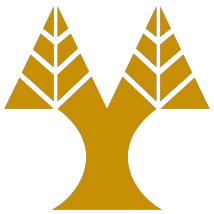
Thesis Dissertation

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 my gratitude to my Thesis Supervising Professor Dr. Elias Athanasopoulos for his valuable guidance, encouragement and advices he provided me over the course of accomplishing my dissertation. During the past one year, Dr. Athanasopoulos interest, excitement and refined knowledge in the field of Cybersecurity has been undoubtedly a source of inspiration for me. All these have made my 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 to a greater project that consists of each ones thesis.

In addition, I would like to thank PhD. candidate Michalis Papapevripides of SREC Lab for its continuous response to any issues that arose and for the assistance it provided me in resolving them.

Furthermore, I would like to thank all my professors from whom I received invaluable knowledge and helped me to become a Computer Scientist during 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 life and supporting me at every step.

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 grey box fuzzing of web applications and the construction of a fuzzing tool that will automate 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 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 section we provide background information, to give a detailed understanding about various key points regarding this thesis. First, we define what a Cross-Site Scripting bug is in web applications, and elaborate on an example regarding this vulnerability. Then, we briefly discuss what fuzzing is and the various categories that constitute it and continue on how instrumentation helps when used during gray box fuzzing. After, we continue discussing the concept of concurrency in Python and we close with the containerization of services using Docker.

2.1 Web Application Bugs

The internet has been growing exponentially since its inception. Although there are over 1 billion pages currently on-line, writing a web application that it is secure from any available vulnerability, can be extremely hard. Every significant web application, especially large-scale 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 the vulnerabilities reported in the Common Vulnerabilities and Exposures database [9]. The OWASP Top 10 represents a broad consensus about the most critical security risks to web applications [16]. 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 these untrusted data can get executed which can 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 [7] vulnerabilities arise when 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. Currently webFuzz focuses in detecting bugs that can lead to Reflected Cross-Site Scripting, which is among the most common of XSS attacks.

It is imperative that we understand what an RXSS (reflected XSS) bug typically looks like, in order to grasp the thesis' perspective. 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.

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 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) [1] 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 technique for discovering unknown vulnerabilities in programs and web applications proven to be very effective, is a technique called fuzzing [14]. With this quality assurance technique, software is exercised using a vast amount 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 [21].

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. [2] 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 [17]. They may also leverage symbolic execution in order to derive what inputs cause each part of a program to execute [22]. 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. In this thesis we explore 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 increasing code coverage increases the

changes of executing unsafe pieces of code where bugs may reside. As we mentioned in the previous section, using instrumentation may be the key yield a higher code coverage percentage. Currently available fuzzers for web applications act in a blackbox fashion [16](FIX THIS REFERENCE FROM PAPER); they just brute force 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 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 in order to realize any interesting requests that would later help to improve the code coverage and as a result, trigger vulnerabilities nested deep in the web application's code.

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 [18] 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. An elaborated approach of the instrumenting functionality provided by webFuzz is out of the scopes of this thesis.

2.4 Concurrency

Concurrency is defined as working on multiple things at the same time [5]. 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 is aimed 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 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 2.1:

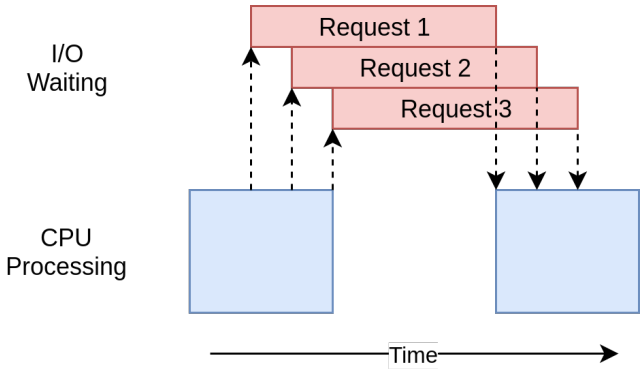


Figure 2.1: Requests over the internet processed in concurrent fashion [5].

More specifically in Python, concurrency can be expressed either through the Threading or AsyncIO(short for Asynchronous Input Output) [4] modules. Due to the infamous Global Interpreter Lock (GIL) [6] 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 instead. Not to mention the complexity using threads and making the program thread-safe is added. In a few words, GIL makes sure 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 provides developers the commodity of creating software locally with the knowledge that it will run identically regardless of the host environment [15]. 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 in 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 predefined configuration.

Chapter 3

Architecture

Contents

3.1 Modelling of Mining Procedure

creating a Session object allows requests to do some fancy networking tricks and really speed things up.

xss payloads found scattered all over the internet and in various github repos and sinaxame ola in one place

3.1.1 Defining Mining Environment

3.1.2 Strategies as State Machines

3.2 Selfish Mining

3.3 Stubborn Selfish Mining

3.3.1 Lead

3.3.2 Equal-Fork

3.3.3 Trail

3.4 Conservative Stubborn Selfish Mining

3.4.1 Safe-Lead

3.4.2 Safe-Equal-Fork

3.5 Hybrid Strategies

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 [20] coding style standard and, regarding documentation, the PEP 257 [19] and Sphinx [**?**] docstring conventions where used so it will be clear and easy to read from programmers. Pylint [10] 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 [11], unittest [12] and mock [3]. 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 [4] 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.

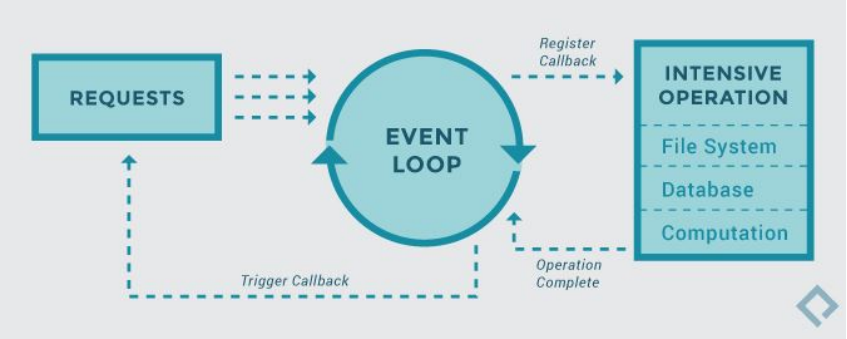


Figure 4.1: AsyncIO mechanism; it provides a high-performance asynchronous frameworks for making our fuzzing requests [13].

Communication with the target's web site is achieved with a blazingly fast asynchronous http client/server framework named aiohttp [8]. 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 modules are becoming compatible with asyncio [8].

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` [**?**] 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 [**?**] 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

grapse giaa to argparser, gia to readme me oles tis odigies p exei gia installation kai requirements.txt.

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

Logging is a module in the Python standard library that provides a richly-formatted log with a flexible filter and the possibility to redirect logs to other sources such as syslog or email.

4.5 Multi-Container Deployment / Methodology kalitera

barto sto evaluation

4.6 argparser

ccc

Chapter 5

Evaluation

Contents

5.1 Methodology

5.2 Dominant Strategies

5.3 Revenue and Comparison with Stubborn Strategies

5.4 Fairness of Blockchain

5.5 Risk Safety Property

Chapter 6

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).

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).

Chapter 8

Conclusion

Contents

8.1 Conclusion

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] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix A

Appendix B