Bachelor's thesis

Double degree ECS network – Embedded Software.

2019.

Rodrigo Baranda Castrillo

RESEARCH AND IMPLEMENTATION OF CYBERSECURITY TECHNIQUES IN THE BACKEND OF A WEB APPLICATION

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Double degree ECS network – Embedded Software.

2019 | 54, 0

Rodrigo Baranda Castrillo

RESEARCH AND IMPLEMENTATION OF CYBERSECURITY TECHNIQUES IN THE BACKEND OF A WEB APPLICATION.

The purpose of this Thesis was to eliminate the vulnerabilities that threaten the database of a web application and to research and document potential improvements that provide greater security to the web application. This thesis provides a theoretical analysis of each of the possible techniques that eliminate these threats to the system culminating with the implementation of these techniques. Besides, it has been done a research work on possible additions that improve the security of the system. This work was carried out to analyze the best security techniques to be implemented on a web application, in addition to improving the analysis, programming and cybersecurity knowledge of the author.

The methodology fort his thesis can be summarized as follows: study of the threat in question, assessment of its degree of criticality, study of possible implementation techniques to eliminate it, analysis of which of these techniques fit the system best, and (in many cases) implementation of this technique. The main results have been, on the one hand, the elimination of threats such as Clickjacking, SQL Injection Attack, Application Error Disclosure and, onn the other hand, the documentation that results from the research work of the system and that can serve to improve the backend security of the web application. The significance of the results will allow the reader to understand the relevance of the threats to which a web application is subjected, and also why one defense technique is better than another and how to implement it. In addition, the thesis presents the results of an investigation about how to improve the database , which can be applied to improve the security of any other database.

KEYWORDS:

Database, Vulnerabilities, Threats, Backend.

CONTENTS

PICTURES

TABLES

LIST OF ABBREVIATIONS

CSRF	Cross-Site Request Forgery
DB	DataBase
OOP	Object Oriented Programming
SQL	Structured Query Language
SQLIA	Structured Query Language Injection Attack
XFO	X-FRAME-OPTIONS

1 INTRODUCTION

This thesis seeks to eliminate the vulnerabilities linked to the backend of a web project, either by implementing the appropriate techniques or developing documentation that could solve either existing or potential problems.

First, before beginning to explain what the work of this thesis specifically consists of, it is important to make a brief contextualization about the project in which it has been commissioned for: the FINCODA survey system, a project by Turku University of Applied Sciences. The FINCODA web project, citing the words of the web itself, aims to develop "… a tool for measuring learning outcomes in internal development activities for universities and working life organizations." *(see https://fincoda.dc.turkuamk.fi/)*

Taking into account that it is a web project, two fundamental parts can be distinguished: frontend and backend. After a study conducted on the project, numerous security vulnerabilities were discovered. This made that the fundamental role that has been played as a member of the project has always been related to the Analysis of Website Security Techniques. The work done has had two ways: (I) implementing these Security Techniques or (II) making the documentation and analysis work necessary before the implementation of these techniques, always focusing mainly on the part near the backend.

Therefore, in relation to how the idea of this thesis arose, it has to be takeinto account that the development of the skills of the author has always focused more on the part of the backend (having worked basically with OOP) and that there was already a background in relation to computer security.If that is combined with the existence of this web application with vulnerabilities in the backend, in the end, the idea of carrying out this thesis arose. It allowed him to personally improve his knowledge about cybersecurity, in addition to specializing in a new language for him as *PHP* and, from the point in view of the project, to finalize or minimize such security problems.

On the one hand, in reference to the literature or studies that can be found regarding the security or protection against vulnerabilities of a system on the web, there are multiple studies on this, practically for each technique that is used, there is a bibliography on the subject, with different modes of implementation, valuing the possible advantages and each of these modes of implementation. On the other hand, it should also be taken into account that when implementing certain security techniques, most of the existing sources are not of an official nature.

The fundamental purpose of developing this thesis has been to enhance whose knowledge in the field of computer security because although the author had never specifically specialized in it before, he did have the basic characteristics and background to do it. Along this path, he has been able to develop his skills as a programmer, learning a completely new language for him (as it has been mentioned before); he has been able to know firsthand the importance of the use of frameworks (*Laravel* in this case); and the importance of prior documentation to perform a job, which allowed him to become fully aware of what he wanted to do and how he wanted to do it, so that the result is the best possible. Therefore, briefly, it could be said that the importance and relevance of this thesis not only has to do with the learning of different

cybersecurity techniques and how to implement them but also the empowerment of the critical capacity to decide which way is the best to develop a task. For this, the objectives were to develop his skills as a programmer, as an analyst and also as a teammate because he was part of a project team.

At the time of developing his work in the thesis, it is true that some of the tasks performed have been of general purposes such as updating the documentation relating to the installation of the project, which can be easily understood by a general public. However, it is also true that some of the techniques to grant the web site safety may be out of the scope of the knowledge of the general public. In the case that general public wants to go into implementation details to try to understand how it has been done, most likely they could not understand it because it might be out of limits for outsiders, although there is a theoretical explanation in every chapter directed to be understood by any reader.

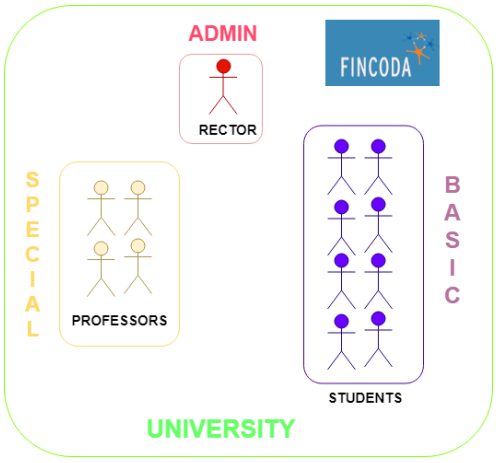
Throughout the following chapters, the different steps that have been taken throughout the project will be explained, which may consist either of pure research and documentation. It is necessary to understand the threats faced by the web application, to provide a necessary background to implement new security techniques in the future (e.g., analysis of how the database could be modified according to the type of user and their permissions); or in a more technical way of how these security techniques have been implemented.

2 BRIEF DESCRIPTION OF THE FINCODA PROJECT

In Chapter 1*)* it was briefly mentioned what the FINCODA project is about. Throughout this chapter, a general overview of the project will be made to contextualize the reader with the project.

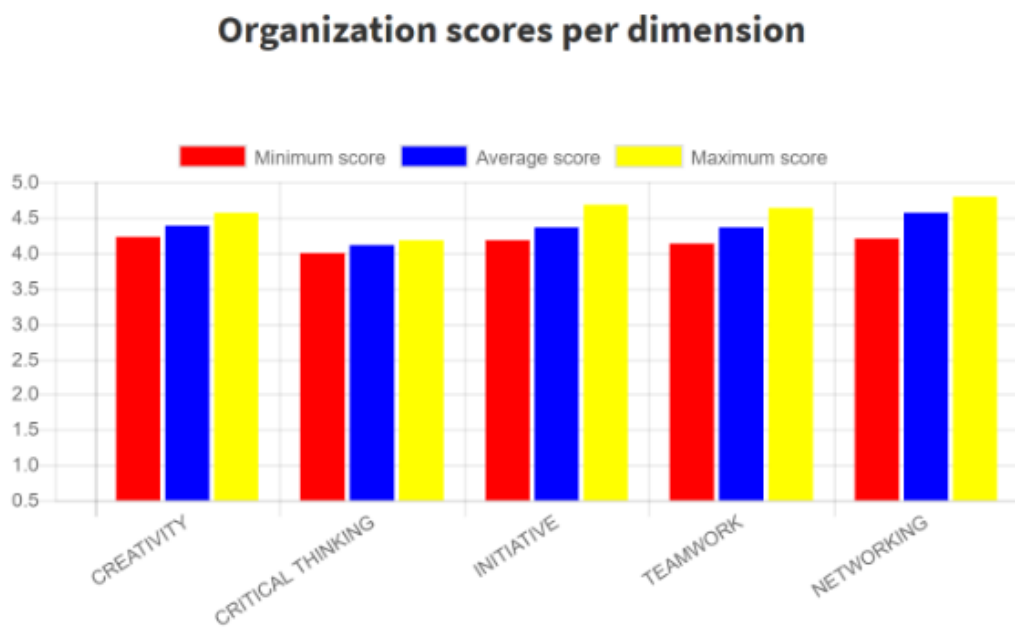
The FINCODA project is a web application developed by the Turku University of Applied Sciences. The purpose of this web application is to serve as a tool for universities or any other type of organization to measure the innovation competencies of the members of the organization. Some of the competencies that are evaluated are the following: Creativity, Critical Thinking, Initiative, Teamwork, and Networking. TThese innovation competencies are measured through user surveys.

For example, a common use of the FINCODA web application is as follows: a university decides to evaluate the innovation competencies of its students. Therefore, the head of the university (e.g., the rector) registers the company in the web application FINCODA, being the administrator thereof; teachers are special users, able to create or delete surveys, which will typically be completed by students (although they could also be completed by teachers); and, finally, the students are those who will complete the aforementioned surveys, obtaining, as a result, an assessment of their innovation competencies. See the picture below:



Picture 1: Common users? of the FINCODA project.

The results of these surveys are represented by graphs like the one in Picture 2.

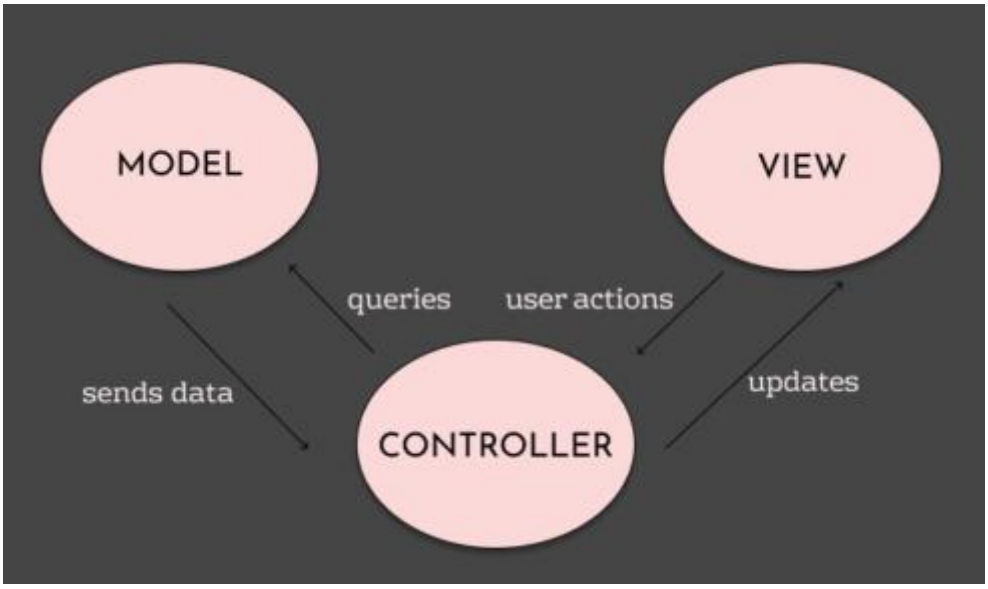


Picture 2: Example of the output obtained after the survey is done.

After this brief introduction on how the web application works, the following paragraphs will explain the technical structure of this web application.

The FINCODA web application is used as a Web Server Environment *WampServer* or *Xampp* (typically WampServer), with *PHPMyAdmin* as the database server. A version of *PHP* 5.5.9 - 7.1. * Is needed, and the framework used is *Laravel* in version 5.2.

As has been mentioned previously, the work carried out has been mainly focused on working on the backend. In order to make this more understandable to the reader, the Model-View-Controller structure will be explained below, since it is the one used by the *Laravel* Framework. In Picture 3 (Ighodaro, 2018), an outline of the operation of the Model-View-Controller architecture can be seen.



Picture 3: How Laravel implements MVC and how to use it effectively. Retrieved from https://blog.pusher.com/laravel-mvc-use/ Ighodaro, N. (2018).

Taking into account that usually a web application is divided into two parts: *frontend* and *backend*; and that in the Model-View-Controller architecture, the View is directly associated with the frontend, and the Model and Controller with the backend, the work on the application has been fundamentally carried out on the Model and the Controller. From Picture 3, it can be seen, briefly, on the one hand, that the Controller executes queries on the Model, which returns data to the Controller; and, on the other hand, that the user performs actions on the View, whose results arrive at the Controller, which gives the order to update the View of the web application.

With the explanation in this chapter, a general idea has already been given about the operation of the project (from the point of view of the experience of a user of the application), and the structure of it has been explained. The work has been developed on this structure.

3 INSTALLING THE PROJECT.

In the introductory chapter, it was indicated that the main task as a member of the project was to eliminate the problems related to the safety of the project, but, in addition to this, he also had to carry out a documentation task to prevent potential errors.

Due to this, the first task he did as a member of the FINCODA project was to update the installation report. In this installation report, there was some information missing needed to install the project properly. In addition, some of the information contained was either erroneous or incomplete, which could lead to an error when installing it.

Next, it will be made a brief tour of the steps necessary to install the project, with special emphasis on the errors that existed previously and how they were solved.

3.1 Cloning the repository.

As it was mentioned earlier, the FINCODA project is a web project. Because of this, a Web Server Environment must be used, typically XAMPP or WAMP. Therefore, the web server root directory will vary between WAMP and XAMPP: See Picture 4.

```
C:\wamp\www\FINCODA
```

Picture 4: Root directory in WampServer.

See Picture 5.

```
C:\wamp\htdocs\FINCODA
```

Picture 5: Root directory in XamppServer.

It will be inside this server root directory where the project code should be cloned. This information in the initial installation example could lead to misinterpretation because, in the example, the project was cloned in a folder like (See Picture 6):

```
C:\users\username\Documents\FINCODA
```

Picture 6: Example in which the project was cloned initially.

In order to make the installation manual more user-friendly (because this previous information could lead to error or make installation difficult), two screenshots containing the correct path were added.

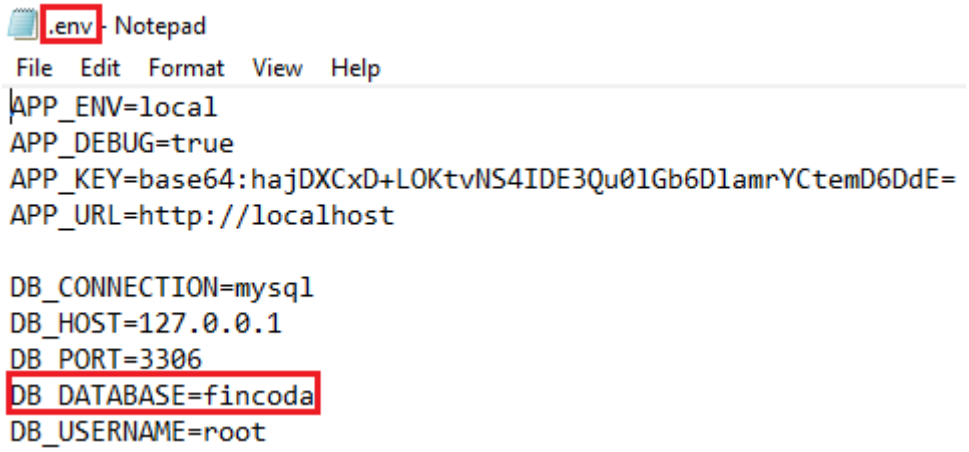
3.2 Managing dependencies.

Another step to install the project has to do with the installation of Composer, node.js, and PHP. In the current installation manual, it was not specified that, for the correct operation of the project, a version of PHP between *5.5.9 - 7.1.** or higher is needed. This is due to the fact that in the web project, the PHP Framework, Laravel, is being used in its version 5.2, which necessarily requires that the PHP version used is between that values (Otwell, 2016).

3.3 Setting up the database.

At the time of the installation and configuration of the database, it was found that there was erroneous information that, if not modified, led to an error when trying to create the database. What happened was that do two steps for creation needed to be done: first, create a database in *phpMyAdmin*, whose name should be 'finn' and then import the *SQL* script with all the information related to it.

However, after creating this database by following the steps above, an error was always obtained. Finally and after a lot of tests, it was discovered that in the '*.env'* file, there was the correct name of the database and that it did not agree with the one that was being used. Although the name of the script was *'finn.sql'*, and in the previous documentation it was specified that the name of the database should be *'finn'*, this led to error, because in this *.env* file, it is specified that the name of the database must be 'fincoda'. See Picture 7.



Picture 7: Proper name of the DB found in the .env file.

3.4 Editing the project config files.

It is at this point that the *.env* file must be created with the environment variables of the system. Simply note that at the time of making this step the previous documentation was perfectly done and that, this was useful to discover the name that should have the database,

because, as it was mentioned earlier, in this .env file, among other things, the name of it can be found.

3.5 Installing all dependencies.

At this point, all project dependencies must be installed. Previously, in the point of Managing Dependencies *(see Managing dependencies.)*, it was mentioned that both Composer and Node.js were installed, as well as *PHP 5.5.9 - 7.1. *.* However, at that point, the software was being downloaded, but now this should be applied to the project, typically executing the *composer install* command in the folder in which the project is located. See Picture 8.

```
PS C:\wamp64\www\FINCODA> composer install
```

Picture 8: Installing the Dependency Management for PHP, Composer.

It is at this point also, where *PHP's Laravel* 5.2 framework must be created. This point of creating the *Laravel* Framework, for example, was not included in the original documentation. When installing *Laravel*, since the requirements of the *PHP* version are those that were specified previously, if the *PHP* version installed in the system does not meet them, certain problems will occur when executing the project. See Picture 9.

```
PS C:\wamp64\www\FINCODA> laravel new
```

Picture 9: Creating the Laravel framework in the project.

3.6 Migrate the database and seeding.

This last point in relation to the creation of the project contains a fundamental step, which is migrating the database and seeding it with initial values. "Migrations are like version control for your database, allowing a team to easily modify and share the application's database schema" (Otwell, 2016). It should be noted that the steps that must be taken to carry out this process properly were perfectly explained in the original documentation.

3.7 Running the project.

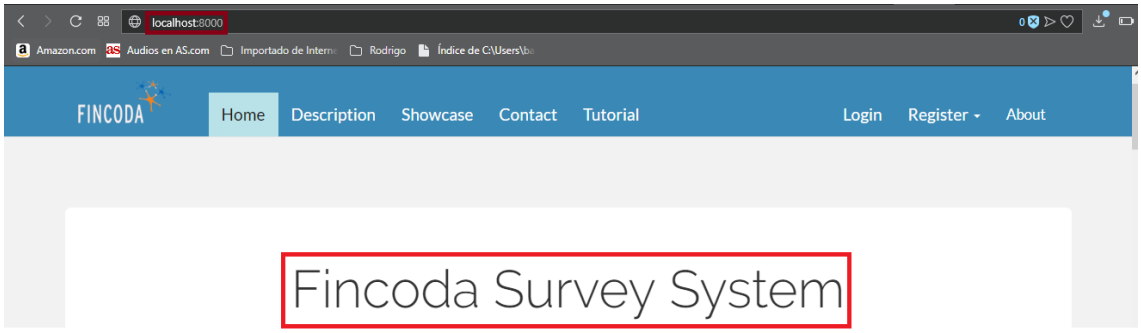
At the end of this installation process, in the original documentation, there was no section directed to the execution of the project. So, to make this project execution smoother and easier for the user, a new section with the basic points for its execution was added.

First, the server used by the user (typically Wampp or Xampp) must be running. Subsequently, within the project folder, we must execute the following command (See Picture 10):

```
PS C:\wamp64\www\FINCODA> php artisan serve
Laravel development server started on http://localhost:8000/
```

Picture 10: Command needed to execute the project.

Finally, if all the steps explained above have been done correctly when opening the localhost in any web browser, the web project should be viewed correctly. See Picture 11.



Picture 11: Screenshot of the project executing.

4 PREVENTION AGAINST CLICKJACKING.

One of the different vulnerabilities that were found in the Client Site, was the risk of Clickjacking (click to hijack), for which there was no security technique implemented. In the first place, he considers it important, before explaining the technique implemented to prevent Clickjacking, to contextualize and try to explain what this malicious technique consists of since it is possible that this question is out of the limits for outsiders in the security topic.

4.1 What is Clickjacking?

Clickjacking is all about visual deception. It is a malicious technique that visually deceives the user into clicking on something different than what is perceived using iframes to hijack the web session of a user. An attacker uses a transparent, invisible iframe over an authentic web page and then allures the user to operate on that page. The users are led to think that they are clicking on the authentic page, but they are actually unwittingly clicking on the hidden page (interacting with the victim site). Therefore, the users are tricked into performing actions that they never intended (Wu and Zhao, 2015). See Picture 12.

Picture 12: Diagram of Clickjacking. *Web Security: A WhiteHat Perspective*. Wu and Zhao. (2015).

By changing the length, width, position, and opacity, among other design values, the attacker can make this iframe completely invisible to the user of the web. The moment the attacker achieves this, it can be said that the page is completely clickjacked.

Therefore, in case that a web application is exposed to clickjacking risk without any protection, the user could complete a chain of clicks without being aware of what their real implications might be. Some of these implications could be: download malware, visit malicious web pages, provide credentials or sensitive information, transfer money or purchase products online among others.

4.2 Different ways to prevent Clickjacking.

When trying to mitigate Clickjacking, two possible ways to do it were taken into account: Client-side methods and Server-side methods.

4.2.1 Client-Side method – *Frame Busting*.

Within the Client-side methods, the idea of introducing the Frame Busting technique was valued. To defend the web against clickjacking attacks, the following simple frame busting code is commonly used by web sites: See Picture 13.

```
if (top.location != location)
  top.location = self.location;
```

Picture 13: Example of frame busting code.

Frame busting code typically consists of a *conditional statement* to test for framing, followed by a *counter-action* that navigates the top page to the correct place if framing is detected. This basic code is fairly easy to bypass. Below are some typical examples of the Frame Busting conditional statement and its respective Counter-action statement (Rydstedt et al., 2010) See Table 1

Table 1. Conditional frame busting examples. *Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites* Rydstedt et al., (2010).

Conditional Statement
if (top != self)
if (top.location != self.location)
if (top.location != location)
if (parent.frames.length > 0)

See -Table 2:

Table 2. Counter-action statements examples. *Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites* Rydstedt et al., (2010).

Counter-action

top.location = self.location
top.location.href = document.location.href
top.location.href = self.location.href
top.location.replace(self.location)
top.location.href = window.location.href
top.location.replace(document.location)

However, while it is true that Frame Busting has been used as a defense technique against Clickjacking, this defense technique can be *easily* circumvented in one way or another. Moreover, taking into account that the majority of web attacks are generic, which means that they can be used against a large number of websites without requiring extra work for the attacker. Over the last few years, the best defensive technique against Clickjacking that has been implemented in web browsers has been the incorporation of *X-FRAME-OPTIONS (XFO)*.

Referring to the conclusion of a study carried out by members of Stanford University, which reads as follows: "After reviewing the available defenses, we propose to JavaScript-based defense to use until browser support for a solution such as *XFO* is widely deployed." (Rydstedt et al., 2010), and knowing that this study dates back to 2010 and that *XFO* are widely spread today, it was concluded that the most complete and simple solution was the implementation of *XFO*.

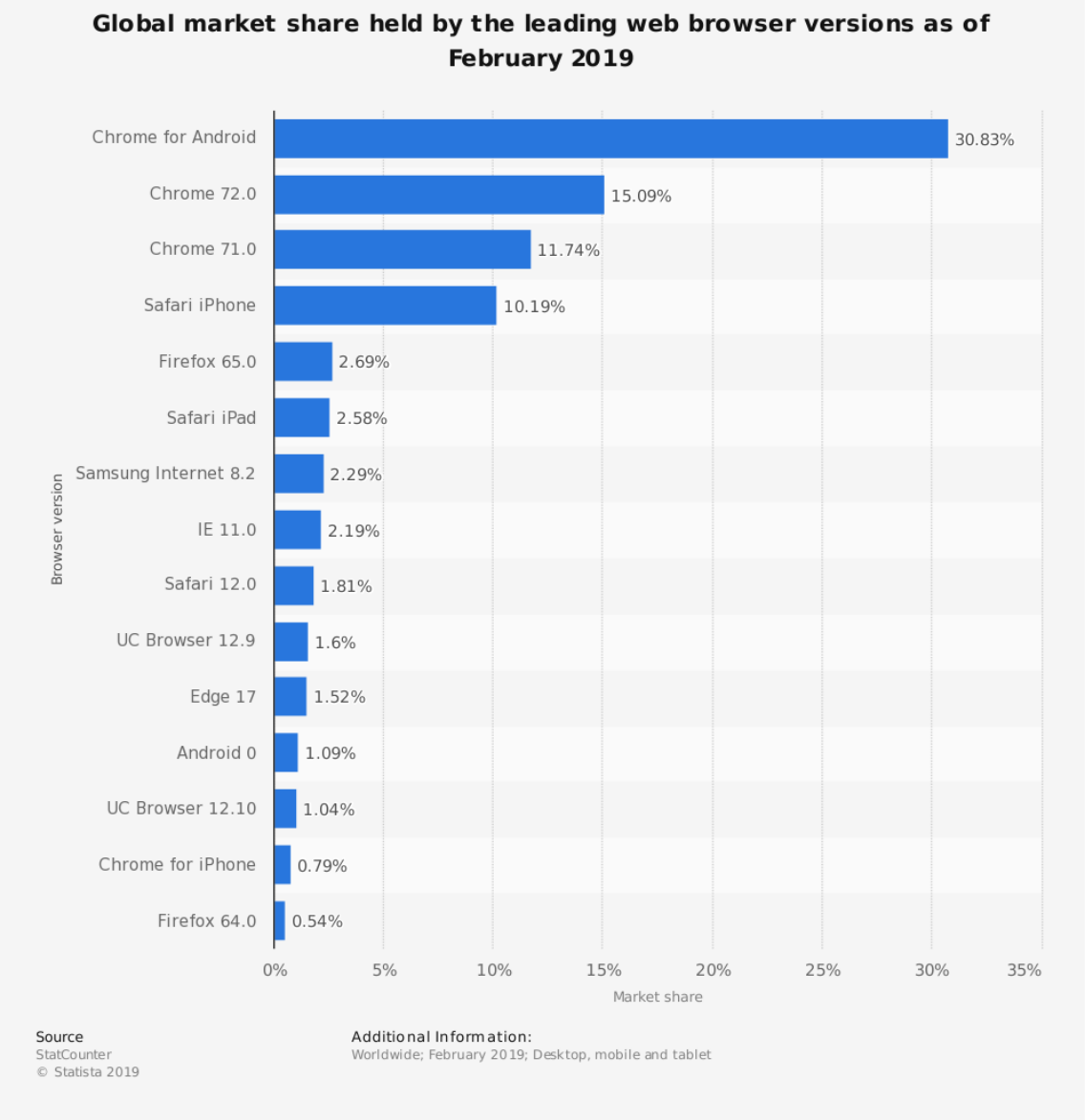
4.2.2 Server-side method - *X-FRAME-OPTIONS*.

When it has been previously talked about *(see)* Client-Side method – *Frame Busting*., it was explained that it was the programmer himself who should manually defend the page of being framed. However, in this case, where the protection is on the Server side, the website itself can be protected from being framed, being the browser the one that deals with the problem, not the programmer directly.

It is necessary to know that, the implementation of the XFO, consists of the implementation of an HTTP Security Header. These HTTP Security Headers what they do is adding an extra layer of security, which facilitates the mitigation of attacks and security vulnerabilities, telling the browser how to behave when handling the content of the site, by not allowing rendering of a page in a frame.

Taking the above into account, and seeing that in this case, it is the browser the one who decides how to behave when handling the content of the site, it can be easily understood that the X-Frame-Options Response Header does provide clickjacking protection by not allowing iframes to load on your site. It is supported by IE 8+, Chrome 4.1+, Firefox 3.6.9+, Opera 10.5+, Safari 4+ (Jackson, 2018).

As it was mentioned above, one of the reasons why it was going to be used the X-FRAME-OPTIONS protection against Clickjacking, was because this HTTP Security Header was already widespread in all browsers currently. As can be seen, this HTTP layer is available in IE 8+, Chrome 4.1+ (and the others mentioned in the paragraph above). In the following image, browsers can be seen along with their version, most used as of February 2019. See Picture 14.



Picture 14: *Global market share held by the leading web browser versions as of February 2019*. In Statista - The Statistics Portal. StatCounter. (n.d.).

As can be seen, the version of IE used is 11.0 (higher than IE 8+), the Chrome version is 72.0 (higher than 4.1+), among others. That is, all current browsers have the necessary HTTP header

to protect against Clickjacking, so, therefore, it has been decided to use this technique as a defense for this threat.

4.3 Implementing the X-FRAME-OPTIONS Header.

When implementing this defense mechanism in Laravel, the technique used has been through a Middleware. A Middleware provides a convenient mechanism to filter HTTP requests in the web application.

The X-FRAME-OPTIONS can be implemented with one of the following three directives: DENY, SAMEORIGIN or ALLOW-FROM uri Directive.

4.3.1 DENY Directive.

The DENY directive completely disables the loading of the page in a frame, regardless of what site is trying. This might be a great way to lock down the site, but it will also break a lot of functionality (Jackshon, 2016). Due to this possible potential problem that could limit the functionality of the web, the use of this directive was discarded.

4.3.2 ALLOW-FROM uri Directive.

The ALLOW-FROM uri directive allows the page to only be loaded in a frame on the specified origin and or domain. This allows you to lock down your site to only trusted origins (Jackshon, 2016).

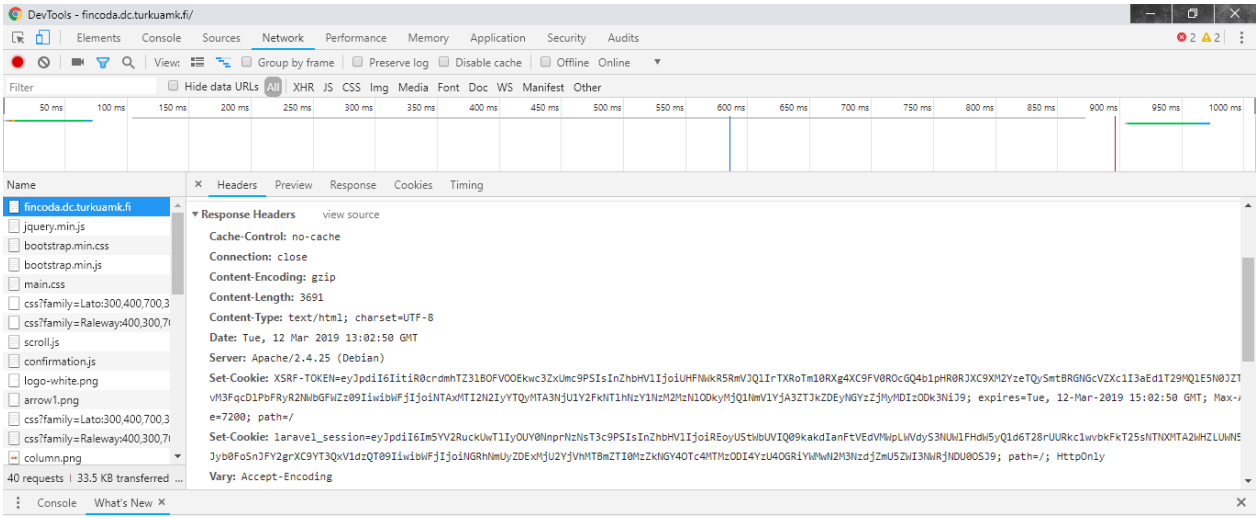
When assessing whether to use this directive, whose idea of operation is similar to that of a White-List with a series of links enabled, this had to be discarded because it is not enabled in search engines like Chrome.

4.3.3 SAMEORIGIN Directive.

The SAMEORIGIN directive allows the page to be loaded in a frame on the same origin as the page itself. With this directive, the page can be used in a frame as long as the site including it in a frame is the same as the one serving the page. This is probably the most commonly used directive out of the three. It is a good balance between functionality and security. (Jackshon, 2016).

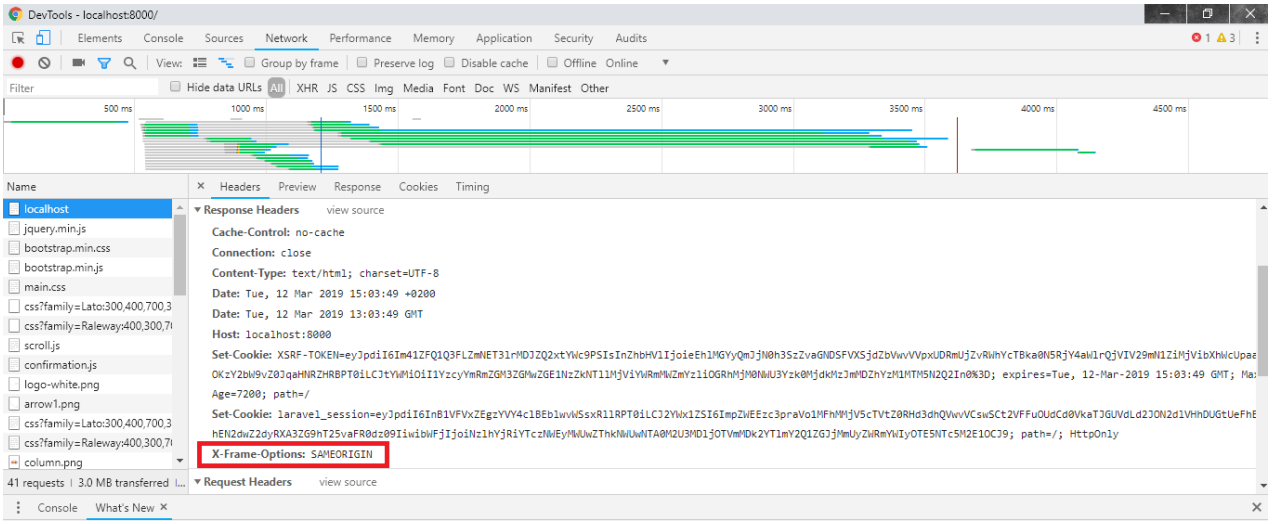
Because of this, the directive chosen to use the HTTP Header to prevent Clickjacking has been SAMEORIGIN.

Below, it is attached an image of how the web looked before the implementation of this technique: See Picture 15.



Picture 15: There is not any X-FRAME-OPTIONS Response Header.

After the implementation of the security technique, which has been implemented by creating a Middleware that processes each HTTP request and includes this Header each time it is passed from one page to another within the web. The Response Header that only Allows the programmer to attach iframes from the same origin of FINCODA can be seen in the next picture: See Picture 16.



Picture 16: X-FRAME-OPTIONS Header Implemented.

With this header included, it can be concluded that the web has already been protected against the possible attack of Clickjacking.

5 POSSIBLE PREVENTION AGAINST CROSS-SITE REQUEST FORGERY.

In the previous chapter, it was talked about a widely exploited web site vulnerability towards which the web was not protected: Clickjacking. *(see Prevention against clickjacking.* The objective of this chapter is to document one of the threats to which the system is currently exposed and how it could be solved with an update of the framework used. It is necessary to point out that the members of the project are evaluating the update to a new version of Laravel, whose latest stable version is 5.8. One of the security threats that would be solved with this update would be prevention against a CSRF attack.

Cross-Site Request Forgery (CSRF or sometimes also abbreviated as XSRF) is another widely exploited web site vulnerability and that does not have any type of protection implemented, and on which this chapter will focus. Again, as it was done in the chapter on Clickjacking, it is important to carry out a contextualization work, because this web vulnerability can be outside the limits for outsiders in the web security topic.

5.1 What is a CSRF attack?

A cross-site request forgery is a type of internet attack in which a hacker tricks a victim, taking advantage of implicit authentication mechanisms of the HTTP protocol and cached credentials in the browser to execute a sensitive action on a target website behalf of an authenticated user without his knowledge (Lalia et al., 2019).

If the success of a CSRF attack is consummated, the consequences of this could be such as damaged client-web relationship, unauthorized money transfers, changing passwords or data theft, among others. The mode of operation of a CSRF attack consists in deceiving the user or victim so that he sends a forged request to a server without his knowledge. Knowing that at the exact moment in which the attack is made, the unsuspecting user is authenticated in the application, it is impossible to distinguish between a legitimate user or a false user.

To get an idea of how dangerous this attack can be, although there are still not too many security measures against him, it is worth mentioning that he has been included in the *top 10 of the OWASP* list (Open Web Application Security Project) several consecutive times.

5.2 The fundamental role of cookies in a CSRF attack.

As the reader may already know, cookies are pieces of information generated by a website, sent to a user, which are stored in the browser of the user while the user is browsing the network. They are used by websites to identify a user, so once the user has logged into the web server, the browser will get an identity login cookie to remember the logged in status of the user. Later, when the user is searching through the different pages of that website, the browser will automatically set that identity login cookie in the requests. It must be known that,

until the browser is closed, that login cookie will remain, which will give a time margin to the attacker that will be used to make the user's browser perform certain actions not desired by the user.

The difficulty for detection and prevention of CSRF attacks, as Jeremiah pointed out is: “To Web servers, one request looks more or less just like another. The unintended request is legitimate (not hacked up), the user is the right user (authenticated), and the request is being made directly to the real Web site (no man-in-the-middle). The only problem is the victim did not intend to make the request, but the Web server does not know that” (Grossman, 2006)

5.3 Defending the web site against CSRF attack – Laravel.

The Laravel 5.8 framework version (note that the current Laravel version used is the 5.2), the one that is being thought to update to, includes in the *Middleware web*, the defensive implementation against CSRF, assigning a Token to each active user session on the web. The way in which Laravel protects the web from suffering a CSRF attack is generating a CSRF token for each active user session managed by the application. This token is used to verify that the already authenticated user is the one making the request to the application. The idea of the implementation is that, in every HTML form of the application, it should be included a hidden CSRF token (or XSRF token, the name does not matter) in the form so that the CSRF protection middleware can validate the request.

In the next picture, which is an example of a regular HTML Post method, but in which a *csrf* token is included, can be seen the idea of how it works: in every Post request, it is included the token of the session of the user. See Picture 17.

```
<form method="POST" action="/profile">
    @csrf
    ...
</form>
```

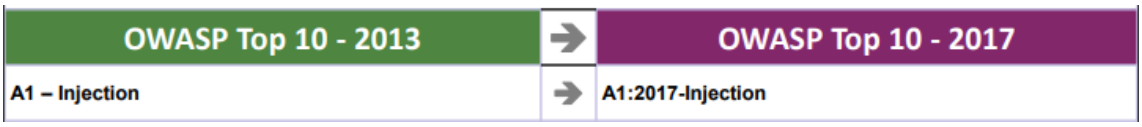
Picture 17: *CSRF Protection – Laravel*. Example of a POST method including the CSRF token. Otwell, T. (2019).

In the 5.8 Laravel version, there is a *VerifyCrsfToken middleware*, which is included in the *web* middleware group, that automatically verifies that the token in the request input matches the token stored in the session.

6 PROTECTING AGAINST SQL-INJECTION.

Throughout this chapter, one of the vulnerabilities to which the system is exposed will be studied: SQL-Injection. An analysis from the theoretical point of view of what this threat implies will be done, its possible solutions will be considered and the chosen implementation technique will be explained.

To be aware of the relevance of this threat, it should be noted that the threat of Injection is ranked number 1 of the top 10 of the list OWASP Top 10 Web Application Security Risks and that it has been in the first position since 2013. See Picture 18.



Picture 18: (OWASP Top 10 - 2017 The Ten Most Critical Web Application Security Risks, 2017)

6.1 What is an Injection Attack?

It must be known that one of the most common and most dangerous types of attacks in the field of web security are injection attacks. In addition, it is also known that web applications need to interact with the information belonging to the backend to extract persistent data, which is presented to the user as a dynamically generated output. This interaction is commonly done by dynamically constructing query strings within a general-purpose programming language. Once this interaction between the data entered by the user and the information presented, taken from the backend of the server is clear, the nature of an injection attack can be explained.

The nature of an injection attack is that the data entered by the user must be used to execute the code. From here, two key points should be considered: the first is that the user controls the data he enters as input; the second is that the code of the program to be executed must be executed by joining this information provided by the user (Wu and Zhao, 2015). Therefore, being aware that this information introduced by the user will be used as part of the code in the execution of the program, it can be understood that the injection attack is one of the most dangerous for web applications.

6.2 What is an SQL Injection attack?

Although in the previous section it was tried to explain what an injection attack was *(What is an Injection Attack?)* from a general perspective, this section will try to explain in particular what an SQL injection attack is.

The SQL injection is a kind of injection attack that makes it possible to execute malicious SQL statements. These statements are intended to attack the database server behind the web

application, such as MySQL (the server used in the FINCODA project), Oracle, or the server used in the web application. Once the attacker is able to bypass the security measures, he will be able to add, modify or delete the records of the DB. As it can be imagined, this sensitive data could be such as customer information, personal data or trade secrets among others. Next, it will be explained, step by step, how an SQL injection attack could affect a database in case it succeeds.

Imagine a form, in which the user is asked to enter his last name. See Picture 19.

```
var lastname;
var query;

username = Request.form("Enter your lastname: ");
query = "SELECT * FROM users WHERE lastname = '" + lastname + "'";
```

Picture 19: Example of a form that may receive an SQL Injection attack.

For example, in my case, if I entered my surname: "Baranda", the query made on the database, which would not be harmful, would be as follows: See Picture 20.

```
SELECT * FROM users WHERE lastname = "Baranda";
```

Picture 20: Example of an input that would not affect the system.

However, suppose now the case in which the user has certain knowledge about the syntax of the SQL language, and introduces the following: See Picture 21.

```
Baranda'; DROP TABLE users--
```

Picture 21: Example of input of malicious data that would affect the DB.

As can be seen in the image above, after the last name, the user enters a drop table using a SQL syntax trick, the inclusion of the single quote. The result of this is that the query made on the database is as follows: See Picture 22.

```
SELECT * FROM users WHERE lastname = 'Baranda'; DROP TABLE users--'
```

Picture 22: Example of how the malicious data introduced by the user is processed in the DB.

Once this malicious data has been introduced into the system, it may happen, either that the "users" table exists and that it is deleted directly, or that an error message appears that may provide information to the attacker. By explaining this simple example of how the SQL Injection attack works, it has been possible to verify the relevance of the two key points mentioned above in an injection attack: (1) the user controls the data as input and (2) the code of the program to be executed must be executed by joining this information provided by the user.

While it is true that there are numerous types of SQL code injection attacks and that these (obviously) are not as simple to execute as the one theoretically demonstrated in this chapter, those attacks will not be explained because they fall outside the competencies of this thesis. However, it will be analyzed the different forms of defense against this threat of SQL injection and will be explained the chosen form to implement the protection against the threat.

6.3 Defending the system against SQL-Injection threat.

Next, a tour of the different solutions evaluated to try to finish with the problem of the injection of SQL code will be carried out.

6.3.1 First approach: the use of a blacklist.

As a first approach to try to solve the problem of the injection, the valued idea was to implement a blacklist containing a series of words, typically, the keywords of the SQL language, such as DROP, DELETE, INSERT, UPDATE, GRANT, etc., so that, in case the user entered that exact string of characters, that input would be denied.

However, while it is true that the simplicity of the solution may be attractive after a little research was done, it was discarded, as attackers are usually able to find new combinations of characters (e.g using special characters) that bypass these restrictions of the blacklist. In addition, this solution has been implemented in some system, but the prerequisites that often lead to its use are the shortage of time or money (or both), so this solution was discarded because it was not among the most recommendable.

6.3.2 Stored procedures to prevent SQL-Injection.

Another technique evaluated to try to end the threat of SQLIA was the use of stored procedures. One stored procedure is a piece of prepared SQL code (subroutine) in the database which the applications can make calls to. They are quite widespread due to the fact that they add an additional abstraction level to the DB which means that the underlying DB structure can easily change if the interface on the stored procedure stays the same (Amirtahmasebi et al., 2009). Regarding the stored procedures, it should be noted that they do not represent a defense against the SQL Injection attack in themselves, but the way in which these methods are implemented will determine whether it is an effective defense against a possible injection attack or not. Every time that the user input is concatenated with SQL code that will be executed in the DB, the risk of suffering a SQL injection attack exists.h

In addition, when considering whether it was convenient to perform protection against injection attack by stored procedures or not, the importance of using parameterized queries was discovered. Once this was known, it was decided that, surely the best way to protect the system against the injection attack was the use of parameterized queries, since these had

already been used in the previous case in the project (so that they were not a novelty for the members of the project) and there were evidences that spoke of the efficacy of said queries.

6.3.3 Using parameterized queries to prevent SQLIA.

The parameterized queries are probably considered to be the best way to prevent SQL injection. Some references that prove the previous statement can be the following:

1."The right way to prevent SQL injection is by using parameterized queries. Doing this allows the server to create an execution plan for the query, which prevents any "injected" SQL from being executed." (Swan, 2010)

2. "Using of parameterized queries is a secure technique against SQL injection. This technique also known as prepared statements." (Sadeghian et al., 2013).

Once provided evidence of experts that the use of parameterized queries is the best solution to end the threat of injection of SQL code, the reason for this affirmation will be explained. The technique of parameterized queries must be implemented in the system in the coding phase. The steps of this technique are the following:

I. Some placeholders in the SQL query are reserved for the variables.
II. The SQL engine first will parse and compile the query without the variables and keep the result. Then, the query plan is constructed on the server before the query is executed with the parameter values. This query plan could be understood as a template on which the query to be executed is inserted: INSERT, DROP, UPDATE...once the query has been entered into the template (query plan), it will be executed only this query that has been defined in the template, without taking into account the rest of malicious statements introduced by the attacker.
III. Right after, the variables will be added and the query will be compiled with them included (so then, the parameter values are supplied at execution time).

Following the steps above, even if the attacker inserts a malicious query into the variable, the SQL engine will treat it as an ordinary string, so it will not affect to the database of the website.

Next, it will be tried to visually represent how the parameterized queries would work, ending the problem of SQL code injection. To imagine this, it will be started from the base of the existence of a form in which the user must enter as input his name and last name, and these will be added to the users table: See Picture 23.

```
var name;
var lastname;

name = Request.form("Name: ");
lastname = Request.form("Last Name: ");

sql = 'INSERT INTO users (users.name, users.lastname) VALUES (name, lastname)';
```

Picture 23: Simple form in which the user enters first and last name and is added to the users table without a parameterized query.

If the input of the user is the following: See Picture 24

```
Rodrigo
Baranda'); DROP TABLE users; PRINT 'You have some security issues!'--
```

Picture 24: Input of the user in the form.

The DB is facing a problem of SQL injection because the query that is being executed is as follows: See Picture 25.

```
INSERT INTO users (name, lastname) VALUES ('Rodrigo', 'Baranda'); DROP TABLE users; PRINT 'You have some
security issues!'--')
```

Picture 25: SQL injection problem without parameterized queries.

As it can be seen in the image above, with the input of the user, it is executed a query that is perfectly correct and that, obviously, will have a result in the database that is not the desired when created the form. Next, it will be showed what the previous code would be like using a parameterized query and it will show how it ends with the problem that has been posed throughout this chapter. See Picture 26.

```
var name;
var lastname;

name = Request.form("Name: ");
lastname = Request.form("Last Name: ");
variables = array(name, lastname);

sql = 'INSERT INTO users (users.name, users.lastname) VALUES (?, ?)';
```

Picture 26: Example of the code using a parameterized query.

To execute the statement, a connection with the DB, the query and the parameters must be provided, looking as follows: See Picture 27.

```
statement = executeQuery (connection, sql, variables);
```

Picture 27: Execution of the statement of the parameterized query.

Before executing the statement, on the server side, it is constructed the next query plan, so then, that and only that query will be executed: See Picture 28-

```
INSERT INTO users (users.name, users.lastname) VALUES (?, ?)
```

Picture 28: Query plan on the server side.

So, imagine that the input of the user is the same as before: See Picture 29.

```
Rodrigo
Baranda'); DROP TABLE users; PRINT 'You have some security issues!'--
```

Picture 29: Example of a malicious input of the user.

What will actually be done when using this parameterized query will be to assign the field *name = Rodrigo and the field lastname = Baranda '); DROP TABLE users; PRINT 'You have some security issues!' -*, but **this malicious attack will never reach the server, so it will not affect the DB**.

6.3.4 Parameterized queries in Laravel.

Laravel has a DB query builder which provides a convenient, fluent interface to creating and running DB queries. It can be used to perform most DB operations in a web application. Actually, what is used by this Laravel query builder is the *PDO class* from *PHP* (a class that represents a connection between *PHP* and a database server), specifically, the PDO parameter binding to protect the application against an SQLIA. See Picture 30.

```
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Dayle']);
```

Picture 30: *Database*: Binding Parameters to a Raw query. *Otwell, T. (2019).*

It should be noted that this technique of Binding Parameters to Raw Queries in Laravel, has already begun to be implemented in some of the queries of the system in which input is needed by the user, but has not yet been implemented in all queries necessary. Therefore, although from the theoretical point of view it has already become clear that the most effective technique to solve the SQLIA problem is that of Parameterized Queries, this has not been fully implemented in the backend of the FINCODA web application.

7 APPLICATION ERROR DISCLOSURE.

Throughout this chapter the problem of disclosure of information about a system will be addressed, studying what this problem consists of, making a brief analysis of the most important types of this information disclosure and analyzing the concrete problem that concerns the FINCODA project and how it has been solved

7.1 What is the problem of disclosure of information about?

When an attacker chooses a specific web application as a target, he usually cannot carry out an attack directly, but must, first of all, carry out an investigation of the web through a series of benign or non-offensive interactions on the web, with the objective of cataloging its content, functionality, determining the technologies it uses and identifying the key point of attack on the web.

Once this mentioned process has been carried out, considered as non-malicious, the attacker usually happens to interact with the web in a malicious and unexpected way for the web, trying to exploit its weaknesses that allows him to collect sensitive information. This revealed information will be more or less dangerous for the web (or beneficial from the point of view of the attacker) depending on the degree of criticality of the leaked information. Some of the possible examples of information revealed can range from the disclosure of server details to the disclosure of administrator account credentials or API secret keys, which could lead to devastating conditions on the web application. (Netsparker.com, 2019)

In short, it is the process in which the attacker looks for possible errors in the system that can reveal useful information about it to carry out future attacks.

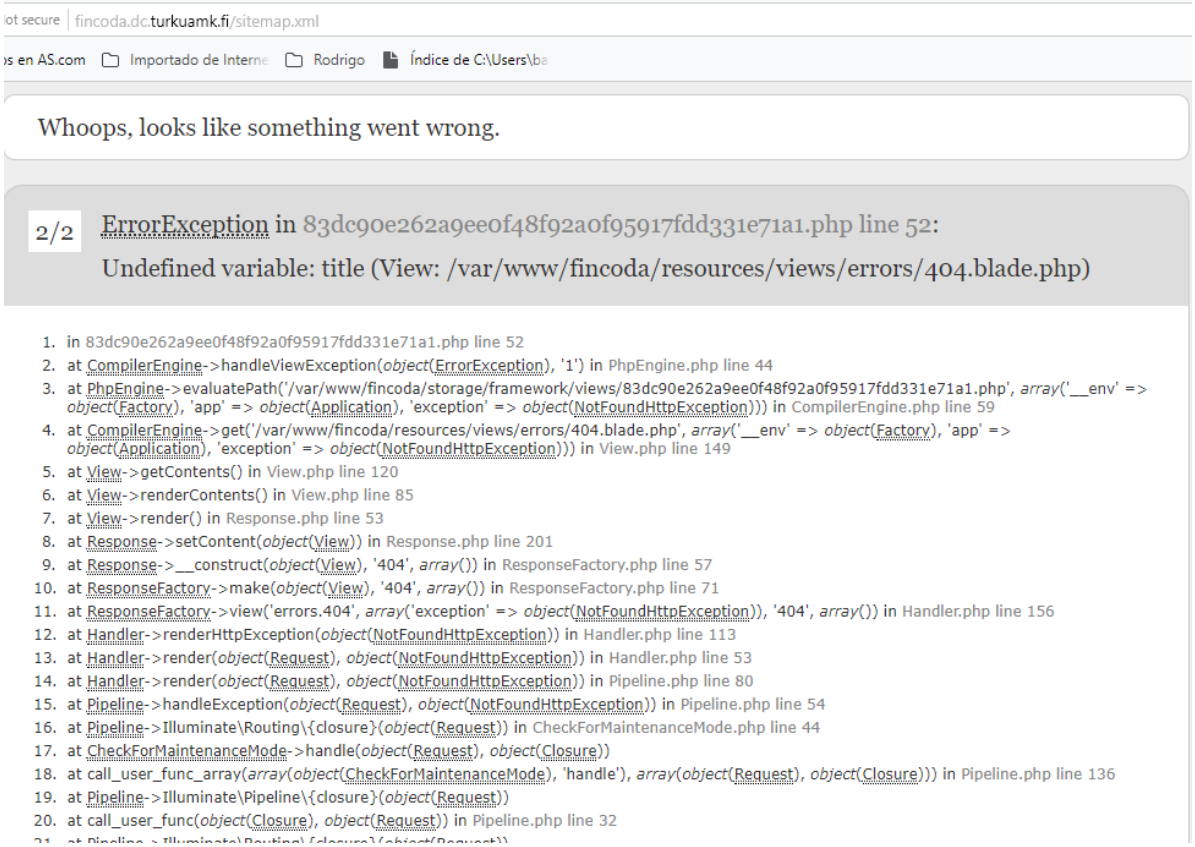
7.2 Types of information disclosure problems.

I. **Banner Grabbing**. In this type of attack, the attacker makes requests to the system looking for it to reveal information such as the version of PHP used or the version of the server. In this type of attack, the attacker does not obtain sensitive information directly on the system, but it can be useful for future attacks.
II. **Source Code Disclosure**. The name of this problem is very representative because it is exactly that: the revelation of part of the code of the backend of the system. Depending on the number of lines of code revealed and the degree of criticality of the lines shown, the problem will be more or less serious. A particularity of this problem is that, throughout the attack time, the system was like a *black box* for the attacker, since he did not have direct access to the inside information, however, when part of the code is revealed, the system can “transform” into a white box.
III. **The filename or file path disclosure**. Since this is the problem that concerns the FINCODA web application, it will be explained more in detail in a different point.

7.3 Filename or file path disclosure issue in the FINCODA web application.

One of the typical cases when disclosure of the information is that, due to an error or poor handling of exceptions it shows, for example, the path of a file within the system, which provides the attacker with sensitive information about the structure of the backend.

One of the issues that may return this path disclosure is, for example, a broken within the system. Due to an implementation of erroneous exception handling, at the moment in which the attacker (although it could also be a benign user) tries to access a broken link, he will obtain information related to the structure of system files, seeing the error path and being able to see, in addition, in which methods the error occurs and even what arguments had been passed to said methods. See Picture 31.



Picture 31: Path error disclosure in the FINCODA system.

In the example of the picture above, the attacker tried to access the sitemap of the website. However, the FINCODA web application does not have a sitemap, triggering the system one exception with a *404 error*. The problem is that this exception class is not properly programmed and it is showing all the error path.

When evaluating what could be the best solution to eliminate this file path disclosure, it was considered to show an error message to the user, with a message such as:

"Sorry, the web application has not implemented the sitemap yet."

However, it was seen that the majority of broken links that caused this same error were not expected to become part of the system at any time, so it was decided that the best thing would be, each time the mentioned *404 error* appeared, the attacker (or user) was returned to the main page of the project, in such a way that he did not have access to this file structure of the project.

Therefore, this exception module has been modified in such a way that, when the attacker enters the same link that was seen before, the one that caused the error (sitemap.xml): See Picture 32.

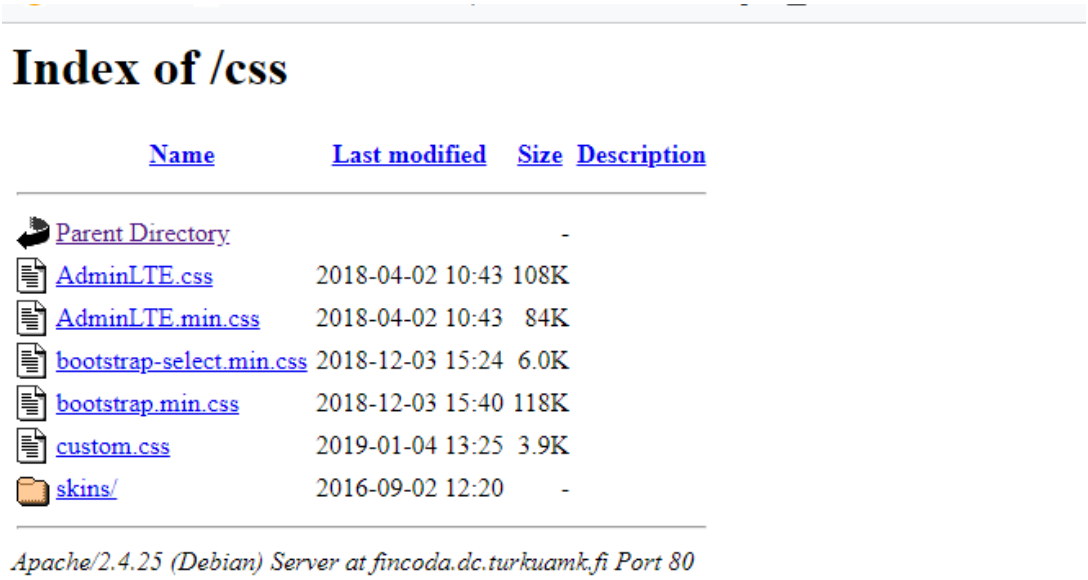
Picture 32: The attacker tries to access the sitemap.xml.

The attacker is automatically returned to the main page of the project (https://fincoda.dc.turkuamk.fi), so he does not have access to the file structure of the system anymore.

7.4 Directory Listing in the FINCODA web application.

A problem of a similar nature to the one explained in the previous point is that of Directory Listing. This problem usually occurs when there is no file such as index.php, showing, therefore, the tree of files that hang from that directory. If this problem occurs with *idex.php*, the files that hang from the root directory of the system will be shown, since the *index.php* file must be in the *root* directory.

Also, in this case, comparing it again with file path error disclosure (*see Filename or file path disclosure issue in the FINCODA web application.*), the problem is greater, because it is not only that the attacker can see the structure and the names of the files that are in the directory, but can also open these, having access to their code. See Picture 33.



Picture 33: Problem of directory listing in the FINCODA project.

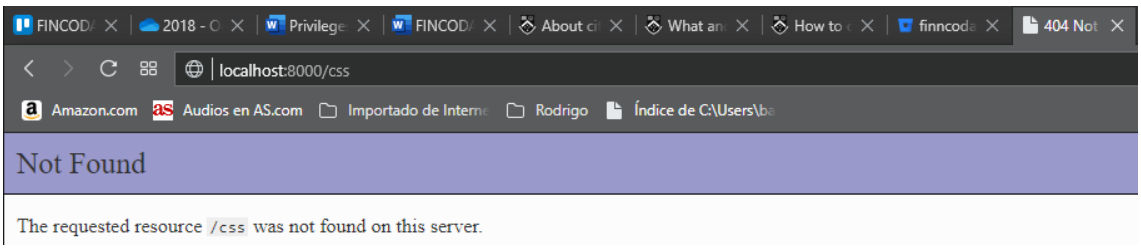
As it has been commented previously, the most typical case in which this problem occurs is because there is no file called index.php. However, in the specific case of the FINCODA web application, this was not the problem.

After researching in the web, the solution was discovered: changing the *.htaccess* file. The *.htaccess* file is a configuration file used by the servers running the *Apache Web Server* that can be used to enable or disable some functionalities of the server. ("What is .htaccess? - Apache .htaccess Guide, Tutorials & Examples", n.d.) See Picture 34.

```
-   RewriteEngine On
6 +     # To avoid directory browsing.
7 +     RewriteEngine on
8 +     RewriteCond %{REQUEST_URI} !^public
9 +     RewriteRule ^(.*)$ public/$1 [L]
```

Picture 34: Changes done in the .htaccess file to prevent directory listing.

There were several de links of the project affected by this directory listing problem. Now, when a user or an attacker tries to browse the same link, it is showing an error from the own Apache Server, and he is not allowed to reach the page. See Picture 35.



Picture 35: Directory listing problem avoided.

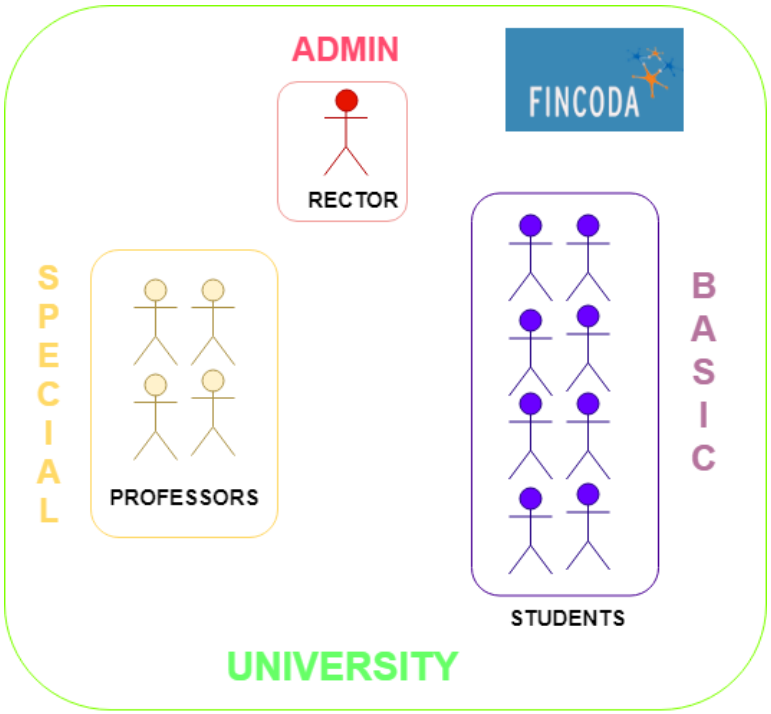
8 IMPROVING THE DB SECURITY – RESEARCH ABOUT THE DIFFERENT PERMISSIONS ON THE DB DEPENDING ON THE ROLE OF THE USER.

Throughout this chapter, the existing problematic in relation to the different changes that can be made on the database by the different types of users of the web application will be analyzed.

First, as a contextualization, it should be noted that the FINCODA web application has three types of users: *admin, special and basic*. These three types of users perform different functionalities, so their actions will have different repercussions in the database.

In order to try that the roles of the different users are understood in a visual way, the following practical case will be considered (the same that was used to explain the basic points of the FINCODA project*, see Picture 1: Common users? of the FINCODA project*.):

Suppose that a university decides to use the FINCODA web application to measure the individual abilities of students in innovation competencies such as creativity, critical reasoning, initiative, teamwork, and networking. Therefore, at the university there will be the next structure: See Picture 36.



Picture 36: Roles of the different users in a University using the FINCODA web app.

So, each user of the application will have a different role. There are 3 types of users defined:

- *Admin*. The administrator, in this specific case, the rector (or maximum entity of the organization), will be in charge of registering or deleting the organization, and, in summary, will have all the permissions on the system.
- *Special.* The special user, in this specific case, a teacher who wants to make a survey in his class, will have the corresponding permissions to create a survey, invite users to it, etc., but, unlike the administrator, for example, he will not be able to create or delete an organization among others.
- *Basic*. The basic user, in this specific case, a student, will not be able to create an organization or surveys, so he can basically only answer the survey.

8.1 What is the problem with which the system is confronted?

The framework that is being used in the system, Laravel 5.2, does not provide any specific mechanism for the creation of different user roles. This means that the changes that a user can make on the database are not directly determined by their role, but are determined by how their methods have been implemented in the controller modules. What does this mean? For example, that a basic user cannot eliminate an organization because, in the modules of its controller, this functionality has not been implemented, but not because there is a higher level assignment of permissions.

In the following image can be seen as an example of how part of the code of an insert is implemented in the table user_*profiles*, in the controller of the user type *admin*. See Picture 37.

```
DB::beginTransaction();
try{
DB::table('user_profiles')
            ->insert([
                'user_id'=>$id,
                'gender'=>'male',
                'country'=>'US',
                'city'=>'NY',
                'Street'=>'Main',
                'phone'=>'+170089777',
                'hired_date'=>'2016-09-04',
                'updated_at'=>Carbon::now()
            ]);
DB::table('users')
            ->where('id',$id)
            ->update([
                'profile_deleted'=>0,
                'enabled'=>1
            ]);
DB::commit();
```

Insert in the table user_profiles.

Picture 37: Example of how an insert in the table user_profiles is made by the admin.

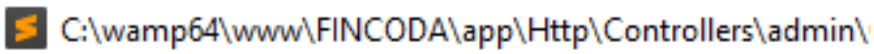
A better situation for the database of the system, would be that, depending on the role that the user had, that user had 'x' permissions to make modifications in ’y’ specific tables of the database, and that, in addition, these permissions were reaffirmed with the implementation of the functionality in the system modules by the programmer. In this way, the security of the system would be doubled, adding more restrictions when making modifications to the DB, thus making the system more secure and minimizing possible errors that could arise from the programmer.

8.2 A first approach to solving the problem.

As already explained, the main problem is that the users of the system have not assigned the permissions on the database based on their role but on the implementation of their modules.

Therefore, in a first approach to solve the problem, a study has been carried out on how each method of each of the modules has been implemented, in order to see what changes have been made or can be made by each type of user (*admin, special and basic*) in the database.

Below is an example of how part of this study has been carried out. For example, in the case of the type of user: admin, an analysis of the files that hang from the following folder was made, analyzing, as has been said previously, the methods of each module of this controller. See Picture 38.



Picture 38: Admin Controller. The files of this folder were analyzed.

As a result of this analysis, the SQL statements that the administrator type user could perform on a series of tables in the database were seen. See Table 3.

Table 3. SQL instructions and the tables in which can be executed by the user admin.

SQL INSTRUCTION.	TABLES OF THE DB IN WHICH THE SQL INSTRUCTION CAN BE EXECUTED.
SELECT	All
INSERT	external_evaluators, peer_surveys, peer_results, results, user_profiles, survey, participants, user_groups, user_in_groups
UPDATE	peer_surveys, participants, users, user_profiles, companies, company_profiles, role_user, surveys, user_groups, user_in_groups
DELETE	user_profiles, surveys, participants, results, peer_results, peer_surveys, user_groups, user_in_groups, peer_results

8.3 Possible implementation to end the problem.

After the current situation of how the modifications are being implemented in the database has been analyzed, the implementation of a new layer on the database will be proposed, specifying specifically what SQL commands can be executed on which tables each type of user.

Taking information from Oracle's own page on how to assign privileges to a user depending on their role ("MySQL:: MySQL 8.0 Reference Manual:: 6.2.10 Using Roles", n.d.), a possible implementation is proposed to solve this problem. See Picture 39.

```
GRANT INSERT, UPDATE, DELETE ON external_evaluators, peer_surveys, peer_results,
results, user_profiles, survey, participants, user_groups, user_in_groups
peer_surveys, participants, users, user_profiles, companies, company_profiles,
role_user, surveys, user_groups, user_in_groups user_profiles,
surveys, participants, results, peer_results, peer_surveys, user_groups, user_in_groups, peer_results
 * TO 'admin';
```

Picture 39: Possible implementation of privileges granting to enhance the security of the DB.

In the image above it can be seen how this 'extra' layer of security could be implemented, using the information extracted throughout the research work.

9 DISCUSSING THE RESULTS ACHIEVED.

Throughout each chapter of the Thesis, each issue has been raised from a theoretical point of view and has been explained either how the solution has been implemented or what the best technique to implement to end the specific threat over the backend of the FINCODA project web application would be. Therefore, two analyzes will be made regarding the defense techniques on the backend of the web application. On the one hand, one related to the specific part of the implementation of these techniques, and, on the other hand, another related to the part of documentation and research on which techniques would be the best to eliminate the threats that are currently threatening the vulnerabilities of the backend of the system.

9.1 Discussing the results of implementing appropriate protection techniques.

In the introduction of this document *(see Introduction)* it was explained that the fundamental purpose was to eliminate the vulnerabilities linked to the backend of the web application, implementing the necessary techniques or carrying out the convenient research work so that they could be resolved in the future.

On the one hand, from a personal point of view, as a result of this implementation process and focusing on the development part of his abilities as a programmer/engineer/analyst it is necessary to emphasize that in his background there was knowledge of other Object Oriented Programming languages, but that he had not worked specifically with *PHP* and that there was certain knowledge of cybersecurity but not of a high level, so that the part which is purely of implementation of the Thesis has served for issues such as: learning to a very high level of the *PHP* language, learning of different implementations of security techniques in a web application and learning how to correctly use the control of versions to work as a team in a coordinated way, among others.

On the other hand and focusing exclusively on the point of view of the project, there were a series of errors in the project (e.g when installing the system), in addition to vulnerabilities that needed the implementation of the appropriate defense technique to solve them. For example, as it has already been explained, there were errors when installing the system or vulnerabilities such as Clickjacking, SQL Injection Attack or Application Error Disclosure that have been resolved by implementing the appropriate techniques in each case. In this regard, it should be noted that the results obtained with respect to the protection against **Clickjacking and Application Error Disclosure** have been **highly positive**, achieving the implementation of headers that ended up with the problem of Clickjacking in the first case, and implementing the appropriate error handling modules to solve the problem of displaying sensitive information from the web application. However, for example, in relation to the results obtained on the defense against the protection of an **SQL injection** code attack, these could be considered **simply positive**, since it has been discovered what would be the best technique to solve the problem, seen how this technique should be implemented and actually implemented in some of the queries of the system; but they can not be considered highly satisfactory in this case

because, due to the great work that it would be needed to change all the queries, this has not been done in full due to a lack of time.

Finally, as a summary of the results on the implementation part, it has to be said that it is possible that the implementation work has not been too extensive. This has been due in large part to the fact that the installation of new technologies is being considered: whether to update the version of the framework used to a more modern version or to use a new framework (and these both things needed big previous research). In addition, a large background is needed about each technique before implementing its solution, which has made the work has been in many cases, more theoretical than practical.

9.2 Discussing the results of researching and documentation.

Analyzing, in this case, the results obtained after the research and documentation process, a series of important points should be highlighted.

On the one hand, it is worth mentioning that, with respect to each of the techniques that have been implemented directly on the system, there is a theoretical background that analyzes the threat in question, the degree of criticality of this, the different techniques that can be chosen to solve this threat and, finally, an explanation of why that particular implementation has been chosen. Therefore, although it is true that this thesis has not innovated with respect to prior knowledge that existed in the matter, understanding as innovated, for example, discover a new defense technique; the results from the **theoretical point of view** of the **defense techniques implemented** can be considered **positive**, since they have served to contextualize, explain the vulnerability in question from the theoretical point of view and, finally, explain theoretically why the chosen solution is adequate.

On the other hand, it has already been commented throughout the file that a purely research part has been carried out, whether this research was on the current system and how to improve the security of the backend, (e.g the database, see *Improving the db security – research about the different permissions on the DB depending on the role of the user.*) or on how possible threats could be solved with the upgrade of the current framework to a higher version. As this research work cannot be completely contrasted, since it is a work that could be developed in the future but has not yet been carried out, the **perception** of the writer of this document is **positive**, since a thorough analysis has been carried out of how those changes (whether updating the Laravel framework or assigning permissions on the database to each user role, among others) would improve the security of the database and are realistic in relation to the work that would take them to cape but, as is logical, it can not be stated 100% that the result obtained is positive.

9.3 Overall conclusion about the results obtained.

In summary, as a general conclusion about the results obtained, it should be noted that they can be considered as positive, since several vulnerabilities or errors have been solved and, in addition, research and documentation have been carried out on possible changes that

would improve the system. However, it should also be noted that, perhaps, the number of resolved vulnerabilities has not been too great, so, although it is true that the results have been positive on those in which the analysis and implementation work has been carried out, the number of system-resolved threats implementing the programming techniques could have been greater in case of having made a better time distribution.

10 CONCLUSION.

In conclusion, the different conclusions and results to which this document has arrived will be presented.

In the first place, this thesis has dealt with: vulnerabilities related to the backend of a web application and a necessary investigation to find out how to implement security improvements in the system. Within these vulnerabilities and this research work, the most important are: implementing protection against Clickjacking, implementing protection against Application Error Disclosure, researching (and implementing part of it) how to solve the SQL Injection Attack threat, researching how to improve the general security of the database depending on the role of the user, researching how new technologies could improve the security of the system, and solving errors related to the installation of the web application. Fundamentally, what has been demonstrated in the process of analysis, can be explained from different points of view. From the theoretical point of view, it has been analyzed each applicable defense technique of the system, analyzing its possible advantages and disadvantages. In some cases, it has been demonstrated in a practical way how to implement these techniques on the backend of a web application and. Finally, in other cases, the key steps that should be followed to implement these techniques in the future have been documented.

The results are significant in the sense that they are the result of a process of analysis and research that culminates either with the implementation of the programming techniques or the documentation that will mitigate potential threats in the future. When each defensive technique has been chosen to end with the threats such as the ones mentioned above (Clickjacking, SQLIA, etc.), it has been explained the reason for the chosen solution and why it is more complete than the others existing for this specific project. In the cases in which documentation has been made, the key steps that have to be take and the benefits that would suppose on the system the implementation of them. In addition, with respect to the above, these results can be significant because, although it is true that the solution offered is particular to the Laravel framework, the theoretical basis raised on each problem is exportable to other systems.

Regarding the limitations and benefits of the results, the following should be noted: the clearest limitation is given by the part related to the purely research work. Since the techniques studied have not been directly implemented on the system, it has not been tested empirically if they work as theoretically stated. The benefits are clearly security improvements and the elimination of several of the vulnerabilities that affected the system.

The results obtained could be applied in those systems that were using a Laravel Framework and the technologies studied could be implemented following the same methodology. In case the web application was not using this same Framework, the implementation could not be done as such, but the theoretical ideas could be extracted of why to implement 'x' or ’y’ defense technique on the backend of the web application to eliminate the vulnerability of it.

The work presented in this thesis could be further developed in the future by completing some of the tasks that were left unfinished due to lack of time (such as including queries parameterized throughout the system to protect SQLIA) or implementing the techniques that

have been investigated and showed how It would improve the system, but they have not been implemented yet.

REFERENCES

Amirtahmasebi, K., Jalalinia, S. R., & Khadem, S. (2009, November). *A survey of SQL injection defense mechanisms.* In 2009 International Conference for Internet Technology and Secured Transactions,(ICITST) (pp. 1-8). IEEE.

Acunetix. (n.d.). *What is SQL Injection (SQLi) and How to Prevent It*? [online] Available at: https://www.acunetix.com/websitesecurity/sql-injection/ [Accessed 22 May 2019].

Ighodaro, N. (2018*). How Laravel implements MVC and how to use it effectively*. Retrieved from https://blog.pusher.com/laravel-mvc-use/

Jackson, B. (2018). *Hardening Your HTTP Security Headers*. [online] KeyCDN. Available at: https://www.keycdn.com/blog/http-security-headers [Accessed 20 May 2019].

Jackshon, B. (2016). *X-Frame-Options - How to Combat Clickjacking*. [online] KeyCDN. Available at: https://www.keycdn.com/blog/x-frame-options [Accessed 20 May 2019].

Grossman, J. (2006, November). *"How does a cross-site request forgery work".* Available: http://searchsoftwarequality.techtarget.com/expert/KnowledgebaseAnswer/0,289625,sid92-gci1230450, 00.html.

Lalia, S., & Moustafa, K. (2019, April). *Implementation of Web Browser Extension for Mitigating CSRF Attack*. In World Conference on Information Systems and Technologies (pp. 867-880). Springer, Cham.

Learning Center. (n.d.). *What is CSRF | Cross Site Request Forgery Example | Imperva*. [online] Available at: https://www.imperva.com/learn/application-security/csrf-cross-site-request-forgery/ [Accessed 21 May 2019].

Netsparker.com. (2019). Information Disclosure Issues and Attacks in Web Applications. [online] Available at: https://www.netsparker.com/blog/web-security/information-disclosure-issues-attacks/ [Accessed 23 May 2019].

Otwell, T. (2016a).*Database: Migrations - Laravel - The PHP Framework For Web Artisans*. [online] Laravel.com. Available at: https://laravel.com/docs/5.2/migrations [Accessed 21 May 2019].

Otwell, T. (2016b). *Installation - Laravel - The PHP Framework For Web Artisans*. [online] Laravel.com. Available at: https://laravel.com/docs/5.2/installation [Accessed 21 May 2019].

Otwell, T. (2019). *Database: Getting Started - Laravel - The PHP Framework For Web Artisans*. [online] Laravel.com. Available at: https://laravel.com/docs/5.8/database#running-queries [Accessed 23 May 2019].

OWASP *Top 10 - 2017 The Ten Most Critical Web Application Security Risks. (2017).* [ebook] Available at: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf [Accessed 22 May 2019].

Rydstedt, G., Bursztein, E., Boneh, D. and Jackson, C. (2010). *Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites*. [online] Seclab.stanford.edu. Available at: http://seclab.stanford.edu/websec/framebusting/framebust.pdf [Accessed 20 May 2019].

Sadeghian, A., Zamani, M., & Ibrahim, S. (2013, September). *SQL injection is still alive: a study on SQL injection signature evasion techniques*. In 2013 International Conference on Informatics and Creative Multimedia (pp. 265-268). IEEE.

StatCounter. (n.d.). *Global market share held by the leading web browser versions as of February 2019*. In Statista - The Statistics Portal. Retrieved May 20, 2019, from https://www.statista.com/statistics/268299/most-popular-internet-browsers/.

Swan, B. (2010). *What's the Right Way to Prevent SQL Injection in PHP Scripts?*. [online] Available at: https://blogs.msdn.microsoft.com/brian_swan/2010/03/04/whats-the-right-way-to-prevent-sql-injection-in-php-scripts/ [Accessed 22 May 2019].

Swan, B. (2008). *How and Why to Use Parameterized Queries*. [online] Microsoft Drivers for PHP for SQL Server Team Blog. Available at: https://blogs.msdn.microsoft.com/sqlphp/2008/09/30/how-and-why-to-use-parameterized-queries/ [Accessed 23 May 2019].

What is .htaccess? - Apache .htaccess Guide, Tutorials & Examples. Retrieved from http://www.htaccess-guide.com

Wu, H. and Zhao, L. (2015). *Web Security: A WhiteHat Perspective*. 13th ed. Florida: CRC Press