Web Browser-Based Interactive Crawling for Security Testing

TRABAJO DE FIN DE GRADO

José Luis Castañón Remy
Web Browser-Based Interactive Crawling for Security Testing

Submitted in partial fulfillment of the requirements for the degree of:

Ingeniería Informática

Author: José Luis Castañón Remy

Advisor
Manuel Carro Liñares

July 2018
Since the inception of the web, new web technologies have been emerging. The introduction of JavaScript, third revision of HTML, the release of Java and Flash technologies, to name a few. Web security bases were needed to be established and as a result, the figure of the security auditor was born. His/her main task has been to carry out all the testing tasks in search of vulnerabilities in web technologies. In order to facilitate this type of tasks, web application security scanners were born. Despite the great variety of web application security scanners, majority of them use crawlers. This thesis focuses on crawlers, in particular the crawlers within web application security scanners (scanners in short). Crawlers let scanners explore the content of a web application.

Crawlers contain many problems. The biggest problem is the underlying logic used by them to explore web applications. Current interaction is based on sending HTTP requests, parsing the response to find as many references (URLs) to access the contents of the web application and sending further HTTP requests to the identified URLs to explore more content. This logic is not enough to find all content stored in web applications. As we focus on solving this, we have found a new approach that can be used by crawlers to explore better the contents of a web application.

We have developed a crawler capable of interacting with a web application, mirroring human behavior. Meaning the crawler fills input fields and text areas, clicks buttons, submits filled forms, etc. We also provide an implementation of our approach, namely Incrawler, which is based on the Python Selenium WebDriver library.

In order to measure the performance of our crawler, we evaluate it against 4 different deliberately-vulnerable web applications (e.g., OWASP WebGoat) and compared its performance with the classic crawler within the OWASP ZAP pen-
etration testing tool. The results we achieved looks promising. For instance, our crawler is able to explore 80% of the content stored in WebGoat whereas the crawler with OWASP ZAP achieves only 14% coverage.
Resumen

Desde el inicio de la web, las nuevas tecnologías web han estado emergiendo, como JavaScript, la tercera revisión de HTML, el primer lanzamiento de Java y Flash, etc. Era necesario establecer unas bases de seguridad y como resultado nace la figura del auditor. Su principal cometido ha sido llevar a cabo el testing de tecnologías en búsqueda de vulnerabilidades. Con el fin de facilitar las tareas realizadas, nacen los escáneres de seguridad en aplicaciones web. A pesar de la gran variedad de herramientas existentes, todas ellas utilizan crawlers. Esta tesis se centra en los crawlers, en particular, en aquellos utilizados en escáneres de seguridad en aplicaciones web. Los crawlers permiten a los escáneres explorar el contenido de las aplicaciones web.

Los crawlers contienen una gran cantidad de problemas. El principal problema es la lógica que utilizan con el fin de explorar las aplicaciones web. La interacción actual se basa en lanzar peticiones HTTP, paesear la respuesta obtenida, con el fin de encontrar el mayor número de referencias (URLs) para acceder al contenido de la aplicación web. Las referencias encontradas serán tratadas de la misma manera con el fin de encontrar más referencias. Esta lógica no es suficiente para descubrir todo el contenido almacenado en las aplicaciones web. Como nos vamos a centrar en resolver este problema, hemos encontrado un enfoque diferente utilizado por los crawlers con el fin de explorar el contenido de aplicaciones web de manera más eficiente.

Hemos desarrollado un crawler capaz de interactuar con una aplicación web imitando el comportamiento humano. Por ello, el crawler rellena los campos ‘input’ y ‘text area’, hace click en los botones visibles, envía información a la aplicación, etc. Además, ofrecemos una implementación sobre el enfoque llamado Incrawler, el cuál se basa en la librería de Python Selenium WebDriver.

Para medir el rendimiento de nuestro crawler, lo evaluamos en 4 aplicaciones
web deliberadamente vulnerables (por ejemplo, OWASP WebGoat) y compara-
mos su rendimiento con el crawler clásico dentro de la herramienta de tests de
penetración OWASP ZAP. Los resultados que logramos parecen prometedores.
Por ejemplo, nuestro crawler puede explorar el 80% del contenido almacenado en
WebGoat, mientras que el crawler de OWASP ZAP logra solo una cobertura del
14%.
To my parents, for their endless love and support
## Contents

1 Introduction ........................................ 1
   1.1 Web Applications ................................ 1
   1.2 Crawling of Web applications ..................... 2
   1.3 Problems in Crawling Web Applications ........... 3
   1.4 Contributions ................................... 4
   1.5 Evaluation ..................................... 4
   1.6 Organization of the thesis ...................... 5

2 Fundamentals ......................................... 6
   2.1 Web Applications ................................ 6
      2.1.1 Structure .................................. 6
      2.1.2 Development ............................... 7
      2.1.3 HTML5 ...................................... 7
   2.2 Crawling & Web Application Security Testing ....... 8
      2.2.1 Security Testing Phases .................... 9
      2.2.2 Crawlers .................................. 10
      2.2.3 ZAP Spider ................................ 14
      2.2.4 Deliberately vulnerable Web Applications .... 17

3 Problems of crawlers ............................... 18
### 3.1 Problems with classic crawlers

- 3.1.1 Elements to explore ................................ 20
- 3.1.2 Values used to fill elements .......................... 21
- 3.1.3 Clicking ............................................. 21

### 3.2 Other problems ........................................... 21

### 4 Approach .................................................... 23

#### 4.1 InCrawler ................................................. 24

- 4.1.1 Crawling .............................................. 24
  - 4.1.1.1 Problem ........................................ 24
  - 4.1.1.2 Solution ......................................... 25
  - 4.1.1.3 Implementation ................................. 27
- 4.1.2 Interaction .......................................... 28
  - 4.1.2.1 Problem .......................................... 28
  - 4.1.2.2 Solution ......................................... 28
  - 4.1.2.3 Implementation ................................. 31
- 4.1.3 Clicking .............................................. 32
  - 4.1.3.1 Problem .......................................... 33
  - 4.1.3.2 Solution ......................................... 33
  - 4.1.3.3 Implementation ................................. 37

#### 4.2 Automatic scanner ....................................... 37

- 4.2.1 Problem .............................................. 37
- 4.2.2 Solution .............................................. 38
- 4.2.3 Implementation ..................................... 38

#### 4.3 Code ..................................................... 39

### 5 Evaluation .................................................. 40

#### 5.1 Test Subject ............................................. 41

#### 5.2 Automatic crawling ..................................... 41

- 5.2.1 Vulnerability detection ............................... 41
- 5.2.2 Coverage ............................................. 42
List of Figures

2.1 Common Crawling Process ........................................ 11
2.2 Depth first search Pre-order crawling algorithm ............ 12
2.3 Depth first search In-order crawling algorithm .......... 13
2.4 Depth first search Post-order crawling algorithm .......... 13
2.5 Large sites first crawling algorithm ........................ 14
2.6 Breadth first search crawling algorithm .................... 14
2.7 ZAP Crawling Process ........................................... 16

4.1 Example tree ..................................................... 27
4.2 Example tree step 1 .............................................. 36
4.3 Example tree step 2 .............................................. 36
4.4 Example tree final step .......................................... 36

6.1 Page rank crawling algorithm .................................. 49
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>HTML5 Usefull tags</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>ZAP Spidering example</td>
<td>18</td>
</tr>
<tr>
<td>5.1</td>
<td>Tree comparison</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>Vulnerability and time comparison</td>
<td>44</td>
</tr>
<tr>
<td>5.3</td>
<td>Browser parameter vulnerabilities</td>
<td>45</td>
</tr>
<tr>
<td>5.4</td>
<td>Tree comparison ZAP</td>
<td>46</td>
</tr>
<tr>
<td>5.5</td>
<td>Vulnerability and time comparison ZAP</td>
<td>46</td>
</tr>
</tbody>
</table>
1.1 Web Applications

Since the Internet Explorer’s birth in the middle nineties, both browsers and web technologies have witnessed significant number of changes.

In 1995 Netscape Communications presents JavaScript, enabling developers to improve the user interface with dynamic elements. JavaScript made the Internet faster and more productive as data did not need to reach the server to generate the whole web page. Currently, the majority of the content of the World Wide Web is produced by JavaScript, HTML and CSS. [1]

Flash was born in 1996 as a revolutionary innovation that made Web richer in content and more interactive. This vector animation player enabled programmers to enrich web pages with animations. Then Adobe Flash included streaming of audio and video in its animations.

Web application development has followed and evolved similarly to software systems. Technologies related, tend to grow rapidly and undergo frequent modifications, due to new technological and commercial opportunities, as well as feedback from the users.

In 1999 appears the concept of web application within the Java language. Later in 2005 Ajax was introduced. Ajax lets web applications to send and retrieve data from the server without interfering working processes on the web page, as downloading the page is not needed.

In 2014 appears HTML5, with its main purpose to represent content into WWW and arrange it into logical structures. HTML5 improves the existing HTML standard to support the brand-new type of multimedia in constant development. [2]

At the same time web technologies evolved, it was possible to see the birth
and evolution of the term web application security.

The term web security comprises technologies, processes and controls that are designed to protect websites, web applications and web services. Web application draws on the principles of application security but applies them specifically to Web systems. [3]

Web security market is expected to grow from USD 1.7 Billion to USD 3.2 Billion by 2022. The figure of the security auditor has gained popularity and supposes a big economic burden for small companies. As a result, different companies, Acunetix, IBM, PortSwigger Web Security, had found a more economical solution. Their main purpose was to develop software able to perform automatic vulnerability scanning in web applications. Those are commonly known as web application security scanners. [4]

1.2 Crawling of Web applications

Crawlers are common tools used in different fields of information technology. This thesis is focused on crawlers included in vulnerability analysis tools.

The aim of crawlers in web application security testing tools is to discover the content of the web application that is going to be tested. The content in web applications can be accessed through URLs, so crawlers will seek to discover as many URLs as possible within the target application.

To discover content, classic crawlers use a linear process. Its main task is to make requests to the application that is being scanned. Next, they will parse the response obtained from the application, which is the HTML code of the page to which the request was made. It will analyze the code, in search of labels that contain information relevant to the crawler, which is, references to other pages or some parts of the page. This information will be stored 'href' attributes inside '<a>' tags. Once a URL is found on the page that is being parsed, it is included in the list of URLs to be parsed if it has not already been parsed.

A common way to follow a certain order when crawling an application, is to store the URLs in a tree data structure. Commonly used algorithms are, large site first, breadth first search, and depth first search. Which will be later studied (see Section 2.2.2).

In order to study crawlers and the logic used to explore content in web applications and later compare results, the crawler of a known application within the world of cyber security is used, ZAP. ZAP is a tool developed by OWASP, worldwide not-for-profit charitable organization focused on improving the security of software. Burp Suite is a tool developed by PortSwigger, a software developing company focused on web security.
Chapter 1. Introduction

The ZAP’s crawler follows a common process when it comes to crawling. It will take three paths. Within these paths, the most important is HTML parsing. We observed, ZAP parses common tags that will allow it to find more links within the page, however ZAP is failing to find most of the URLs.

Finally, we introduce 5 different web applications that have been used to develop and test the solution proposed. WebGoat is an application within those five used to test our crawler. WebGoat has been developed by OWASP. OWASP used to use WebGoat to test analysis security testing tools like ZAP.

WebGoat contains vulnerabilities developed with a teaching purpose, we use it to scan vulnerabilities in the HTTP traffic generated by the crawler and observe the obtained coverage from the crawler. Coverage refers to the number of URLs found.

1.3 Problems in Crawling Web Applications

As we observed after the analysis and use of ZAP, classic crawlers have problems, which will be demonstrated through our experiments against the ZAP crawler. These problems prevent the crawler from identifying more content and therefore makes the subsequent security testing process less effective.

We have found the biggest problem within classic crawlers is the underline logic used to explore web applications. Many crawlers are not able to look behind forms, buttons, field inputs, etc. In this way all the content behind these elements or generated by these elements will not be found.

Problems related to logic behind crawler’s way to explore content in web applications can be divided into three main problems.

- **Elements to explore**: This problem is focus on what elements are needed to be explored. In specific, those elements that require user interaction to be explored. We are focusing on input and text area fields, buttons and submit elements.

- **Values used to fill elements**: This problem is focus on the way to explore input and text area fields. Exploring this content needs to be done by filling these fields with acceptable values.

- **Clicking**: This problem is focus on the way to explore buttons and submit elements. Exploring this content needs to be done by clicking these elements.

In addition, crawlers face many other challenges. We briefly describe them (see Section 3.2) also even though they are not the focus of this thesis.
1.4 Contributions

This thesis offers the following contributions:

- We develop a crawler capable to explore content in web applications, filling inputs, filling text areas, submitting information and clicking elements in the application through a browser based simulation.

- We analyze classic crawlers, more specifically ZAP’s crawler. We analyze our crawler in order to compare obtained results. We compare vulnerability detection, coverage and time for both of them.

- We develop a script to run ZAP scanner automatically along with several scripts to log into different testing web sites. The script generates an HTML report with all vulnerabilities found when scan is finished.

We will refer to the crawler developed for now on as *InCrawler*.

1.5 Evaluation

In order to carry out an evaluation on the proposed solution, we make a comparison between, crawling and scanning an application by the *InCrawler* and crawling and scanning by a user. This means, we scan traffic generated by the *InCrawler* and traffic generated by a user on WebGoat application. From these two experiments we analyze more specifically the detection of vulnerabilities, (the number of specific vulnerabilities each one have found), coverage (how many URLs within the application each one had found), and time taken within the whole process for each one.

We come to the conclusion that the proposed solution get solid results in front of a manual crawl, however it loses some content of the scanned application.

We also compare the proposed solution with a crawl and automatic scanning of WebGoat by ZAP. WebGoat contain 108 different URLs. Analyzing results obtained from both crawlers, our solution is finding 86 different URLs, which is 80% of the URLs stored in WebGoat. However, ZAP’s crawler is finding 15 URLs, which is 14% of the URLs found in WebGoat. Results obtained in this section by the proposed solution are considerably better than results obtained by ZAP.
1.6 Organization of the thesis

Finally, this thesis is organized as follows: The first chapter introduces the necessary foundations for a good understanding of the thesis, (see Chapter 2). The next chapter offers the most common problem found in classic crawlers that will be solved, in addition to a brief introduction of different problems, (see Chapter 3). The following chapter offers the solution to each problem encountered along with its development and implementation details, (see Chapter 4). The following chapter offers an evaluation of our crawler implementation, (see Chapter 5). Finally, we explain limitations of the solution, some future lines, (see Chapter 6) and a conclusion, (see Chapter 7).
In this chapter, different concepts to understand the thesis and its context are introduced. We first explain concepts related to web applications. Then we explain crawlers and how they are related to security tools. Finally, we introduce five different sites used to develop and test our solution.

2.1 Web Applications

In order to get an idea about web applications, it is necessary to introduce diverse concepts, so it is possible to identify the role of crawlers within web application and security testing.

2.1.1 Structure

The first concept is related to the structure of a web application.

Nowadays thanks to the large number of technologies related to web development, it is possible to find a huge variety of web applications. However, the base structure is commonly the same.

It is possible to reduce the structure of a web application in three layers: browser layer, server layer and data layer. [5]

Web browsers usually offer the first layer. They interpret HTML or XML languages in order to show information to the user based on the definition of the developer. In addition, scripting languages such as JavaScript are often used in order to enrich the user experience.

The server layer stores all the logic behind the application. It usually deals with data obtained by the user through the first layer as well as data stored in
the third layer.

Data layer is composed of a database, which is designed to satisfy all needs of the application.

There are many types of vulnerabilities related to web applications. For example, XSS vulnerabilities are usually found in the browser layer, logic flaws vulnerabilities are commonly found in the server layer and SQL injection vulnerabilities are related to the data layer. Commonly, the majority of the vulnerabilities found in web pages are found through the browser layer.

### 2.1.2 Development

Development of web applications is an important part to take into account. Any type of inconsistency in the developed code will generate a set of adverse conditions to the application, which will be later called vulnerabilities. This is the current origin for every vulnerability.

Nowadays it is easy to develop web content. Developing is commonly done through different technologies, like programming frameworks, that will help developers to code easily and faster, giving them some bases already coded or generating code automatically. Some of these technologies can be WordPress, Django or Ruby on Rails for instance.

Thanks to this type of frameworks, it is possible to find different programming standards, which allow the security auditor to understand more specifically the way the application works.

However, these frameworks can be turned against users. The fact that they are considerably used, makes them to be the focus when finding web vulnerabilities.

### 2.1.3 HTML5

HTML5 is the last revision made about the HTML language. HTML (HyperText Markup Language) is the most basic building block of the Web. It describes and defines the content of a web page along with its basic layout. Other technologies besides HTML are generally used to describe a web page’s appearance, CSS or JavaScript for instance. [6]

HTML5 allows establishing the bases of an open design to any open web developer. It is possible to define a series of resources on the technology itself that will define content in web applications. [7]

- **Semantics**: Semantics allows users to describe more precisely the content into the web application.
• Connectivity: Connectivity allows communication with the server.

• Non connection nor storage: It allows web pages to store data locally on
the client side to operate more efficiently.

• CSS3: CSS3 allows the user to establish the design of the web page.

As specified in the standard developed by W3C, it is possible to focus on a
set of specific tags that crawlers will later focus on. [8]

The following table contains some tags the solution proposed will later use to
perform efficiently.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Purpose</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;a&gt;</td>
<td>Defines a hyperlink</td>
<td>href, type, target, class, id, text</td>
</tr>
<tr>
<td>&lt;div&gt;</td>
<td>Defines a section in a document</td>
<td></td>
</tr>
<tr>
<td>&lt;input&gt;</td>
<td>Defines an input field</td>
<td>name, type, value</td>
</tr>
<tr>
<td>&lt;li&gt;</td>
<td>Defines a list item</td>
<td>value</td>
</tr>
<tr>
<td>&lt;textarea&gt;</td>
<td>Defines a text area</td>
<td>name</td>
</tr>
<tr>
<td>&lt;ul&gt;</td>
<td>Defines an unordered list</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: HTML5 Usefull tags

These tags allow the crawler proposed to identify exactly what elements to
interact with when crawling web applications. Efforts will be focused on <a>
tags more specifically. They are the most common structures containing rich
information for crawlers. In addition, <input> tags will be focused, so they are
an important part to be covered by the solution proposed. [9]

After researches accomplished, the table above satisfies the needs when it
comes to parsing HTML code behind each application.

2.2 Crawling & Web Application Security Testing

We will explain a simple process followed to detect vulnerabilities. In this process
it is possible to find out where the crawler is taking place.

In order to maintain certain security on web applications, the term penetration
test was coined around the year 2000. For now on referred as pentest, the term
is commonly used to describe an authorized, simulated attack over computer
systems. Its main purpose is to find both weaknesses and strengths to complete
a full risk assessment over the aiming system. A penetration test should help
determine whether a system is vulnerable, if defenses were enough and which of
them were defeated. [10]
The penetration testing execution standard consists of seven main sections. These covers everything related to a penetration test, from the initial communication and reasoning behind a pentest, through the intelligence gathering and threat modeling phases where testers are working behind the scenes in order to get a better understanding of the tested organization, through vulnerability research, exploitation and post exploitation, where the technical security expertise of the testers come to play and combine with the business understanding of the engagement, and finally to the reporting, which captures the entire process, in a manner that makes sense to the customer and provides the most value to it. [11] [12]

Following are the main sections defined by the standard as the bases for penetration testing execution:

### 2.2.1 Security Testing Phases

- **Pre-engagement**: This phase focuses on bringing certain aspects related to the pentest to the client.

- **Intelligence gathering**: The purpose of this document is to provide a standard designed specifically for the pentester performing reconnaissance against a target.

- **Threat modeling**: In this phase, a plausible attack methodology is usually defined on the objective or the different objectives.

- **Vulnerability analysis**: This process aims to find flaws within the target system. These flaws can range anywhere from host and service misconfiguration, or insecure application design. Although the process used to look for flaws varies and is highly dependent on the particular component being tested. This phase allows to relate the term crawler with the analysis of vulnerabilities in web applications. Even, it is possible to understand the term crawler as a sub phase in the analysis of web vulnerabilities. It allows to discover the content of the web application.

- **Exploitation**: The exploitation phase of a penetration test focuses solely on establishing access to a system or resource by bypassing security restrictions. If the prior phase, vulnerability analysis was performed properly, this phase should be well planned and a precision strike.

- **Post-exploitation**: The purpose of the Post-Exploitation phase is to determine the value of the machine compromised and to maintain control of the machine for later use. The value of the machine is determined by the sensitivity of the data stored on it and the machine’s usefulness in further compromising the network.
• Reporting: It is the final phase where results of the pentest are summarized.

Crawlers role will take place in the vulnerability analysis phase. It may also be found in the previous phase, threat modeling.

Before engaging in crawlers, it is important to identify which kind of security applications use crawlers to focus our solution.

Commonly known as scanners, vulnerability analysis applications have their main purpose over automatic scanning. It is possible to define two different type of scanners, proxy type or framework type.

Proxy type finds vulnerabilities analyzing HTTP traffic between the application and the scanner. In this case, human interaction is needed to generate traffic from the application.

Framework type scanners find vulnerabilities deploying specific scripts for each attack. This kind of scanners are also known as "point-and-click pentesting" scanners.

Nevertheless, scanners have limitations. Firstly, they provide no guarantee of soundness. Indeed, several reports have shown they fail to detect a significant number of vulnerabilities in tested applications. Reports usually warn about naive use of scanners.

Secondly, and as a reason to miss detection, scanners fail to crawl pages correctly. Crawling is a crucial phase for scanners. This will provide the scanner with URLs, those compounding the application, to scan and detect vulnerabilities in the whole application.

Finally, scans have high ratio false positive detection. To be more specific, the false positives are all those vulnerabilities detected by scanners that in this case are not real vulnerabilities in the application. This is a result of the high number of different technologies used to develop web applications, like complex HTML forms, extensive JavaScript or Flash code.

We will focus on proxy type scanners. We have used OWASP ZAP (Zed Attack Proxy) crawler to funnel our efforts and offer a better solution.

2.2.2 Crawlers

Crawlers, normally known as spider, are born to index web pages in search engines. Applied to the field of computer security, the term will not vary much. Spiders facilitate the work of the pentester remarkably. They allow to scan an application in order to list the content included in it. In this way it is possible to scan the application in search of vulnerabilities more efficiently. Generally, after the investigations carried out on this field, it is possible to define a common
methodology followed by crawlers (see Figure 2.1). [13]

![Diagram of the common crawling process]

The algorithm behind a crawler is quite simple. To start, it is needed a list of URLs, initial ones, to start crawling the web application. This URLs will be called seeds. Seeds are commonly introduced into the frontier, which is the list containing URLs still to be crawled. Then with each URL, firstly the algorithm will make a request with that URL as the target to get the HTML page represented by that URL. Then it will parse the HTML, looking for specific tags that store information to discover other URLs. Finally, new URLs found will be added to the frontier and it will start again until all URLs are finished.

The question to be answered now is: How do crawlers decide what URL to chose? When crawling, URLs are usually stored into tree data structures. This
is the most efficient way to maintain order for the crawler. To chose a URL, the crawler will need to walk the tree looking for the next URL to pick. This will be done using a tree iteration algorithm.

Most common algorithms are:

ALG.1 Depth first search:
The algorithm starts at the root node and explores as far as possible along each branch before backtracking. This algorithm contains three steps to traverse the tree. With each node, it can be done the following, no matter the order. [14]

L Left subtree:
This step focuses on traversing recursively the left subtree of the node.

R Right subtree:
This step focuses on traversing recursively the right subtree of the node.

N Node:
This step focuses processing the node itself.

As said previously, the order does not matter, so it is possible to find the following traverse methods.

NLR Pre-order:
This algorithm will firstly analyze the node, then it will traverse the left subtree calling recursively to pre-order function and finally it will traverse the right subtree calling recursively to pre-order function.

From the following example, we will get the following order: ’a - b - d - e - f - c - g - h’ (see figure 2.2).

![Diagram](image_url)

**Figure 2.2: Depth first search Pre-order crawling algorithm**

LNR In-order:
This algorithm will firstly traverse the left subtree calling recursively
to in-order function, then it will analyze the node and finally it will traverse the right subtree calling recursively to in-order function. From the following example, we will get the following order: 'd - b - e - f - a - c - g - h' (see figure 2.3).

![Figure 2.3: Depth first search In-order crawling algorithm](image)

**LRN Post-order:**

This algorithm will firstly traverse the left subtree calling recursively to post-order function, then it will traverse the right subtree calling recursively to post-order function and finally it will analyze the node. From the following example, we will get the following order: 'd - e - f - b - g - h - c - a' (see Figure 2.4).

![Figure 2.4: Depth first search Post-order crawling algorithm](image)

**ALG.2 Large sites first:**

This is a common algorithm used when crawling. It defines the order of each node depending on the number of pending links each node has. The more pending links it has the earlier it will be visited. This can also be seen as the node with more sons will be the first one to be visited. The number inside the node indicates the order given by the algorithm. (see figure 2.5).

In this case it is possible to get the following order: 'b - a - c - d - e - f - g - h'.
ALG.3 Breadth first search:

This is one of the most common algorithms used to walk trees. It starts at the tree root and explores all the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. [14]

In this case it is possible to get the following order: 'a - b - c - d - e - f - g - h'. In case the tree has numerous branches the algorithm will not perform effectively (see Figure 2.6).

To implement the solution designed, ALG.3, depth first search algorithm will be used while crawling. In specific, it will be used pre-order iteration. As we will see later, we will add nodes to the tree while we crawl. Using in order or post order algorithms do not fit with this kind of execution as analyzing the node is not the first thing it is going be done. So adding nodes after traversing, it does not matter whether they are added, they will not be traversed.

### 2.2.3 ZAP Spider

As previously said, ZAP spider will be used as a comparison to test and make a better crawler. We have considered introducing who has developed ZAP and how does it work.
OWASP is an open community dedicated to enabling organizations to conceive, develop, acquire, operate, and maintain applications that can be trusted. All the OWASP tools, documents, forums, and chapters are free and open to anyone interested in improving application security. OWASP Foundation, came online on December 1st, 2001 and it was established as a not-for-profit charitable organization in the United States on April 21st, 2004.

OWASP Zed Attack Proxy (ZAP) is a free security tool born to carry out security audits on web applications. ZAP is mainly a proxy server, letting users to manipulate all the traffic that passes through it, including HTTPS traffic. Thanks to different modules, it will analyze the traffic, giving information related to security flaws. ZAP has an existing API to programmatically interact with it, through different languages. In this case we will use python to interact with ZAP.

ZAP has different functionalities which allow users to perform different activities depending on their objective. We will be focusing on the spider, active scanner and passive scanner. After analyzing the source code of ZAP, it is possible to make a small outline of how its crawler algorithm works (see Figure 2.7). [15] [16]

As an initial seed, ZAP uses 4 common resources:

- robots.txt
- sitemap.xml
- SVN metadata
- GIT metadata

However, ZAP does not use any previously introduced crawler algorithm, since it does not use trees as the main structure on the algorithm. They use a map, based on the frontier list previously explained, to represent URLs that have been traversed and those that remain to be traversed.

Subsequently, the most common problems related to crawlers will be later introduced, however, it should be noted that both ZAP and other tools of the same category do not fulfill the function for which they have been developed.

Many of them are designed to crawl and scan any web application automatically. However, most of the time, human factor is necessary for the performance of these applications to be optimal.

In addition to the crawler, ZAP offers two tools used specifically to detect vulnerabilities, analyzing the traffic obtained by the crawler. Both tools are passive scanner and active scanner.
Chapter 2. Fundamentals

Figure 2.7: ZAP Crawling Process
Both scanners analyze the application to search the same vulnerabilities. However, the passive scanner performs the analysis in the background, without interrupting the user’s activity in the application. The policy used by the passive scanner is not definitely intrusive. It will try to keep a low profile in terms of noise generated, which means keeping the number of requests sent to the application, low. [17]

On the other hand, the active scanner allows the user to configure the policy the application is going to be scanned with. The scanner runs in the foreground so that no other action interrupts while it is active. [18]

2.2.4 Deliberately vulnerable Web Applications

Then a series of applications that contain known vulnerabilities, which have been developed with a training / teaching purpose, are briefly introduced. We use them in order to test the proposed solution in the subsequent stages of development.

- Damn Vulnerable Web App
- WebGoat
- Bodgeit
- WackoPicko

Both DVWA and WebGoat are applications developed with a teaching purpose. They contain known vulnerabilities, separated into different sections, to show how vulnerabilities work, and the way it is possible to exploit them. More specifically, DVWA contains an execution mode in which it is impossible to exploit any vulnerability, in order to show web developers how to avoid web vulnerabilities. [19] [20]

Hackazon, Bodgeit and WackoPicko are web applications that mimic the way different real web applications work. They also contain known vulnerabilities, in order to more realistically test known vulnerabilities. They have been developed with a training purpose. [21] [22] [23]

The development based its tests on the DVWA application. Once we got a stable algorithm, we went to test the application on WebGoat.

Both applications allowed us to define the path of the algorithm. To solve all difficulties they presented, we added a feature to the crawler in order to overcome the difficulty.

Finally, we used Hackazon, Bodgeit and WackoPicko in order to test the crawler over real applications. These applications allowed us to modify the features introduced previously, in order to solve real problems.
As we briefly discussed in Chapter 2, the crawlers used within web application security scanners have problems.

In order to illustrate some common problems, we use the following example.

<table>
<thead>
<tr>
<th>URLs</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total found</td>
<td>36</td>
</tr>
<tr>
<td>Seed</td>
<td>26</td>
</tr>
<tr>
<td>Real found</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 3.1: ZAP Spidering example

In Table 3.1, we illustrate the results of crawling WeboGoat (a prominent, deliberately-vulnerable web application) with ZAP’s crawler. The terminologies used are as follows. ‘Total found URLs’ are all the URLs that the ZAP’s crawler returns as a result. ‘Seed URLs’ are all those URLs, included in total found, that ZAP uses as initial seed to launch the crawler. ‘Real found URLs’ are all those URLs, included in total found, that ZAP has actually found when crawling.

Classic crawlers need initial sowing of URLs (called seed URLs) in order to reach certain results. If we analyze the results shown in the table, it is possible to observe that 26 seed URLs have been used to find just 10 URLs. This implies the number of URLs found is less than the number of URLs used as the initial sowing, which means initial sowing may not be efficient.

Additionally, we will later see (in Subsection 5.3.2) that WebGoat has at least 108 URLs found when crawling manually. This means, ZAP is losing up to 67% of the content of the application.

This is an indication that more than half of the information stored in the application has been lost. This is a problem because in automatic security scanners, the subsequent phase after crawling is automatic security testing. In this
phase, the scanner tests the web application under test by probing it through the
URLs identified during the crawling. This means, if the crawler did not identify
certain URLs, these URLs will not be considered during the probing and the
vulnerabilities associated to these URLs are missed.

Therefore, it is possible to reach the conclusion that ZAP crawler is obviously
failing in its purpose. In the following section we discuss a series of observable
problems in classic crawlers.

3.1 Problems with classic crawlers

The logic used by crawlers to explore content in web applications is the biggest
problem.

A large part of the content generated nowadays in web pages is done dynam-
ically. This implies that the user must interact with the website so that it can
load the desired content. In this way, most of the crawlers used by vulnerability
detection tools are not able to explore all the existing content.

To illustrate this, imagine trying to crawl a web application. This web appli-
cation has a login panel with some credentials for users and guests. The panel’s
logic is correct, that means you are not able to get into the application unless
credentials have been submitted correctly. The application content will not be
accessible until we log into it. If we launch the crawler from the login panel, it
will only parse the HTML from it. The number of URLs found by the crawler
at this point will be definitely low. Even if it is able to discover the URL of the
main page of the application, it will not be able to travel to that URL as login
credentials has not been introduced. At this point, user interaction is needed to
fill the login panel and let the crawler start from the main page.

As we have done earlier (see Table 3.1), imagine we throw the crawler from the
main page of the application. No login credentials are needed as we are already
inside. Generally, web pages have content to interact with the user. Common
content will be filling inputs, clicking buttons, submit forms, etc. Crawlers logic
is not design to explore this kind of content. This leads us to another problem:
What happens with content generated dynamically? Probably, it will be lost.
Crawlers within web application security scanners are focused on discovering
content stored in web applications. Later, web application security scanners will
request and scan content found by crawlers in order to detect vulnerabilities.

To illustrate this, imagine we throw the crawler over a contact form. The
crawler will not discover inputs as they are not design to explore this content. At
this point, user interaction is truly needed, in order to fill the form, letting ZAP
capture the request.
We observe crawlers are not design to interact with an application in the same way users would interact.

Crawlers current interaction with a web application is simple. They send HTTP requests and receive responses that they will later parse in order to get as many references as possible (URLs).

As it will be seen later in the thesis (see Chapter 4), we propose a new approach for crawlers to explore content in web applications. In order to find this new approach, we must answer the following question: Why crawlers are not able to interact with web applications like users interact?

Mainly, the answer is, this type of software is not designed to perform interaction with web applications further than sending requests and receiving responses to parse. Crawlers are usually included in web application security scanners in order to let them explore content in web applications.

We propose a new approach for crawlers to explore content within web applications. The approach implies to interact with elements that need human interaction within web applications by exploring them. This approach avoids human interaction with the application so it is possible to have a full automatic crawling and testing over web applications.

To achieve this solution, we still have some challenges to face. First of all, what elements do the crawler need to explore? Secondly, exploring fillable fields, requires values to be introduced by the crawler. What values do elements need to be filled with? Finally, how does the crawler explore elements that do not need to be filled?

### 3.1.1 Elements to explore

It is not possible to explore all elements found in web pages. We have decided to explore common elements used to introduce data in the application under test. In specific, we are focusing on fill input fields and text area fields. We also explore submit elements and buttons. This is a set of elements that require human interaction.

Once we have chosen what elements to explore, how do we find them? To find elements within web applications, we need to check where are elements stored. We need to parse the HTML code behind the web page itself to find elements to click. Finding elements needs to be done through the tags that compound the element itself. The solution proposed should also answer, what tags do we need to look for?
3.1.2 Values used to fill elements

Once we have clear what values to explore on the web application under test, we need to establish what values are needed to fill elements that are required to be filled, in this case input and text area fields. These fields need to be filled so exploration has sense. It is not possible to find content behind these elements unless they are submitted to the application. Also, filling should be done properly, as filling wrong values prevent the exploration from being carried out.

3.1.3 Clicking

Once we have a way to explore fillable elements, we need to answer how is the crawler exploring non fillable elements? Solving this problem makes the difference between classic crawlers and the solution proposed. The way to solve this problem is clicking not fillable elements, this means clicking buttons and submitting.

Currently, clicking is not achieved by crawlers. To click, it is necessary to find what tag (HTML) represents the element to be clicked. Once the tag is selected, clicking should not be a problem. However, what happens when elements are not able to be clicked?

We observe there are two reasons. Each reason supposes a problem to be solved.

Web pages can have partial visible elements. This means they have elements that will become visible as soon as interaction with other element is done. For example, choices behind drop down lists are not visible until the list is expanded. This means it is not possible to make a choice until the list is expanded. The element is there but it is not visible.

Another problem resides on changing elements. It is common to see pages updating their sites, web market stores for instance. Reloading the page would make an element found previously, disappear. This means the element is being loaded in a different part of the application or it has simply been removed from the page. So clicking the selected element will fail.

Next are introduced common problems found while analyzing classic crawlers. These problems are not focused in this thesis.

3.2 Other problems

There are more problems within classic crawlers. This is a rough list about the most common problems that will not be approached in this thesis.
• Response time:
  Due to map and network configuration implemented, it is possible for many
  servers to have a slow response to requests made. This implies that if we
  make a high number of requests over the application when crawling, it is
  possible to get time out answers, that will prevent the application to be
  crawled.

• Anti scrapping tools:
  These tools are used to prevent pages to be crawled. Common techniques
  are the use of captchas or software to generate infinity loops. A clear ex-
  ample can be a calendar plugin with hyper links in each day. If the plugin
  generates infinity days, the crawler will find infinity links to crawl.

• Overload traffic:
  Crawlers will usually make a request for each URL. Getting a big tree from
  the page being crawled will make requests increase considerably.
  Generally it is considerable to avoid this type of problems since they suppose
  what in cybersecurity is known as DoS.
  DoS stands for Denial of Service Attack. Denial of service attacks seek to
  make a machine or network resource unavailable by temporarily or indefi-
  nitely disrupting services of a host connected to the Internet. [24]
  Denial of service is typically accomplished by flooding the targeted machine
  or resource with superfluous requests in an attempt to overload systems and
  prevent some or all legitimate requests from being fulfilled.

• Non uniform structures:
  This is a very common problem faced by crawlers. Nowadays it is impossible
  to find two pages whose design is the same. This supposes that the HTML
  code found after each page is very varied and therefore very complicated to
  maintain an order of what to parse or not.

• Repeated paths:
  As seen previously, crawlers will parse HTML code to find other links into
  applications. Commonly, pages inside applications will have links and refer-
  ences to other pages inside the same application, called anchors, for example
  if there is a static button to go to the home page, when parsing HTML this
  reference will be parsed each time the crawler travels to a different page and
  parses the button again. Bad parsing repeated common or repeated URLs
  can lead the algorithm behind the crawler to never end.
In order to solve problems related to logic behind crawler’s way to explore content in web applications, we have designed a crawler algorithm with a different logic in order to explore content in web applications.

As we have seen earlier (see Chapter 3) crawlers fail to explore content of web applications. In order to let web application security scanners, in specific ZAP, analyze this content, we have used the following process.

1. Login into the application
2. Launch ZAP
3. Navigate through application
4. Launch active scanner
5. Check results

Within those steps, step 3 is actually the most important one. It will let ZAP explore the content in the application. This step is done by a user as logic behind crawlers to explore content in web applications is failing. Currently the InCrawler replaces this step.

We propose a crawling solution to click and fill all input values, text area and submit elements to explore the whole content of the web application under test. ZAP captures traffic generated in this interaction. It later scans this traffic to find and analyze vulnerabilities.

To have a total automatic testing, we have designed a script which deploys ZAP’s scanner. We have also designed different scripts in order to log into web applications under test. Whenever they finish logging, the InCrawler is launched so it can interact with the web application under test.
The InCrawler is facing three challenges. These need to be solved in order to redesign logic to explore content in web applications.

4.1 InCrawler

4.1.1 Crawling

First challenge to face is crawling the application. Crawling is the main part of the solution proposed. It is needed to design a stable algorithm to maintain order while crawling.

Previously diving into problems, we need to introduce driver concept. The driver is the main controller of the browser. Whatever is needed to do with the browser is done through the driver. Driver role is related to URL management. This will lead the driver to click or get to the URL stored in each node. Differences between click and get will be later explain. This concept allows identifying how the simulation is going to be done.

Still, it is needed to define an algorithm to maintain an order while we are crawling. This will be given by the data structure used.

4.1.1.1 Problem

When facing the crawler, it is possible to find a series of problems that will define the algorithm itself.

The first problem that needs to be faced is the application structure itself. It is needed a data structure stable enough to not lose control while crawling. This will avoid loosing control over the driver too. This problem is solved using tree data structures.

The second problem is given by the way we run the tree and its integration with the driver. As the tree can be complemented at the same time it is traversed, it is possible to find the problem that the driver does not know how to jump to the neighbor node, or to another node not linked to him. To solve this problem, it is necessary to backtrack the tree obtained at that point, with the driver. This problem is solved using depth first search algorithm pre-order iteration.

With the purpose to avoid this kind of problems, we offered the following solution where all previous problems are solved.
4.1.1.2 Solution

As said before, the first problem is solved using tree data structures. The tree stores URLs found while crawling. Each node stores what we call DATA. DATA structures stores the name of the node, which is the URL itself, text stored in the HTML tag, target attribute stored in the HTML tag, class attribute stored in the HTML tag, id attribute stored in the HTML tag and a reference to the parent.

The following algorithm explains the crawling solution proposed.

```
TREE = Set initial tree root node starting URL;
DATA = Call GETELEMS, get elements from current driver page;
for all a tags in DATA do
    if href in a tag not in TREE then
        add href to tree;
    end
end
for node in TREE DepthFirstSearch Iteration do
    if node is root node then
        jump over;
    end
    if node href contains 'logout' then
        jump over;
    end
    if node href does not contain 'localhost' then
        jump over;
    end
    GET driver to parent node;
    Call CLICK(Node) driver to son node;
    DATA = Call GETELEMS, get elements from current driver page;
    for a tags in DATA do
        if href not in TREE then
            add href to TREE;
        end
    end
    Call FILLINPUTS, fill input tags in current page;
    Call FILLTEXTAREAS, fill textarea tags in current page;
    Call SUBMIT, click submit tags in current page;
    GET driver to parent node;
end
```

Algorithm 1: Crawler algorithm
First of all, the initial tree needs to be set. Root node will store the URL the driver is on when login scripts are done. From there we call function ’getelems’ which will give us a list of DATA structures that will compose the initial tree.

With the initial tree formed, the algorithm will start traversing the tree following pre-order iteration, previously introduced (see Subsection 2.2.2).

While traversing the tree, mainly, with each node is done the following:

1. Get the driver to the URL stored in the parent node
2. Click the driver to the URL stored in the current node
3. Call function ’getelems’
4. Update the tree
5. Fill all values and submit elements in the current driver page
6. Get the driver back to the URL stored in the parent node

This steps can be controversial, however, all of them have a reason. Getting to the parent of the node ensures the node we are going to click to is the real son. As we are using tree data structures the driver will only be able to click to an element if there is a real link to that element, which means clicking will only be done from the father to the son.

Clicking the driver from the father is actually the current purpose to this crawler implementation.

Call function ’getelems’ will find the URLs referred in the current node. This new URLs found will be added to the tree, if they are not already in the tree, in the following step. ’Getelems’ function will be later explain (see Subsubsection 4.1.2.2).

Filling all values is an add to the current purpose to this crawler implementation. This is where the InCrawler interacts with the current page. This will be later explained too (see Subsubsection 4.1.2.2).

Going back to the parent at the end of the loop is an essential step. It avoids problems while clicking to a neighbor node.

As an example to clarify tree walking problems along with driver movement, we can use the following tree.
Chapter 4. Approach

According to the algorithm, to iterate the tree, node order given by the algorithm is: 'a - b - d - i - e - f - c - g - h'.

The driver will follow the same order, so on, it starts in node 'a', clicking to 'b', 'd' and 'i'. However, what will happen when the driver tries to reach 'e' node? It will surely fail as 'e' node is not related to 'i' node. To avoid this problem it is necessary to make the driver going back to the parent, so we ensure each node is possible to be clicked.

For the following node order 'a - b - d - i - e - f - c - g - h' the driver will do: 'a - b - a - b - d - b - d - i - d - b - e - b - f - b - a - c - a - c - g - c - h'.

In this way it is possible to avoid problems while traversing the tree.

4.1.1.3 Implementation

To implement this solution, we use the following libraries.

Anytree is used to endow development environment with tree data structures. It allows the algorithm to redefine the nodes of the tree and walk them in different ways. We have used this library for its simplicity and accessibility to both documentation and implementation. With Anytree we can manage the tree and nodes behind the crawler algorithm.

Selenium WebDriver Python library is the most important library in the whole project. Selenium is a portable software-testing framework for web applications. Selenium WebDriver has been chosen for its integration with Python. WebDriver will interact with the application following the crawling algorithm previously introduced.

This algorithm is the only one that manages the tree and the nodes composing the tree. It is the only one calling Anytree library. Calls to specific parameters of the node are done, the name, ‘node.name’ and the parent, ‘node.parent’, with
the purpose to know the value from both elements. In addition, calls to the specific constructor of the node, DATANODE, are done. Different parameters, specifically 5 are kept into the node along with the parent node to be attached, the father node.

Apart from the rest of the functions being called, it uses a specific function from Selenium library. As it is specified in the crawling algorithm, 2 calls to function GET are done. Those calls are done through the driver, as ’driver.get’. The entry parameter is always the father’s node name, ’node.parent.name’. This allows the driver to travel directly to the URL stored in the father node.

4.1.2 Interaction

To continue, after checking how it is possible to crawl the application, we will face interaction challenge. Interaction with the application supposes exploring fill inputs, text areas and submit information to the application. This is done to access traffic generated by these elements to later analyze for vulnerabilities.

4.1.2.1 Problem

First problem resides on element’s exploration. In this case it is possible to explore three different elements that commonly compose user input within applications: inputs, text areas and submits. These are usually identified with HTML tags: <input> and <txtareas>.

Second problem is focused on how is the InCrawler finding elements being explored. That means how are we going to identify input fields, text area fields and submit elements.

The third problem focuses on identification over values to introduce in each field, in case a specific value is needed to be filled. This means if an input element requires an email to be filled, the value introduce should be an email.

The fourth problem is related to invisibility. It is possible to find elements with non visible attributes, in its HTML tag. This will make the element to be impossible to click, as the driver will not find it.

4.1.2.2 Solution

To continue, it is offered the following solution to each problem. This section has a list of algorithms used to avoid this problems.
Second problem is solved through the following algorithm. This is a base algorithm behind getelems function.

**Result:** DATA List 
Tag list = find all a tags; 
for tag in Tag list do
  DATA href = Get href from tag; 
  DATA target = Get target from tag; 
  DATA class = Get class from tag; 
  DATA text = Get text from tag; 
  DATA id = Get id from tag;
  Append DATA to list;
end

**Algorithm 2:** Get elements algorithm

The algorithm is getting attributes explained above from any <a> tag found in the page the driver is on. This is made to find specific elements to click. All <a> tags are stored in a list which will be later added to the crawler.

The following algorithms offer a solution to the first and the third problem. They are design to fill input and text areas within pages while walking the tree. Also, submitting the information filled by the crawler is explained.

Still, before diving into the algorithms, dictionary should be introduced.

In order to detect what kind of value can be used to fill inputs, text areas or submits, we will use a dictionary to store common values within input tags found while testing the algorithms.

The dictionary is as simple as a `.csv` file. This file has for columns where data is stored. First column stores the name of the tag itself. Second column stores the name attribute value. Third column stores the type attribute value and finally the fourth column stores the value the crawler uses to fill the element we are trying to fill.

The dictionary is design to grow as much as possible, so structure is so simple.

The crawler interacts with the dictionary through three different functions. Those are quite simple. They read the dictionary, and depending on the number of parameters, they give the specific combination asked for, if stored in the dictionary. If we give three parameters we will get the specific value linked to that three element combination. Giving two parameters will return any combination were only two are needed. In this case the third value is not needed so it will be represented as NN. This is the case of password filling. Only tag name and type attribute values are needed. In case there is no combination for that specific element, we are using a default value, which is returned by giving one parameter to the function.
As well as defining what values to fill, the dictionary is used to avoid some elements that cannot be clicked. This is how it is possible to solve fourth problem. It will be stored different attribute values found to be common within tags when referring to invisible elements. This solution is implemented in fill inputs algorithm (see Algorithm 3).

With the dictionary concept clear, element interaction algorithms can be explained.

This is the algorithm design to fill inputs.

Tag list = find all input tags;
for tag in Tag list do
  type = tag type value;
  name = tag name value;
  value = Look into the dictionary (input, type, name);
  if value == hidden then
    CONTINUE;
  end
  if value not found then
    value = Look into the dictionary (input, type);
    if value == hidden then
      CONTINUE;
    end
    if value not found then
      CONTINUE;
    end
  end
  Fill element with value;
end

Algorithm 3: Fill inputs algorithm

This algorithm is designed to solve all the problems introduced in this section.

The algorithm initially generates a list of input tags. This list represents all the fillable elements identified through the input tag.

With each element, first it checks if it is an invisible element, which will be skipped. Then it will make three calls to the three functions responsible for finding the specific values to be entered each element.

Once we have the value, it fills the element and moves on to the next one. This algorithm is designed to fill <textarea> tags within pages while traversing the tree.
Tag list = find all textarea tags;

for tag in Tag list do
  value = Look into the dictionary (textarea);
  Fill element with value;
end

Algorithm 4: Fill text areas algorithm

In this case, the algorithm is much shorter, since identifying the value to enter in <textarea> tags is very simple. Only one call to the dictionary is necessary, since knowing the name of the tag is enough.

This is the algorithm behind the function designed to click submit tags within pages while walking the tree.

Tag list = find all input tags;

for tag in Tag list do
  type = tag type value;
  value = Look into the dictionary (input, type);
  if value == submit then
    Submit;
    BREAK;
  end
end

Algorithm 5: Submit algorithm

In this case, the algorithm is also short. It is necessary to look for values in combination of two, since identifying these elements with the name of the tag input and the type value equal to submit is enough.

4.1.2.3 Implementation

Get elements algorithm calls Selenium library. Its only function is to make a list of elements to click, so calls are done to find elements by tag name.

Find elements by tag name receives the name of the tag as parameter and its purpose is to find all tags with this name, found in the web page. The implemented solution uses the parameter 'a' in this specific case.

With each specific tag, it calls get attribute function four times. Get attribute function returns the value of the attributes according to the parameter passed to this function. The solution uses 'href', 'target', 'class' and 'id' as parameters. In addition, it obtains the tag’s text accessed through tag.text.

Fill inputs algorithm calls Selenium library, in addition it calls the auxiliary functions to access the dictionary.

The algorithm calls 'find elements by tag name' function using the value 'in-
Chapter 4. Approach

put’ as a parameter. This will let the algorithm identify all input tags existing in the web page.

Later, values of type and name will be got from the tag, through get attribute function. Both values in addition to tags name, are used to call auxiliary functions to find the specific value stored in the dictionary. This value returned by the dictionary is used to fill the element.

Finally it adds the value to the element in the browser through ‘send keys function’ from Selenium library too, using the value from the dictionary as the parameter.

Fill text areas algorithm calls Selenium library. It will call ‘find elements by tag name’ function with ‘textarea’ as a parameter. This will let the algorithm find all <textarea> tags existing in the web page.

It will call ‘send keys function’ too, with its main purpose to fill text area fields. Therefore it must look into the dictionary for the specific value to fill this fields with.

Submit algorithm only calls Selenium library. It calls ‘find elements by tag name’ function with ‘input’ as a parameter. With each tag, the algorithm checks type value, obtained after calling get attribute function using ‘type’ as a parameter, is equal to the value ‘submit’. It then calls submit function which mimics a click on the element previously selected, to send data stored in different values from the same page.

To avoid sending empty elements, the crawling algorithm will call this function the last one.

To access the dictionary we have designed 3 functions. These functions only read the file ‘dictionary.csv’. To read the file they call the same function, ‘reader’, using the name of the file as a parameter and ‘,’ as delimiter. This function is stored in csv library which is stored within common python libraries.

4.1.3 Clicking

To continue, it is introduced the last challenge the InCrawler faces. This challenge is clicking elements.

Before diving into the clicking algorithm, the difference between clicking and getting are explained as well as XPATH concept.

Get introduces the URL on the navigation bar of the browser. Click, function developed for the algorithm, interacts with the element represented in the DATA structure stored in the node given as entry parameter.

Clicking allows the algorithm to treat the application the way needed. Not
clicking will make ZAP loose all data and URLs behind tags the solution is able to click. Getting to the URL instead of click, would be an expensive copy of a classic crawler. This answers why clicking is needed.

Another concept to introduce is XPATH. XPATH is a syntax to define and navigate through expressions in XML and HTML documents. As nodes contain the specific data to find a specific element, we will use XPATH to identify the specific element we are going to click on.

### 4.1.3.1 Problem

The main problem found is related to element visibility. This problem can be divided in two problems.

First problem is related to hidden elements. These elements are all those elements that can be found after performing some type of interaction with another element of the page. A simple example is to click elements behind collapsibles. It is necessary to open the collapsible to click the element.

Second problem resides on the elements themselves. In case the application contains elements that are randomly redefined or re-positioned in different parts of the application, it will be a big challenge for the driver to click those elements since the position in the element tree will have changed and it will not be able to click the element.

### 4.1.3.2 Solution

Clicking algorithm is the most important within the whole algorithm. It carries out the main purpose of the crawler.

This sub-algorithm obtains as an entry parameter a node given. It returns the driver in the page represented in the node.

In this case we use four ways to try to click on the specific element represented in the node.
Data: Node
while number of attempts less than 4 do
  if attempt == 0 then
    Find element by XPATH(Node text);
    Click Driver to node;
    BREAK;
  end
  if attempt == 1 then
    Find element by XPATH(Node class);
    Click Driver to node;
    BREAK;
  end
  if attempt == 2 then
    Find element by XPATH(Node href);
    Click Driver to node;
    BREAK;
  end
  if attempt == 3 then
    if backtrack == true then
      Add parent to XPATH;
      Click driver to new element;
      BREAK;
    end
    Find element by XPATH(Node id);
    Click Driver to node;
    BREAK;
  end
  if Element not Found then
    attempt ++;
    CONTINUE;
  end
  if Element not Visible then
    backtrack = True;
    CONTINUE;
  end
end
GET Driver to node;

Algorithm 6: Clicking algorithm

Before the explanation. Click is not the same as CLICK. Click refers to the current click method from Selenium library.

The algorithm tries to click in four different ways. The order within the attempts is nothing to care about.
Whenever attempt 1 is tried, XPATH is, //a[contains(text(),'Node text')]. This will find an '<a>' tag containing text equals to text data stored in the node. If this attempt fails, attempt 2 XPATH is, //a[@class='Node class']]. This will find an '<a>' tag with a class attribute equals to class data stored in the node. If this attempt fails, attempt 3 XPATH is, //a[@href='Node name']]. This will find an '<a>' tag with an href attribute equals to the name of the node which, as previously said, stores the URL itself. If this attempt fails, attempt 4 XPATH is, //*[@id='Node id']]. This will find an '<a>' tag with an id attribute equals to id data stored in the node.

It is possible to fail in two different ways. One is that the element is not found, which means the actual element the driver is looking for is not in the current page the driver is located. This increases attempt counter, as a consequence, clicking algorithm tries another way to click.

Whenever we click an element if it is not visible, the driver throws an exception.

While testing the solution, the algorithm was failing to click elements behind collapsible items. Collapsible items are commonly know as drop down lists or spans. The problem is simple, imagine there are three links behind a collapsible form. Whenever the driver tries to click, element not visible exception is thrown. To avoid this, the easiest way is to click the parent to make the link visible. To do this the algorithm will add /parent::* to the XPATH. This will find automatically the HTML tag parent of the tag we were looking for. If the tag under the collapsible item trying to be clicked is found using //*[@id='Node id'] and we get element not visible exception, we will then look for //*[@id='Node id']/*/parent::*.

Out from the algorithm, there are many different exceptions that will break the program. To avoid stopping execution, if an element is not clickable, the only viable option to continue crawling is to get the driver to the element itself.

Getting to the element is the solution proposed to solve problem number two. However, we dive more into this problem.

Imagine the following initial tree.
Chapter 4. Approach

While walking the tree we find ‘d’ node that we will attach to the tree as it is not already included in the tree.

At the moment the algorithm is going back to the parent, those pages will change their element’s location, imagine ‘c’ node is not a child from ‘a’ node anymore. The real tree will be the following.

However, the tree we are walking is the one represented by the previous figure (see Figure 4.3). Whenever the driver needs to click down to node ‘c’, which is no longer son of node ‘a’, it will not be able to click the element as it is not represented in the node’s page.

The easiest way to solve this problem is getting to the URL represented in the node. Other solution could be re-parsing the tree looking for the real father at that moment to click to the element. However, that solution slows down the whole execution considerably.
4.1.3.3 Implementation

Click algorithm tries to click a specific element in many different ways, as a consequence it only calls Selenium library. In each trial, the same calls are done to the same two functions, 'find element by xpath' and 'click'.

Find element by xpath function receives an XPATH string as an entry parameter. XPATH string is used to identify a specific element within the web page to be clicked later. [16]

Click function does not receive any kind of parameter. It is used to click a specific element. There are different ways to select an element. It is used the function explained earlier, get elements.

Finally, in case none of the earlier trials reaches to click the element, the algorithm calls get method from Selenium library. Thanks to this, the driver is able to access the URL represented by the element.

4.2 Automatic scanner

We have designed a simple algorithm to run ZAP scanner automatically over each URL within ZAP record.

4.2.1 Problem

It is possible to find two problems more when developing the crawler. The first of them focuses on automating ZAP’s scan so it is not necessary to launch it after having made a crawl.

The second problem lies in the results returned by ZAP. The results returned by ZAP in a programmatic manner are not well parsed. The parsing of the results in order to offer them in JSON format is erroneous.
4.2.2 Solution

Below we include the solution to the first problem.

URL list = ZAP records;
Setting ZAP proxy settings to listen over 127.0.0.1:8081 for URL in URL
list do
   Scan URL;
end
Sort ZAP alerts, from High to Low;
Generate REPORT;

**Algorithm 7: Automatic scanning algorithm**

In this case ZAP records are the list of URLs that ZAP has in its own record. ZAP alerts are the vulnerability records ZAP makes when finishing each scan.

After configuration process, the automatic scanner throws ZAP to scan each URL. Then, a report is generated.

In order to solve the second problem, we develop an interface in HTML generated automatically by the scanner to show results in clearer way.

4.2.3 Implementation

This algorithm only calls ZAP API and Yattag libraries. Form ZAP API library, it only calls ‘urls’, ‘scan’, ‘status’ and ‘alerts’ functions. Previously any kind of call, it uses ‘ZAPv2’ as a constructor. The constructor initializes the object, to call above functions and to set proxy settings to IP address 127.0.0.1 and port 8081.

Function ‘urls’ lets the algorithm to obtain a list of URLs found in ZAP’s records. This function is stored in the core of the api, as a consequence it is called ‘core.urls’. The algorithm sets the list of URLs to scan.

After obtaining the list, it looping calls scan function through the scan packet. This packet allows the access to the active scan from ZAP. The call ‘ascan.scan’ receives a URL as a parameter. To not start scanning the following URL before the current URL finishes, it calls ‘status’ function, ‘ascan.status’, to check the status of the current scan.

Finally, it calls ‘alerts’ function which belongs to the core of the api too. Calling core.alerts, lets the automatic scanner to obtain results from URL scanning.

Results obtained are sorted from higher to lower risk. It uses Yattag library to generate the report in HTML.

From this library, the following functions will be called, ‘asis’, ‘tag’, ‘attr’ and
Function 'asis' allows the scanner to write text directly into the HTML document. Text is given as a parameter to the function.

Function tag allows the scanner to create tags. The name is given as a parameter. To add attributes to tags, function 'attr' is used to add specific attributes given as a parameter.

Finally, 'text' function lets the scanner to add text to tags.

4.3 Code

As a proof that we have developed the project, we offer the following Git based repository. [25] It contains the code developed for this thesis.
In this chapter we explain tests done over the InCrawler.

Results have been obtained in a biddable way, executing the automatic crawler over one deliberately vulnerable application later introduced, WebGoat. To be able to compare results, the behavior of a security auditor with the application is reproduced, in order to later scan the application through ZAP.

In this way it is possible to mirror steps explain below (see chapter 4).

Before getting into the details of our evaluation, it is important to introduce some aspects to explain how to carry the evaluation and why it is done this way.

We evaluate the crawler focusing on three concepts, vulnerability detection, coverage and time. In addition to have a comparison to test everything right, we evaluate these three concepts over a manual test.

Manual test has been done reproducing the behavior of a security auditor with the web application under test. Starting from the login panel, credentials were submitted and the chronometer started. From this point, all visible buttons were clicked, and all fillable elements where filled in order to find vulnerabilities and check whether there is or not a vulnerability behind each element.

Before starting, proxy settings from the browser where set to IP address 127.0.0.1 and port 8081. This is where ZAP is listening, capturing all HTTP traffic generated in the application.

After testing is done, ZAP scanner is thrown to scan for vulnerabilities within its records that will have now all URLs from the application. The scanning policy is set to 'insane' which means it insanely tries to find the following vulnerabilities: server side include vulnerabilities, cross-site scripting vulnerability, both persistent and reflected, SQL injections, server side code injections, remote OS command injection, buffer overflow vulnerabilities, format string error vulnera-
Chapter 5. Evaluation

This is not intended to specifically check how many vulnerabilities have been found, but it is intended to compare the crawler with manual detection and check how is the crawler doing.

5.1 Test Subject

As previously introduced, we have used WebGoat as a test subject in order to test the application developed.

WebGoat has been developed by OWASP, which makes it a perfect benchmark for all those tools focused on security in web applications.

We use WebGoat 7 (November-2017). This is not the last release, however, to get some stability and an easy deployment, we use Docker. The last publicly available image of web goat within Docker Hub is WebGoat-7.1. Docker is software performing operating-system-level virtualization or containerization. Docker allows independent containers to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines. Docker Hub is a registry service on the cloud storing Docker images built by other communities.

WebGoat is divided into different lessons. Out of 19 lessons, 16 contain different vulnerabilities.

5.2 Automatic crawling

Next we will show results obtained when using our crawler to explore content in WebGoat and later scanning the HTTP traffic generated.

5.2.1 Vulnerability detection

Vulnerability detection focuses on results given by ZAP’s scanner. This will let us analyze the coverage of the crawler.

Initially, vulnerabilities found are stored in Tot. count column (see Table 5.2). We have analyzed results in order to get true positives. Results contain false positives. To have a real comparison, the last column stores real vulnerabilities currently found on WebGoat.

Automatic testing has found 11 out of 15 true positive XSS vulnerabilities. False positive ratio from total vulnerabilities detected is 26.66%. This supports the theory that it is simpler to find vulnerabilities related to the browser. Auto-
matic testing has found 3 out of 55 true positive SQL Injections. False positive ratio from total vulnerabilities detected is 94.54%.

Browser related vulnerabilities (XSS), are easier to find for the crawler, hence, easier to detect for ZAP. On the contrary, vulnerabilities related to the data layer (SQL Injection) are quite difficult to detect, and exploiting may not be automatic.

5.2.2 Coverage

Coverage measures the ability of the crawler to discover the whole web application. Discovering the whole application means how many URLs the crawler has found within the application. To have a representative measure, trees behind each execution will be used. Counting nodes between the automatic crawler and the manual crawler tree, will give us the opportunity to analyze the crawler’s performance.

The tree generated by the application is quite wide. Seventy-nine nodes are direct descendants from the root node. This implies a considerable wide extension, compared with the number of nodes from levels 3 and 4 (see Table 5.1).

5.2.3 Time

Time is divided into crawling time and scanning time for each time analyzed. Automatic crawler has a better time than manual crawling. Crawler has taken 6 minutes and 19 seconds to crawl the application and the scanner has taken 4 minutes and 58 seconds.

5.3 Manual crawling

Next we show results obtained when manually interacting with WebGoat and later scanning the HTTP traffic generated.

5.3.1 Vulnerability detection

ZAP could detect 21 XSS vulnerabilities. Out of 21, 11 are true positives, which means false positive ratio detection is 47.61%. SQL Injections discovered by ZAP are 115. Out of 115, 112 are false positives, which means false positive ratio detection is 97.39%. Results agree with tendency observed previously, (see Subsection 5.2.1). Nevertheless, ratios have increased due to coverage while crawling manually. Also, one parameter tampering vulnerability has been detected which supports coverage results (see subsection 5.3.2).
5.3.2 Coverage

Tree generated through manual crawling is wider than the tree generated by the automatic crawler. It is observed that total node number is 108. This means levels 2, 3 and 4 need to store the difference, growing to 83 nodes in level 1, 12 nodes in level 2 and 12 nodes in level 3, against automatic crawler tree 79 nodes in level 1, 10 nodes in level 2 and 3 nodes in level 3.

Clearly manual crawling coverage of the application is better.

5.3.3 Time

Manual interaction has taken 7 minutes and 22 seconds. Scanning has taken 9 minutes. Interaction has been done doing the following. We have clicked, filled all values and submit information, found while navigating in WebGoat. In this case, results obtained differ from the possible trend found in the automatic testing results. Scanning has taken around 2 minutes more than crawling. This is something common, as manual testing has found more URLs within the application than automatic testing. Time for scanning should be higher as ZAP is trying to find as many vulnerabilities as possible.

With each attack specified in this chapter’s introduction, it will try one key word from a list of common keywords to reproduce each attack. For example, while trying SQL injections, it will send one request filling an input value with a specific value to test the injection, for example ‘’or 1=1–’. In case this trial is not working, it will try again with other ‘injection’. ZAP uses a list with at least 7000 different injections. This is not intended to demonstrate how ZAP finds vulnerabilities, it is intended to demonstrate why ZAP takes that much time to scan WebGoat.

5.4 Comparing results

Comparing trees from both manual crawling and automatic crawling, manual crawling tree is bigger, which means it has found more URLs. The difference is about 15 URLs more. It implies there are more URLs to scan. Now, depending on the difference between levels, we can find more or fewer vulnerabilities.

In this case the highest difference is between level 4 nodes. However, the difference between level two nodes will make the tree wider on top which means more URLs can be found at the bottom.

As a consequence, manual crawling has found more vulnerabilities even riddling results. To support this results, (see Table 5.3) shows options ZAP is rating
as vulnerabilities which are not real vulnerabilities. Those are risky configurations within the browser used. This table is meant to show how manual detection has gone further than automatic detection, however the automatic crawler is not further from real testing.

Time is another consequence of the differences between coverage. After analyzing results obtained, manual crawling is taking around 5 minutes more to finish than automatic crawling. This high difference comes from time scanning. It is obvious the automatic scanner is missing 15 URLs.

Changing the point of view, if we compare time taken for each test, automatic crawling is faster. The difference can be measured through the number of URLs. If manual crawling is taken 7 minutes and 22 seconds for 108 URLs, means each URL is taking around 4 seconds. Automatic crawling is taking the same time 4 seconds for each URL. This means scanning is making the difference.

It is possible to observe results from automatic crawler will not differ too much from manual crawler. The most notable difference resides within coverage, however vulnerability detection differences are so high.

In this specific case, vulnerabilities found within WebGoat, are found behind elements to be filled by the user. This tends to demonstrate the automatic crawler is missing URLs from WebGoat, however, despite the difference with manual crawling, automatic crawler is finding where vulnerabilities are stored.

<table>
<thead>
<tr>
<th>Statistics</th>
<th>Automatic crawler</th>
<th>Manual crawler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Levels</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Total nodes</td>
<td>93</td>
<td>108</td>
</tr>
<tr>
<td>Level 1 nodes</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Level 2 nodes</td>
<td>79</td>
<td>83</td>
</tr>
<tr>
<td>Level 3 nodes</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>Level 4 nodes</td>
<td>3</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 5.1: Tree comparison

<table>
<thead>
<tr>
<th></th>
<th>Automatic crawler</th>
<th>Manual crawler</th>
<th>Real Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSS</td>
<td>15</td>
<td>11</td>
<td>21</td>
</tr>
<tr>
<td>SQL Injection</td>
<td>55</td>
<td>3</td>
<td>115</td>
</tr>
<tr>
<td>Parameter tampering</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.2: Vulnerability and time comparison
Chapter 5. Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Automatic crawler</th>
<th>Manual crawler</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-Frame-Options Header Not Set</td>
<td>110</td>
<td>139</td>
</tr>
<tr>
<td>Cookie No HttpOnly Flag</td>
<td>5</td>
<td>59</td>
</tr>
<tr>
<td>No Cache-control and Pragma HTTP Header Set</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Web Browser XSS Protection Not Enabled</td>
<td>109</td>
<td>140</td>
</tr>
<tr>
<td>X-Content-Type-Options Header Missing</td>
<td>179</td>
<td>179</td>
</tr>
</tbody>
</table>

Table 5.3: Browser parameter vulnerabilities

5.5 ZAP testing comparison

The aim of this chapter is not comparing exactly ZAP with the solution implemented; however, as ZAP can be used in a totally automatic way, we compare some results obtained from using ZAP automatically to crawl and scan.

To obtain the best results from ZAP, the spider is thrown after login into WebGoat as a guest user. This gives ZAP access to the main page of the application. After the crawler is finished, the scanner is thrown with the specific parameters, introduced in this chapter’s introduction.

Analyzing coverage obtained from analyzing WebGoat automatically with ZAP, it is possible to observe poor results. It has found only 22 different URLs. ZAP’s algorithm lack of stability can be noticed too. Comparing results with table 3.1 from chapter 3 introduction, results differ and experiments have been done the same way.

Comparing trees, it has been obtained 5 levels. The number of nodes from level 2 is quite low. It is one of the levels with fewer nodes, in this case 3 nodes. This implies the fewer branches there are, the less leave nodes we will have, which means coverage will be low as information is lost in not visited branches.

As a consequence of poor results obtained in coverage, differences found between possible vulnerabilities are considerably high. In fact the scanner has only found that vulnerabilities. It could not find any other vulnerability from previous tables (see Table 5.2). This table is added to demonstrate number of request from ZAP to the application, even with insane scanning politic is quite low, which means too much information has been lost.

Finally, time difference is around 10 minutes. This is a consequence of all information ZAP is loosing. Currently time taking to scan is 40 seconds. Time taken for ZAP to crawl is not measurable in seconds.

It is possible to conclude the solution implemented obtains considerably better results than ZAP’s crawler, which is not even stable.
Table 5.4: Tree comparison ZAP

<table>
<thead>
<tr>
<th>Statistics</th>
<th>Automatic crawler</th>
<th>ZAP crawler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Levels</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Total nodes</td>
<td>93</td>
<td>22</td>
</tr>
<tr>
<td>Level 1 nodes</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Level 2 nodes</td>
<td>79</td>
<td>3</td>
</tr>
<tr>
<td>Level 3 nodes</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>Level 4 nodes</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>Level 5 nodes</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

We have also used DVWA, Bodgeit and WackoPicko to evaluate the performance of the InCrawler. We have considered only including results from WebGoat as they were the most significant ones. However, coverage obtained crawling Bodgeit and Damn Vulnerable Web App (DVWA) by the InCrawler is 100%. Coverage obtained crawling WackoPicko is 74%. Both Bodgeit and DVWA applications have been explored completely. However, ZAP is able to explore only 16% of the content of DVWA and 56% of the content of Bodgeit. ZAP has only explored 64% of the content of WackoPicko. Results obtained by the scanner have not turned out to be relevant in terms of vulnerability detection.

Table 5.5: Vulnerability and time comparison ZAP

<table>
<thead>
<tr>
<th></th>
<th>Automatic crawler</th>
<th>ZAP crawler</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-Frame-Options Header Not Set</td>
<td>110</td>
<td>7</td>
</tr>
<tr>
<td>Web Browser XSS Protection Not Enabled</td>
<td>109</td>
<td>7</td>
</tr>
<tr>
<td>X-Content-Type-Options Header Missing</td>
<td>179</td>
<td>14</td>
</tr>
<tr>
<td>Time</td>
<td>11:23:13</td>
<td>00:40:76</td>
</tr>
</tbody>
</table>
Our goal was to develop a crawler capable to explore content stored in web applications in a different way classic crawlers do. We have managed to explore all those input elements and text area elements. In addition, exploring all those clickable elements and to submit information to the application has been achieved.

However, the crawler have certain limitations that are discussed below.

6.1 Limitations

It is possible to find limitations in all areas of the project. In order to give certain lines of future work, the most notable limitations found in the project are offered.

L.1 Trees:

The main problem of tree data structures lies in the structure itself. It is a stable structure, but it is difficult to adapt it to the needs of the project. It is only possible to traverse the tree in a specific way since the amount of data increases as it is being crafted, so it is not possible to make another route. What we mean with this is, as nodes are being attached to the tree as they are being discovered, the only way to traverse the tree is through pre-order iteration.

Representation of web applications can be accurately stored in tree data structures, however, the real structure is not represented. Imagine two URLs having a link in common. This means it is possible to visit that link from both of them. Current algorithm will attach the link (node) to the page it has used to visit the link, so the father would be just one of the pages, when the node should have two parents.
L.2 Dictionary:
The dictionary is a simple structure of four columns storing data used to fill input fields and text area fields. At this moment the dictionary is really small. Also, it is not ready to put more specific values, such as numbers. Currently, the dictionary is not prepared to fill numbers, for instance. It has no way to detect which specific values to store into input fields. For example if an eight digit password is needed, it is prepared to detect if the input value is a password, but not the policy used for passwords.

L.3 Clicking:
Clicking limitation is related to implementation. So far the number of attempts is four, which are developed one by one. Adding a new attempt implies developing it. Currently, as long as another way to click is needed, it should be previously implemented, which means adding its Python code to the existing project. This is a limitation since adding a new attempt should be more simple. For example, if we want to add a new attempt trying to click an `<a>` tag, found by its type value, it is necessary to identify the XPATH syntax, in this case, `//a[@type='type value']`. In addition, else if clauses, return line, click line and exception line should be added.

L.4 Non focus problems:
In this case one of the most common problems that are not being solved in this thesis is that our crawler avoids repeated URLs but not repeated content. If there are two different URLs that access to the same content, the crawler will access the same content twice and therefore lose time. For example, crawling a web page whose home button is being referenced from different parts of the web page. In this case, clicking the button and a redirection from other page by filling a form could be both ways to travel to the same page. The crawler first finds the URL parsing tags within HTML code. Next it fills the form and it is redirected to the same page it has found while parsing tags. The same content is referenced two times but URLs storing this references will not be the same.

L.5 Efficiency:
Finally the efficiency of the crawler is questionable. Depending on the architecture where testing is done, the crawler could perform in different ways, which implies some instability.

The crawler is a simulation. That means the application we are crawling needs to be simulated (which normally is quite heavy, it needs lots of resources to work properly) and the interaction is accessing different resources (each time a value needs to be filled a lecture operation over a `.csv` file needs to be done).
6.2 Future works

Afterwards crawler implementation, it is possible to focus future work in different and varied developing lines.

A possible solution for the first limitation can be focused to data structure used within the crawler algorithm. It is possible to change tree data structures into graph data structures. Mapping web applications can be represented through a graph. This will solve the problem of a page with two parents, current data structure cannot solve. Interaction between driver and graph is quite better. Using a graph lets the driver travel to pages without going back to the parent. For example, in the case two neighbor nodes are linked. This means it is possible to find references from one node to its neighbor. Tree representation forces the driver to go back to the parent to travel the neighbor. However, using graphs will simplify traveling, as both neighbors will be linked. Also, using graphs implies a possibility to use the page rank algorithm. Page rank algorithm will simplify and accelerate the crawler execution. Integration with the driver will be more optimal. Page rank algorithm is based on the following formula (see figure 6.1).

$$PR(A) = (1 - d) + d \sum_{i=1}^{n} \frac{PR(i)}{C(i)}$$  \hspace{1cm} (6.1)

Figure 6.1: Page rank crawling algorithm

`PR(A)` is the page rank itself for 'A' page. 'd' is a dumping factor. 'PR(i)' is the page rank from each page linking 'A' page. 'C(i)' is the number of links coming out from 'A' page.

This is only supposed to replace the node order given by the algorithm. Page rank algorithm is assigning a numerical weight to every page found within the application under test. This weight is given by analyzing links pages contain, and links pages refer to. The higher the numerical weight a page has, the earlier it will be visited. Most important pages will be first crawled so the simulation should go faster by the end of the graph.

A solution for the second limitation, more feasible, would be to enlarge the dictionary in a way crawling will work in a higher number of web applications. This will let the crawler identify and interact with elements within applications. The larger the dictionary gets, more possibilities to fill specific values will be achieved. Enlarging the dictionary is the easiest way to solve this problem. It is only needed a representative sample of web applications, in order to study the components included in them, to introduce the way to detect them and what
value to fill in the dictionary. The most difficult way to solve this is making the
dictionary smarter enough to detect what values to fill into input or text area
values. This solution implies certain text understanding artificial intelligence to
let the dictionary understand what values to fill.

A possible solution for the third limitation could be changing the dictionary
structure to fill different ways to click. In this way, implementation could be
simplified to add XPATH syntax for each way to click to the dictionary. This
simplifies clicking method to a single clicking line and exception capture lines.

A possible solution for fourth limitation could be fixing problems found in
classic crawlers that are not approached in this thesis. This line could be focused
on finding a solution to detect repeated content stored in the tree. This could
be done comparing content from each page. Mapping and comparing elements
could be done while crawling. Also, to detect redirections by checking the URL
the driver is going to crawl, will avoid crawling different times the same page.

Finally, a possible solution for fifth limitation could be upgrading the algo-
rithm in a way it would be capable to crawl faster and more efficiently. Light-
weighting the crawler execution means making a simple simulation.
In this thesis, we developed a crawler able to explore content in web applications in a different way classic crawlers do. We began our work studying classic crawlers within vulnerability detection applications to check their current state. Then we concluded their results indicated their logic to explore content in web applications is not enough, so they need user interaction. Next, having problems stated, we proposed a different approach to explore content in web applications. Finally, we offered an evaluation showing results obtained from our crawler compared with results obtained from real testing.

After studying the initial idea and its future evolution into the current idea, we could demonstrate our solution gets to imitate quite accurately human behavior with a web application.

From the implementation point of view, after analyzing the problems found and their possible solutions, it is possible to reach the following conclusion, Selenium’s WebDriver is not focus on making general simulations. This tends to explain Selenium is focused on making specific tests over web applications. It is not prepared to simulate such a general behavior as this thesis is doing, clicking all elements, fill all input values, submit all elements, etc.

On the contrary, it works properly when looking for specific objects. XPATH function is one of the most reliable functions within Selenium. However, general elements usually fail to be found through this method.

As a consequence, four different methods to click are needed. This slows down the crawler execution. Besides, if we test a big application, the algorithm tends to slow down while traversing top elements from the tree. Execution can be even slower.

Nonetheless, along with Python, designing a non so extensive solution and properly human behaving is possible.
Still, the use of depth first search pre-order algorithm, allows to define a stable solution, however it is slowing down the crawler. Relation between the driver and the algorithm is not entirely optimal.


| **Firmante** | CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES |
| **Fecha/Hora** | Fri Jul 06 19:48:49 CEST 2018 |
| **Emisor del Certificado** | EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES |
| **Numero de Serie** | 630 |
| **Metodo** | urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature) |