420-B01-TV

PREVENTION AND SECURITY IN WEB DEVELOPMENT – 09061

Prof. Pargol Poshtareh

TEAM 6

Project Report

SQL Injection

TEAM MEMBERS:

2230911 – Ruben Sebastian Graneros Rojas

2310779 – Felipe Galvao De Andrade Gomes

MONTREAL

Aug 2023

TABLE OF CONTENTS

INTRODUCTION

The aim of this project is to explore and address one of the most prevalent and damaging security vulnerabilities in web development, namely SQL Injection (SQLi). SQLi attacks pose a significant threat to web applications by exploiting input validation and allowing malicious agents to manipulate database queries, potentially gaining unauthorized access to sensitive information or threatening database integrity.

By carefully crafting and executing SQL commands, an unauthorized user gains the ability to impersonate higher-privileged users, potentially granting themselves or others administrator privileges. This can lead to unauthorized data manipulation, transaction and balance alterations, as well as the retrieval and potential destruction of all data stored on the server.

To better understand how SQLi works, how dangerous it is, and how it can be prevented, it is important to grasp some basic concepts, such as what are databases and their value, what is SQL and how to secure the input provided by users against exploitation.

1. DATABASES AND THEIR VALUE

The main target of SQLi are the databases largely used in modern applications.

Data protection is an important issue, because there are many aspects of our lives that we want to remain private and many others that are sensible because their disclosure might be dangerous, like our financial status, account numbers, balance, social security number.

Thanks to advancements in technology our processing power, communications speed, connectivity and data storage are increasing at an incredible rate. That’s why researchers speak about the 3 Vs related to the Big Data phenomenon.

<u>The first V is for Volume</u>. According to Forbes, more data has been created in the past two years than in all human history. You have all seen Instagram and tiktok and can imagine that the amount of photos and videos there is clearly bigger than all humanity has produced until then.

<u>The second V is for variety</u>. Data was once synonym of database files - such as, excel, csv and access – but now it’s being extracted from non-traditional forms, like video, text, pdf, and graphics on social media, as well as other means such as wearable devices.

The variety of information being collected also relates to many aspects of our lives. Connected devices like smartphones, smartwatches and such can gather data about geographic localization, heart rate, temperature, oxygen level, physical activity. Smart doorbell cameras can gather information about deliveries and catalog faces and voices. Voice recognition devices that can now understand natural language and that is only possible due to the variety of data gathered.

The third V is for velocity. All that involves data is faster: the collection is immediate and automatic, made by smart devices and computers. The transmission of data is also faster thanks to increased internet speeds and reliability, that is why 5G is so important for the evolution of some technologies like automatic cars, because it increases the speed and reliability of data transmission. And also, the processing of data is being automatized by Artificial Intelligence and greater processing power of modern computers.

All those facts work together to make data an essential asset of the modern world. And that's why some people say that "DATA IS THE NEW OIL", since it provides a high commercial value, as well as geopolitical power, but at the same time it poses challenges and dangers.

SQLi targets these valuable assets, allowing the attacker to gain access to sensitive information about users or even to manipulate the databases' contents, potentially causing great damage to individuals, companies and even governments.

2. STRUCTURED QUERY LANGUAGE (SQL) BASICS

Structured query language (SQL) is a domain-specific language used to create, maintain, access, and manipulate databases, especially relational database management system (RDBMS) and handling structured data.

It is a 4th generation, high-level language, meaning it is closer to human language and is written in the form of statements that represent the commands to be performed in the database.

SQL is a non-procedural language, because the user doesn't need to write the steps to be executed in order to perform the query. SQL abstracts away the specific steps required to achieve a result. Instead of telling the computer exactly what to do and how to do it, you express what you want to achieve.

SQL allows you to declare what you want to retrieve or manipulate from a database without specifying how to do it. You state your query in a declarative manner, focusing on the desired outcome rather than the sequence of steps.

Some common SQL commands include:

- SELECT: Used to retrieve data from one or more tables based on specified criteria.
- INSERT: Used to add new records into a table.
- UPDATE: Used to modify existing records in a table.
- DELETE: Used to remove records from a table.

- CREATE: Used to create new tables, views, indexes, and other database objects.
- ALTER: Used to modify existing database objects like tables, columns, or constraints.
- DROP: Used to delete tables, views, and other database objects.
- JOIN: Used to combine data from multiple tables based on a related column.
- WHERE: Used to filter data based on specified conditions.
- ORDER BY: Used to sort the result set in a specific order.
- GROUP BY: Used to group rows that have the same values in specified columns.

The simplicity of these commands makes SQLi vulnerabilities all the more dangerous, since simple insertions can execute major changes in the database, such as showing sensitive information, altering data, or even deleting entire tables, causing great damage.

3. DETECTING SQLi VULNERABILITIES

SQL injection is a technique where an attacker exploits flaws in application code responsible for building dynamic SQL queries.

The input provided by the user can be compared as a puzzle piece that will be inserted and executed by the application, according to the source code developed by the programmer.

In this project, the team created a simple webpage with user register and login capabilities. The unsafe version source code contains a vulnerability in the query statement:

```
$query = "SELECT * FROM users WHERE user_name = '$user_input_username'
AND password = '$user_input_password'";
$result = $conn->query($query);
```

As we can see, the variables <$user_input_username> and <$user_input_password> are passed directly to the query that will be executed in the server, allowing for exploitation by a malicious user.

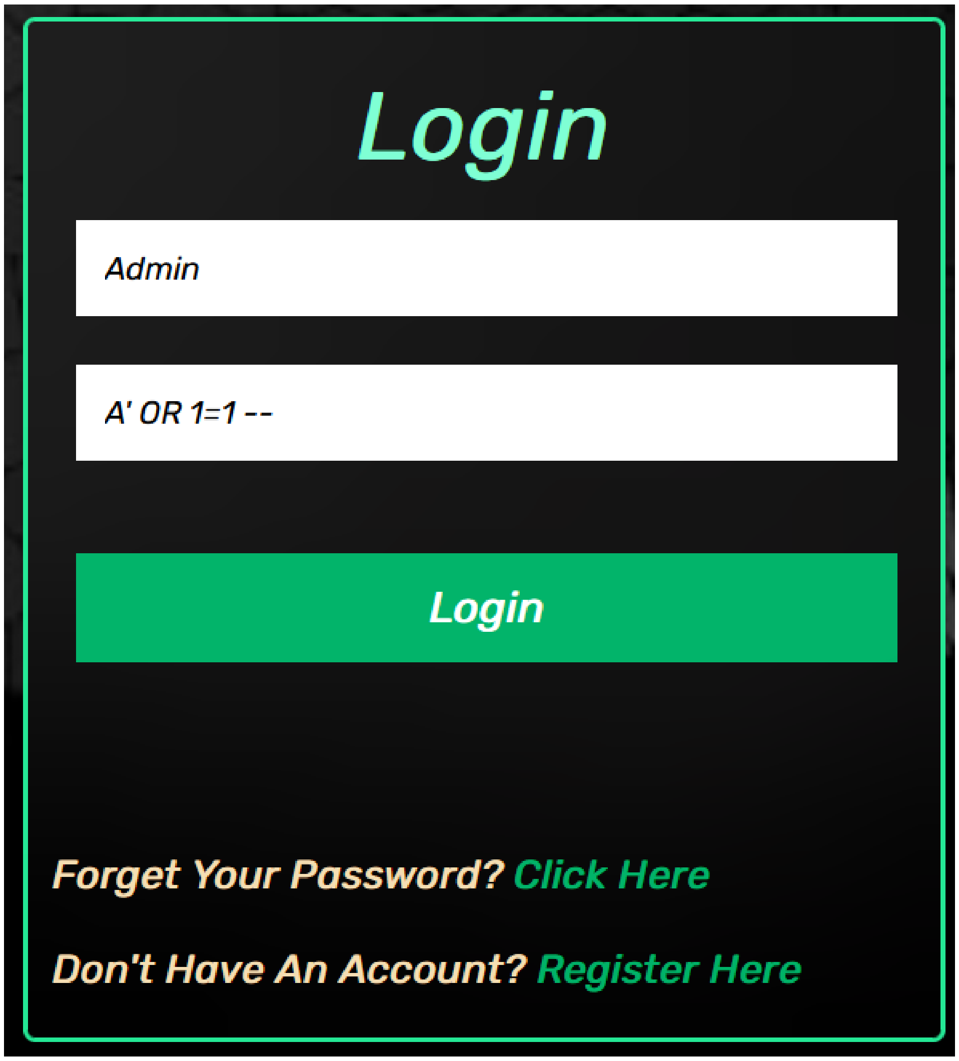
In the login page, when inserting the character <'> in any of the fields throws the following error:

> **Fatal error**: Uncaught Error: Call to a member function fetch_assoc() on bool in C:\xampp\htdocs\Project\login_process.php:23 Stack trace: #0 {main} thrown in **C:\xampp\htdocs\Project\login_process.php** on line **23**

This shows the use of this character is being interpreted as part of the query, resulting in a syntax error.

Knowing this, the attacker can test different query structures to exploit the vulnerability.

If the attacker knows the administrator username is <admin>, they can insert the following input as password to gain access to the administrator account and privileges without knowing the password:


Login
Admin
A' OR 1=1 --
Login
Forget Your Password? Click Here
Don't Have An Account? Register Here

In this case, the text input <*a' OR 1=1 --* > will be directly passed into the query and executed by the application, successfully accessing the database with the wrong password.

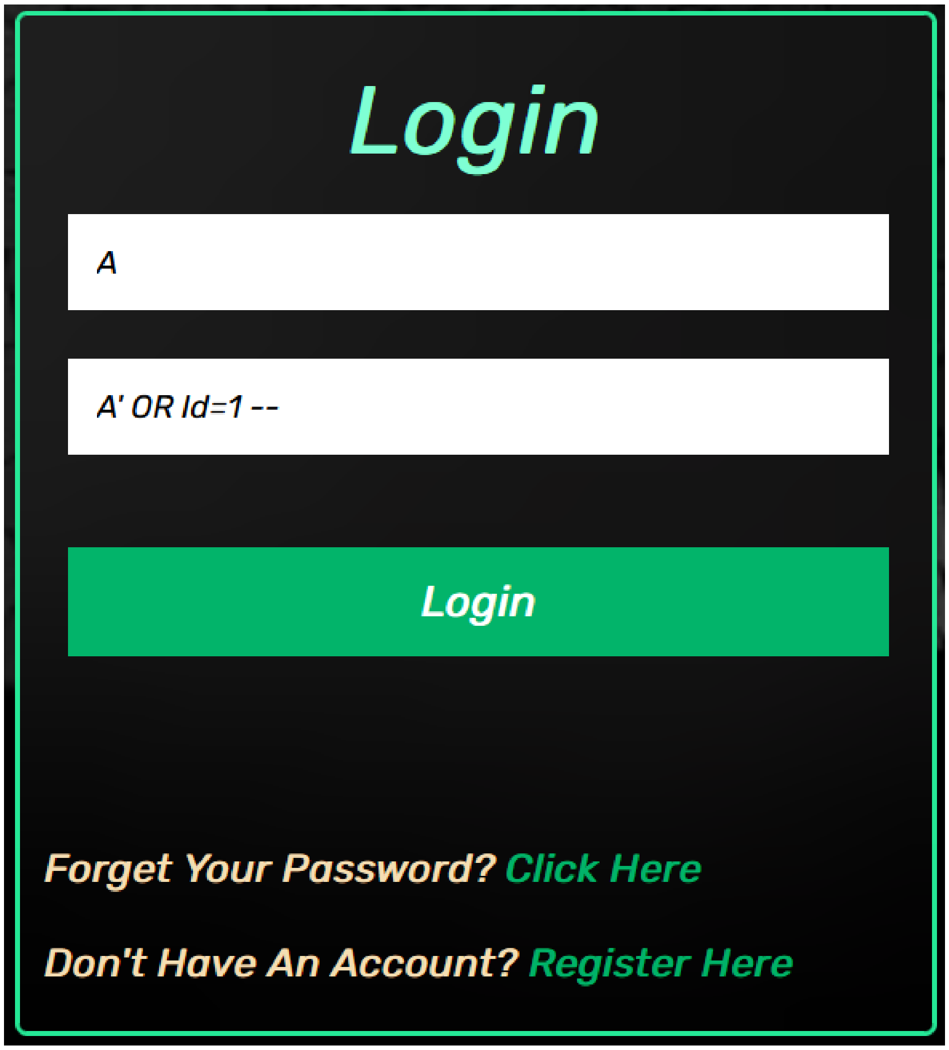
The query will be passed as following:

```
SELECT * FROM users WHERE user_name = '$user_input_username' AND
password = '$user_input_password'
```

```
SELECT * FROM users WHERE user_name = 'Admin' AND password = 'A' OR
1=1-- "
```

Since the statement <1=1> is true, the query will successfully make the selection, granting access to the table information even if the password is incorrect.

Another way of exploiting this vulnerability, without even knowing the username, is specifying the user id in the password field, like so:



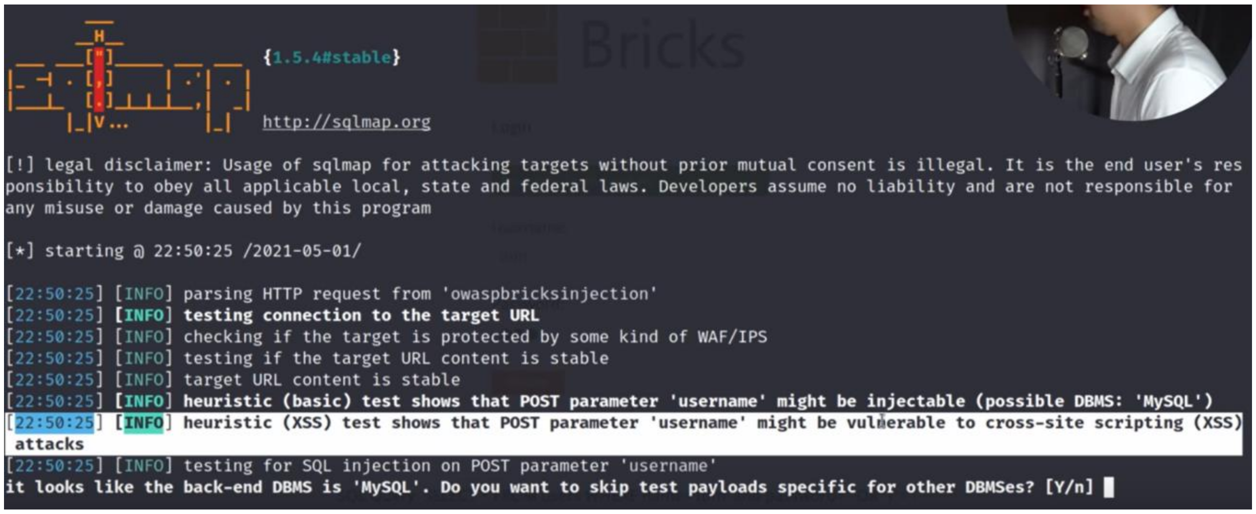
A

A' OR Id=1 --

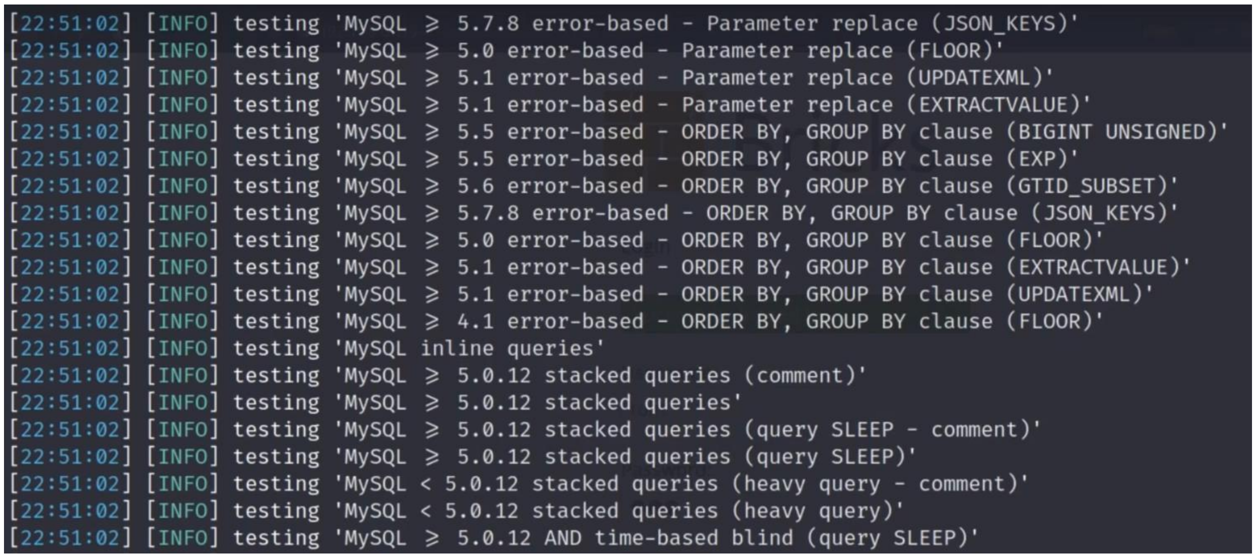
Login
Forget Your Password? Click Here
Don't Have An Account? Register Here

This will grant access to the user whose ID is ‘1’, which in this case is the administrator account. Changing the ID number in the query field will grant access to different users’ accounts.

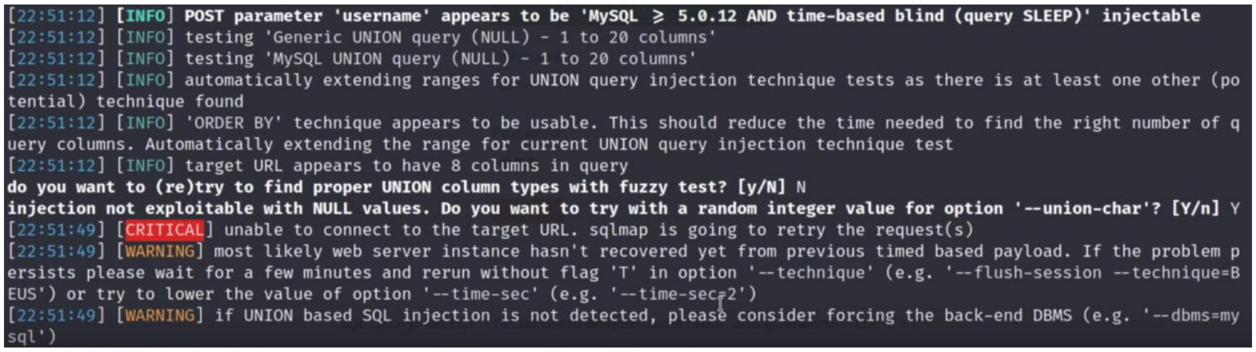
Even when the potential attacker doesn’t have access to the source code, they can use specialized software to test and show vulnerabilities, such as SQL Map:



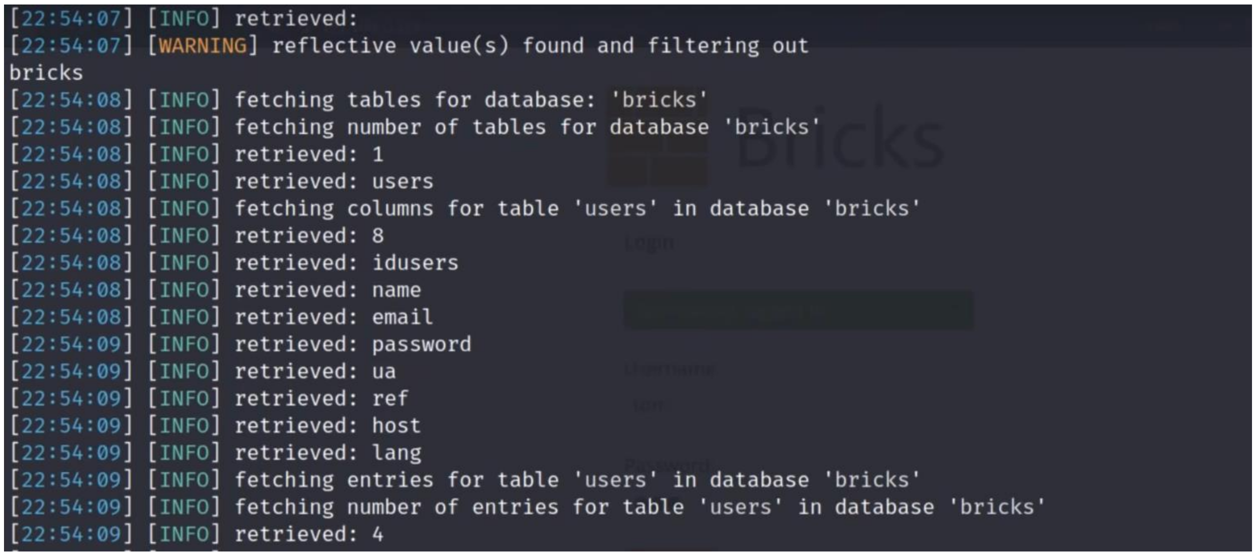
```
        ___
       __H__
 ___ ___[)]_____ ___ ___  {1.5.4#stable}
|_ -| . [,]     | .'| . |
|___|_  [)]_|_|_|__,|  _|
      |_|V...       |_|   http://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's res
ponsibility to obey all applicable local, state and federal laws. Developers assume no liability and are not responsible for
any misuse or damage caused by this program

[*] starting @ 22:50:25 /2021-05-01/

[22:50:25] [INFO] parsing HTTP request from 'owaspbricksinjection'
[22:50:25] [INFO] testing connection to the target URL
[22:50:25] [INFO] checking if the target is protected by some kind of WAF/IPS
[22:50:25] [INFO] testing if the target URL content is stable
[22:50:25] [INFO] target URL content is stable
[22:50:25] [INFO] heuristic (basic) test shows that POST parameter 'username' might be injectable (possible DBMS: 'MySQL')
[22:50:25] [INFO] heuristic (XSS) test shows that POST parameter 'username' might be vulnerable to cross-site scripting (XSS)
 attacks
[22:50:25] [INFO] testing for SQL injection on POST parameter 'username'
it looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific for other DBMSes? [Y/n]
```



```
[22:51:02] [INFO] testing 'MySQL ≥ 5.7.8 error-based - Parameter replace (JSON_KEYS)'
[22:51:02] [INFO] testing 'MySQL ≥ 5.0 error-based - Parameter replace (FLOOR)'
[22:51:02] [INFO] testing 'MySQL ≥ 5.1 error-based - Parameter replace (UPDATEXML)'
[22:51:02] [INFO] testing 'MySQL ≥ 5.1 error-based - Parameter replace (EXTRACTVALUE)'
[22:51:02] [INFO] testing 'MySQL ≥ 5.5 error-based - ORDER BY, GROUP BY clause (BIGINT UNSIGNED)'
[22:51:02] [INFO] testing 'MySQL ≥ 5.5 error-based - ORDER BY, GROUP BY clause (EXP)'
[22:51:02] [INFO] testing 'MySQL ≥ 5.6 error-based - ORDER BY, GROUP BY clause (GTID_SUBSET)'
[22:51:02] [INFO] testing 'MySQL ≥ 5.7.8 error-based - ORDER BY, GROUP BY clause (JSON_KEYS)'
[22:51:02] [INFO] testing 'MySQL ≥ 5.0 error-based - ORDER BY, GROUP BY clause (FLOOR)'
[22:51:02] [INFO] testing 'MySQL ≥ 5.1 error-based - ORDER BY, GROUP BY clause (EXTRACTVALUE)'
[22:51:02] [INFO] testing 'MySQL ≥ 5.1 error-based - ORDER BY, GROUP BY clause (UPDATEXML)'
[22:51:02] [INFO] testing 'MySQL ≥ 4.1 error-based - ORDER BY, GROUP BY clause (FLOOR)'
[22:51:02] [INFO] testing 'MySQL inline queries'
[22:51:02] [INFO] testing 'MySQL ≥ 5.0.12 stacked queries (comment)'
[22:51:02] [INFO] testing 'MySQL ≥ 5.0.12 stacked queries'
[22:51:02] [INFO] testing 'MySQL ≥ 5.0.12 stacked queries (query SLEEP - comment)'
[22:51:02] [INFO] testing 'MySQL ≥ 5.0.12 stacked queries (query SLEEP)'
[22:51:02] [INFO] testing 'MySQL < 5.0.12 stacked queries (heavy query - comment)'
[22:51:02] [INFO] testing 'MySQL < 5.0.12 stacked queries (heavy query)'
[22:51:02] [INFO] testing 'MySQL ≥ 5.0.12 AND time-based blind (query SLEEP)'
```



```
[22:51:12] [INFO] POST parameter 'username' appears to be 'MySQL ≥ 5.0.12 AND time-based blind (query SLEEP)' injectable
[22:51:12] [INFO] testing 'Generic UNION query (NULL) - 1 to 20 columns'
[22:51:12] [INFO] testing 'MySQL UNION query (NULL) - 1 to 20 columns'
[22:51:12] [INFO] automatically extending ranges for UNION query injection technique tests as there is at least one other (po
tential) technique found
[22:51:12] [INFO] 'ORDER BY' technique appears to be usable. This should reduce the time needed to find the right number of q
uery columns. Automatically extending the range for current UNION query injection technique test
[22:51:12] [INFO] target URL appears to have 8 columns in query
do you want to (re)try to find proper UNION column types with fuzzy test? [y/N] N
injection not exploitable with NULL values. Do you want to try with a random integer value for option '--union-char'? [Y/n] Y
[22:51:49] [CRITICAL] unable to connect to the target URL. sqlmap is going to retry the request(s)
[22:51:49] [WARNING] most likely web server instance hasn't recovered yet from previous timed based payload. If the problem p
ersists please wait for a few minutes and rerun without flag 'T' in option '--technique' (e.g. '--flush-session --technique=B
EUS') or try to lower the value of option '--time-sec' (e.g. '--time-sec=2')
[22:51:49] [WARNING] if UNION based SQL injection is not detected, please consider forcing the back-end DBMS (e.g. '--dbms=my
sql')
```



```
[22:54:07] [INFO] retrieved:
[22:54:07] [WARNING] reflective value(s) found and filtering out
bricks
[22:54:08] [INFO] fetching tables for database: 'bricks'
[22:54:08] [INFO] fetching number of tables for database 'bricks'
[22:54:08] [INFO] retrieved: 1
[22:54:08] [INFO] retrieved: users
[22:54:08] [INFO] fetching columns for table 'users' in database 'bricks'
[22:54:08] [INFO] retrieved: 8
[22:54:08] [INFO] retrieved: idusers
[22:54:08] [INFO] retrieved: name
[22:54:08] [INFO] retrieved: email
[22:54:09] [INFO] retrieved: password
[22:54:09] [INFO] retrieved: ua
[22:54:09] [INFO] retrieved: ref
[22:54:09] [INFO] retrieved: host
[22:54:09] [INFO] retrieved: lang
[22:54:09] [INFO] fetching entries for table 'users' in database 'bricks'
[22:54:09] [INFO] fetching number of entries for table 'users' in database 'bricks'
[22:54:09] [INFO] retrieved: 4
```

The software can automatize the testing of several parameters that can be injected and provide the information to the attacker.

In this case, the software showed the vulnerability and also retrieved all the columns of the database table.

4. PREVENTING SQLi

As exposed, the unsafe version of the website's login page and application allows the user to fill the input fields for username and password with SQL queries, using characters that instead of being interpreted as strings, will be interpreted as part of SQL syntax by the server application.

This can be prevented by restricting the input field:

```
// Prepare and execute a query to validate user credentials
$query = "SELECT user_name FROM users WHERE user_name = ? AND password
= ?";
$stmt = $conn->prepare($query);
$stmt->bind_param("ss", $user_input_username, $user_input_password);
$stmt->execute();
```

In this version, the direct insertion of user inputs into the SQL query was replaced by placeholders (?). The statement is then prepared using $conn->prepare() and bind the parameters using $stmt->bind_param(). This way, user inputs are properly sanitized and separated from the query, preventing SQL injection attacks.

By specifying “ss” for the parameters, the source code restricts the data types of the parameters that the programmer is binding to the prepared statement.

So, "ss" indicates that the code is binding two parameters, both of which are strings. In this case, the code is binding the user input username and password, both as strings, to the prepared statement.

This way, even if the user inserts characters that could be used as SQL syntax, the application will not interpret them as part of the query, but only as strings, avoiding the injection.

Apart from the use of prepared queries as demonstrated, there are other strategies to prevent SQLi attacks.

Escaping user-supplied input is a technique used to prevent some characters from being interpreted as part of the SQL query's syntax. Instead of allowing these characters to be processed as SQL commands or conditions, the database treats them as literal text.

When writing SQL queries, specific characters and words have special meanings in the SQL language. For example, the * character is often used as a wildcard to represent any column, and the word OR is a logical operator that combines conditions. However, when users provide input that contains these characters (either intentionally or accidentally), it can lead to unintended behavior or security vulnerabilities in the application.

Escaping means transforming characters with special meanings into their literal form. Most programming languages and database libraries provide functions or methods for escaping input. For example, PHP's mysqli_real_escape_string() function escapes special characters so they are treated as literal text:

```
$escaped_input = $conn->real_escape_string($user_input);
$query = "SELECT * FROM users WHERE username = '$escaped_input'";
```

Stored procedures are precompiled sets of one or more SQL statements that are stored in a database and can be executed later. They provide a way to encapsulate and manage complex database operations, improving performance and security.

Finally, enforcing least privilege is a fundamental principle in cybersecurity. It involves giving users or applications only the minimum level of access rights and permissions necessary to perform their required tasks. This approach aims to limit potential damage and exposure in case of a security breach. When it comes to database security and SQL injection prevention, enforcing the principle of least privilege helps prevent unauthorized manipulation of the database and minimizes the potential impact of a successful SQL injection attack.

5. SQLi VARIETIES

There are four main sub-classes of SQL injection:

- Classic SQLi

This is the type of injection performed in this project’s practical demonstration.

Classic SQL injection is the most common and well-known type of SQL injection attack. It occurs when an attacker is able to insert malicious SQL code into a vulnerable SQL statement. This often happens through user inputs that are improperly sanitized or validated before being incorporated into the query. The injected code can manipulate the query's behavior, potentially exposing sensitive information or modifying the database.

- Blind or Inference SQL injection

Blind SQL injection occurs when an attacker doesn't directly see the query's results but can infer information based on the application's response. This type of attack is prevalent when the application doesn't display query results directly on the webpage, but attackers can still use techniques to infer data through conditional statements. Blind SQL injection can be time-based (delays in server responses) or boolean-based (true/false responses) depending on how the attacker gathers information.

- Database management system specific SQLi

This type of SQL injection targets the specific syntax and features of different database management systems (DBMS). Different DBMSs might have variations in SQL syntax, functions, and behavior. Attackers tailor their injection techniques to the targeted DBMS to maximize the chances of success. Examples include Microsoft SQL Server-specific attacks (using @@version), MySQL-specific attacks (using LIMIT), and Oracle-specific attacks (using DUAL).

As shown in the previous topic, the use of specific software can reveal what type of database system is being used and reveal to the potential attacker the system specific SQLi.

- Error-Based SQL Injection

Attackers intentionally trigger errors in the application's responses to gather information about the database structure or the data stored within it.

Once the vulnerability is identified, attackers deliberately inject malformed or malicious SQL code into the input fields. The goal is to manipulate the application's database queries in a way that causes errors. These errors are then presented in the application's response, which can reveal valuable information about the database's structure, contents, or even error messages containing sensitive data.

- Union-Based SQL Injection

Another type of cyber-attack that targets web applications by exploiting vulnerabilities in their database handling.

In SQL, the UNION operator is used to combine the result sets of two or more SELECT queries into a single result set.

Once a vulnerability is identified, attackers inject specially crafted input that manipulates the original SQL query. They add a UNION statement to the end of the query to include data from another table they want to access.

- Out-of-Band SQL Injection

A type of SQL injection attack that doesn't rely on the direct communication between the attacker and the target database server. Instead, it leverages alternative channels or techniques to extract data from the database or interact with the attacker-controlled infrastructure. This approach can be useful when the target environment has restrictions on direct outbound communication from the database server, such as firewalls or security measures that block external connections.

Unlike traditional SQL Injection attacks that rely on the application's direct interaction with the database, Out-of-Band attacks use alternative channels to communicate. These channels can include DNS (Domain Name System) requests, HTTP requests, or even email requests.

One common Out-of-Band technique involves triggering DNS requests to domains controlled by the attacker. The injected SQL code manipulates the database to perform DNS lookups, which encode the extracted data into the domain name or subdomain. The attacker monitors their DNS server to capture the data.

Attackers can also craft SQL queries to make HTTP requests to URLs controlled by them. These requests can carry extracted data or instructions for the attacker's infrastructure to follow. The attacker then collects the data or observes the actions performed by their infrastructure.

- Compounded SQLi

Compounded SQL injection, also known as second-order SQL injection, occurs when an application initially sanitizes user input but stores it in a way that's later used in a vulnerable query. For instance, the input might be stored in a database and then retrieved later to be used in a query without proper validation. Attackers manipulate the stored input to exploit vulnerabilities when the input is used in a different context.

6. CONCLUSION

The examination of SQL injection vulnerabilities within the scope of this report underscores the critical importance of maintaining robust security measures when developing and maintaining web applications that interact with databases.

The potential risks associated with inadequate input validation and improper handling of user-generated data need to be mitigated by employing techniques such as parameterized queries, input validation, and least privilege access can greatly enhance the resilience of web applications against SQL injection threats.

By addressing SQL injection vulnerabilities in a systematic and proactive manner, organizations can enhance the security posture of their web applications, safeguarding sensitive data and ensuring the integrity of their databases against malicious activities that could damage individuals, companies, and governments.

REFERENCES

[1] **What is SQL injection?**. Found on CloudFlare:
https://www.cloudflare.com/learning/security/threats/sql-injection/

[2] **SQL Injection**. Found on php.net:
https://www.php.net/manual/en/security.database.sql-injection.php

[3] **How To: Protect From SQL Injection**. Found on Microsoft Learn:
https://learn.microsoft.com/en-us/previous-versions/msp-n-p/ff648339(v=pandp.10)

[4] **SQL Injection For Beginners** by Loi Liang Yang. Found on Youtube:
https://www.youtube.com/watch?v=cx6Xs3F_1Uc

[5] **SQL Injections are scary!! (hacking tutorial for beginners)** by NetworkChuck. Found on Youtube: https://www.youtube.com/watch?v=2OPVViV-GQk

[6] **How to make a site vulnerable to SQLi?.** Found on Information Security:
https://security.stackexchange.com/questions/36272/how-to-make-a-site-vulnerable-to-sqli

[7] **Protecting Against SQL Injection.** Found on Hacksplaining:
https://www.hacksplaining.com/prevention/sql-injection