SQL Injection Vulnerability and Attacks

This article is all about learning SQL Injection Vulnerabilities and the ways to remediate them. It will assist the developers in understanding how actually the attackers look for the loopholes which they can exploit and get access of the Web application or Database.

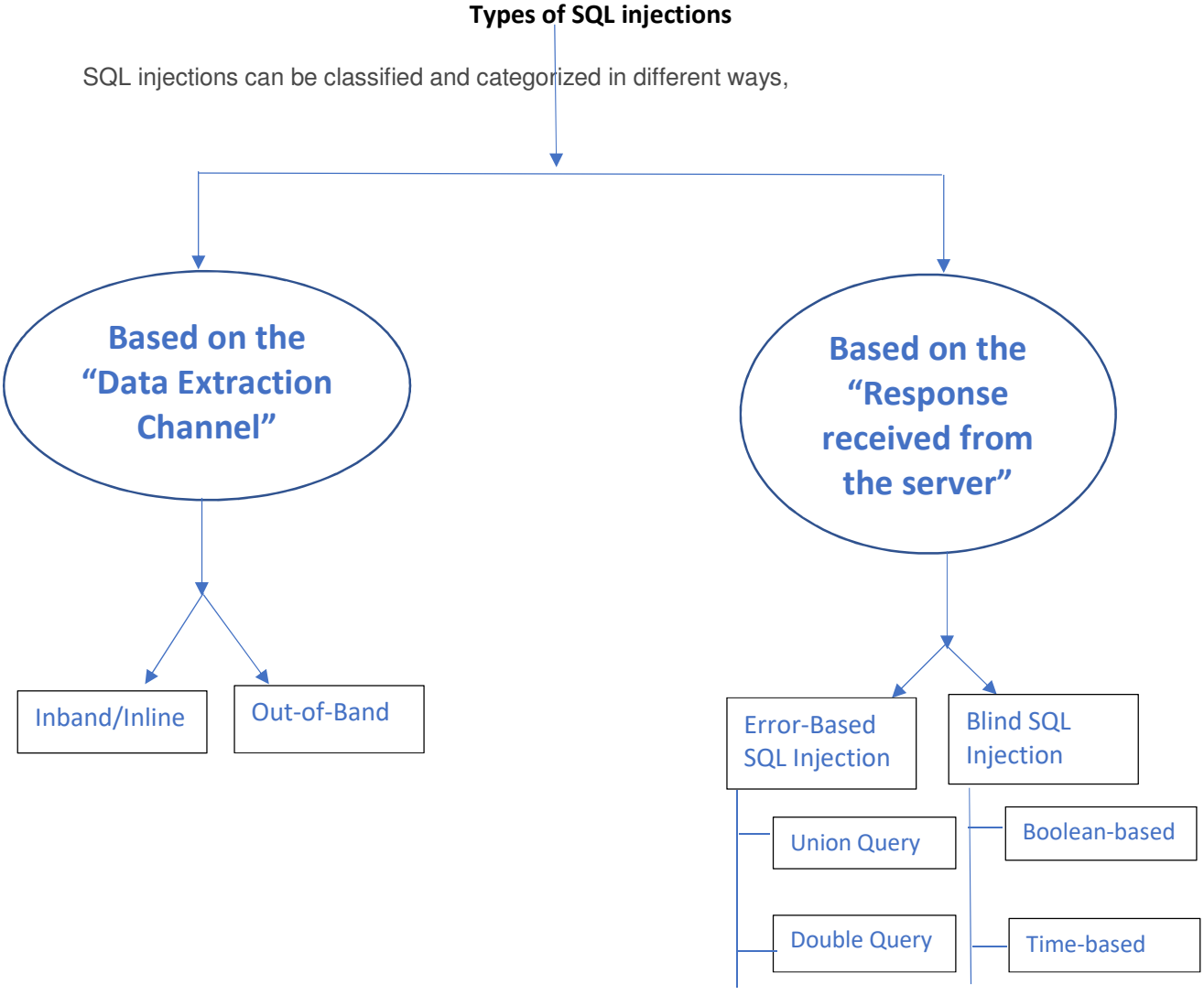
What is SQL injection?

An SQL injection is a kind of injection vulnerability in which the attacker tries to inject arbitrary pieces of malicious data into the input fields of an application, which, when processed by the application, causes that data to be executed as a piece of code by the back- end SQL server, thereby giving undesired results which the developer of the application did not anticipate. The backend server can be any SQL server (MySQL, MSSQL, ORACLE, POSTGRESS, etc.)

The ability of the attacker to execute code (SQL statements) through vulnerable input parameters empowers him to directly interact with the back-end SQL server, thereby leveraging almost a complete compromise of system in most cases.

Types of SQL injections

SQL injections can be classified and categorized in different ways,



A- Based on the “Data Extraction Channel”

1- Inband/Inline:

SQL injections that use the same communication channel as input to dump the information back are called **inband or inline SQL Injections**.

Example:

A query parameter, if injectable, leads to the dumping of info on the web page.

2- Out-of-Band:

Injections that use a secondary or different communication channel to dump the output of queries performed via the input channel are referred to as **out-of-band SQL injections**.

Example:

The injection is made to a web application and a secondary channel such as DNS queries is used to dump the data back to the attacker domain.

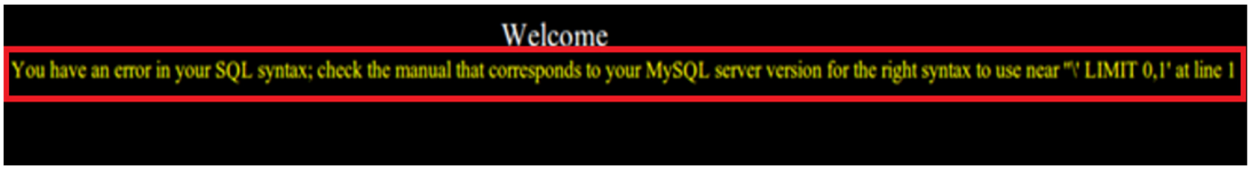
B- Based on the “Response received from the server”

1- Error-Based SQL Injection

Error-based SQL injections are primarily those in which the SQL server dumps some errors back to the user via the web application and this error aids in successful exploitation.

Example:

In the image below, the yellow line displays the error,



2- Blind SQL Injection

Blind SQL injections are those injections in which the backend database reacts to the input, but somehow the errors are concealed by the web application and not displayed to the end users OR the output is not dumped directly to the screen. Therefore, the name “blind” comes from the fact that the injector is blindly injected using some calculated assumptions and tries.

Why does SQL injection happen?

Generally, when an application is communicating with the backend database, it does so in the form of queries with the help of an underlying database driver. This driver is dependent on the application platform being used and the type of backend database, such as MYSQL, MSSQL, DB2, or ORACLE.

A generic login query would look something like this:

`SELECT Column1, Column2, Column3 FROM table_name WHERE username='$variable1' AND password='$variable2'; `

We can split this query into two parts, **code section** and the **data section**. The data section is the **$variable1** and **$variable2** and quotes are being used around the variable to define the string boundary.

Example:

Say at the login form, the username entered is **Admin** and password is p**@ssw0rd** which is collected by application and values of **$variable1** and $**variable2** are placed at their respective locations in the query, making it something like this.

`SELECT Column1, column2, Column3 FROM table_name WHERE username='Admin' AND password='p@ssw0rd'; `

Now the developer assumes that users of his application will always put a username and password combination to get a valid query for evaluation by database backend.

Concern: What if the user is malicious and enters some characters which have some special meaning in the query? (**example**, a single quote).

So, instead of putting Admin, he puts **Admin'**, thereby **causing an error thrown by the DB driver**.

Reason Behind-> Because of the unpaired quote entered by the user breaking the application logic. (refer the screenshot below)

FUZZING

Generally, the developer of the application assume that the user would input integer values. But an attacker tries to fuzz all input points of the application.

So, what exactly is Fuzz/fuzzing?

It is a process for supplying arbitrary dumb patterns as input with the objective to see application behaviour and try to find the discrepancies in the responses. The discrepancies indicate the possibility of vulnerability.

Below are some arbitrary inputs which we can add, append and use for purpose of detecting basic error based SQLi:

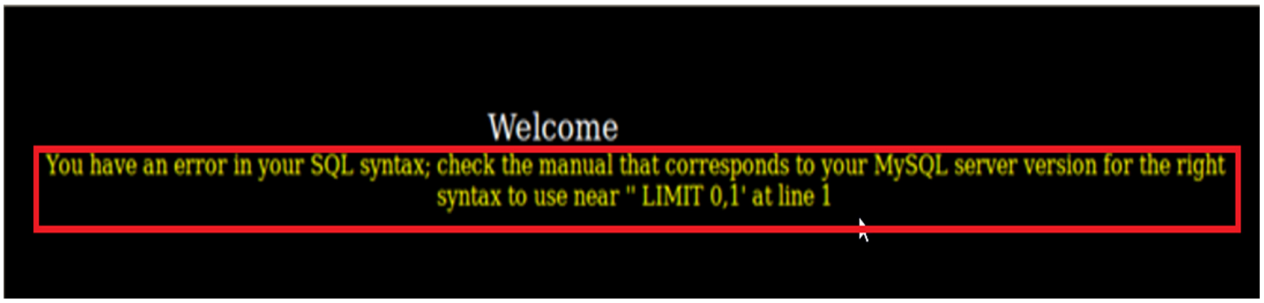
‘
”
\
;
%00
)
(
aaa

Example-1: Trying Single Quote (')

URL: http://pexa.com.au/?id=1'

Error: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near "1" LIMIT 0,1' at line 1

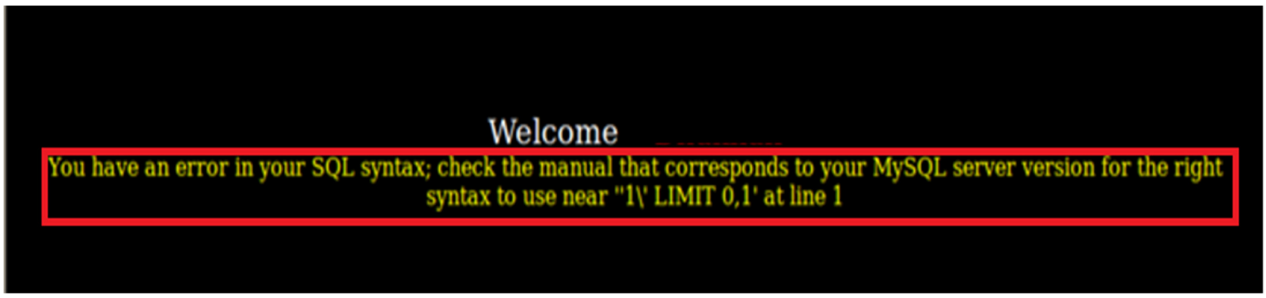
Refer the screenshot below,



Example-2: Trying Backslash (\)

URL: http://pexa.com.au/?id=1\

Refer the screenshot below,



Example-3: Commenting Query (-- OR #)

URL: http://pexa.com.au/?id=1'-- OR URL: http://pexa.com.au/?id=1#

So, the database query **SELECT * FROM TABLE_NAME WHERE ID='$ID' LIMIT 0,1** effectively becomes:

SELECT * FROM TABLE_NAME WHERE ID= **1' --** LIMIT 0,1 **AND,**

SELECT * FROM TABLE_NAME WHERE ID= **1' #** LIMIT 0,1

Refer the screenshot below,

Thus, in this way, the attacker could bypass the authentication and even can circumvent the authorised functionalities meant only for the restricted roles.

Remediation Techniques

To better understand the remediation techniques for SQL injection attacks, one should understand the types of the SQL queries being formed and used during the development phase.

Types of SQL Queries:

1- Dynamic SQL Queries
2- Parameterised SQL Queries

Dynamic SQL Queries:

It creates an SQL query with the user input all together. A dynamic query directly uses user's input into the query. It may or may not have implemented input escaping before using it in the SQL query.

A normal user authentication query should have been like this in a Dynamic SQL Query:

SELECT *username, password* FROM *users* where username="*hacker*" AND password="*India@123*" LIMIT 0, 1;

With this SQL query built and then executed, it would have worked flawlessly.

But an authentication query with SQL injection could have been like this:

SELECT *username, password* FROM *users* where username=" *hacker*" AND password=" *India@123" or 1=1 —*" LIMIT 0, 1;

This SQL query when executed would authenticate the user because the password field evaluates to TRUE because of user input containing ***" or 1=1 —***

User input is used to build dynamic SQL query, hence it becomes a part of the logic of SQL query, leading to SQL injection vulnerability. Due to this, using dynamic queries is not a good development strategy.

Parameterised SQL Queries:

It forces the user to implement the logic of SQL query first and then inserting user input into it. This forces the SQL query to be built before entering any user input in it.

Advantage of using SQL parameterized query is that it forces the data type of user input for a particular field in SQL query.

Example:

A SQL query expects a user to enter a number and then SQL is used to fetch a result depending upon that input. SQL parameterized query implementation forces the input data to be of integer type, and only then further processing will be done. Otherwise it will show an error or throw an exception

SQL parameterized query would be implemented like this in Java:

```
SELECT username, password FROM users where username=? AND password=? LIMIT 0, 1;

ps.setString(1,user_var);

ps.setString(2,pass_var);

ps.executeQuery();
```

NOTE: It is just a small code snippet and not the actual code. It is not the complete way of implementing parameterized query in Java.

This technique separates the SQL logic from the user input and defines the type of data expected in the fields.

This rule out the possibility of a SQL query getting executed with user input containing a SQL injection string such as **-1" or 1=1 —**

The main difference between a Dynamic Query and a SQL Parameterized Query is that in the former, the SQL logic is built along with the user input. But in the later, SQL logic is defined first and locked, then user input is passed as parameters along with its data type defined.

Another development strategy to be kept in mind is that SQL injection vulnerability not only exists while fetching data but also when executing any database query. SQL parameterized query should be implemented in in all ***SELECT, INSERT, UPDATE, DELETE*** queries.

Below is shown the SQL parameterized query implementation in Java:

SELECT Query:

```
String user = request.getParameter("user");

String pass = request.getParameter("pass");

PreparedStatement ps = (PreparedStatement) con.prepareStatement("SELECT username, password FROM users WHERE username=? AND password=? limit 0,1");

ps.setString(1,user);

ps.setString(2,pass);

ResultSet rs=ps.executeQuery();
```

INSERT Query:

```
String user = request.getParameter("user");

String pass = request.getParameter("pass");

String email = request.getParameter("email");

PreparedStatement ps = (PreparedStatement) con.prepareStatement("INSERT INTO users VALUES(?,?,?)");

ps.setString(1,user);

ps.setString(2,pass);

ps.setString(3,email);

int rs=ps.executeUpdate();
```

UPDATE Query:

```
String user = request.getParameter("user");

String email = request.getParameter("email");

PreparedStatement ps = (PreparedStatement) con.prepareStatement("UPDATE users SET email=? WHERE user=?");

   ps.setString(1,email);

   ps.setString(2,user);

   int rs=ps.executeUpdate();
```

DELETE Query:

```
String user = request.getParameter("user");

PreparedStatement ps = (PreparedStatement) con.prepareStatement("DELETE FROM users WHERE user=?");

   ps.setString(1,user);

   int rs=ps.executeUpdate();
```

Conclusion

Parameterized SQL Queries' key features over the Dynamic SQL Queries:

1- Protection from SQL injection
2- Improved efficiency
3- Higher reliability
4- Higher performance
5- Modular implementation

Key Notes

1- **Parameterized SQL Query has proven to be the best prevention techniques for SQL injection** and has shown how we can create **more efficient, reliable and secure applications.**
2- It is used for **better performance, high efficiency and prevention of *SQL* injection vulnerability.**

3- **Using dynamic SQL queries in an insecure manner is the root cause of SQL injection vulnerability.**
4- The **best choice for preventing SQL injection** has been found to be **"SQL Parameterized Queries"**.
5- **OWASP also recommends it as the first choice of prevention techniques for this vulnerability. (please refer the link below for further details)**

https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet#Defense_Option_1:_Prepared_Statements_.28Parameterized_Queries.29