Automated Smart Fuzzing Vulnerability Testing

João Coutinho, j.afonso.coutinho@tecnico.ulisboa.pt, Instituto Superior Técnico

***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.**

***Index Terms*—Web Applications, Fuzzing, Vulnerability Testing, Genetic Algorithms**

I. 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.

A. 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 II-B. 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.

II. Background

In the following sections, the concepts necessary for the understanding of the proposed work are introduced. In Section II-A, the concept of web applications is explained, their advantages, architecture, and main vulnerabilities. Section II-B 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 II-C introduces what genetic algorithms are and how they work.

A. 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 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.

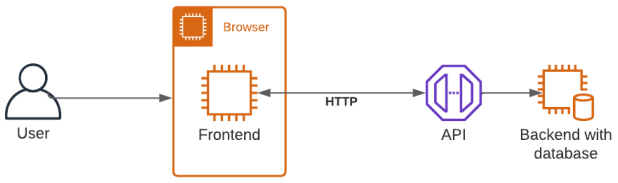

Fig. 1. Web application architecture

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.

B. 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 [7]. 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 [8]:

- *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 [9]. 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* [8].

- *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 *black-box*, having no knowledge of the application source code or structure [10]. Others use a *whitebox* approach, which takes advantage of knowing source code to track the fuzzer's results and progressively increase code coverage [11] [12]. 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 [13] [14] [15].

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* [8].

- *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.

C. 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.

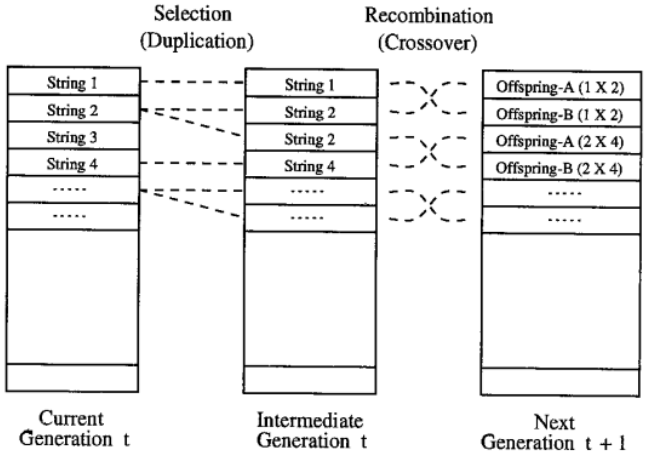

Fig. 2. Selection and crossover stages of a genetic algorithm [3]

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. 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.

D. PNG

PNG, or Portable Network Graphics, is a file format that allows lossless and portable storage of images [16]. 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 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.

E. 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 [17] [18]. 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. 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 [19]. 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.

III. Implementation

In this chapter, the implementation of the proposed solution, depicted in Figure 3, is described in detail. Section III-A describes the web crawler used to provide the other components with information necessary for their execution. Section III-B specifies the file parser and how it operates on various file types. And finally, Section III-D performs an overview of the genetic algorithm and its various parameters, as well as a comparison between two different variations of this technique.

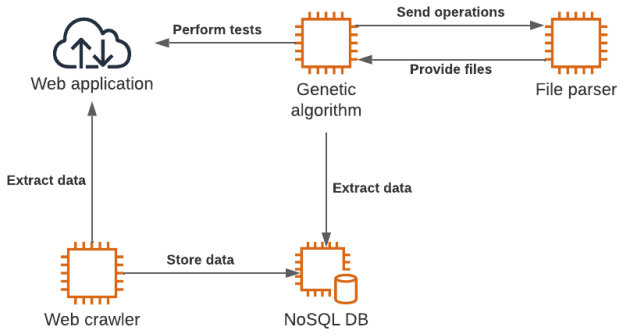

Fig. 3. Architecture of implementation

A. 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 framework [20] 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 [21] and queue the following requests, while all of the management of performing those requests and creating new threads is handled by the framework.

In order to pass information in and out of the crawler's context, a NoSQL [22] database is used. This database is implemented using the TinyDB [23] 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. 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.

B. File parser

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 III-D) 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. 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.

C. Tree Structure

Keeping the structure described in Section II-D 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 stored in an extra node alongside the other sections. The resulting tree 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.

When it comes to JFIF images, the structural requirements, described in Section II-E, are very similar to the ones for a PNG image. Thus, the resulting tree follows the same organization as the one described previously. 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.

D. 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 III-B, 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) [24][10], which are described in Sections III-D1 and III-D2, respectively.

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 II-C. 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”.

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 to extensions that could cause some form of code execution. 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. 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”. 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 by downloading the image back from the web server and verifying if any vulnerability was triggered.

Executing all of these operations sequentially, as depicted in Figure 4, will result in a guided search for a combination of parameters that result in code execution through the upload of a malicious file.

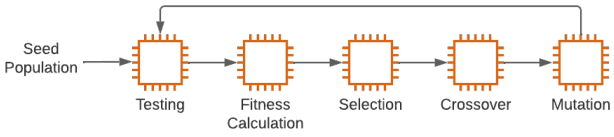

Fig. 4. Workflow of generic GA

2) FAexGA: The second variant of the genetic algorithm that was implemented is the Fuzzy-Based Age Extension of Genetic Algorithms (FAexGA) [24][10]. 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

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

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 [24][10], the likelihood of surpassing this obstacle increases. Based on the findings described in [10], 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 described in [24]. When two chromosomes are selected to crossover, the chances of it taking place are determined by their lifetimes. Similar to the work described in [24][10], these chances are higher with "middle-aged" chromosomes. 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. Its overall workflow is depicted in Figure 5.

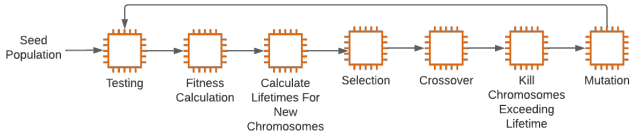

Fig. 5. Workflow of FAexGA

IV. 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 [25][26], a purposefully vulnerable web application known as Damn Vulnerable Web Application (DVWA) [27] and a set of production web applications with known vulnerabilities. These scenarios are listed in Table I.

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.

A. 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 [24][10] is discussed to determine whether or not its implementation improved upon the results with a generic GA.

#	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 [28] - CVE-2021-4080
13	CMS Made Simple [29] - CVE-2022-23906
14	WikiDocs [30] - CVE-2022-23375

TABLE I
List of test scenarios

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 II. 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 IV-A4, 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 [28] - CVE-2021-4080	No
13	CMS Made Simple [29] - CVE-2022-23906	Yes
14	WikiDocs [30] - CVE-2022-23375	Yes

TABLE II
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 [27]. 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.

The reasoning behind implementing a web crawler instead of using a model inference technique to model the web application, was 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.

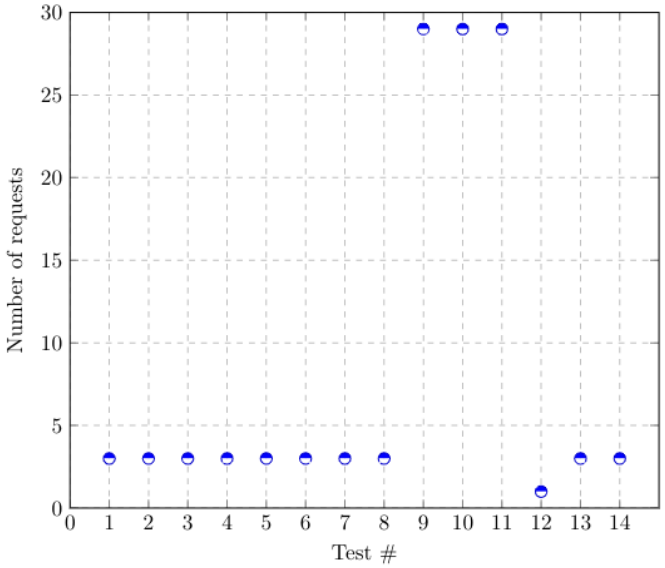

Fig. 6. Number of requests performed by the crawler

2) Fuzzer: The data depicted in Figures 7 and 8 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 7 and 8. 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 III-D, 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.

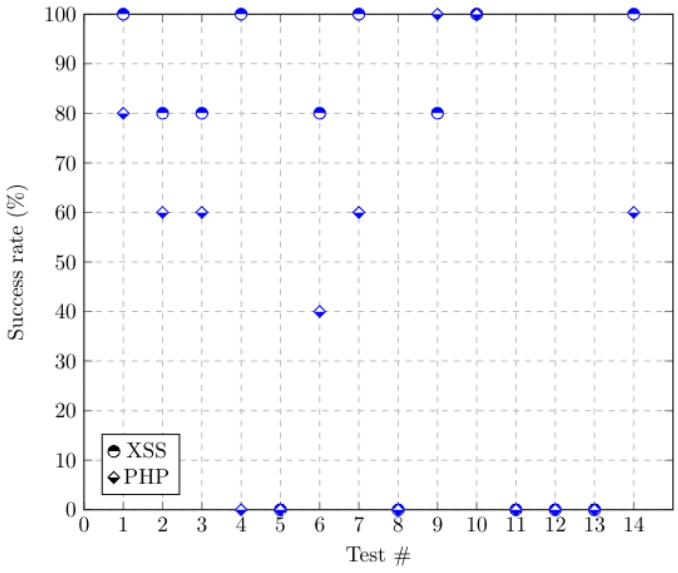

Fig. 7. Fuzzer success rate

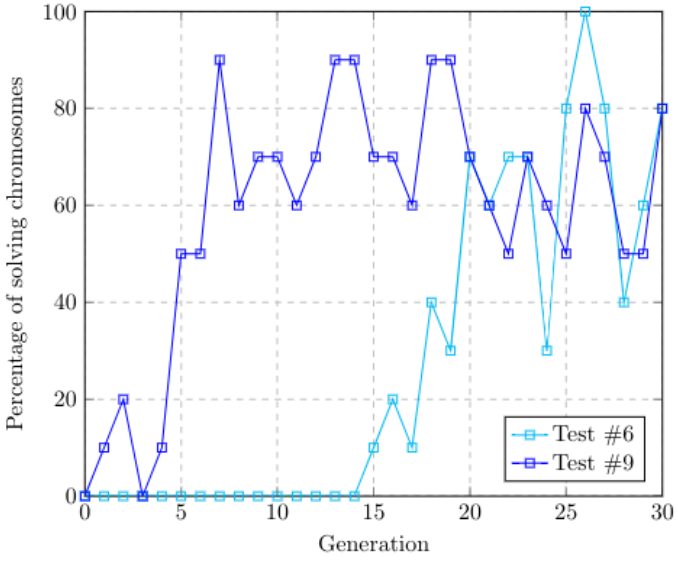

Fig. 8. Evolution of Generic Algorithm

The remaining tests (#5, #8, #11, #12 and #13) were not successful. As mentioned before, test #12 will be discussed in Section IV-A4. 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 [24][10] 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 9. 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.

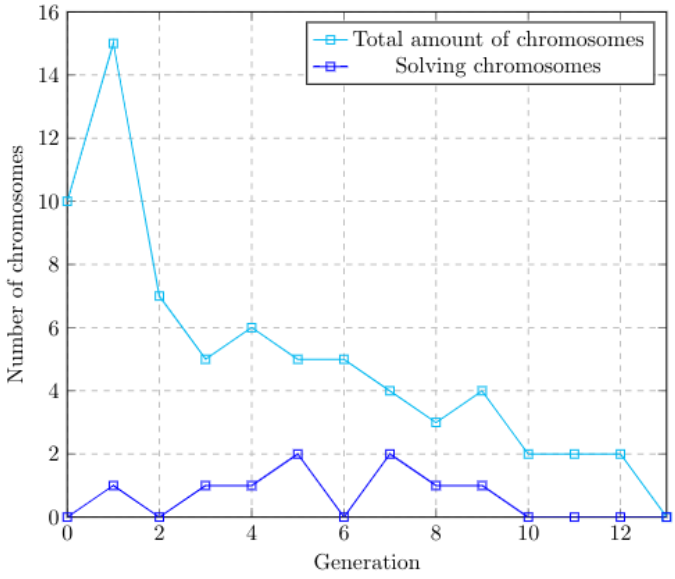

Fig. 9. Evolution of FAexGA in Test #2

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. 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 [24][10] 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.

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.

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 [31] 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 [32] with the file encoded in Base64 [33] 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 [34]. This methodology, however, would require a big sacrifice in the autonomy of the tool, as will be discussed in the following Section.

B. Comparison

The tool that was selected to compare the results with is the PortSwigger Upload Scanner [35]. This tool is an extension for the BurpSuite [34] 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 [34]. Furthermore, the tool is developed by PortSwigger [25], 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 IV-A4 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 [34] 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.

V. 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.

A. 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) [24][10] 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.

B. 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 [34] 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).

REFERENCES

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

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

[3] D. Whitley, "A genetic algorithm tutorial," *Statistics and Computing 1994 4:2*, vol. 4, pp. 65–85, 6 1994. [Online]. Available: https://link.springer.com/article/10.1007/BF00175354

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

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

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

[7] "Fuzzing — owasp foundation," [Accessed Nov 17th, 2021]. [Online]. Available: https://owasp.org/www-community/Fuzzing

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

[9] B. Miller, "Computer sciences department university of wisconsin-madison cs 736 bart miller fall 1988 project list," 1988.

[10] M. Last, S. Eyal, and A. 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)*, vol. 3875 LNCS, pp. 134–148, 2006. [Online]. Available: https://www.researchgate.net/publication/221471850_Effective_Black-Box_Testing_with_Genetic_Algorithms

[11] V. T. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," *ASE 2016 - Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 543–553, 8 2016. [Online]. Available: https://www.researchgate.net/publication/310819146_Model-based_whitebox_fuzzing_for_program_binaries

[12] L. K. Shar, T. N. B. Duong, L. Jiang, D. Lo, W. Minn, G. K. Y. Yeo, and E. Kim, "Smartfuzz: An automated smart fuzzing approach for testing smartthings apps," *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, vol. 2020-December, pp. 365–374, 12 2020.

[13] V. T. Pham, M. Bohme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, pp. 1980–1997, 11 2018. [Online]. Available: https://arxiv.org/abs/1811.09447v1

[14] "american fuzzy lop," [Accessed Dec 15th, 2021]. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[15] Y. Li, S. Ji, C. Lv, Y. Chen, J. Chen, Q. Gu, and C. Wu, "V-fuzz: Vulnerability-oriented evolutionary fuzzing," 1 2019. [Online]. Available: https://arxiv.org/abs/1901.01142v1

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

[17] E. Hamilton, "Jpeg file interchange format," 1992.

[18] "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.

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

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

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

[22] "What is nosql? nosql databases explained — mongodb," [Accessed Jul 21st, 2022]. [Online]. Available: https://www.mongodb.com/nosql-explained

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

[24] M. Last and S. Eyal, "A fuzzy-based lifetime extension of genetic algorithms," *Fuzzy Sets and Systems*, vol. 149, pp. 131–147, 1 2005. [Online]. Available: https://dlnext.acm.org/doi/abs/10.1016/j.fss.2004.07.011

[25] "Web application security, testing, scanning - portswigger," [Accessed Dec 26th, 2021]. [Online]. Available: https://portswigger.net/

[26] "File uploads — web security academy," [Accessed Sep 16th, 2022]. [Online]. Available: https://portswigger.net/web-security/file-upload

[27] "digininja/dvwa: Damn vulnerable web application (dvwa)," [Accessed Sep 16th, 2022]. [Online]. Available: https://github.com/digininja/DVWA

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

[29] "Open source content management system : : Cms made simple," [Accessed Sep 16th, 2022]. [Online]. Available: http://www.cmsmadesimple.org/

[30] "Wiki—docs," [Accessed Sep 16th, 2022]. [Online]. Available: https://www.wikidocs.it/

[31] "Selenium," [Accessed Dec 15th, 2021]. [Online]. Available: https://www.selenium.dev/

[32] "Json," [Accessed Sep 19th, 2022]. [Online]. Available: https://www.json.org/json-en.html

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

[34] "Burp suite - application security testing software - portswigger," [Accessed Dec 26th, 2021]. [Online]. Available: https://portswigger.net/burp

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