Automated Smart Fuzzing Vulnerability Testing

João Afonso Lopes de Almeida Pinto Coutinho

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor(s): Prof. Ricardo Jorge Fernandes Chaves
Dr. Rui Carvalho

Examination Committee

Chairperson: Prof. Luís Manuel Antunes Veiga
Supervisor: Prof. Ricardo Jorge Fernandes Chaves
Member of the Committee: Prof. David Rogério Póvoa de Matos

November 2022

Dedicated to father Eduardo Coutinho.

Acknowledgments

I would like to thank Instituto Superior Técnico and InnoWave for accompanying me during the development of this thesis. I would also like to thank my family, my mother, my sister and my brother, as well as my friends and colleagues Rita Baptista and João Porto for their relentless support.

Resumo

Desde que aplicações web se tornaram cada vez mais comuns pela internet, ser capaz de testa-las de forma eficiente é crucial para o seu sucesso. Técnicas de fuzzing sempre foram relevantes na testagem de software, ainda mais em aplicações web. No entanto, de modo a torná-las mais eficientes, smart fuzzing é uma extensão extremamente importante deste método de testagem. Nesta tese propõe-se e desenvolve-se uma ferramenta autónoma de smart fuzzing evolucionário emparelhada com um crawler de web para o teste de aplicações web para a identificação de vulnerabilidades. De modo a aproveitar os pontos fortes de fuzzing, esta ferramenta foca-se especificamente nos pontos de upload de ficheiros, como método para causar execução de código. Os resultados experimentais comprovam a validade da aplicação de algoritmos genéticos na testagem e identificação de vulnerabilidades no uploads de ficheiros, enquanto demonstra o potencial do crawler como método auxiliar para aumentar autonomia.

Palavras-chave: Aplicações Web, Fuzzing, Teste de Vulnerabilidades, Algoritmos Genéticos

Abstract

Since web applications have become more and more common throughout the internet, being able to test them efficiently is crucial to their success. Fuzzing techniques have always been relevant in testing software, even more so in web applications. However, to make it more efficient, smart fuzzing is an extremely important extension of this testing method. This project proposes and tests an autonomous smart evolutionary fuzzing tool paired with a web crawler dedicated to testing web applications for vulnerabilities. To play into the strengths of fuzzing, it specifically targets file upload endpoints in an attempt to cause code execution. This work proves the validity of applying genetic algorithms to testing file uploads while showcasing a crawler as a possible auxiliary tool to increase autonomy.

Keywords: Web Applications, Fuzzing, Vulnerability Testing, Genetic Algorithms

Contents

Acknowledgments . . . v

Resumo . . . vii

Abstract . . . ix

List of Tables . . . xiii

List of Figures . . . xv

1 Introduction **1**

1.1 Work Objectives . . . 2

1.2 Thesis Outline . . . 3

2 Background **5**

2.1 Web Applications . . . 5

2.2 Taint analysis . . . 6

2.3 Fuzzing . . . 7

2.4 Genetic Algorithms . . . 9

2.5 PNG . . . 10

2.6 JFIF . . . 11

3 Related Work **13**

3.1 Web Application Vulnerability Scanners . . . 13

3.2 Web Application Fuzzing . . . 15

3.3 State Of The Art Fuzzers . . . 20

3.4 Greybox Fuzzing . . . 21

3.5 Evolutionary Fuzzing . . . 24

3.6 Overview . . . 25

4 Proposed Solution **27**

4.1 Attack Vector . . . 27

4.2 Data Generation . . . 28

4.3 Input Structure Awareness . . . 29
4.4 Web Crawler . . . 30
4.5 Architecture . . . 30

5 Implementation **33**
5.1 Web Crawler . . . 33
5.2 File Parser . . . 34
5.2.1 Tree Structure . . . 34
5.3 Genetic Algorithm . . . 36
5.3.1 Generic GA . . . 37
5.3.2 FAexGA . . . 40

6 Evaluation and Results **43**
6.1 Test Results . . . 44
6.1.1 Crawler . . . 44
6.1.2 Fuzzer . . . 45
6.1.3 Full Stack . . . 48
6.1.4 Overview . . . 49
6.2 Comparison . . . 50

7 Conclusions **53**
7.1 Achievements . . . 53
7.2 Future Work . . . 54

Bibliography **55**

List of Tables

3.1 Crossover probability combinations [11] 25

5.1 File extension mutations 39

5.2 Crossover probability percentages 41

6.1 List of test scenarios 43

6.2 Performance of crawler in parsing upload forms 44

List of Figures

2.1 Web application architecture 6
2.2 Selection and crossover stages of a genetic algorithm [3] 10

3.1 Weighted message parse tree. (a) A sample message. (b) WMPT of the sample message. [31] 16
3.2 Architecture of KameleonFuzz [37] 18
3.3 Proposed architecture [44] 22

4.1 Architecture of implementation 31

5.1 Flowchart of crawler's execution 35
5.2 Format of tree structure 36
5.3 Format of parsed PNG 37
5.4 Format of parsed JFIF 37
5.5 Workflow of generic GA 39
5.6 Workflow of FAexGA 41

6.1 Number of requests performed by the crawler 45
6.2 Fuzzer success rate 46
6.3 Evolution of Generic Algorithm 47
6.4 Evolution of FAexGA in Test #2 48

Chapter 1

Introduction

With the rapid rise of technology dependence, more and more organizations move their core services onto software applications to keep up with their competitors. Nowadays, it is very hard to find a service that does not feature an app for its customers to use. However, since the number of applications increases, also does the number of possible attacks. When trying to secure a system against malicious actors, it is very easy to overlook simple things or forget to deploy the most basic security measures. Frequently, these mishaps result in vulnerabilities that are a direct consequence of not using proper security testing other than static code analysis or in some cases not testing at all. While web applications might hide most of their code in a backend server, their high availability still allows attackers to attempt to exploit them with ease. Numerous vulnerabilities can be discovered in a web application [1] and being able to identify them before they are exploited is crucial for the success of an organization. Furthermore, having third-party audits cannot only help in this process but it is also an essential step for receiving ISO [2] certifications, which go a long way in building a reputation. As a result, incorporating security testing in the application development process is very important to ensure its longevity and reliability.

Security testing is a very common and effective way of discovering bugs and vulnerabilities in an application. It allows security professionals to perform an analysis from a different perspective than the developers, which will often bring light to issues and scenarios that were overlooked beforehand. Application testing can be complex and requires a certain level of knowledge about its internals and so fuzzing techniques can be used to, not only find exposed application paths but also trigger crashes that may result in a vulnerability. Although a common practice, fuzzing is sometimes heavy, since it tries to explore every possibility. As a result, smart fuzzing is key to not only delivering faster results but also to do it in a way that does not have a significant impact on the web server running the application.

Since web applications are continuously updated (see Section 2.1), the need for continuous testing becomes more relevant. Being able to deploy changes to the application more efficiently means that possible vulnerabilities also reach production environments with the same ease. Furthermore, web applications have complex architectures, as discussed in Chapter 2, meaning that performing unit tests on each module might not reveal vulnerabilities that only occur when the application is fully deployed. This phenomenon promotes performing testing on the application as a whole, emulating the user's perspective. Fuzzing techniques are ideal in this scenario since they attempt to exploit the application through input methods publicly available to the users.

1.1 Work Objectives

The goal of this project is to develop a tool able to perform security assessments on a web application. This tool considers the use of fuzzing techniques to discover vulnerabilities in the target application. It should be able to function autonomously with very little manpower or maintenance, which would result in the opportunity to perform periodic evaluations automatically. The fuzzing algorithm falls in the smart fuzzing category, which is explained in Section 2.3. The tool should also be a viable option in third-party testing with no access to source code or application internals. As a result, it was developed with a blackbox approach.

The final solution will target file upload endpoints of web applications. It uses genetic algorithms [3] with a custom file parser and is paired with a web crawler to provide a higher level of autonomy. Two different versions were implemented and evaluated as a means to determine which is more appropriate to the given environment. The experimental results show that this technique is a valid method for identifying vulnerabilities in file upload endpoints, albeit with a few limitations. Although the fuzzer itself is extremely successful, rounding the necessary conditions for it to be applicable is not trivial. These limitations are discussed in detail in the later Chapters and serve as the basis for future work suggestions.

1.2 Thesis Outline

The work contained in this Thesis displays the research, development, and evaluation process of the proposed tool. Chapters 2 and 3 describe the related research, relevant methods for developing such a tool, and the relevant state of the art presented by other authors in that field. Chapters 4 and 5 describe the architecture and implementation of the developed tool alongside external libraries and technologies used to aid its execution. Chapter 6 discusses the results of tests performed on the tool, interpreting their outcomes, implications, and whether or not they fill the designated requirements. Finally, Chapter 7 summarizes the various achievements of this work as well as proposes avenues for future research on the topic.

Chapter 2

Background

In the following sections, the concepts necessary for the understanding of the proposed work are introduced. In Section 2.1, the concept of web applications is explained, their advantages, architecture, and main vulnerabilities. Section 2.2 introduces the taint analysis method for software testing. Section 2.3 explains fuzzing in a more detailed manner, namely what it is, its various categories and parameters, and what distinguishes different kinds of fuzzers. Finally, Section 2.4 introduces what genetic algorithms are and how they work, as this will be an important aspect of the research described in Section 3.5.

2.1 Web Applications

Web applications are a form of delivering software to consumers via the internet [4]. Instead of having the consumer use a program by downloading a binary, web applications host their software on servers that are accessible via the internet. The user connects to these servers by using the HTTP protocol through a browser, such as Chrome, Firefox, or Safari. The main benefits of using web applications include:

- Does not take space since it doe not need to be installed
- Updates can be rolled out more frequently as they do not need to be downloaded by users
- It is much easier to develop software compatible with a small set of browsers than it is with a large set of hardware/operating system combinations

The overall architecture of web applications, depicted in Figure 2.1, follows a client-server paradigm. This means that the users interact with a dedicated client, named the frontend, which performs operations by sending requests to the API (application programming interface) running on a backend server, through the HTTP protocol. An API specifies a set of public endpoints

that perform specific operations on the internal application. These operations performed on the backend usually interact with a database that is not accessible via the internet and the results are sent back by the API to the frontend, again via the HTTP protocol. This usually results in the frontend serving a web page to the user via a browser.

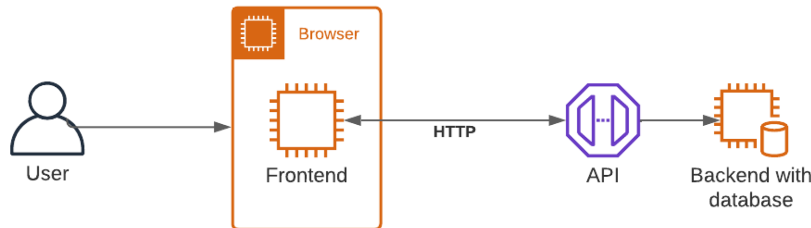

Figure 2.1: Web application architecture

There are a lot of vulnerabilities that can exist in a web application, due to its nature [1]. Two of the most important ones are SQLi (SQL injection) [5] and XSS (cross-site scripting) [6], which will be important later on. Both of them are based on poor handling of user-supplied input and can result in the application running malicious code, either in the database (SQLi) or in the browser (XSS). SQLi occurs when the backend performs some operation on the database that includes user-supplied input. If the input itself contains valid SQL code, this could be executed on the database and allow a user to run arbitrary code, at will. XSS works in a very similar way, but instead of targeting SQL running in the database, it targets JavaScript running in the frontend. If the web page served by the frontend includes user-supplied input that contains valid code, it could be run by the browser. Usually, this is exploited by sending a victim a malicious link that will run the attacker's crafted code on the victim's browser. Both of these vulnerabilities can be patched by correctly sanitizing all user-supplied input.

2.2 Taint analysis

Taint analysis is a software testing method focusing on spreading user-controlled data throughout an application's source code [7]. To understand this concept, it is important to grasp the meaning of "taint", "source" and "sink". A piece of data is considered to be tainted if the user has some control over its content. Sources are input vectors whose data is controlled by the user and is therefore tainted. Finally, sinks are sections of code that could cause harm if the input data is not sanitized. For example, in XSS, the query parameters in a URL are sources since they are controlled by the user, while the page's document is a sink since it can run JavaScript code with script tags. Tainted data can propagate through operations such as attribution or concatenation

but can also be sanitized to remove the taint. For XSS, special characters like “<” can be filtered out or replaced by safe encoding to prevent script tags from being run. The objective of this sort of analysis is to determine if data can propagate from a source to a sink without being properly sanitized somewhere along the way. A very simple example in Python code is depicted in Listing 2.1.

```python
a = input() # a is now tainted
b = a # taint spreads from a to b
c = b + d # taint spreads from b to c
b = sanitize(b) # b is sanitized and is no longer tainted
system(c) # VULNERABILITY: system is a sink and is being called with a tainted argument
system(b) # NOT A VULNERABILITY: b is not tainted
```

Listing 2.1: Taint example

2.3 Fuzzing

Fuzzing is a method of software testing that consists of discovering bugs and vulnerabilities in a program by supplying unexpected data in an automated way [8]. Fuzzing techniques are a very effective method of discovering misconfigurations and/or errors in code. When successful, often these result in the disclosure of information, crashes, or unexpected behavior that could compromise the application and the system it runs on. However, there are many obstacles to overcome when it comes to building a fuzzer [9]:

- *Human intervention*: Some fuzzers require knowledge about the application they are testing and this must be supplied by the user.
- *Test case generation*: Guiding the fuzzer when performing the tests to be more efficient, instead of using brute force methods.
- *Target interfacing*: Allowing the fuzzer to interact with the target program through whichever method it receives its input.
- *Outcome interpretation*: Determining if the target program handled the input well or had unexpected behavior/crashed.

When they were first conceived, fuzzing programs generated the data to be supplied as input completely at random [10]. Over time, multiple techniques have been developed to improve the efficiency of this process. The fuzzing programs available today fall into one of these categories: *generation*, *mutation* or *evolutionary* [9].

- *Generation* based fuzzers can generate input from scratch, either completely at random or by following a user-supplied model (e.g. network protocol, file format). These two options are the distinction between a *dumb* and *smart* fuzzer, which will be formally defined later on.

- *Mutation* based fuzzers apply transformations to a collection of seeds supplied by the user. These may include changing, adding, or removing bytes from the seed input before supplying it to the target program.

- *Evolutionary* based fuzzers are a more advanced technique that allows fuzzers to learn from each test case by measuring its success and adapting accordingly. It usually relies on genetic algorithms and may require the need of binary instrumentation to assess the target program's behavior.

The level of program structure awareness can also vary through fuzzing programs. Some treat targets like a *blackbox*, having no knowledge of the application source code or structure [11]. Others use a *whitebox* approach, which takes advantage of knowing source code to track the fuzzer's results and progressively increase code coverage [12] [13]. Finally, there are *greybox* fuzzers, which use partial application knowledge such as binary instrumentation to assess code coverage, without the dramatic increase in overhead that results from analyzing source code [14] [15] [16].

Besides the conception of data, fuzzing programs are also responsible for interfacing with the target program and interpreting the output of its test. The former can be as simple as communicating through some network protocol or generating command line arguments but can escalate to simulating key presses or mouse movements. The latter refers to determining whether or not the target program handled the input correctly by parsing the response or detecting a crash.

Often the programs that are being tested perform sanity checks on the data that they receive (e.g. testing if a supplied number is not negative or if a picture is a valid PNG). As a result, fuzzing programs must find the right balance between "expected data" and "random data". What this means is that for a fuzzing program to work, the data it supplies must be "valid enough" such that it passes initial sanity checks, but also "invalid enough" such that it causes some unexpected behavior in the program. Besides the conception of data, fuzzing programs can also be categorized when it comes to their input structure awareness. Those categories are *dumb fuzzers* and *smart fuzzers* [9].

- *Dumb fuzzing* means that the fuzzer program is not aware of the input structure and therefore the data it creates is somewhat random. It might flip random bits or insert

"interesting bytes" in random locations. As a result, data created from this sort of fuzzer might fail sanity checks performed by the target program, e.g. it is unlikely for a dumb fuzzer to create data that has a valid checksum.

- *Smart fuzzing* on the other hand is aware of the input model and can create data that respects this model, such as a file format or a formal grammar. It can generate data from scratch or apply modifications to seed inputs while still maintaining the desired structure. This method of fuzzing allows for much better results since it can easily bypass sanity checks performed by the target program. However, it requires more human interaction to function, since the input structure must be known from the start.

With the use of *smart fuzzing*, those initial sanity checks performed by a target program become a much smaller obstacle, if not eliminated. As a result, this type of fuzzer can immediately start testing relevant program logic and will have a much higher success rate than its counterpart. However, it does come bearing some obstacles. *Smart fuzzers* are much more complex than *dumb fuzzers* and require more human intervention. Determining the input structure is not as straightforward as it may seem. If it is too rigid then the fuzzer might not be able to cause any damage, however, if it is too soft the data might get caught by those sanity checks, beating the purpose of a *smart fuzzer*. Finding the right balance is a complex task that must be performed by the programmer when developing a fuzzing program. Besides that, it also does not fix the main problem behind fuzzing in general, which is the transformations applied to the data. Being *generation*, *mutation*, or *evolutionary* based, there are countless possibilities when it comes to the final piece of data that is to be fed to the target program. Without some sort of guidance, fuzzing programs become extremely close to brute forcing programs, which can be very taxing on both the fuzzing and the target machine's processor. Furthermore, most bugs will be caused due to known edge cases such as null bytes or empty strings. If the transformations are random, these known values might never be tested.

2.4 Genetic Algorithms

Genetic algorithms are a machine learning technique based on the theory of evolution [3]. They rely on the concept that the subjects who are most fit for completing a certain task survive throughout the generations. This technique is used to find the optimal solution for a problem by generating populations of candidates, named chromosomes, and each generation selects the most successful ones to "reproduce". To generate new chromosomes, first, a crossover operation is performed on two other chromosomes, and then the offspring is mutated, so it can introduce new

behavior. As a result, with each passing generation, the population is composed of increasingly optimal chromosomes, and the success rate increases. An example iteration of this process is depicted in Figure 2.2.

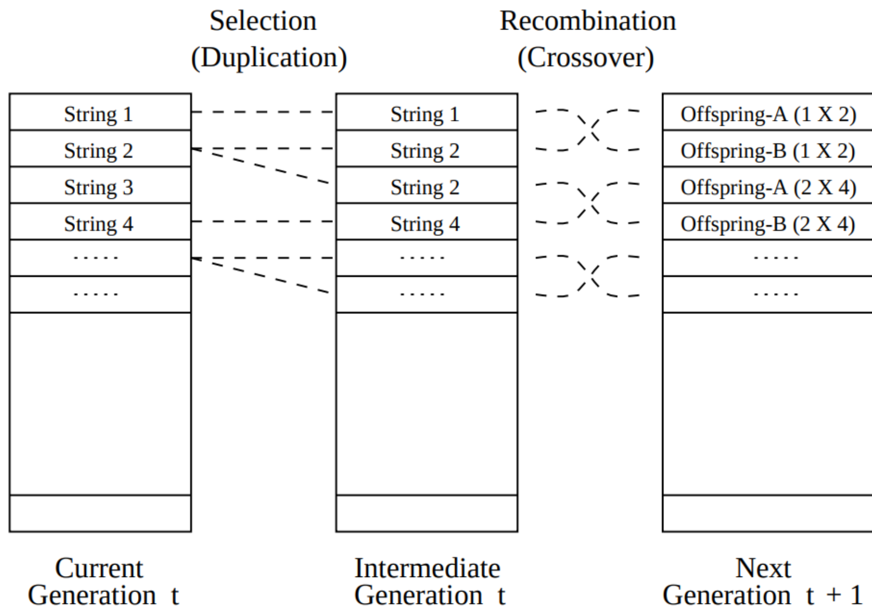

Figure 2.2: Selection and crossover stages of a genetic algorithm [3]

Some parameters must be defined when implementing genetic algorithms. The population size is usually determined to be the same number across the entire execution of the algorithm, although this is not always the case (see Section 3.5). Then, the chromosomes must be defined in a way that their behavior is easily mutable during runtime to effectively apply mutations. Afterward, the fitness function must be established as an efficient and effective way of calculating the success of a chromosome. For the breeding phase, a crossover and mutation probability must be defined and they will be applied after the breeding chromosomes have been selected. The stop condition for the algorithm can vary between finding a solution that is considered optimal or defining a maximum number of generations, after which the algorithm halts.

2.5 PNG

PNG, or Portable Network Graphics, is a file format that allows lossless and portable storage of images [17]. Its specification defines an image by splitting it into numerous structured chunks, following an 8-byte signature at the start of the file. Each of these chunks is composed of 4 different fields:

- Length: 4-byte unsigned integer specifying the number of bytes in the chunk's data field
- Chunk Type: 4-byte case-sensitive chunk identifier, restricted to uppercase and lowercase ASCII letters
- Chunk Data: Data bytes of chunk with size Length
- CRC: 4-byte Cyclic Redundancy Check [18] calculated on Chunk Type and Data

Each of the chunks in the PNG format has a different purpose and may have some sort of constraints, be it ordering, cardinality, or whether or not they are optional. In the context of this work, 4 chunks must be well understood: IHDR, IEND, IDAT, and tEXt. The IHDR chunk is a critical chunk that specifies details about the image, such as width, height, bit depth, color type, compression method, filter method, and interlace method. It can only be present once in the file and must be at the very start, immediately following the signature. The IEND chunk is another critical chunk that does not contain any data and must be the last in the file. The IDAT chunk is also critical but can be found multiple times in the file, as long as they are consecutive, and contains the actual image data. Finally, the tEXt chunk is an optional chunk that can be found anywhere in the file any number of times, as long as it does not break any of the other constraints, and contains text information saved by the encoder.

2.6 JFIF

JFIF, or JPEG File Interchange Format, is another file format that allows for the storage of images, in this case using the JPEG compression method [19] [20]. Much like the previously mentioned PNG format, a JFIF image is also defined with a sequence of several similar blocks, named markers. Each marker is composed of a type and optional data following it. The type is 2 bytes long, always starting with the byte 0xFF, followed by another byte that identifies the marker. If the marker includes data, the 2 bytes following the type indicate the length of said data (including the 2 length bytes). Otherwise, the marker type is immediately followed by another marker, which can be recognized by the byte 0xFF. A marker can also be followed by entropy-coded data which is not included in the length. The termination of this data is identified by the start of the following marker. Similar to what was described in the previous section, 4 markers must be taken into account when manipulating a JFIF image: SOI, EOI, SOS, and COM [21]. The SOI (Start of Image) marker does not contain any data and must always be at the very start of the file. Similarly, the EOI (End of Image) marker also does not contain any data and must always be at the very end of the file. The SOS (Start of Scan) marker denotes

the start of the actual image data. Finally, the COM (Comment) marker includes textual data saved by the encoder.

Chapter 3

Related Work

In the following sections, existing research work developed on the topic is presented. First, Sections 3.1 and 3.2 showcase projects dedicated to testing web applications, namely, more generic scanners and dedicated fuzzers respectively. Then, in Section 3.3, some famous fuzzing projects are showcased. They are mentioned multiple times through this section, usually as a base of comparison. Finally, Sections 3.4 and 3.5 describe fuzzing research that aims to extend the technique. The former focuses on greybox fuzzing, while the latter focuses on evolutionary fuzzing.

3.1 Web Application Vulnerability Scanners

Wapiti [22] is a free and open-source security auditing tool designed for performing blackbox scans on web applications or websites. The scan begins by performing a crawling phase through the target, detecting possible input methods such as forms or scripts that read data. It then uses a fuzzing methodology, by attempting to inject various payloads into the application and evaluating the results to assess whether or not the target is vulnerable. There are numerous modules packaged with Wapiti, capable of testing against a wide variety of vulnerabilities from simple file enumerations to more advanced XSS. Although released a long time ago, it still sees continuous development and already includes a module for the recently disclosed Log4Shell [23] vulnerability. The usage is very user-friendly, as it features a command line utility that can be as simple as providing a URL, or as complicated as necessary by providing flags to specify various settings. Since it also features the option to write reports in well-defined formats like JSON or CSV, it can also be scheduled to run periodically on a certain application.

Arachni [24] is a web application security scanner framework written in Ruby. Much like Wapiti [22], it begins with a scanning phase that crawls through the website and learns about

its regular activity. By analyzing these actions, it can identify the legitimacy of test results, effectively avoiding false positives. It is aware of the dynamic nature of web applications and can adjust itself to the various states that are encountered, allowing it to identify more attack vectors than simpler scanners. Since it features an embedded browser, not only can it test client-side code, but also use technologies like JavaScript and AJAX to navigate the website. Although its modules cover a lot of web-based vulnerabilities, development seized in 2017, which means it cannot detect more recently discovered vulnerabilities. One of its key features, however, is its versatility, as it can be used as a simple command line utility, a Ruby library that can be used in scripts, or even a multi-user web-based collaborating platform.

W3af [25] is another web application audit and attack framework. It is written in Python and employs a plugin-based workflow, meaning its behavior is defined by the active plugins when it is run. There are three main types of plugins, which are crawl, audit, and attack. Crawl plugins, much like the previously mentioned scanners, search through the application for new URLs and input methods. It allows for multiple variations running at the same time, each providing more URLs to the others. Audit plugins inject simple payloads into the inputs found by the crawl plugins to test if these are vulnerable or not. Attack plugins send more carefully crafted payloads to the vulnerable inputs, attempting to exploit them. Besides these main plugins, there are also some auxiliary plugins such as brute force for logins and output for reports. The usage works through a command line interface or a GUI, which is used to not only select the plugins and run the scan but features profiles that allow the user to save and load configurations for repeated use.

Zap [26] is a free and open-source web application scanner developed by OWASP [27]. It works as a man-in-the-middle proxy that stands between the user's browser and the target application. As a result, it constantly intercepts and analyses requests and interactions performed by the user, passively. This is the simplest use case for the application, however, it is not very extensive and usually misses a lot of vulnerabilities. For a more complete report, Zap [26] supports automated scans, which actively crawl through the application collecting requests and attempting to exploit possible vulnerabilities that might be encountered. If neither report method suits the user's desire, there is also the option of a manual scan, which overlays a heads-up display on the browser with various reports, all while the crawling is performed by the user. Besides the out-of-the-box features, there is a public marketplace where users can download plugins for additional functionality. The application ships with a GUI and allows users to save their work in sessions, which persist the reports returned by the various scans.

Burp Suite [28] is another web application scanner developed by PortSwigger [29]. It also

employs a man-in-the-middle proxy approach, performing passive scans to requests performed by the user and active scans in the background. Most of its autonomous work is performed in the background scans, using a very complex crawling algorithm that is capable of handling dynamic content and the complexity of modern web applications. Its built-in browser also allows the algorithm to traverse pages that rely heavily on JavaScript and as a result, the attack surface returned by the crawler is quite vast and complete. To increase performance, the scanner uses a fingerprinting technique so it can skip requests that result in the same state. Burp [28] can detect a wide range of vulnerabilities [30], however, this list can be trimmed down to the user's preference so as not to overload the scanner and increase efficiency. Besides the regular scans, it can also save intercepted requests and perform more specific attacks with them, usually by fuzzing its parameters. Much like Zap [26] it is used through a GUI and allows for sessions to be saved for persistent work.

Since these scanners were not introduced alongside a technical whitepaper, all developed research relied solely on project homepages and documentation. This means there is not much insight into the algorithms' work internally. However, it can be concluded that all scanners showcased in this Section include a crawling phase to detect application entry points and attack vectors, although some of them are more advanced, like Burp [28]. Furthermore, for the testing and exploitation phase, they all employ a blackbox fuzzing methodology, by injecting payloads into the application in hopes of causing some unplanned behavior. The origin of these payloads, however, is very poorly developed from the fuzzing standpoint. Instead of having some sort of generation or mutation algorithm, most payloads are predefined in the codebase. Burp [28] does have a mutation engine built in, but it can only be used on user-provided wordlists with a predefined set of rules. This leaves a lot of room for improvement in the fuzzing portion of these scanners. Implementing more advanced techniques for creating payloads, including but not necessarily smart fuzzing, would provide bigger opportunities for encountering more obscure vulnerabilities.

3.2 Web Application Fuzzing

In 2019, authors D. Wang et al. presented WMIFuzzer [31], which implemented a brand new method for fuzzing commercial off-the-shelf (COTS) IoT devices. These sorts of devices are one of the main reasons behind the extremely rapid increase of internet-connected devices over the last few years. They present a very convenient attack vector for internal networks since they lack the computing power to run security-related software. Furthermore, instead of the conventional mouse and keyboard interface, IoT devices usually present their users with a web

application for configuration and management. As a result, WMIFuzzer [31] was designed to target the device's web interface. Due to the nature of web applications, the fuzzer was built with a blackbox approach, since it does not have access to the application running in the backend. This limitation brings forward some obstacles. The authors describe the generation of seeds and their corresponding mutations as challenges to overcome. For the generation of seeds, WMIFuzzer [31] relies on UI automation [32] techniques. In the initial phase, it crawls through the web frontend while performing various calls to the backend through the UI. The requests generated by these calls are assumed to be valid and correct. They are, therefore, intercepted and stored to serve as the seeds for the fuzzing algorithm. After collecting various seeds, the fuzzer must apply transformations to them to try and cause undefined behavior in the application with a smart fuzzing method, meaning that the overall structure of the seeds must remain valid. Since these are composed of HTTP requests, some mutating strategies must be used. The authors propose storing the seeds as a Weighted Message Parse Tree (WMPT), depicted in Figure 3.1. This structure can represent HTTP requests in the form of an abstract syntax tree. The various fields and values of the request are stored in the leaf and internal nodes of the tree. The mutation operations applied on the tree are: changing leaf nodes to random values, deleting an internal node, or deleting a child of an internal node.

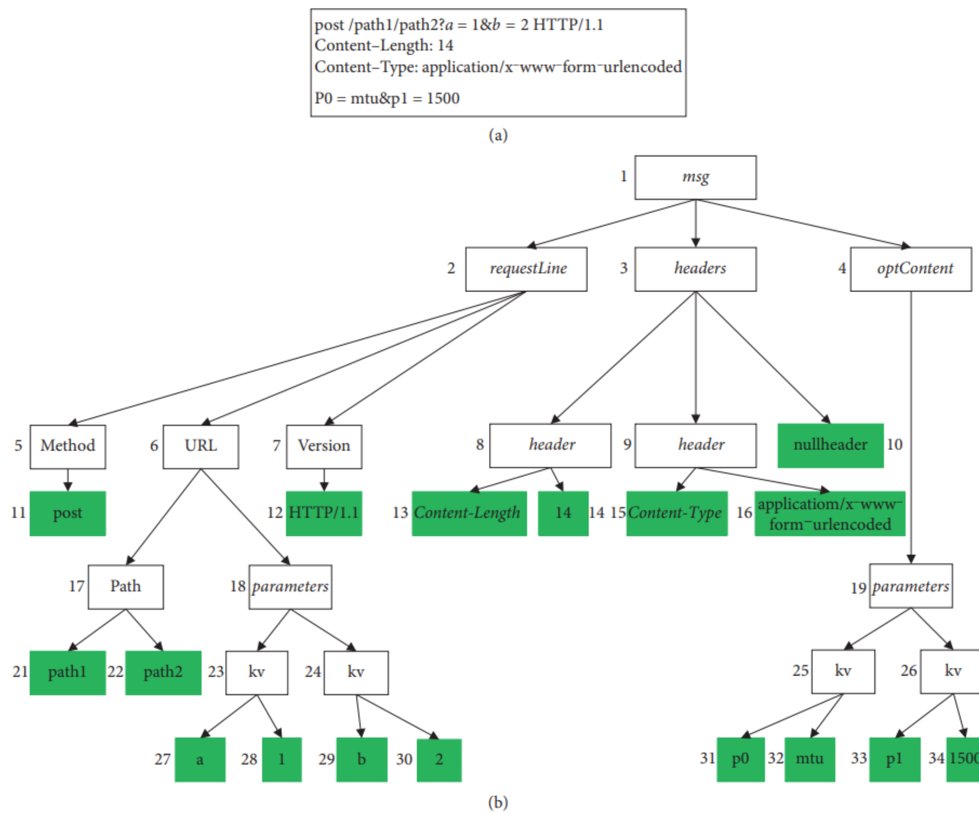

Figure 3.1: Weighted message parse tree. (a) A sample message. (b) WMPT of the sample message. [31]

Authors L. K. Shar et al. presented in 2020 a smart fuzzing tool dedicated to testing Samsung SmartThings IoT apps, named SmartFuzz [13]. Their tool was built using a whitebox approach and its workflow can be split into five steps:

- *Static analysis*: Analyzes source and instruments app
- *App deployment*: Deploys instrumented app
- *Dynamic analysis*: Analyzes UI and tracks coverage
- *Test generation*: Generates test inputs
- *Test execution*: Supplies test inputs to the app

The first phase of the fuzzing procedure is static analysis. To accomplish this, first, an inter-procedural control flow graph of the target app is generated, through a third-party program. When traversing this graph, starting at the app's entry points, there are three responsibilities for the fuzzer First, extract information about the app's parameters, capabilities, user preferences, and endpoints. These include possible app behaviors, what conditions must be fulfilled for these behaviors to trigger, and how to trigger them. Then, it must identify the possible sinks of the app, i.e. endpoints that could trigger critical operations. And finally, it performs instrumentation on the app such that, during testing, it can send back information about executed sinks and injects extra endpoints so the test case generator can simulate user events with API calls. After the first phase, the instrumented app is deployed and the dynamic analysis step begins. It has two main objectives, identify possible valid values for the app's parameters, by navigating the UI using Selenium [33] and track sink coverage during the test execution phase. The generation of the test inputs is done through three different methods consecutively. It starts by generating all possible pairs of parameters and testing these results. Afterward, it permutes through those pairs and tests those results. Finally, the last batch of test cases is generated through an all-combinations method on the whole set of parameters. Besides generation, each of these methods also applies some randomization to the results. The test execution is done by supplying the generated inputs to the app through Selenium [33].

Authors F. Duchene et al. presented in 2012 a method of fuzzing web applications for XSS vulnerabilities [34]. Even in 2021, XSS remains on the top of the most dangerous vulnerabilities in software [35]. The publication suggests a fuzzing application specifically targeted to discover this sort of vulnerability. It relies on two main components: model inference to estimate the possible states of the web application [36] and evolutionary fuzzing guided by a genetic algorithm. Much like the approach showcased in [31], the technique presented by the authors begins by

running a crawler through the web application. However, besides collecting the requests to the backend, it also saves the various states and transitions it finds throughout this process. As a result, the fuzzer can maintain the model of the application in a graph structure, where each node is a possible state and each edge is a transition. This model is used as a way to predict where the injected payloads are reflected. The evolutionary fuzzing portion of the program uses a generic genetic algorithm with some key aspects. The initial generation is composed of the requests collected through the model inference stage, it uses a formal grammar to generate valid XSS payloads, respecting both JavaScript and HTML syntax and the fitness function is a complicated expression that takes into account the amount of reached states, the amount of injected characters and the validity of the returned page.

Two years after presenting the work showcased in [34], authors F. Duchene et al. introduced KameleonFuzz [37]. Much like their previous work, it is a fuzzer application designed to detect XSS vulnerabilities in web applications. It uses LigRE [38], developed by the same authors, to perform the model inference and taint inference stage. This allows the fuzzer to generate a model of the target application's states, as well as a taint inference model. It then uses a genetic algorithm paired with a more precise taint analysis as a guide. The overall architecture of the tool is displayed in Figure 3.2.

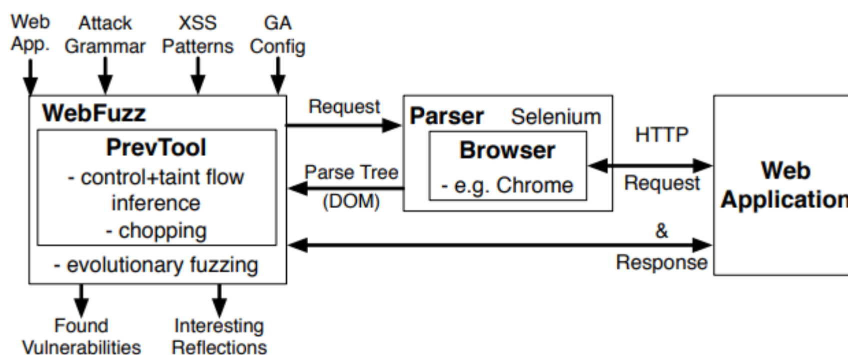

Figure 3.2: Architecture of KameleonFuzz [37]

After each test case, a new taint analysis phase is run. It is used as a way to complete the previously completed taint flow analysis by combining those results with the response from each test case. As a result, it can mark exactly in the response where the tainted values are, if any, and better evaluate the effectiveness of the payload. The genetic algorithm that drives the fuzzer utilizes an attack grammar to create the various chromosomes of the population. This grammar is developed by a human and serves as the model to be maintained during the crossover and mutation stages of the algorithm. When it comes to fitness calculation, the expression used takes into account the fuzzed request, its response, the result of the taint analysis, and the

overall application model. Besides those parameters, extra weight is added to the test case if some conditions occur. For example, successfully injecting characters in the response adds a lot of weight, while returning an unexpected page adds much less.

In 2021, authors F. Gauthier et al. presented BackREST [39], a web application fuzzer that employs a greybox approach. Unlike [34], it does not just target XSS vulnerabilities, as it can also detect SQLi and command injection, among others. It uses model inference to generate the test cases and coverage/taint feedback to evaluate them. Its model inference method is very similar to the one presented in [34], with a few key differences. Instead of parsing the HTTP requests collected from the crawler as trees, they are parsed and reassembled into a graph structure. The fuzzer uses these graphs as models, and when traversed, they generate valid HTTP requests. This resembles a generation based approach instead of the evolutionary one showcased in [34]. Besides that, this phase is only used as a means to generate the model of the requests and does not keep a record of the various states and transitions of the application. To replace the feedback provided by knowing the states and transitions of the application, BackREST [39] uses coverage and taint feedback to guide itself. As part of the fuzzer's architecture, there is coverage and taint analysis tools running on the server side that report back with information after every request. This information includes which parts of the code are being hit by the various payloads, as well as the possible vulnerability types present. This also means that access to the server running the application is required and therefore this technique can only be used in internal testing scenarios.

All the fuzzers showcased in this Section target web applications in some way. Due to their blackbox nature, out of all the different aspects of a fuzzing program, evaluating test cases is the most interesting one for this category. The fuzzers presented in [13] and [39] however, rely on some sort of code analysis to run and are therefore not as relevant for this project. The work showcased in [31] uses a mutation based method to generate test cases, and as a result, its evaluation method is not as important to the algorithm and relies solely on pinging the device and checking for failed responses. However, the fuzzers presented in [34] and [37] are driven by a genetic algorithm and must pay extra attention to test evaluation. They both use a similar method, which requires an initial model inference phase before fuzzing. Not only is it very costly to generate this model, but its accuracy also becomes a direct dependency on the fuzzing's effectiveness. As a result, the problem is derived from how fast and how accurate can the application model be generated.

3.3 State Of The Art Fuzzers

American Fuzzy Lop [15], or AFL, is a fuzzing program developed by Google. It is meant to be used for testing binaries against malformed inputs and uses a greybox approach. The generation of test cases is done through a genetic algorithm that aims at increasing code coverage through each iteration, which is aided by instrumentation performed on the target binary. AFL [15] is meant to work as a general-purpose fuzzer that works out of the box, requiring no configuration and only needing one valid input example to kickstart the algorithm. However, it only accepts inputs from stdin or command line argument and has no consideration for the input structure, therefore it is classified as a dumb fuzzer. Although its development was halted a few years ago, the open-source community continues to maintain a develop a fork of the original fuzzer, named AFL++ [40].

LibFuzzer [41] is another fuzzing program, developed by the LLVM organization, that is very similar to AFL [15]. It is also driven by a genetic algorithm that is guided by code coverage, which is done through instrumentation performed on the target binary. Unlike AFL [15], it does not require providing an example of the input to be fuzzed, although it is recommended. However, it does require the programmer to write a dedicated entry point to the program. This is a function that takes a sequence of bytes and then performs some operation that is supposed to be tested. As a result, it is more versatile, since the entry point itself is written in code, which allows for more variations on how to interact with the program. It also supports parallel fuzzing supported by spawning multiple processes to increase efficiency. By default, LibFuzzer [41] performs random mutation operations on the inputs, however, it is possible to have it behave like a smart fuzzer at the cost of writing a custom mutator.

BooFuzz [42] is a more recent fuzzer that aims at providing high extensibility. It was created as a fork and successor to the recently halted project Sulley [43]. Much like the previously mentioned fuzzers, it supports instrumentation and can be used to test binary programs. However, it can also be used to target various protocols like FTP or HTTP. Instead of being attached to a program at compile time, it is used as a Python library. Therefore, the fuzzing scenario is entirely specified in a Python program, including target protocol, input structure, fuzzable values, and mutation operations. Although this allows for even more versatility and customization, it also means that it requires a specific setup for each use case instead of a more plug-and-play approach. As a result, it can be very taxing and tedious to get it to run with the desired specification, as the setups written in Python can become quite verbose very easily.

The authors for both AFL [15] and LibFuzzer [41] chose binary programs as the desired target for these fuzzers and built them focused on that one use case. Both projects are extremely

competent in this particular scenario, requiring very little setup but also allowing for some extensibility when needed by the user. BooFuzz [42], however, was built to be a more general-purpose fuzzer. Its philosophy of high extensibility allows it to be used in many use cases and scenarios but comes bearing the cost of requiring a lot of setup. Furthermore, the default mutations applied to the fuzzed values are random and writing custom mutators only extends the setup process. As a result, the fuzzer has no guidance and resembles more of a brute-force approach.

3.4 Greybox Fuzzing

In 2011, Bekrar S. et al. presented in Finding Software Vulnerabilities by Smart Fuzzing [44] a method for improving fuzzing techniques to make them faster and more efficient. They claimed that fuzzing techniques, although fast by nature, suffer from poor awareness of their coverage in the tested program. This results in wasting test cases, and consequently time and resources, on code paths that will never result in a vulnerability. The tool proposed in the paper is meant to target file processors and network protocols. It relies on coverage and taint analysis to track the paths that have the highest potential of containing a vulnerability. The architecture of this tool is composed of six distinct parts, depicted in Figure 3.3:

- *A. Vulnerability Pattern*: The first stage of the fuzzer should be to identify vulnerability patterns, named "VUPENS", within the binary. The example provided is the *strcmp* function with user-controlled parameters.
- *B. Taint Analysis*: It should use taint analysis [45][46] to identify paths between known sources and dangerous sinks.
- *C. Test Generation*: The proposed method for selecting the most relevant test cases is by tracking code coverage using search algorithms. Genetic algorithms [3] are also suggested as an alternative.
- *D. Coverage Analysis*: The assessment of the fuzzing should be done through code coverage by tracing the basic blocks that are executed.
- *E. Property Checking*: Besides code coverage, the tool should also check whether or not the test case was handled correctly.
- *F. Exploitability pattern*: In case a vulnerability is found, evaluate whether or not is exploitable.

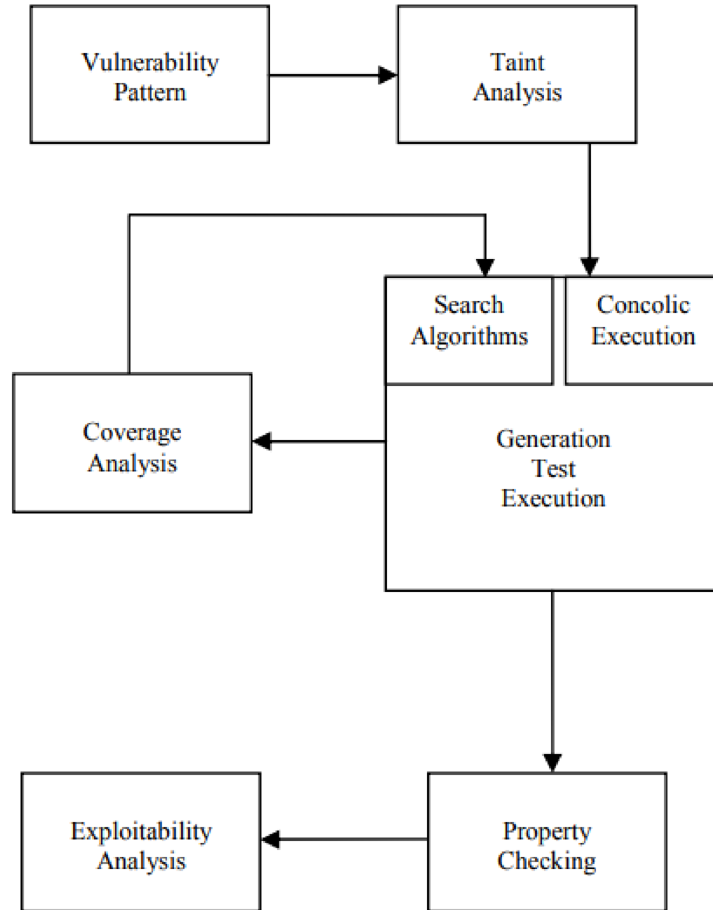

Figure 3.3: Proposed architecture [44]

Authors V. -T. Pham et al. presented in 2019 a smart greybox fuzzing tool named AFLSmart [14]. They claim that blackbox fuzzing is very ineffective and that the need for coverage analysis is crucial for a fuzzer's success. They decided to approach this problem by extending AFL [15] and implementing input awareness to create AFLSmart [14]. This smart greybox fuzzer can manipulate chunk-based files by first parsing them into a virtual structure in the form of a tree. The tree's nodes are either chunks or attributes. Chunks represent sequences of bytes from the file and compose the internal nodes of the tree. Attributes hold important data that is structurally irrelevant and make up the leaves of the tree. To create new seeds from this tree while maintaining the original file structure, the fuzzer can perform the following operations:

- *Smart deletion*: Deletes a random chunk from the seed
- *Smart addition*: Add a chunk from a second seed in an arbitrary position
- *Smart splicing*: Switch two chunks of the same type between two seeds

These operations also change the attributes of chunks to maintain coherence, like changing

the start and end indexes of chunks following a newly added chunk. Although this method should maintain the general structure of the file, there are still some challenges mentioned by the authors. First, they point out that the file format might have some more specific rules, e.g. a certain chunk only being allowed once in the file. AFLSmart [14] does not consider this issue, the authors decided that this sort of configuration could lead to more vulnerabilities. Then, they point out more low-level restrictions within chunks, e.g. changing an attribute might turn a checksum invalid. This issue, however, can be easily addressed. Checksums and chunk dependant properties can be recalculated later and might not even change if the chunk itself is not mutated. After implementing this mutation workflow, the authors introduce the notions of "stacking mutations", "fragment- and region-based mutation" and "deferred parsing". The latter one is the most important as it describes the probability of a file being parsed into the virtual structure. It takes into account the time since the last discovered path and as this time increases, so does the probability. Without this mechanism, all files would be parsed after a test and the efficiency of the fuzzer would be compromised. The use of a greybox approach also takes effect when calculating the lifetime, or "energy", of a seed. The authors introduce a method named "validity-based power schedule" which calculates the energy assigned to a seed by applying a transformation into the base power assigned by AFL [15]. This transformation depends on the "validity" of the seed, which is determined by the parser. As a result, seeds that result in "more valid" files are assigned more energy, and therefore last longer.

In another attempt at improving AFL [15] in 2019. Authors Yuwei Li et al. introduced a fuzzer based on predicting vulnerabilities using neural networks and evolutionary fuzzing named V-Fuzz [16]. The way they approach AFL [15] is different, however. Instead of implementing smart mutations to seeds, they decided to use neural networks to predict the vulnerabilities and use them to guide the fuzzer. As a result, V-Fuzz [16] can be split into two main components: the vulnerability prediction model and the evolutionary fuzzer. The purpose of the vulnerability prediction model is to identify where in the code there can be vulnerabilities. It relies on binary files being represented in a structure on which a neural network model can be applied. They decided to adapt "Graph Embedding Network" [47] [48] so it can be applied to binaries in the "Attributed Control Flow Graph (ACFG)" [49] form. When fully trained, the model is capable of a "Vulnerability Prediction" to each function in a binary program. These results will then be sent to the fuzzer to provide some guidance. The evolutionary fuzzer receives the results from the prediction model and uses VP to calculate a "Static Vulnerability Score" for each basic block in the binary. These scores are later used when evaluating the results of the tests. V-Fuzz [16] runs the fuzzing loop using the workflow of a genetic algorithm [3], with one major difference.

The fitness score of each test case is calculated as the sum of all SVS values hit by it. This way, it prioritizes seeds that get closer to vulnerable sections of the code. After running tests, the prediction model proved to be very effective, with accuracy levels reaching 80% consistently. It is also quite efficient, with a training time of about 200 minutes, which can be done offline without affecting the fuzzer. When it comes to the fuzzer, it was also very effective, finding vulnerabilities in many Linux applications, totaling 10 CVEs. Furthermore, while code coverage is not the main goal of V-Fuzz [16], the results show that it is not reduced when compared to other similar fuzzers.

The fuzzer showcased in [44] is presented only as a theoretical approach, with no prototype or complete implementation. The authors describe its main challenges as the binary and assembly analysis as well as identifying the vulnerability patterns. An implemented solution that attempts to tackle these problems is present in [16] and described in 3.5. AFLSmart [14] on the other hand, showed very positive results as it was able to discover much more bugs than its predecessor AFL [15]. The ability to parse files into a tree structure that can be freely mutated in a smart manner is not only crucial to maintaining efficiency, but also easy to integrate into a genetic algorithm driving the fuzzer. Therefore, the ability to integrate this process presents a very positive extension of the original AFL [15] project. The alternative extension presented in this Section, V-Fuzz [16], took a completely different approach. Instead of relying on smart mutations, the use of a neural network to identify bugs in code is also very efficient in guiding the fuzzer. It comes however with the added cost of not only training but also running the model before the fuzzing process can begin. When access to the code is possible, this latter scenario seems to be ideal as it leverages the most the situation at hand.

3.5 Evolutionary Fuzzing

Authors Last M. et al. presented in 2005 a black box fuzzing program based on genetic algorithms named Fuzzy-Based Age Extension of Genetic Algorithms (FAexGA) [50][11]. In its essence, it follows the generic genetic algorithm [3] workflow to generate test cases with an increasing success rate. It is based on GAVaPS [51] which implements a genetic algorithm with a varying population by size by assigning a designated lifetime to each chromosome at birth. The main difference implemented in this extension is assigning crossover probability in a dynamic way instead of being a static value. In FAexGA, crossover probability depends on the age of the chromosomes, prioritizing chromosomes that are "middle-aged". As a result, they present this property in their work through a table similar to Table 3.1:

For evaluation purposes, FAexGA [50][11] was tested against a complex boolean expression

Parent II / Parent I	“Young”	“Middle-age”	“Old”
“Young”	Low	Medium	Low
“Middle-age”	Medium	High	Medium
“Old”	Low	Medium	Low

Table 3.1: Crossover probability combinations [11]

with 100 boolean attributes and the logic gates AND, OR, and NOT. After generating the expression (E), a second expression (E') was generated by changing one random OR gate into an AND gate (or vice-versa). From here, they would supply a 100-bit binary string (T) to both expressions and if their results differ, it means that T revealed the bug in the circuit. This was tested with the expression depicted in Equation 3.1 [11]:

$$F(T) = \begin{cases} 1 \; if \; E(T) \neq E'(T) \\ \\ 0 \; if \; E(T) = E'(T) \end{cases} \tag{3.1}$$

After running the tests with 3 different configurations for FAexGA [50][11], results showed that all of them outperformed both a simple genetic algorithm and GAVaPS [51]. One of the configurations managed to find at least one solution in 95% of runs with 99.934% of the chromosomes in its final populations being valid solutions. It also managed to outperform the same previous algorithms in more famous tests like the Travelling Salesman Problem [52].

Evolutionary fuzzing can generate very positive results when it is applicable. Letting a genetic algorithm guide the fuzzer results in less human intervention and setup, when it is properly implemented. While most of the showcased evolutionary fuzzers rely on a greybox approach, the work presented in [50][11] introduces a new method for applying this sort of algorithm in a blackbox manner. Using this technique allows for a genetic algorithm to function properly in a scenario where there is very little feedback. While the authors only tested this on a boolean circuit, it would be very interesting to attempt to apply it in a more realistic fuzzing use case.

3.6 Overview

The fuzzing methodology is widely adopted in the testing of web applications. Its ability to be coupled with other techniques, such as taint analysis or genetic algorithms, allows high flexibility to accommodate various conditions. Furthermore, fuzzing itself is highly extensible and modular, with multiple parameters (input structure, program awareness, data generation) that can be

tuned individually for specific configurations. However, most of the projects described in Sections 3.1 and 3.2 use fuzzing as a technique but don't maximize its true potential, focusing more on traversing the application and widening the scope. In turn, the projects described in Sections 3.3, 3.4 and 3.5 attempt to tune data generation and mutation to increase performance, at the cost of much narrower scopes. The trend is that, when applied to web applications, fuzzing is used as a tool to identify the widest possible range of vulnerabilities, despite the more specific fuzzers being able to take advantage of more specialized configurations for their use cases. As a result, fuzzers designed for web applications are much more generic, which does not permit them to maximize their algorithms' potential. Attempting to craft a program that can take full advantage of the fuzzing methodology for a web application remains to be seen.

Chapter 4

Proposed Solution

From the applications described in Sections 3.1 and 3.2 it can be concluded that fuzzing is a very powerful technique for testing web applications. By their very nature, a web application's entry points are very easy to access. A user can interact with the application through the browser by navigating the frontend and filling out forms, text boxes, clicking buttons, etc. Furthermore, they can make direct calls to the backend API through an HTTP client and bypass the frontend entirely. Fuzzing at its core revolves around testing entry points with data that the application might not be ready to deal with, which means it is a prime candidate for testing web applications. For this project, a web application testing tool that takes advantage of the fuzzing methodology is proposed, to identify vulnerabilities, specifically in file upload endpoints. Applying an evolutionary algorithm, similar to the ones mentioned in Chapter 3, in order to the create and manipulate malicious files. This will allow the tool dynamically construct files that trigger vulnerabilities in the target application through continuous testing.

4.1 Attack Vector

Web applications have a lot of possible vulnerability types [1]. For a fuzzing program to be able to detect even a small set of these types, some drawbacks would inevitably occur. Since the fuzzing would have to be split among all of these vulnerabilities, which are exploited in different ways, the program would become very slow in trying to attack them all. To counteract this, fuzzers with a more general scope tone down the test case generation to become more efficient, or require the user to specify the target. BooFuzz [42] implements both of these techniques to achieve its versatility, but the downsides are evident. As mentioned in Section 3.3, the mutations applied to test cases are completely random, with no guidance whatsoever, and its setup process is very verbose and can get quite complicated very quickly. As a result, this project will be

implemented in a way that only targets one input method, so there can be extra focus on data generation to optimize it for that specific input.

From simple profile pictures in social networks to documents in file-sharing services, uploading files to web applications is quite common across the web. Although numerous vulnerabilities can be present in this entry point [53], there is very little research regarding this specific method of web application testing. While most applications showcased in Sections 3.1 and 3.2 are designed to detect and exploit XSS vulnerabilities through textual inputs, the proposed project will instead target file uploads in web applications. The main reason behind this decision is that XSS textual payloads are designed around writing JavaScript code segments, which is inherently a language designed for humans to work with (the same applies to other web vulnerabilities). The concept of fuzzing is based around delegating to the machine the task of creating data to be supplied to the target application. This means that when a fuzzer is generating XSS textual payloads, it is not only a complex problem for a machine, but also all generated payloads could eventually be developed by a human, although a lot more slowly. As a result, the only aspect gained from this technique is efficiency in test case generation. In the case of file uploads, the payload is an actual file, whose format is not meant for humans to work with. Therefore, by having the fuzzer generate files, it is a much simpler task and the results are test cases that a human would not be able to replicate since manually changing a file's contents is a lot more complex than writing code. As a result, not only does technique increase efficiency, but also efficacy, as the generated files would be much harder to achieve in any other manner, thus increasing the scope of possible vulnerabilities. Thus, the proposed fuzzer will be dedicated to fuzzing file upload endpoints in web applications, in an attempt to upload malicious images capable of causing code execution.

4.2 Data Generation

When developing a fuzzing program, some parameters must be established, namely the ones described in Section 2.3. As mentioned before, there are various alternatives as to how a fuzzer comes up with its test cases. This project proposes that the program implements an evolutionary fuzzing approach powered by a genetic algorithm. This method will allow the fuzzer to take advantage of a workflow that can guide itself toward the objective without the need for human intervention. Unlike other conventional methods, generation and mutation based fuzzing, evolutionary fuzzing allows the program to learn from each batch of test cases and improve itself over time. As a drawback, the algorithm is much more complex, but since it is already established that the fuzzer will only target one attack vector, the cost is not expected to be that large.

As established in Section 1.1, the fuzzer must comply with a blackbox approach. This presents an important obstacle in the implementation of a genetic algorithm. The lack of code awareness means that the only output provided by each test case is the response from the web server and possibly the rendered page. As a result, the fitness function for each of the chromosomes becomes difficult to define. To address this issue, the fuzzy-based age extension, introduced in [50] and described in Section 3.5 is herein considered. As demonstrated in [11], this extension to the genetic algorithm allows it to be applied in a blackbox scenario, where output is limited. Furthermore, the evaluation performed in [11] is on a boolean circuit, which has a binary evaluation function, while a web application would provide much higher granularity in its output, namely the response content. The pairing of this extension with the output provided by the HTTP response is believed to be sufficient for driving the genetic algorithm into finding solutions with a high success rate.

4.3 Input Structure Awareness

Besides their accessibility, web applications also have their entry points very well defined. This means that the format of the data they receive is predetermined and very specific. In the case of file uploads, a web application will usually only accept files that follow a certain format, so invalid files will get immediately rejected. As a result, the implemented solution must integrate smart fuzzing components in its algorithm, so that the test cases created by the fuzzer comply with the desired format, to pass initial sanity checks performed by the application. This means that these test cases will be able to go deeper into program logic and cover more code, widening the chances of finding vulnerabilities.

To maintain input structure and accomplish a smart fuzzing strategy, the project will implement a method similar to the ones showcased in [14] and [31], which are described in Sections 3.4 and 3.2, respectively. By parsing files and HTTP requests into a tree-like structure, it is possible to perform operations on their internals while still maintaining the overall structure. This works especially well in chunk-based files such as PNG or WAV, among others. Coupling this method with the genetic algorithm involves performing these operations at the crossover and mutation stages, with the former consisting of transferring nodes between trees and the latter consisting of deleting nodes or altering their values. As a result, when breeding new chromosomes they will remain valid files.

4.4 Web Crawler

In all of the work showcased in Chapter 3, there is a method to evaluate whether or not a vulnerability was triggered in the tested application. In the case of XSS payloads, for example, these tools would retrieve pages in the application that reflect these payloads as a way of testing for their execution. While most of them would use model inference techniques, this method was deemed computationally heavy and possibly unreliable. As an alternative, this project will implement a web crawler to search for uploaded images with a similar methodology, but two key differences. First, it will begin crawling the application from the upload page instead of its root. This decision relies on the idea that an uploaded image will be reflected on the application only a few clicks away from the upload page. Second, instead of modeling the entire application, it will only search for the uploaded images and stop there. This is motivated by the fact that the inputted text might be reflected on multiple pages, an uploaded file lives in only one location on the web server. Furthermore, this web crawler can also be used to parse the upload form and gather any necessary values from it, aside from the actual file.

4.5 Architecture

Figure 4.1 depicts an overview of the components that comprise the solution. This work implements a smart fuzzing tool that consists of three main modules, which will be described in detail in the following Chapter: *file parser*, *genetic algorithm*, and *web crawler*. Each of these components plays a crucial part in the fuzzer's ability to detect vulnerabilities in a web application's file upload endpoints. The *file parser* is responsible for the manipulation of the images used in the tests, representing them in a structured format described in Section 5.2. The *genetic algorithm* is the driver of the fuzzer, performing a guided search to discover how to trigger a vulnerability in the tested application. The *web crawler* runs before the other components and gathers information about the application that will be necessary for the algorithm.

Section 5.1 describes the web crawler is implemented using an external framework, which means its execution runs in a separate context from the rest of the program. This creates an obstacle in communicating between the crawler and the other components. To address this issue, there is a fourth auxiliary component in a small NoSQL [54] database. The use of a small database allows the crawler to pass data in and out of its context, breaking the framework's barrier. Furthermore, it supplies a simple API that also forces the data to be inserted in the database in a structured JSON format.

After the web crawler extracts all relevant information to the database, the file parser and

the genetic algorithm work in tandem to perform the tests on the application. The parser reads the initial population seeds from the file system and stores them in a state that can be easily manipulated. The genetic algorithm performs the tests on the application and decides on the operations to be performed on the images, which are passed to the parser. An overview of the interactions between the components is depicted in Figure 4.1.

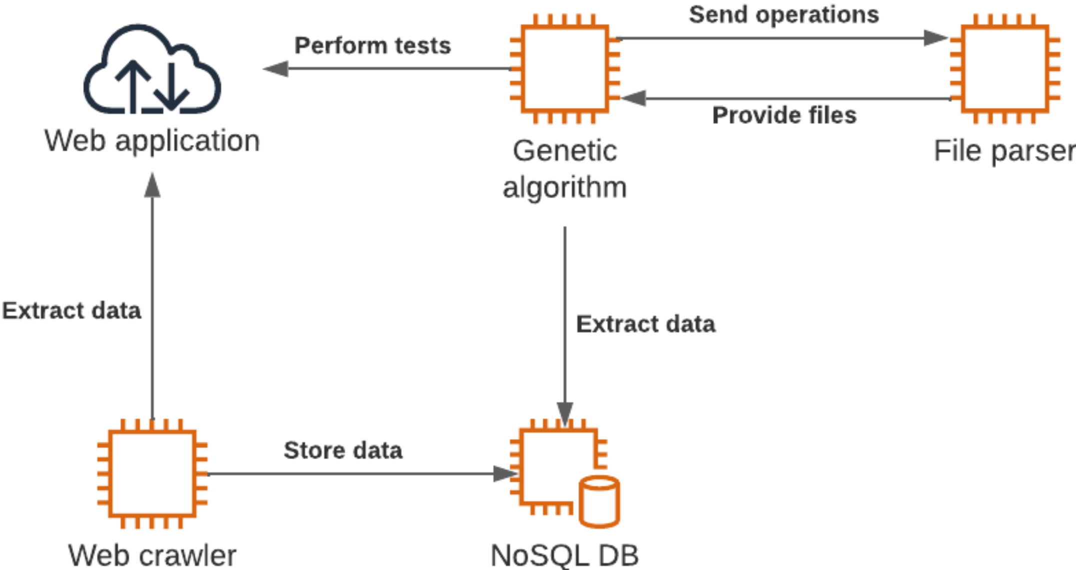

Figure 4.1: Architecture of implementation

Chapter 5

Implementation

In this chapter, the implementation of the proposed solution described in detail. Section 5.1 describes the web crawler used to provide the other components with information necessary for their execution. Section 5.2 specifies the file parser and how it operates on various file types. And finally, Section 5.3 performs an overview of the genetic algorithm and its various parameters, as well as a comparison between two different variations of this technique.

5.1 Web Crawler

For the fuzzing algorithm to function properly, some aspects of the web application must be known before its execution. These are the file upload endpoint of the web application, any default values present in the upload request, authentication parameters like HTTP headers or cookies, and the web page where the image will be displayed. Aside from the authentication parameters, all other artifacts are obtained through the execution of a web crawler on the application. The implementation of the crawler relies on the scrapy [55] framework which provides a lot of features, such as multithreading, request queue management, repeated requests prevention, and parsing of the web page's source. This is accomplished by having the crawler run in Scrapy's runtime so that the framework can have full control over the processes. As a result, the code for the crawler focuses on methods that find information on the returned web pages using XPath [56] and queue the following requests, while all of the management of performing those requests and creating new threads is handled by the framework.

As was mentioned in Section 4.5, there is a NoSQL [54] database used to pass information into and out of the crawler's context. This database is implemented using the TinyDB [57] library. It is used to provide the crawler with the necessary input and for the crawler to write the artifacts that it found on the web page. The input received from the user is the web page

where the file upload form is present and the authentication parameters. With this information, the crawler begins by parsing the upload form and saving any input fields that are filled by default in the database. Then, it uploads a sample image with a unique string embedded inside, saving the upload endpoint and the file's key within the form as well. Afterward, the crawling process begins on the same page that includes the upload form, checking all images on the page to verify if they include the sample string embedded in the upload. This process is repeated for all links that belong to the same domain until the uploaded sample image is found. Finally, the link to the page that displays the image is also saved. The reasoning behind saving the page instead of the actual location of the image on the web server is to account for a situation where the server changes the image's name on storage. This execution flow is depicted in Figure 5.1. There is also the option for the user to provide a list of possible download paths before execution. In this case, the crawler will only save the upload form's specification and will not search for the uploaded image.

5.2 File Parser

As was described in Section 4.3, for the uploaded files to maintain their validity, they must be maintained in a formal and well-defined manner such that they can be manipulated without sacrificing their structure. This is accomplished by the file parser component of the fuzzer. It is responsible for reading the files from the file system, parsing them into a tree-like structure, operating on said tree (these operations are detailed in Section 5.3) and writing them into the file system to eventually be uploaded. This sort of structure allows the images to be split into sections, which themselves are composed of blocks, as depicted in Figure 5.2. This way, by operating on the blocks layer, there is a guarantee that the sections retain their relative relationships, while operating on the sections layer, guarantees that blocks within each section retain their relative relationships. As a result, the overall layout of the file remains unaltered throughout its manipulation. Since the parser will be used with image files, the reading and writing processes vary slightly depending on the image format that is being worked on. However, the parser's API must be the same for all image formats.

5.2.1 Tree Structure

Keeping the structure described in Section 2.5 in mind, a parsed PNG will form a tree with 3 sections: a "meta section" containing all chunks between the signature and the first IDAT chunk, a "data section" containing all IDAT chunks, and an "end section" containing all chunks after the last IDAT chunk. Since the signature cannot be altered or moved in any way, it is

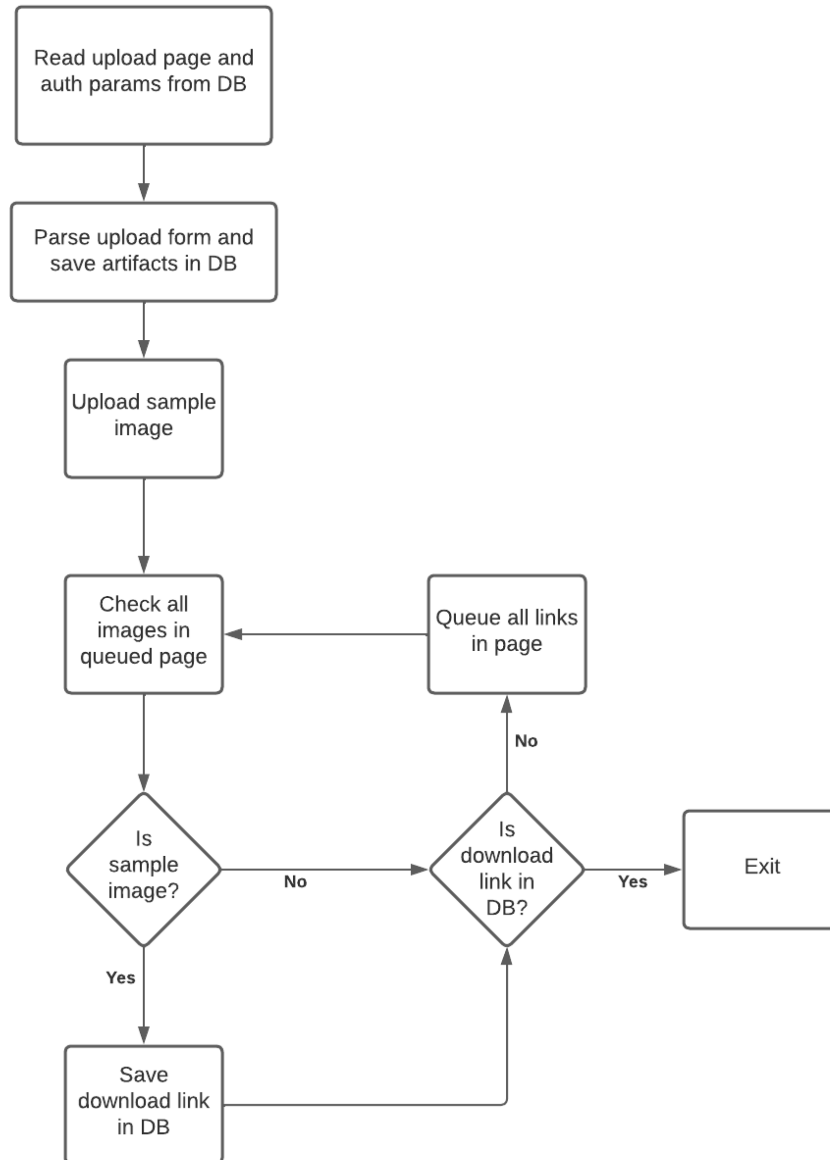

Figure 5.1: Flowchart of crawler's execution

stored in an extra node alongside the other sections. The resulting tree, depicted in Figure 5.3, allows chunks to be added, swapped, and deleted from the Meta and End sections at will without sacrificing the validity of the image, as long as the constraints related to the IHDR and IEND chunks remain true. The file is parsed by reading the signature and each of the chunks in their original order, placing them in the correct sections. To save some memory, only the Chunk Type and Chunk Data are saved. Rewriting the file in the file system is accomplished by writing the signature first, followed by each of the sections in order: Meta, Data, and End. Writing a section is done by writing each of the chunks in the order they are currently in, and recalculating the Length and CRC fields in the process.

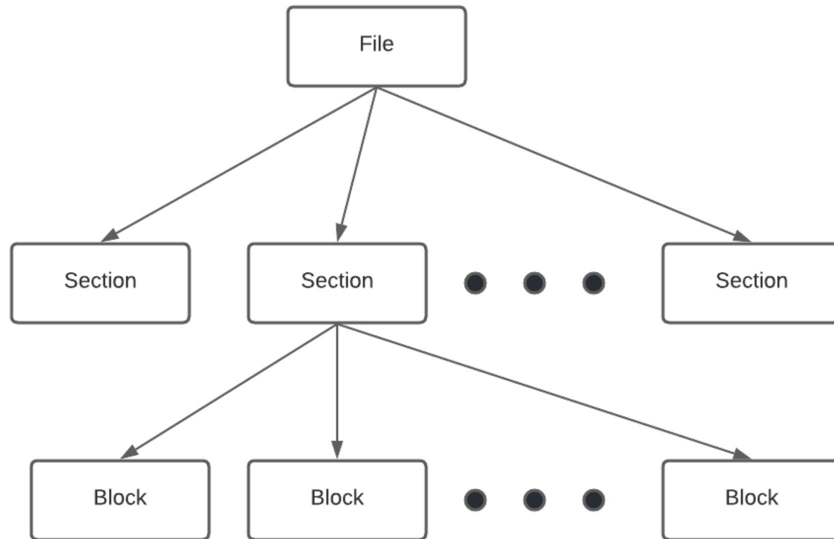

Figure 5.2: Format of tree structure

When it comes to JFIF images, the structural requirements, described in Section 2.6, are very similar to the ones for a PNG image. Thus, the resulting tree, depicted in Figure 5.4, follows the same organization as the one depicted in Figure 5.3. The one difference between them is the lack of a signature in a JFIF image. The reading and writing process is identical to the one described before, with 2 key differences. The entropy-coded data is also saved in the marker object, when present, and there is no longer a need to recalculate a CRC during serialization.

5.3 Genetic Algorithm

The final component of this project is the genetic algorithm that drives the program. This is where the actual fuzzing happens, where the various test cases are generated and evaluated against the target application. By using an evolutionary method, such as a genetic algorithm, the fuzzer can dynamically manipulate the images in real-time, depending on the results of the running tests. In the case of fuzzing file uploads, the chromosomes are HTTP requests that upload a file to a web application, while the genes for these chromosomes are the content of the file, represented as described in Section 5.2, and its name represented as a string. It starts with a population of sample, unaltered, images and performs consecutive manipulations to their content and name, to try and find the correct combination of genes that trigger a vulnerability. There are two variants of the genetic algorithm implemented for this project, a generic GA and the Fuzzy-Based Age Extension of Genetic Algorithms (FAexGA) [50][11], which are described

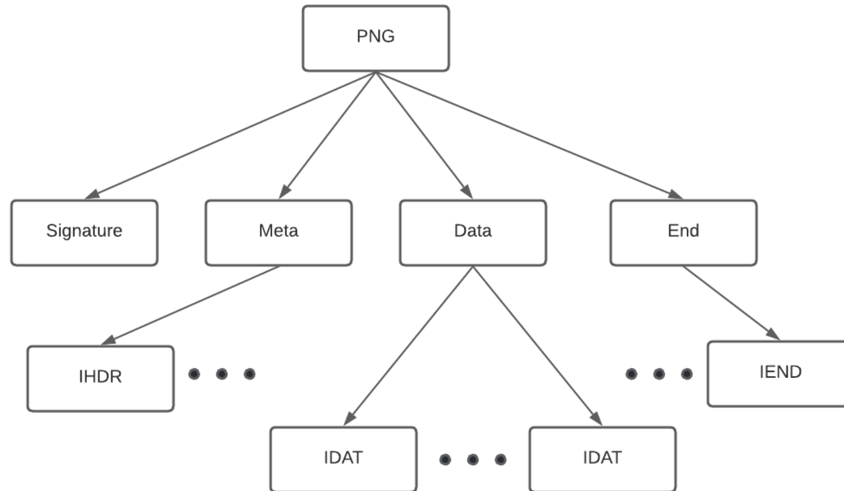

Figure 5.3: Format of parsed PNG

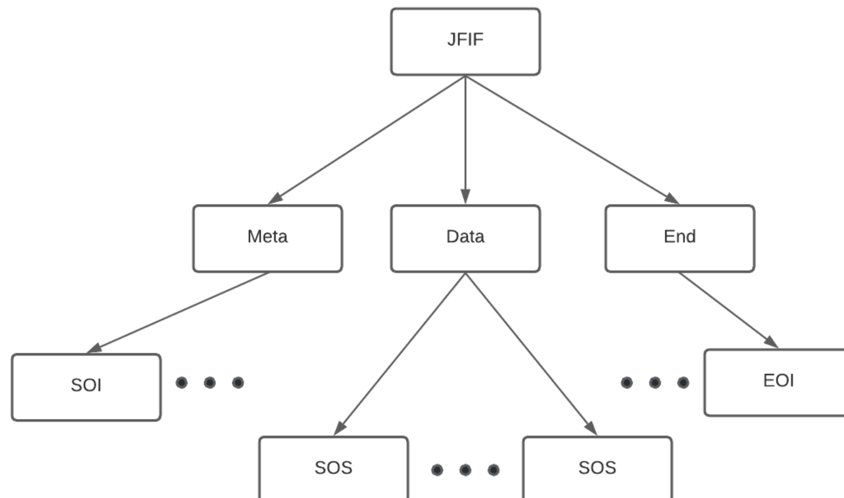

Figure 5.4: Format of parsed JFIF

in Sections 5.3.1 and 5.3.2, respectively.

5.3.1 Generic GA

For the generic GA, there were no major changes made to the overall workflow of the algorithm. Therefore, it will remain very similar to what was described in Section 2.4. Starting with the initial population, the seeds for the algorithm are a collection of PNG and JFIF images that vary in resolution. Since there are no mutations applied to this population, the first generation of tests will serve to detect any filters that may be applied on the image level. For example, if the application only accepts PNGs smaller than 512 by 512 pixels, any image that does not fill these criteria will fail to upload, causing it to have lower fitness and therefore a lower chance

to reproduce. This is a result of the selection process, which follows a standard roulette wheel method, where the chromosomes that were assigned higher fitness values are more likely to be selected to reproduce.

Chromosomes are selected in pairs, to perform the crossover operation[1]. This is accomplished by cloning one of the parents, and then passing genes from the second parent to the duplicate, dubbed the "offspring". First, the filename is selected. Since this is represented as a regular string, each parent has a 50% chance to pass on its filename as is. Any changes to this gene may occur in the mutation phase later on. Afterward, the contents of both files must be mixed without tampering with their integrity. The parent that was not cloned will randomly select a comment block and copy it over to the offspring, placing it in the same section that it was selected from.

After enough offspring have been generated to refill the population, the mutation phase can begin. Each new chromosome has a 30% chance to perform a mutation operation on itself. When this happens, there are three distinct actions taken. First, the filename is mutated. There is a list of possible extensions for the file to "choose" from, ranging from image extensions (.png, .jpeg,...) to extensions that could cause some form of code execution (.html, .php,...). These extensions are then randomly structured in one of seven possible ways: double extension, null separated, neutral suffix, casing change, semicolon separated, interpolated extension, and regular extension. Examples of these arrangements are found in Table 5.1. The second action is to inject a payload into the file's contents. A random payload is selected from a predefined list and is inserted in the file as a comment block at either the end of the "meta section" or the start of the "end section". As a result, none of the file's structural constraints are broken and it remains valid. It is also important to note that all of the payloads in the predefined list must attempt to cause the result that the fitness function tests for, which is explained later. The final action is to delete a random comment block from the file. This only occurs with a 5% chance, as to not constantly delete the algorithm's progress but still prevent files from increasing in size indefinitely.

The testing operation performed by the algorithm is the upload of each file to the web application and subsequent verification for any vulnerability that may have been triggered. The uploaded file is obtained through the serialization of the chromosome, while all other parameters required to fill the upload request are read from the database that was previously filled by the crawler. After the upload is complete, the fitness value for the corresponding chromosome is calculated. This is done through four distinct consecutive verifications, each of them increasing

[1]The same chromosome can be selected twice and crossover with itself

Arrangement	# of extensions selected	Example
Double extension	2	.jpeg.php
Null separated	2	.php%00.jpeg
Neutral suffix	1	.php/
Casing change	1	.PHP
Semicolon separated	2	.php;.jpeg
Interpolated extension	1	.ph.phpp
Regular extension	1	.php

Table 5.1: File extension mutations

the fitness value if successful. First, the status code of the upload response is verified. If this code is between 400 and 600, the upload is deemed failed and no fitness is awarded. Otherwise, a low value is summed to the fitness and the verifications proceed by attempting to download the file back. In case the user supplied a list of possible paths for the file to live on, the fuzzer will check these locations, by appending the filename to each path and attempting a download. If the crawler found a web page where the uploaded files are reflected on, the fuzzer will search through this page for the uploaded file. In both of these cases, the file is recognized by checking its contents for a unique sequence of letters that was injected into the file right before the upload. If the file is successfully downloaded and identified, a low value is summed to the fitness and the final verifications occur. The fuzzer will now check if the file triggered either XSS or PHP code execution. To verify XSS, the fuzzer uses the Selenium [33] library to simulate a browser and check for any alerts when opening the file's location. To check for PHP execution, the contents of the downloaded file are searched for the expected output of the payloads present in the predefined list. A high value of fitness is summed for each of these vulnerabilities triggered.

Executing all of these operations sequentially, as depicted in Figure 5.5, will result in a guided search for a combination of parameters that result in code execution through the upload of a malicious file.

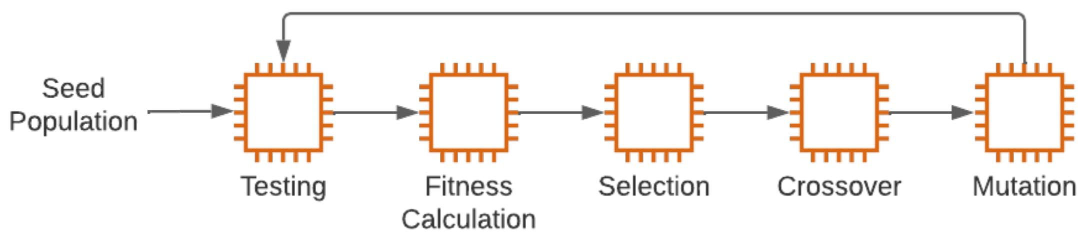

Figure 5.5: Workflow of generic GA

5.3.2 FAexGA

The second variant of the genetic algorithm that was implemented is the Fuzzy-Based Age Extension of Genetic Algorithms (FAexGA) [50][11]. Since the evaluation process described in the previous Section has a discrete progression, it is highly likely for the algorithm to be "stuck" between two of the steps that it measures. For example, the search space between a file that is successfully re-downloaded and a file that triggers a vulnerability is still quite large, but there are no verifications between these two scenarios. This causes difficulty for the algorithm to find the correct path onward. By implementing FAexGA [50][11], the likelihood of surpassing this obstacle increases. Based on the findings described in [11], this variation of the genetic algorithm has a high success rate in blackbox scenarios.

To implement this extension, the concepts of lifetime and reproduction ratio must be introduced into the algorithm. Lifetime will determine the number of generations that a chromosome lives for and is determined during the crossover, using the bi-linear method depicted in Equation 5.1 [50], with $\eta = \frac{1}{2}(MaxLT - MinLT)$ and the minimum and maximum lifetime parameters ($MinLT$ and $MaxLT$) assigned as 1 and 7, respectively.

$$LT(i) = \begin{cases} MinLT + \eta \frac{fitness[i] - MinFit}{AvgFit - MinFit} \ if \ AvgFit \geq fitness[i] \\ \\ \frac{1}{2}(MinLT + MaxLT) + \eta \frac{fitness[i] - AvgFit}{MaxFit - AvgFit} \ if \ AvgFit < fitness[i] \end{cases} \tag{5.1}$$

The reproduction ratio determines how many pairs of parents are selected to reproduce in the selection phase of the algorithm. The authors of [50] assigned it as 0.4, meaning that the number of pairs selected is roughly 40% of the number of chromosomes in the population. When two chromosomes are selected to crossover, the chances of it taking place are determined by the methodology depicted in Table 3.1. A filled version of this table is depicted in Table 5.2. It was determined that a chromosome is considered "middle-aged" if its age is within the two middle quarters of the population's age range. Any chromosome below or above this interval is considered "young" or "old", respectively. The probability values were determined as 10%, 60% and 100% for the "low", "medium" and "high" categories, respectively.

Finally, the method for selecting a parent is a random choice, instead of the previously described roulette wheel. The fitness of each chromosome is already reflected in its lifetime, which directly results in higher-fit genes surviving longer. Aside from the changes already mentioned, the rest of the algorithm functions the same as described in the previous Section.

Age II / Age I	$age < \frac{min+avg}{2}$	$\frac{min+avg}{2} < age < \frac{avg+max}{2}$	$\frac{avg+max}{2} < age$
$age < \frac{min+avg}{2}$	10%	60%	10%
$\frac{min+avg}{2} < age < \frac{avg+max}{2}$	60%	100%	60%
$\frac{avg+max}{2} < age$	10%	60%	10%

Table 5.2: Crossover probability percentages

Its overall workflow is depicted in Figure 5.6.

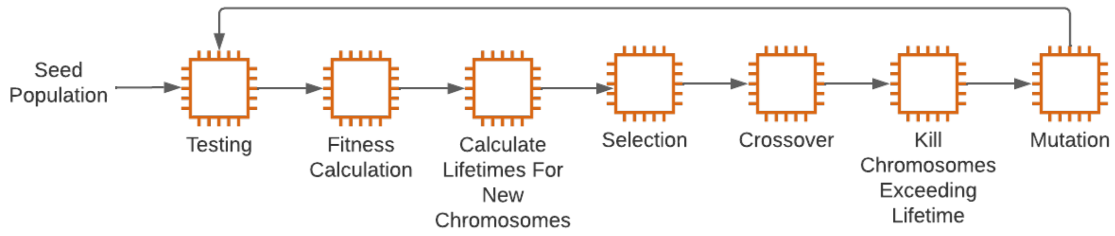

Figure 5.6: Workflow of FAexGA

Chapter 6

Evaluation and Results

Following the development of the tool, it is important to determine whether or not the requirements and work objectives were met. In order to assess this, the tool must be tested against various environments. For evaluation purposes, the tool was tested against 14 different scenarios spanning a custom web server, lab exercises provided by PortSwigger [29][58], a purposefully vulnerable web application known as Damn Vulnerable Web Application (DVWA) [59] and a set of production web applications with known vulnerabilities. These scenarios are listed in Table 6.1.

#	Test
1	Custom PHP Web Server
2	PortSwigger Lab - Web Shell Upload
3	PortSwigger Lab - Content-Type bypass
4	PortSwigger Lab - Path Traversal
5	PortSwigger Lab - Extension Blacklist
6	PortSwigger Lab - Obfuscated Extension
7	PortSwigger Lab - Polyglot Web Shell
8	PortSwigger Lab - Race Condition
9	DVWA - Low Difficulty
10	DVWA - Medium Difficulty
11	DVWA - High Difficulty
12	Crater [60] - CVE-2021-4080
13	CMS Made Simple [61] - CVE-2022-23906
14	WikiDocs [62] - CVE-2022-23375

Table 6.1: List of test scenarios

The following sections describe the outcomes of these tests, as well as an analysis of the results and their implications. First, the functionality of the tool will be evaluated, to determine whether or not it is successful in discovering vulnerabilities. Afterward, the results are compared to those of an existing tool designed to discover vulnerabilities in file upload endpoints.

6.1 Test Results

In this section, an analysis is performed on the efficacy of the developed tool. The crawler and the fuzzer are evaluated separately before a complete analysis is performed. Furthermore, the applicability of FAexGA [50][11] is discussed to determine whether or not its implementation improved upon the results with a generic GA.

6.1.1 Crawler

The purpose of implementing the web crawler is twofold: parsing the upload form's specification and finding where the uploaded image is reflected on the web application. In terms of parsing the form, the crawler was extremely successful, being able to correctly collect all necessary information regarding the upload forms in all tests, except for test #12. As a result, the fuzzer was able to correctly perform upload requests to tested applications in all but one of the tested scenarios, as depicted in Table 6.2. Test #12 represents a subset of web applications that caused all modules to fail, thus the results for this test will be fully discussed in Section 6.1.4, instead of one module at a time like the other tests.

#	Test	Success
1	Custom PHP Web Server	Yes
2	PortSwigger Lab - Web Shell Upload	Yes
3	PortSwigger Lab - Content-Type bypass	Yes
4	PortSwigger Lab - Path Traversal	Yes
5	PortSwigger Lab - Extension Blacklist	Yes
6	PortSwigger Lab - Obfuscated Extension	Yes
7	PortSwigger Lab - Polyglot Web Shell	Yes
8	PortSwigger Lab - Race Condition	Yes
9	DVWA - Low Difficulty	Yes
10	DVWA - Medium Difficulty	Yes
11	DVWA - High Difficulty	Yes
12	Crater [60] - CVE-2021-4080	No
13	CMS Made Simple [61] - CVE-2022-23906	Yes
14	WikiDocs [62] - CVE-2022-23375	Yes

Table 6.2: Performance of crawler in parsing upload forms

The second purpose of the crawler is much more challenging than the first. As such, the results are not as positive. Tests #1-8 and #13-14 were successful, as the crawler was able to correctly locate the uploaded image reflected on the web application. Tests #9-11 represent the scenarios for the application DVWA [59]. This application does not reflect uploaded images back on the web page and instead only shows a link pointing to the location where the image was stored. Since the location of the image is reflected, the user may provide this link to the crawler

beforehand, bypassing this limitation at the cost of extra actions by the user. These results show that the crawler's implementations proved to be very effective, being able to complete its objectives in most of the test cases. In the cases where the image is not reflected, it was expected for the crawler to fail, since its goal, a page reflecting the uploaded image, does not exist. The implemented alternative of supplying a link to the uploads directory before the crawler's execution also proved to be an effective method to circumvent this limitation with minimal cost.

In Section 4.4, it is mentioned that implementing the crawler is preferred over the model inference technique, due to the assumption that uploaded images are reflected on web pages "near" the upload page. In all the tests where the crawler was able to identify the reflected image, the uploaded image was reflected on the same page that contained the upload form. As such, this assumption remained true and the computational cost of the crawler can be determined as extremely reduced since it only needs to visit one page, and thus performed a minimal number of requests, as depicted in Figure 6.1.

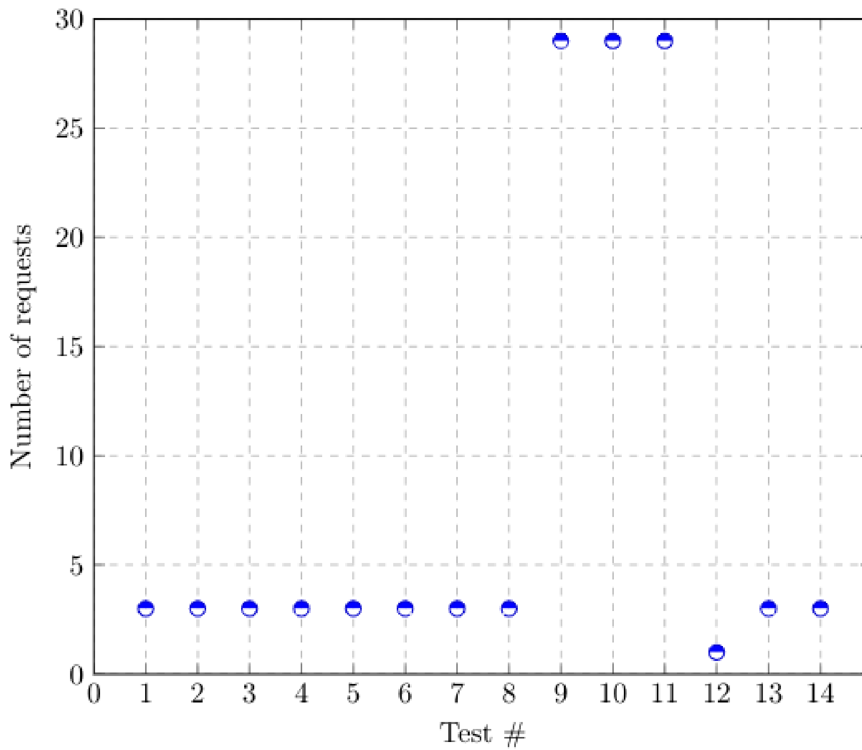

Figure 6.1: Number of requests performed by the crawler

6.1.2 Fuzzer

The data depicted in Figures 6.2 and 6.3 was collected by running the fuzzer 5 times against each test. This data shows that the fuzzer module powered by the genetic algorithm was able

to successfully identify and report back to the user, vulnerabilities present in tests #1-4, #6-7, #9-10, and #14 with high success rates. While for most of the tests the algorithm was able to converge on a solution within 10 generations, for test #6 it took around 15 to 20 generations to even generate a chromosome that represented a solution, with a few more generations to converge afterward. Examples of these progressions are also depicted in Figures 6.2 and 6.3. It is not usual for genetic algorithms to converge this quickly, however, this result can be attributed to the small search space that the problem presents. As explained in Section 5.3, each chromosome only has two genes with limited possibilities and due to the nature of genetic algorithms, continuously eliminating bad combinations will cause it to converge quickly. A problematic result would be if even with a genetic algorithm a brute force approach would be more efficient, which is not the case. When it comes to test #6, this particular case requires the algorithm to generate a filename with a very specific extension. Since there is no way to calculate progress on this gene, as the extension can either be right or wrong with no in-between, after figuring out the other parameters, the algorithm has to randomly guess the extension. This is not the case for the other tests, since their solution did not require one specific extension. When it comes to the injection of payloads in the files, the parser showed no problems at all and the algorithm was able to inject and transfer payloads between files without sacrificing their validity.

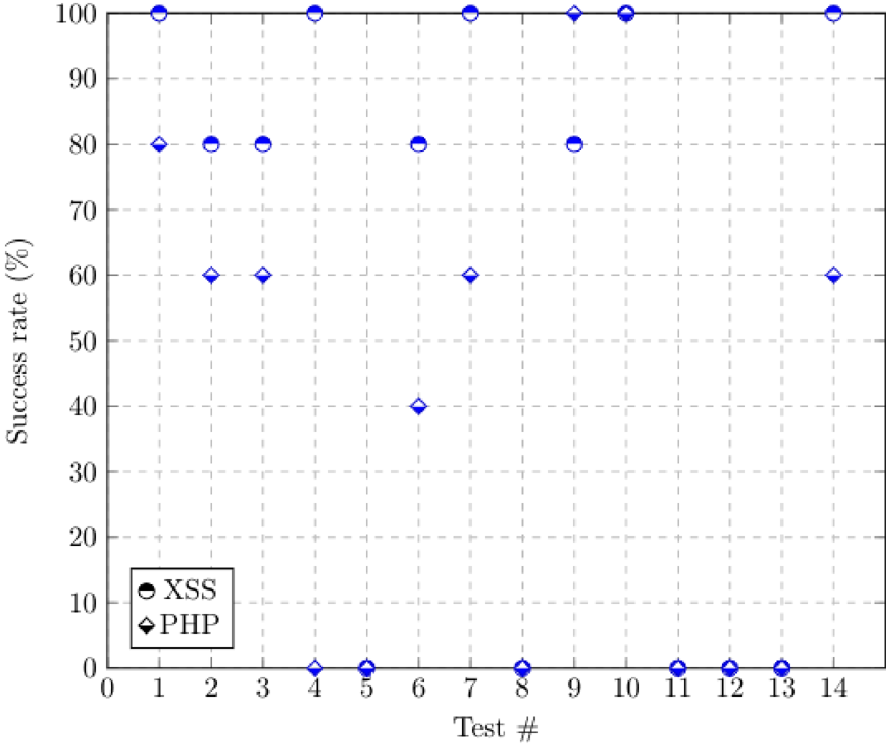

Figure 6.2: Fuzzer success rate

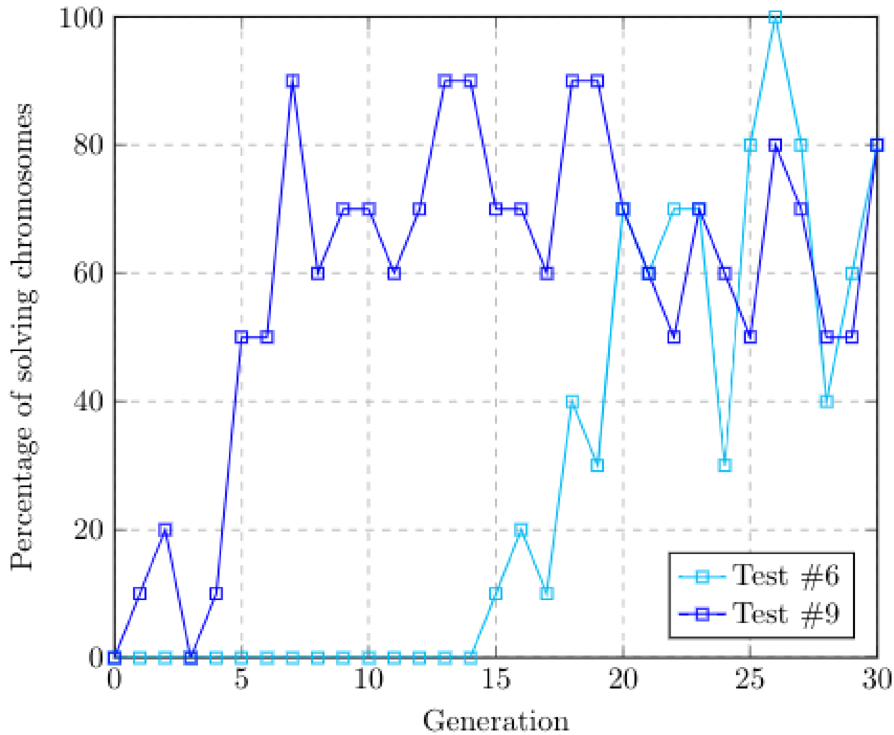

Figure 6.3: Evolution of Generic Algorithm

The remaining tests (#5, #8, #11, #12 and #13) were not successful. As mentioned before, test #12 will be discussed in Section 6.1.4. Tests #5, #8, #11, and #13 all failed for the same reason. To trigger the vulnerability present in these cases it is necessary to perform an additional action in the web application after uploading the malicious image, like renaming the file for example. Since this action is unpredictable and cannot be automated, it is considered to be out of scope for the developed tool to attempt to trigger these vulnerabilities, as stated in the requirements that the tool should be autonomous. As a result, the developed tool cannot trigger these vulnerabilities and thus cannot report them back to the user.

As of now, all of these results were obtained by running the fuzzer with the generic version of the genetic algorithm. The FAexGA [50][11] version of the algorithm, unfortunately, presented worse results than its counterpart. Since this version allows the size of the population to vary over time, when the algorithm converges, the population starts to decrease until one final solution is all that remains. When running this algorithm against test #2, it showed that a solution was discovered as fast, or faster, than the generic version in most executions. However, the solving combination of genes would quickly die out before the population converged, resulting in a non-solving chromosome being deemed the “solution”, as depicted in Figure 6.4. After repeating this test multiple times, it showed that the solving chromosome only survived to the end of the algorithm in 20% of the runs. This outcome did not change when running against other tests.

As a result, the generic version of the algorithm is considered more appropriate to this problem and the agreed-upon solution.

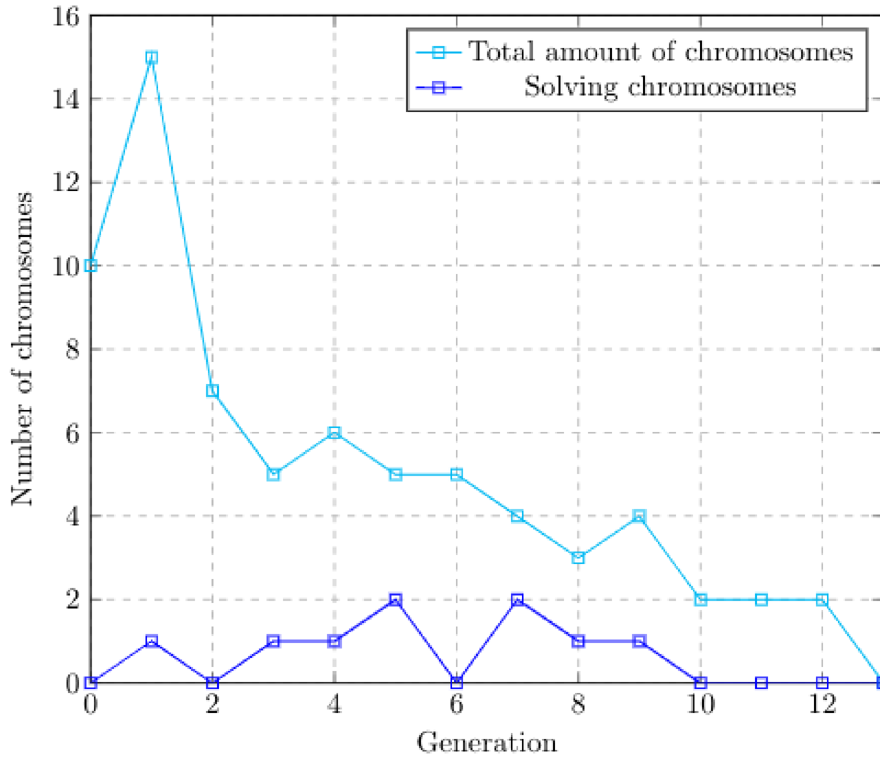

Figure 6.4: Evolution of FAexGA in Test #2

These results showed the pairing of a genetic algorithm with a custom file parser as a valid method for identifying vulnerabilities in file upload endpoints. Counter to what was predicted in Chapter 4, the blackbox approach did not prevent the genetic algorithm from functioning properly. Furthermore, it was extremely fast in its execution. The reasoning behind the disappointing results of the FAexGA [50][11] implementation can be attributed to either an inappropriate application of this technique or a misconfiguration of its parameters. Since this version was implemented as a method to combat the difficulty of running a generic GA in a blackbox environment, the cause of its failure was not further researched, as the generic GA did show positive results.

6.1.3 Full Stack

When the modules are combined, there are two important results to be addressed. First, there are two test cases where the fuzzer succeeds, but the crawler does not, such as tests #9 and #10. For the tool to fully function against tests of this nature, the crawler needs to be dismissed, which means the user must supply the tool with possible download paths before execution. With this method, the tool works properly. Secondly, for tests #6 and #14, each of the modules works

properly when separated, but not when run in tandem. These two tests showcase an interesting scenario that causes this behavior, as the web applications allow for the upload of malicious images, but only reflect on the page images that are not malicious. As a result, the crawler can successfully identify the page that reflects the initial upload, as it is a safe image, but the fuzzer cannot download any of its uploaded images because it will keep checking the page returned by the crawler. To prevent this outcome, the user must once again supply the download paths themselves. However, in tests #9 and #10, it is trivial to note that the application does not reflect any of the uploaded images, and thus it is easier for the user to adjust their use of the tool. In the case where only malicious images are not reflected, it is much harder to identify this behavior, as the user must monitor the tool and notice that it is failing all of its downloads, and even then it might just be a case where the application is not vulnerable. Even though it goes against the requirement for autonomy to monitor the results in real-time, it is unlikely for this specification to change in an application, and thus it can be seen as a one-time adjustment.

The coupling of the web crawler and the fuzzer also proved to be very effective. The use of a shared database to transfer information showed no complications and proved to be appropriate for this scenario. Once again, although there are some cases where this pairing was unable to function, the alternative methodology of providing download links was able to counteract these cases with minimal cost. The most important takeaway from these results is the case where malicious images are not reflected on the application since the identification of this scenario is in itself a challenge for the user. However, after being identified, adjusting the tool is just as simple as the other cases.

6.1.4 Overview

Overall, the tool showed positive results, as it was able to identify vulnerabilities in most of the test cases. Even when one of the modules does not function properly for a certain application, it is possible to adjust the workflow to bypass this limitation, aside from the test cases that fell out of scope. The exception to this statement is test #12. Unlike all of the other tests, this scenario presented a single page application, with dynamic pages rendered by the browser. This immediately caused the crawler to fail, as the page returned by the backend of the application does not represent what a user interacts with in the browser. A possible way to bypass this would be to couple the crawler with the Selenium [33] library, to access the source code of the fully rendered page. However, this effort would not have solved the overall issue. As a dynamic page, the method for uploading a file to the backend can be designed in any way imaginable by the programmer, creating unpredictability. In this situation, there is no upload form to parse.

Furthermore, the format of the upload request is also unpredictable, as the request and file can be arranged in numerous ways, such as a straight PUT request to the server or even JSON [63] with the file encoded in Base64 [64] or even hexadecimal form. The only method that could adapt to this scenario would be to intercept the upload request with a proxy, in a workflow similar to that of BurpSuite [28]. This methodology, however, would require a big sacrifice in the autonomy of the tool, as will be discussed in the following Section.

6.2 Comparison

The tool that was selected to compare the results with is the PortSwigger Upload Scanner [65]. This tool is an extension for the BurpSuite [28] program and as mentioned before, some of the problems portrayed in the testing process could be addressed with a workflow similar to the one of BurpSuite [28]. Furthermore, the tool is developed by PortSwigger [29], which provides some of the used test cases. As a result, this upload scanner appeared as an appropriate counterpart to perform this comparison. It contains a variety of modules, some of which aim to detect XSS and PHP code execution, just like the tool developed in this work. Unlike the developed tool, however, it does not automatically download uploaded files. There is a way to enable this feature by either providing a download link beforehand or sending a "preflight" request and adding markers to the response to dictate where the download link is. Furthermore, there is a feature named "FlexInjector", which when configured correctly allows the scanner to function with single page applications. It is also not powered by a genetic algorithm and follows instead a predetermined sequence of upload requests, depending on the enabled modules.

The scanner was run against the same tests that were listed at the start of the Chapter, with the download feature enabled. It was unable to identify vulnerabilities in tests #5, #8, #11, and #13-14. For tests #5, #8, #11, and #13 it was determined that they were out of scope due to the necessity of an extra action besides the upload and it is therefore expected for these tests to fail. There is, however, an interesting result. In test #5, the extra action required was to override a configuration file on the web server before uploading the malicious file. Observing the logs of the scanner shows that it did attempt this action, but was unable to do it in a particular manner that this server was vulnerable to. This demonstrates that attempting to automate an extra action besides the upload is in itself not a trivial task, further proving why these tests should be out of the scope of automated tools. Test #14 fails for a different reason. Although the download link is provided to the scanner beforehand, the redownload request does not account for changes to the uploaded filename. In this particular case, all uploaded filenames are changed to lowercase by the web server, causing the redownload to fail. The developed tool is

unaffected by this particular case because all uploaded filenames are already lowercase, however, it does showcase a limitation. If the images are not reflected and the filenames are changed, there is no way to guess where it was stored, and thus, no way to automate its download. It is important to note that the scanner was able to identify vulnerabilities in test #12. The way it circumvents the limitations described in Section 6.1.4 is through its "FlexInjector" feature. When correctly configured, the scanner can identify where in the upload request the file is, and how it is encoded, allowing it to function properly with single page applications.

Although the scanner presented results very similar to the developed tool, with the added ability to test single page applications, there is one big limitation. Due to BurpSuite's [28] proxy workflow, it lacks the autonomy to be run unsupervised. The user must capture the upload request and configure the scan in real-time. Furthermore, there is no real "report" back to the user. While the developed tool explicitly reports which requests caused which vulnerabilities, the scanner presents a list of request/response pairs and it's up to the user to interpret whether or not a vulnerability was detected. With multiple modules enabled, this list of pairs easily increases to the hundreds, which is quite cumbersome to analyze. It is also important to note that when applicable, the ability to have the crawler replace providing a download link further decreases the configuration necessary for the developed tool to run. The upload scanner does not perform downloads by default and always requires the extra configuration to enable this feature.

Chapter 7

Conclusions

After researching the topic, applying fuzzing techniques to web application testing is the most appropriate method, since it performs testing from the perspective of the user. Smart fuzzing will allow test case generation to be more efficient by eliminating invalid inputs from the start. Pairing this technique with genetic algorithms means that the fuzzer will be self-guiding and will not require input models such as formal grammars. Finally applying this methodology to the generation and manipulation of files will play to the strengths of fuzzing algorithms, taking advantage of its full potential. The purpose of this project was to assess the viability of applying smart evolutionary fuzzing on testing file upload endpoints while minimizing user interaction to increase autonomy. Whether or not this was successful and within the requirements will be discussed in the following Sections.

7.1 Achievements

The fuzzing module proved to be not only effective but also efficient in detecting vulnerabilities within the defined scope. Aside from single page applications, all obstacles were able to be overcome with minimal sacrifice or change in the requirements. The application of genetic algorithms for testing file uploads is valid and yielded positive results. However, implementing the Fuzzy-Based Age Extension of Genetic Algorithms (FAexGA) [50][11] ended up decreasing performance and thus, file uploads are not a proper scenario for implementing this idea. Furthermore, the file parser was very effective in injecting files with malicious payloads while maintaining their structure and validity, and since this technique did not depend on the injected payloads, extending the tool to detect more vulnerabilities requires minimal code changes.

The implemented crawler was able to serve its various purposes in most of the tested scenarios. Once again, aside from single page applications, the parsing of the upload forms worked

well to provide the fuzzer with the necessary parameters. The search for uploaded images also proved to be effective when it is applicable, although a simple alternative was provided that was able to bypass this issue.

7.2 Future Work

It remains an open problem to apply these methods in a way that can test single page applications. Their unpredictability in the use of dynamic pages poses a challenge for any attempt at generically automating user interaction. Although tools like BurpSuite [28] are viable in these scenarios, the sacrifice in autonomy and increase in user interaction and configuration means that the process cannot be automated. A tool that can apply the methods described in this work with a high level of autonomy and the ability to target single page applications would mean a step forward in the realm of fuzzing and web application testing. This can only be achieved with the use of techniques that could analyze a dynamic page and accurately determine the upload method (backend endpoint, request format, file encoding, etc).

Bibliography

[1] Owasp top ten web application security risks — owasp, . URL https://owasp.org/www-project-top-ten/. [Accessed Dec 27th, 2021].

[2] Iso - international organization for standardization. URL https://www.iso.org/home.html. [Accessed Dec 30th, 2021].

[3] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing 1994 4:2*, 4:65–85, 6 1994. ISSN 1573-1375. doi: 10.1007/BF00175354. URL https://link.springer.com/article/10.1007/BF00175354.

[4] What is a web application? how it works, benefits and examples — indeed.com. URL https://www.indeed.com/career-advice/career-development/what-is-web-application. [Accessed Jan 12th, 2022].

[5] What is sql injection? tutorial examples — web security academy. URL https://portswigger.net/web-security/sql-injection. [Accessed Dec 27th, 2021].

[6] Cross site scripting (xss) software attack — owasp foundation. URL https://owasp.org/www-community/attacks/xss/. [Accessed Dec 27th, 2021].

[7] What is taint analysis and why should i care? - dzone security. URL https://dzone.com/articles/what-is-taint-analysis-and-why-should-i-care. [Accessed Jan 13th, 2022].

[8] Fuzzing — owasp foundation, . URL https://owasp.org/www-community/Fuzzing. [Accessed Nov 17th, 2021].

[9] Our guide to fuzzing — f-secure. URL https://www.f-secure.com/us-en/consulting/our-thinking/15-minute-guide-to-fuzzing. [Accessed Jan 12th, 2022].

[10] Bart Miller. Computer sciences department university of wisconsin-madison cs 736 bart miller fall 1988 project list. 1988.

[11] Mark Last, Shay Eyal, and Abraham Kandel. Effective black-box testing with genetic algorithms. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3875 LNCS:134–148, 2006. ISSN 03029743. doi: 10.1007/11678779_10. URL https://www.researchgate.net/publication/221471850_Effective_Black-Box_Testing_with_Genetic_Algorithms.

[12] Van Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. *ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 543–553, 8 2016. doi: 10.1145/2970276.2970316. URL https://www.researchgate.net/publication/310819146_Model-based_whitebox_fuzzing_for_program_binaries.

[13] Lwin Khin Shar, Ta Nguyen Binh Duong, Lingxiao Jiang, David Lo, Wei Minn, Glenn Kiah Yong Yeo, and Eugene Kim. Smartfuzz: An automated smart fuzzing approach for testing smartthings apps. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 2020-December:365–374, 12 2020. ISSN 15301362. doi: 10.1109/APSEC51365.2020.00045.

[14] Van Thuan Pham, Marcel Bohme, Andrew E. Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 47:1980–1997, 11 2018. ISSN 19393520. doi: 10.1109/TSE.2019.2941681. URL https://arxiv.org/abs/1811.09447v1.

[15] american fuzzy lop, . URL https://lcamtuf.coredump.cx/afl/. [Accessed Dec 15th, 2021].

[16] Yuwei Li, Shouling Ji, Chenyang Lv, Yuan Chen, Jianhai Chen, Qinchen Gu, and Chunming Wu. V-fuzz: Vulnerability-oriented evolutionary fuzzing. 1 2019. URL https://arxiv.org/abs/1901.01142v1.

[17] Rfc 2083 - png (portable network graphics) specification version 1.0. URL https://datatracker.ietf.org/doc/html/rfc2083#section-3. [Accessed Jul 28th, 2022].

[18] Cyclic redundancy checks. URL https://www.mathpages.com/home/kmath458.htm. [Accessed Jul 28th, 2022].

[19] Eric Hamilton. Jpeg file interchange format. 1992.

[20] Terminal equipment and protocols for telematic services information technology-digital

compression and coding of continuous-tone still images-requirements and guidelines recommendation t.81. 1993.

[21] Iso - iso/iec 10918-1:1994 - information technology — digital compression and coding of continuous-tone still images: Requirements and guidelines. URL https://www.iso.org/standard/18902.html. [Accessed Jul 29th, 2022].

[22] Wapiti : a free and open-source web-application vulnerability scanner in python for windows, linux, bsd, osx. URL https://wapiti.sourceforge.io/. [Accessed Dec 25th, 2021].

[23] Nvd - cve-2021-44228. URL https://nvd.nist.gov/vuln/detail/CVE-2021-44228. [Accessed Dec 25th, 2021].

[24] Arachni/arachni: Web application security scanner framework. URL https://github.com/Arachni/arachni. [Accessed Dec 25th, 2021].

[25] w3af - open source web application security scanner. URL http://w3af.org/. [Accessed Dec 25th, 2021].

[26] Owasp zap. URL https://www.zaproxy.org/. [Accessed Dec 26th, 2021].

[27] Owasp foundation — open source foundation for application security, . URL https://owasp.org/. [Accessed Dec 26th, 2021].

[28] Burp suite - application security testing software - portswigger, . URL https://portswigger.net/burp. [Accessed Dec 26th, 2021].

[29] Web application security, testing, scanning - portswigger, . URL https://portswigger.net/. [Accessed Dec 26th, 2021].

[30] Issue definitions - portswigger, . URL https://portswigger.net/kb/issues. [Accessed Dec 26th, 2021].

[31] Dong Wang, Xiaosong Zhang, Ting Chen, and Jingwei Li. Discovering vulnerabilities in cots iot devices through blackbox fuzzing web management interface. *Security and Communication Networks*, 2019, 2019. ISSN 19390122. doi: 10.1155/2019/5076324. URL https://www.researchgate.net/publication/337028643_Discovering_Vulnerabilities_in_COTS_IoT_Devices_through_Blackbox_Fuzzing_Web_Management_Interface.

[32] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. *MobiSys 2014*

- *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, pages 204–217, 2014. doi: 10.1145/2594368.2594390. UI automation: write PUMAscript that instruments app to guide crawler.

[33] Selenium. URL `https://www.selenium.dev/`. [Accessed Dec 15th, 2021].

[34] Fabien Duchene, Roland Groz, Sanjay Rawat, and Jean Luc Richier. Xss vulnerability detection using model inference assisted evolutionary fuzzing. *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012*, pages 815–817, 2012. doi: 10.1109/ICST.2012.181. URL `https://www.researchgate.net/publication/234834141_XSS_Vulnerability_Detection_Using_Model_Inference_Assisted_Evolutionary_Fuzzing`.

[35] Cwe - 2021 cwe top 25 most dangerous software weaknesses. URL `https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html`. [Accessed Jan 12th, 2022].

[36] Matthias Buchler, Karim Hossen, Petru Florin Mihancea, Marius Minea, Roland Groz, and Catherine Oriat. Model inference and security testing in the spacios project. *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014 - Proceedings*, pages 411–414, 2014. doi: 10.1109/CSMR-WCRE.2014.6747207.

[37] Fabien Duchene, Sanjay Rawat, Jean Luc Richier, and Roland Groz. Kameleonfuzz: Evolutionary fuzzing for black-box xss detection. *CODASPY 2014 - Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, pages 37–48, 2014. doi: 10.1145/2557547.2557550.

[38] Fabien Duchene, Sanjay Rawat, Jean Luc Richier, and Roland Groz. Ligre: Reverse-engineering of control and data flow models for black-box xss detection. *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 252–261, 2013. ISSN 10951350. doi: 10.1109/WCRE.2013.6671300. URL `https://www.researchgate.net/publication/255821637_LigRE_Reverse-Engineering_of_Control_and_Data_Flow_Models_for_Black-Box_XSS_Detection`.

[39] François Gauthier, Behnaz Hassanshahi, Benjamin Selwyn-Smith, Trong Nhan Mai, Max Schlüter, and Micah Williams. Backrest: A model-based feedback-driven greybox fuzzer for web applications. 8 2021. URL `https://arxiv.org/abs/2108.08455v1`.

[40] Aflplusplus/aflplusplus: The fuzzer afl++ is afl with community patches, qemu 5.1 upgrade, collision-free coverage, enhanced laf-intel redqueen, aflfast++ power schedules, mopt

mutators, unicorn_mode, and a lot more!, . URL https://github.com/AFLplusplus/AFLplusplus. [Accessed Dec 15th, 2021].

[41] libfuzzer – a library for coverage-guided fuzz testing. — llvm 13 documentation. URL https://llvm.org/docs/LibFuzzer.html. [Accessed Dec 15th, 2021].

[42] jtpereyda/boofuzz: A fork and successor of the sulley fuzzing framework. URL https://github.com/jtpereyda/boofuzz. [Accessed Dec 15th, 2021].

[43] Openrce/sulley: A pure-python fully automated and unattended fuzzing framework. URL https://github.com/OpenRCE/sulley. [Accessed Dec 15th, 2021].

[44] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. Finding software vulnerabilities by smart fuzzing. *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation, ICST 2011*, pages 427–430, 2011. doi: 10.1109/ICST.2011.48.

[45] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. volume 5, pages 3–4, 2005.

[46] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). *Proceedings - IEEE Symposium on Security and Privacy*, pages 317–331, 2010. ISSN 10816011. doi: 10.1109/SP.2010.26.

[47] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. 1 2016. Apply neural networks to structured data like graphs and trees.

[48] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. *Proceedings of the ACM Conference on Computer and Communications Security*, pages 363–376, 10 2017. ISSN 15437221. doi: 10.1145/3133956.3134018. URL http://dx.doi.org/10.1145/3133956.3134018. Apply graph embedding networks to code similarity detection.

[49] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. *Proceedings of the ACM Conference on Computer and Communications Security*, 24-28-October-2016:480–491, 10 2016. ISSN 15437221. doi: 10.1145/2976749.2978370. URL https://experts.syr.edu/en/

publications/scalable-graph-based-bug-search-for-firmware-images. Attributed Control Flow Graph: control graph that gives attributes to the BBs.

[50] Mark Last and Shay Eyal. A fuzzy-based lifetime extension of genetic algorithms. *Fuzzy Sets and Systems*, 149:131–147, 1 2005. ISSN 01650114. doi: 10.1016/J.FSS.2004.07.011. URL https://dlnext.acm.org/doi/abs/10.1016/j.fss.2004.07.011.

[51] Jaroslaw Arabas, Zbigniew Michalewicz, and Jan Mulawka. Gavaps - a genetic algorithm with varying population size. *IEEE Conference on Evolutionary Computation - Proceedings*, 1:73–78, 1994. doi: 10.1109/ICEC.1994.350039.

[52] Karla L Hoffman, Manfred Padberg, and Giovanni Rinaldi. Traveling salesman problem. 2001. doi: 10.1007/1-4020-0611-X_1068.

[53] Imam Riadi and Eddy Irawan Aristianto. An analysis of vulnerability web against attack unrestricted image file upload. *Computer Engineering and Applications Journal*, 5:19–28, 1 2016. ISSN 2252-4274. doi: 10.18495/COMENGAPP.V5I1.161. URL https://www.researchgate.net/publication/303873848_An_Analysis_of_Vulnerability_Web_Against_Attack_Unrestricted_Image_File_Upload.

[54] What is nosql? nosql databases explained — mongodb. URL https://www.mongodb.com/nosql-explained. [Accessed Jul 21st, 2022].

[55] Github - scrapy/scrapy: Scrapy, a fast high-level web crawling scraping framework for python. URL https://github.com/scrapy/scrapy. [Accessed Jul 29th, 2022].

[56] Xpath — mdn. URL https://developer.mozilla.org/en-US/docs/Web/XPath. [Accessed Jul 29th, 2022].

[57] Github - msiemens/tinydb: Tinydb is a lightweight document oriented database optimized for your happiness :). URL https://github.com/msiemens/tinydb. [Accessed Jul 21st, 2022].

[58] File uploads — web security academy, . URL https://portswigger.net/web-security/file-upload. [Accessed Sep 16th, 2022].

[59] digininja/dvwa: Damn vulnerable web application (dvwa). URL https://github.com/digininja/DVWA. [Accessed Sep 16th, 2022].

[60] crater-invoice/crater: Open source invoicing solution for individuals businesses. URL https://github.com/crater-invoice/crater. [Accessed Sep 16th, 2022].

[61] Open source content management system : : Cms made simple. URL http://www.cmsmadesimple.org/. [Accessed Sep 16th, 2022].

[62] Wiki—docs. URL https://www.wikidocs.it/. [Accessed Sep 16th, 2022].

[63] Json. URL https://www.json.org/json-en.html. [Accessed Sep 19th, 2022].

[64] Base64 - mdn web docs glossary: Definitions of web-related terms — mdn. URL https://developer.mozilla.org/en-US/docs/Glossary/Base64. [Accessed Sep 19th, 2022].

[65] Portswigger/upload-scanner: Http file upload scanner for burp proxy. URL https://github.com/PortSwigger/upload-scanner. [Accessed Sep 30th, 2022].