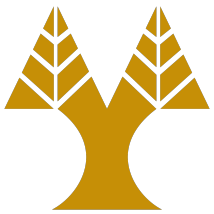
Thesis Dissertation

WEBFUZZ: IMPLEMENTATION OF A GRAY-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 Gray-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

A highly automated testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities. The name comes from modem applications' tendency to fail due to random input caused by line noise on fuzzy telephone lines.A highly automated testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities. The name comes from modem applications' tendency to fail due to random input caused by line noise on fuzzy telephone lines.

Numerous fuzzers have been developed in the past few years that try to optimize the fuzzing process by proposing various methodologies [7, 8, 20, 21, 35, 42, 46]. For instance, most of the fuzzers take advantage of instrumentation on the source or binary 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 concolic/symbolic execution in order to extract useful information about the program and

use that information for improving the input generation process [20, 21, 46]. However, all these fuzzers are currently targeted towards finding vulnerabilities in native code, while web applications have received limited attention.

mutation-based fuzzer, might actively see the code paths executed in the target and make adjustments accordingly, which is very smart. EFS and AFL do exactly this

1.1 Motivation

1.2 Related Work

1.3 Contributions

1.4 Outline Contents

Chapter 2

Background

Contents

In this chapter we provide background information, to give 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 [21]. The OWASP Top 10 represents a broad consensus about the most critical security risks to web applications [29]. 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 [17] 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 **??** and of the former see Figure **??**. 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.

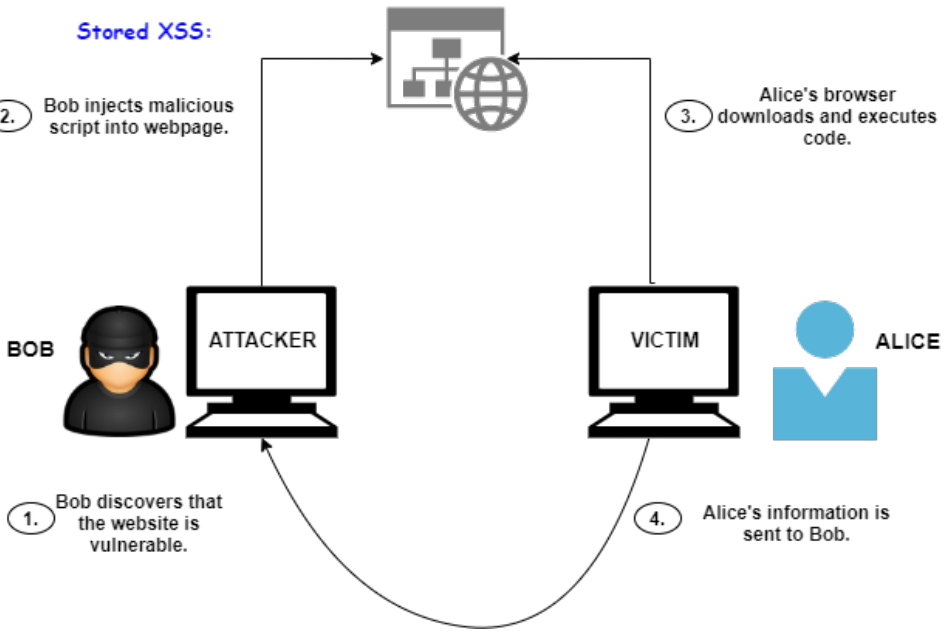


Figure 2.1: How Stored Cross-Site Scripting can be exploited by an attacker

The code above is faulty for two reasons. First, letting the user know that the username exists can help an attacker 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 freely be interpreted by

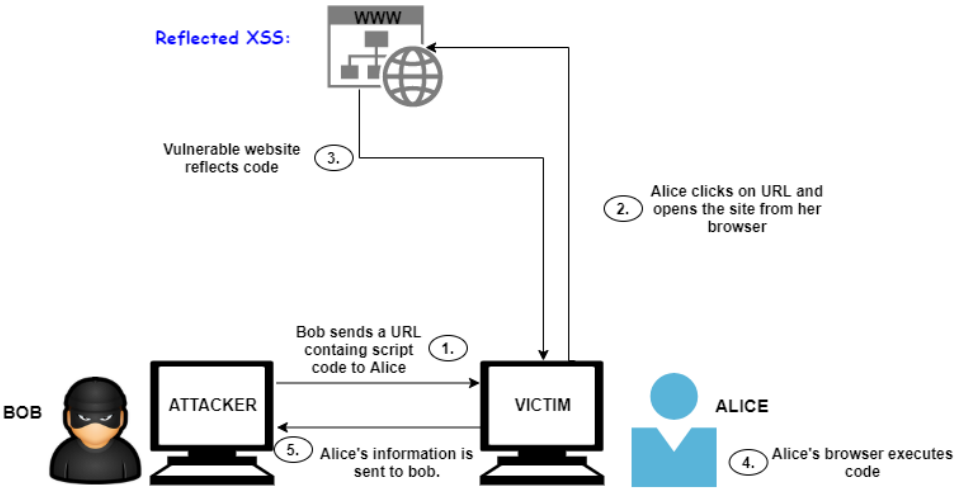


Figure 2.2: How Reflected Cross-Site Scripting can be exploited by an attacker

the HTML parser according to whatever its content is. `Exploit:` A victim is fooled 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 similar to defending against other types of code injection. The input must be sanitized. User input containing HTTP code needs to be escaped or encoded in order to avoid its execution. Also, systemwide measures such as Content Security Policy(CSP) [9] may be set as well 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) [27]. 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 [34].

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 [10]. 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 [30]. They may also leverage symbolic execution in order to derive what inputs cause each part of a program to execute [35]. 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. 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 to assist in generating better test cases.

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 [**?**, **?**]. Aiming at increasing the global

code coverage may be less effective in finding new bugs than say focusing on increasing the code coverage in particular error prone code areas as AFLGo [**?**] does. It should therefore be regarded as a secondary metric with number of bugs found as primary [**?**]. Nevertheless, measuring coverage is important for any fuzzer.

Currently, available fuzzers for web applications act in a black-box fashion [25]; 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 fair 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 [31] 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 [13]. 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:

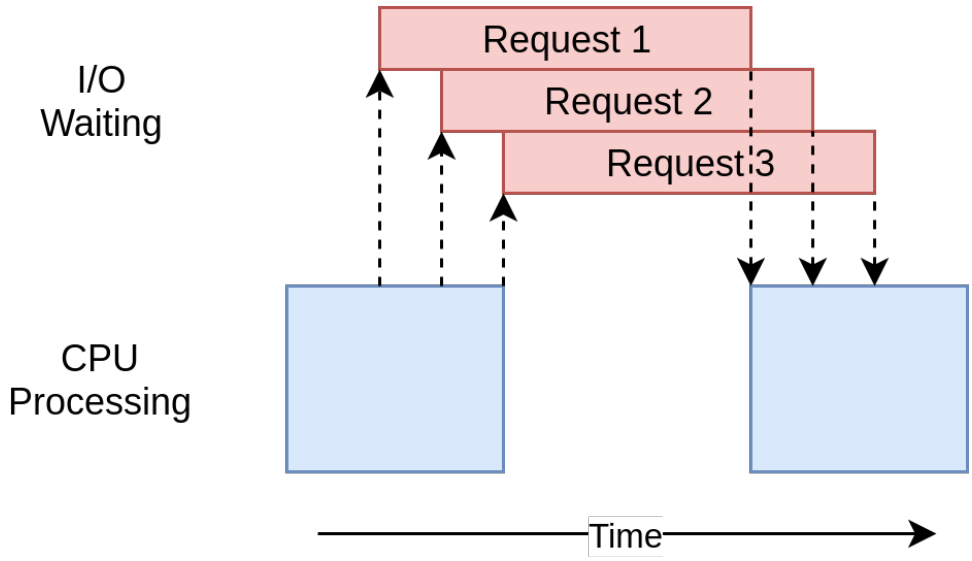


Figure 2.3: Requests over the internet processed in concurrent fashion [13].

More specifically in Python, concurrency can be expressed either through the Threading or AsyncIO(short for Asynchronous Input Output) [12] modules. Due to the infamous Global Interpreter Lock (GIL) [14] Python has, both AsyncIO and Threading are single-threaded, single-process design. Thus, there was no clear advantage of using the latter so AsyncIO was opted for instead. 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 Section 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 [28]. Containers are an encapsulation of an application with its dependencies that share resources with the host OS, unlike Virtual Machines. During the evaluation, which can be seen detailed in Section 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 [7].

Chapter 3

Architecture

Contents

This chapter illustrates the general design of the fuzzer without getting into too much implementation details. The in-depth breakdown of the fuzzer's components is more thoroughly described in Chapter 4. The key components that will be elaborated here 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 of two intertwined components that work together towards 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 what 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 they are adapted to work in web applications. The second component is the fuzzing application with all its core functionalities which is 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 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 at Figure 3. It displays the process from the point the request is sent up to the point where a response is received. A request may be produce in one of two ways; it can be 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 triggered, coverage score, and record newly discovered links.

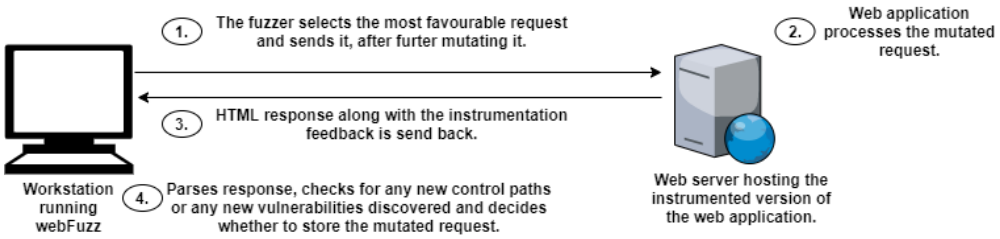


Figure 3.1: Disentangled fuzzing session process using webFuzz

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. 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 in order to avoid stagnation on a suboptimal plateau in the search space [**?**]. 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 to conservative, only a 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 [**?**].

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 certain areas of the program to execute. Regarding the known XSS payloads, unlike many fuzzers that employ malicious payload generation via the use of a specific grammar, webFuzz chooses randomly from a corpus that consists of real-life XSS payloads. The corpus was created with payloads that where found scattered all over the internet, in open-source repositories mainly [**?**, **?**]. Such payloads may get further mutated 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 [**?**], webFuzz will also mix GET or POST parameters 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. In order to detect the aforementioned vulnerabilities, we do a string-matching for the injected, possibly malicious, payload in the returned HTML response. This method may be the most efficient in terms of speed, however, it can result in high number 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. One example of this would be if 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 are discussed in Chapter 7.

Chapter 4

Implementation

Contents

In this section, we discuss the technical aspects, some of the key characteristics that constitute webFuzz and the way we deployed our WordPress application using containers to evaluate our fuzzer's performance.

4.1 Coding Standards

As 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 course of this thesis, one of our aims was to write clear, readable and eye-pleasing code by following various standard that most professional tools adhere to. For this, we enforced the latest conventions, as recommended from the Python community, in order 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 [33] coding style standard and, regarding documentation, the PEP 257

[32] and Sphinx [15] docstring conventions where used so it will be clear and easy to read from programmers. Pylint [22] was also used to check for errors in Python code and try to enforce the aforementioned coding standards and look for code smells.

Furthermore, to add on the good practises mentioned, unit tests where also created through which individual modules of the tool's source code are put under various tests to determine a particular unit's correctness and whether they are fit for use. More precisely, parts of the application's code are validated by using test cases that stress test the tool and ascertain the quality of your code by checking it against the expected response. For this part, popular python test frameworks where used like pytest [23], unittest [24] and mock [11]. 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 [12] 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 being updated. With aid 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 and 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 we optimise 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 gets run. Each task can either be in

ready state, which states 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 starts 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 again 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.

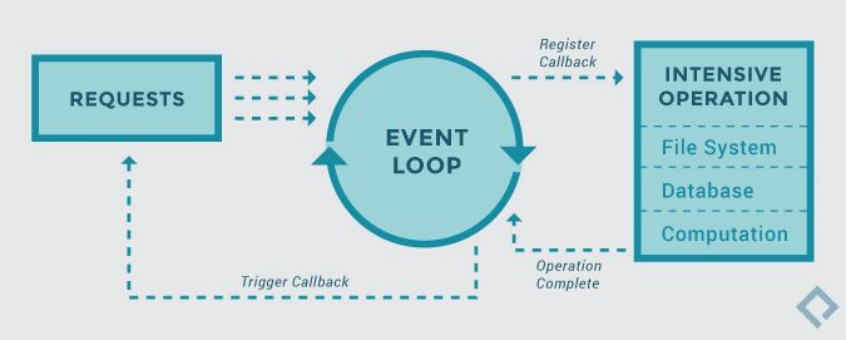


Figure 4.1: AsyncIO mechanism; it provides a high-performance asynchronous frameworks for making our fuzzing requests [26].

Communication with the target's web site is achieved with a blazingly fast asynchronous http client/server framework named aiohttp [19]. 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. The aiohttp paired with asyncio really speed things up.

It is important to note here that note all available Python modules are compatible with asyncio. For instance, for our requests, we could not use the default and recommended Python's 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 means Python would not like the await statement. However, more and more mod-

ules are becoming compatible with asyncio [19].

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 was made. Each response contains the HTML document which is then parsed using the `Beautiful Soup` [20] module in order to extract the form and anchor elements from it. These elements are useful as they can provide us with new URLs which translates to potentially new code paths and bugs to further explore and find. When found, they are added to the crawler's pending request list, and if they happen to be 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 lead 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 we have injected in all requests so far. A high-level pseudocode for the parsing process can be seen at Algorithm 1. As the pseudocode shows clearly, parsing relies heavily on the urllib.parse [16] Python module. This module, and more precisely the urlparse method, is used for breaking the Uniform Resource Locator (URL) string up in components; such as addressing scheme, network location, path etc. An object is return that contains 6-item tuple with all the URL sub-fields. The reverse can also be achieve through the urlunparse method; a URL object can be converted into string.

4.4 Curses Interface

An textual user interfaces (TUI) for webFuzz has been created using the curses module which contains various information and essential statistics, gathered while our gray-box

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

fuzzer is running. The curses library supplies a terminal-independent screen-painting and keyboard-handling facility for text-based terminals [2], 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 elaborated in the next subsection. Although many may think this is an obsolete technology, it can prove to be quite valuable for Unix-based operating systems that do not provide any graphical support. The Python module, which is the one we utilise, 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 at Figure 4.2. As the figure illustrates, statistics are divided in to three categories; namely the process statistics, the overall progress and the examining node details. As the fuzzing tools expands, more and more valuable information will be included on the interface.

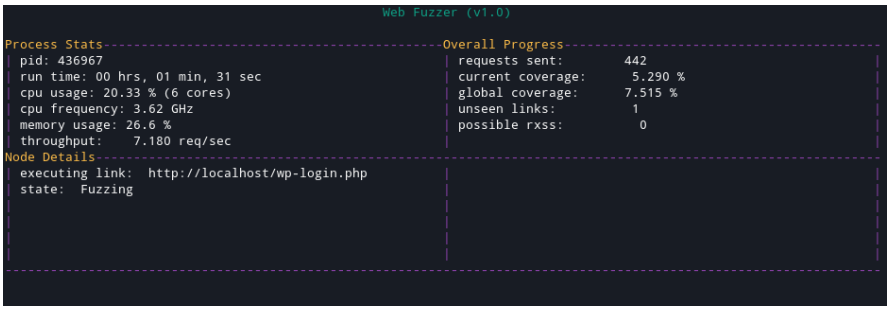


Figure 4.2: A screenshot of the webFuzz interface. The interface is implemented using the Curses module.

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, are listed in a "requirements" text file. Other dependencies, executing and installing dependencies 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 at 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 [6] 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.

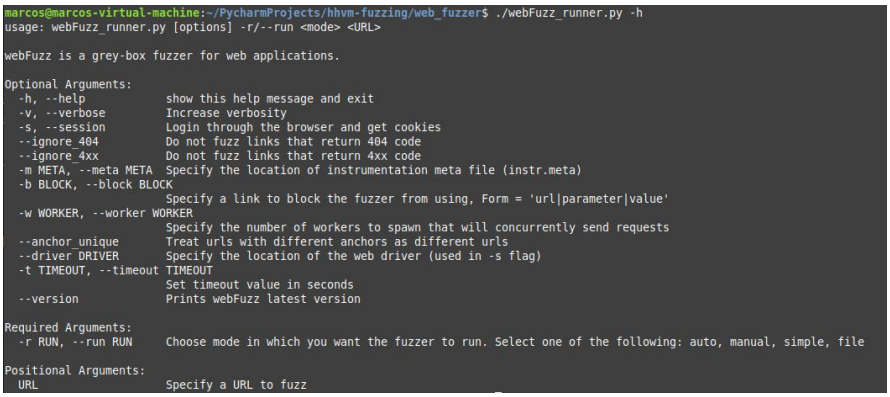


Figure 4.3: webFuzz help menu includes all available arguments which we can use to run it with.

Chapter 5

Evaluation

Contents

5.1 Methodology

For the evaluation of our tool, we opted for convenience to use Docker [3], 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 due to the instrumentation. What are the best web applications to evaluate our tool on?

The WordPress CMS (Content Management System) [3] 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 [8]. 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. [5]
- `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 [4].

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

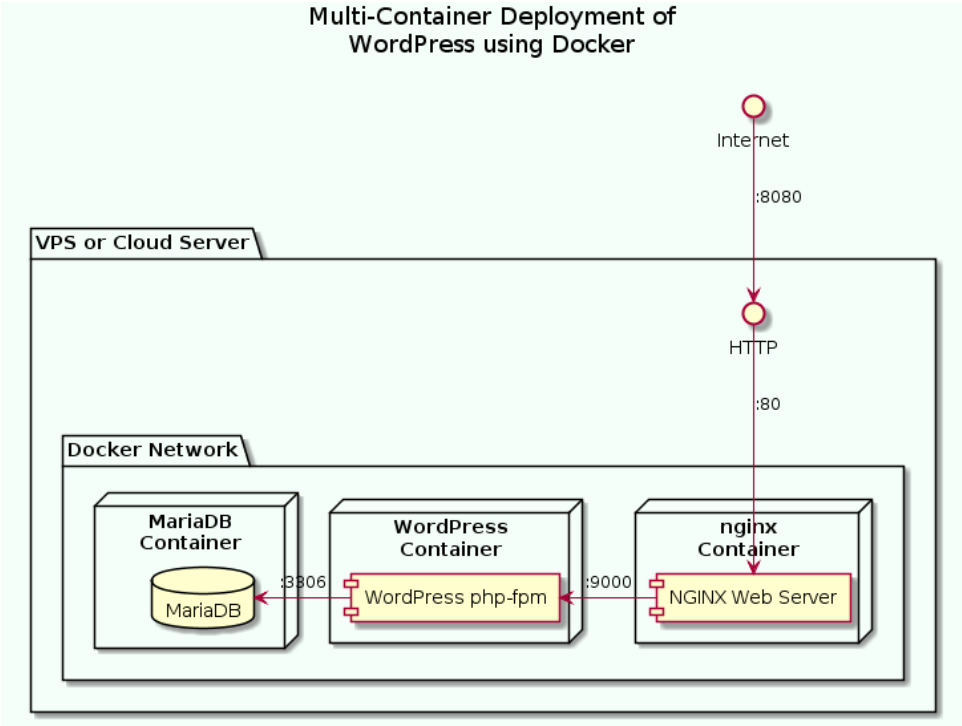


Figure 5.1: Evaluation followed the above Multi-Container Deployment of WordPress using Docker [18].

Chapter 6

Related Work

examples of concurrency in this article run only on a single CPU or core in your computer. The reasons for this have to do with the current design of CPython and something called the Global Interpreter Lock, or GIL. SAY ABOUT aimultiprocessing... Hold out on adding concurrency until you have a known performance issue and then determine which type of concurrency you need. As Donald Knuth has said, “Premature optimization is the root of all evil (or at least most of it) in programming.” Parallelism consists of performing multiple operations at the same time. Multiprocessing is a means to effect parallelism, and it entails spreading tasks over a computer’s central processing units (CPUs, or cores).

Chapter 7

Future Work

examples of concurrency in this article run only on a single CPU or core in your computer. The reasons for this have to do with the current design of CPython and something called the Global Interpreter Lock, or GIL. SAY ABOUT aimultiprocessing... Hold out on adding concurrency until you have a known performance issue and then determine which type of concurrency you need. As Donald Knuth has said, “Premature optimization is the root of all evil (or at least most of it) in programming.” Parallelism consists of performing multiple operations at the same time. Multiprocessing is a means to effect parallelism, and it entails spreading tasks over a computer’s central processing units (CPUs, or cores).

better string matching algorithm

Chapter 8

Conclusion

Contents

8.1 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.

8.2 Future Work

As Donald Knuth has said, "Premature optimization is the root of all evil (or at least most of it) in programming."

Bibliography

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

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

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

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

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

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

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

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

[9] 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.

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

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

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

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

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

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

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

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

[18] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix A

Appendix B