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 sincere gratitude to my Thesis Supervising 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 expert knowledge in the field of Cybersecurity has undoubtedly been a source of inspiration to me. 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 participation in a greater 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 has been 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 for memory-corruption vulnerability.

Although fuzzing significantly evolved in analysing native code, web applications, invariably, have received limited attention, so far. This thesis explores the technique of gray-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. WebFuzz leverages instrumentation for successfully detecting reflective Cross-Site Scripting (XSS) vulnerabilities faster than other black-box fuzzers. The functionality of webFuzz is demonstrated using web applications written in PHP; namely WordPress and Drupal.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Related Work 2

1.3 Contributions 3

1.4 Thesis Outline 4

2 Background **5**

2.1 Web Application Bugs 5

2.2 Fuzzing 8

2.3 Instrumentation 10

2.4 Concurrency 11

2.5 Docker 12

3 Architecture **14**

3.1 A Fuzzing Session 14

3.2 Mutations 15

3.3 Detecting Vulnerabilities 17

4 Implementation **19**

4.1 Coding Standards 19

4.2 Asynchronous I/O 20

4.3 Parser 22

4.4 Curses Interface 23

4.5 Running webFuzz 25

4.6 Interactive and Black-Box Functionalities 25

5 Evaluation 28

5.1 Methodology 28

5.2 Automated Vulnerability Addition 30

5.3 Evaluation Details 31

5.4 Vulnerabilities Detected 32

5.5 Throughput 35

5.6 Global Code Coverage 36

6 Discussion 39

6.1 Limitations 39

6.2 Future Work 40

7 Related Work 43

7.1 Generic Fuzzing 43

7.2 Web Applications Fuzzing 44

8 Conclusion **46**

Bibliography **54**

Appendix A **A-1**

Appendix B **1**

List of Figures

2.1 How Stored Cross-Site Scripting can be exploited by an attacker 8

2.2 How Reflected Cross-Site Scripting can be exploited by an attacker 9

2.3 Requests over the internet processed in concurrent fashion [19] 12

3.1 High-level overview of a webFuzz fuzzing session 15

4.1 AsyncIO mechanism 21

4.2 Interface of webFuzz 25

4.3 webFuzz usage menu 26

5.1 Multi-Container Deployment of WordPress using Docker 30

5.2 Number of artificial XSS bugs uncovered by webFuzz and Wfuzz 34

5.3 Throughput of webFuzz and Wfuzz when fuzzing Drupal and WordPress 35

5.4 Accumulated Global Code Coverage using webFuzz 37

5.5 Accumulated Global Code Coverage using Wfuzz 38

List of Tables

3.1 Cross-Site Scripting payloads corpus 17

5.1 Vulnerability detection summary 32

Chapter 1

Introduction

Contents

1.1 Motivation 1

1.2 Related Work 2

1.3 Contributions 3

1.4 Thesis Outline 4

In the introductory chapter to this thesis we analyse what motivated us to explore this research area of grey-box fuzzing and start this project. We also discuss related studies in the field of fuzzing, the contribution that this thesis makes to it while outlining the chapter topics to follow.

1.1 Motivation

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 in an automated process has been precipitated with the introduction of AFL [77], a state-of-the-art fuzzer that produces feedback during fuzzing by leveraging instrumentation of the analysed program. By creating this *feedback loop*, fuzzers can significantly improve their performance as they can determine whether an input is interesting, namely it triggers a new code path, and uses that input to produce other test cases.

Software-testing plays a vital role in the software development cycle because when vulnerabilities are present, they can have severe and even irreparable consequences. By exploiting software bugs, adversaries can perform data breaches, install malicious malware or even take complete control of a device. Detecting bugs before they get exploited is a possible but demanding task. Mainly because bugs are triggered when an unexpected input is given to the program, something which is difficult to fully simulate through statically written unit tests. This is due to the fact unit tests usually revolve around expected inputs to test the intended functionality of code [35].

Although automated software testing has become an attractive field of research, it still has a long way to go, especially for web applications [45]. As the Internet infrastructure expands, much more of the software written in native code(precompiled 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

Numerous fuzzers recently developed try to optimize the fuzzing process by proposing different methodologies [35, 36, 49, 50, 63, 68, 72]. For example, most of the 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 and use that information for improving the input generation process [48,49,72]. However, all these fuzzers are currently targeted towards finding vulnerabilities in native code while web applications - which do not run on native code - have received limited attention. A more detailed analysis is given in Chapter 7.

Traditionally, fuzzers come under three categories; black-, white- or grey-box which are clearly define for native applications. When it comes to web applications grey-box methods 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 [45]. That is to say they use brute force to bombard their targets with URLs that embed known web-attack payloads. Recently, there were breakthroughs with white-box fuzzing also [27, 33], 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 begins sending 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, the first grey-box fuzzer realized for discovering vulnerabilities in web applications. webFuzz applies instrumentation on the target web application for guiding the entire fuzzing process. Instrumentation can be applied on the AST level of PHP-based web applications for creating a feedback loop and utilizing it in order to increase code coverage.

2. We thoroughly evaluate webFuzz in terms of coverage, throughput and efficiency in finding unknown bugs. For a better understanding of the measured capabilities of webFuzz we compare our results with three existing web-application fuzzers. webFuzz is the only fuzzer that reports coverage information. Specifically, web-

Fuzz can cover about 21.5% of WordPress, which has a codebase of approximately half a million lines of PHP code, in 50 hours of fuzzing. As expected, webFuzz is slower, in terms of throughput, due to the involved instrumentation. In fact, another popular fuzzer, Wfuzz [57] is three times faster when fuzzing Drupal, but this is something to be expected, since the reduction of the throughput due to the instrumentation pays off in increased coverage in the long run. Finally, webFuzz, compared to the other three fuzzers, finds the most injected vulnerabilities (30 with the second one being Wfuzz with 28) during a fuzzing session that lasts 65 hours. The evaluation of webFuzz can be seen in detail in Chapter 5

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

1.4 Thesis Outline

This thesis consists of eight chapters. In the first chapter we present our inspiration for undertaking this research, related work on the said 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 details. The fourth chapter is dedicated to discussing the technical aspects of the fuzzing tool developed. The fifth chapter is an evaluation to see how well webFuzz performs in terms of 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 over recent years. In the eighth and final chapter we summarise and reflect on the research done, and conclude on the evaluation of our approach.

Chapter 2

Background

Contents

2.1 Web Application Bugs 5

2.2 Fuzzing 8

2.3 Instrumentation 10

2.4 Concurrency 11

2.5 Docker 12

In this chapter we provide background information giving a detailed understanding on several key points concerning this thesis. First, we define what a Cross-Site Scripting bug is in web applications, and elaborate by giving a specific example on how this vulnerability may occur. Then, we briefly discuss what fuzzing is and the various categories that constitute it and show how instrumentation helps when used during gray-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 on-line, writing a web application so that it is secure from any vulnerability, can be extremely difficult. Every significant web application, especially large-scale ones that are composed with thousands of lines of code, have bugs in them. 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 (19.2%) (DoS) is ranked second and Cross-Site Scripting (XSS) (12.5%) is fourth among the top Cybersecurity vulnerabilities [38].

The Open Web Application Security Project (OWASP) Top 10 represents a broad consensus on the most critical security risks to web applications [64]. One of the most pressing security problems 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 [23] 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 first permanently stored in storage such as a database residing on a server, and is only later outputted by an unsuspecting query. Example locations where Stored XSS may occur include Web forums or blog comments.

webFuzz focuses in detecting bugs that can lead to both Reflected or Stored Cross-Site Scripting, which are among the most common of XSS attacks. A step-by-step illustration of the latter can be seen in Figure 2.1 and of the former in Figure 2.2. In both illustrations, the attacker and victim is 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 the 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 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 code above 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 which 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 website. This malicious form is designed to trigger the vulnerability found in the above login form. As soon as the form is submitted the vulnerable login page is opened with the XSS script executed in it. If the victim now tries to login, the XSS script can easily send the credentials to the attacker as well.

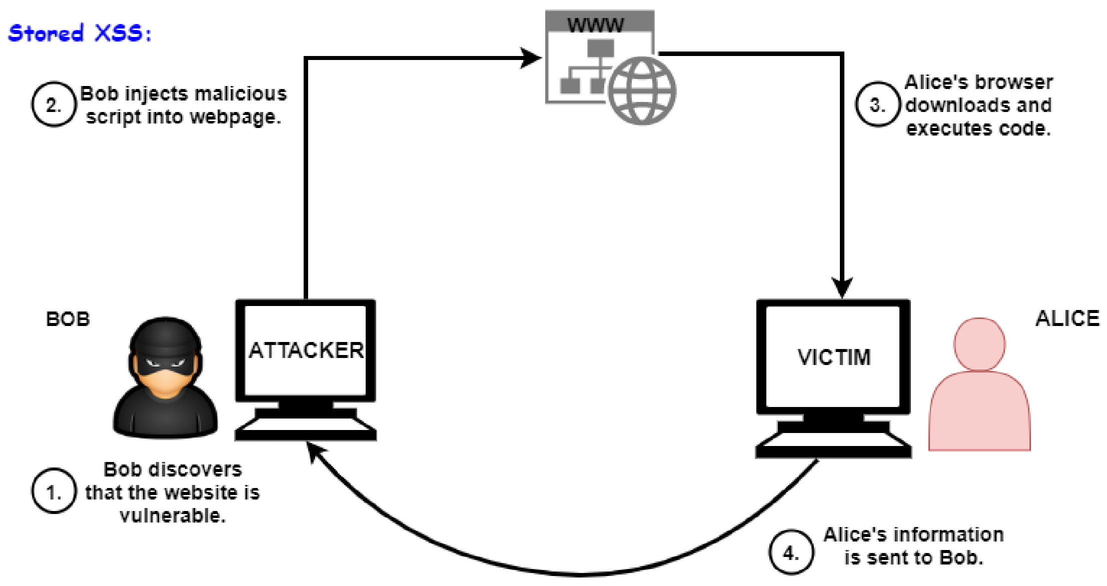

Stored XSS:
2. Bob injects malicious script into webpage.
WWW
3. Alice's browser downloads and executes code.
BOB
ATTACKER
VICTIM
ALICE
1. Bob discovers that the website is vulnerable.
4. Alice's information is sent to Bob.

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

Defeating XSS attacks is not dissimilar to defending against other types of code injection. The input must be sanitized. User input containing HTTP code needs to be escaped or encoded to avoid its execution. Also, system-wide measures such as Content Security Policy(CSP) [14] may be enabled to eliminate or mitigate XSS attacks. Nevertheless, flaws such as Buffer Overflows (CVE ranked 3 [38]) or Cross-Site Scripting issues comprise a majority of security incidents, and malicious hackers abuse them on a daily basis.

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) [59]. Fuzzing was originally introduced by Barton Miller at the University of Wisconsin, as one of several tools to test UNIX utilities [60]. With this quality assurance technique, software is exercised using a vast number of anomalous inputs for inferring if any of them introduce security-related side-effects. A fuzzer, which is the tool that can automate the aforementioned stress-testing process, can be categorized in relation to its awareness of the program structure as black-, white-, or gray-box [73].

A black-box fuzzer treats the program as a 'black box' and is unaware of internal program

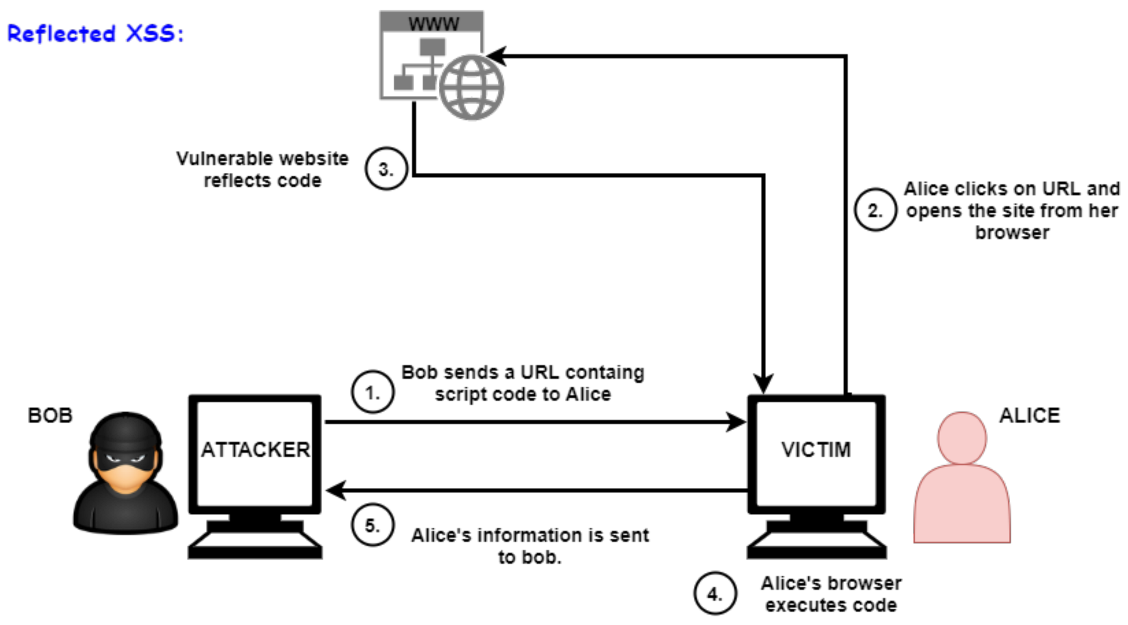

Reflected XSS:
WWW
Vulnerable website reflects code
3.
2.
Alice clicks on URL and opens the site from her browser
1.
Bob sends a URL containg script code to Alice
BOB
ATTACKER
VICTIM
ALICE
5.
Alice's information is sent to bob.
4.
Alice's browser executes code

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

structure. It conducts its test on the target through external interfaces and produces random inputs using no information of the target's underlying structure. More often than not, black-box fuzzers are only able to scratch the surface and expose "shallow" bugs [15].

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. Program analysis can be based on either static or dynamic analysis, or their combination [65]. They may also leverage symbolic execution to derive what inputs cause each part of a program to execute [74]. This makes them very effective at exposing bugs that hide deep in the program. By studying the application code, you may be able to detect optional or proprietary features, which should be tested as well.

A fuzzer is considered gray-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 [39, 77]. This thesis explores in detail gray-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 can 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. This can be explained by the fact that 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 [52, 56]. 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 [32] does. Therefore, code coverage should be considered a secondary metric and the number of bugs found as primary [56]. Nevertheless, measuring coverage is important for any fuzzer.

Available fuzzers for web applications act in a black-box fashion [45]; 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 starts sending 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. This gives us a good idea of the coverage but omits crucial information such as combinations of basic blocks that were visited one after the other.

We instrumented web applications for delivering feedback once they were 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 Abstract Syntax Tree (AST) of PHP files and then reverting it back to source code form. This provided us crucial feedback on the basic blocks that are visited during analysis. Instrumentation performed by webFuzz on our targeted web application is similar to how AFL instruments binaries, but it was adapted to work in web applications. A more detailed approach of the instrumenting functionality provided by webFuzz is beyond the scope of this thesis.

2.4 Concurrency

Concurrency is defined as working on multiple tasks at the same time [19]. 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, the state of each task is stored, so it can be restarted 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 down dramatically when they are obliged to wait often for I/O operation from some external resource. An example of such a resource are requests on the internet or any kind of network traffic that can take several orders of magnitude longer than CPU instructions. An illustration of the above can be seen in Figure 2.3:

In Python, concurrency can be expressed either through the Threading or AsyncIO(short

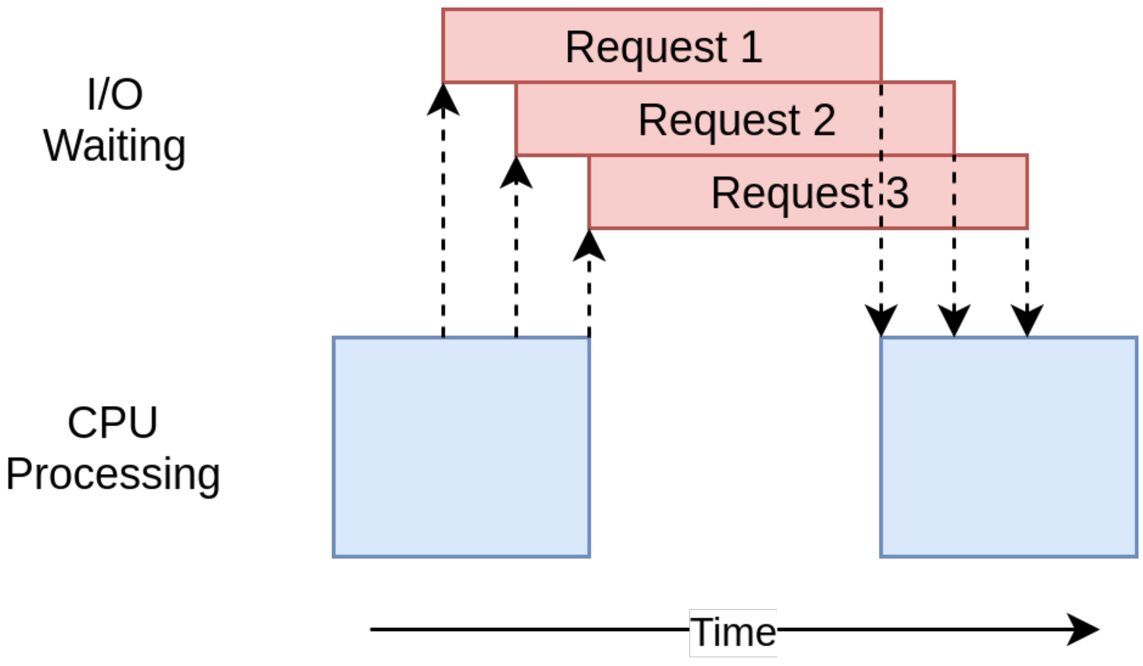


Figure 2.3: Requests over the internet processed in concurrent fashion [19]

for Asynchronous Input Output) [18] modules. Due to the infamous Global Interpreter Lock (GIL) [20] Python has, both AsyncIO and Threading are single-threaded, single-process design. There was no clear advantage in using the latter so AsyncIO was opted for instead, although some initial work was done with threading it was later 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 [1] provide developers the commodity for creating software locally with the knowledge that it will run identically regardless of the host environment [61]. 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-composed 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 [12].

Chapter 3

Architecture

Contents

3.1 A Fuzzing Session 14

3.2 Mutations 15

3.3 Detecting Vulnerabilities 17

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 A Fuzzing Session

webFuzz constitutes two intertwined components that work together in providing a guided fuzzing approach with the aim of finding 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 so as 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 all 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 provide new, unseen paths of the web application to further explore.

A regular fuzzing session using webFuzz can be 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 that has been 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.

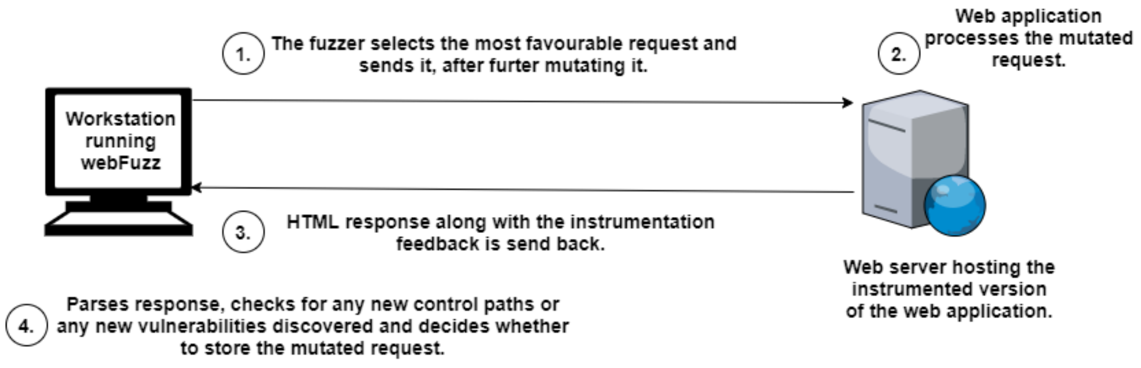

1. The fuzzer selects the most favourable request and sends it, after furter mutating it.
2. Web application processes the mutated request.
Workstation running webFuzz
3. HTML response along with the instrumentation feedback is send back.
Web server hosting the instrumented version of the web application.
4. Parses response, checks for any new control paths or any new vulnerabilities discovered and decides whether to store the mutated request.

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

3.2 Mutations

In most cases, sending randomly generated inputs will be 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 [39] and AFL [77] 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 because we need it to maintain diversity in our test cases to avoid stagnation on a suboptimal plateau in the search space [70]. Choosing which mutation function to use 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 and lead to the test cases failing at a premature stage of the execution [76].

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.

This can be 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 [46], webFuzz chooses randomly from a corpus that consists of real-life known XSS payloads. The corpus was created with payloads that where found scattered across the internet, mainly in open-source repositories [2,3,58].

A small sample of XSS payloads contained in the corpus can be viewed at 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 it is, nevertheless, time consuming.

Although arrays in URL strings are not clearly defined in RFCs and their format is 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 that is expected to be parsed as a string in the web application is transformed into an array or vice versa. Web applications not equipped to process unexpected types of input can be prone to glitches and bugs.

	Corpus of known Cross-Site Scripting payloads
1	<form onsubmit=alert(1)><input type=submit>
2	<a draggable="true" ondragstart="alert(1)">test</a>
3	<abbr id=x tabindex=1 onbeforedeactivate=alert(1)></abbr><input autofocus>
4	<body onscroll=alert(1)><div style=height:1000px></div><div id=x></div>
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 Cross-Site Scripting payloads from the corpus webFuzz uses during its fuzzing session. The corpus consists of thousands of payloads

Using evolutionary algorithms in the test case creation process is widely practised in fuzzers to optimize solution searching [70], webFuzz will also mix GET or POST parameters from various favourable requests to generate new inputs. Contrary to how evolutionary algorithms work, this 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 is able to detect Reflected and Stored Cross-Site Scripting(XSS) 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 not accounted for. False positives arise when the tool reports that an XSS was detected when in fact there is none. One example is when the XSS payload is 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 then the payload will not be executable. There are plans to improve the efficiency of our XSS detection method which is discussed in Chapter 6.

Chapter 4

Implementation

Contents

4.1 Coding Standards 19

4.2 Asynchronous I/O 20

4.3 Parser 22

4.4 Curses Interface 23

4.5 Running webFuzz 25

4.6 Interactive and Black-Box Functionalities 25

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

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, our main aim was to write clean, readable and eye-pleasing code by following best practice that the best 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 good programming habits and practices.

More specifically, our fuzzing tool is fully written in Python 3.8 using the PEP 8 [67] coding style standard and, regarding documentation, the PEP 257 [66] and Sphinx [21] docstring conventions were adopted for it to be clear and easy to read for programmers. Pylint [40] was also used to check for errors in Python code and to implement the aforementioned coding standards and search for code smells.

To enhance good 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 like pytest [41], unittest [42] and mock [16]. In the appendix, an example of unit testing for the Parser module can be found.

4.2 Asynchronous I/O

webFuzz utilises concurrent programming (see Section 2) with the help of the asyncio [18] 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 aforementioned module, some of the 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 other 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 mean-time. 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 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.

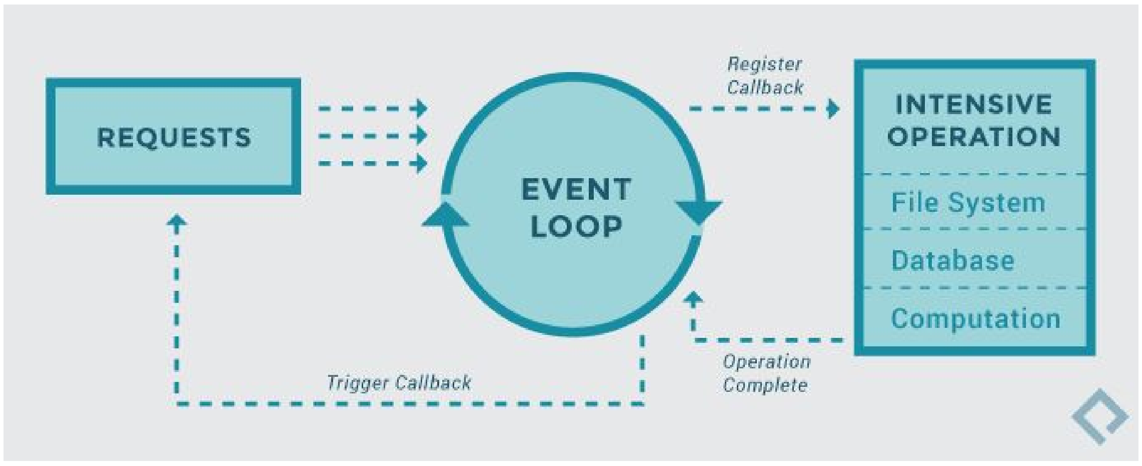


Figure 4.1: AsyncIO mechanism; it provides a high-performance asynchronous frameworks for making our fuzzing requests [47]

Communication with the target's website is achieved with a rapidly fast asynchronous http client/server framework named aiohttp [26]. 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 in turn uses Python's http and socket modules. Socket operations are blocking and not `awaitable` which signals that Python will not like the `await` statement. However, more modules are becoming compatible with asyncio [26].

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 then parsed using the `Beautiful Soup` [37] 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 (see Section 3) they will also be fuzzed in the future. At this stage the HTML document is also checked for XSS vulnerabilities. The metadata that we stored for each request, can tell us which XSS payloads were injected into it and led to this vulnerability. If they happen to reside in the HTML document, which signal an RXSS vulnerability, a warning is triggered, incrementing the total number of XSS found and logging the related information. The document is also checked for Stored XSS vulnerabilities by scanning the document for all the XSS payloads that were injected in all the requests. A high-level pseudo-code for the parsing process can be seen at Algorithm 1.

As the pseudo-code shows clearly, parsing relies heavily on the urllib.parse [22] Python module. This module, more precisely 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 6-item tuple with all the URL sub-fields. The reverse can also be achieved through the urlunparse method; a URL object can be converted into string.

4.4 Curses Interface

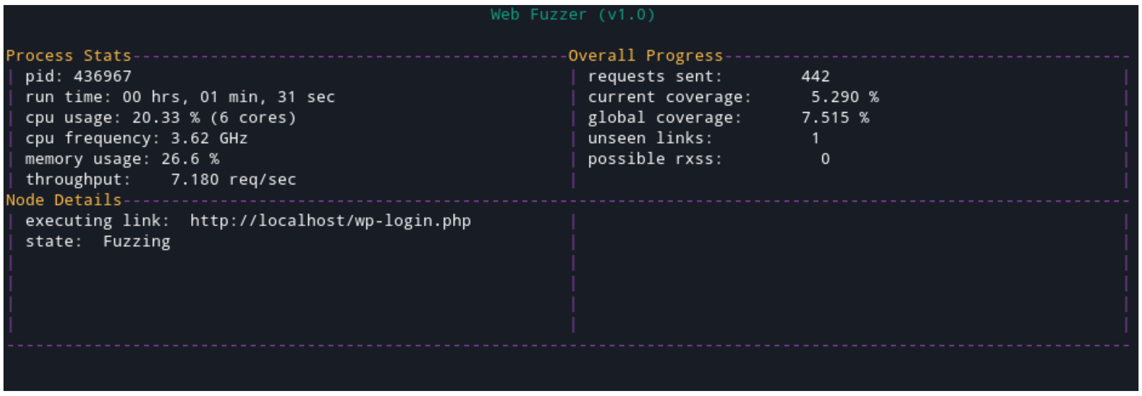
A Textual User Interface (TUI) for webFuzz has been created using the curses module which contains information and essential statistics, gathered while our grey-box fuzzer is running. The curses library supplies a terminal-independent screen-painting and keyboard-handling facility for text-based terminals [4], 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. So 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 sub-section. Although many may think this is obsolete technology, it can prove to be 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 provided by the first and original curses. A snapshot of the interface provided by webFuzz can be 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 expected to be included on the interface.

Algorithm 1 Parsing new HTML documents method 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
```


```
                                 Web Fuzzer (v1.0)

Process Stats-------------------------------------------Overall Progress----------------------------------
| pid: 436967                                            | requests sent:        442                      |
| run time: 00 hrs, 01 min, 31 sec                       | current coverage:      5.290 %                 |
| cpu usage: 20.33 % (6 cores)                           | global coverage:      7.515 %                  |
| cpu frequency: 3.62 GHz                                | unseen links:          1                       |
| memory usage: 26.6 %                                   | possible rxss:          0                      |
| throughput:     7.180 req/sec                          |                                                |
Node Details---------------------------------------------------------------------------------------------
| executing link:  http://localhost/wp-login.php         |                                                |
| state:  Fuzzing                                        |                                                |
|                                                        |                                                |
|                                                        |                                                |
|                                                        |                                                |
|                                                        |                                                |
---------------------------------------------------------------------------------------------------------
```

Figure 4.2: Interface of webFuzz. The interface is implemented using the Curses module

4.5 Running webFuzz

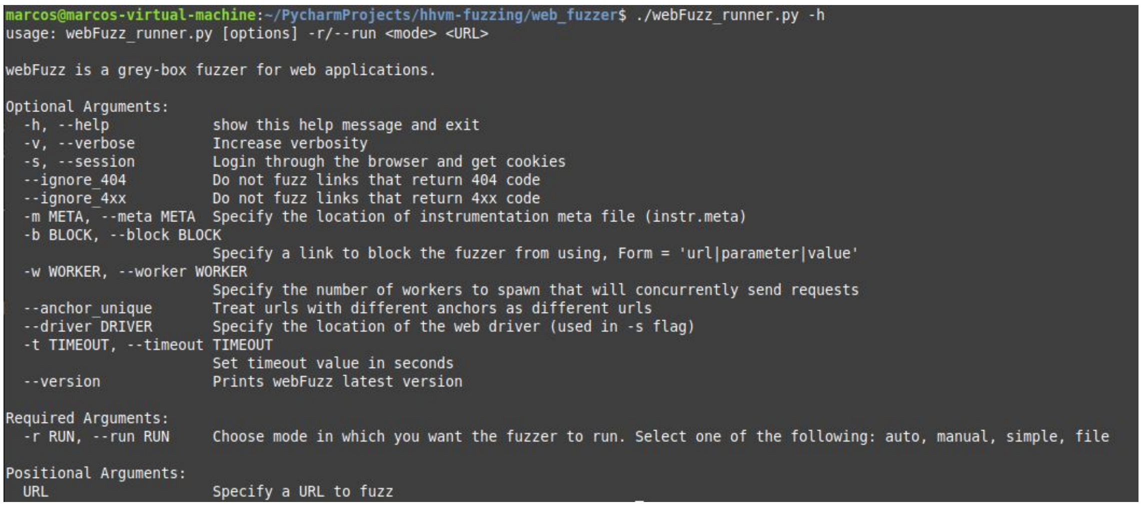
Running webFuzz is straightforward. All necessary modules accompanied with their exact version needed to be installed for webFuzz to work smoothly. These are listed in a "requirements" text file. Other executing and installing dependency instructions can be found at the README.md file in the tool's repository.

A help menu that shows all available arguments in which webFuzz can run in, are shown in Figure 4.3. As you can see, arguments are separated in 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 Positiona are the arguments that must be included. For the creation of the usage menu and parsing the arguments, the argparse [9] Python module was used.

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

4.6 Interactive and Black-Box Functionalities

Our fuzzing tool provides manual functionality also. This functionality allows the user to engage through an interactive session with our fuzzer. After the user provides the target



```
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 which we can use to run it with

to be fuzzed, a connection its establish and the session begins. Before a request is made, forms are extracted from the target and all input fields are presented to the user. Then, the user chooses from a menu of option which fields to fuzz and how. More specifically, options consist of filling the fields with manually inserted data, with XSS payloads from the corpus previously seen at 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 ate 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 4.1, we can see a code snippet of the interactive mode.

```
def fuzz():
  ....
  print("Please choose one of the following fuzzing methods:")
  print("(1) Select a random XSS payload from corpus")
  print("(2) Mutate previous inputs")
  print("(3) Manually insert data")
  print("(4) Switch to Black-box fuzzing")
  print("(5) Enter new link to fuzz")
  ans = input()

    if ans == "1":
    data[str(i)] = str(xss_fuzzing_payload())
  elif ans == "2":
    choose_mutating_function(data[str(i)])
  elif ans == "3":
    value = input("Insert data for {}: ".format(i))
    data[str(i)] = value
  elif ans == "4":
    black_box_fuzzing()
  elif ans == "5":
    link = input("Insert new link")
     ...
```

Listing 4.1: Options menu and their processing during interactive mode fuzzing

The user can opt to switch at any time to a more automated, brute-forcing like session where input fields of the forms at the given fuzz target are filled with XSS payloads, needless 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 mention this far still applies, as the core functionalities are shared between the two. The only thing that it not taken into consideration with this approach, is of course, the instrumentation.

As this thesis aims to demonstrate the capabilities of webFuzz as a grey-box fuzzer and because no rigorous tests have been conducted to show the efficacy, no further discussion is made in regards to its interactive and black-box functionalities.

Chapter 5

Evaluation

Contents

5.1 Methodology 28

5.2 Automated Vulnerability Addition 30

5.3 Evaluation Details 31

5.4 Vulnerabilities Detected 32

5.5 Throughput 35

5.6 Global Code Coverage 36

In this chapter we examine the evaluation of webFuzz against other black-box vulnerability scanners. We start by describing in detail the methodology that was used to evaluate our tool and explaining the automated bug injection process to our fuzz targets. Then, we discuss the details of the evaluation such as the metrics that are going to be used. After, we proceed reviewing the results of each metric used.

5.1 Methodology

For the evaluation of our tool, we opted for convenience to use Docker [5], 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 improve application development and deployment considerably.

As we already 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 [11]. For this reasons, we opted 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) [5] is one of 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. While it powers more than a third of the web, what was more important about 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.

We tested our tool on a second web application, Drupal CMS [13]. 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 back-end framework for at least 2.1% of all Web sites worldwide ranging from personal blogs to corporate, political, and government sites.

Using Docker, and more specifically its docker-compose functionality, we where 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 which handles all the HTTP request made by webFuzz and forwarded to our WordPress or Drupal web applications. [7]
- *WordPress and Drupal*: Both open-source CMS web application. Since having access to the code, we began by examining the existing system in terms of injecting bugs and performing our instrumentation.
- *MariaDB*: A popular open source relational databases which we used to store and manipulate the WordPress data [6].

The official images for the above services can be found for free at Docker Hub. An illustration of the above infrastructure for WordPress, can be viewed in Figure 5.1. Files and instructions for replicating this process can be found at the fuzzer's repository. The respective docker-compose can be seen in Appendix B.

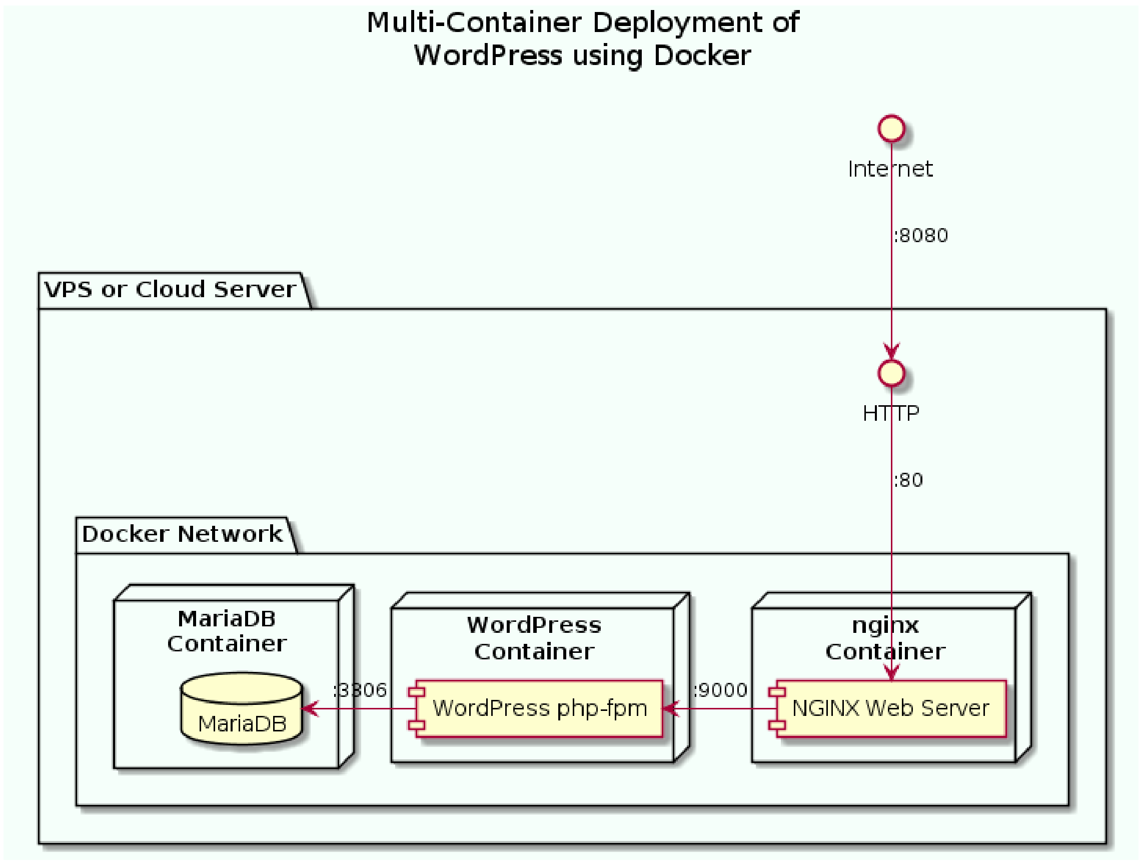


Figure 5.1: Evaluation followed the above Multi-Container Deployment of WordPress using Docker [24].

5.2 Automated Vulnerability Addition

Evaluating fuzzing processes proved to be a challenging task [56]. Migrating known vulnerabilities to existing software, to test the capabilities of the fuzzer in finding bugs, can be a tedious process [62]. For evaluating webFuzz, but also other fuzzers for web applications, an automated bug injection methodology, inspired by LAVA [43] was used for automatically injecting bugs in web applications written in PHP. Injecting vulnerabilities in web code was challenging, since 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, the vulnerability injection methodology leverages the instrumentation infrastructure we used for building webFuzz, in the first place. The automated bug injection method is able to inject hundreds of common vulnerabilities such as Reflected Cross-Site Scripting in reasonable time. Details about the automated vulnerability addition, like the instrumentation, are not in the scopes of this thesis.

5.3 Evaluation Details

For the evaluation of 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 64GB 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, and (c) an instrumented Drupal 9.0.6. 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:

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

To compare the Vulnerabilities Detected and the Throughput of webFuzz against other black-box fuzzers we used Wfuzz [57], Burp Suite Professional [10] and OWASP ZAP [25]. 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 DEF CON. 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 ease to extend. The choice of Wfuzz as the main comparison of our tool is further elaborated on in Chapter 6.

5.4 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 perform in comparison to webFuzz. The real-life RXSS bugs were found from CVE [38] and have the following ids: CVE-2018-7280, CVE-2019-11843, CVE-2020-7104, CVE-2020-7107.

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.

When analysing the specifics of the 4 real-life RXSS vulnerabilities that where manually injected, it was realised that 1 out of these depends on JavaScript code to create its triggering link dynamically in the form of an anchor element. The other 3 depend on JavaScript code to dynamically append the vulnerable POST parameter upon form submission. Also, for the vulnerability CVE-2018-7280 to be triggered it is compulsory for the XSS payload to be injected inside a specific JSON object at one of the vulnerable form's parameters. As Table 5.1, all tools involved in the evaluation only managed to find one real-life RXSS bug. This is because none of them employ complex enough JavaScript code analysis nor do they run a request's client-side code so as to uncover these dynamic links and parameters.

At this point, 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 opted to avoid these kinds of features since webFuzz currently lacks this functionality and thus, it would be an unfair comparison.

As we can clearly observe from Table 5.1, 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, hence we opted to only customise Wfuzz.

To make 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 brute forcer and not a black-box fuzzer. Firstly, 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 artifi-

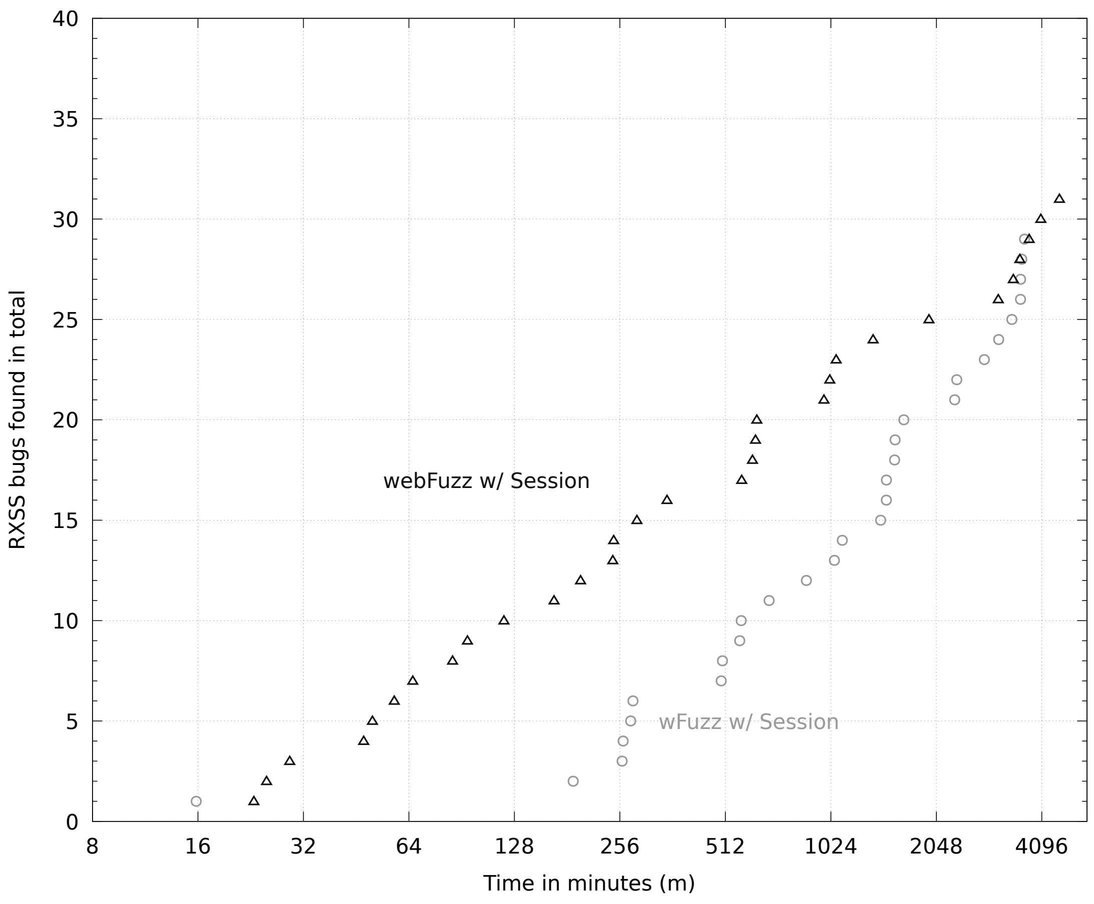


Figure 5.2: Artificial Reflected Cross-Site Scripting bugs detected over time by webFuzz and Wfuzz. webFuzz manages to uncover more bugs quicker during the fuzzing process.

cial RXSS bugs found can be seen in Figure 5.2. Although webFuzz has the lead through the entire experiment, the difference kept on decreasing until the end when the difference is marginal. webFuzz uncovered 30 artificial bugs, two more than the Wfuzz's 28.

By taking advantage of the instrumentation feedback-loop, webFuzz could detect 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.

Using this method, the finding of a magic number is done in an incremental fashion - one correct digit at a time - which is much faster than guessing the whole number at once like Wfuzz does. As 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 out of approximately half a million Lines of Code(LoC), with 48,040 basic blocks instrumented in total.

5.5 Throughput

One reason for the effectiveness of fuzzers in uncovering vulnerabilities is their capability to test vast amounts of inputs per second. It is essential that the overhead caused by instrumentation does not severely degrade the web application's response time and the fuzzer's processing time for each request is kept as brief as possible.

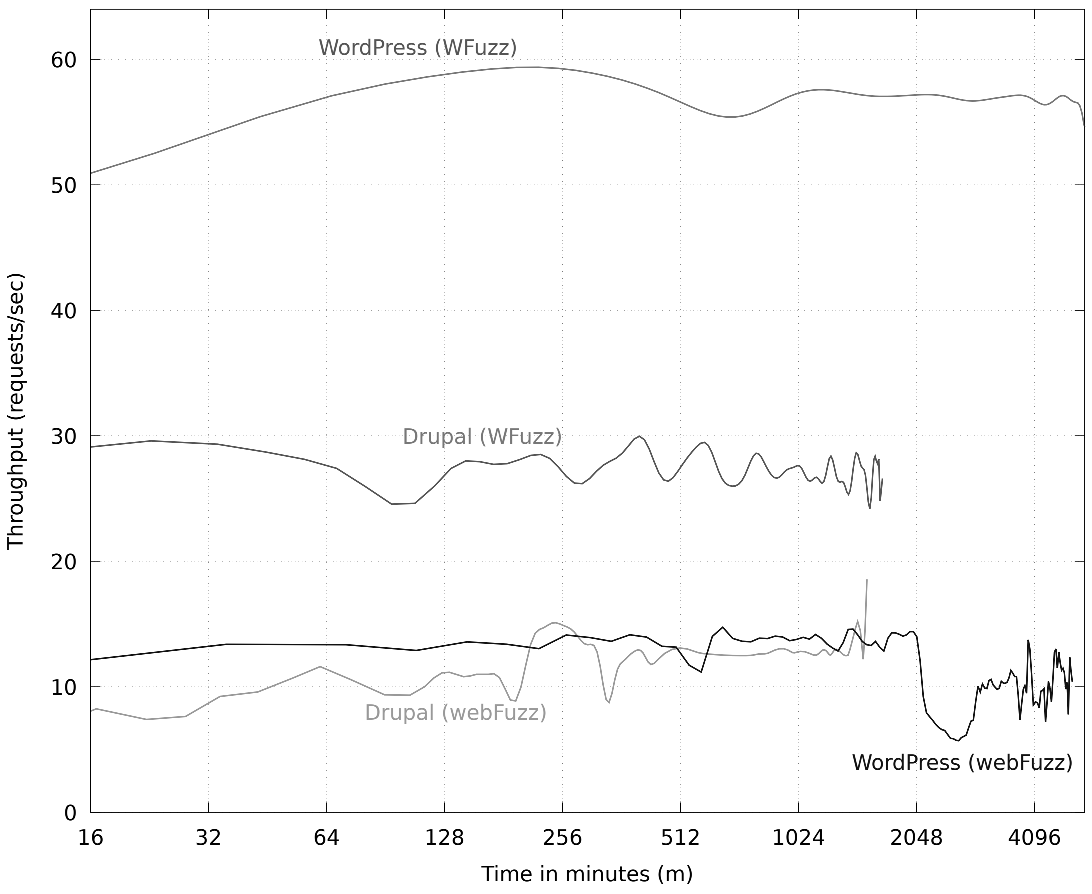


Figure 5.3: Requests made per second over time for three different scenarios. Both webFuzz and Wfuzz are evaluated on Drupal and WordPress. Wfuzz takes the lead with a difference.

As observed in Figure 5.3, 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 in the case of WordPress, and due to webFuzz's increased statefulness in tracking, analysing and ranking all the requests, increases the per request processing time.

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

Looking at Figure 5.3, we can state with confidence that much has to be done to improve the throughput of webFuzz, since it is nowhere near that of native applications fuzzers such as AFL and EFS nor is it comparable to black-box web fuzzers such as Wfuzz. This was no more than expected as native applications have no need to address overhead from sending requests over a network. Needless to say, necessary improvements are discussed in Chapter 6.

5.6 Global Code Coverage

Utilizing the instrumentation feedback, webFuzz has calculated the Global Code Coverage for both authenticated and guest sessions of WordPress and for an authenticated session run of Drupal. Observing Figure 5.4, we can see how the aforementioned metric rises over time for the two authenticated session scenarios.

During the unauthenticated session scenario of WordPress, access is forbidden to various links such as Administrative Dashboard related links. For this reason, the code coverage calculated is the smallest and remains stagnant at about *7%* for the best part of the experiment. However, the authenticated sessions have access to the Administrative Dashboard provided by Drupal and WordPress, and managed to achieve global code coverages as high as *30%* and *21.5%* respectively.

An important thing to note here is that the code coverage achieved by webFuzz in the authenticated session run of WordPress indicates rises in the global code coverage even after 50 hours of execution time. A good signal that the mutation functions used are

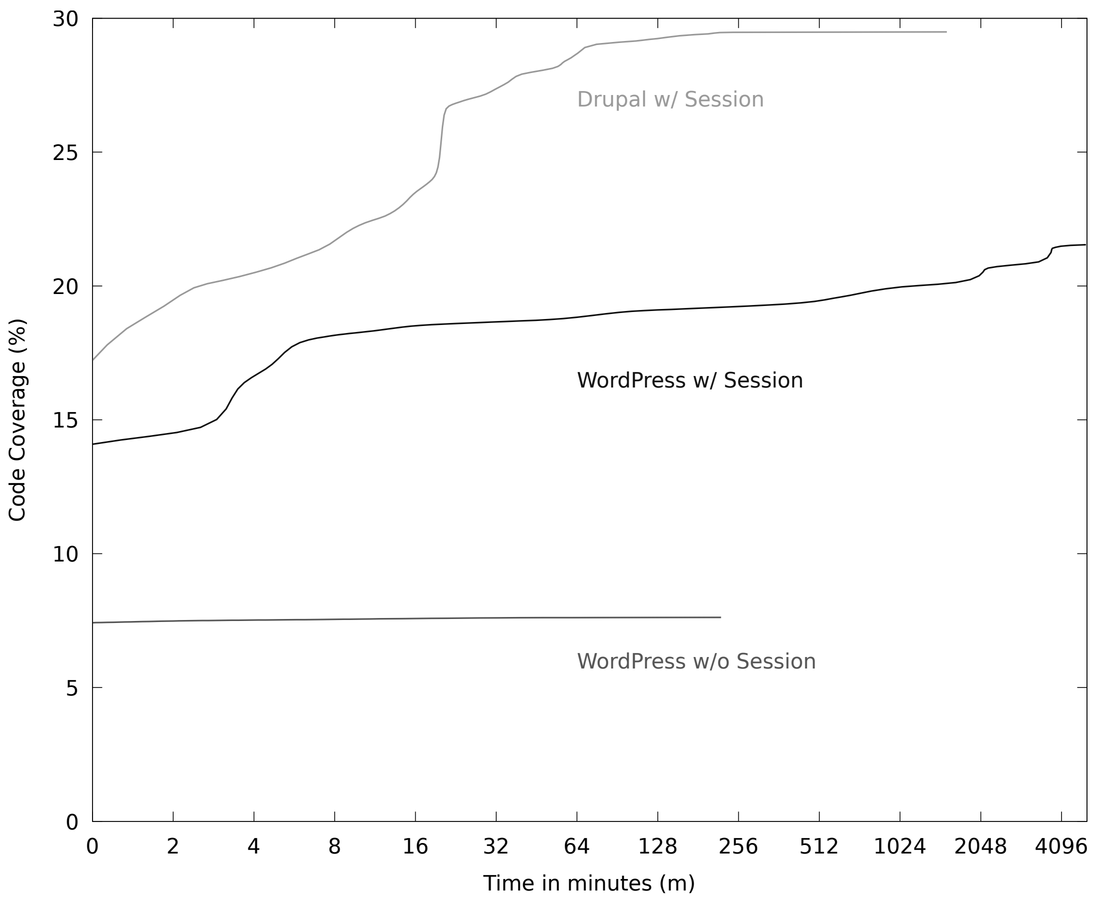


Figure 5.4: Three different execution scenarios and the accumulated code coverage gained over time in each one. Exponential increases witnessed at the start of the experiments are due to the initial exploration of the target web application's site map by the crawler.

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

In Figure 5.5 we can see the Global Code Coverage achieved by a 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 done solely to check how well a black-box fuzzer performs in terms of code coverage against a gray-box fuzzer, like webFuzz, that leverages instrumentation feedback.

When looking at both Figures 5.4 and 5.5 we can safely surmise that the instrumentation feedback provides webFuzz 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

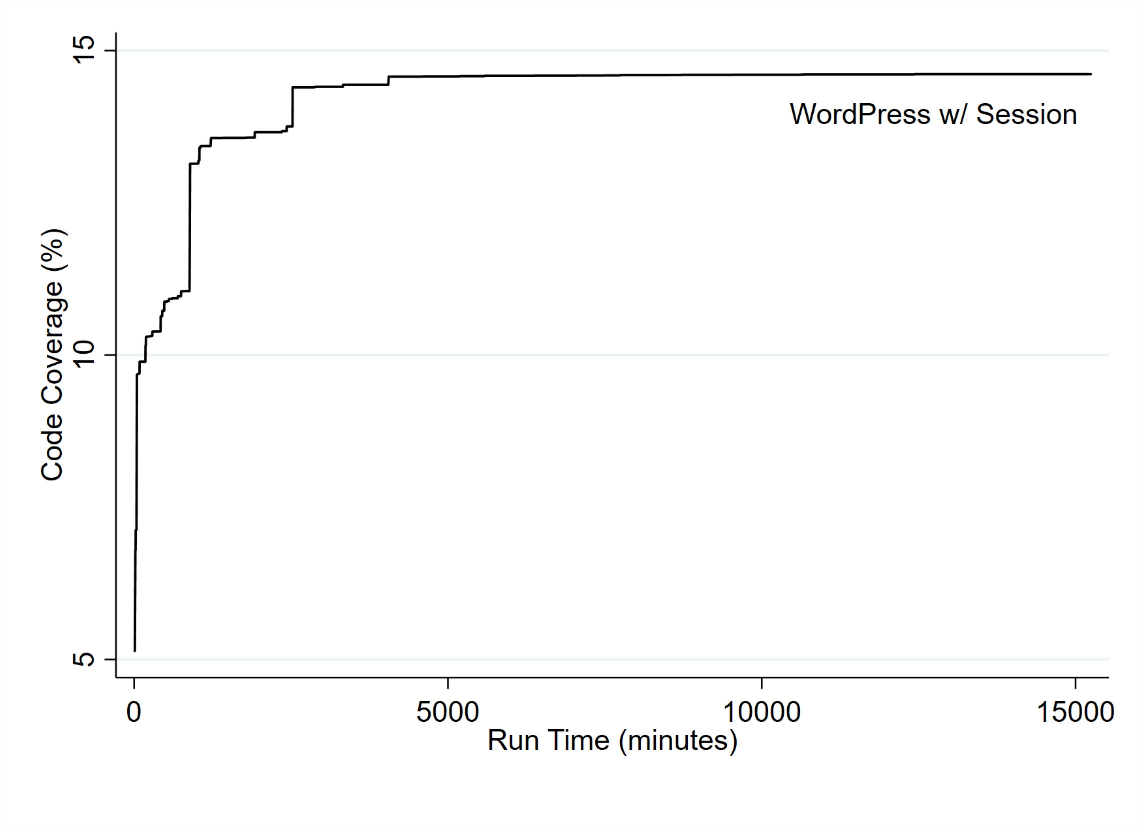


Figure 5.5: Code coverage achieved by Wfuzz over time when fuzzing WordPress using an authenticated session. After running for approximately *5000* minutes (3.5 days) the code coverage remained stagnant at *14.6%* for the rest of the experiment.

running for a much longer time than webFuzz but with no luck since it stayed stagnant on the score it achieved in 3.5 days at *14.613%*.

Chapter 6

Discussion

Contents

6.1 Limitations 39
6.2 Future Work 40

In the discussion chapter, the limitations faced - such as a lack of fuzzers to compare with - during the development of webFuzz are outlined in detail and future what plans are being considered to upgrade our fuzzing tool are also highlighted.

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 webFuzz with. After extensive research, it became clear that there are not as many black-box fuzzers available as there was a decade ago. Many older and renowned black-box fuzzers mentioned in various web sites and published papers [31, 45, 46] 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 is classified as a 'brute-forcer', by providing the aforementioned functionality we can add code and 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 can not be achieved with Burp Suite Professional or OWASP ZAP.

Additionally, during the evaluation phase, more evidence could 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 will 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. [29] 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 the time being, 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 will go undetected.

6.2 Future Work

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

There are future plans to include more functionalities in our tool kit to weed out other critical web-app vulnerabilities through our detection suite, so it can provide wider security protection that goes beyond Cross-Site Scripting. Such core vulnerabilities can be found at OWASP Top 10 [64] the most common form of bug 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 [58] - 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 can currently record. This can be achieved by taking into consideration the location of the payload in the HTML document. These types of improvement 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 has 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 excluding JavaScript. Moreover, to improve our evaluation we may adopt similar approaches that Backes et al. did [29] 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 on improving our fuzzer is that certain core functions of the fuzzer might eventually be ported to faster languages; such as C and Java, that can 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 [69], 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, and 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 [17]. Another module that could boost the performance of webFuzz is aiomultiprocess [8]. 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 and 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 these, ideas of optimization are one thing, putting them into practice is an entirely different matter. Every step has to be properly assessed and examined scientifically before they can be 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

7.1 **Generic Fuzzing** 43
7.2 **Web Applications Fuzzing** 44

In this chapter any relevant work done in the past year at research level to the field of fuzzing is stated here.

7.1 Generic Fuzzing

Fuzzing has been perceived through several techniques and algorithms over the years. Firstly, we have the black-box fuzzers [51, 71, 75] which are unaware of the fuzz target's internals and thus are trying 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 [63,68]. 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 is proven that the feedback is vital for a fuzzer's performance since it can be used to steer the fuzzer towards exploring new code paths, resulting in a 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 [48].

A renowned fuzzer that is classified as coverage-based is AFL [77]. AFL is a state-of-the-art grey-box fuzzer which is deemed the foundation for the majority of the recent proposed works. However, AFL fails to intelligently generate inputs to explore deep paths in programs that are hidden behind checksums or magic number *if* statements.

Having this in mind, recent research work 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 [72], DART [48] and SAGE [49].

In spite of all these efforts to improve the fuzzing process with the use of symbolic/-concolic execution based fuzzers, it has been noticed that this type of fuzzers suffer from scalability problems because when fuzzing sizeable targets, we notice the problem of state explosion [34]. 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.

Consequently, some other research proposals try to accomplish what symbolic/concolic execution based fuzzers offer with a less expensive approach. One example is REDQUEEN [36] that utilizes the input-to-state correspondence in order to infer the values that would be later used and try control them. Another such example is VUzzer [68], 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 have been given to build fuzzers with the aim to weed out vulnerabilities in native code, little attention has been given to web applications 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 the web application [31,45].

Such as SecuBat [54], 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 [46], 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. [44] proposed a way to navigate through a web application's states to discover whether an input is interesting by noticing the changes of the output.

As an alternative, there is the white-box approach to consider with access to the web application's source code. Kieyzun et al. [55] used a technique exploiting information about the code to automatically generate inputs that target SQLi and XSS vulnerabilities.

Moreover, Artzi et al. [28] 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. Prime examples are Pixy [53] which uses static analysis at the source code level to detect vulnerable code.

Another tool combining static and dynamic analysis is Saner [30] 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 are used to generate other test cases that will hopefully result in wider code coverage that could possibly trigger more vulnerabilities. Unlike other fuzzers mentioned who generate their own XSS payloads [46], 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 for many. While it is used extensively to uncover many important bugs and security vulnerabilities in native apps, web applications have received limited attention, so far.

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

Furthermore, for the evaluation of webFuzz we used two web applications; namely WordPress and Drupal, for the the following 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 test. More specifically it was able to weed out *30* bugs with the second on being Wfuzz with *28*.

Also, in terms of throughput, unfortunately currently 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 seems to be hefty.

Other findings suggest that our fuzzing tool can achieve coverage of the entire WordPress and Drupal code as high as *21.5%* and *30%* respectively, in a roughly 50 hours fuzzing session.

Bibliography

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

[2] payloadbox/xss-payload-list. https://github.com/payloadbox/xss-payload-list.

[3] swisskyrepo/payloadsallthethings. https://github.com/swisskyrepo/PayloadsAllTheThings.

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

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

[6] Mariadb.org foundation. https://mariadb.org/, 2019.

[7] Nginx | high performance load balancer, web server, reverse proxy. https://www.nginx.com/, 2019.

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

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

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

[11] C. zapponi - githut. http://githut.info/, 2020.

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

[13] Drupal - open source cms. https://www.drupal.org/, 2020.

[14] Mdn web docs: 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.

[15] Owasp: Fuzzing. https://owasp.org/www-community/Fuzzing, 2020.

[16] Python docs: unittest.mock — mock object library — python 3.9.1 documentation. https://docs.python.org/3/library/unittest.mock.html, 2020.

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

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

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

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

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

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

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

[24] Wordpress deployment with nginx, php-fpm and mariadb using docker compose. https://medium.com/swlh/wordpress-deployment-with-nginx-php-fpm-and-mariadb-using-docker-compose-55f 2020.

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

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

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

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

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

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

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

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

[40] P. docs. Pylint - code analysis for python | www.pylint.org. `https://www.pylint.org/`, 2020.

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

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

[48] 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.

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

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

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

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

[53] 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.

[54] 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.

[55] 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.

[56] 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.

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

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

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

[60] 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.

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

[62] 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.

[63] 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.

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

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

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

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

[68] 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.

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

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

[71] 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.

[72] 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.

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

[74] Tutorialspoint. Symbolic execution, 2020. `https://www.tutorialspoint.com/software_testing_dictionary/symbolic_execution.htm`, last accessed in November 2020.

[75] 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.

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

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

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
    restart: always
  nginx:
    image: nginx:alpine
    volumes:
      - ./data/nginx:/etc/nginx/conf.d
      - ./data/wordpress:/var/www/html
    ports:
      - 8080:80 # Host machine port 8080 mapped to the container port
        80.
    links:
      - 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 - as listed in the requirements.txt in our repository - so webFuzz can run smoothly