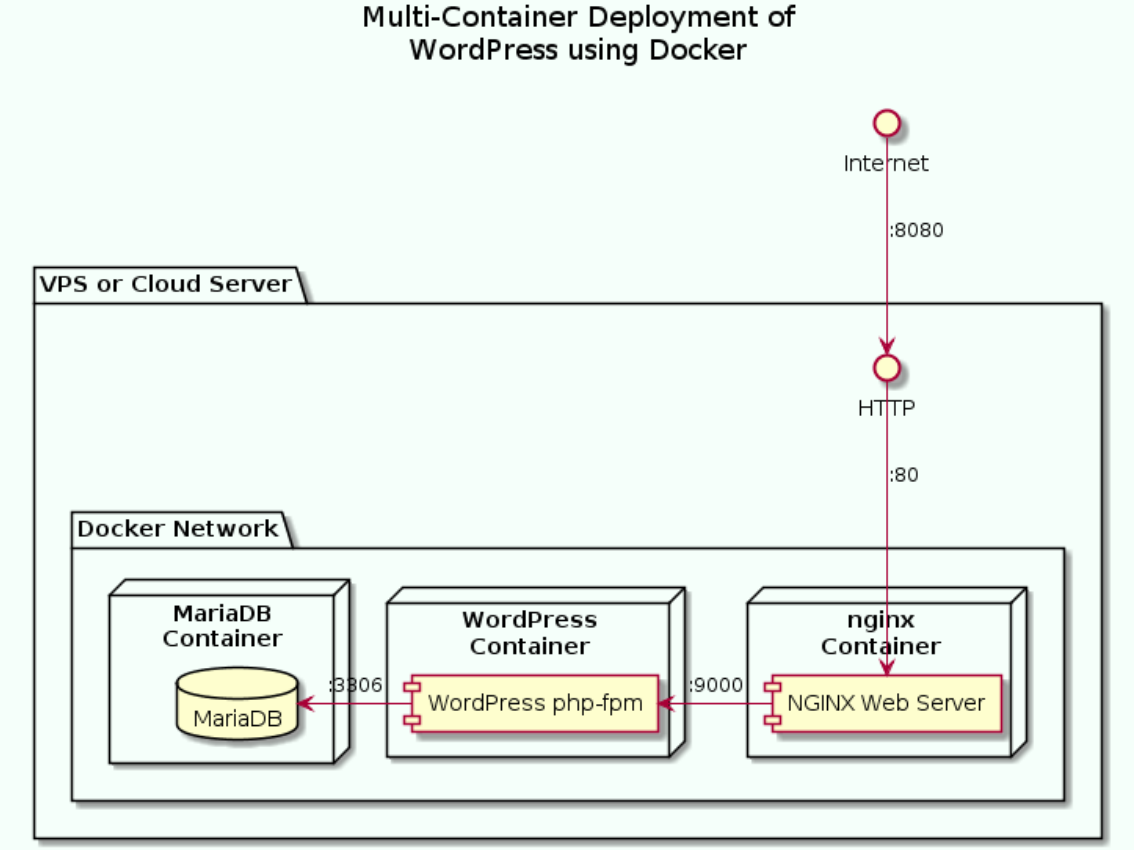


Figure 5.1: *Evaluation followed multi-container deployment of WordPress using Docker*

5.2 Automated Vulnerability Addition

Evaluating fuzzing processes proved to be a challenging task [43]. Migrating known vulnerabilities to existing software, to test the fuzzer's capabilities in detecting bugs can be a tedious process [54]. For evaluating webFuzz, and other fuzzers for web applications, an *automated bug injection* tool named `Centaur`, inspired by LAVA [23] was used for automatically injecting bugs in web applications written in PHP.

Injecting vulnerabilities in web code was a demanding task, since the necessary tools for analysing *native code* and injecting vulnerabilities (*e.g.,* taint-tracking and information-flow frameworks), are not available for web applications.

To overcome the lack of available tools, `Centaur` uses vulnerability injection methodology to leverage the *instrumentation* infrastructure. The automated bug-injection method can inject hundreds of common vulnerabilities such as Reflected Cross-Site Scripting in reasonable time.

5.3 Evaluation Details

To evaluate webFuzz's performance we used two Ubuntu 18.04 LTS Linux machines both possessing a 3.20 GHz quad-core Intel® Xeon® W-2104 Processor and 64 GB of RAM. Targeted web applications consist of (a) an instrumented WordPress 5.5.1 with artificial bugs, (b) a vanilla WordPress 5.5.1 with artificial bugs, (c) an instrumented Drupal 9.0.6 (d) an instrumented Firefly-III 5.4.6, and (e) an instrumented Mautic 3.0. The term vanilla refers to web-apps in their original form, with no customization or frameworks added to them.

All artificial bugs were created with the automated vulnerability injection tool mentioned in Section 5.2. Using this methodology, we managed to inject 150 identical Reflected Cross-Site Scripting bugs successfully in both the instrumented and vanilla versions of WordPress. Lastly, the Docker stack of services described in Section 5.1 was deployed to run the aforementioned web applications.

For evaluating the performance of webFuzz, the following metrics were used in order of importance:

- *Vulnerabilities Detected*: Number of Reflected Cross-Site Scripting bugs reported
- *Global Code Coverage*: Accumulated coverage score of the web application's code
- *Throughput*: Requests made per second

To compare the Vulnerabilities Detected and the Throughput of webFuzz against other black-box fuzzers we used Wfuzz [49], Burp Suite Professional [62] and OWASP ZAP [59]. All three of these tools are considered essential in any penetration tester's arsenal as they are included by default in Kali Linux and widely used in Capture The Flag competitions such as GoogleCTF. Other tools; such as `nikto`, `w3af`, `skipfish` and `wapiti` were also used during the evaluation phase but as they were not able to uncover any real or artificially injected bugs we opted not to include them in our final evaluation. The main comparison was made against Wfuzz due to the ease of operation and to extend. The choice of Wfuzz as the main comparison of our tool is further elaborated on in Chapter 6.

5.4 Evaluated Metrics

5.4.1 Vulnerabilities Detected

To evaluate how well webFuzz performs in terms of bug detection, we injected 150 artificial Reflected Cross-site bugs with the methodology we discussed in Section 5.2 and 4 real Reflected Cross-site Scripting bugs to the instrumented version of WordPress and tested how 3 well-known black-box fuzzers performed in comparison with webFuzz. The real-life RXSS bugs were found from CVE [15] and have the following ids: CVE-2018-7280, CVE-2019-11843, CVE-2020-7104, CVE-2020-7107.

When analysing the specifics of the four real-life RXSS vulnerabilities manually injected, it was realised that CVE-2019-11843 depends on JavaScript code to create its triggering

Vulnerability Detection				
Tool	**Version**	**Real Bugs**	**Artificial Bugs**	**Runtime**
webFuzz	1.0.0	1	30	65h
Wfuzz	2.4.5	1	28	65h
Burp Suite Professional	2020.9.2	1	0	7h
OWASP ZAP	2.9.0	1	0	1h

Table 5.1: *Summary of the vulnerability detection evaluation with the findings of 4 fuzzers including webFuzz. The WordPress web-app included 4 RXSS bugs found from CVE and 150 artificial RXSS bugs injected manually*

link dynamically in the form of an `anchor` element. Also, for the vulnerability CVE-2018-7280 to trigger it is compulsory for the XSS payload to be injected inside a specific JSON object at one of the vulnerable `form`'s parameters. The other three depend on JavaScript code to dynamically append the vulnerable POST parameter upon `form` submission.

The real RXSS bug that all four tools were able to detect was CVE-2020-7107. This bug was related to the Ultimate FAQ plugin for WordPress. More specifically, the HTML code generated by the FAQ shortcode (WordPress-specific code that simplifies complex commands) did not sanitise the Display-FAQ GET parameter, leading to the unauthenticated RXSS issue on pages where such shortcode is used. This vulnerability was fixed in a later version (1.8.30) of the plugin by sanitizing the GET parameter with the `intval()` function.

As Table 5.1 shows, all tools involved in the evaluation only managed to find one real-life RXSS bug. This is because none of them employs complex enough JavaScript code analysis nor do they run a request's client-side code to uncover these dynamic links and parameters.

It is important to note that vulnerability scanners such as Burp Suite Professional provide a Proxy service that can intercept web browsing traffic so that requests created dynamically by client-side code can be fuzzed as well. Nevertheless, we chose to avoid these features since webFuzz currently lacks this functionality and thus, it would be an unfair

comparison.

As clearly observed in Table 5.1, the results of all fuzzers for real-life bugs are disappointing. For this reason, we further evaluated the fuzzing tools on how well they perform with artificially injected bugs (Section 5.2). Because OWASP ZAP and Burp Suite Professional do not generate the required format of injection payloads unless advanced features and modifications are in place, they were unable to detect any of the artificially injected bugs. Using their advanced features requires extensive research and training. As the learning curve is steep, we decided to only customise Wfuzz.

For a fair comparison of the vulnerability detecting capabilities of webFuzz and Wfuzz, some modifications were made to the latter since originally the tool was meant to be a simple brute-forcer and not a black-box fuzzer. An independent crawling process was added at the start to infer the control flow of the web application.

The findings are stored and fed to Wfuzz as a list of fuzz targets. Utilising the Python module version of Wfuzz, a Python script was created that fuzzes the list of links found by the *crawler*, indefinitely. Payloads used during the fuzzing process are customised to resemble the mutated payloads that webFuzz uses. They consist of random strings, HTML syntax tokens, and random numbers all concatenated with the same XSS payloads that webFuzz uses from its corpus described in Chapter 3.

The results of our 65-hour experiment, comparing Wfuzz and webFuzz in terms of artificial RXSS bugs found is seen in Figure 5.2. Although webFuzz leads throughout the entire experiment, the difference decreased until the end when it became marginal. webFuzz uncovered 30 artificial bugs, two more than the Wfuzz's 28.

By taking advantage of the *instrumentation feedback loop*, webFuzz detected the artificial bugs faster than Wfuzz's brute force approach. Whenever a digit of a *magic number*, situated in a vulnerable payload, is guessed correctly, our fuzzing tool will detect this change and prioritize the request that causes it.

With this method, finding a *magic number* is done incrementally - one correct digit at a time - which is much faster than guessing the whole number at once like Wfuzz does. As

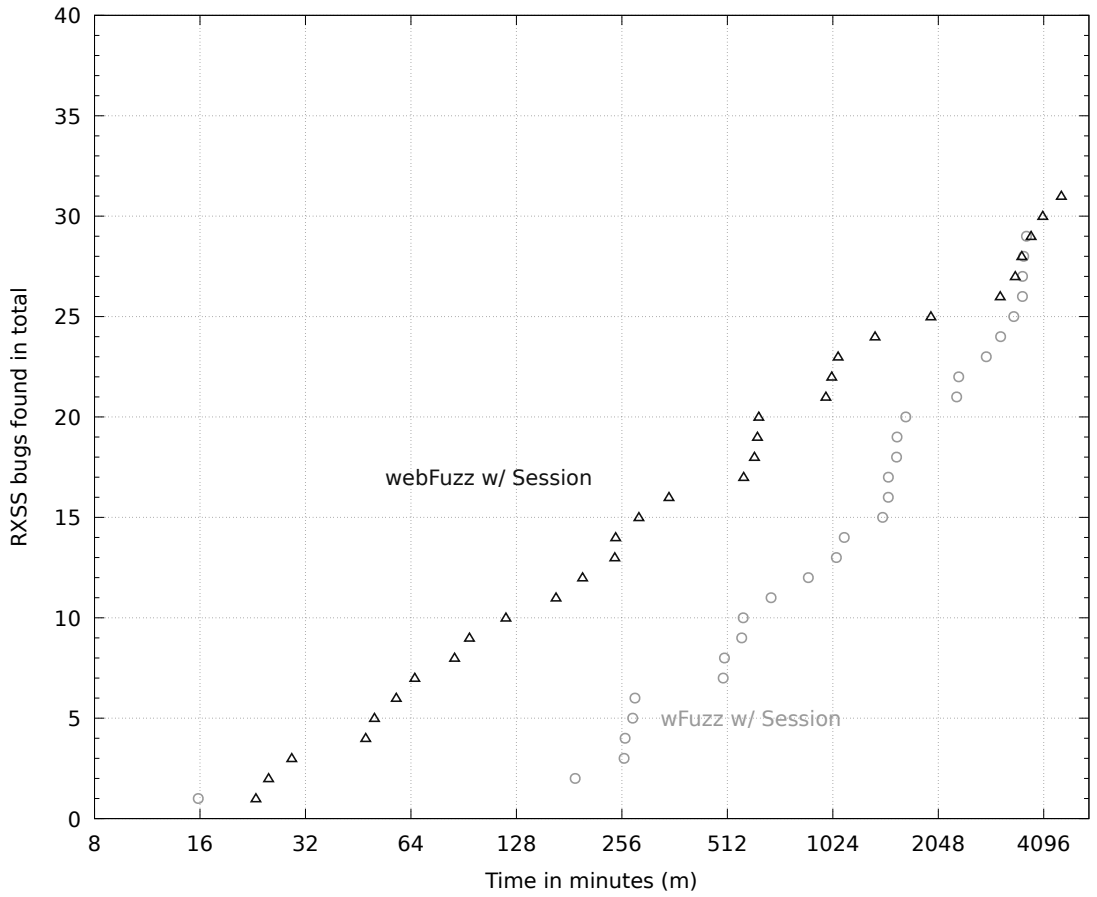


Figure 5.2: *Artificial Reflected Cross-Site Scripting bugs detected over time by webFuzz and Wfuzz. webFuzz manages to uncover more bugs quicker during the fuzzing process*

a real-world analogy, each digit of the *magic number* can represent one correct mutation that brings us closer to the vulnerable basic block. A reason for the gradual decrease of webFuzz's detection performance lies in its growing request queue size. WordPress is composed of approximately half a million LoC, with 48,040 basic blocks instrumented in total.

5.4.2 Global Code Coverage

Utilizing the *instrumentation* feedback, webFuzz has calculated the global code coverage for WordPress, Drupal, Firefly-III and Mautic. For these four PHP open-sourced projects, we decided to test only the authenticated session scenarios as the unauthenticated session would prevent us from accessing various links such as the administrative dashboard

related links. In Figure 5.3, we see how the metric changed over time in the four authenticated session scenarios.

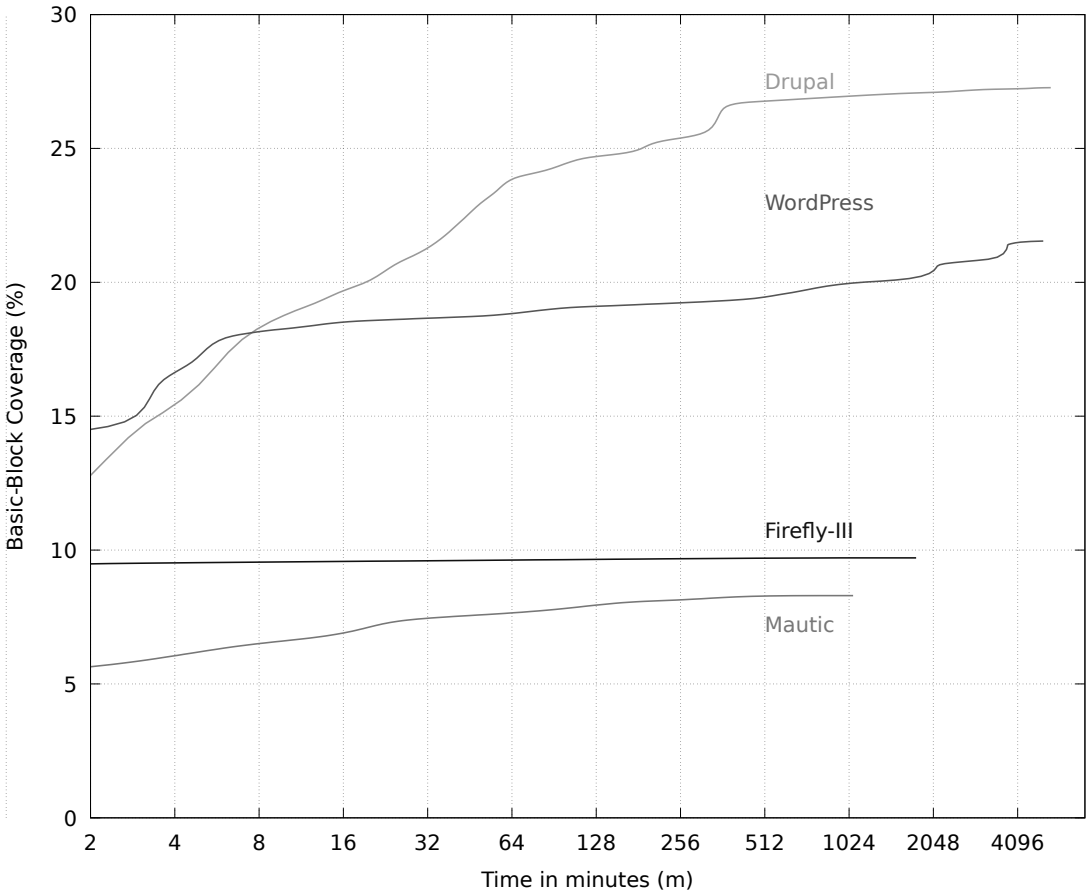


Figure 5.3: *Four different execution scenarios and the accumulated code coverage gained over time. Exponential increases at the start of the experiment are due to initial exploration of the target web application's site map by the crawler*

Our experiment showed the two main web applications of our evaluation, Drupal and WordPress accrued the most basic-block coverage with *27.3%* and *21.5%* respectively. Firefly-III was stopped after 1.3 days as the coverage remained static at *9.7%* during the entire time. In the case of the fourth fuzz target Mautic, we agreed to terminate this test early (0.7 days) as the throughput was arduously slow in reaching the desired level of code coverage within a reasonable time limit (see Table 5.2).

More importantly, the code coverage achieved by webFuzz in both the Drupal and WordPress scenarios indicate a steady rise in the global code coverage even after 6000 minutes of execution time. An encouraging signal that the mutation functions used are effective

Code coverage achieved with webFuzz		
Fuzz target	**Run time (minutes / days)**	**Code coverage (%)**
Drupal	6000 / 4.2	27.3
WordPress	6000 / 4.2	21.5
Firefly-III	1900 / 1.3	9.7
Mautic	1024 / 0.7	8.3

Table 5.2: *Code coverage achieved by webFuzz when fuzzing four open-source web applications*

enough to trigger new code paths, even after the crawling process has finished.

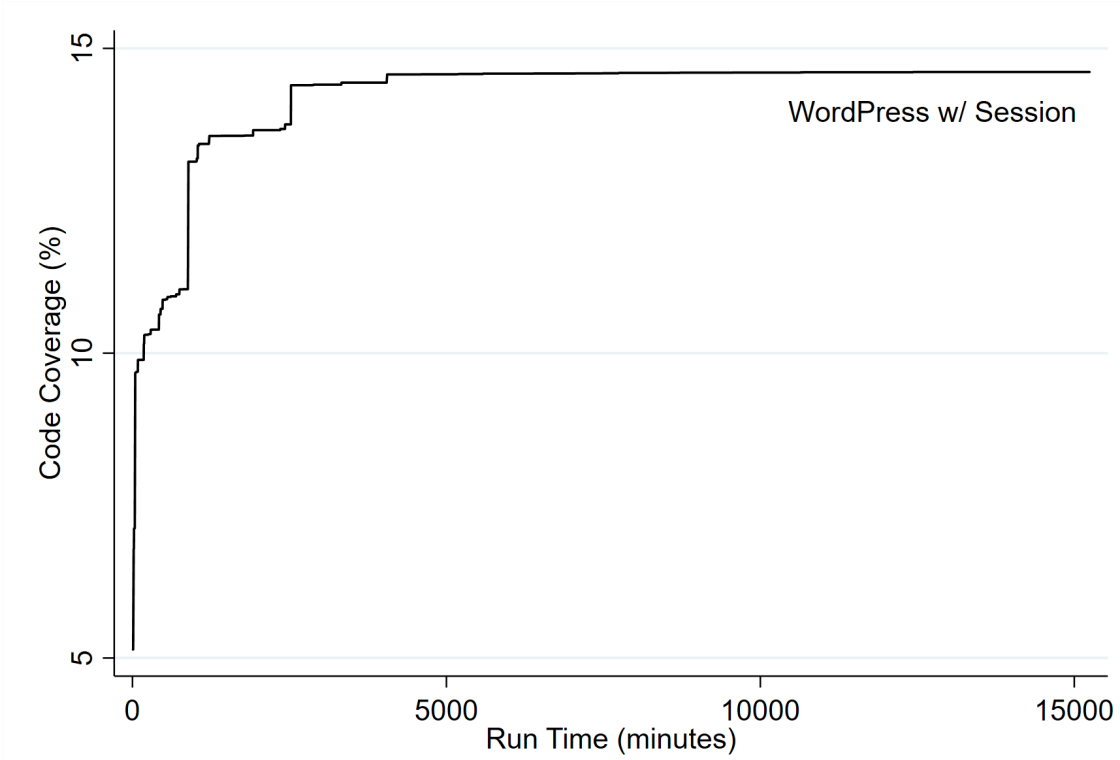


Figure 5.4: *Code coverage achieved by Wfuzz over time when fuzzing WordPress using an authenticated session. After running for approximately 5000 minutes (3.5 days) the code coverage remained stagnant at 14.6% for the rest of the experiment*

In Figure 5.4 we can see the Global Code Coverage achieved by the black-box fuzzer Wfuzz when fuzzing WordPress using an authenticated session. The peak of this experiment was reached in roughly *3.5* days (5000 minutes) when code coverage of *14.613%* was reached. This experiment was solely to check how well a black-box fuzzer performs

in terms of code coverage against a grey-box fuzzer, like webFuzz, that leverages *instrumentation* feedback.

When looking at Figures 5.3 and 5.4, we can safely surmise that the *instrumentation* feedback provides webFuzz with the edge needed to surpass the performance of Wfuzz. More precisely, when both were fuzzing WordPress with an authenticated session, webFuzz managed to get almost *1.5* times higher code coverage than Wfuzz. Wfuzz was left running for a much longer time (10.4 days) than webFuzz with no luck since it stayed stagnant on the score it achieved after 3.5 days at *14.613%*.

5.4.3 Throughput

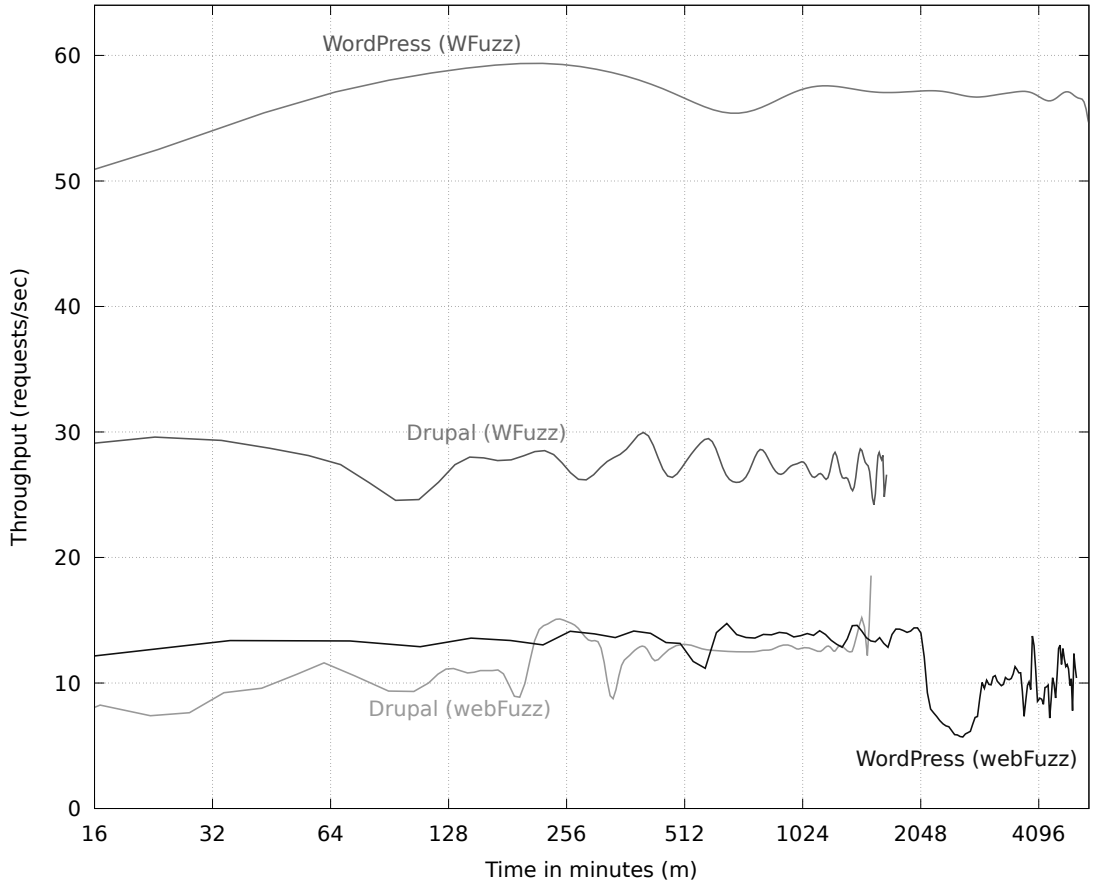


Figure 5.5: *Requests made per second over time for three different scenarios. Both webFuzz and Wfuzz are evaluated on Drupal and WordPress. Wfuzz takes the lead with a difference*

One reason for the effectiveness of fuzzers in uncovering vulnerabilities is their capability

to test vast amounts of inputs per second. The overhead caused by *instrumentation* must not severely degrade the web application's response time and the fuzzer's processing time for each request is kept as brief as possible.

As observed in Figure 5.5, the black-box version of Wfuzz has about *3* times higher throughput than webFuzz in the case of Drupal and *4.5* times in WordPress. This is plausible as the overhead added from *instrumentation* roughly doubles the page response time for WordPress, and due to webFuzz's increased statefulness in tracking, analysing and ranking all the requests, it increases the per-request processing time.

After 2,048 minutes in WordPress using webFuzz with an authenticated session established, the throughput plummets for a lengthy period. This implies that the fuzzer was stuck on fuzzing particular links that have lofty response times. The fact our fuzzer keeps on fuzing these links mean they have a high coverage score and mutating them is effective enough to trigger new code paths.

Looking at Figure 5.5, we can state with confidence that much needs to be done to improve the throughput of webFuzz, since it is nowhere near that of native application fuzzers such as AFL and EFS nor is it comparable to black-box web fuzzers such as Wfuzz. That was no more than expected as native applications have no need to address overhead from sending requests over a network. Needless to say, the necessary improvements are discussed in the next chapter.

Chapter 6

Discussion

Contents

In the discussion chapter, the limitations faced - *i.e.,* a lack of fuzzers to compare with - during the development of webFuzz are outlined in detail. There are also deliberations on future plans and what is under consideration for upgrading our fuzzing tool.

6.1 Limitations

During the development of webFuzz, we faced various obstacles that must be addressed to produce a more productive tool.

Our first major obstacle was the choice of Wfuzz as the main fuzzer to compare with webFuzz. After extensive research, it became apparent there are fewer black-box fuzzers available today than there were a decade ago. Many older, renowned black-box fuzzers cited in various websites and published papers [7, 25, 27] have either ceased to exist or are no longer developed and maintained.

During our research we discovered that Wfuzz is the only tool that can be imported as a module in Python, thereby extending its functionality. Although Wfuzz classifies as a 'brute-forcer', by providing this functionality we can add code to make it operate as a

black-box fuzzer. This enabled us to make a reasonable comparison with our fuzzing tool. Wfuzz was easier to use as it did not require time-consuming research and extra training. This could not be achieved with Burp Suite Professional or OWASP ZAP.

Additionally, during the evaluation phase, more evidence can be submitted to further explore the potential of webFuzz in detecting vulnerabilities. For instance, the tool can be evaluated on more open-source projects written in PHP and tackle other complex real-world XSS vulnerabilities that reflect real-world scenarios. A case in point, CVE is a good source of finding publicly-known XSS vulnerabilities. A recent paper by Backes et al. [5] proposes ideas on such large-scale analysis of web application code to find real-world XSS bugs. We will use this as a reference point going forward.

For now, webFuzz's vulnerabilities detection suite is limited to Reflected and Stored Cross-Site Scripting. DOMbased XSS vulnerabilities that rely on the browser's JavaScript runtime context, are beyond the fuzzer's scope. These types of attacks require no interaction with the server, and succeed when the JS code does not sanitize the user input before rendering it unfiltered (*e.g.,* using the `innerHTML` property). For detecting these vulnerabilities, we would need to render the HTML and run the JavaScript code of each request. This would severely degrade the fuzzer's throughput that is why this type of detection was not included for the initial version of webFuzz. Unfortunately, by excluding JavaScript, due to a time deficit, many potential XSS vulnerabilities went undetected.

6.2 Future Work

Our work is not yet done. Despite our initial accomplishments, there is much we need to do to elevate this promising fuzzing tool to a higher level. Undoubtedly, improvements need to be made to ensure it is an effective and trustworthy tool. Below are some ideas on future progress.

There are plans to include more functionalities in our tool kit to weed out other critical web-app vulnerabilities through our detection suite, offering wider security protection beyond Cross-Site Scripting.

Such core vulnerabilities can be found at OWASP Top 10 [60]. The most common form of bugs in web applications is Injection and Broken Authentication. Injection flaws, such as SQL and NoSQL, occur when untrusted data is sent to an interpreter or database as part of a query. For this specific vulnerability, various known payloads have already been collected [50] - the same way as the XSS payloads are - and stored in the repository waiting for the respective functionality to be added to webFuzz.

There are also plans to implement a more efficient string-matching algorithm that will decrease the number of false positives we currently record. This is achieved by taking into consideration the location of the payload in the HTML document. These improvements will enable us to detect Cross-Site Scripting vulnerabilities that are triggered due to HTML attributes such as `onchange` and `onclick`, and not because of the HTML's <script>.

As we mentioned in the limitations, previous research used techniques such as analysis of JavaScript code or Selenium-based *crawlers* to include the JavaScript-generated request URLs in their analysis. We could adopt similar approaches since we are currently missing many bugs by excluding JavaScript. Moreover, to improve our evaluation we may adopt similar approaches that Backes et al. did [5] where they propose a way to build code property graphs for 1,854 popular open-source PHP projects on GitHub, storing them in a graph database and detect vulnerabilities through flow-finding traversals.

One idea of improving our fuzzer is that certain core functions will eventually be ported to faster languages; such as *C* and *Java*, to substantially enhance speed performance and reduce memory consumption. Besides, a per-link time-out will be introduced, to avoid I/O heavy web pages from stalling the fuzzing process. Initial work has also be done with netmap [77], a framework that modifies kernel modules to effectively bypass the Operating System's network stack, which often creates a bottleneck between client and server communication, to achieve a high-speed packet I/O.

Also to be included, are more Python modules to improve the overall performance of webFuzz. Since our fuzzer requires a lot of file I/O to do its logging work, the `mmap` module can be utilised by using lower-level operating system APIs to load a file directly

into the computer memory and read/write files as if they were one large string or array [74].

Another module that could boost the performance of webFuzz is `aiomultiprocess` [1]. As we briefly mentioned in Chapter 2, AsyncIO is limited to the speed of GIL, and multiprocessing entails spreading tasks over a computer's cores. By combining the two, we can overcome these obstacles to truly achieve 'parallelism' in Python. Achieving 'parallelism' would be a beneficial outcome as today's PCs/laptops have processing units with multiple cores.

Having said that, ideas of optimization are one thing, putting them into practice is an entirely different matter. Every step will be properly assessed and examined scientifically before being added to our tool.

"Premature optimization is the root of all evil (or at least most of it) in programming," said Donald Knuth - the father of algorithms analysis.

Chapter 7

Related Work

Contents

In this chapter, all relevant, acclaimed or related academic work achieved in recent years at research level in the field of fuzzing is referenced below. This chapter divides into two sections; generic fuzzing on *native code* and fuzzing web applications. Taking this into consideration, we present our approach compared to what went before.

7.1 Generic Fuzzing

Fuzzing has been perceived through several techniques and algorithms over the years. Firstly, we have the black-box fuzzers [36, 79, 85] which are unaware of the fuzz target's internals and try to trigger vulnerabilities by randomly generating the inputs. While the black-box fuzzers category might not be as performant as others, they offer the advantage of compatibility with any program [57, 72]. The other two categories are white- and grey-box fuzzers. These two leverage *instrumentation* to obtain feedback concerning the inputs' precision in discovering unseen paths.

It has been proven that feedback is vital for a fuzzer's performance since it can be used to steer the fuzzer towards exploring new code paths, resulting in better code coverage also

known as coverage-based fuzzers. Otherwise, we have the directed based fuzzers that use feedback to direct the fuzzer towards particular execution paths [30].

A renowned fuzzer that is classified as coverage-based is AFL [87]. AFL is a state-of-the-art grey-box fuzzer which is the foundation for the majority of recently-proposed research. However, AFL fails to intelligently generate inputs to explore deep paths in programs hidden behind checksums or *magic number if* statements.

Having that in mind, recent research makes use of symbolic and concolic execution to enhance the input generation procedure by extracting valuable information about the program. Some examples consist of DRILLER [81], DART [30] and SAGE [31]. DRILLER is also an example of a hybrid vulnerability searching tool as it combines fuzzing and symbolic execution.

Despite efforts to improve the fuzzing process with the use of symbolic/concolic execution-based fuzzers, these types of fuzzers suffer from scalability problems because when fuzzing sizeable targets, we notice the phenomenon of state/path explosion [11]. This problem is observed when the number of state variables in the system increases, the size of the system state space grows exponentially making it impossible to explore the entire state space with limited resources of time and memory.

While trying to explore every path in the code (*i.e.,* for a conditional branch, they often create an input that causes the branch to be taken and another that does not) they succumb to path explosion, greatly limiting their scalability.

Consequently, other research proposals try to accomplish what symbolic/concolic execution-based fuzzers offer with a less expensive approach. One example is REDQUEEN [13] that utilizes the input-to-state correspondence to infer the values that can later be used to try and control them. Another such example is VUzzer [72], an application-aware evolutionary fuzzer that leverages control and data-flow features using static and dynamics analysis to infer fundamental properties of the fuzz target.

7.2 Web Applications Fuzzing

Even though a huge effort is directed toward building fuzzers with the aim to weed out vulnerabilities in *native code*, little attention has been given to web application bugs. Tools currently available that target web application vulnerabilities behave predominantly in a black-box fashion, therefore, they are unable to uncover vulnerabilities that are embedded deep into a web application [7, 25].

Such as SecuBat [40], a web vulnerability scanner that uses a black-box approach to detect SQL injection(SQLi) and Cross-Site Scripting(XSS) vulnerabilities. Another example is KameleonFuzz [27], a black-box fuzzer for web vulnerabilities targeting XSS susceptibilities.

There have been attempts to overcome the shortcomings of black-box techniques. Doupé et al. [24] proposed a way to navigate through a web application's states to discover whether an input is interesting by noticing the changes in the output.

Alternatively, there is the white-box approach to consider with access to the web application's source code. Kieyzun et al. [41] used a technique exploiting information about the code that automatically generates inputs targeting SQLi and XSS vulnerabilities.

Moreover, Artzi et al. [4] developed another tool for discovering web application vulnerabilities by collecting information about the target extracted through concrete and symbolic execution.

White-box methods outperform black-box approaches by having access to the source code of the target being fuzzed. However, black-box processes are more scalable when the source code is not available.

To conclude, web vulnerability scanners are also realized through static analysis tools [6, 38,39,47,48]. Prime examples are Pixy [38] which uses static analysis at the source code level to detect vulnerable code. Another tool combining static and dynamic analysis is Saner [6] which tries to identify any sanitization processes that do not work as expected to, resulting in allowing attackers to introduce exploits.

Contrary to the above research work for identifying web vulnerabilities, our technique adopts the grey-box approach. webFuzz *instruments* the fuzz target to receive feedback on whether a generated input is interesting. These inputs were used to generate other test cases that resulted in wider code coverage that triggered more vulnerabilities (Chapter 5).

Unlike other fuzzers mentioned that generate their own XSS payloads [27], our tool's main objective is finding trigger points on the target web application and supplying them with known XSS payloads.

Chapter 8

Conclusion

Fuzzing has evolved significantly in analysing native applications, becoming a 'hot' field for research. While used extensively to uncover destructive bugs and security vulnerabilities in native apps, web applications have received scarce attention.

In this thesis, we presented webFuzz, a prototype open-source grey-box fuzzer for discovering Cross-Site Scripting vulnerabilities in web applications. webFuzz utilises *instrumentation* on the target web application to produce a *feedback loop*, employing it to boost code coverage score. Consequently, it increased the total of potential vulnerabilities found.

For the evaluation of webFuzz, we used four web applications; namely WordPress, Firefly-III, Mautic and Drupal, for the following three metrics: competence in detecting Reflected Cross-Site Scripting bugs, Throughput and Global Code Coverage.

Regarding the first metric, webFuzz was able to detect the most artificially injected vulnerabilities compared to the other three black-box fuzzers in the test. More specifically, it was able to expunge *30* bugs followed by Wfuzz with *28*.

Secondly, in terms of throughput, unfortunately, webFuzz does not match the throughput of native applications fuzzers such as AFL nor black-box web-app fuzzers such as Wfuzz as the overhead from the *instrumentation* is hefty.

Other outcomes in our third metric, suggest that our fuzzing tool can achieve coverage of the WordPress and Drupal codebase up to *21.5%* and *27.3%* respectively, in *4.2* days of fuzzing.

Bibliography

[1] aiomultiprocess documentation. `https://aiomultiprocess.omnilib.dev/en/stable/`, 2020.

[2] aiohttp maintainers. Welcome to aiohttp — aiohttp 3.7.3 documentation. `https://docs.aiohttp.org/en/stable/index.html`, 2020.

[3] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan. Navex: Precise and scalable exploit generation for dynamic web applications. In *27th USENIX Security Symposium*, 2018.

[4] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering*, 2010.

[5] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 334–349, 2017.

[6] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, 2008.

[7] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *2010 IEEE Symposium on Security and Privacy*, 2010.

[8] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[9] A. Borges. Bazinga: Whitebox fuzzing for detecting web application vulnerabilities. 2018.

[10] BuiltWith. Cms technologies web usage distribution. https://trends.builtwith.com/cms, 2020.

[11] E. Clarke, W. Klieber, M. Novavcek, and P. Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, 2012.

[12] Cornelius Aschermann et al. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[13] Cornelius Aschermann et al. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[14] Crummy. Beautiful soup documentation. https://www.crummy.com/software/BeautifulSoup/bs4/doc/, 2020.

[15] CVE. Common vulnerabilities and exposures (cve). https://www.cvedetails.com/vulnerabilities-by-types.php, 2020.

[16] S. D. Bekerman. Imperva: The state of vulnerabilities in 2019. https://www.imperva.com/blog/the-state-of-vulnerabilities-in-2019/, 2019.

[17] DARPA. Cyber grand challenge (cgc). https://www.darpa.mil/program/cyber-grand-challenge, 2016.

[18] J. D. DeMott, R. J. Enbody, and W. F. Punch. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. DEFCON, 2007.

[19] DevDungeon. Curses programming in python. https://www.devdungeon.com/content/curses-programming-python, 2019.

[20] Docker. Empowering app development for developers. https://www.docker.com/, 2019.

[21] Docker. Docker hub. https://hub.docker.com/, 2020.

[22] Docker. What is a container? https://www.docker.com/resources/what-container, 2020.

[23] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016.

[24] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 523–538, Bellevue, WA, aug 2012. USENIX Association.

[25] A. Doupé, M. Cova, and G. Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In C. Kreibich and M. Jahnke, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[26] Drupal. Open source cms. https://www.drupal.org/, 2020.

[27] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz. Kameleonfuzz: Evolutionary fuzzing for black-box xss detection. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, page 3748, New York, NY, USA, 2014. Association for Computing Machinery.

[28] M. Flaxman. Python 3's killer feature: asyncio. https://eng.paxos.com/python-3s-killer-feature-asyncio, 2020.

[29] Forbes. Hackers take control of giant construction cranes. https://www.forbes.com/sites/thomasbrewster/2019/01/15/exclusive-watch-hackers-take-control-of-giant-construction-cranes/?sh=f630b981d0a5, 2019.

[30] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. Association for Computing Machinery.

[31] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 2012.

[32] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[33] Hack. Programming productivity without breaking things. `https://hacklang.org/`, 2020.

[34] hackerone. Twitter - bug bounty program. `https://hackerone.com/twitter?type=team&view_policy=true`, 2020.

[35] A. Hoffman. *Web Application Security: Exploitation and Countermeasures for Modern Web Applications*. O'Reilly Media, 2020.

[36] A. D. Householder and J. M. Foote. Probability-based parameter selection for black-box fuzz testing. Technical report, 2012.

[37] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. International Conference on Software Engineering (ICSE), 2014.

[38] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 6–pp. IEEE, 2006.

[39] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, PLAS '06, page 27–36, New York, NY, USA, 2006. Association for Computing Machinery.

[40] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. Secubat: A web vulnerability scanner. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, page 247256, New York, NY, USA, 2006. Association for Computing Machinery.

[41] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *2009 IEEE 31st International Conference on Software Engineering*, pages 199–209, 2009.

[42] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[43] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery.

[44] S. F. M. Isaak. The new york times: Facebook security breach exposes accounts of 50 million users. `https://www.nytimes.com/2018/09/28/technology/facebook-hack-data-breach.html`, 2019.

[45] MariaDB.org. Mariadb.org foundation. `https://mariadb.org/`, 2019.

[46] MDN. Content security policy (csp). `https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP#:%7E:text=Content%20Security%20Policy%20(CSP)%20is,XSS)%20and%20data%20injection%20attacks.&text=If%20the%20site%20doesn't,the%20standard%20same%2Dorigin%20policy.`, 2020.

[47] I. Medeiros, N. Neves, and M. Correia. Dekant: A static analysis tool that learns to detect web application vulnerabilities. ISSTA 2016, page 1–11, New York, NY, USA, 2016. Association for Computing Machinery.

[48] I. Medeiros, N. F. Neves, and M. Correia. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, page 63–74, New York, NY, USA, 2014. Association for Computing Machinery.

[49] X. Mendez. Wfuzz - the web fuzzer. `https://github.com/xmendez/wfuzz`, 2014.

[50] D. Miessler. Seclists. `https://github.com/danielmiessler/SecLists`.

[51] Miller. Fuzz testing of application reliability. `http://pages.cs.wisc.edu/~bart/fuzz/`, last accessed in November 2020., 2008.

[52] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[53] A. Mouat. *Using Docker: Developing and Deploying Software with Containers*, volume 1. O'Reilly Media, Inc, 2015.

[54] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 919–936, Baltimore, MD, Aug. 2018. USENIX Association.

[55] L. Newman. How facebook catches bugs in its 100 million lines of code. `https://www.wired.com/story/facebook-zoncolan-static-analysis-tool/`, 2019.

[56] NGINX. High performance load balancer, web server, reverse proxy. `https://www.nginx.com/`, 2019.

[57] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium*, pages 2289–2306. USENIX Association, aug 2020.

[58] OWASP. Fuzzing. `https://owasp.org/www-community/Fuzzing`, 2020.

[59] OWASP. The zap homepage. `https://www.zaproxy.org/`, 2020.

[60] owasp.org. Owasp top ten web application security risks. `https://owasp.org/www-project-top-ten/`, 2017.

[61] M. Pezzè and C. Zhang. *Chapter One - Automated Test Oracles: A Survey*, volume 95 of *Advances in Computers*. Elsevier, 2014.

[62] PortSwigger. Burp suite - application security testing software. `https://portswigger.net/burp`, 2020.

[63] PortSwigger. Web security academy: What is reflected xss (cross-site scripting)? `https://portswigger.net/web-security/cross-site-scripting/reflected`, 2020.

[64] PythonDocs. argparse - parser for command-line options, arguments and sub-commands - python 3.9.1 documentation. https://docs.python.org/3/library/argparse.html, 2020.

[65] PythonDocs. Pylint - code analysis for python | www.pylint.org. https://www.pylint.org/, 2020.

[66] PythonDocs. pytest: helps you write better programs — pytest documentation. https://docs.pytest.org/en/stable/, 2020.

[67] PythonDocs. unittest — unit testing framework — python 3.9.1 documentation. https://docs.python.org/3/library/unittest.html, 2020.

[68] PythonDocs. unittest.mock - mock object library - python 3.9.1 documentation. https://docs.python.org/3/library/unittest.mock.html, 2020.

[69] PythonDocs. urllib.parse - parse urls into components - python 3.9.1 documentation. https://docs.python.org/3/library/urllib.parse.html, 2020.

[70] Python.org. Pep 257 – docstring conventions. https://www.python.org/dev/peps/pep-0257/, 2020.

[71] Python.org. Pep 8 – style guide for python code. https://www.python.org/dev/peps/pep-0008/, 2020.

[72] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[73] RealPython. Async io in python: A complete walkthrough. https://realpython.com/async-io-python/, 2020.

[74] RealPython. Python mmap: Improved file i/o with memory mapping. https://realpython.com/python-mmap/, 2020.

[75] RealPython. Speed up your python program with concurrency. https://realpython.com/python-concurrency/, 2020.

[76] RealPython. What is the python global interpreter lock (gil)? https://realpython.com/python-gil/, 2020.

[77] L. Rizzo and M. Landi. Netmap: Memory mapped access to network devices. *SIGCOMM Comput. Commun. Rev.*, 41(4):422–423, Aug. 2011.

[78] S. M. Seal. Optimizing web application fuzzing with genetic algorithms and language theory. Master's thesis, 2016.

[79] S. Sparks, S. Embleton, R. Cunningham, and C. Zou. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 477–486, 2007.

[80] Sphinx. Sphinx 4.0.0+ documentation. `https://www.sphinx-doc.org/en/master/`, 2020.

[81] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[82] swisskyrepo. swisskyrepo/payloadsallthethings. `https://github.com/swisskyrepo/PayloadsAllTheThings`, 2020.

[83] A. Takanen, J. Demott, C. Miller, and A. Kettunen. *Fuzzing for Software Security Testing and Quality Assurance*. Artech, second edition, 2018.

[84] I. Tasdelen. payloadbox/xss-payload-list. `https://github.com/payloadbox/xss-payload-list`, 2020.

[85] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling black-box mutational fuzzing. CCS '13, page 511522, New York, NY, USA, 2013. Association for Computing Machinery.

[86] M. Zalewski. Binary fuzzing strategies: what works, what doesn't. `https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html`, aug 2014.

[87] M. Zalewski. American fuzzy lop. `http://lcamtuf.coredump.cx/afl`, 2015.

[88] C. Zapponi. Githut. `http://githut.info/`, 2020.

Appendix A

```
version: '3.3'
services:
  mariadb:
    image: mariadb:10.5
    volumes:
      - ./data/mariadb:/var/lib/mysql
    ports:
      - 3306:3306
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: db_fuzz
      MYSQL_USER: user
      MYSQL_PASSWORD: password
    restart: always
  wordpress:
    image: wordpress:php7.3-fpm
    volumes:
      - ./data/wordpress:/var/www/html
    depends_on:
      - mariadb
    environment:
      WORDPRESS_DB_HOST: mariadb
      MYSQL_ROOT_PASSWORD: root
      WORDPRESS_DB_NAME: db_fuzz
      WORDPRESS_DB_USER: user
      WORDPRESS_DB_PASSWORD: password
      WORDPRESS_TABLE_PREFIX: wp_
    links:
      - mariadb
```

```
30      restart: always
31    nginx:
32      image: nginx:alpine
33      volumes:
34        - ./data/nginx:/etc/nginx/conf.d
35        - ./data/wordpress:/var/www/html
36      ports:
37        - 8080:80 # Host port 8080 mapped to the container port 80.
38      links:
39        - wordpress
```

Listing A.1: *Docker-compose file used during the deployment of WordPress*

Appendix B

```
aiohttp==3.7.2
argparse==1.4.0
asyncio==3.4.3
pathlib==1.0.1
psutil==5.7.3
jsonschema==3.2.0
selenium==3.141.0
bs4==0.0.1
lxml==4.6.1
mock==4.0.2
browsermob-proxy==0.8.0
jsonpickle==1.4.1
pyfiglet==0.7
termcolor==1.1.0
pytest==6.1.2
kids-cache==0.0.7
typed-argument-parser==1.6.1
typing==3.7.4.3
```

Listing B.1: *Modules needed to be installed so webFuzz can execute smoothly*

Appendix C

```
1  function handlePost(): void {
2      $input = $_POST["name"];
3      $a=$_POST["pass"];
4      $magic_number=5;
5      if ($a==$magic_number){
6        echo $input."␣";
7        echo $a;
8       }else{
9        echo "Welcome!";
10     }
11 }
12
13 <<__EntryPoint>>
14 function main(): noreturn{
15   if ($_SERVER['REQUEST_METHOD'] === 'POST') {
16     handlePost();
17   }
18   exit(0);
19 }
```

Listing C.1: *An XSS vulnerability which can be exploited through a URL that will give, as a parameter, the magic number needed to execute the malicious script. The malicious script is passed through the name parameter. This code is written in Hack [33]*