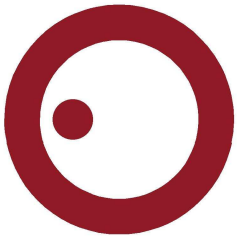


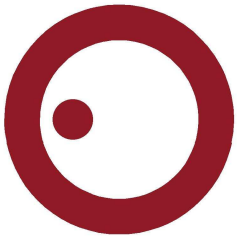
escola superior de tecnologia e gestão
instituto politécnico de leiria

Instituto Politécnico de Leiria
Escola Superior de Tecnologia e Gestão
Departamento de Engenharia Informática
Mestrado em Cibersegurança e Informática Forense

AUTOMATED, SCHEDULED AND CI/CD WEB INJECTION

MYKYTA ZHYGULSKYY

Leiria, Fevereiro de 2021



escola superior de tecnologia e gestão
instituto politécnico de leiria

Instituto Politécnico de Leiria
Escola Superior de Tecnologia e Gestão
Departamento de Engenharia Informática
Mestrado em Cibersegurança e Informática Forense

AUTOMATED, SCHEDULED AND CI/CD WEB INJECTION

MYKYTA ZHYGULSKYY
Número: 2180074

Projeto realizada sob orientação do Professor Ricardo Gomes (ricardo.p.gomes@ipleiria.pt).

Leiria, Fevereiro de 2021

ACKNOWLEDGEMENTS

I thank my advisor Ricardo Jorge Pereira Gomes, for all the guidance and availability along this path, for all the help, for all the technologies and tools that were taught to me, and more especially for many of the doubts solved, and also in the way of thinking, problem-solving. I would also like to thank Professor Mário Antunes for providing me this advisor for the implementation of this project.

ABSTRACT

This report is made within the Curricular Unit (UC) Project, in the 2nd year of the Master in Cyber-security and Forensic Informatics (MCIF) provided by the Polytechnic Institute of Leiria (IPL). The purpose of this project is to study SQL Injection vulnerabilities in web applications. According to OWASP (Open Web Application Security Project) [20][19], this is one of the more prevalent attacks on web applications. As part of this work a web application was implemented, which can from a URL address, go through all the endpoints of the target application and test for SQL Injection vulnerabilities. The application also makes allows for scheduling of the tests and it is integrable with Continuous Integration / Continuous Delivery (CI/CD) environments. According to the literature on the subject, there are several algorithms that can be employed to test for existing SQL Injection vulnerabilities in a web application. In this document, we analyze them both from a theoretical and an implementation point of view. In order to better understand the subject, and produce a useful tool in this space. With the development of this project, we concluded that it is possible to integrate SQL vulnerability tests, with CI/CD pipeline and automate the development process of an application, with the execution of SQL injection tests in an automated way.

RESUMO

Este relatório é feito no âmbito da Unidade Curricular (UC) Projecto, no 2º ano do Mestrado em Cibersegurança e Informática Forense (MCIF) fornecido pelo Instituto Politécnico de Leiria (IPL). O objectivo deste projecto é o estudo das vulnerabilidades de SQL Injection em aplicações web. Segundo o OWASP (Open Web Application Security Project)[20][19], este é um dos ataques mais comuns a aplicações web. Como parte deste trabalho, foi implementada uma aplicação web, que pode a partir de um endereço URL, analisar todos os endpoints da aplicação alvo e testar as vulnerabilidades de SQL Injection. A aplicação também permite o agendamento dos testes e é integrável com ambientes de Integração Contínua / Entrega Contínua (CI/CD). De acordo com a literatura sobre o assunto, existem vários algoritmos que podem ser utilizados para testar as vulnerabilidades de SQL Injection existentes numa aplicação web. Neste documento, analisamo-los tanto do ponto de vista teórico como do ponto de vista da implementação. A fim de melhor compreender o assunto, e produzir uma ferramenta útil neste espaço. Com o desenvolvimento deste projecto, concluímos que é possível integrar testes de vulnerabilidade SQL, com a pipeline CI/CD e automatizar o processo de desenvolvimento de uma aplicação, com a realização de testes de injecção SQL de uma forma automatizada.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS

API	Application Programming Interface.
CD	Continuous Delivery.
CI	Continuous Integration.
CNI	Container Networking Interface.
CSI	Container Storage Interface.
CSS	Cascading Style Sheets.
DNS	Domain Name System.
DVWA	Damn Vulnerable Web Application.
HCL	HashiCorp Configuration Language.
HTML	Hypertext Markup Language.
HTTP	Hypertext Transfer Protocol.
IaC	Infrastructure as Code.
IPL	Polytechnic Institute of Leiria.
IT	Information Technology.
JSON	JavaScript Object Notation.
LDAP	Lightweight Directory Access Protocol.
MCIF	Master in Cybersecurity and Computer Forensics.
MVC	Model View Controller.
ORM	Object-Relational Mappers.
OS	Operating system.

OWASP	Open Web Application Security Project.
PDO	PHP Data Objects.
PHP	Hypertext Preprocessor.
PII	Personally Identifiable Information.
REST	Representational State Transfer.
SaaS	Software as a Service.
SMB	Server Message Block.
SMTP	Simple Mail Transfer Protocol.
SQL	Structured Query Language.
SQLi	SQL injection.
UC	Curricular Unit.
URI	Uniform Resource Identifier.
URL	Uniform Resource Locator.
VM	Virtual Machine.
XML	Extensible Markup Language.
XPath	XML Path Language.
XSS	Cross-Site Scripting.
XXE	XML External Entity.

1

INTRODUCTION

This project started with the idea of deepening the knowledge of software vulnerability attack vectors, with the focus from the start on injection vulnerabilities. Reviewing the literature it became apparent that SQL injection vulnerabilities were among not only the most prevalent but also the most costly.

The focus then became learning more about the internals SQL injection vulnerabilities and its detection techniques, as discussed in chapter 2, but also putting them in the context of the software development life cycle, in particular within the context of CI/CD processes and pipelines.

In order approach this problem with the ability to carry out incremental testing, of our understanding of the environment and the tools we might need to address it, we started by focusing yet again on a subset of the techniques for detecting SQL injection vulnerabilities, discussed in more detail on chapter 2, 3 and 4, and developed a web application that can execute those techniques on all local endpoints present on any web page. This implied a first step of managing to identify those endpoints in an way that required close to none user interaction, which we achieved by recursively parsing all the linked pages on the website and interpreting the requirements of each endpoint.

This application was built on the PHP Laravel framework which allowed us to take advantage of a number of built-in design patterns and tools, like the Model-View-Controller design pattern or the Eloquent Object-Relational Mapping tool. As described in the 3 and 4 chapters this application allows for multiple ways to execute the tests separated in three major ways. First my a simple manual task definition, secondly by a time base schedule and third using a REST endpoint. The implementation of these options allows the application to be used in a number of different scenarios and contributes heavily to its usefulness.

Having the first iteration of our testing application ready we started to look deeper at the software life cycle part of the problem. This was particularly insightful because we discovered that almost all, if not all, the tools in this space are targeted at security professionals and that means they required a fair set of knowledge to configure, operate and even interpret its results. We wanted to design a way to obfuscate most of this knowledge requirement by allowing our tool to integrate with existing frameworks and processes of Continuous Development. So the focus became

how to create an environment to learn, experiment, and ultimately create a tool to bridge this gap.

Finally this process ended by our need to have this validated outside our local environments, and so the infrastructure on Digital Ocean and usage on GitHub's workflows and Actions became the final piece of the puzzle.

For this, we took to the latest set of DevOps tools, like Infrastructure-as-code with Terraform, and containers and their orchestration with Docker and Kubernetes. Terraform allowed us to create our infrastructure in a real world provider in a controllable and replicable way. This is very important on project list this that require a lot of iterative work and so our infrastructure needed to be rebuilt a lot of times. Kubernetes was a way for us to not only deploy our testing application in a manageable way but also to provide a usable environment for our test application (DVWA) and it's life cycle.

As its better detailed on the following chapters all theses steps were necessary to have a firm grasp of the problem and to provide a tool that can be a first attempt at bridging the gap between security and development.

2

LITERATURE REVIEW

This chapter describes the state of the art, in the various topics that the work proposes, such as software vulnerabilities, security risks of the most common applications, SQL injections, Techniques of Detection of Injection Vulnerabilities, Scanning and Detection Tools, DevOps, Containers, Docker, Kubernetes, Infrastructure such as Code, CI/CD, and CI/CD tools.

2.1 SOFTWARE VULNERABILITIES

Increasingly, the vulnerability of software and the risks it can cause in companies is one of the most important and worrying issues for companies. These can lead to the loss or disclosure of private data, financial consequences and loss of reputation can be fatal for the company. According to current cybersecurity statistics [61], the average cost of a malware attack on a company is $2.6 million. The average cost of a data breach is $3.9 million, the average cost of the lost business is $1. 42 million, companies will fall victim to a rescue attack every 14 seconds at that time, 50% of large companies (with over 10,000 employees) spend $1 million or more annually on security, with 43% spending $250,000 to $999,999, being SQL Injection one of the most common security risks [19], these attacks and their causes are in part possible because of SQL Injection vulnerabilities.

2.1.1 *Injection*

According to OWASP which is an international organization dedicated to enhancing the security of web applications, based on the documents "The Ten Most Critical Web Application Security Risks" [20] and "The Ten Most Critical API Security Risks" [19], Injection in one of 10 most common type of attacks present in Web Application, Injection flaws can be found in SQL, LDAP, XPath, or NoSQL queries; OS commands; XML parsers, SMTP Headers, expression languages and ORM queries, injection flaws occur when unreliable data are sent as part of a command or query. The attack can then trick the target system into executing unwanted commands. An attack can also provide access to protected data to untrusted agents[69], in figure 1 SQL query is demonstrated in which the parameters sent by

```
SELECT UserList.Username
FROM UserList
WHERE UserList.Username = 'Username'
AND UserList.Password = 'password' OR '1'='1'
```

Figure 1: Sql Injection Query

the users, are not expected data but SQL command, which makes the answer not as expected, because the data the user sends is not treated correctly, so the user input is interpreted as a command and not as a parameter for the query, and in the case of the query in figure 1, a user is always returned, regardless of its existence. The most popular tool among all available SQL injection tools is SQLmap [66], it is the open source tool, this tool facilitates the exploitation of the SQL injection vulnerability of a web application and takes over the database server. It comes with a powerful detection engine that can easily detect most vulnerabilities related to SQL injection [8].

2.1.2 *The Other Most Common Application Security Risks*

In this work we selected the vulnerabilities of SQL injections, although there are other vulnerabilities that also need to be addressed as they have great impact and importance in the management and implementation of a web application today, and the work in this project can be augmented to support testing agains these in the future [55].

1. **Broken Authentication** - Application functions related to authentication and session management are often implemented incorrectly, a possible attack scenario, filling of credentials, the use of known password lists, is a common attack. If an application does not implement automated threat protection or credential filling, the application can be used as a password oracle to determine whether the credentials are valid.
2. **Sensitive Data Exposure** - Do not properly protect sensitive data, such as financial, healthcare, and PII, a possible attack scenario, the password database uses unsalted or simple hashes to store everyone's passwords. A file upload flaw allows an attacker to retrieve the password database. All the unsalted hashes can be exposed with a rainbow table of pre-calculated hashes. Hashes generated by simple or fast hash functions may be cracked by GPUs, even if they were salted.

3. **XML External Entity(XXE)** - Use External entities to disclose internal files using the file URI handler, internal SMB file shares on unpatched Windows servers, internal port scanning, remote code execution, and denial of service attacks, such as the Billion Laughs attack, a possible attack scenario, the attacker attempts to extract data from the server, figure 2.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<foo>&xxe;</foo>
```

Figure 2: XML External Entities

4. **Broken Access Control** - Restrictions on what authenticated users are allowed to do are not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access to other user accounts, view sensitive files, modify other user data, change access rights, and others, a possible attack scenario, an attacker simply modifies the 'acct' parameter in the browser to send whatever account number they want. If not properly verified, the attacker can access any user's account, figure 3.

```
http://example.com/app/accountInfo?acct=notmyacct
```

Figure 3: Broken Access Control

5. **Security Misconfiguration** - Incorrect configuration or insecure default configurations, a possible attack scenario, the application server comes with sample applications that are not removed from the production server. These sample applications have known security flaws attackers use to compromise the server. If one of these applications is the admin console, and default accounts weren't changed the attacker logs in with default passwords and takes over.
6. **Cross-Site Scripting (XSS)** - XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, a possible attack scenario, the application uses untrusted data in the construction of the following HTML snippet without validation or escaping, figure 4.

```
(String) page += "<input name='creditcard' type='TEXT'
value='" + request.getParameter("CC") + "'>";
```

Figure 4: Cross-Site Scripting (XSS)

7. **Insecure Deserialization** - Occur when an application receives hostile serialized objects. Insecure deserialization leads to remote code execution. Even if

deserialization flaws do not result in remote code execution, serialized objects can be replayed, tampered or deleted to spoof users, conduct injection attacks, and elevate privileges, a possible attack scenario, PHP forum uses PHP object serialization to save a “super” cookie, containing the user’s user ID, role, password hash, and other state, figure 5, an attacker changes the serialized object to give themselves admin privileges.

```
a:4:{i:0;i:132;i:1;s:7:"Mallory";i:2;s:4:"user";
i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}
```

Figure 5: Insecure Deserialization

8. **Using Components with Known Vulnerabilities** - Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover.
9. **Insufficient Logging & Monitoring** - Ineffective or missing integration with incident response allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract or destroy data, a possible attack scenario, an attacker uses scans for users using a common password. They can take over all accounts using this password. For all other users, this scan leaves only one false login behind. After some days, this may be repeated with a different password.

2.1.3 *SQL Injection*

Given the vastness of possible vulnerabilities, and the fact that it is one that is testable in an automatable way we decided to focus on it for this project. It was decided to focus mainly on SQL Injection, as it is the most used Injection type of attack [3], with the objective of developing an application, which performs the vulnerability test automatically, implementing some existing vulnerability testing techniques that will be described below.

An SQL injection attack consists of the insertion or "injection" of a SQL query through the customer’s input data into the application. A successful SQL injection exploit can read sensitive data from the database, modify data from the database (Insert/Update/Delete), perform database administration operations, an attacker may escalate an SQL injection attack to compromise the underlying server or other back-end infrastructure or perform a denial of service attack, some of the possible examples of SQL Injection are demonstrated in the figures, 6, figure 7 and figure 8 [63] [64].

```
SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name' OR 'a'='a';
```

Figure 6: SQL Injection Example

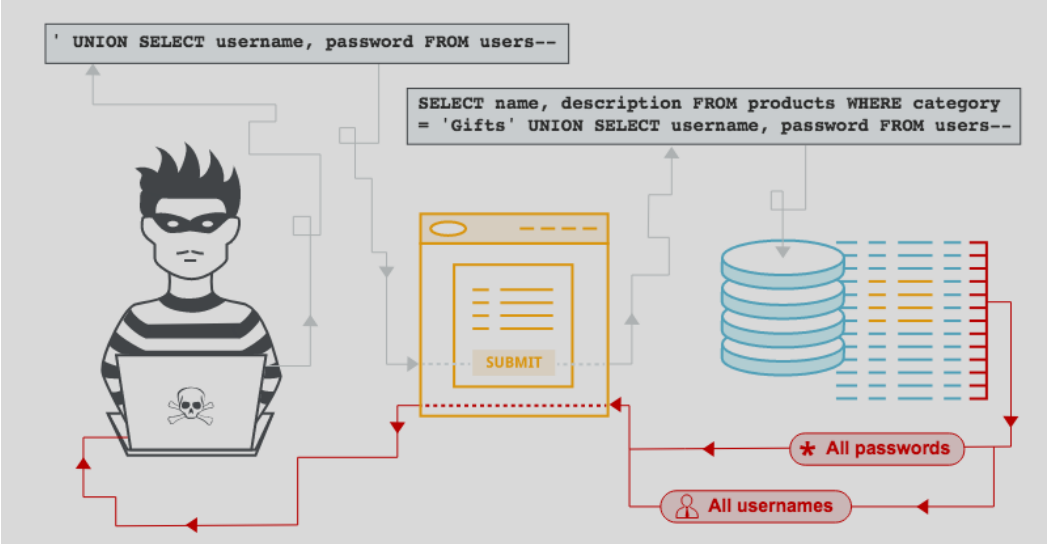


Figure 7: SQL Injection Example [64]

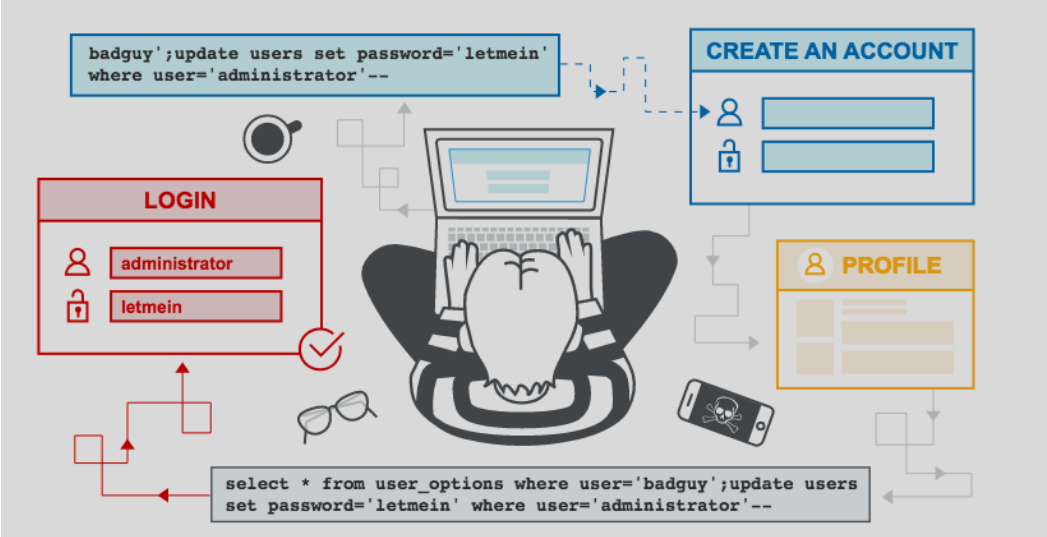


Figure 8: SQL Injection Example [64]

Currently, web applications are parts of our daily lives, such as access to online bank accounts, online shopping, and others. These applications generally interact with databases to access persistent data, which contains personal information. These applications have been continuously targeted by highly motivated malicious users to acquire monetary gain. Currently, in application development, customers are not

concerned with the security that the application has, and are more focused on the functionality of the application, until the moment an incident occurs. That is why it is the developers and system administrators who are responsible for the security of the application.

In order to automate this process, there are automatic vulnerability scanners, which provide vulnerability testing mechanisms to ensure that applications are minimally secure before they go into production [35].

In the following section we explain the existing techniques to detect SQLI vulnerabilities.

2.2 INJECTION VULNERABILITIES DETECTION TECHNIQUES

Two important characteristics of SQL injection attacks are the injection mechanism and different kinds of attacks that can be carried out. Below are some examples of SQL injection attack techniques [53].

1. In-band SQLi: It's one of the easiest to exploit, compared to other types. The attacker injects malicious code into a web application and the results are usually returned to the attacker's screen, Error-based SQLi is an example of an In-band SQLi [53].
2. Error-based SQLi: This technique is based on the error messages what are returned by database server after some reserved characters or malicious codes are posted on a form of a web application. That gives the attacker some information about the structure of the database and data stored inside the database [58]. This technique works when the target web application has been configured to disclose error messages which are used for testing during the phase of application development and should be disabled or logged to a file with restricted access. This technique is effective enough for an analyst to enumerate an entire database, tables, and the stored data.
3. Double Query-based SQLi: This technique uses a combination of two queries in a single query, with the purpose that the backend database returns the error message what usually contains information the attacker is trying to extract.
4. Blind SQLi: This technique is more complex because it requires that the attacker creates a well constructed logical query that will inject into a web application, "forcing" the application to return a different result and observe the result of this query in a backend database. This technique takes more time because the parameters are injected without knowing the database structure. And the results of these attacks are not dumped on the screen of the user or visible to the attacker thereby getting its name Blind SQL injection. There

are two types of Blind SQL Injection, the Boolean-based blind SQLi, and time-based Blind SQLi.

a) Boolean-based Blind SQLi: This technique consists of sending a valid SQL query, then changing the query and compare whether the query returns a TRUE or FALSE result. Depending on the result, the content within the HTTP response will change, or remain the same, can conclude if the endpoint is injectable or not [1].

b) Time-based Blind SQLi: This technique sends queries to the database which causes a delay in the amount of time, it takes the database to respond. To create this delay in time, the attacker must build a proper query to force the server to work. The response time indicates to the attacker whether the result of the query is TRUE or FALSE.

5. Union-based SQL injection: this injection technique that leverages the UNION SQL operator to combine the results of two or more SELECT statements into a single result. The combined results are then returned as part of the HTTP response, which could be displayed on the web page. This technique works when the web application page passes directly the output of the statement SELECT within a for loop or similar so that each line of the query output is printed on the page content [46].

In this project we implemented the Boolean-based Blind SQLi technique because it seems to be one of the most interesting and simple techniques to understand, which consists of sending requests, forcing the application to return a different value, changing the payload of each request, and checking whether or not the response remained the same, it is possible to verify this process in the figure 9 [9].

2.3 SCANNING AND DETECTIONS TOOLS

There are several types of vulnerability scanners, with different costs and features, below are some examples of them [70], and in the end, some of the applications that were tested.

2.3.0.1 *Paid Tools*

1. Rapid7 Nexpose - is a vulnerability scanner that aims to support the entire vulnerability management lifecycle, including discovery, detection, verification, risk classification, impact analysis, reporting, and mitigation. It integrates with Rapid7's Metasploit for vulnerability exploitation. It is sold as standalone

Figure 9: Boolean-based blind SQL injection

software, an appliance, virtual machine, or as a managed service or private cloud deployment. User interaction is through a web browser [50] [49].

2. Netsparker - is an web application security scanner automatically detects SQL Injection, Cross-site Scripting (XSS) and other vulnerabilities in all types of web applications [47] [48].

3. Acunetix - is an automated web application security testing tool that audits your web applications by checking for vulnerabilities like SQL Injection, Cross site scripting, and other exploitable vulnerabilities [74] [2].

4. ManageEngine Vulnerability Manager Plus - allows the detection, and assessment of vulnerabilities to their elimination with an automated patch workflow, all aspects of vulnerability management are facilitated with a centralized console. It is also possible to manage security settings, harden web servers, mitigate zero-day vulnerabilities, perform end-of-life audits, and eliminate risk software. Simplify vulnerability management with a remotely deployable agent, web-based interface, and infinite scalability [43].

2.3.0.2 *Free Tools*

1. Nmap - is a free and open-source network scanner.Used to discover hosts and services on a computer network by sending packets and analyzing the responses [52].

2. OpenVAS - is a software framework of several services and tools offering vulnerability scanning and vulnerability management[54].

3. Arachni - is an Open Source, feature-full, modular, high-performance Ruby framework aimed towards helping penetration testers and administrators evaluate the security of web applications [5].

4. XssPy - is a python tool for finding Cross Site Scripting vulnerabilities in websites [78].

5. w3af - is an open-source web application security scanner. The project provides a vulnerability scanner and exploitation tool for Web applications [73].

6. Nikto - is a free software command-line vulnerability scanner that scans webservers for dangerous files/CGIs [51].

7. OWASP ZAP - is an open-source web application security scanner. It is intended to be used by both those new to application security as well as professional penetration testers [56].

8. Grabber - is a web application scanner. Basically it detects some kind of vulnerabilities in website [30].

9. Golismero - is an open source framework for security testing [28].
10. OWASP Xenotix XSS - is an advanced Cross Site Scripting (XSS) vulnerability detection and exploitation framework [57][40].

2.3.0.3 *Tested Tools*

1. Sqlmap - is an open source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws, have a powerful detection engine, many niche features for the ultimate penetration tester and a broad range of switches lasting from database fingerprinting, over data fetching from the database, to accessing the underlying file system and executing commands on the operating system via out-of-band connections [66], SQLmap has features such as detection of the database used, existing tables, the structure of the database tables, and data from the tables, these features are possible to view in the figures 10, 11, 12 and figure 13.

```
[09:49:17] [INFO] the back-end DBMS is MySQL
web application technology: PHP 7.3.8, PHP, Nginx
back-end DBMS: MySQL > 5.0.11
[09:49:17] [INFO] fetching database names
[09:49:17] [INFO] used SQL query returns 5 entries
[09:49:17] [INFO] starting 3 threads
[09:49:17] [INFO] retrieved: 'information_schema'
[09:49:17] [INFO] retrieved: 'injections'
[09:49:17] [INFO] retrieved: 'mysql'
[09:49:17] [INFO] retrieved: 'data'
[09:49:17] [INFO] retrieved: 'performance_schema'
available databases [5]:
[*] data
[*] information_schema
[*] injections
[*] mysql
[*] performance_schema
```

Figure 10: SQLmap Databases

2. jSQL Injection - is a lightweight application used to find database information from a distant server, this application has characteristics like the visualization of the structure of the database in interface mode, this feature is possible to visualize in the figure 14 [36].
3. SQLiv - is a Python-based scanning tool that uses Google, Bing, or Yahoo for targeted scanning, focused on reveal pages with SQL Injection vulnerabilities, this feature is possible to visualize in the figure 15 [65].
4. Burp Suite Community Edition - is an integrated platform for performing security testing of web applications. Its various tools work seamlessly together to support the entire testing process, from initial mapping and analysis of

```
[09:52:23] [INFO] the back-end DBMS is MySQL
web application technology: PHP 7.3.8, PHP, Nginx
back-end DBMS: MySQL > 5.0.11
[09:52:23] [INFO] fetching tables for database: 'injections'
[09:52:23] [INFO] used SQL query returns 4 entries
[09:52:23] [INFO] starting 3 threads
[09:52:23] [INFO] retrieved: 'password_resets'
[09:52:23] [INFO] retrieved: 'users'
[09:52:23] [INFO] retrieved: 'failed_jobs'
[09:52:23] [INFO] retrieved: 'migrations'
Database: injections
[4 tables]
+-----------------+
| failed_jobs     |
| migrations      |
| password_resets |
| users           |
+-----------------+
```

Figure 11: SQLmap Tables

```
[09:53:57] [INFO] starting 3 threads
[09:53:57] [INFO] retrieved: 'id','bigint(20) unsigned'
[09:53:57] [INFO] retrieved: 'email','varchar(255)'
[09:53:57] [INFO] retrieved: 'name','varchar(255)'
[09:53:57] [INFO] retrieved: 'email_verified_at','timestamp'
[09:53:57] [INFO] retrieved: 'password','varchar(255)'
[09:53:57] [INFO] retrieved: 'remember_token','varchar(100)'
[09:53:57] [INFO] retrieved: 'created_at','timestamp'
[09:53:57] [INFO] retrieved: 'updated_at','timestamp'
Database: injections
Table: users
[8 columns]
+--------------------+---------------------+
| Column             | Type                |
+--------------------+---------------------+
| created_at         | timestamp           |
| email              | varchar(255)        |
| email_verified_at  | timestamp           |
| id                 | bigint(20) unsigned |
| name               | varchar(255)        |
| password           | varchar(255)        |
| remember_token     | varchar(100)        |
| updated_at         | timestamp           |
+--------------------+---------------------+
```

Figure 12: SQLmap Columns

```
Database: injections
Table: users
[20 entries]
+----+-----------------------+-----------------------+--------------------------------------------------------------+---------------------+---------------------+----------------+---------------------+
| id | name                  | email                 | password                                                     | created_at          | updated_at          | remember_token | email_verified_at   |
+----+-----------------------+-----------------------+--------------------------------------------------------------+---------------------+---------------------+----------------+---------------------+
| 3  | Dr. Efrain Balistreri | willms.alba@example.net    | $2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi | 2019-10-19 11:13:59 | 2019-10-19 11:13:59 | VM4vsvJtLZ     | 2019-10-19 11:13:59 |
| 4  | Luella Ratke          | ecartwright@example.org    | $2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi | 2019-10-19 11:13:59 | 2019-10-19 11:13:59 | rJdHS0KYZV     | 2019-10-19 11:13:59 |
| 5  | Mia Wuckert           | barton.greta@example.org   | $2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi | 2019-10-19 11:13:59 | 2019-10-19 11:13:59 | GlWvWs60b8     | 2019-10-19 11:13:59 |
| 6  | Effie Erdman          | murray.beaulah@example.com | $2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi | 2019-10-19 11:13:59 | 2019-10-19 11:13:59 | yPBJsse8Wn     | 2019-10-19 11:13:59 |
| 7  | Dr. Kendall Harvey V  | ehahn@example.org          | $2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi | 2019-10-19 11:13:59 | 2019-10-19 11:13:59 | FAGwuI7f7y     | 2019-10-19 11:13:59 |
| 8  | Jacklyn Lesch         | khermann@example.org       | $2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi | 2019-10-19 11:13:59 | 2019-10-19 11:13:59 | D0WhrHq7uS     | 2019-10-19 11:13:59 |
| 9  | Mrs. Chelsea Simonis  | hailey.oreilly@example.net | $2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi | 2019-10-19 11:13:59 | 2019-10-19 11:13:59 | hwvDSB6Vr8     | 2019-10-19 11:13:59 |
| 10 | Tomasa Bergnaum DDS   | abby52@example.org         | $2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi | 2019-10-19 11:13:59 | 2019-10-19 11:13:59 | ZR5GQNvQmU     | 2019-10-19 11:13:59 |
| 11 | Letha Prohaska        | hbartoletti@example.com    | $2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi | 2019-10-19 11:13:59 | 2019-10-19 11:13:59 | xnFBvLZFqi     | 2019-10-19 11:13:59 |
| 12 | Donavon Bartell       | rkuphal@example.org        | $2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi | 2019-10-19 11:13:59 | 2019-10-19 11:13:59 | bezSkvthzM     | 2019-10-19 11:13:59 |
| 13 | Tyra Padberg          | leonie18@example.com       | $2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi | 2019-10-19 11:13:59 | 2019-10-19 11:13:59 | lbwtDkSAMb     | 2019-10-19 11:13:59 |
| 14 | Mr. Lawson Brekke Jr. | cremin.enola@example.net   | $2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi | 2019-10-19 11:13:59 | 2019-10-19 11:13:59 | 8C9BIXbESE     | 2019-10-19 11:13:59 |
| 15 | Rolando Hansen        | szieme@example.org         | $2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi | 2019-10-19 11:13:59 | 2019-10-19 11:13:59 | 7FzkrpQxdU     | 2019-10-19 11:13:59 |
| 16 | Neoma Schulist        | otilia.jacobi@example.net  | $2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi | 2019-10-19 11:13:59 | 2019-10-19 11:13:59 | Cr3Wn8pKJY     | 2019-10-19 11:13:59 |
| 17 | Otho Koch             | sjenkins@example.net       | $2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi | 2019-10-19 11:13:59 | 2019-10-19 11:13:59 | IXhYYnvVDF     | 2019-10-19 11:13:59 |
| 18 | Naomi Kerluke         | modesto.koch@example.org   | $2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi | 2019-10-19 11:13:59 | 2019-10-19 11:13:59 | K7DOk2Pfjj     | 2019-10-19 11:13:59 |
| 19 | Lurline Friesen       | koepp.lydia@example.com    | $2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi | 2019-10-19 11:13:59 | 2019-10-19 11:13:59 | OHE4gmRQtA     | 2019-10-19 11:13:59 |
| 20 | Marcelo Stokes Sr.    | dsauer@example.com         | $2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi | 2019-10-19 11:13:59 | 2019-10-19 11:13:59 | jf4gjBY2VX     | 2019-10-19 11:13:59 |
| 21 | dadsa                 | dasd@dasd.com              | $2y$10$lH7wVAZcv06cdb.yrF4ZT.kNgXd4/IHHf.0NjG7lroNDzg8FlcWXG | NULL                | NULL                | NULL           | NULL                |
| 22 | bal bl                | bal@dasd.com               | $2y$10$8JOavEfUrci.iR6QkeM9dO4eNQ/M650koJds7ekDKOhRUyhYQ7Dxm | NULL                | NULL                | NULL           | NULL                |
+----+-----------------------+-----------------------+--------------------------------------------------------------+---------------------+---------------------+----------------+---------------------+

[09:54:38] [INFO] table 'injections.users' dumped to CSV file '/root/.sqlmap/output/injection-base.test/dump/injections/users.csv'
[09:54:38] [INFO] fetched data logged to text files under '/root/.sqlmap/output/injection-base.test'

[*] ending @ 09:54:38 /2019-10-20/

root@kali:~#
```

Figure 13: SQLmap Table Data

an application's attack surface, through to finding and exploiting security vulnerabilities [12].

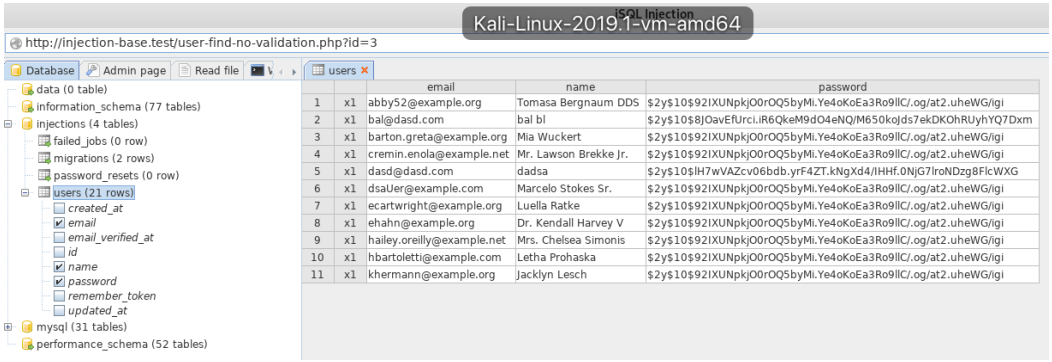


Figure 14: jSQL Injection

Figure 15: SQLiv

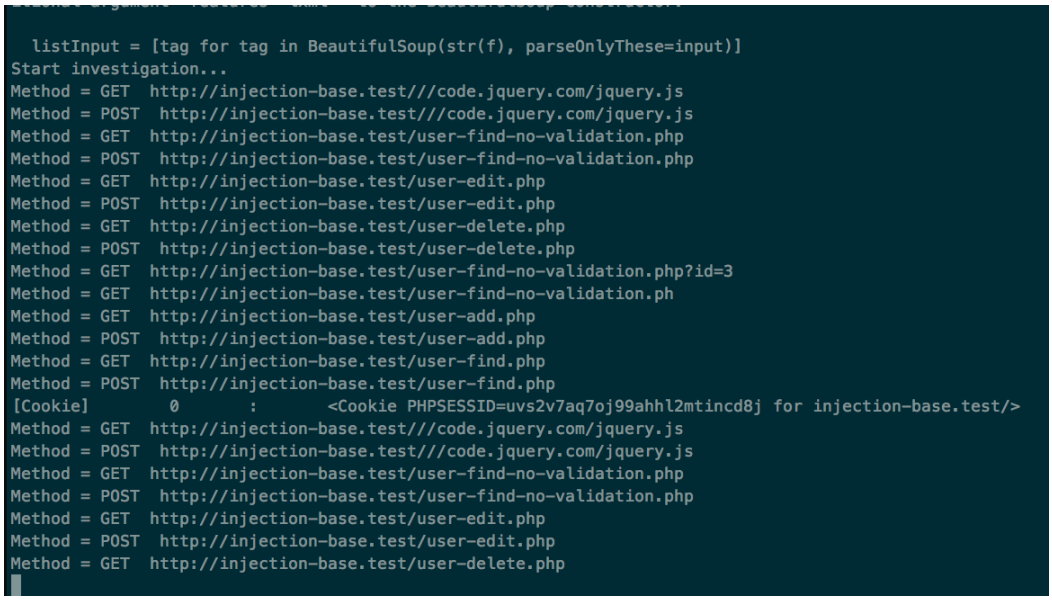


Figure 16: Grabber

5. Grabber - is a web application scanner that detects some kind of vulnerabilities in websites is simple, not fast but portable and adaptable, is designed to scan small websites such as personal sites, forums, and other sites of the same type [29], figure 16.

6. BBQSQL - is a blind SQL injection framework written in Python, useful for attacking complicated SQL injection vulnerabilities, it is a semi-automatic tool, allowing a little customization for triggering the results of SQL injection,

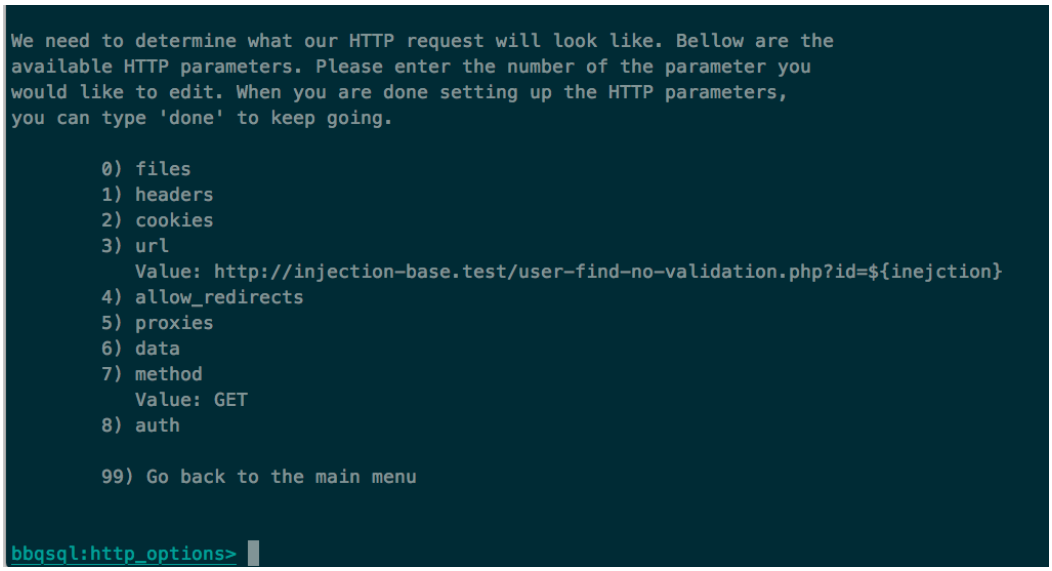


Figure 17: BBQSQL

it is built to be agnostic to the database and is extremely versatile, it also has an intuitive UI to make setting up attacks much easier, in figure 17, it is possible to observe the great possibility of the parameters to configure, in the SQL injection attack configuration [7].

The tested applications were chosen to be tested because most were already installed by default in the Kali Linux Operating System [37]. The applications tested do not support the integration with the CI/CD pipeline and the capture of the Login Session of it was implemented in the developed application.

2.4 DEVOPS

DevOps is a set of practices that work to automate and integrate processes between software development and IT teams. It aims to reduce and automate the systems development lifecycle so they can build, test, and launch software more quickly and reliably [76][16]. The DevOps life cycle consists of six phases, plan, build, integrate and implement continuously (CI/CD), monitor, operate, and respond to continuous feedback, figure 18.

The developed project is included in the Build phases, where the application is being built, Continuous Integration and Continuous Deployment with the CI / CD mechanism, and Continuous Feedback with the scheduling mechanism.

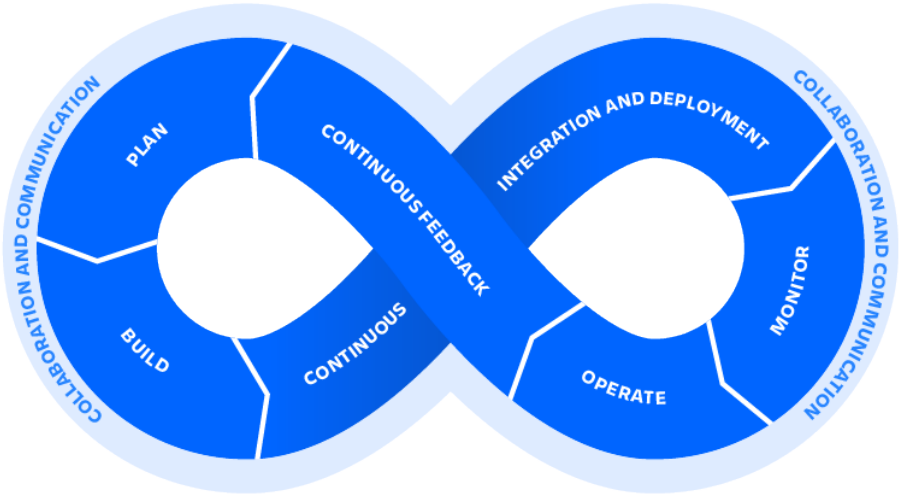


Figure 18: DevOps Lifecycle

2.4.1 *Containers*

A container is a standard software unit that packages code and all its dependencies, which allows programmers to isolate their application from its environment, and runs quickly and reliably from one computing environment to another [75][15].

2.4.1.1 *Docker*

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. The containers allow the application to be separated into parts according to its needs or dependencies, which will be treated as an entire application and will be executed in other systems in the same way. The Docker is a bit like a virtual machine, with some advantage it allows applications to use the same Linux kernel, which removes the extra layer and gives an increase in performance [77].

2.4.1.2 *Kubernetes*

Kubernetes is an open source system for managing containerized applications across multiple hosts. It provides basic mechanisms for deployment, maintenance, and scaling of applications [38]. Haff and Henry wrote "Kubernetes, or k8s, is an open source platform that automates Linux container operations. It eliminates many of the manual processes involved in deploying and scaling containerized applications" [10].

Benefits of Kubernetes

1. **Availability.** Kubernetes clustering has very high fault tolerance built-in, allowing for extremely large scale operations.
2. **Auto-scaling.** Kubernetes can scale up and scale down based on traffic and server load automatically.
3. **Extensive Ecosystem.** Kubernetes has a strong ecosystem around Container Networking Interface (CNI) and Container Storage Interface (CSI) and inbuilt logging and monitoring tools [39].

In the project developed, Kubernetes manages the containers and the necessary resources for them and ensures that they are always available.

2.4.2 *Infrastructure as Code*

IaC is the process of managing and provisioning computer data centers through machine-readable definition files, rather than physical configuration of interactive hardware or configuration tools[32], which can be in conjunction with the application's source code and evolve with it.

2.4.2.1 *Terraform*

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. Users define and provision data center infrastructure using a declarative configuration language known as HCL, or optionally JSON. The infrastructure Terraform can manage includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc[33][68]. In the developed project, Terraform allows us to have the configuration of the infrastructure described in a file, from which it is possible to create it automatically.

2.4.3 *CI/CD*

Continuous integration and continuous delivery (CI/CD) is a practice that enables development teams to deliver high-quality code more frequently and reliably [14][21], with a continuous integration strategy, the new code changes in an application are regularly created, tested, and merged into a shared repository. In order to ensure that those changes do not corrupt the application. And with a continuous delivery strategy refers to continuous delivery and/or continuous deployment of

the application in a live production environment automatically [59]. Apart from tailor-made software or large vulnerability detection software packages integrated into enterprise environments, there are no tools available to add to the CI/CD process elements of vulnerability detection in a simple way for development teams or DevOps.

CI / CD pipelines traditionally focus on functionality, but due to the distributed nature of current software and the number of cyberattacks the non-functional aspects have also started to be addressed. As a result, pipelines must not only maintain functional tests but also include specific safety tests, such as checking imported libraries with Software Composition Analysis tools such as OWASP Dependency-Check or Retire.js, analyzing and hardening the infrastructure by using, for example, Inspec, Nmap, or cloud-based tools, checking for secrets out in the open with git-secrets or similar solutions, targeting specific issues with SQLMap, SSLyze, and others [13].

Here are some examples of CI/CD tools in use in the industry.

CI/CD Tools

1. Github Actions - are a relatively new feature to Github that allow, to set up CI/CD workflows using a configuration file right in your Github repository [22] [23].
2. Jenkins - is a free and open source automation server, helps automate the parts of software development related to building, testing, and deploying, facilitating continuous integration and continuous delivery [34].
3. Gitlab CI/CD - is a free and self-hosted Continuous Integration tool built into GitLab CI/CD, has a community edition, and provides git repository management, issue tracking, code reviews, wikis, and activity feeds [25] [26].
4. Travis CI - is a hosted continuous integration service used to build and test software projects hosted at GitHub and Bitbucket, was the first CI service which provided services to open-source projects for free, however free open-source plans were removed in the end of 2020 [71].
5. GoCD - is an open-source tool that is used in software development to help teams and organizations automate the continuous delivery of software. It supports automating the entire build-test-release process from code check-in to deployment [27].

In conclusion, the area of analysis and detection of vulnerabilities is both vast and mature, and to some extent, the same can be said for the area of CI/CD. So it would be expected that the volume of attacks dependent on software vulnerabilities would be decreasing, however, this is not the case. In the analysis of the state of

the software, we concluded that more work is needed in linking these two areas of computer science, and this project is our contribution.

3

ARCHITECTURE

The architecture used in this project is the Client-Server model, in which the client makes the request to the server using a computer network, the server is always listening for requests, in order to receive an order, it will handle that request and return the response to the user.

In our case, the server is a Web application, implemented using the Laravel framework, and it communicates with the MariaDB database.

As we can see in figure 19, our application receives the request from the user that must include an URL to be tested, it then does a set of requests to that URL to collect all the testable URLs it can find. The application does this by parsing the HTML and looking for all references to local URLs (URLs with the same hostname, or without explicit hostnames). After that first set of requests, it goes through all the URLs and runs the vulnerability testing on each one.

Finally, it returns the result to the user with the list of tested URLs and the status for each test.

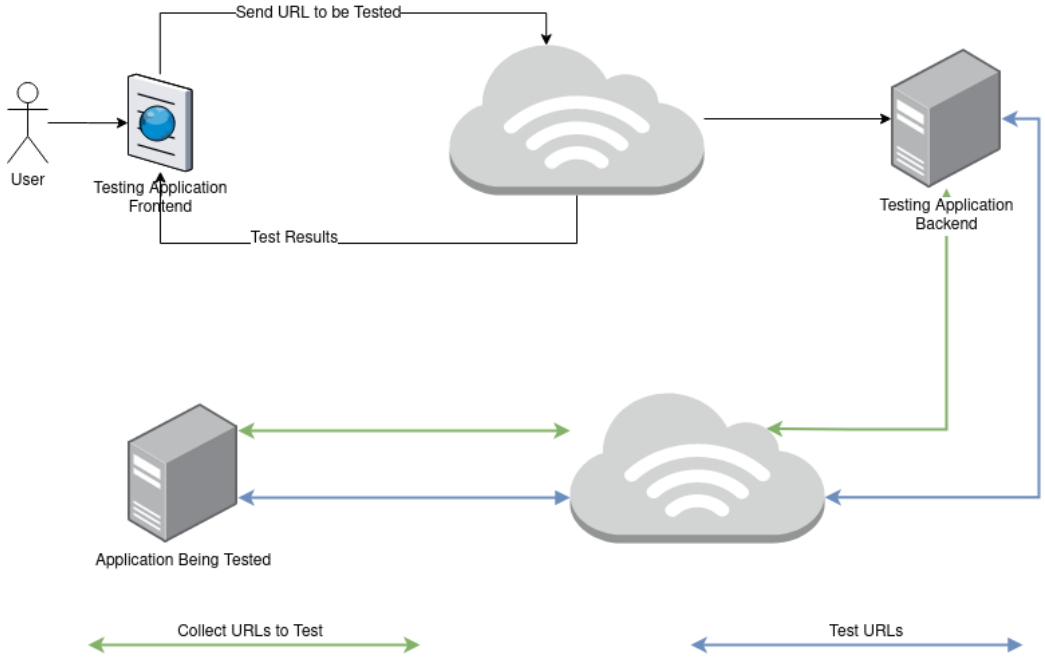


Figure 19: Architecture

The testing mechanisms are explained in further detail in the "Testing Mechanisms" section

With the purpose of supporting the usage of the Testing Application in CI/CD pipelines our application also exposes a REST endpoint that receives the URL to be tested and an authentication token. In this case the user / browser components of figure 19 are a simple HTTP request to our REST endpoint.

Another feature of application testing that has some advantages is to continuously test live applications. To support this approach our Testing Application also supports scheduling a test to be done in the future as a one-off test or as a periodic event. The details of this functionality are explained in the "Scheduling and Management" section.

Laravel applications use the MVC model (Model View Controller), which is a software architecture pattern that separates the logic of the application, the user interface and the components responsible for communicating with the database, in figure 20 we can see the MVC flow, which starts with the user making a request, that request goes through a certain route, which will reach a controller, the controller will execute some logic, for example, communicate with the model, which is connected to the database, and this, in turn, will return some data that will be presented to the user.

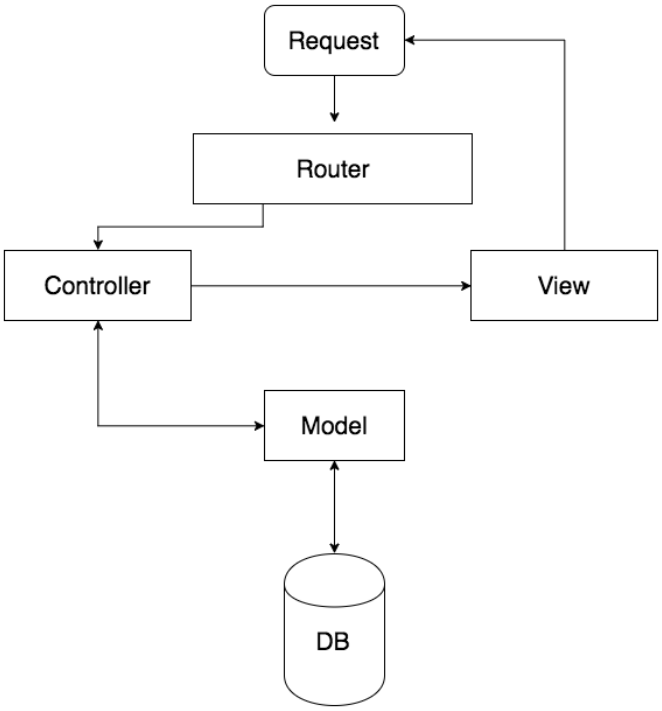


Figure 20: MVC Flow

3.1 LOGIN SESSIONS DVWA

A mechanism has been implemented to capture login sessions in order to support testing of vulnerabilities in authorized endpoints.

Before the SQL injection test runs, cookies, and other session state elements, are captured. At the moment it is only implemented for the DVWA application, but the mechanism was built to support other types of applications, for this purpose, there is a table in the database that stores the type of application that the Test Application supports for testing and some new fields in the tables tasks and endpoints. That stored information will be used in injection tests, in addition, the code structure has been designed in such a way that the new type of application to be supported is easy to integrate with the rest of the application. For the DVWA application, the session capture has the following procedures:

- A GET request is made the DVWA login page.
- Capture `"user_token"` that is present in the login page source code, figure 21.

```
<form action="login.php" method="post">
<fieldset>
        <label for="user">Username</label> <input type="text" class="loginInput" size="20" name="username"><br />

        <label for="pass">Password</label> <input type="password" class="loginInput" AUTOCOMPLETE="off" size="20" name="password"><br />
        <br />
        <p class="submit"><input type="submit" value="Login" name="Login"></p>
</fieldset>
<input type='hidden' name='user_token' value='385976d902e041817304655b2a2af8ff' />
</form>
<br />
```

Figure 21: DVWA User Token.

- Save in the DataBase the PHP SESSION TOKEN Cookie also comes with a response, to use in the next requests to this page.
- Then a POST request is executed on the login page with the captured `"user_token"` and the Cookie that was returned by the DVWA application.
- In case of the answer redirects to the Home page, it means that the login is done successfully and now it is possible to use the application to execute SQL Injection tests, figure 22.

Figure 22: DVWA Welcome Page.

The SQL Injection applications that were tested do not support the capture of sections except the "The Burp Suite" application, which supports something similar, more flexible and global, but a more complex, which requires configuration by the user[12], so this feature has been implemented that allows automatic capture of the session and subsequent testing of SQL injections, without the need for prior configuration by the user.

3.2 APPLICATION DATA BASE ARCHITECTURE

The implemented application use relational database MariaDB[44], so that the test results are persistent. The database is constituted by several tables as we can see in schema present in figure 23.

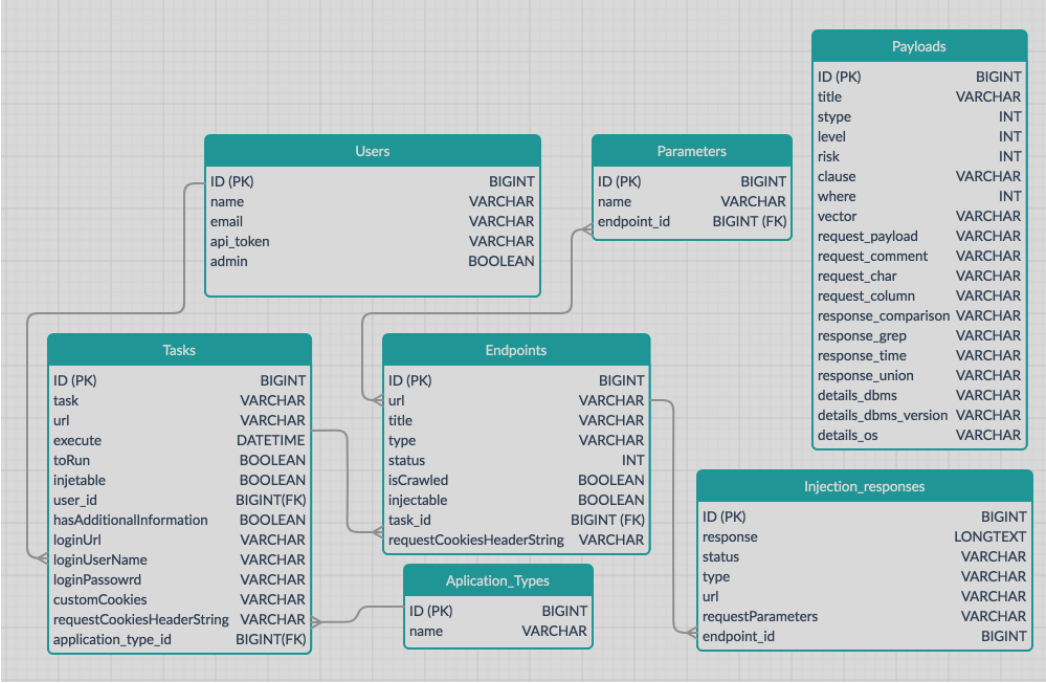


Figure 23: Application Relational Data Base

The table "Users", which contains the users of the application, has the column `"api_token"`, which is used by the users to create tasks from our REST endpoint, and column admin, which is a flag column to indicate if the user is admin.

Table "Tasks" stores the tasks that the users create, That table has the field "task" which is the name of the task to be executed, field "URL" is the address of the application that the user wants to test, and the date when the test should be executed, all the tasks are associated to the user who created them from by foreign key `"user_id"`.

The table "Endpoint" contains the URL field that is the endpoint for the test that was executed, the type of endpoint, the code returned to the test execution and if it is injectable or not, each endpoint belongs to a specific task.

The table "Parameters" stores the parameters associated with each endpoint and the table `"injection_responses"`, which has the answers of the tests executed to each endpoint, that can be used for later analysis and execution of other types of SQL attack techniques.

Injection_Response table stores the response of the requests that are made in the injection tests, the response status, URL, request parameters, and the endpoint_id that is the foreign key of the endpoints table.

Application_Types table saves the types of application that can be tested, used for session capture.

Payloads table stores detailed information about the payloads, their testing depth, the risk that is applied, where the payload is applied, identifier if the injected payload was successful, and others, the explanation of them is further below.

The following are the detailed structures of the various tables for easy reference.

3.2.1 *Data Base Architecture List View*

3.2.1.1 *Users Table*

- **id (BIGINT)** - Data base primary key, user identifier.
- **name (VARCHAR)** - User Name.
- **email (VARCHAR)** - User Email.
- **password (VARCHAR)** - User Password Hash.
- **`api_token` (VARCHAR)** - User Api access token.
- **admin (BOOLEAN)** - Admin flag.

3.2.1.2 *Tasks Table*

- **id (BIGINT)** - Data base primary key, task identifier.
- **task (VARCHAR)** - Task Name.
- **url (VARCHAR)** - Url of application to test.
- **Execute (DateTime)** - Execute Date and Time.
- **toRun (BOOLEAN)** - Flag of the task, if the task was tested.

- **injectable (BOOLEAN)** - Flag if url is injectable.
- `user_id` **(BIGINT Foreign)** - User Identifier.
- `hasAdditionInformation_id` **(BOOLEAN)** - Flag if the task has additional information.
- `loginUrl` **(VARCHAR)** - Login URL of application to test.
- `loginUserName` **(VARCHAR)** - Login username of application to test.
- `loginPassword` **(VARCHAR)** - Login password of application to test.
- `customCookies` **(VARCHAR)** - Personalized cookies that the user can pass.
- `requestCookiesHeaderString` **(VARCHAR)** - Cookies that is used in the injection test.
- `application_type_id` **(BIGINT Foreign)** - Application identifier.

3.2.1.3 *Endpoints Table*

- **id (BIGINT)** - Data base primary key, endpoint identifier.
- **url (VARCHAR)** - Url to test
- **title (VARCHAR)** - Page title
- **type (VARCHAR)** - Page type
- **status (INT)** - Page request status
- **isCrawled (BOOLEAN)** - Flag of the endpoint, if the endpoint was tested
- **injectable (BOOLEAN)** - Flag if url is injectable
- `task_id` **(BIGINT Foreign)** - Task Identifier

3.2.1.4 *Parameters Table*

- **id (BIGINT)** - Data base primary key, parameter identifier.
- **name (VARCHAR)** - Parameter Name
- `endpoint_id` **(BIGINT Foreign)** - Endpoint Identifier

3.2.1.5 *`Injection_Responses` Table*

- **id (BIGINT)** - Data base primary key, response identifier.
- **response (LONGTEXT)** - Response
- **status (VARCHAR)** - Request status

- **type (VARCHAR)** - Request type
- **url (VARCHAR)** - Request url
- **requestParameters (VARCHAR)** - Request parameters String
- **`endpoint_id` (BIGINT Foreign)** - Endpoint Identifier

3.2.1.6 *`Injection_Responses` Table*

- **id (BIGINT)** - Data base primary key, response identifier.
- **response (LONGTEXT)** - Response
- **status (VARCHAR)** - Request status
- **type (VARCHAR)** - Request type
- **url (VARCHAR)** - Request url
- **requestParameters (VARCHAR)** - Request parameters String
- **`endpoint_id` (BIGINT Foreign)** - Endpoint Identifier

3.2.1.7 *`Application_Types` Table*

- **id (BIGINT)** - Data base primary key, application identifier.
- **status (VARCHAR)** - Application name

3.2.1.8 *`Payloads` Table*

- **id (BIGINT)** - Data base primary key, payload identifier.
- **title (VARCHAR)** - Title of the test.
- **stype (INT)** - SQL injection family type.
- **level (INT)** - From which level check for this test.
- **risk (INT)** - Likelihood of a payload to damage the data integrity.
- **clause (VARCHAR)** - In which clause the payload can work.
- **where (INT)** - Where to add our <prefix> <payload><comment> <suffix> string.
- **vector (VARCHAR)** - The payload that will be used to exploit the injection point.
- **request_payload (VARCHAR)** - The payload to test for.
- **request_comment (VARCHAR)** - Comment to append to the payload, before the suffix.

- **request_char (VARCHAR)** - Character to use to bruteforce number of columns in UNION query SQL injection tests.
- **request_columns (VARCHAR)** - Range of columns to test for in UNION query SQL injection tests.
- **response_comparison (VARCHAR)** - Perform a request with this string as the payload and compare the response with the <payload> response. Apply the comparison algorithm.
- **response_grep (VARCHAR)** - Regular expression to grep for in the response body.
- **response_time (VARCHAR)** - Time in seconds to wait before the response is returned.
- **response_union (VARCHAR)** - To test for UNION query (inband) SQL injection.
- **details_dbms (VARCHAR)** - What is the database management system (e.g. MySQL).
- **details_dbms_version (VARCHAR)** - What is the database management system version (e.g. 5.0.51).
- **details_os (VARCHAR)** - What is the database management system underlying operating system.

With the Data Base architecture developed, it is quite simple to add other SQL injection techniques and also other applications to capture the Login Session.

3.3 CONTINUOUS INTEGRATION AND CONTINUOUS DELIVERY

Since one of the goals of our Testing Application is to integrate with CI/CD pipelines, we included the implementation of that feature by using our code repository tool - GitHub - and it's GitHub Actions feature. In the application that is the target of the SQL Injection tests, a configurable automated process (called Workflow in GitHub Actions) was developed, triggered by each push to the repository.

3.3.0.1 *GitHub action procedures:*

1. Fetch the latest version of the application.
2. Install the doctl command-line client[24].
    a) Pass Digital Ocean token by GitHub Secrets [25].

```yaml
- name: Install doctl
  uses: digitalocean/action-doctl@v2
  with:
    token: ${{ secrets.DIGITALOCEAN_ACCESS_TOKEN }}
```

Figure 24: Intall doctl

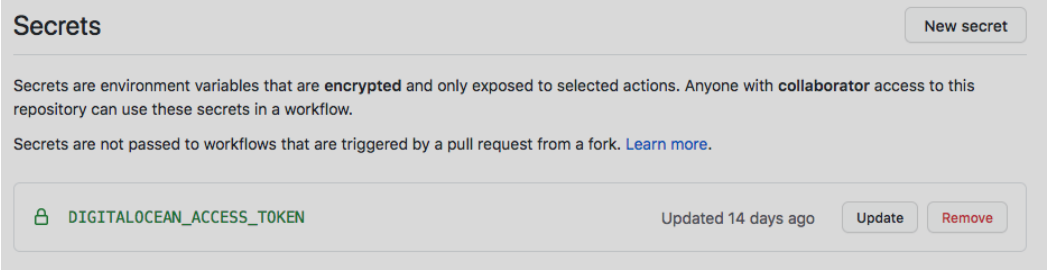


Figure 25: GitHub Secret

3. Save Digital Ocean cluster kubeconfig.
4. Build Docker images and apply them to Kubernetes cluster 26.

```yaml
    # docker images build
- name: Image Build
  run: docker build . -f Dockerfile

    # kubectl apply
- name: Apply Build
  run: kubectl apply -f kubernetes
```

Figure 26: Build Docker images and Apply to Kubernetes cluster

5. Execute "webhook.sh" file that is in the root of the repository 27.
    a) Make POST request to our test application API, which will create a new task and perform the SQL Injection tests.
6. In case Success response.
    a) Workflow will apply the current images to the production cluster.
7. In case of any endpoint be injectable of SQL Injection.
    a) Workflow is terminated without publishing the application in production 28.

```
HTTP_RESPONSE=$(curl --request POST \
  --silent --write-out "HTTPSTATUS:%{http_code}" \
  --max-time 600000 \
  --url $server_ip/api/tasks \
  --header 'authorization: Bearer 93beedc9481fca5580e2e462c3adbb113c24f1bcfac4efda80012170f1b21076' \
  --header 'content-type: application/json' \
  --data "{
  \"name\": \"New task\",
  \"url\": \"dvwa.test.$ip.nip.io/vulnerabilities/sqli/\",
  \"hasAdditionalInformation\": true,
  \"application_type_id\": 1,
  \"loginUrl\": \"dvwa.test.$ip.nip.io/login.php\",
  \"loginUserName\": \"admin\",
  \"loginPassword\": \"password\",
  \"customCookies\": \"security=low\"
}")

echo $HTTP_RESPONSE

# Get Body
HTTP_BODY=$(echo $HTTP_RESPONSE | sed -e 's/HTTPSTATUS\:.*//g')

# Remove '[ & ]'
HTTP_BODY="${HTTP_BODY//[}"
HTTP_BODY="${HTTP_BODY//]}"

# Get Status
HTTP_STATUS=$(echo $HTTP_RESPONSE | tr -d '\n' | sed -e 's/.*HTTPSTATUS://')
echo "Status $HTTP_STATUS"
# Print Injectable Endpoints
if [ $HTTP_STATUS -eq 400 ]; then
  printTable ',' "Injected Endpoints\n${HTTP_BODY//,/\\n}"
  exit 1
fi

exit 0
```

Figure 27: Webhook

b) Return list of injectable endpoints.

The set of elements designed and implemented allows us to have a high degree of confidence that this approach provides a meeting between the disciplines of Security and CI/CD in order to allow, without effort or acquisition of new skills by development teams, a greater surface of timely detection of software vulnerabilities. At the moment the proposed architecture focuses on one type of SQLI vulnerabilities, but it is easily expanded to support not only more types but other vulnerabilities.

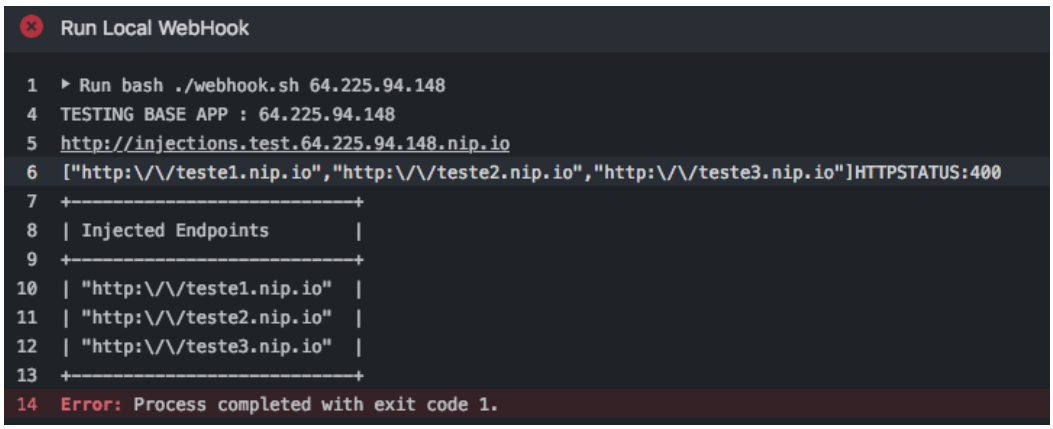


Figure 28: GitHub actions webhook

4

DEVELOPMENT

This chapter describes the application that has been implemented, namely, how the SQL Injection mechanism works, the first application that was used as a target application for testing, the infrastructure used, and the settings for its implementation, the CI/CD mechanism, some of the choices made during the development of the project, and some of the tools used for its implementation.

4.1 APPLICATION

An application was implemented using the Laravel framework [41]. The main tool that is used to execute tests is Guzzle, a PHP HTTP client[18], this allows us to make outgoing HTTP requests in a simple and quick mannager[31]. The test starts with the GET request to the URL address of the web application the user wants to test, the system analyzes the HTML response, and tries to find the links to the subsections of the test application. This process is recursive and the same applies to all links found. At the end of the endpoint search process, the SQL Injection vulnerability test will be executed, using the "Boolean-based Blind SQLI" Technique, explained in the Literary Review section. After all the endpoints are tested, the results are returned and contain which of them suffers from an SQL Injection vulnerability.

4.1.1 *Testing Application*

At the beginning of the project, a very simple application was implemented in order to have SQL injection vulnerability, which served as a target for the execution of SQL injection tests. But over time there was a need to change the target of the tests to a more reliable application and the DVWA application was chosen because its goal is to help security professionals test their skills and tools in a legal environment.

4.2 INFRASTRUCTURE

The following sub-sections describe our infrastructure components and the decisions that were made.

4.2.1 *DigitalOcean*

DigitalOcean is a cloud computing provider that offers an Infrastructure as a Service (IaaS) platform. It is very popular among open source developers, offers a reliable and easy to use virtual server, object storage as well as managed services linke Kubernetes [17][60]. DigitalOcean offers us the service with which it is possible to create our infrastructure, automatically, since it has the integration with tools such as Terraform and dispose our application for the internet.

4.2.2 *Terraform Configuration*

```
resource "digitalocean_kubernetes_cluster" "kubernetes-1" {
  name = "kubernetes-1"
  region = "fra1"
  version = "1.18.8-do.1"

  node_pool {
    name       = "worker-pool"
    size       = "s-1vcpu-2gb"
    node_count = 3
  }
}
```

Figure 29: Terraform Config

The Kubernetes cluster is created using the terraform tool in which resources are defined with the necessary pool. As we can see in figure 29 our pool is composed of three nodes of a specific type. Kubernetes will always try to make sure that the defined infrastructure is up and running, and since this is a managed cluster on Digital Ocean all the underlying resources are created automatically. Terraform allows us to have the guarantee that we can create the necessary infrastructure from nothing without human intervention, and as our project is based on the idea that automation mechanisms allow sustainable results this feature is indispensable.

4.2.3 *CI/CD Mechanism*

In order to make sure our Testing Application can indeed be used inside a CI/CD pipeline we used GitHub Actions to define a simple pipeline as an example.

A workflow (name for pipeline in GitHub Actions) was configured, workflow starts with install the doctl command-line client with past token created through Digital Ocean, which enables it to interact with DigitalOcean services[67]. Then the Docker images are built and applied to the Kubernetes cluster. Next, an HTTP POST will be sent to the API endpoints of our test application.

The current application will be tested, at the end of which a response will be returned and in case of success the workflow will apply the current images to the production cluster, if there is any endpoint that is vulnerable to SQL Injection, the execution flow is terminated without publishing the application in production. The CI/CD mechanism allows us to call the test application automatically, without the need for someone to interact with it, and to apply the new version of the application that has been tested, automatically, in production, in case of test success. A more detailed layout of this process is described in it's own chapter.

4.3 CHOICES

Throughout the life of the project, it was necessary to make some choices for the project to come to an end with the main objective of the completed application. One of them was at the beginning of the project we started to use the Laravel Dusk test tool[42], but at the end of the implementation, it was observed that it had some of the limitations as session capture, as this point was considered important and interesting to explore, it was decided to use HTTP requests with the Guzzle tool.

Another choice that was made is to publish the implemented application and the application that will be tested (DVWA) for the Internet, which solves a series of limitations that the approach of building applications within a GitHub Actions CI/CD pipeline and makes the scenario equal to production, thus the applications are totally independent and the GitHub Actions CI/CD pipeline only interacts with the applications.

The implementation of the necessary services using the Kubernetes orchestraor was another change the project had. In its first implementation, the publication of the application was through the application of Ansible, which installed the services and configured the necessary services in a resource created through the Terraform tool. This strategy was changed because the necessary services are now managed

through Kubernetes. Although they are the same way hosted and requested, through the Terraform application to DigitalOcean services.

4.4 USED APPLICATIONS

Although these applications are not used in the final application, they were important for the process of discovery of these areas and preliminary testing of the concept and proposed approaches.

4.4.1 *Ansible*

Ansible is an open-source software provisioning, configuration management, and application-deployment tool enabling infrastructure as code[4].

4.4.2 *Minikube*

Minikube implements a local lightweight Kubernetes cluster. That creates a VM on your local machine and by deploying a simple cluster containing only one node[72][24]. This was used as our target cluster during development.

5

TEST AND RESULTS

5.1 TESTS SETUP

```php
public static function find($userId)
{
    return self::fetchFirst(
        'SELECT * FROM users WHERE id = ?',
        [$userId]
    );
}
public static function findWithoutValidation($userId)
{
    return self::fetchWithoutValidation(
        'SELECT * FROM users WHERE id = ' . $userId
    );
}
```

Figure 30: PDO::Query Not sanitized

The test application was adapted from the application that was developed in the Applications for the Internet course of the Computer Engineering BsC. The application is constituted by a list of users, in which it is possible to add a new user, change an existing one, or delete it. It was developed using the languages PHP, HTML, and CSS (using the Bootstrap CSS framework).

The application was changed to be injectable, in figure 30 we can see executing of the SQL query with parameters passage without prepared statements [62], which makes the application injectable if the parameters are not sanitized. The sanitized parameters are when the user's input goes through methods that will remove any illegal character from the data.

5.2 TEST SCENARIOS

5.3 DVWA (DAMM VULNERABLE WEB APP)

DVWA is a free web application, implemented with PHP/MySQL, and is used to practice some of the most common web vulnerabilities, with various levels of difficulty, with a simple and direct interface, which will be used as an application that will be tested with a developed application. There is an image of the dock with a DVWA application, which will be used in the integration of the application with the CI/CD pipeline. DVWA application help web developers better understand the processes of securing web applications and to aid both students and teachers to learn about web application security in a controlled class room environment [6].

The test engine starts by taking the address of the application to be tested and view all the code on the page looking for HTML <a> tags to which the "href" attribute will be extracted and add to the endpoints to be tested, from the URL that is added extracts all the parameters of the query if they exist and adds them to the parameters of the endpoint, and also looks for all the HTML <form> tags look for the action attribute to validate if the URL is valid and then add it to the endpoint to test after, and also all inputs present in the form will extract the name from the input and add it to the parameters of the endpoint. This research is recursive, after this task end, all found endpoints are tested, in figure 31 we can view this process in the diagram.

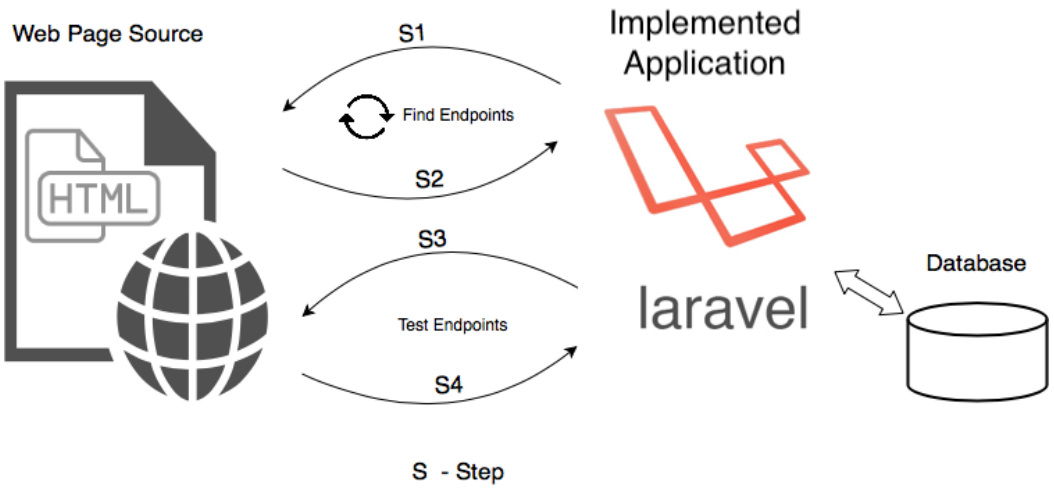


Figure 31: Testing Mechanism Diagram

5.4 TESTING MECHANISMS

At the moment, only one mechanism has been implemented, Boolean-based Blind SQLi. This attack is typically slow (especially on large databases) since an attacker would need to enumerate a database, character by character[45]. The effectiveness of data extraction with this technique depends on the number of requests made to the database[11]. In figure 9, it is demonstrated in a simple way how the testing mechanisms works.

5.5 MANAGEMENT SCHEDULING

In the Testing Application, a scheduling mechanism was implemented, so that users can run the tests, in more desirable periods of time, so as not to disturb the smooth functioning of the user's applications, and to support recurring tests of live applications.

This mechanism is implemented with the functionality available by the Laravel framework called "Task Scheduling" and for its operation, just need to add a new line in the server's crond, this command will execute the "schedule:run" command in the application folder every minute.

```
* * * * * cd / path-to-your-project \&\& php artisan schedule :
                                                run >> / dev / null 2> \& 1
```

This defines the scheduled task to run every minute at which time we check the database for tasks to run and run them.

In the figure 32, we can see one of the possible ways of creating a new task in the application, in which it is necessary to define the task name, the URL we want to test and the date on which we want the task to be executed, which can also be created with a POST request to the application API. In figure 33, it is demonstrated the creation of the task from a POST request, where the data are sent in the request and not filled in the platform form.

After creating the Task, the application will execute the task at the appropriate time and save the test result in the Database. The test results are accessible to the user that created them in the Tasks tab as we can see in figure 34. After the test run, when clicking on the task it is possible to view the test result, figure 35.

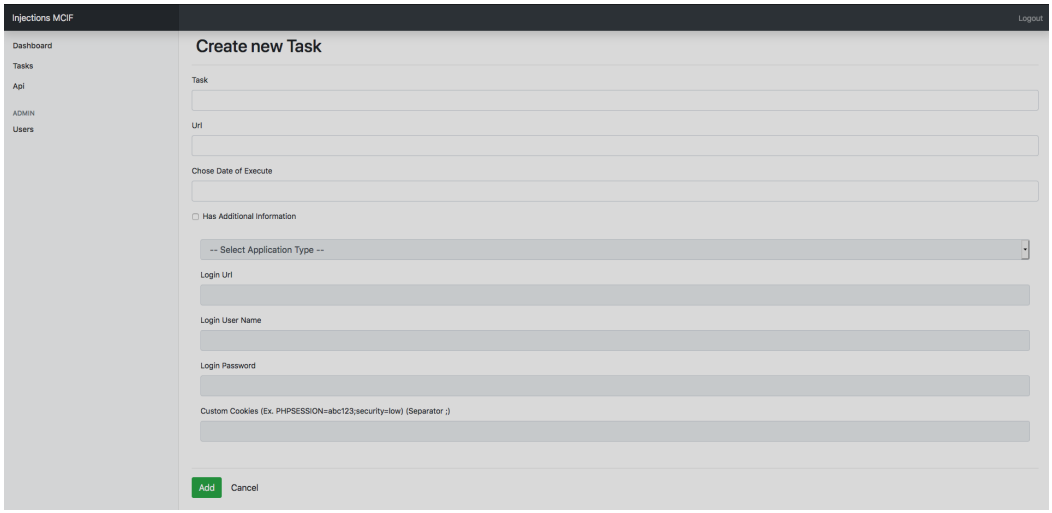


Figure 32: Create New Task

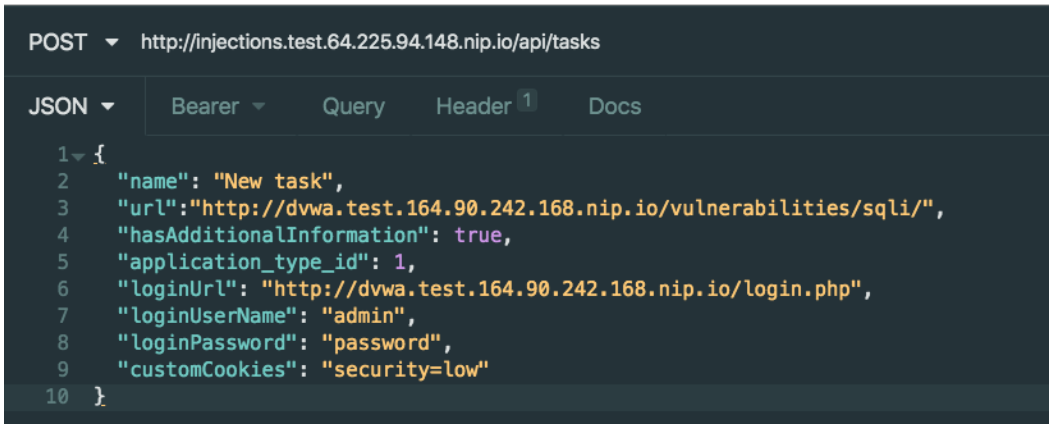


Figure 33: Create Task Api

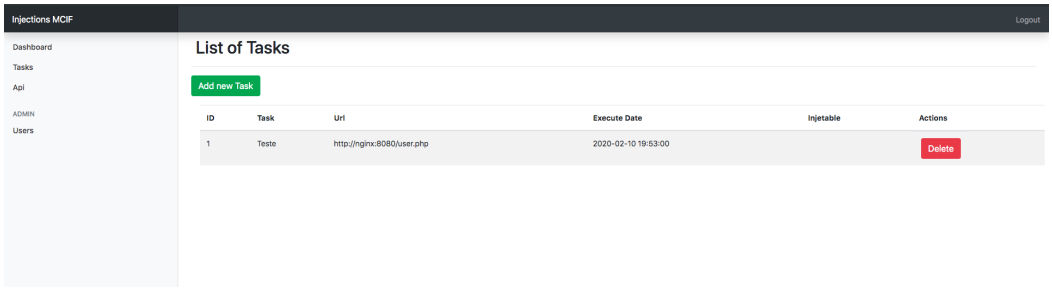
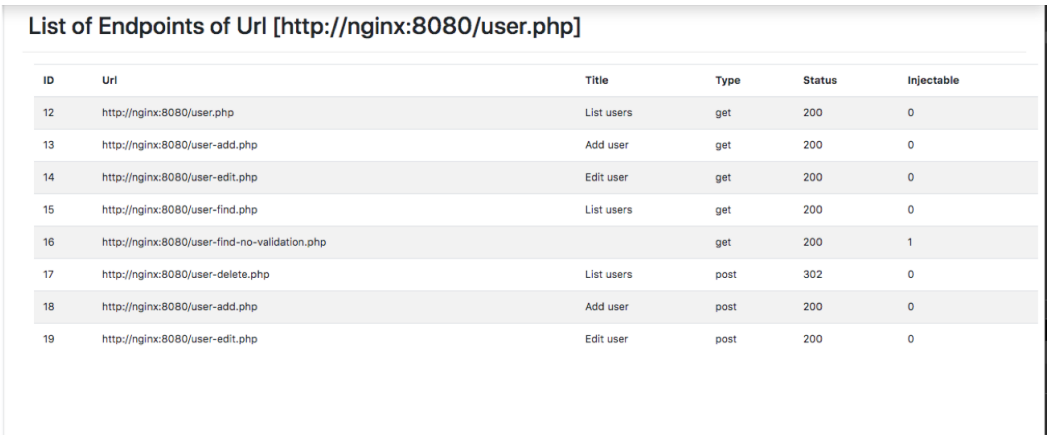


Figure 34: Manage Tasks



List of Endpoints of Url [http://nginx:8080/user.php]

ID	Url	Title	Type	Status	Injectable
12	http://nginx:8080/user.php	List users	get	200	0
13	http://nginx:8080/user-add.php	Add user	get	200	0
14	http://nginx:8080/user-edit.php	Edit user	get	200	0
15	http://nginx:8080/user-find.php	List users	get	200	0
16	http://nginx:8080/user-find-no-validation.php		get	200	1
17	http://nginx:8080/user-delete.php	List users	post	302	0
18	http://nginx:8080/user-add.php	Add user	post	200	0
19	http://nginx:8080/user-edit.php	Edit user	post	200	0

Figure 35: Test Results

6

CONCLUSION

This project was a journey in learning, experimentation, development, and infrastructure management, and although it had the focus on detecting SQL Injection vulnerabilities it provided an overview of a large part of the software development cycle. One key insight taken from this project is that the multiple aspects of software development should not live in isolation, security cannot live in it's walled garden, and neither can development or infrastructure.

As a result of the scope of this project, the first major benefit was the broadening of knowledge both in dept in some case as in breath on others. Understanding the internals of an SQL Injection vulnerability and it's detection techniques is much more powerful in the full context of the life of a piece of software.

The second result, but by no means less relevant, was the production of an application and it's surrounding processes that can be used as a model for bridging the gap between security and development or DevOps. Although the market has no shortage of security tools targeting the detection of software vulnerabilities, there seems to be a significant gap in this space. Most, if not all, tools have an underlying requirement of a big set of skills and knowledge. Our contribution to this space is a way to bridge this gap by providing a tool that can obfuscate that requirement and allow teams, that have experience and knowledge in other fields of the software development world, to reap the benefits of a security tool directly in their normal workflow.

As a personal note, this project allowed me to learn, what are Containers, DevOps, deepen my knowledge in SQL Injection, and know tools like Docker, Kubernetes, Terraform, Ansible, that some of them, I use every day now.

We conclude by suggesting that what we produced in this project could and should be augmented, not only on the security aspect, by implementing new testing techniques or new vulnerability targets, but also on the software development life cycle component by making it integrable with other types of mechanisms in the CI/CD space.

BIBLIOGRAPHY

[1] Acunetix. *Types of SQL Injection (SQLi)*. URL: https://www.acunetix.com/websitesecurity/sql-injection2/.

[2] *Acunetix Web Vulnerability Scanner*. Dec. 2020. URL: https://hakin9.org/acunetix-web-vulnerability-scanner/.

[3] Akamai. *Web Attack Visualization*. Mar. 2020. URL: https://www.akamai.com/uk/en/resources/our-thinking/state-of-the-internet-report/web-attack-visualization.jsp.

[4] *Ansible (software)*. URL: https://en.wikipedia.org/wiki/Ansible_(software).

[5] *Arachni Package Description*. Dec. 2020. URL: https://tools.kali.org/web-applications/arachni.

[6] avishek. *WHAT IS DVWA AND WHY ETHICAL HACKER LOVE THIS!* June 2018. URL: https://khannasecurity.com/blog/what-is-dvwa-and-why-ethical-hacker-love-this/.

[7] *BBQSQL*. Dec. 2020. URL: https://tools.kali.org/vulnerability-analysis/bbqsql.

[8] *Best Free and Open Source SQL Injection Tools*. Feb. 2019. URL: https://resources.infosecinstitute.com/topic/best-free-and-open-source-sql-injection-tools/.

[9] *Boolean-based blind SQL injection*. Dec. 2020. URL: https://www.rangeforce.com/blog/how-to-prevent-blind-sql-injection.

[10] Kevin Casey. *How to explain Kubernetes in plain English*. Sept. 2020. URL: https://enterprisersproject.com/article/2017/10/how-explain-kubernetes-plain-english.

[11] Alex Chapman. *Blind SQL injection optimization*. Jan. 2017. URL: https://ajxchapman.github.io/security/2017/01/14/blind-sql-injection.html.

[12] *Configuring Burp's Session Handling rules*. URL: https://portswigger.net/support/configuring-burp-suites-session-handling-rules.

[13] Daitan. *Enable Security into CI/CD pipeline with DevSecOps*. Dec. 2019. URL: https://medium.com/swlh/enable-security-into-ci-cd-pipeline-with-devsecops-9370c93d87a1.

[14] et al. David Stacy. *Practicing Continuous Integration and Continuous Delivery on AWS*. June 2017. URL: https://d0.awsstatic.com/whitepapers/DevOps/practicing-continuous-integration-continuous-delivery-on-AWS.pdf.

[15] *Developers bring their ideas to life with Docker*. URL: https://www.docker.com/why-docker.

[16] *DevOps*. URL: https://en.wikipedia.org/wiki/DevOps.

[17] *DigitalOcean*. Jan. 2016. URL: https://searchcloudcomputing.techtarget.com/definition/DigitalOcean.

[18] Michael Dowling. *Guzzle Documentation*. 2015. URL: https://docs.guzzlephp.org/en/stable/index.html.

[19] The OWASP Foundation. *The Ten Most Critical API Security Risks*. 2019. URL: https://raw.githubusercontent.com/OWASP/API-Security/master/2019/en/dist/owasp-api-security-top-10.pdf.

[20] The OWASP Foundation. *The Ten Most Critical Web Application Security Risks*. 2017. URL: https://www.owasp.org/images/b/b0/OWASP_Top_10_2017_RC2_Final.pdf.

[21] Somya Garg. *Automated Cloud Infrastructure, Continuous Integration and Continuous Delivery using Docker with Robust Container Security*. Feb. 2019. URL: https://www.researchgate.net/publication/331131851_Automated_Cloud_Infrastructure_Continuous_Integration_and_Continuous_Delivery_using_Docker_with_Robust_Container_Security.

[22] *Github Actions*. Dec. 2020. URL: https://github.com/features/actions.

[23] *Github Actions*. Dec. 2020. URL: https://www.freecodecamp.org/news/what-are-github-actions-and-how-can-you-automate-tests-and-slack-notifications/.

[24] *GitHub kubernetes/minikube*. URL: https://github.com/kubernetes/minikube.

[25] *GitLab CI/CD*. Dec. 2020. URL: https://docs.gitlab.com/ee/ci/.

[26] *GitLab CI/CD*. Dec. 2020. URL: https://www.lambdatest.com/blog/jenkins-vs-gitlab-ci-battle-of-ci-cd-tools/.

[27] *GoCD*. Dec. 2020. URL: https://www.gocd.org/.

[28] *GoLismero*. Dec. 2020. URL: https://github.com/golismero/golismero.

[29] *Grabber*. Dec. 2020. URL: https://tools.kali.org/web-applications/grabber.

[30] *Grabber Package Description*. Dec. 2020. URL: https://tools.kali.org/web-applications/grabber.

[31] *HTTP Client*. URL: https://laravel.com/docs/8.x/http-client.

[32] *Infrastructure as code*. URL: https://en.wikipedia.org/wiki/Infrastructure_as_code.

[33] *Introduction to Terraform*. URL: https://www.terraform.io/intro/index.html.

[34] *Jenkins*. Dec. 2020. URL: https://www.jenkins.io/.

[35] Henrique Madeira José Fonseca Marco Vieira. *Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks*. Jan. 2008. URL: https://www.researchgate.net/publication/4322871_Testing_and_Comparing_Web_Vulnerability_Scanning_Tools_for_SQL_Injection_and_XSS_Attacks.

[36] *jSQL Injection*. Dec. 2020. URL: https://tools.kali.org/vulnerability-analysis/jsql.

[37] *Kali*. URL: https://www.kali.org/.

[38] *Kubernetes*. URL: https://github.com/kubernetes/kubernetes.

[39] *Kubernetes*. URL: https://www.docker.com/products/kubernetes.

[40] Chandar Kumar. *12 Open Source Web Security Scanner to Find Vulnerabilities*. Jan. 2020. URL: https://geekflare.com/open-source-web-security-scanner/.

[41] *Laravel Framework*. URL: https://laravel.com/.

[42] Connor Leech. *Use Laravel Dusk, browser automation and PHP to programmatically surf the web*. Aug. 2018. URL: https://medium.com/employbl/use-laravel-dusk-browser-automation-and-php-to-programmatically-surf-the-web-7dc3b2232220.

[43] *ManageEngine Vulnerability Manager Plus*. Dec. 2020. URL: https://www.capterra.com/p/185510/ManageEngine-Vulnerability-Manager-Plus/.

[44] *MariaDB*. Dec. 2020. URL: https://mariadb.com/.

[45] Sonali Mishra. *SQL Injection Detection Using Machine Learning*. May 2019. URL: https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1727&context=etd_projects.

[46] BALAJI N. *SQLMAP-Detecting and Exploiting SQL Injection- A Detailed Explanation*. Dec. 2018. URL: https://gbhackers.com/sqlmap-detecting-exploiting-sql-injection.

[47] *Netsparker*. Dec. 2020. URL: https://www.netsparker.com/.

[48] *Netsparker Security Scanner*. Dec. 2020. URL: https://sourceforge.net/software/product/Netsparker/.

[49] *Nexpose*. Dec. 2020. URL: https://sectools.org/tool/nexpose/.

[50] *Nexpose Vulnerability Scanner*. Dec. 2020. URL: https://www.rapid7.com/products/nexpose/.

[51] *Nikto*. Dec. 2020. URL: https://github.com/sullo/nikto.

[52] *Nmap*. Dec. 2020. URL: https://nmap.org/.

[53] Rami J. Haddad Olajide Ojagbule Hayden Wimmer. "Vulnerability Analysis of Content Management Systems to SQL Injection Using SQLMAP". In: (2018).

[54] *OpenVAS*. Dec. 2020. URL: https://github.com/greenbone/openvas.

[55] *OWASP Top 10 Security Risks & Vulnerabilities*. Feb. 2020. URL: https://sucuri.net/guides/owasp-top-10-security-vulnerabilities-2020/.

[56] *OWASP ZAP*. Dec. 2020. URL: https://github.com/zaproxy/zaproxy.

[57] *OWASP-Xenotix-XSS-Exploit-Framework*. Dec. 2020. URL: https://github.com/ajinabraham/OWASP-Xenotix-XSS-Exploit-Framework.

[58] Razman Hakim Abdul Raman. "ENHANCEDAUTOMATED-SCRIPTING METHOD FOR IMPROVED MANAGEMENT OF SQL INJECTION PENETRATION TESTS ON A LARGE SCALE". In: (2019).

[59] RedHat. *What is CI/CD?* URL: https://www.redhat.com/en/topics/devops/what-is-ci-cd.

[60] Margaret Rouse. *The Benefits of DigitalOcean – Review*. URL: https://miloszkrasinski.com/the-benefits-of-digitalocean/.

[61] *Security Spending and Cost Statistics*. Nov. 2020. URL: https://www.varonis.com/blog/cybersecurity-statistics/.

[62] Kevin Smith. *Protect Your PHP Application from SQL Injection*. Apr. 2018. URL: https://kevinsmith.io/protect-your-php-application-from-sql-injection.

[63] *SQL Injection*. Dec. 2020. URL: https://owasp.org/www-community/attacks/SQL_Injection.

[64] *SQL injection*. Dec. 2020. URL: https://portswigger.net/web-security/sql-injection.

[65] *SQLiv*. Dec. 2020. URL: https://github.com/the-robot/sqliv.

[66] *SQLMAP*. Dec. 2020. URL: http://sqlmap.org/.

[67] Andrew Starr-Bochicchio. *GitHub Actions for DigitalOcean*. June 2020. URL: https://github.com/digitalocean/action-doctl/blob/v2/README.md.

[68] *Terraform (software)*. URL: https://en.wikipedia.org/wiki/Terraform_(software).

[69] *Top 10 Most Common Software Vulnerabilities*. July 2020. URL: https://www.perforce.com/blog/kw/common-software-vulnerabilities.

[70] *Top 15 Paid and Free Vulnerability Scanner Tools in 2020*. Jan. 2020. URL: https://www.dnsstuff.com/network-vulnerability-scanner.

[71] *Travis CI*. Dec. 2020. URL: https://travis-ci.com/.

[72] *Using Minikube to Create a Cluster*. Oct. 2020. URL: https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/.

[73] *W3af*. Dec. 2020. URL: http://w3af.org/.

[74] *Web Application Security with Acunetix*. Dec. 2020. URL: https://www.acunetix.com/vulnerability-scanner/web-application-security/.

[75] *What is a Container?* URL: https://www.docker.com/resources/what-container.

[76] *What is DevOps?* URL: https://www.atlassian.com/devops/what-is-devops.

[77] *What is Docker?* URL: https://opensource.com/resources/what-docker.

[78] *XssPy - Web Application XSS Scanner*. Dec. 2020. URL: https://github.com/faizann24/XssPy.