# Bagley

Por
Víctor Fresco Perales


**UNIVERSIDAD COMPLUTENSE MADRID**

Grado en Grado de Ingeniería Informática
Facultad de Facultad de Informática

*Dirigido por*
José Luis Vázquez-Poletti

**Bagley**

Madrid, 2021–2022

# Bagley

## Automated tool for reconnaissance and vulnerability detection in Bug Bounty environments

*Memoria que se presenta para el Trabajo de Fin de Grado*

**Víctor Fresco Perales**

*Dirigido por*

**José Luis Vázquez-Poletti**

**Facultad de Informática**
**Universidad Complutense de Madrid**

**Madrid, 2022**

# Abstract

Bug Bounties are monetary rewards that companies pay to independent security researchers when they successfully find and report an exploitable vulnerability. A bounty for a critical vulnerability in a big company can reach the equivalent to a year's salary in Spain, and this amount is not defined by the complexity of the bug, but by the impact of it. This means that very simple to find and exploit bugs that affect critical infrastructure can report a very big amount of money if the person who finds it is in the right place, in the right moment. The goal of this project is to build and maintain an automated tool that runs on its own, in a Virtual Private Server and is able to perform reconnaissance and detect these simple vulnerabilities in a target. It also implements a communication interface over Discord, so that the researcher can operate it at any moment with any device and find out immediately if something is discovered, making it the perfect tool for assisting bug hunters.

# Agradecimientos

Gracias a mis padres Pilar y Gelo por apoyarme siempre, incondicionalmente. A mis abuelos Isabel y Felipe por haber trabajado más de una vida por todos nosotros. A mi tia Vivi por enseñarme cómo disfrutar de las cosas, y a mi tio Emilio por enseñarme a ver más allá de ellas.

Gracias a Patricia por haberme aguantado a su lado durante todo el proceso. A mis amigos, a quienes les dedicaría mucho más que un simple párrafo de agradecimientos, por estar siempre ahí.

Especial gracias a José Luis Vázquez-Poletti y a David Pacios Izquierdo por haberme dado la motivación necesaria para emprender un camino en la seguridad informática.

# Sobre TEFLONX

Teflon X(cc0 1.0(documentación) MIT(código))es una plantilla de LaTeX creada por David Pacios Izquierdo con fecha de Enero de 2018. Con atribuciones de uso CC0.

Esta plantilla fue desarrollada para facilitar la creación de documentación profesional para Trabajos de Fin de Grado, Trabajos de Fin de Máster o Doctorados. La versión usada es la X

V:X Overleaf V2 with XeLaTeX, margin 1in, bib

**Contacto**
**Autor:** David Pacios Izquiero
**Correo:** dpacios@ucm.es
**ASCII:** ascii@ucm.es
Despacho 110 - Facultad de Informática

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

Nowadays, every medium and big company must develop software at some point of their life, whether it's just a static web page or it's a complex multi-platform application running on millions of devices. These companies tend to already have security experts that audit their code and recommend them with best practices in order to stay safe, however, the security field is too wide, and attackers only need a single mistake from the developers to inflict great harm. For this reason, some of these companies are willing to pay big amounts of money to anyone who finds and reports a vulnerability that could be exploited by a malicious actor in any of their products. These payments are known as Bug Bounties and they have spread a lot in recent years, so much that there are entire platforms dedicated to connecting companies and security researchers, acting as intermediaries between them. The most notorious ones are *HackerOne*, *Bugcrowd* and *Intigriti*.

The amount paid by these companies usually varies between $50 and $10.000 per vulnerability, depending mostly on the impact of it and the company itself [1], but there have been cases of much bigger amounts. Many of these vulnerabilities are caused by very complex bugs in the code of the applications, which require days or even months of extensive research and a lot of previous knowledge of the technologies behind them. However, there are others which are not difficult to find or exploit at all, and just require that the researcher is in the right place at the right moment so they can be discovered. These vulnerabilities are known as *low-hanging-fruit* and are usually found by the first person who pays attention to the place where the bug is. Some examples of them are *subdomain takeovers*, *data leaks* or even some easy *SQL injections* in URL parameters or POST data.

An important aspect to consider when doing Bug Bounties is the reconnaissance, commonly known as *recon*. It is the first phase of the research process when targeting one of these companies, at least those offering web services, and it involves listing all available assets from a target, finding hidden services, etc. It results in impossible to do this manually, so researchers use automated tools to do so. It's crucial to perform reconnaissance in depth, since it allows to expand the surface to find bugs on. It can lead to finding

[1] `https://www.hackerone.com/resources/reporting/hacker-powered-security-report-industry-insights-21`

components that are not as exposed as others so developers don't put that much effort into securing them and can contain critical bugs. There is also a possibility of discovering something that shouldn't be publicly accessible such as an administration panel or API keys, which would also be considered "low-hanging-fruit".

Most of the finding of this *low-hanging-fruit* and the reconnaissance can be automated to be running 24 hours a day on a Virtual Private Server and notify the researcher when something interesting is found, so he or she can focus on analyzing these results and finding more complex bugs. The payments received from reporting these bugs will be greater than the cost of maintaining the tool (mainly the cost of the Virtual Private Server), so it can be a profitable project in the long run.

The name comes from the popular video game saga *Watchdogs*, precisely, *Watchdogs Legion*, in which the main characters are assisted by an AI named **Bagley**, which helps them along their adventures. Although this project does not contain any AI, we thought it was a good name.

## 1.2 Goals

The main goal of this project is to build and maintain an automated tool that can perform basic tasks of reconnaissance and vulnerability detection to a set of given targets without any required supervision, and notify the operator when a potential vulnerability is found. This project will be focused only on web applications, so targets will be domains or subdomains which the operator has permission to test on. The ultimate goal is to find as many vulnerabilities as possible with very little interaction from the person operating it. The tool must be able to:

- Be running indefinitely and independently from the operator, for example, in a VPS (Virtual Private Server).
- Communicate with the operator via an existing messaging app. It must be able to send different notifications for different events (errors, logs, vulnerabilities found, etc.) and the operator must be able to send predefined commands to perform basic actions in the tool, (*start*, *stop*, *add target*, etc.).
- *Crawl* a target to find as much content as possible, imitating a real user. It must be able to render JavaScript so that modern web applications are crawlable too.
- Perform content discovery with alternative methods such as *brute forcing* or consulting third party sources.
- Look for known vulnerabilities in the technologies used by the targets.
- Check if any subdomain can be claimed to an external hosting provider (*subdomain takeover*).
- Discover simple authorization bypasses.
- Analyze client-side JavaScript looking for vulnerabilities and endpoints.
- Analyze responses looking for patterns in order to find credentials or API keys.

- Discover injection vulnerabilities such as *SQL injection*, *XSS* or *SSTI* by brute-forcing parameters in an efficient way.

Some of these goals can be accomplished with existing free, open-source applications maintained by big communities. In those cases, the task will be to make them work as if it was all one big application.

## 1.3 Document Structure

First chapter is the Introduction, which contains the motivation behind this project, the goals of it and the structure of the document, which is this section. Second chapter is the State of Art, which presents every aspect surrounding this project. It contains a section explaining what is a vulnerability, a section digging into what are Bug Bounties, a section enumerating the different options available for remote management and a section about other applications related to this project. Third chapter explains everything about the architecture of the system, including its database, its main script, its deployment and the Virtual Private Server in which it is hosted. Forth chapter presents the use case for the system and fifth chapter explains the conclusions obtained from the project and the future of it. Lastly, there is an annex detailing how to use the system.

# Chapter 2

# State of the Art

## 2.1 Vulnerabilities

This project will deal extensively with vulnerabilities, so it's important to give them a formal definition before anything else. A vulnerability is a weakness in an application that allows a malicious actor to perform some unpermitted actions or gain access to information they shouldn't otherwise be allowed [1]. A bug is not strictly the same as a vulnerability, since a bug can be a simple error with no security implications, however, they will be treated as synonyms in this document, due to the wide usage of the word *bug* to refer to a vulnerability in the Bug Bounty world.

## 2.2 Bug Bounties

In order to better understand the goals and limitations of this project and the technologies surrounding it, we have to extend the earlier definition given to Bug Bounties and explain some key concepts about them. So, as previously mentioned, Bug Bounties are payments that security researchers receive when finding and reporting a valid bug or vulnerability that could be exploited by a malicious actor in a company's service [1]. This obviously includes web applications, but also desktop applications, mobile applications, smart contracts deployed on the Blockchain, physical devices such as IoT, etc. Security researchers who are dedicated to this are usually called *bug hunters*.

Those companies interested in offering this kind of rewards may have a Bug Bounty program on their own (i.e Google or Apple) or a program running on other independent platforms, such as HackerOne, BugCrowd or Intigriti (i.e PayPal, IBM, Twitter, Uber...). These platforms help companies connect with researchers and vice-versa, while also ensuring that everybody is treated fairly and researchers receive the reward they really deserve. They also provide transparency, allowing everybody to see very precise information about the programs, such as rewards paid in the last 90 days or average response time, and about the researchers, such as vulnerabilities found in the last 90 days or percentage of valid reports. There is also a possibility for companies to enroll in these platforms as private Bug Bounty programs, which require an invitation to see their statistics, test on their services and report vulnerabilities.

[1] `https://www.hackerone.com/vulnerability-management/what-are-bug-bounties-how-do-they-work-examples`

The amount of the payments vary greatly depending on the impact of the vulnerability and the company offering the bounty. The impact can be defined as the potential damage that a malicious actor can do to the company or its customers if the vulnerability is successfully exploited. HackerOne measures the impact as low, medium, high and critical, while BugCrowd measures it as P1, P2, P3... being P1 the highest, but it's essentially the same. An example of a critical impact vulnerability can be a *Remote Code Execution* in the backend server or an *account takeover*, while a low impact vulnerability can be an *Open Redirect* or a *Captcha bypass*.

In 2021, the average bounty given by software companies for critical vulnerabilities was $7000, while for low vulnerabilities was less than $200[2]. As an example, GitLab, which has one of the biggest programs in HackerOne, pays from $20.000 to $35.000 for critical vulnerabilities. Besides that, if a vulnerability is considered critical enough, the bounty will usually be bigger than what the company usually pays for other critical vulnerabilities. That is why it's important for a researcher to demonstrate the impact with easy to understand reports or Proof of Concept videos. However, not every program offers bounties. There are some programs that offer some kind of recognition such as a signed certificate, points inside a platform, etc. or even company products, such as clothes, stickers, etc. (commonly known as *swag*).

Another important concept in Bug Bounties is the scope. It's the set of services or assets that the company allows to test on, and therefore, it's willing to pay a bounty for a vulnerability affecting these services or assets. If a researcher reports a vulnerability affecting a service which is out of scope, the program is not required to pay a bounty. In the case of web applications, the scope is usually given as a list of domains or subdomains, or even complete URLs. Sometimes, there are some vulnerabilities that are also out of scope, depending on the program. They follow the same logic: if a researcher reports an out of scope vulnerability, the program is not required to pay a bounty. Some examples can be *self XSS* or a *misconfiguration in security attributes of a cookie*.

Finally, each program has its own set of requirements, which are rules that the researcher must follow to participate in it. They define what the bug hunter is allowed to do and under what circumstances it is allowed to do that. If a valid vulnerability is reported but these requirements are not met, the program may choose not to pay the bounty, same as with the scope. Some of the most common requirements are to limit the requests per second or to include special headers or cookies in order for them to be able to tell which requests come from a researcher. There are some programs that won't allow automated tools at all.

As a bug hunter, there are a lot of options to choose when doing Bug Bounties, however, this project will focus only on hunting for bugs in web applications. It's also important to mention that the author has more experience in HackerOne than in any other platform, so there may be some concepts greatly tied to it.

[2] https://www.hackerone.com/resources/reporting/hacker-powered-security-report-industry-insights-21

### 2.2.1 Methodology

When hunting for bugs, each researcher has its very methodology and each vulnerability has a unique path that the bug hunter has traversed in order to find it. However, there are some common practices that the vast majority of researchers use in order to find those bugs. They don't guarantee finding anything, but they narrow the process and make it easier. This subsection aims to explain these common practices to later understand better how automated tools can really assist the bug hunter.

Once the researcher has chosen a target, which, as has already been said, can be a domain, a set of subdomains or complete URLs, the first step is to manually explore the web application. This is, basically, using it as a regular user: creating an account if an authentication system is implemented, using all available services, etc. The main goal is to learn how the application is designed, what are its main functionalities and how the developers expect the users to interact with it. This allows the researcher to understand how the application was built and where the developers might have missed something that can potentially be exploited, that is, a bug or a vulnerability.

While the researcher makes this initial contact, he or she will be looking at the requests that the browser is sending and the responses produced by the server. This is done with a proxy that sits between the browser and the server in the machine of the researcher, such as the one integrated with *Burp Suite* or the one in *OWASP ZAP*. It lets the bug hunter analyze the cookies being sent, the headers, type of requests, etc. but also intercept and change those requests, repeat them... When something unusual happens, such as an unsigned cookie specifying the username, or many requests made to an endpoint resulting in strange errors, the researcher focuses on that part of the application, to check if something can be exploited.

Many times, if one of those unusual behaviors looks like it can be product of a widely known vulnerability, such as an SQL error, which might mean there's an SQL injection, the researcher can use automated tools to try to exploit it and save time. Maybe the application is protected by a WAF and the exploitation needs the researcher to work on a bypass for that, but if that's not the case, the tool will succeed and probably take less time than the researcher. There are very powerful tools to exploit widely known vulnerabilities that don't require complex bypasses, such as *sqlmap* for SQL injections, *dalfox* for XSS and many more.

When everything has been inspected and nothing remarkable to focus on has been found, the researcher still can expand the area to look for bugs. This is called *recon*, a short for *reconnaissance*. One of the methods to do recon is content discovery, which is finding resources that are not accessible just by navigating the application, for example, a subdomain that is used by employees but is not directly linked to the main web application. This is usually done by *brute-forcing* or by consulting third-party sources, such as *Shodan* or *the Wayback Machine*. Content discovery is usually aimed to finding subdomains and URL paths, and can uncover very interesting resources, such as functionalities that are no longer maintained but still affect the company so they are susceptible to having impactful bugs or even content that is not supposed to be publicly accessible, such as credentials or administrator panels. In order to perform this technique, researchers use automated tools such as *gobuster* or *ffuf* for brute-forcing or *gau* for consulting third-party sources.

All these explained techniques have a disadvantage, which is very common among cybersecurity, and it's that they all belong to *black box testing*, which means that they do not look at the internal workings of the technology that is being tested, so the researcher never knows for sure how the web application is working in the backend. On the contrary, *white box testing* means to actually look at the internal functionalities of the technology being tested [2], for example, code analysis. In Bug Bounty environments, programs may have the backend code of the web application available as open source, however, that's not a regular thing. Nonetheless, every modern Web Application runs client-side JavaScript, which can be inspected and even debugged. Reviewing this code can reveal a lot of client-side vulnerabilities such as *DOM-based XSS* or *Open Redirects*. A researcher can even find new endpoints in the web application, credentials, API keys, etc.

### 2.2.2 Automation

As already discussed, vulnerabilities can have different impacts, depending on the potential harm they can produce. However, this impact is not directly related to the difficulty of finding and exploiting them. This means that a critical vulnerability doesn't have to be hidden deeply in the code, protected by multiple layers of security. Some very critical ones can be *in plain sight*, waiting for them to be discovered by the first researcher who focuses on that part of the application. An example of this was the CVE-2021-44228 vulnerability, commonly known as *log4shell*. This bug was one of the most impactful ones in recent years, due to the amount of servers that were affected and its ease of exploitation [3], but the origin of the vulnerability itself was not obscure at all. There was even a conference talking about the same issue that produced this vulnerability in Java applications from a couple of years before it was discovered [4].

The easiest bugs to find are commonly known as *low-hanging fruit* or *low-hanging vulnerabilities*, and they just require that the researcher is the first one to report them. An example of them are *subdomain takeovers*, which are a type of vulnerability in which the researcher is able to claim a subdomain from a legitimate site, usually hosted on a third party provider [1]. In this case, the researcher just has to notice that the subdomain is unclaimed and contact the provider in order to earn a bounty for a medium to high impact vulnerability. However, manual testing is often very slow and it would be tedious for a single researcher to be testing every possible endpoint of an application. Low-hanging vulnerabilities may be easy to find, however, they can be anywhere, so its difficulty relies on testing all potential points of failure. The solution to this problem is automation. A researcher can create tools that perform all this tedious testing, whenever there is no complex reasoning behind and the output is predictable.

Vulnerability detection is not the only task that can be automated. It has been described in the past subsection how researchers use existing tools to perform recon on a web application. These tools can be further chained one after another, so that the process is fully automated, integrating even the vulnerability detection. This whole automation will allow the researcher to easily find low-hanging vulnerabilities without having to test everything manually, allowing him or her to focus on more complex vulnerabilities that require some research and investigation.

## 2.3 Remote management

The most common way of managing a tool that is running on an external server, such as a VPS, is to directly connect to the server and then manage the tool from the inside. This connection is usually done by SSH (Secure Shell), which is a protocol for secure remote login over insecure networks[3]. This is a perfect option if the operator wants to do maintenance, check the logs and state of the tool, modify its behavior through parameters, etc. However, sometimes the optimal situation is that the operator gets notified when a significant event happens, for example, an error in the tool, an important result, etc. This is even more critical in the Bug Bounty environment, since there is always a possibility of another researcher finding the same vulnerability. In those cases, the first one that reports it is usually the one getting the bounty, so it's crucial for the researcher to know that a vulnerability has been discovered as soon as possible. This section will explore the available options to implement a remote management system that fits into this project.

A simple option to implement a management system with notifications is to use email. It would allow the tool to send and receive emails to respectively notify the operator on various events and accept commands to manage it. To do so, many programming languages offer libraries to interact with SMTP protocol, but an email and a password must be hard-coded in the code or provided every time the tool is initialized. A good alternative is to use a service like Sendgrid, which is a cloud-based solution to send transactional and marketing emails[4]. It allows using API keys so that no password is used, adding extra security since API keys can be easily revoked and regenerated, and its access can be limited. The main inconvenience about this is that emails are not instantaneous and can sometimes be misidentified as spam. It would be tedious for the operator to manage the system having to wait for the results of each command.

The perfect email alternative is to use an instantaneous messaging application that supports automation through an API. This is the case of Telegram, which is a free application that offers cross-platform, cloud-based instant messaging and file-sharing services among others[5]. It offers an API compatible with many programming languages in order for the system to be able to use it[6]. Since it's an instant messaging application, notifications would be delivered right away, and the management of the system through commands would be a comfortable experience for the operator. Since the application is cross-platform, it can be used in any device, allowing the operator to use it anywhere.

[3] `https://datatracker.ietf.org/doc/html/rfc4251`
[4] `https://sendgrid.com/`
[5] `https://telegram.org/`
[6] `https://core.telegram.org/`

Although Telegram is already a very valid solution, there are other options offering a few shallow improvements, such as Discord or Slack. Both of them are platforms that allow users to communicate with instant messaging and VoIP, allowing also to send files, create servers with different channels, etc. Their main difference between them relies on the initial goal design of each one: Discord was oriented to gaming environments while Slack to the workplace[7][8]. Both of them offer an API in many languages, which can be used to create bots or other kinds of automation over the platform. The main improvement that these applications provide over Telegram is the use of channels in a server. They can be used to classify notifications so that each channel corresponds to a different event. This way, channels with notifications regarding errors and findings can be active while others dedicated to logs and showing the state of the system can be muted. Discord was finally used in this project simply because the author uses Discord a lot, while he has never used Slack.

## 2.4 Similar software

As already stated, each researcher has a very unique methodology, and although there are common practices, finding a bug is the result of taking many decisions that depend greatly on the person behind the keyboard. This means that building an automation tool for Bug Bounty environments is a very personal process, in which each researcher chooses to implement a different set of tools and techniques that he or she thinks will be more efficient for finding bugs.

Besides that, this set of techniques and methodologies that are implemented in an automation tool defines how the researcher will discover bugs in Bug Bounty programs, resulting in the hunter earning some money. It's easy to understand that those researchers who successfully create and maintain their systems don't want to make them public, so that other hunters don't find and report the same vulnerabilities as them. This would result in the original developer earning less money than if the tool had remained private.

These two situations explain why there is not a single successful public tool with the similar capabilities to those of this system. Although there are many recommendations for building automation, and many tools such as Nikto[9] or Nuclei[10], which can be used to scan entire web applications and look for some vulnerabilities, there is not a single tool that can do recon and vulnerability detection in a target, can be managed remotely and can be easily deployed anywhere.

[7] https://discord.com/
[8] https://slack.com/
[9] https://github.com/sullo/nikto
[10] https://github.com/projectdiscovery/nuclei

# Chapter 3

# Architecture

This chapter will explain the architecture of this project. It consists of a main script written in Python that performs the recon and the vulnerability detection, and a MariaDB database which stores all produced data, both running in separate Docker containers. These containers are deployed in a Linode VPS. The main script controls a Discord bot, through which it communicates with the operator. In case of an emergency or maintenance, the operator can connect directly to the VPS via ssh.

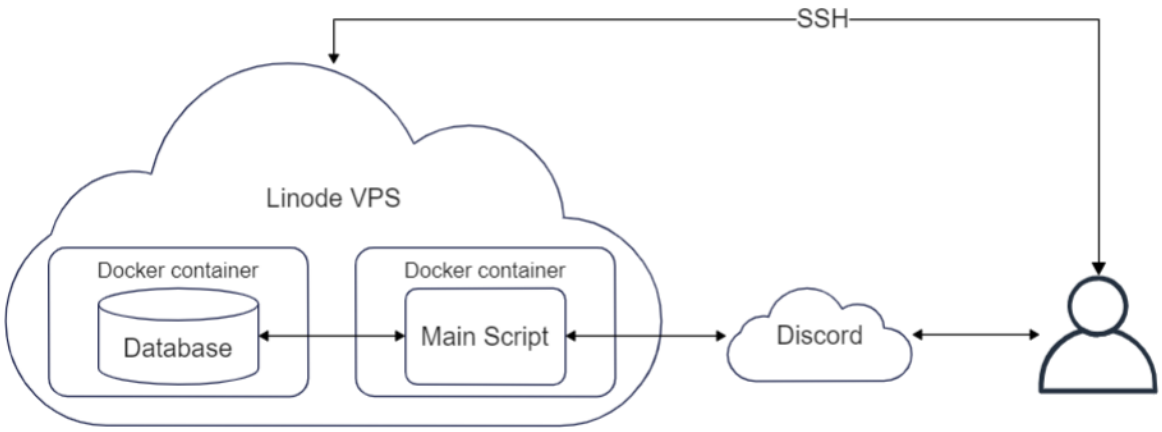


Figure 3.1: Overall architecture of the project

## 3.1 Database

The database used is MariaDB, which is an open-source relational database. It was born as a fork from MySQL, in order to stay free and open-source, so it uses the same syntax as MySQL[1].

[1] https://mariadb.org/

A relational database was used instead of a ORM (Object Relational Mapper) because of a series of factors:

- The access to the database needs to be fast and reliable, taking into account that there are several threads writing and reading at the same time, and ORMs are not as efficient as directly accessing the database.
- The application uses objects and queries that may not work well with current ORMs due to complexity, including some objects combining storage in database and in files.
- The author of this project was very comfortable with SQL databases but not with ORMs at the time of developing this project.

Its main purpose is to store a model that represents the targets of the tool. This model is built by the main script when doing recon and then queried by it to look for vulnerabilities or to do further recon. The database also stores some parameters used by the main script and the actual vulnerabilities found.

At the conceptual level, this model can be represented as an Entity Relationship model, as shown in the diagram in figure 3.2. It's composed by the following entities:

- **Domain**: Represents a domain or a group of subdomains that are in scope. Targets are specified to the tool as domains or groups of subdomains, so they are inserted here. Each domain has an ID, a domain name, a set of headers, a set of cookies and a set of excluded submodules. Groups of subdomains are stored with the format `.<parent_domain>`, so for example, the group of subdomains `.example.com` would contain subdomains such as `www.example.com` or `api.example.com` . If a port is specified, it's inserted as part of the domain name, for example `example.com:80`. The headers and cookies associated with a domain must be sent with every request made to it. In case of subdomains, it also applies to every child that is inserted in the database after it. This can be useful to scan web applications that require authentication or when a Bug Bounty program specifies headers or cookies that must be sent with every request. They are references to header and cookie entities. The excluded submodules specify to the main script which submodules not to execute for this domain. In the case of subdomains, it also applies to all children.
- **Path**: Represents a path to a resource inside a URL. Each path has an ID, a protocol, a domain, an element, a parent and a set of technologies. The protocol specifies which one is used to access the resource. It is usually `http` or `https`. The domain is the actual domain in which the resource is located, and it's a reference to a domain entity. The element is the resource that the path is pointing to. The parent is the directory in which the resource is, and it's a reference to another path entity. For example, for a path representing `https://example.com/animals/cat`, the element would be `cat` and the parent would be the path representing `https://example.com/animals/`. The set of technologies are those that are used in the resource that is represented by the path. They are references to technology entities.

- **Request**: Represents a request that the browser has made to a website. Every request has an ID, a path, a method, parameters, some data, a response, headers and cookies. The path is the requested resource, and it's a reference to a path entity. The method is the HTTP method used, commonly `GET` or `POST`, although it can be many others[2]. The parameters of the requests are the query of the URL, which is stored as a single string because, although URL queries are usually parameters in key/value pairs, the standard only specifies that it must be a string[3]. The data is what is sent as the body of the request when it is a `POST` request. The response, if any, is the response received back from the server, and it's a reference to a response entity. Headers and cookies represent those sent with the request, and both are references to the corresponding entities.
- **Response**: Represents a response received by the browser. Each response has an ID, a hash, a code, a body, a set of headers and a set of cookies. The hash is obtained from the body and the code. It's used to check if a response already exists in the database without having to compare the whole body and the code. The code is the HTTP response code received[4]. The body is the text included with the response, either html, JSON, etc. Headers and cookies represent those sent with the response, and both are references to the corresponding entity.
- **Header**: Represents an HTTP header. Each header has an ID, a name and a value.
- **Cookie**: Represents an HTTP cookie. Each cookie has an ID, a name, a value and every standard cookie attribute: domain, path, expires, maxage, httponly, secure and samesite[5].
- **Script**: Represents a client-side JavaScript file that is interpreted and executed by the browser. Each script has an ID, a hash, a set of paths and a set of responses. The hash is obtained from the script itself (its contents). The paths are the locations in which the script is located in the server. That is, the URLs in which the script can be found. They are references to path entities. The responses are all those in which the script is used. This includes scripts directly located between the `<script>` tags and those referenced by them. These responses are references to response entities. As an example, a script can have associated responses but no associated path if it's embedded in those responses between `<script>` tags, so it doesn't have its own location in the server.

  Scripts are internally stored as files in the container that runs the Python main script, with their IDs as their filenames. This facilitates the analysis with some third-party tools that only accept files as input. Besides that, some scripts that are minified and/or obfuscated will be unpacked, if possible, with the Unwebpack_sourcemap tool (which is explained later), producing folders with many files. These folders will also be named after the script ID.

[2] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods`
[3] `https://url.spec.whatwg.org/#url-representation`
[4] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Status`
[5] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie#attributes`

- **Technology**: Represents a technology that a website is using. Each technology has an ID, a CPE, a name and a version. The CPE is the Common Platform Enumeration name that the technology receives. It's a standardized method of describing and identifying many types of technologies[6].
- **CVE**: Represents a CVE ID, which is an identification code assigned to every vulnerability that is cataloged by the CVE program[7]. Given the CVE ID, it's really easy to look for more information about the vulnerability, including proof of concepts or even fully functional exploits. It has an ID, which is the CVE ID itself and a technology, which is a reference to the Technology entity where the vulnerability is.

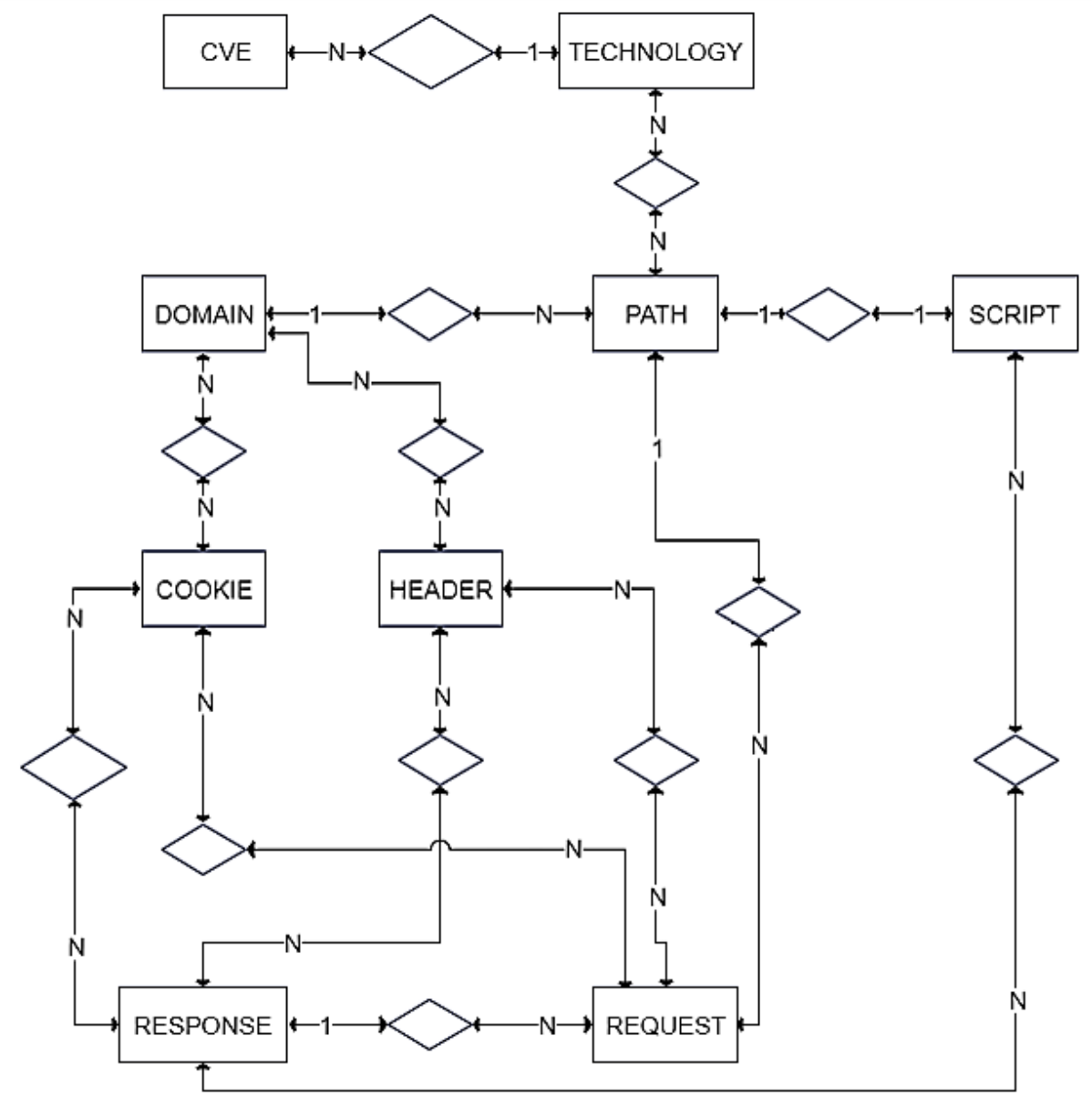


Figure 3.2: Simplified Entity Relational Diagram of the model representing the targets

[6] https://csrc.nist.gov/Projects/Security-Content-Automation-Protocol/Specifications/cpe
[7] https://www.cve.org/

When this Entity Relationship model is implemented in the SQL database, some extra tables are generated so that all relationships can be created. These tables are `domain_headers`, `domain_cookies`, `request_headers`, `request_cookies`, `response_headers`, `response_cookies`, `response_scripts` and `script_paths`. They are shown in figure 3.3 among the other tables in the database.

As previously mentioned, the database has other information than this model. It stores all out of scope domains explicitly specified by the operator. So for example, if a group of subdomains like `.examaple.com` is specified, the operator can also specify some subdomains to be out of scope, such as `admin.example.com`, so they are not scanned by the main script. The database also stores all potential vulnerabilities found by the main script. Each vulnerability has a type (i.e SQLi, XSS, subdomain takeover, etc.), a description of the vulnerability, the element affected by it and the command used to discover it (if any). This command is useful in the case of scanners such as sqlmap or dalfox. These vulnerabilities don't include CVEs, which already belong to the model. The database also stores some counters for the main script to be able to correctly iterate over all entities even after a restart.

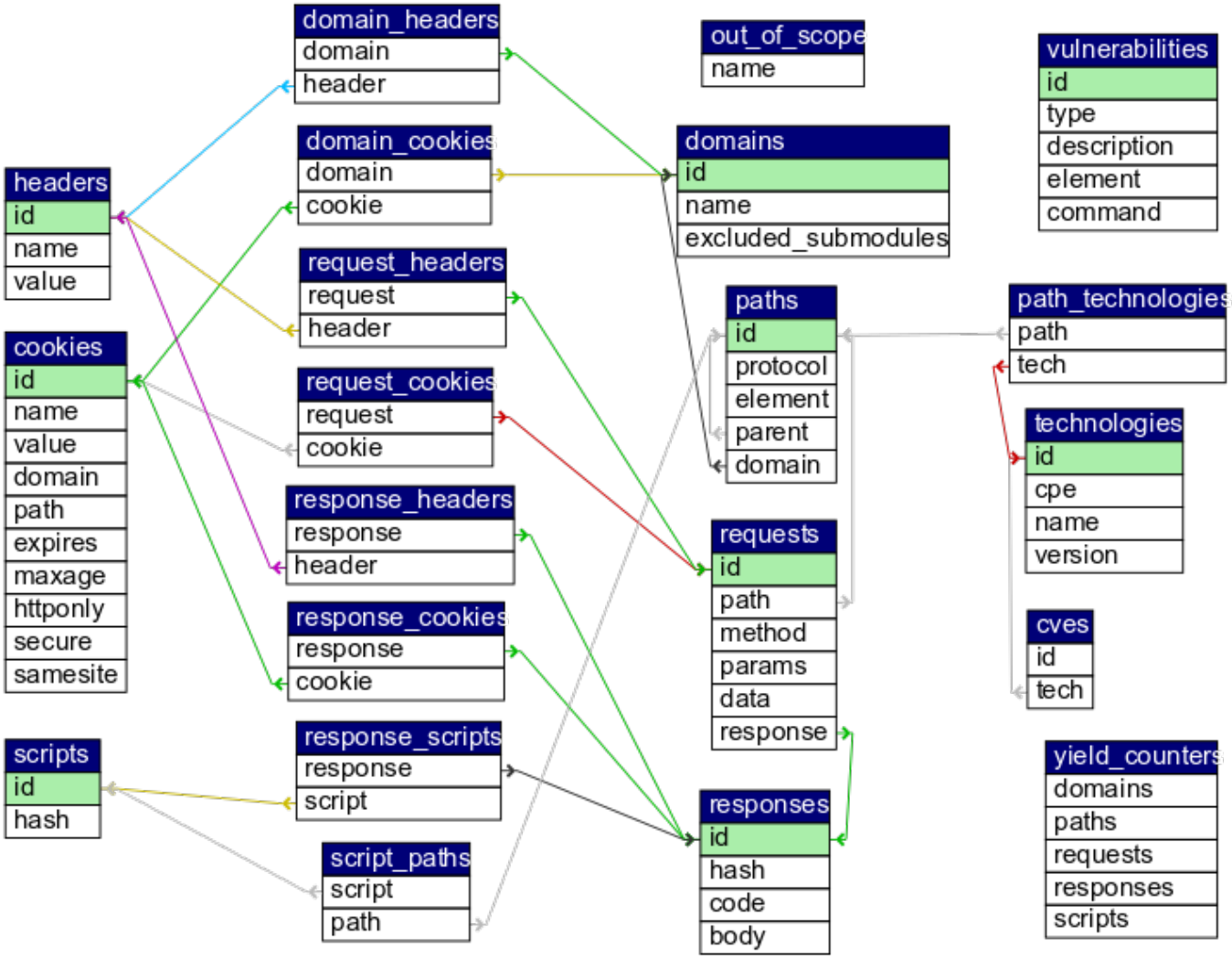


Figure 3.3: UML representation of the database

## 3.2 Main Script

The main script powering the system is built using Python. Python is a high-level, interpreted programming language that is very easy to write compared to other programming languages and, therefore, can speed up the development of the applications built with it[8]. Since this is a big project that had to be done in less than a year, this fact was a key element when choosing which language to use. On the other hand, Python is not as efficient as other programming languages, mainly due to this ease and flexibility, however, it's sufficient enough to perform the required tasks. Besides that, Python can very easily work with external programs, making it a perfect match for this project since it uses a lot of third-party tools that will be presented later.

This script is in charge of doing all the reconnaissance and vulnerability detection and communicating with the operator via Discord. It's composed of several files, which are shown in figure 3.4:

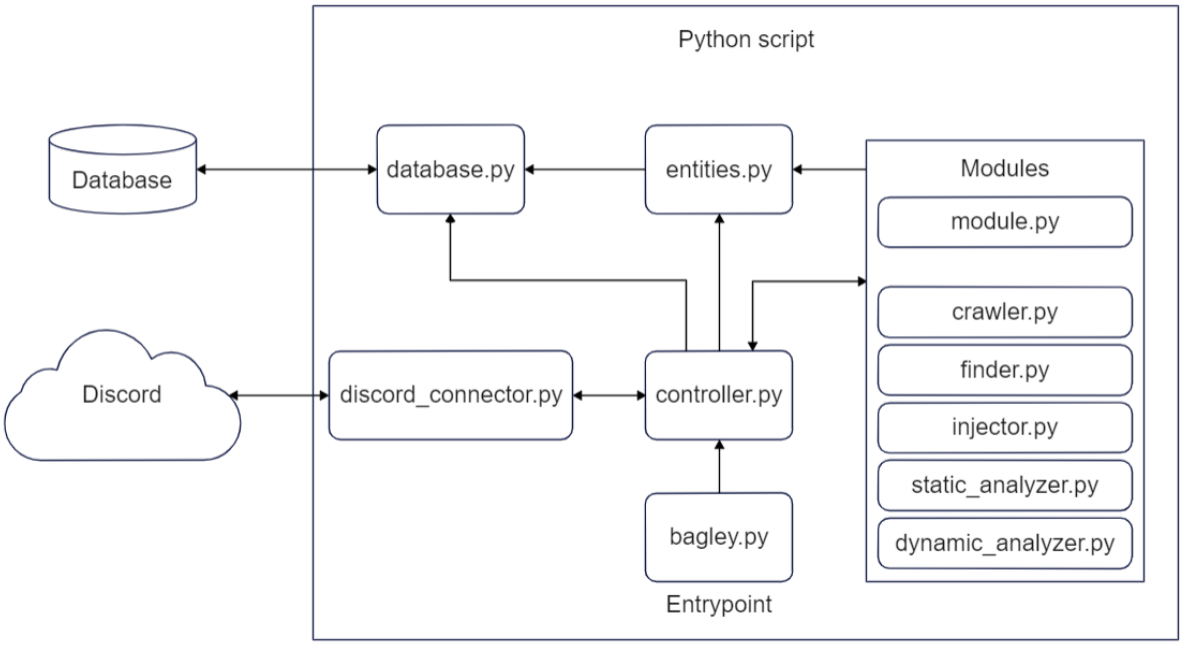


Figure 3.4: Main script architecture

The entry point of the script is `bagley.py`, which initializes the `Controller` in `controller.py`. This is the heart of the script. Its main purpose is managing the discord_connector in `discord_connector.py` and all available modules. Each of these modules runs in a different thread and performs a different technique or a set of similar techniques. The crawler is responsible for crawling a target, the finder for performing the content discovery, the injector for discovering injection vulnerabilities by brute-forcing, the static analyzer for analyzing obtained data locally and the dynamic analyzer for analyzing obtained data by querying external resources (or the targets themselves). All of them are defined in their respective files. These modules work with entities, defined in `entities.py`. These entities correspond to entities from the relational database, whose access is managed within `database.py`. The following subsections will explain all these files in more detail.

[8] `https://www.python.org/`

### 3.2.1 Database connection

The database connection is performed by the `DB` class, which is implemented in the `database.py` file. It uses the `mariadb` Python library, which offers a very powerful API for connecting to MariaDB databases.

In order to understand the `DB` class, it's important to highlight that the database can be accessed simultaneously by six threads at most (all five modules plus the controller, which can access the database in some situations described in later subsections), so the class must be able to manage all six connections independently. To do so, it follows a *Singleton* design pattern that takes into account which thread is calling the `DB` constructor to return the corresponding `DB` object, which contains the corresponding database connection. Each of these database connections is extracted from a `ConnectionPool`, which is an object from the `mariadb` library that allows previously initializing a pool of connections and then getting them individually[9]. It's much faster and safer than creating and managing the connections one by one.

In order to return a `DB` object with the correct connection to each thread, the `DB` class keeps a static record of those which have already called the constructor once and their corresponding `DB` object in a dictionary named `__instances`. This way, when a thread calls the constructor, it checks if that thread already has an object. If that's the case, it returns it, else, it creates a new `DB` object with a new connection to return it and save it in `__instances`. Figure 3.5 shows the method `__new__` of the `DB` class, which behaves as its constructor.

```python
class DB:
    __connectionPool = None
    __instances = {}

    def __new__(cls):

        # If it's the first time that DB() is called, connect to the database
        if DB.__connectionPool is None:
            DB.__connectionPool = mariadb.ConnectionPool(
                host=config.DB_HOST,
                user=config.DB_USER,
                database=config.DB_NAME,
                password=config.DB_PASSWORD,
                autocommit=True,
                pool_name="bagley",
                pool_size=6 )

        tid = threading.get_ident()

        # If the thred hasn't called DB() yet
        if DB.__instances.get(tid) is None:

            # Create an instance of DB
            instance = super(DB, cls).__new__(cls)

            # Assign a connection to this instance
            instance.__connection = DB.__connectionPool.get_connection()

            # Assign this instance to the thread calling DB()
            DB.__instances[tid] = instance

        return DB.__instances.get(tid)
```

Figure 3.5: Constructor for `DB` class in `database.py`

[9] `https://mariadb.com/docs/connect/programming-languages/python/connection-pools/`

`DB` class also defines some methods to query the database that act as different wrappers for the `exec` method of the `mariadb` library. These methods are the following:

- **`exec`**: Executes a query.
- **`exec_and_get_last_id`**: Executes a query and returns the generated ID value if the executed query was an `INSERT` statement on a table that had an `AUTO_INCREMENT` column.
- **`query_one`**: Executes a query and returns only the first row.
- **`query_all`**: Executes a query and returns all rows.
- **`query_string_like`**: Executes a query and returns all rows with the same format as the MariaDB or MySQL CLIs. Figure 3.6 shows an example of it.

```
+----+--------------------+---------------------+
| id | name               | excluded_submodules |
+----+--------------------+---------------------+
|  1 | .example.com       | NULL                |
|  2 | api.example.com    | NULL                |
|  3 | www.example.com    | NULL                |
|  4 | static.example.com | NULL                |
+----+--------------------+---------------------+
```

Figure 3.6: MariaDB CLI result example

### 3.2.2 Entities

Entities are Python classes that represent the entities from the relational database. They are all located in the `entities.py` file. All of them are defined in the model representing the targets, except for `Vulnerability` class, which although is in the database as a table, it's not in the model itself. Each class has its own set of attributes, which corresponds to those that the corresponding database entity has, and their own methods, depending on which actions can be performed with or on them.

Entity classes provide a layer of abstraction over the database, so that the rest of the script can operate with them without knowing how they are internally stored or represented. Its methods are queries to the database with some extra operations to accept a "high level" representation of the entity or to ensure the correctness and stability of the system. An example of the first case is the class `Path`, which has a static function to get a path by its ID called `getById` and another static function to get a path by URL simply called `get`. The second one accepts a URL, which can be seen as the "high level" representation of a path itself.

In total, there are more than seventy methods from all entities in `entities.py`, where the vast majority are not interesting at all. So, those methods whose behavior has some real value to understand some aspect of the system will be explained once they are used in their context, for example, in the Controller or in the Crawler module.

### 3.2.3 Modules

The modules of the script are the components that perform the recon and vulnerability detection, and are managed by the Controller. Each of them groups a set of similar sets of techniques and runs in a different thread. They all inherit from a parent class called `Module`, located in `module.py`. At the same time, this class inherits from the`Thread` class, from the`threading` library. It allows each module to implement a `run` function which will be executed in a new thread when running the method `start` on them. `Module` class defines some attributes and methods that are used by all modules to perform the following tasks:

- **Check dependencies**: In order to meet all proposed goals, there are a lot of different techniques that this script must be able to perform against a target. It would be impossible for a single programmer to build an application that can do them all in a correct and efficient way, so some third-party tools have been selected to do so. Each module uses a different set of these tools, so `Module` class has a mechanism to check that they are all present in the system.

  `Module` constructor has an attribute named `dependencies` that must be supplied by each individual module on creation, which is a list containing the paths to all its dependencies. When initializing the script, the method `checkDependencies()`, defined in `Module` class, is called for every module, raising an exception upon not finding a dependency. Figure 3.7 shows this method and figure 3.8 shows the Finder module calling its parent constructor (`Module` constructor) supplying its dependencies, which in this case are the tools `gobuster`, `subfinder` and `gau`.

```
def checkDependencies(self):
    for d in self.dependencies:
        if not shutil.which(d):
            msg = '%s not found in PATH' % d
            raise Exception(msg)
```

Figure 3.7: `checkDependencies()` method

```
class Finder(Module):
    def __init__(self, controller, stop, rps, active_modules, lock, crawler):
        super().__init__(["gobuster", "subfinder", "gau"], controller, stop, rps,
```

Figure 3.8: `Finder` class calling its parent constructor with its dependencies

- **Traffic control**: As already mentioned in the Methodology section, some Bug Bounty programs may require to limit the requests per second of the tools that the researcher is using, so, although there is always a bit of flexibility with this limit, a mechanism to control the traffic must be present.

  To do this, the `Module` class implements a method called `applyDelay`. This method gets the milliseconds that a module has to wait between requests based on the limit of requests per second and how many modules are currently active. Then, if the time elapsed from the last request made is less than those milliseconds, it computes the difference and sleeps for that amount of time. The class attribute `t`

indicates when was the last time that the module made a request, and it's updated with the method `updateLastRequest`. Figure 3.9 shows these methods. It's the responsibility of the module making the requests to apply the delay and update `t`. In order to know the number of active modules at any time, the external variable `active_modules` is used. This variable is located in the Controller and is supplied to each module on creation, so they all share the same one. To ensure the integrity of this variable, a lock is also used. Modules use `setActive` and `setInactive` methods to update this variable, as shown in figure 3.10.

The explained traffic control system is only required when is the script itself the one making the requests. When the requests are made by third-party tools, it is as easy as supplying the the delay that must be introduced between each request. It can be obtained with the method `getDelay`, which is the same method that `applyDelay` uses, as can be seen in figure 3.9. This is possible because all third-party tools used in this project include a `-delay` option or similar.

```
# Compute the delay that must be applied between each request based on which modules are active
def getDelay(self):
    if (self.active_modules is not None) and (self.rps is not None):
        return self.active_modules/self.rps
    else:
        return None

# Apply delay
def applyDelay(self):
    if (self.active_modules is not None) and (self.rps is not None):
        # If the time elpased between last request is less than the delay, apply the difference
        if (datetime.datetime.now() - self.t).total_seconds() < self.getDelay():
            t = self.getDelay() - (datetime.datetime.now() - self.t).total_seconds()
            if t > 0:
                time.sleep(t)

# Update delay timer (must be done after every call to applyDelay())
def updateLastRequest(self):
    self.t = datetime.datetime.now()
```

Figure 3.9: Methods managing delay

```
# Set module as innactive
def setInactive(self):
    if (self.active is not None) and (self.active_modules is not None) and (self.lock is not None) and (self.active):
        self.active = False
        self.lock.acquire()
        self.active_modules -= 1
        self.lock.release()

# Set module as active
def setActive(self):
    if (self.active is not None) and (self.active_modules is not None) and (self.lock is not None) and (not self.active):
        self.active = True
        self.lock.acquire()
        self.active_modules += 1
        self.lock.release()
```

Figure 3.10: Methods managing active status

- **Communicating with the Controller**: In order to communicate what is happening in each module, `Module` class has a set of methods that send the information to the Controller, so that it decides where to send it. To do this, an instance of the `Controller` is supplied upon module creation, and then used to send regular messages, error messages, important messages containing information about a potential vulnerability discovered and files. Figure 3.11 shows these methods.

```python
# Send message to controller
def send_msg(self, msg, channel):
    self.controller.send_msg(msg, channel)

# Send error message to controller
def send_error_msg(self, msg, channel):
    self.controller.send_error_msg(msg, channel)

# Send vulnerability message to controller
def send_vuln_msg(self, msg, channel):
    self.controller.send_vuln_msg(msg, channel)

# Send file to controller
def send_file(self, filename, channel):
    self.controller.send_file(filename, channel)
```

Figure 3.11: Methods that send information to the Controller

Once the parent class has been defined, these are the actual modules of the script:

### Crawler

The Crawler is the module in charge of crawling a target, which means navigating the web site to discover as much content as possible. It's the only module that doesn't use third-party applications, but also the one that inserts the most amount of data in the database: it's the main producer of information. Although there are already a lot of crawlers available, the vast majority are limited to just parsing HTML, and the ones that really navigate a website like a human are not free or are only accessible by the owners of the web applications. The main feature that these available crawlers miss is rendering JavaScript, which is really important nowadays since most modern web applications use it extensively. To do so, the page must be rendered in a browser. This can be easily done with Selenium WebDriver.

Selenium WebDriver is a very powerful tool commonly used for automating testing of web applications. It runs a web browser, such as Google Chrome or Firefox, that can be entirely controlled with an integrated API for languages such as Python, Java or C#[10]. In order to run, it needs that the browser which will be used is installed in the machine, as well as a special driver. Google Chrome, which is the browser used in this project, needs ChromeDriver[11], and its version must match Google Chrome's. In order to control the Selenium WebDriver, this project uses its API for Python[12]. Besides that, Selenium-Wire is also used. It's an open-source Python library that extends the functionality of

[10] `https://www.selenium.dev/documentation/webdriver/`
[11] `https://chromedriver.chromium.org/`
[12] `https://selenium-python.readthedocs.io/`

Selenium WebDriver giving access to the requests made by the browser, since it won't allow that *as is*[13].

On initialization, the `Crawler` class creates a Selenium WebDriver object associated with an instance of Headless Chrome. This is a way of running Chrome without actually displaying the GUI, so it's all done internally. This object will be referred to as the `driver`, since that is the name of the attribute that it's assigned to. This is the object that allows the Crawler to control the browser.

When the Crawler starts running, it retrieves URLs from its internal queue, where modules can add URLs with the method `addToQueue` so that they are crawled. In order for them to be queued, they must not be already crawled, they must not be already in the queue, they must be inside scope and they must have an allowed file extension. This can be checked by everyone trying to add a URL with `isQueueable()`, shown in figure **??**. On the other hand, if the queue is empty, the Crawler retrieves targets from the database using the method `yieldAll()` from `Domain` class, representing the Domain entities. This method yields all domains and groups of subdomains in the database as `Domain` objects, with their associated `Header` and `Cookie`. The Crawler checks if the domain uses `http` or `https` and builds the corresponding URL. If it uses both protocols, the Crawler starts with the one using `http` and adds the other to its queue. Once it has a valid URL, it checks that the domain is in scope and checks that the URL hasn't been crawled yet. If everything is correct, it calls `__crawl()`, which is the main method of the Crawler.

```
# If URL already exists in db or in queue, if it's not in scope or if file extension is blackilsted, it returns False
def isQueueable(self, url):
    if Path.check(url) or (url in self.queue) or (not Domain.checkScope(urlparse(url).netloc)) or (not Path.checkExtension(url)):
        return False

    return True
```

Figure 3.12: `isQueueablel()` method from the Crawler

Method `__crawl()` receives an URL, a method and optional data, which is the body of `POST` requests. First, it updates the cookies with the `__updateCookies()` method, which means getting the cookies associated to the domain of the URL and inserting them in the browser with `driver.add_cookie()`. This ensures that if the browser deletes all cookies because of a logout in a web application or something similar, they are inserted again so that those associated with the domain are always sent with the requests. Then, it updates the local storage of the browser for the given URL with the `__updateLocalStorage()` method[14]. The reason behind this is that some web applications save in there the session data, so it's very useful for the operator of the system to be able to insert its own. However, the `WebDriver` object doesn't have a method to modify the local storage. Instead, it has one to execute JavaScript named `driver.execute_script()`, which can be used to modify it. The data inserted is taken from an attribute of the `Crawler` that maps URLs to local storage data so it gets updated when those URLs are crawled. New local storage data can be added to this attribute with the `addToLocalStorage()` method.

[13] `https://github.com/wkeeling/selenium-wire`

[14] Browsers have different local storage for different locations, being a location the combination of protocol, domain and port

Once the browser is ready, the request is sent. If it's a `GET` request, `__crawl()` creates a `request_interceptor` that injects headers associated to the domain in the request. However, if it's not a `GET` request, the `driver` cannot send it, so, again, JavaScript is used instead. `__crawl()` calls a method called `__request` that sends requests with any method by a JavaScript code. In this case, headers associated with the domain are sent directly too. Figure 3.13 shows this method.

```
# Function to send requests using any HTTP method by using JavaScript in the browser
def __request(self, url, method, data, headers):
    current_requests = len(self.driver.requests)

    headers_dict = {}
    for header in headers:
        headers_dict[header.key] = header.value

    self.driver.execute_script("""
    function req(url, method, data, headers) {
        fetch(url, {
            method: method,
            headers: headers,
            body: data
        }).then((result)=>{return result;});
    }

    req(arguments[0], arguments[1], arguments[2], arguments[3]);
    """, url, method, data, headers_dict)

    # Wait until request is received by selenium or until it times out
    i = 0
    while (len(self.driver.requests) == current_requests) and (i < config.TIMEOUT):
        time.sleep(1)
        i += 1
```

Figure 3.13: `__request` method from the Crawler

If the request is correctly sent, `__crawl()` analyzes all requests sent by the browser. Requests made to resources containing script files are processed by `__processScript()`, where they are inserted in the database as Script entities. The first request sent, named *main request*, and its response are processed respectively by `__processRequest()` and `__processResponse()`, which insert them as a Request and Response entities. If the response code is `3XX`, meaning a redirection, `__crawl()` calls `__isCrawlable()` to check if the request to the URL where the redirection is pointing can be made. This method checks if the domain is in scope, if the extension is not blacklisted and if a similar request has not already been made, as shown in figure 3.14. Parameter `content_type` is an optional parameter for `POST` requests used to better parse the data sent with the body. If the URL can be crawled, `__crawl()` is recursively called to follow the redirection. If the response code is not `3XX`, the driver takes an screenshot of the rendered response with the `driver.save_screenshot()` method and the previously mentioned `send_file()`, defined in `Module`, is called so the controller delivers the screenshot to the operator. Then, the response is queued for further processing in a list named `responses2analyze`. All requests made as a result of the main request, named *dynamic requests*, are checked to know if they can be crawled with`__isCrawlable()` so that they can be added to `responses2analyze` for further processing.

```python
# Check if requests can be crawled based on the scope, the type of resource to be requested and
# if the request has been already made
def isCrawlable(self, url, method='GET', content_type=None, data=None):
    domain = urlparse(url).netloc

    if Domain.checkScope(domain) and Path.checkExtension(url) and not Request.check(url, method, content_type, data):
        return True

    return False
```

Figure 3.14: `isCrawlable()` method from the Crawler

All responses in `responses2analyze` are then analyzed using `__parseHTML()`. This method traverses all HTML entities in the response looking for URLs in forms, anchor tags, etc. When it finds one, it calls `__isCrawlable()` to check if the request can be made and, if that's the case, it calls `__crawl()` recursively. In the case of buttons, it uses `driver.execute_script()` to execute a JavaScript script that actually presses the button. Then it checks if requests have been made or if the URL has changed. In those cases, it also calls `__isCrawlable()` and `__crawl()` with the observed requests.

Figure 3.15 shows the flowchart for the entire `__crawl()` method.

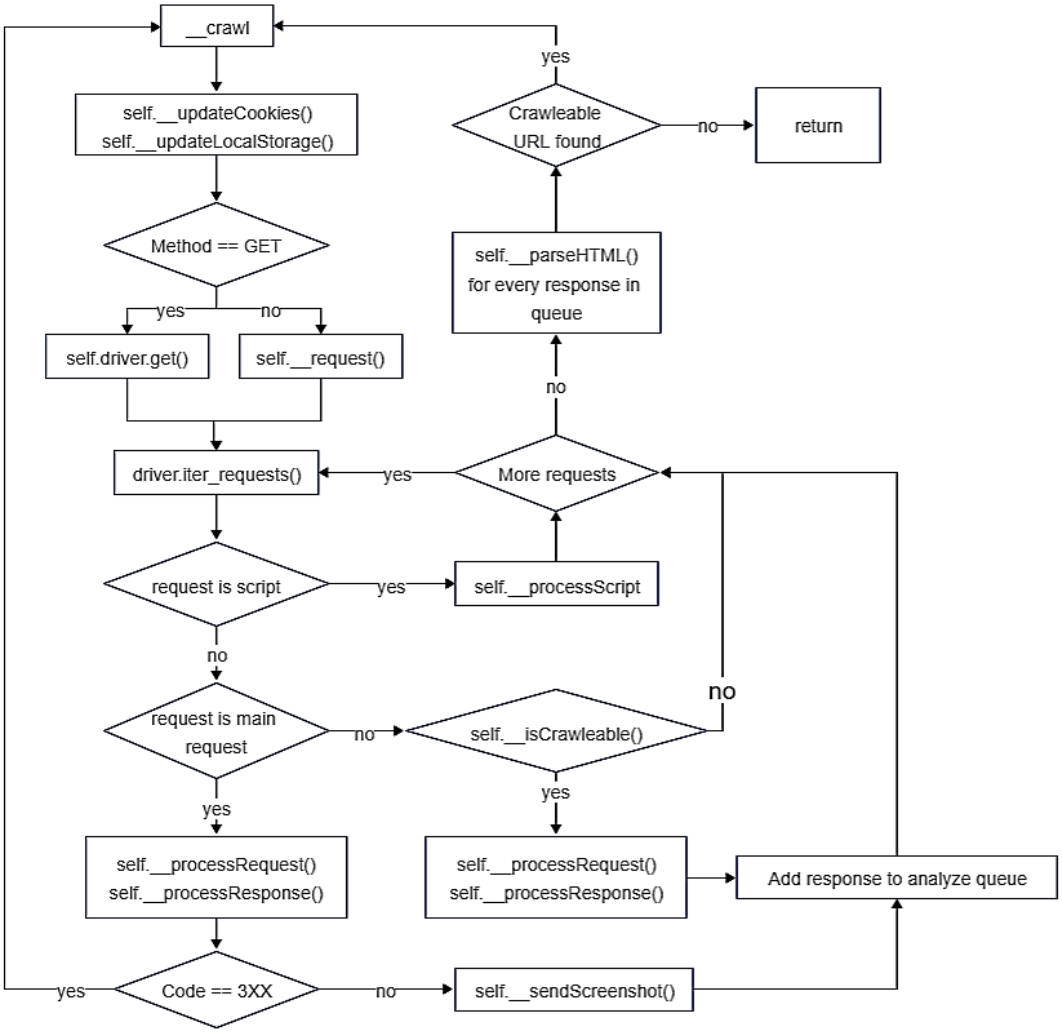


Figure 3.15: Flowchart for `__crawl()` method from the Crawler

### Finder

The Finder module is the one in charge of performing the other part of content-discovery. While the Crawler navigates the web applications, the Finder looks for new paths and new subdomains with alternative methods. These methods are brute-forcing and consulting online resources. This module has four submodules, which are the following:

- **Fuzz Subdomains**: Uses a tool called Gobuster to fuzz for subdomains given a `Domain` object representing a group of subdomains. Gobuster is an open-source tool used to brute-force URL paths, subdomains, virtual host names and Open Amazon S3 buckets[15], however, in this project it's only used for brute-forcing URL paths and subdomains. The main reason to use it over other tools such as *Dirbuster* or *ffuz* is that it's a very simple tool combining both subdomain and URL paths discovery, that only performs the brute-force (which is the desired behavior here) and does not do it recursively, so the researcher can exactly control what is brute-forced.
- **Find Subdomains**: Uses a tool called Subfinder to find subdomains given a `Domain` object representing a group of subdomains. Subfinder is an open-source tool used to discover valid subdomains from a domain by using passive online sources. It's very optimized and it's built for this purpose only, making it very compact[16]. In order to find the subdomains, it consults websites such as *Alienvault*, *Wayback Machine*, *GitHub* and many more.
- **Fuzz Paths**: Also uses Gobuster, but to fuzz for paths given a `Path` object representing a directory.
- **Find Paths**: Uses a tool called Gau to find paths given a `Domain` object. Gau implements the same concept as Subfinder to discover URL paths from a target. It's also an open-source tool and uses *Alienvault*, *Wayback Machine*, *Common Crawl* and *URLScan* to find them[17]. This tool is also a very known one in the community.

### Injector

The Injector module is the one in charge of finding injection vulnerabilities with brute-forcing. It has two submodules:

- **SQLi**: Uses Sqlmap to find SQL injections in valid requests which have either `POST` data or URL parameters. Sqlmap is probably one of the most popular penetration testing tools ever built. It's an open-source tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers[18]. It has a lot of options for any possible situation regarding SQL injections and many payloads to test with. It's for sure the most complete tool for exploiting this kind of vulnerability and its reliability has been largely proven

[15] `https://github.com/OJ/gobuster`
[16] `https://github.com/projectdiscovery/subfinder`
[17] `https://github.com/lc/gau`
[18] `https://github.com/sqlmapproject/sqlmap`

- **XSS**: Uses Dalfox to find XSS vulnerabilities in valid requests which have a response with HTML. DalFox is an open-source tool written in Go used for detecting and verifying XSS vulnerabilities. As secondary features, it can also detect other flaws such as SQLi or SSTI and perform grepping in order to find credentials or detect application errors[19]. This project will only use DalFox to detect XSS vulnerabilities, since the other functionalities are very faulty and do not work well. Although it is not a very compact tool due to it having other unused functionalities, it's the best option for XSS detection. It detects DOM based, reflected and stored XSS and rarely gives false positives, which is a very common failure among other XSS scanners such as XSStrike.

It is worth mentioning that there is a third module implemented which uses CRLFuzz to find CRLF injection vulnerabilities, however, it's not used yet because the application does not provide an option to control the traffic produced by it, and it generates too much. CRLFuzz is an open-source tool written in Go that is used to find CRLF injection vulnerabilities by brute-forcing the headers. It has a lot of options to customize the scan so that it's as efficient as possible depending on the context[20]. An issue has been opened in GitHub in case the developer wants to adds the traffic control feature

### Static Analyzer

The static analyzer module analyzes the gathered data without generating significant network traffic. It has three submodules:

- **Search Keys**: Parses all gathered responses and scripts looking for leaked credentials. To do so, it uses the `rure` Python library, which is a binding of the Rust's regex library[21]. It has a slightly different syntax than the regular regex in Python, but it increases the efficiency a lot. The list of regular expressions used to find the credentials is taken from the tool Dora[22].
- **Find Paths**: Uses Linkfinder to look for URLs in the gathered scripts, validating if the URL points to a valid resource. Linkfinder is a Python script used to discover endpoints and parameters inside JavaScript files with regular expressions[23]. Its use is very straightforward and it's quite known among the bug hunters community.
- **Analyze Script**: Uses CodeQL to analyze all gathered scripts looking for vulnerabilities such as XSS, Open Redirect or Prototype Pollution. CodeQL is an analysis engine that processes the code to be analyzed and treats it as data. It can be used, among other things, to find security vulnerabilities by running predefined queries against a database created from the audited code[24]. From the point of view of the security researcher, it can be considered as a tool to automate security-related static analysis. It's a unique tool created by GitHub that has a very big community behind it that creates custom queries for any possible vulnerability that could exist.

[19] `https://github.com/hahwul/dalfox`
[20] `https://github.com/dwisiswant0/crlfuzz`
[21] `https://pypi.org/project/rure/`
[22] `https://github.com/sdushantha/dora`
[23] `https://github.com/GerbenJavado/LinkFinder`
[24] `https://codeql.github.com/docs/codeql-overview/about-codeql/`

In this case, the system uses predefined queries taken from the CodeQL repository[25], which look for client-side security vulnerabilities. They are all defined in `src/lib/modules/config/bagley_codeql.qls`, and they look for: XSS (CWE-079), code injection (CWE-094), improper sanitanization of output (CWE-116), accepting messages from any origin via `postMessage` (CWE-201), interpreting hard-coded data as code (CWE-506), open redirect (CWE-601) and prototype pollution (CWE-915).

Besides that, there is another technique that can really enhance the use of CodeQL. Nowadays, more and more web developers choose to build their applications as SPAs, which means loading a single web document and updating the body of the site via JavaScript APIs[26]. Many of these SPAs use Webpack to bundle all scripts and resources that the application is using into a packed, minified, obfuscated file containing all necessary information for the website to work[27]. In order for the developers to debug the application easily, they add a JavaScript Sourcemap. This allows the browser to restore the original script and resources so the developer can actually see them instead of the minified file that was deployed in the application. This means that researchers auditing an application containing a Javascript Sourcemap can also retrieve the original scripts and resources to better understand the behavior, successfully bypassing the offuscation. Unwebpack_sourcemap is a Python script that parses JavaScript Sourcemaps and retrieves all original scripts and resources to a folder[28]. There exists some other options to unpack sourcemaps, however, this one was written by the author of the original post detailing this technique[5].

If a sourcemap is not found for the analyzed script, it's beautified with `jsbeautify`, so that at least is easier to read, although if the original script was obfuscated, the resulting one will be too.

### Dynamic Analyzer

The dynamic analyzer performs analysis on gathered data producing a significant amount of network traffic. It has three submodules:

- **CVE Finder**: Finds which are the technologies running in the web application with Wappalyzer. This is an open-source tool that identifies the technologies used on websites such as CMS, web frameworks, ecommerce platforms, JavaScript libraries... Although it's usually used as a browser extension, there is also a CLI providing the same functionalities, which is what is used in this project[29]. This tool is a standard in the web security field and it's used by many researchers as it is very precise.

[25] `https://github.com/github/codeql/tree/main/javascript/ql/src/Security`
[26] `https://developer.mozilla.org/en-US/docs/Glossary/SPA`
[27] `https://webpack.js.org/`
[28] `https://github.com/rarecoil/unwebpack-sourcemap`
[29] `https://github.com/wappalyzer/wappalyzer`

If Wappalyzer is also able to find the version of any technology, then the NIST vulnerability API[30] is consulted to check if there is any vulnerability associated to that technology. This can be done thanks to the CPE of the technology, which is included in the Wappalyzer report and accepted as a search parameter by the NIST API.

- **Subdomain Takeover**: Checks if a subdomain is vulnerable to subdomain takeover with Subjack. Subjack is an open-source tool written in Go used to identify if a subdomain can be hijacked, that is, if it's vulnerable to subdomain takeover[31]. It's the most used tool for testing this vulnerability and is fast and reliable.
- **Bypass 403**: Tries to bypass `403` HTTP response codes, meaning forbidden access, by tampering with headers and HTTP methods.

### 3.2.4 Discord connection

The Discord connection is managed by `Discord_Connector` class, located in `discord_connector.py` file. It uses the `discord` Python library control the Discord bot, which has access to the server where it can communicate with the operator. Upon construction, this class creates the `discord.Client` object, assigns it to a class attribute named `bot` and defines the handlers for the events. There events handler are shown in figure 3.16 and are the following:

- **`on_ready()`**: Handler function that executes with the `ready` event, which is fired when the `discord.Client` establishes the connection with the server. It displays a welcome message.
- **`on_message()`**: Handler function that executes with the `message` event, which is fired when a message is received in any channel of the server. It checks that it hasn't been sent by the `discord.Client` itself and then parses each line as a command. The `message` event is also fired when files are sent, so this function can also read them. This allows sending very big commands in text files, such as those containing cookies or many targets. Otherwise it wouldn't be possible since Discord has a maximum of characters that can be sent in a regular message.
- **`on_bagley_msg()`**: Handler function that executes with the `bagley_msg` event. This is a custom event used so that the rest of the script can send messages to the server with it. Since the `discord` library uses asynchronous code but the rest of the script doesn't, other components of the script cannot just use its functions. Because of this, events are the perfect way of communication. Any component of the script can dispatch a `bagley_msg` event which will be asynchronously handled by this function. This event is received with a message and a channel, so this function sends that message to the selected channel of the server that the `discord.Client` is connected to.
- **`on_bagley_file()`**: Handler function that executes with the `bagley_file` event. This is another custom event with the same purpose as the one before, but for sending files instead of messages.

[30] `https://nvd.nist.gov/developers/vulnerabilities`
[31] `https://github.com/haccer/subjack`

In order for the other components of the script to be able to dispatch the custom events, the function `discord.Client.dispatch()` is used. It does not appear on the documentation of the library, however, there is enough information online to be able to use it.

In order to parse the commands, the `Discord_Connector` class uses a class named `CommandParser`. This class implements a *Command* design pattern, which makes the addition of new commands really easy and fast. It has a list of objects whose classes implement each command, which inherit from a parent class named `Command`. It defines a method `checkArgs()`, that checks if the arguments supplied are valid, and a method `run()`, that actually executes the task to be performed. It also defines some arguments to be supplied by each command, which are `name`, `help_msg` and `usage_msg`. Figure 3.17 shows these methods and attributes in `SetRPSCommand`. `CommandParser` has a main method `parse()` that receives the text to be parsed, as shown by figure 3.18. It iterates over the commands list to check if the supplied command and supplied match with anyone. If that's the case, the command is executed. Upon construction, `CommandParser` creates all command objects by providing them an instance of the `Controller` and an instance of the `Discord_Connector`, both supplied by the `Discord_Connector` itself, which is the one creating the `CommandParser` object.

```python
@self.bot.event
async def on_ready():
    logging.info("Connected to Discord bot")
    for c in self.bot.get_all_channels():
        if c.name == "terminal":
            await c.send("`Hello`")

@self.bot.event
async def on_message(message):
    if message.author.id != self.bot.user.id:
        # If message is a regular message
        if len(message.attachments) == 0:
            for line in message.content.split('\n'):
                await cp.parse(line)
        # If message is a file
        else:
            try:
                for attachment in message.attachments:
                    # Get text file directly as r
                    async with aiohttp.ClientSession().get(attachment.url) as r:
                        txt = await r.text()
                        if txt:
                            for line in txt.split('\n'):
                                await cp.parse(line)
            except:
                return

@self.bot.event
async def on_bagley_msg(msg, channel):
    await self.send_msg(msg, channel)

@self.bot.event
async def on_bagley_file(filename, channel):
    await self.send_file(filename, channel)
```

Figure 3.16: Event handlers from `Discord_Connector` class in `discord_connector.py`

```python
class SetRPSCommand(Command):
    def __init__(self, controller, discord_connector):
        super().__init__(controller, discord_connector, "setRPS", "Set requests per second", "Usage: setrps <RPS>")

    def checkArgs(self, args):
        if (len(args) != 2):
            return False

        try:
            args[1] = int(args[1])
            return True
        except:
            return False

    async def run(self, args):
        rps = int(args[1])
        self.controller.setRPS(rps)
        await self.discord_connector.send_msg("Requests per second set to %d" % rps, "terminal")
```

Figure 3.17: Example of `setRPSCommand` implementing `Command`

```python
async def parse(self, line):
    args = line.lower().split()
    if len(args) == 0:
        await self.discord_connector.send_msg('Use "help" to see the available commands', "terminal")

    elif args[0] == 'help':
        help_msg = "Available commands:\n\n"
        help_msg += "help        Print this message\n"
        for c in self.commands:
            help_msg += c.name.ljust(12) + c.help_msg + "\n"
        await self.discord_connector.send_msg(help_msg, "terminal")

    else:
        try:
            for c in self.commands:
                if args[0] == c.name.lower():
                    if c.checkArgs(args):
                        await c.run(args)
                    else:
                        await self.discord_connector.send_msg(c.usage_msg, "terminal")
                    return

            await self.discord_connector.send_msg('Cannot understand "%s". Type "help".' % line, "terminal")
        except:
            await self.discord_connector.send_msg(traceback.format_exc(), "terminal")
```

Figure 3.18: `parse()` method from `CommandParser` in `discord_connector.py`

`Discord_Connection` class has a method `init()` to initialize the `discord.Client` with a Discord token. This is done by the Controller after creating the `Discord Connection` object.

### 3.2.5 Controller

The Controller is the component that puts everything together. It's implemented by a class also named `Controller`. The entry point `bagley.py` creates the `Controller` object by providing a logger object so that all messages are logged. Upon construction, it creates all attributes needed to perform the traffic control among the modules, creates them, checks their dependencies, creates the `Discord_Connector` supplying an instance of itself and initializes it with a Discord Token. Figure 3.19 shows the constructor of `Controller` and figure 3.20 shows the sequence diagram for the whole script initialization.

`Controller` class has methods to manage the modules, to manage some entities and directly querying the database, all of them used by the commands to implement their functionalities so that the operator can manage the system. On the other hand, the `Controller` also has methods to send messages and files through the `Discord_Connection`, which are used by modules to communicate with the operator.

```
class Controller:
    def __init__(self, logger):
        self.stopThread = threading.Event()

        self.rps = config.REQ_PER_SEC
        self.active_modules = 0
        self.lock = threading.Lock()
        self.logger = logger

        self.initModules()

        # Check dependencies for the modules
        for m in self.modules.values():
            m.checkDependencies()

        self.discord_connector = lib.discord_connector.Discord_Connector(self)
        self.discord_connector.init(config.DISCORD_TOKEN)

    def initModules(self):
        crawler = lib.modules.Crawler(self, self.stopThread, self.rps, self.active_modules, self.lock)
        finder = lib.modules.Finder(self, self.stopThread, self.rps, self.active_modules, self.lock, crawler)
        injector = lib.modules.Injector(self, self.stopThread, self.rps, self.active_modules, self.lock)
        dynamic_analyzer = lib.modules.Dynamic_Analyzer(self, self.stopThread, self.rps, self.active_modules, self.lock)
        static_analyzer = lib.modules.Static_Analyzer(self, self.stopThread, crawler)

        self.modules = {
            "crawler": crawler,
            "finder": finder,
            "injector": injector,
            "dynamic_analyzer": dynamic_analyzer,
            "static_analyzer": static_analyzer
        }
```

Figure 3.19: `Controller` constructor

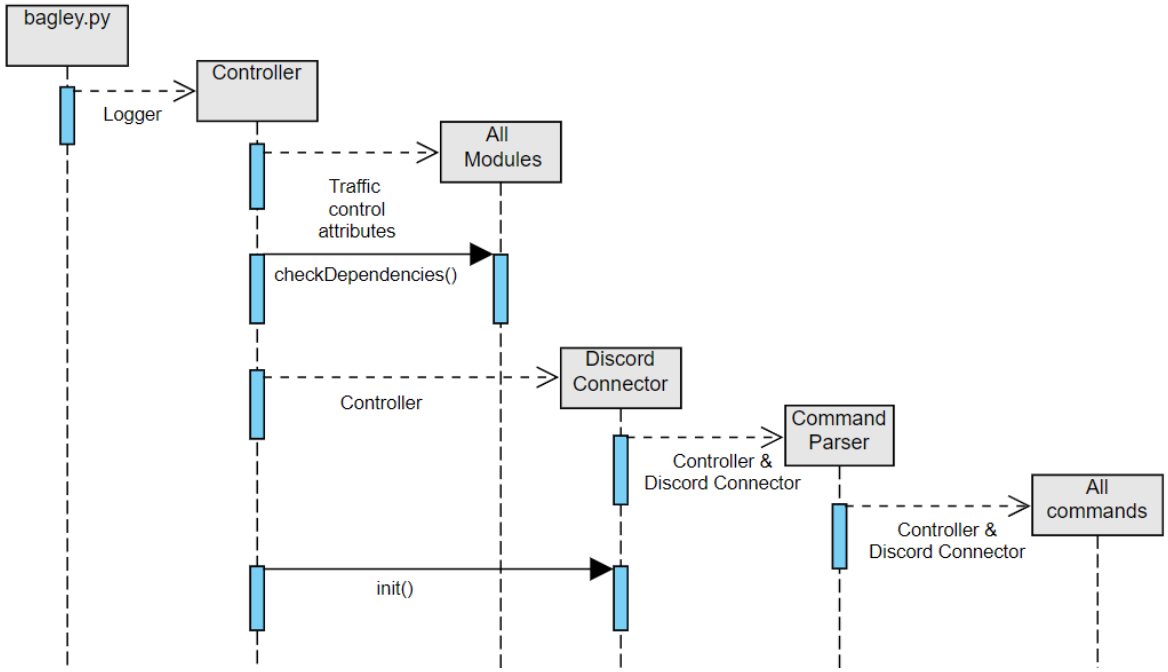


Figure 3.20: Sequence diagram for the system initialization

## 3.3 Deployment

To deploy such a complex system with such different technologies, Docker Engine is used. It is an open-source containerization technology used to deploy applications. It has a CLI client named `docker`[32] that is intuitive and easy to use. Docker Engine can easily automate the process of installing and configuring all the tools that this system needs, just by writing the commands to be executed on a `Dockerfile`. It can also quickly deploy existing applications that are uploaded to Docker Hub just with one single command[33]. This is the case for MariaDB server[34].

The first idea was to deploy everything together in a single container. However, the configuration of the MariaDB server from the `Dockerfile` can be really tedious, and having the main script and the database in separate containers allows the operator to rebuild the script container in case of maintenance without having to rebuild the database container, which has been really nice. So in order to manage both containers efficiently, Docker Compose is used. It is a tool for defining and running multi-container applications described in a YAML file[35]. In short, given a `Dockerfile` and a `docker-compose.yml`, the whole system can be deployed using `docker-compose up`.

Figure 3.21 shows the `docker-compose up` file used to deploy both containers. The MariaDB server is initialized with some environment variables defining the password for the `root` user, the database name, the database user and its password. The SQL script defining the database structure is copied so that it's executed on boot. Its port 3360 is exposed so that it can be accessed. On the other hand, the container running the actual script,

[32] `https://docs.docker.com/engine/`
[33] `https://hub.docker.com/search?q=`
[34] `https://hub.docker.com/_/mariadb/`
[35] `https://docs.docker.com/compose/`

named `bagley`, is also initialized with some environment variables regarding the database access and the Discord token. Its network is configured not to be isolated from the host, so that it can access the Internet freely in order to perform all the scans. Besides that, the given shared memory is 1 GB, since the browser inside Selenium needs a lot.

```yaml
version: '3.9'
services:
  mariadb:
    image: mariadb
    environment:
      - MARIADB_ROOT_PASSWORD=root_password
      - MARIADB_DATABASE=bagley
      - MARIADB_USER=bagley
      - MARIADB_PASSWORD=db_password
    volumes:
      - ./sql/bagley.sql:/docker-entrypoint-initdb.d/bagley.sql
    ports:
      - 3306:3306

  bagley:
    build: .
    environment:
      - DB_HOST=127.0.0.1
      - DB_NAME=bagley
      - DB_USER=bagley
      - DB_PASSWORD=db_password
      - DISCORD_TOKEN=discord_token
    cap_add:
      - NET_ADMIN
    depends_on:
      - mariadb
    network_mode: "host"
    shm_size: '1gb'
```

Figure 3.21: `docker-compose.yml` file

## 3.4 Virtual Private Server

The Virtual Private Server chosen to deploy the system is Linode. Linode is a company offering cloud-hosting services, that allows running a VPS with any Linux distribution at a very low price[36]. The user can connect to the server via SSH without any VPN needed. The selected plan for this project consists of a server running Ubuntu 20.04, with 2 CPUs, 4GB of RAM, 80GB of storage and an allowed network transfer of 4TB per month at 40Gbps in and 4Gbps out, with a price of $30 per month. What's more, it's very easy to find coupons that give $100 at registration, meaning three months for free. Linode was chosen among its other competitors because it's a very cheap solution while being very reliable and easy to use.

[36] `https://www.linode.com/`

# Chapter 4

# Use Case

As already commented during this document, this system is intended to be used in Bug Bounty environments. Its main purpose is to assist the bug hunter so that he or she can focus on other activities, such as finding other vulnerabilities that require more effort to be discovered and/or exploited. Thus, this is oriented to researchers that already have experience with Bug Bounties, since it still requires that the operator chooses a target and reports the vulnerabilities found. This chapter will explain which is the intended use case for this system.

When choosing a target to be scanned with this tool, there are a couple of aspects about the chosen Bug Bounty program that should be considered in order to properly set up the system. First, the program must allow researchers to use automated tools. If that is the case, some programs specify the limit of requests per second that these tools must comply with, so the tool must be provided with this limit. Otherwise, they may prevent the system from accessing their servers. Then, the researcher must check if the program specifies a Header or a Cookie that all requests made to the servers must contain, such as `X-Bug-Bounty: <username>`. This is so that the company engineers can differentiate legitimate researchers from actual malicious actors and identify them. If they do provide one of these headers or cookies, they must also be supplied to the tool.

Depending on the number of targets, if they are domains or groups of subdomains, if they are single web applications or multiple ones, if they have or don't have many functionalities exposed to unauthenticated users, etc. the operator must decide if it's worth supplying the system with the elements to be authenticated, taking into account the extra research that this suppose. If the operator chooses to do that, he or she must manually analyze the authentication mechanism present in the web application to check if it's based on headers, cookies or even local storage variables, and provide the corresponding element so that the system is authenticated.

Then, the system is ready to start. Once it's running, the operator can just ignore it and check the system only when a notification is sent, such as when a vulnerability is found or an error occurs. The system can also be checked periodically to see the discovered content, look at the screenshots of the crawled pages, check the technologies that everything is using, etc. This information can be used by the researcher as a starting point to look for other vulnerabilities that the system may not be considering, so that it acts as an assistant in the process.

If a vulnerability is found, the researcher must analyze it to check that it's not a false positive or that it can be exploited. If everything is correct and the vulnerability is indeed a valid one, it must be reported to the Bug Bounty program with as much detail as possible. This process can be easier with the help of the system, since everything is logged in the Discord server and in its internal logs, so that the researcher doesn't miss anything.

Given that the VPS costs $30 a month and that the average bounty for low vulnerabilities is $150[1], it's enough for the system to find a low impact vulnerability each 4 months so that the operator doesn't lose any money. Besides that, some vulnerabilities such as SQL injections or reflected XSSs (both detectable by this system) are usually considered critical ones, whose average bounty is $3000. Programs such as those which have been launched recently, private ones or those with new targets or new functionalities, usually have a lot of low-hanging vulnerabilities. If the operator can spot these programs and set up the system accordingly, it can result in quite a profitable project.

[1] https://www.hackerone.com/resources/reporting/hacker-powered-security-report-industry-insights-21

# Chapter 5

# Conclusions and Future Work

After many months of development, and a couple of weeks of serious usage, the main conclusion reached in this project is that automation is not for every researcher doing Bug Bounties. Building a solid system that is suitable for the workflow of the developer requires quite a lot of time, since it's a very personal process that nobody can teach. Then, the researcher must learn which are the Bug Bounty programs in which the system works better. For example, a system that looks for subdomains that are vulnerable to subdomain takeover, such as this one, works better if the program contains a group of domains as a target, instead of only containing single domains. Some bug hunters are not willing to spend the time needed for all this, if they do, they develop a simple one based on Bash with two or three techniques.

Besides, it's a fact that critical and interesting vulnerabilities are found manually, since the process of discovering one of these cannot be automated. The most interesting bugs usually require that the researcher thinks out of the box and follows a never traversed path to reach there, so it's impossible that a tool can reproduce that process. Some researchers prefer focusing only on these kind of vulnerabilities, since they are really rewarding and may give the hunter some recognition among the community, which is sometimes even more valuable than an economic reward.

In the couple of weeks that this system has been running against targets in HackerOne, it has discovered four vulnerabilities, however, none of them were successfully exploited or rewarded. From these four, two of them were subdomain takeovers that couldn't be exploited due to the hosting plan that the company had contracted. Another one was an exposed Google API key, which is no longer considered a vulnerability due to it only providing access to cheap Google services whose access can be easily revoked. The last one was a Prototype Pollution in a client-side script which wasn't exploitable. Unfortunately, none of these potential vulnerabilities nor the programs they were found in can be disclosed.

However, the relation between the time that the system has been running and the potential vulnerabilities that were found gives very promising results. Any of these bugs could have been rewarded more than $200, which would be enough to pay six months of Linode hosting, just from using the system for two weeks. Up to this point, the author considers the project to be a success, and it's very likely that the system finds a valid vulnerability in the near future.

Next steps will be to continue using the system, adding new modules that can execute more techniques against targets and optimizing those that are already implemented. In the best case scenario, this system will be running for a long time, more and more efficient and powerful, providing support for the vulnerability research conducted by the author.

# List of Figures

# Bibliography

[1] P. Yaworski, *Real-World Bug Hunting: A Field Guide to Web Hacking.* No Starch Press, 2019.

[2] J. G. H.-S. J. T. Y. Wu, *Testing and Quality Assurance for Component-based Software.* Artech House, 2003.

[3] A. S. Foundation. Cve-2021-44228. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-44228

[4] O. M. Álvaro Muñoz, "A journey from jndi/ldap manipulation to remote code execution dream land." [Online]. Available: https://www.youtube.com/watch?v=Y8a5nB-vy78

[5] Rarecoil. Spa source code recovery by un-webpacking source maps. [Online]. Available: https://medium.com/@rarecoil/spa-source-code-recovery-by-un-webpacking-source-maps-ef830fc2351d

# Appendix A

# Operator Manual

## A.1 Deployment

It's required to have `docker` and `docker-compose` tools installed so that the system can be deployed. To download all necessary files, clone the GitHub repository located in `https://github.com/hacefresko/Bagley`. Then, open `docker-compose.yml` and write the Discord token of the bot to be used. File `src/config.py` contains application parameters such as the initial requests per second or the timeout of the internal browser along with filenames and other configuration parameters. Run `docker-compose up` to deploy the system.

## A.2 Configuring Discord

In order to get this system working with a Discord server, it must have a series of channels:

- A channel to interact with the send commands to the system, named `terminal`.
- A channel for notifying vulnerabilities, named `vulnerabilities`.
- A channel for notifying errors, named `errors`.
- A channel for each module, named `crawler`, `finder`, `injector`, `static-analyzer` and `dynamic-analyzer`.

Each channel can be muted depending on the user's preferences. Figure A.1 shows an example of this configuration.

Then, a bot needs to be created and added to the server. To create a bot, go to `https://discord.com/developers/applications` and create an application. Figure A.2 shows an example of a Discord application in the Developer Portal. Inside that application, add a new bot. Figure A.3 shows an example of a bot. Once it is successfully added, the Discord token to control it will be available. This token can only bee seen once, so if it gets lost, a new one will have to be created. This is the token that must be written in `docker-compose.yml`.

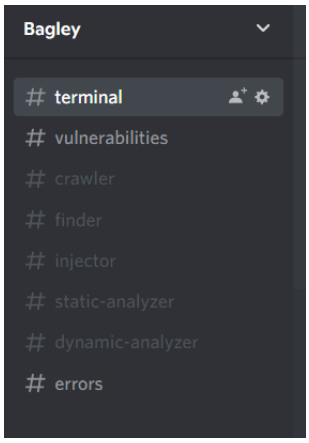


Figure A.1: Example of Discord channels configuration

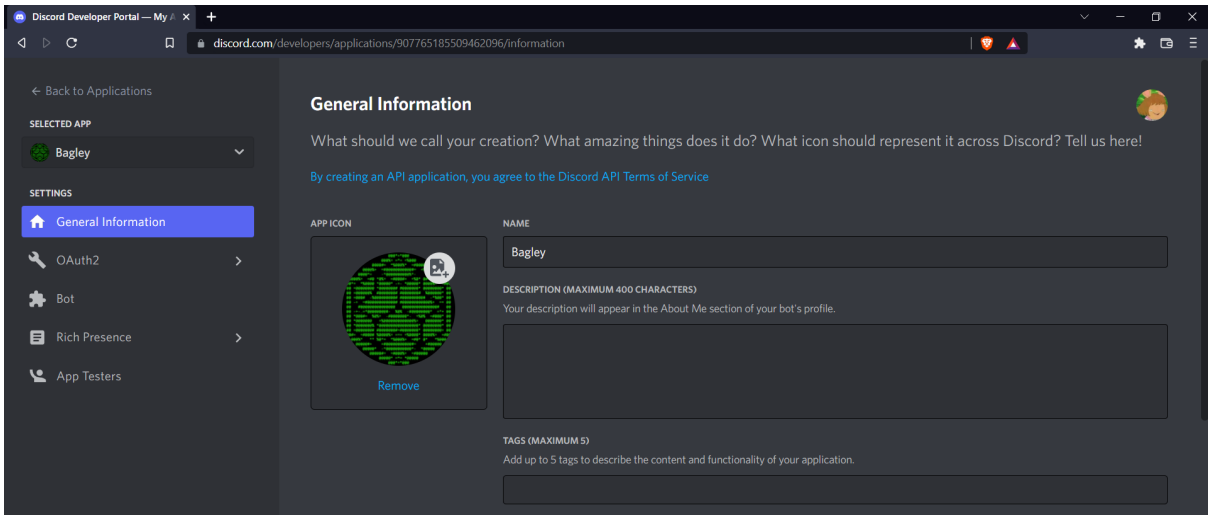


Figure A.2: Example of Discord application in the Developer portal

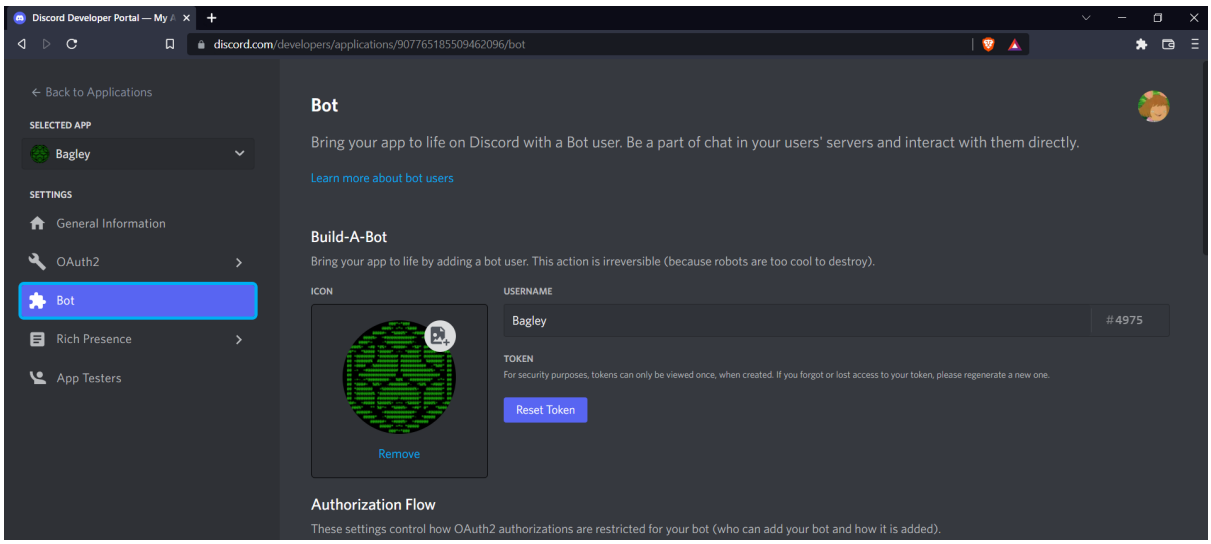


Figure A.3: Example of Discord bot in the Developer portal

## A.3 Usage

Once the system is deployed and ready, the bot will say *Hello* in the `terminal` channel of the server. The operator can send the command `help` to get a list of available commands, as shown in figure A.4.


Figure A.4: Help message

Although these commands are pretty self explanatory, such as `start` or `stop`, a couple of them require some explanation . First, `add` command offers a help message, since it accepts many options. This help message is shown in figure A.5. These options are parameters to be inserted with the target. For example, in order to add the group of subdomains `.example.com`, excluding `test.example.com` so that it's not scanned (out of scope), adding `https://www.example.com/example?e=1337` to queue so that it's crawled first, and inserting `{"session": 1337}` in the browser local storage for `https://www.example.com`, the command would be: `add .example.com {"excluded": ["test.example.com"], "queue": ["https://www.example.com/example?e=1337"], "localStorage": [{"key": "session", "value": "1337", "url": "https://www.example.com"}]}`

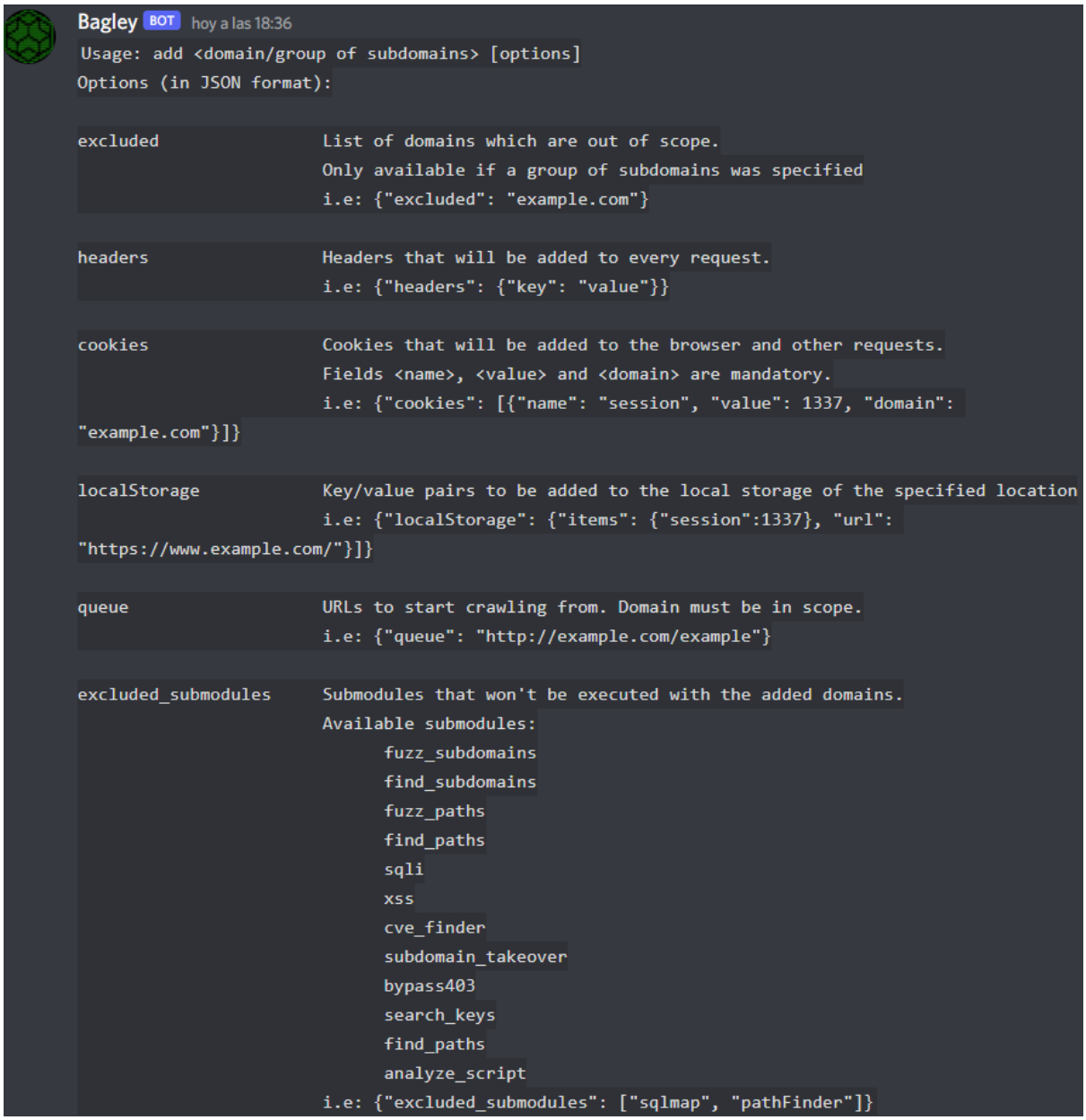


Figure A.5: Command `add` help message

Another command whose behavior is worth mentioning is `getScript`. This command receives the ID of a script and prints all paths where the script has been found (its locations) and all responses in which the script is used. It also sends a file containing the script itself. If it was already processed by the static analyzer and it extracted the original files from the sourcemap, it also sends a `zip` file containing them, as shown in figure **??**.

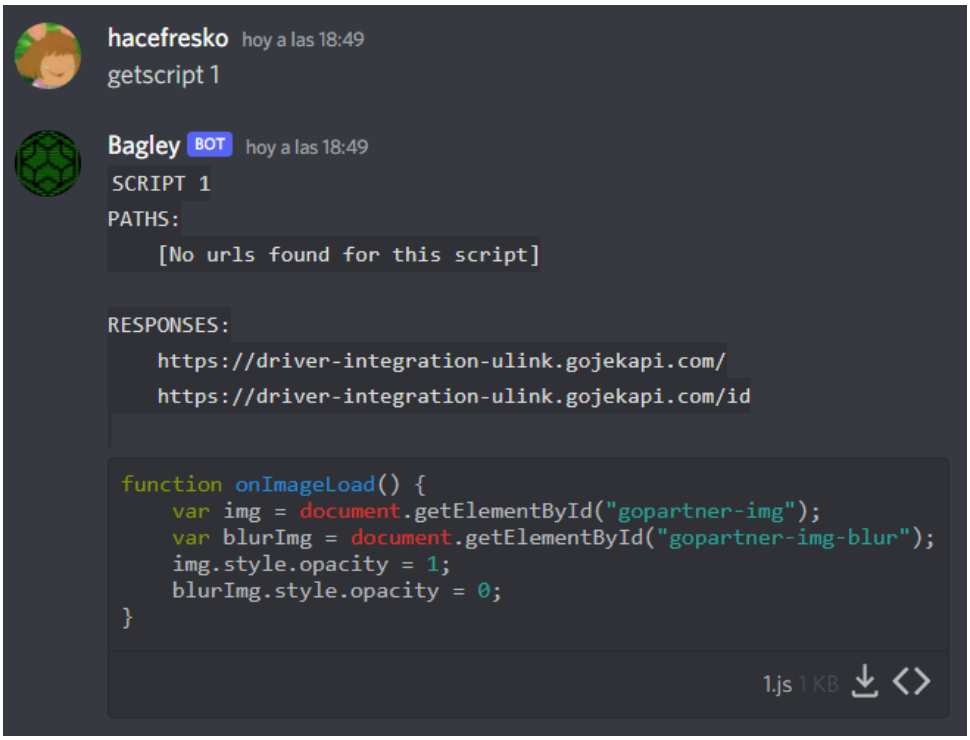


Figure A.6: Command `getScript`

Once the `start` command is used, the system starts scanning the targets. Figure A.7 shows the addition of a target from a public program in HackerOne, and the start of the system. At this point, the system starts producing results, which are shown in the different channels. Figures A.8, A.9, A.10, A.11 and A.12 show examples of these messages.

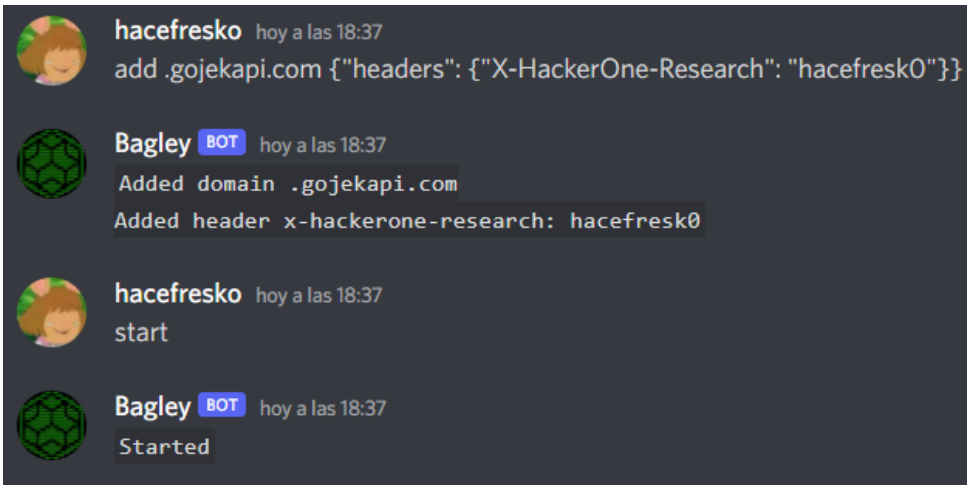


Figure A.7: Addition of target and start of the system


Figure A.8: Example of `crawler` channel messages

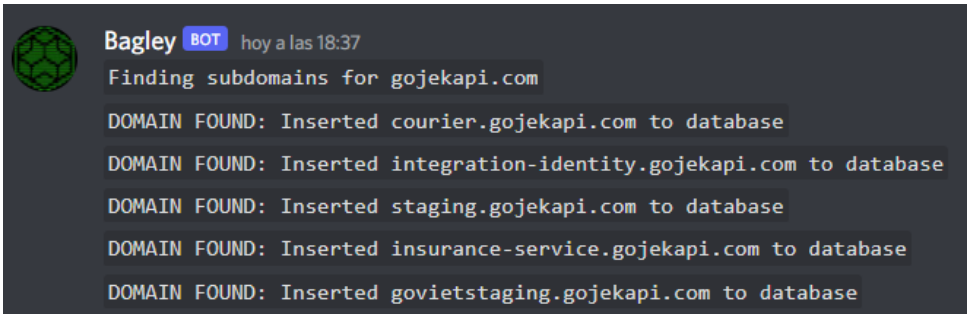


Figure A.9: Example of `finder` channel messages


Figure A.10: Example of `injector` channel messages


Figure A.11: Example of `static-analyzer` channel messages

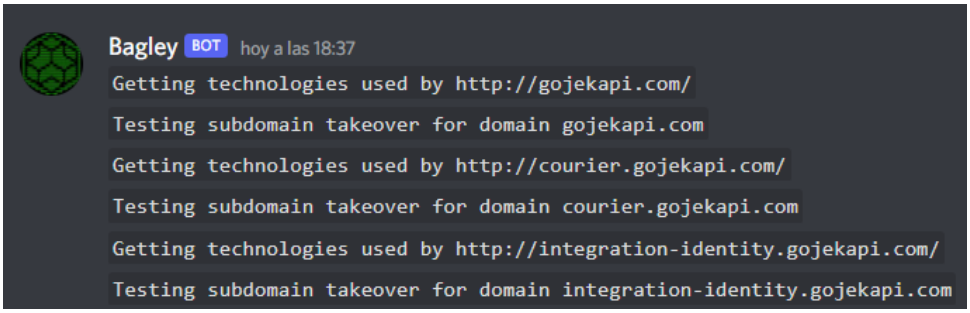


Figure A.12: Example of `dynamic-analyzer` channel messages

When a vulnerability is found, the message is sent to the channel corresponding to the module that discovered it and to the `vulnerabilities` channel, as shown if figure A.13.

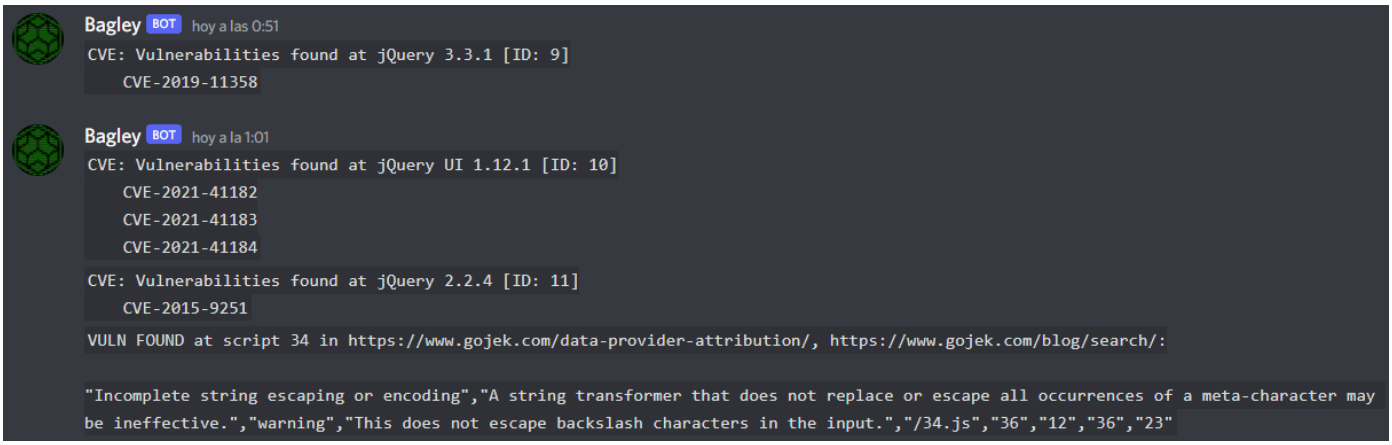


Figure A.13: Example of `vulnerabilities` channel messages

Víctor Fresco Perales
2022
Ult. actualización May 20, 2022
LaTeX lic. LPPL & powered by TEFLON CC-ZERO