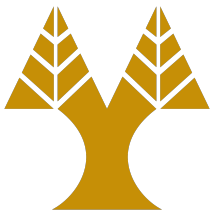
Thesis Dissertation

WEBFUZZ: IMPLEMENTATION OF A GREY-BOX FUZZING TOOL 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 Fuzzing Tool 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.

Furthermore, I would like to thank PhD. candidate Michalis Papapevripides of SREC Lab for his timely response to any issues that arose and for the assistance he provided me in resolving them.

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 the first gray-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, WordPress and Drupal.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

This the first and introductory chapter of this thesis. Here we analyse what motivated us to do research in the area of grey-box fuzzing and start this project, any related work regarding this area, the contribution that this thesis makes and the outline of the topics of the chapters included in this thesis.

1.1 Motivation

Fuzzing is now 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 fashion has been precipitated with the introduction of AFL [56], 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 possible while also being a 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 because unit tests usually revolve around expected inputs in order to test the intended functionality of code [25].

Although automated software testing has become an attractive field of research, it still has a long way to go, especially for web applications [32]. As the Internet infrastructure expands, more software that is written in native code is migrating to web applications. This attracts more malicious attacks on web applications. Hence, there is 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 various methodologies [25, 26, 36, 37, 43, 49, 52]. For example, most of the fuzzers take advantage of instrumentation on the binary or source level. That is, inserting code to the program in order to receive feedback when a code block gets triggered and try to adjust the generated inputs to improve code coverage. Others utilize symbolic/concolic execution for extracting useful information about the program and use that information for improving the input generation process [35, 36, 52]. However, all these fuzzers are currently targeted towards finding vulnerabilities in native code, while web applications have received limited attention. More related work will be see again at Chapter 7.

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 [32]. This is to say they use brute force to bombard their targets with URLs that embed known web-attack payloads. Recently there where breakthroughs with white-box fuzzing also [**?**, **?**]. On the contrary, 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 better understanding the measured capabilities of webFuzz we compare our results with three existing web-application fuzzers. webFuzz is the only fuzzer that reports coverage information; in particular, webFuzz can cover about 21.5% of the entire WordPress code, which contains around half a million LoCs, 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 [**?**] 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) for a fuzzing session that lasts 65 hours.

3. To foster further research in the field, webFuzz is release as open source.

1.4 Thesis Outline

The thesis consist of eight chapters. In the first chapter we present the incentive of this thesis, any related work on the topic and the contributions of this thesis. In the second chapter we state any relevant background information required to grasp the perspective of this thesis. 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 for discussing the technical aspects of the fuzzing tool developed. The fifth chapter consists of the evaluation to see how well webFuzz performs in terms of finding bugs, code coverage and throughput against other fuzzers. In the sixth chapter, a discussion is included that consist of the limitation faced during the process and any future plans we have for our tool. In the seventh chapter we elaborate on the related work made in the area of fuzzing over the years. In the eight and final chapter we give our conclusion.

Chapter 2

Background

Contents

In this chapter we provide background information giving a detailed understanding on various 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 elaborate on 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 available 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 ones can be the root of irreparable damage when they are exploited by attackers with malicious intentions. In fact, web application vulnerabilities account for the majority of vulnerabilities reported in the Common Vulnerabilities and Exposures database [28].

The OWASP Top 10 represents a broad consensus about the most critical security risks to web applications [44]. One of the most pressing security problems on the Internet, according to the aforementioned list, is Cross-Site Scripting, also known as XSS.

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. As a result, such untrusted data can be executed then, in turn, hijack the browser, deface the web site, redirect the user to dangerous sites and many other attacks. Some XSS types include Reflected(aka Non-Persistent or Type II), Stored (Persistent or Type I) and DOM-based(Type-0).

Reflected XSS [21] vulnerabilities arise when arbitrary data is copied from a request and echoed into the application's immediate response. This way, scripting language code included within a request can be dynamically executed. In the case of Stored XSS vulnerabilities, the malicious payload is first permanently stored in storage such a database residing on a server, and is only later outputted by an unsuspecting query. Examples might be Web forums or blog comments.

webFuzz focuses in detecting both bugs that can lead to both Reflected or Stored Cross-Site Scripting, which are among the most common of XSS attacks. An illustration of the latter can be seen at Figure 2.1 and of the former see Figure 2.2. In both the 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. Most of the time, 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.

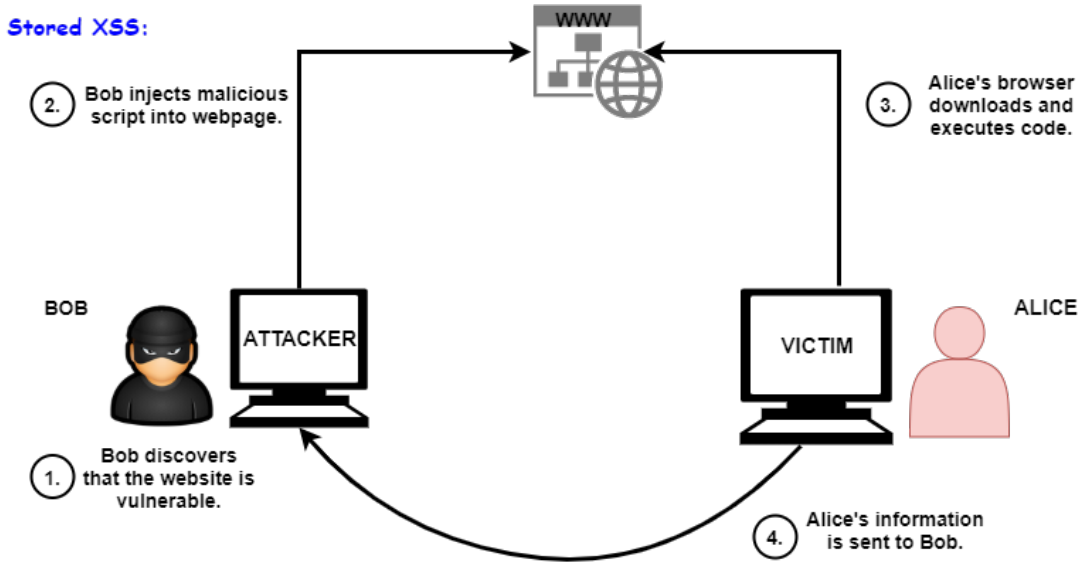


Figure 2.1: How Stored Cross-Site Scripting can be exploited by an attacker

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 tainted variable that has not been sanitized, an attacker can inject malicious payload in this field which will be interpreted by the HTML parser according

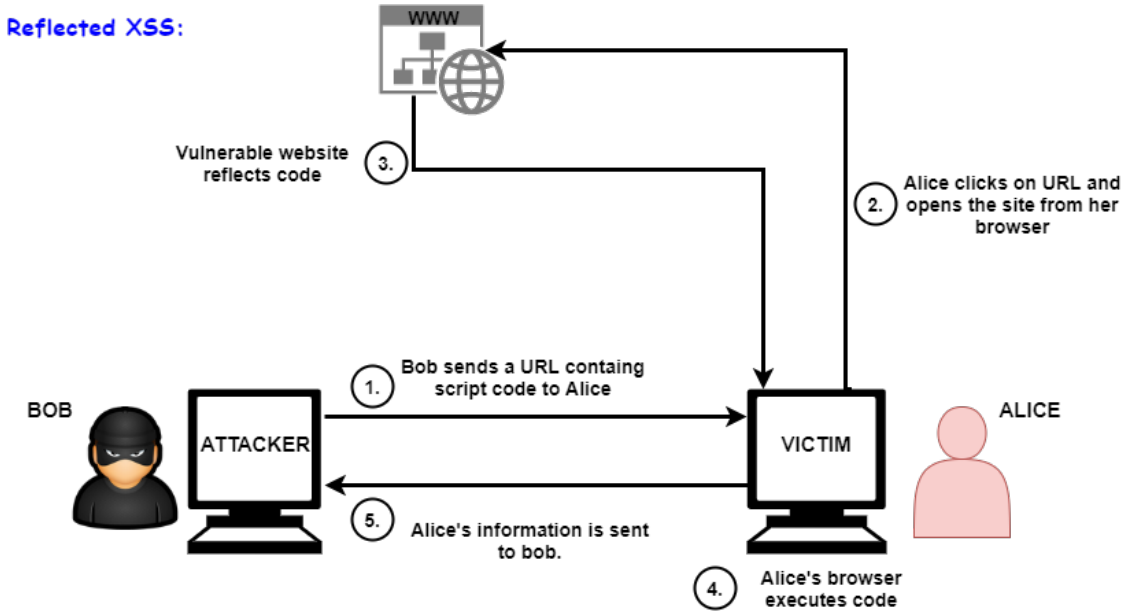


Figure 2.2: How Reflected Cross-Site Scripting can be exploited by an attacker

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.

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, systemwide measures such as Content Security Policy(CSP) [12] may be enabled to eliminate or mitigate XSS attacks. Nevertheless, flaws such as Buffer Overflows 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) [41]. The technique was developed by Barton Miller et al. at the University of Wisconsin. With this quality assurance technique, software is exercised using a vast number of anomalous inputs for inferring if any of them introduces 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 [53].

A black-box fuzzer treats the program as a black box and is unaware of internal program structure. It conducts its test on the target through external interfaces and produces random inputs using no information of the target's underlying structure. Hence, black-box fuzzers are only able to scratch the surface usually and expose "shallow" bugs [13].

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 [45]. They may also leverage symbolic execution in order to derive what inputs cause each part of a program to execute [54]. Therefore, they can be 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 [**?**, 56]. This thesis explores in detail gray-box fuzzing, which is a combination of both the white-box and black-box approaches since it uses the internals of the software at a certain extend to assist in generating better test cases but still does not have full access to the code. Also the feasibility for constructing a fuzzing tool that will automate the process of discovering bugs in web applications is explored. This is succeeded by providing randomized invalid inputs to an under-analysis instrumented web application, mutating these inputs according to the feedback received and finding test cases that

cause a crash or make them act inappropriately to better ensure the absence of 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 about the correlation between code coverage and the number of bugs found [38, 39]. 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 [24] does. Therefore code coverage should be considered a secondary metric and the number of bugs found as primary [39]. Nevertheless, measuring coverage is important for any fuzzer.

Currently, available fuzzers for web applications act in a black-box fashion [32]; they just use brute force at 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.

On the other hand, 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 calculate code coverage we simply 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 informations such as combinations of basic blocks that were visited one after the other.

We instrument web applications for delivering feedback once they are fuzzed. As opposed to native applications, where several options exist for instrumenting their source or binary representation, we decide to instrument web applications by modifying the Abstract Syntax Tree (AST) of PHP files and then reverting it back to source code form. This, in turn, provides us feedback on the basic blocks that are visited during analysis. For altering the AST of PHP files, PHP-Parser [46] is used. Instrumentation performed by webFuzz on our targeted web application is similar to how AFL instruments binaries, but adapted to work in web applications. A more elaborate approach of the instrumenting functionality provided by webFuzz is beyond the scopes of this thesis.

2.4 Concurrency

Concurrency is defined as working on multiple tasks at the same time [17]. 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 right from the point where it left off.

Concurrency aims to speed up the overall performance of input/output (I/O) bound problems, 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 at Figure 2.3:

More specifically in Python, concurrency can be expressed either through the Threading or AsyncIO(short for Asynchronous Input Output) [16] modules. Due to the infamous Global Interpreter Lock (GIL) [18] Python has, both AsyncIO and Threading are single-threaded, single-process design. Thus, there was no clear advantage of using the latter

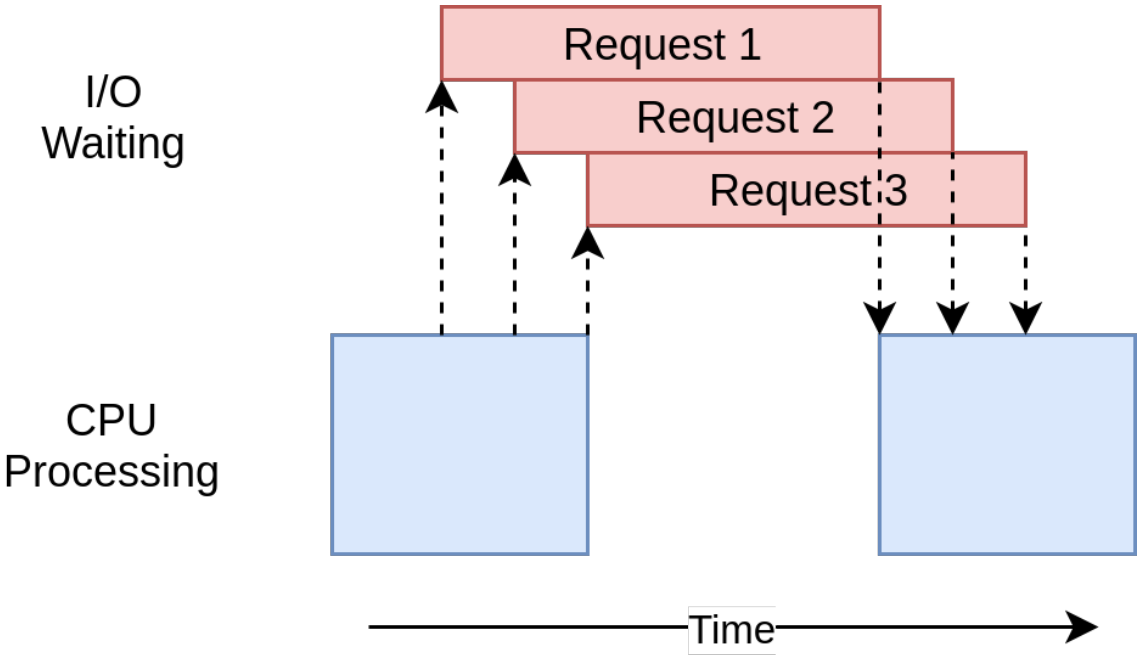


Figure 2.3: Requests over the internet processed in concurrent fashion [17].

so AsyncIO was opted for instead, although the initial plan was to use threading. 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 info 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 of creating software locally with the knowledge that it will run identically regardless of the host environment [42]. Containers are an encapsulation of an application's dependencies that share resources with the host OS, unlike Virtual Machines. During the evaluation, which can be seen detailed in Chapter 5, a docker-compose YAML file was created to allow multiple containers to be initiated and managed at once 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 [10].

Chapter 3

Architecture

Contents

This chapter illustrates the general design of the fuzzer without going in to too much technical detail. The in-depth breakdown of the fuzzer's components is thoroughly described in Chapter 4. The key components that are 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 it is designed to detect.

3.1 A Fuzzing Session

webFuzz constitutes of two intertwined components that work together in providing a guided fuzzing approach with the goal 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 as to how AFL instruments binaries but these are adapted to work in web applications.

The second component is the fuzzing application with all its core functionalities is re-

sponsible 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 in order 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. It displays the process from the point the request is sent up to the point 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.

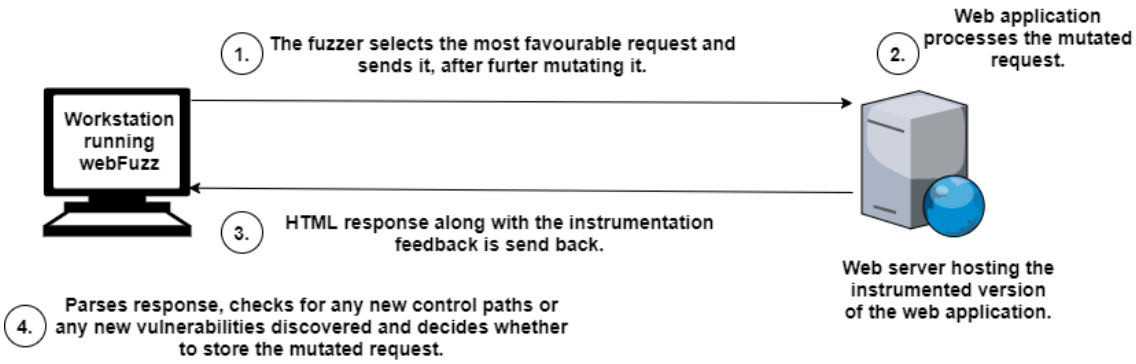


Figure 3.1: High-level overview of a fuzzing session using webFuzz

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 [**?**] and AFL [56] 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 an essential 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 [51]. Choosing which mutation function to use in order 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 too aggressive tweaks can destroy much of the input data structure and lead to the test cases failing at an early stage of the execution [55].

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 it employs 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 too.

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 [33], 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 [2,3,40]. Such payloads can further mutate by prepending or appending to them random strings or specific HTML, JavaScript and PHP syntax tokens.

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.

Using evolutionary algorithms in the test case creation process is widely practised in fuzzers to optimize solution searching [51], webFuzz will also mix GET or POST pa-

rameters from various favourable requests to generate new inputs. Opposite 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 conduct 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 it was not. 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 **??**.

Chapter 4

Implementation

Contents

This chapter is dedicated to discussing 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(known as the creator of the Python programming language) said; "Code is read much more often than it is written". For this reason, throughout the duration of this thesis, our main aim was to write clear, readable and eye-pleasing code by following best practice that most 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 [48] coding style standard and, regarding documentation, the PEP 257 [47] and Sphinx [19] docstring conventions where used so it will be clear and easy to read for programmers. Pylint [29] was also used to check for errors in Python code and try to implement the aforementioned coding standards and search for code smells.

To bolster the good practises mentioned, unit tests were also 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 [30], unittest [31] and mock [14]. 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 [16] Python module. In our case, asyncio has made it possible to send, continuously, HTTP requests to the target website while at the same time various statistics regarding the fuzzing session are printed on the user's screen and a respective log file is updated. With the 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, have been 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 meantime. This process is called cooperative multitasking and although it involves doing extra

work up front, the benefit is that you always know where your task will be swapped out, thus optimising to yield better performance.

In a brief summary, 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, and the waiting state means that 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 never get interrupted in the middle of an operation. A more elaborate depiction of the asynchronous process executed by asyncio can be viewed in Figure 4.1.

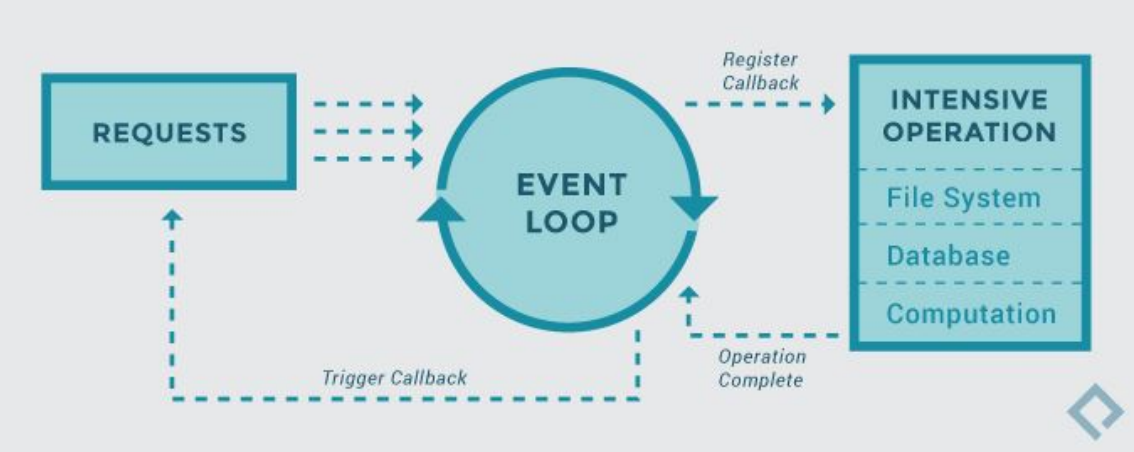


Figure 4.1: AsyncIO mechanism; it provides a high-performance asynchronous frameworks for making our fuzzing requests [34].

Communication with the target's web site is achieved with a rapidly fast asynchronous http client/server framework named aiohttp [23]. 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 dictates that Python does not like the await statement. However, more modules are becoming compatible with asyncio [23].

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` [27] 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 store 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 requests so far. 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 [20] Python module. This module, more precisely the urlparse method, is used for breaking the Uni-

form Resource Locator (URL) string up in 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 gray-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.

As you can imagine, 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 crush.

There are of-course ways to run webFuzz without this interface which will be explained in the next subsection. 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

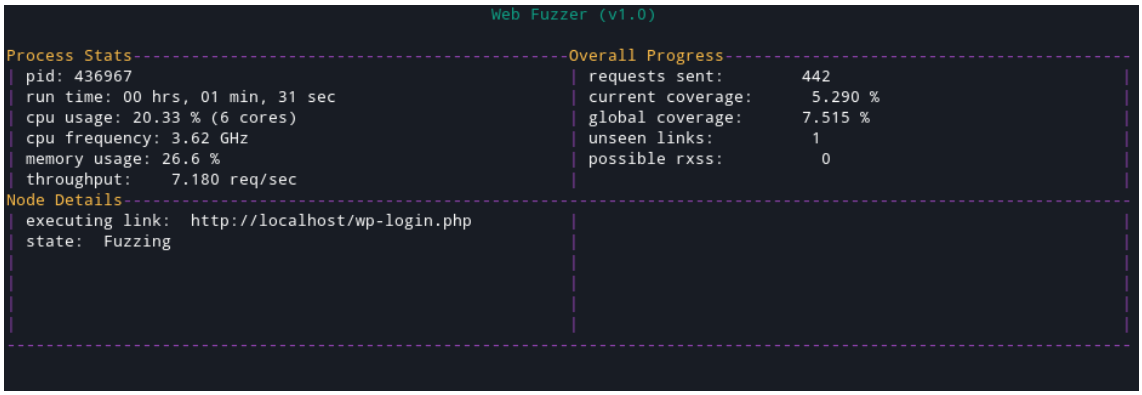


Figure 4.2: A screenshot of the webFuzz interface. The interface is implemented using the Curses module.

4.5 Running webFuzz

Running webFuzz is fairly simple. 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 positional 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.

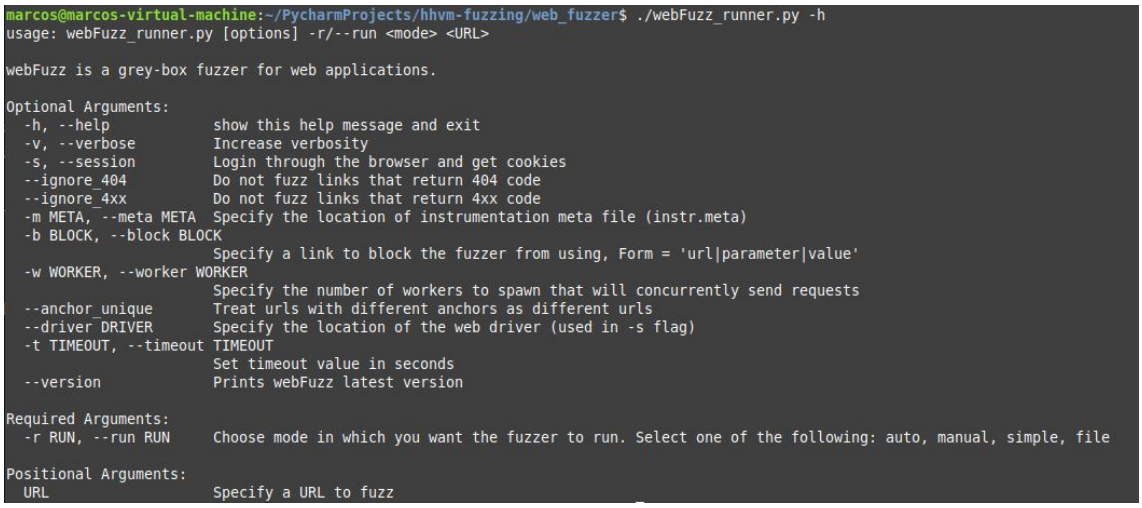


Figure 4.3: webFuzz help menu includes all available arguments which we can use to run it with.

Chapter 5

Evaluation

Contents

5.1 Methodology

For the evaluation of our tool, we opted for convenience to use Docker [5], which we also discussed in Chapter 2. Docker is a 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 [**?**]. For this reasons, we opted to evaluate our tool on web-apps developed in PHP.

The first web application we test our tool on is 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 is more important about it, 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 web sites.

We tested our tool on second web application, Drupal CMS [11]. 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 achieved 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 in the case of WordPress, can be viewed at Figure 5.1. Files and instructions for replicating this process can be found at the fuzzer's repository.

5.2 Automated Vulnerability Addition

Evaluating fuzzing processes has proven to be a challenging task [39]. Migrating known vulnerabilities to existing software, in order to test the capabilities of the fuzzer in finding

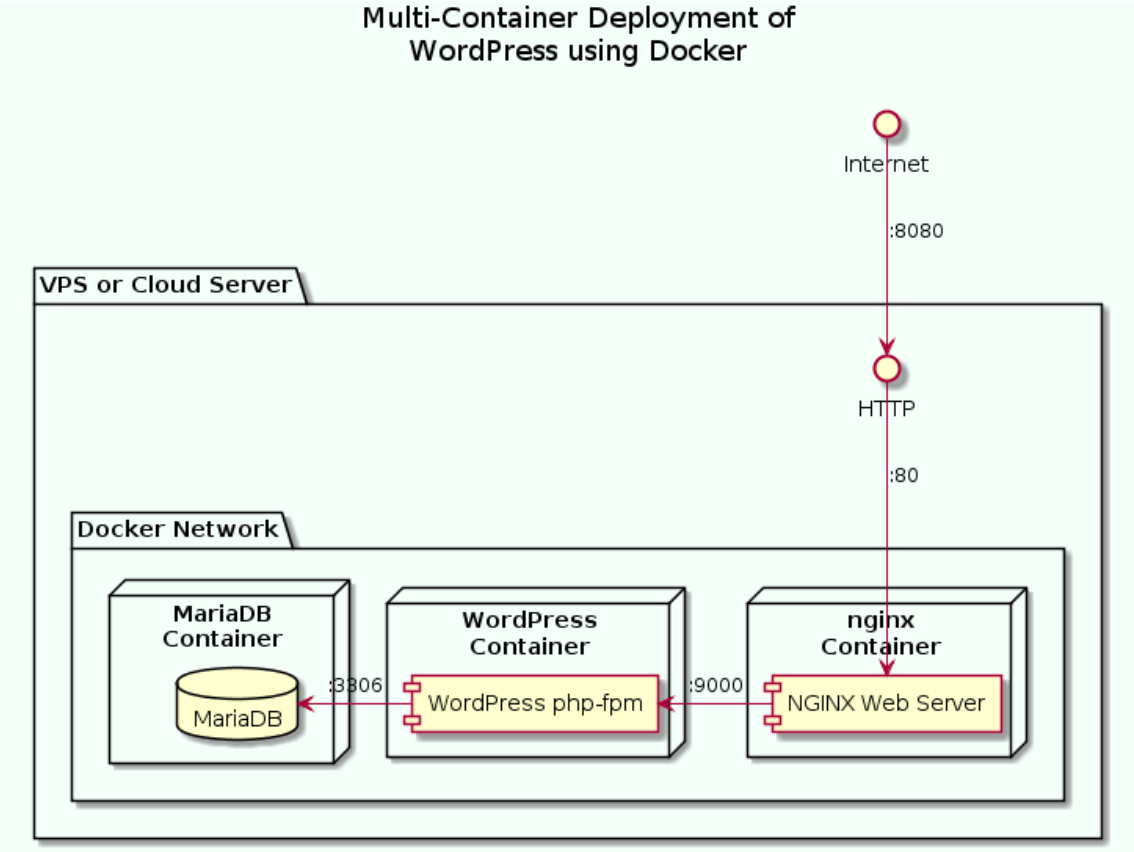


Figure 5.1: Evaluation followed the above Multi-Container Deployment of WordPress using Docker [22].

bugs, can be a tedious process [**?**]. Thus, for evaluating webFuzz, but also other fuzzers for web applications, an automated bug injection methodology, inspired by LAVA [**?**] was used for automatically injecting bugs in web applications written in PHP. Injecting vulnerabilities in web code, again, is challenging, since important tools used for analysing native code and injecting vulnerabilities (e.g., taint-tracking and information-flow frameworks), are not available for web applications. To overcome this lack of available tools, the vulnerability injection methodology leverages the instrumentation infrastructure we use 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 automated vulnerability addition, like the instrumentation, are out of 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 quad-core Intel® Xeon® W-2104 Processor @3.20 GHz 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 customizations or frameworks added to them. All artificial bugs where 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 is deployed and running the web applications.

Chapter 6

Discussion

Contents

In the discussion chapter, any limitations faced during the development of webFuzz are stated and plans that we have for the future of our fuzzing tool.

6.1 Limitations

During the development of webFuzz we faced some obstacles that we discuss below.

Concerning the choice of Wfuzz as the main fuzzer to compare webFuzz with, after extensive research, it became apparent that there are not as many black-box fuzzers around as there used to be a decade ago. Many older and renowned black-box fuzzers mentioned in various web sites and published papers [**?**, 32, 33] have either ceased to exist or are no longer developed and maintained. We have found that Wfuzz is the only fuzzer that could easily be extended to our needs without requiring extensive research and training.

In addition, during the evaluation phase, more evidence can be submitted to explore deeper the potential of webFuzz in detecting vulnerabilities. For instance, the tool can be evaluated on more open-source projects written in PHP and on more complex real-world XSS vulnerabilities, that will reflect the real-world scenarios. Such large-scale analysis of

web application code to find real-world XSS bugs where proposed by Backes et al. [**?**].

At the time being webFuzz's detection suite is limited to Reflected and Stored Cross-Site Scripting. DOMbased XSS vulnerabilities that rely on the browser's JavaScript runtime context, are so far out the fuzzer's scope. This kind of attacks require no interaction with the server, and succeed when the JS code does not sanitize the user input before rendering it (e.g., using the `innerHTML` property). For detecting these, we would have to render the HTML and run the JavaScript code of each request. This would in turn severely degrade the fuzzer's throughput, that is why it has been avoided for the initial version of webFuzz. Unfortunately, by the complete exclusion of JavaScript many potential XSS vulnerabilities are missed.

Moreover, as I had no prior experience with fuzzing and the different categories that constitute it, I devoted much time at the start of thesis to educated my self at the work being done at research level in the field.

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 [44] the most common form of bug in wed applications is Injection and Broken Authentication. Injection flaws, for instance, 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 [40] - 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 works have used techniques such as analysis of JavaScript code or Selenium-based crawlers in order to include the JavaScript-generated request URLs in their analysis. We can adopt similar approaches since we are currently missing many bugs be excluding JavaScript.

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 [50], 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 [15]. 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 this, 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 the analysis of algorithms.

Chapter 7

Related Work

The tool's main objective is finding trigger points on the target web application and supplying them with known XSS payloads instead of generating our own XSS payloads like others fuzzers do [33].

Chapter 8

Conclusion

Fuzz testing is a promising technology that has been used to uncover many important bugs and security vulnerabilities. This promise has prompted a growing number of researchers to develop new fuzz testing algorithms.

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

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

[12] 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.

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

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

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

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

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

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

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

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

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

[22] 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.

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

[24] 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.

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

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

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

[28] CVE. Common vulnerabilities and exposures (cve). `https://cve.mitre.org/`, 2020.

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

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

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

[32] 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.

[33] 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.

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

[35] 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.

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

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

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

[39] 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.

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

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

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

[43] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2289–2306. USENIX Association, Aug. 2020.

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

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

[46] N. Popov. Php parser. `https://github.com/nikic/PHP-Parser`.

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

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

[49] 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.

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

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

[52] 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.

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

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

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

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

Appendix A

Appendix B